

# 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         /// <summary>
20         /// <para>Presents the Range in readable format.</para>
21         /// <para>Представляет диапазон в удобном для чтения формате.</para>
22         /// </summary>
23         /// <returns><para>String representation of the Range.</para><para>Строковое представление диапазона.</para></returns>
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected virtual TElement GetZero() => default;
26
27         /// <summary>
28         /// <para>Presents the Range in readable format.</para>
29         /// <para>Представляет диапазон в удобном для чтения формате.</para>
30         /// </summary>
31         /// <returns><para>String representation of the Range.</para><para>Строковое представление диапазона.</para></returns>
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected virtual bool EqualToZero(TElement value) => EqualityComparer.Equals(value, GetZero());
34
35         /// <summary>
36         /// <para>Presents the Range in readable format.</para>
37         /// <para>Представляет диапазон в удобном для чтения формате.</para>
38         /// </summary>
39         /// <returns><para>String representation of the Range.</para><para>Строковое представление диапазона.</para></returns>
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         protected virtual bool AreEqual(TElement first, TElement second) => EqualityComparer.Equals(first, second);
42
43         /// <summary>
44         /// <para>Presents the Range in readable format.</para>
45         /// <para>Представляет диапазон в удобном для чтения формате.</para>
46         /// </summary>
47         /// <returns><para>String representation of the Range.</para><para>Строковое представление диапазона.</para></returns>
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected virtual bool GreaterThanZero(TElement value) => Comparer.Compare(value, GetZero()) > 0;
50
51         /// <summary>
52         /// <para>Presents the Range in readable format.</para>
53         /// <para>Представляет диапазон в удобном для чтения формате.</para>
54         /// </summary>
55         /// <returns><para>String representation of the Range.</para><para>Строковое представление диапазона.</para></returns>
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected virtual bool GreaterThan(TElement first, TElement second) => Comparer.Compare(first, second) > 0;
58
59         /// <summary>
60         /// <para>Presents the Range in readable format.</para>
61         /// <para>Представляет диапазон в удобном для чтения формате.</para>
62         /// </summary>
```

```

63     /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     protected virtual bool GreaterOrEqualThanZero(TElement value) => Comparer.Compare(value,
    ↪ Zero) >= 0;
66
67     /// <summary>
68     /// <para>Presents the Range in readable format.</para>
69     /// <para>Представляет диапазон в удобном для чтения формате.</para>
70     /// </summary>
71     /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
72     [MethodImpl(MethodImplOptions.AggressiveInlining)]
73     protected virtual bool GreaterOrEqualThan(TElement first, TElement second) =>
    ↪ Comparer.Compare(first, second) >= 0;
74
75     /// <summary>
76     /// <para>Presents the Range in readable format.</para>
77     /// <para>Представляет диапазон в удобном для чтения формате.</para>
78     /// </summary>
79     /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     protected virtual bool LessOrEqualThanZero(TElement value) => Comparer.Compare(value,
    ↪ Zero) <= 0;
82
83     /// <summary>
84     /// <para>Presents the Range in readable format.</para>
85     /// <para>Представляет диапазон в удобном для чтения формате.</para>
86     /// </summary>
87     /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     protected virtual bool LessOrEqualThan(TElement first, TElement second) =>
    ↪ Comparer.Compare(first, second) <= 0;
90
91     /// <summary>
92     /// <para>Presents the Range in readable format.</para>
93     /// <para>Представляет диапазон в удобном для чтения формате.</para>
94     /// </summary>
95     /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     protected virtual bool LessThanZero(TElement value) => Comparer.Compare(value, Zero) < 0;
98
99     /// <summary>
100    /// <para>Presents the Range in readable format.</para>
101    /// <para>Представляет диапазон в удобном для чтения формате.</para>
102    /// </summary>
103    /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
104    [MethodImpl(MethodImplOptions.AggressiveInlining)]
105    protected virtual bool LessThan(TElement first, TElement second) =>
    ↪ Comparer.Compare(first, second) < 0;
106
107    /// <summary>
108    /// <para>Presents the Range in readable format.</para>
109    /// <para>Представляет диапазон в удобном для чтения формате.</para>
110    /// </summary>
111    /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
112    [MethodImpl(MethodImplOptions.AggressiveInlining)]
113    protected virtual TElement Increment(TElement value) =>
    ↪ Arithmetic<TElement>.Increment(value);
114
115    /// <summary>
116    /// <para>Presents the Range in readable format.</para>
117    /// <para>Представляет диапазон в удобном для чтения формате.</para>
118    /// </summary>
119    /// <returns><para>String representation of the Range.</para><para>Строковое
    ↪ представление диапазона.</para></returns>
120    [MethodImpl(MethodImplOptions.AggressiveInlining)]
121    protected virtual TElement Decrement(TElement value) =>
    ↪ Arithmetic<TElement>.Decrement(value);
122
123    /// <summary>
124    /// <para>Presents the Range in readable format.</para>

```

```

125     /// <para>Представляет диапазон в удобном для чтения формате.</para>
126     /// </summary>
127     /// <returns><para>String representation of the Range.</para><para>Строковое
    ↳ представление диапазона.</para></returns>
128     [MethodImpl(MethodImplOptions.AggressiveInlining)]
129     protected virtual TElement Add(TElement first, TElement second) =>
    ↳ Arithmetic<TElement>.Add(first, second);
130
131     /// <summary>
132     /// <para>Presents the Range in readable format.</para>
133     /// <para>Представляет диапазон в удобном для чтения формате.</para>
134     /// </summary>
135     /// <returns><para>String representation of the Range.</para><para>Строковое
    ↳ представление диапазона.</para></returns>
136     [MethodImpl(MethodImplOptions.AggressiveInlining)]
137     protected virtual TElement Subtract(TElement first, TElement second) =>
    ↳ Arithmetic<TElement>.Subtract(first, second);
138
139     /// <summary>
140     /// <para>Returns minimum value of the range.</para>
141     /// <para>Возвращает минимальное значение диапазона.</para>
142     /// </summary>
143     protected readonly TElement Zero;
144
145     /// <summary>
146     /// <para>Returns minimum value of the range.</para>
147     /// <para>Возвращает минимальное значение диапазона.</para>
148     /// </summary>
149     protected readonly TElement One;
150
151     /// <summary>
152     /// <para>Returns minimum value of the range.</para>
153     /// <para>Возвращает минимальное значение диапазона.</para>
154     /// </summary>
155     protected readonly TElement Two;
156
157     /// <summary>
158     /// <para>Returns minimum value of the range.</para>
159     /// <para>Возвращает минимальное значение диапазона.</para>
160     /// </summary>
161     protected readonly EqualityComparer<TElement> EqualityComparer;
162
163     /// <summary>
164     /// <para>Returns minimum value of the range.</para>
165     /// <para>Возвращает минимальное значение диапазона.</para>
166     /// </summary>
167     protected readonly Comparer<TElement> Comparer;
168
169     /// <summary>
170     /// <para>Presents the Range in readable format.</para>
171     /// <para>Представляет диапазон в удобном для чтения формате.</para>
172     /// </summary>
173     /// <returns><para>String representation of the Range.</para><para>Строковое
    ↳ представление диапазона.</para></returns>
174     protected GenericCollectionMethodsBase()
175     {
176         EqualityComparer = EqualityComparer<TElement>.Default;
177         Comparer = Comparer<TElement>.Default;
178         Zero = GetZero(); //-V3068
179         One = Increment(Zero); //-V3068
180         Two = Increment(One); //-V3068
181     }
182 }
183 }

```

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

```

```

39     }
40
41     public void AttachAsFirst(TElement element)
42     {
43         var first = GetFirst();
44         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
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

```

```

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> :
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              SetPrevious(newElement, baseElement);
26              SetNext(newElement, baseElementNext);
27              if (AreEqual(baseElement, GetLast(headElement)))
28              {
29                  SetLast(headElement, newElement);
30              }
31              SetPrevious(baseElementNext, newElement);
32              SetNext(baseElement, newElement);
33              IncrementSize(headElement);
34          }
35
36          public void AttachAsFirst(TElement headElement, TElement element)
37          {
38              var first = GetFirst(headElement);
39              if (EqualToZero(first))
40              {
41                  SetFirst(headElement, element);
42                  SetLast(headElement, element);
43                  SetPrevious(element, element);
44                  SetNext(element, element);
45                  IncrementSize(headElement);
46              }
47              else
48              {
49                  AttachBefore(headElement, first, element);
50              }
51          }
52
53          public void AttachAsLast(TElement headElement, TElement element)
54          {
55              var last = GetLast(headElement);
56              if (EqualToZero(last))
57              {
58                  AttachAsFirst(headElement, element);
59              }
60              else
61              {
62                  AttachAfter(headElement, last, element);
63              }
64          }
65
66          public void Detach(TElement headElement, TElement element)
67          {
68              var elementPrevious = GetPrevious(element);
69              var elementNext = GetNext(element);
70              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> :
8          ↳ DoublyLinkedListMethodsBase<TElement>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         protected abstract TElement GetFirst(TElement headElement);
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected abstract TElement GetLast(TElement headElement);
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected abstract TElement GetSize(TElement headElement);
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected abstract void SetFirst(TElement headElement, TElement element);
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected abstract void SetLast(TElement headElement, TElement element);
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         protected abstract void SetSize(TElement headElement, TElement size);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected void IncrementSize(TElement headElement) => SetSize(headElement,
30             ↳ Increment(GetSize(headElement)));
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected void DecrementSize(TElement headElement) => SetSize(headElement,
34             ↳ Decrement(GetSize(headElement)));
35     }
36 }

```

### 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         }
18     }
19 }

```



```

15     }
16     else
17     {
18         SetNext(baseElementPrevious, newElement);
19     }
20     SetPrevious(baseElement, newElement);
21     IncrementSize(headElement);
22 }
23
24 public void AttachAfter(TElement headElement, TElement baseElement, TElement newElement)
25 {
26     var baseElementNext = GetNext(baseElement);
27     SetPrevious(newElement, baseElement);
28     SetNext(newElement, baseElementNext);
29     if (EqualToZero(baseElementNext))
30     {
31         SetLast(headElement, newElement);
32     }
33     else
34     {
35         SetPrevious(baseElementNext, newElement);
36     }
37     SetNext(baseElement, newElement);
38     IncrementSize(headElement);
39 }
40
41 public void AttachAsFirst(TElement headElement, TElement element)
42 {
43     var first = GetFirst(headElement);
44     if (EqualToZero(first))
45     {
46         SetFirst(headElement, element);
47         SetLast(headElement, element);
48         SetPrevious(element, Zero);
49         SetNext(element, Zero);
50         IncrementSize(headElement);
51     }
52     else
53     {
54         AttachBefore(headElement, first, element);
55     }
56 }
57
58 public void AttachAsLast(TElement headElement, TElement element)
59 {
60     var last = GetLast(headElement);
61     if (EqualToZero(last))
62     {
63         AttachAsFirst(headElement, element);
64     }
65     else
66     {
67         AttachAfter(headElement, last, element);
68     }
69 }
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))
28                          {
29                              RightRotate(ref root);
30                          }
31                          else
32                          {
33                              IncrementSize(root);
34                              root = ref left;
35                          }
36                      }
37                      else // node.Key greater than left.Key
38                      {
39                          var leftRightSize = GetSizeOrZero(GetRight(left));
40                          if (GreaterThan(Increment(leftRightSize), rightSize))
41                          {
42                              if (EqualToZero(leftRightSize) && EqualToZero(rightSize))
43                              {
44                                  SetLeft(node, left);
45                                  SetRight(node, root);
46                                  SetSize(node, Add(leftSize, Two)); // Two (2) - node the size of
47                                  ↳ root and a node itself
48                                  SetLeft(root, Zero);
49                                  SetSize(root, One);
50                                  root = node;
51                                  return;
52                              }
53                              LeftRotate(ref left);
54                              RightRotate(ref root);
55                          }
56                          else
57                          {
58                              IncrementSize(root);
59                              root = ref left;
60                          }
61                      }
62                  }
63                  else // node.Key greater than root.Key
64                  {
65                      if (EqualToZero(right))
66                      {
67                          IncrementSize(root);
68                          SetSize(node, One);
69                          right = node;
70                          return;
71                      }
72                      if (FirstIsToTheRightOfSecond(node, right)) // node.Key greater than
73                      ↳ right.Key

```

```

71     {
72         if (GreaterThan(Increment(rightSize), leftSize))
73         {
74             LeftRotate(ref root);
75         }
76         else
77         {
78             IncrementSize(root);
79             root = ref right;
80         }
81     }
82     else // node.Key less than right.Key
83     {
84         var rightLeftSize = GetSizeOrZero(GetLeft(right));
85         if (GreaterThan(Increment(rightLeftSize), leftSize))
86         {
87             if (EqualToZero(rightLeftSize) && EqualToZero(leftSize))
88             {
89                 SetLeft(node, root);
90                 SetRight(node, right);
91                 SetSize(node, Add(rightSize, Two)); // Two (2) - node the size
92                 ↪ of root and a node itself
93                 SetRight(root, Zero);
94                 SetSize(root, One);
95                 root = node;
96                 return;
97             }
98             RightRotate(ref right);
99             LeftRotate(ref root);
100         }
101         else
102         {
103             IncrementSize(root);
104             root = ref right;
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     }
158 }

```

```

145         LeftRotate(ref left);
146         RightRotate(ref root);
147     }
148     else
149     {
150         DecrementSize(root);
151         root = ref right;
152     }
153 }
154 else // key equals to root.Key
155 {
156     if (GreaterThanZero(leftSize) && GreaterThanZero(rightSize))
157     {
158         TElement replacement;
159         if (GreaterThan(leftSize, rightSize))
160         {
161             replacement = GetRighttest(left);
162             DetachCore(ref left, replacement);
163         }
164         else
165         {
166             replacement = GetLefttest(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

```

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     }
33 }

```

```

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),
67                 ↵ GetSize(node))));
68         }
69         else
70         {
71             SetSize(lefttestNode, Increment(GetSize(nodeToDetachLeft)));
72         }
73         replacementNode = lefttestNode;
74     }
75     else if (!EqualToZero(nodeToDetachLeft))
76     {
77         replacementNode = nodeToDetachLeft;
78     }
79     else if (!EqualToZero(node))
80     {
81         replacementNode = node;
82     }
83     if (AreEqual(root, nodeToDetach))
84     {
85         root = replacementNode;
86     }
87     else if (AreEqual(GetLeft(parent), nodeToDetach))
88     {
89         SetLeft(parent, replacementNode);
90     }
91     else if (AreEqual(GetRight(parent), nodeToDetach))
92     {
93         SetRight(parent, replacementNode);
94     }
95     ClearNode(nodeToDetach);
96 }
97
98 private void LeftMaintain(ref TElement root)
99 {
100     if (!EqualToZero(root))
101     {
102         var rootLeftNode = GetLeft(root);
103         if (!EqualToZero(rootLeftNode))
104         {
105             var rootRightNode = GetRight(root);
106             var rootRightNodeSize = GetSize(rootRightNode);
107             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 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) ||
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 GetRighttest(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 GetLefttest(TElement current)
41         {
42             var currentLeft = GetLeftOrDefault(current);
43             while (!EqualToZero(currentLeft))
44             {
45                 current = currentLeft;
46                 currentLeft = GetLeftOrDefault(current);
47             }
48             return current;
49         }
50
51         public override bool Contains(TElement node, TElement root)
52         {
53             while (!EqualToZero(root))
54             {
55                 {
56                     if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
57                     {
58                         root = GetLeftOrDefault(root);
59                     }
60                     else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
61                     {
62                         root = GetRightOrDefault(root);
63                     }
64                     else // node.Key == root.Key
65                     {
66                         return true;
67                     }
68                 }
69             }
70             return false;
71         }
72
73         protected override void PrintNode(TElement node, StringBuilder sb, int level)
74         {
75             base.PrintNode(node, sb, level);
76             sb.Append(' ');
77             sb.Append(GetLeftIsChild(node) ? 'l' : 'L');
78             sb.Append(GetRightIsChild(node) ? 'r' : 'R');
79             sb.Append(' ');
80             sb.Append(GetBalance(node));
81         }
82
83         [MethodImpl(MethodImplOptions.AggressiveInlining)]
84         protected void IncrementBalance(TElement node) => SetBalance(node,
85     ↪ (sbyte)(GetBalance(node) + 1));
86
87         [MethodImpl(MethodImplOptions.AggressiveInlining)]
88         protected void DecrementBalance(TElement node) => SetBalance(node,
89     ↪ (sbyte)(GetBalance(node) - 1));
90     }
91 }

```

```

83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 protected override TElement GetLeftOrDefault(TElement node) => GetLeftIsChild(node) ?
85     ↪ GetLeft(node) : default;
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 protected override TElement GetRightOrDefault(TElement node) => GetRightIsChild(node) ?
89     ↪ GetRight(node) : default;
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected abstract bool GetLeftIsChild(TElement node);
93
94 [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 protected abstract void SetLeftIsChild(TElement node, bool value);
96
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 protected abstract bool GetRightIsChild(TElement node);
99
100 [MethodImpl(MethodImplOptions.AggressiveInlining)]
101 protected abstract void SetRightIsChild(TElement node, bool value);
102
103 [MethodImpl(MethodImplOptions.AggressiveInlining)]
104 protected abstract sbyte GetBalance(TElement node);
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 protected abstract void SetBalance(TElement node, sbyte value);
108
109 protected override void AttachCore(ref TElement root, TElement node)
110 {
111     unchecked
112     {
113         // TODO: Check what is faster to use simple array or array from array pool
114         // TODO: Try to use stackalloc as an optimization (requires code generation,
115         ↪ because of generics)
116
117 #if USEARRAYPOOL
118     var path = ArrayPool.Allocate<TElement>(MaxPath);
119     var pathPosition = 0;
120     path[pathPosition++] = default;
121 #else
122     var path = new TElement[_maxPath];
123     var pathPosition = 1;
124 #endif
125
126     var currentNode = root;
127     while (true)
128     {
129         if (FirstIsToTheLeftOfSecond(node, currentNode))
130         {
131             if (GetLeftIsChild(currentNode))
132             {
133                 IncrementSize(currentNode);
134                 path[pathPosition++] = currentNode;
135                 currentNode = GetLeft(currentNode);
136             }
137             else
138             {
139                 // Threads
140                 SetLeft(node, GetLeft(currentNode));
141                 SetRight(node, currentNode);
142                 SetLeft(currentNode, node);
143                 SetLeftIsChild(currentNode, true);
144                 DecrementBalance(currentNode);
145                 SetSize(node, One);
146                 FixSize(currentNode); // Should be incremented already
147                 break;
148             }
149         }
150         else if (FirstIsToTheRightOfSecond(node, currentNode))
151         {
152             if (GetRightIsChild(currentNode))
153             {
154                 IncrementSize(currentNode);
155                 path[pathPosition++] = currentNode;
156                 currentNode = GetRight(currentNode);
157             }
158             else
159             {
160                 // Threads
161                 SetRight(node, GetRight(currentNode));
162                 SetLeft(node, currentNode);

```



```

159         SetRight(currentNode, node);
160         SetRightIsChild(currentNode, true);
161         IncrementBalance(currentNode);
162         SetSize(node, One);
163         FixSize(currentNode); // Should be incremented already
164         break;
165     }
166 }
167 else
168 {
169     throw new InvalidOperationException("Node with the same key already
170     ↳ attached to a tree.");
171 }
172 // Restore balance. This is the goodness of a non-recursive
173 // implementation, when we are done with balancing we 'break'
174 // the loop and we are done.
175 while (true)
176 {
177     var parent = path[--pathPosition];
178     var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
179     ↳ GetLeft(parent));
180     var currentNodeBalance = GetBalance(currentNode);
181     if (currentNodeBalance < -1 || currentNodeBalance > 1)
182     {
183         currentNode = Balance(currentNode);
184         if (AreEqual(parent, default))
185         {
186             root = currentNode;
187         }
188         else if (isLeftNode)
189         {
190             SetLeft(parent, currentNode);
191             FixSize(parent);
192         }
193         else
194         {
195             SetRight(parent, currentNode);
196             FixSize(parent);
197         }
198     }
199     currentNodeBalance = GetBalance(currentNode);
200     if (currentNodeBalance == 0 || AreEqual(parent, default))
201     {
202         break;
203     }
204     if (isLeftNode)
205     {
206         DecrementBalance(parent);
207     }
208     else
209     {
210         IncrementBalance(parent);
211     }
212     currentNode = parent;
213 }
214 #if USEARRAYPOOL
215     ArrayPool.Free(path);
216 #endif
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         else
427         {
428             SetRightIsChild(parent, false);
429             SetRight(parent, GetRight(currentNode));
430             DecrementBalance(parent);
431         }
432     }
433     else // node has a right child
434     {
435         var successor = GetNext(currentNode);
436         SetLeft(successor, GetLeft(currentNode));
437         var right = GetRight(currentNode);
438         if (AreEqual(parent, default))
439         {
440             root = right;
441         }
442         else if (isLeftNode)
443         {
444             SetLeft(parent, right);
445             IncrementBalance(parent);
446         }
447         else
448         {
449             SetRight(parent, right);
450             DecrementBalance(parent);
451         }
452     }
453 }
454 else // node has a left child
455 {
456     if (!GetRightIsChild(currentNode))
457     {
458         var predecessor = GetPrevious(currentNode);
459         SetRight(predecessor, GetRight(currentNode));
460         var leftValue = GetLeft(currentNode);
461         if (AreEqual(parent, default))
462         {
463             root = leftValue;
464         }
465         else if (isLeftNode)
466         {
467             SetLeft(parent, leftValue);
468             IncrementBalance(parent);
469         }
470     }
471 }

```

```

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);

```

```

546         if (currentNodeBalance < -1 || currentNodeBalance > 1)
547         {
548             balanceNode = Balance(balanceNode);
549             if (AreEqual(balanceParent, default))
550             {
551                 root = balanceNode;
552             }
553             else if (isLeftNode)
554             {
555                 SetLeft(balanceParent, balanceNode);
556             }
557             else
558             {
559                 SetRight(balanceParent, balanceNode);
560             }
561         }
562         currentNodeBalance = GetBalance(balanceNode);
563         if (currentNodeBalance != 0 || AreEqual(balanceParent, default))
564         {
565             break;
566         }
567         if (isLeftNode)
568         {
569             IncrementBalance(balanceParent);
570         }
571         else
572         {
573             DecrementBalance(balanceParent);
574         }
575         balanceNode = balanceParent;
576     }
577 }
578 ClearNode(node);
579 #if USEARRAYPOOL
580     ArrayPool.Free(path);
581 #endif
582 }
583 }
584
585 [MethodImpl(MethodImplOptions.AggressiveInlining)]
586 protected override void ClearNode(TElement node)
587 {
588     SetLeft(node, Zero);
589     SetRight(node, Zero);
590     SetSize(node, Zero);
591     SetLeftIsChild(node, false);
592     SetRightIsChild(node, false);
593     SetBalance(node, 0);
594 }
595 }
596 }

```

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

```

1  // #define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
2
3  using System;
4  using System.Diagnostics;
5  using System.Runtime.CompilerServices;
6  using System.Text;
7  using Platform.Numbers;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Collections.Methods.Trees
12 {
13     public abstract class SizedBinaryTreeMethodsBase<TElement> :
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     }

```

```

27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     protected abstract TElement GetSize(TElement node);
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     protected abstract void SetLeft(TElement node, TElement left);
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected abstract void SetRight(TElement node, TElement right);
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected abstract void SetSize(TElement node, TElement size);
38
39     [MethodImpl(MethodImplOptions.AggressiveInlining)]
40     protected abstract bool FirstIsToTheLeftOfSecond(TElement first, TElement second);
41
42     [MethodImpl(MethodImplOptions.AggressiveInlining)]
43     protected abstract bool FirstIsToTheRightOfSecond(TElement first, TElement second);
44
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     protected virtual TElement GetLeftOrDefault(TElement node) => AreEqual(node, default) ?
47         ↪ default : GetLeft(node);
48
49     [MethodImpl(MethodImplOptions.AggressiveInlining)]
50     protected virtual TElement GetRightOrDefault(TElement node) => AreEqual(node, default) ?
51         ↪ default : GetRight(node);
52
53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     protected void IncrementSize(TElement node) => SetSize(node, Increment(GetSize(node)));
55
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     protected void DecrementSize(TElement node) => SetSize(node, Decrement(GetSize(node)));
58
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     protected TElement GetLeftSize(TElement node) => GetSizeOrZero(GetLeftOrDefault(node));
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected TElement GetRightSize(TElement node) => GetSizeOrZero(GetRightOrDefault(node));
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected TElement GetSizeOrZero(TElement node) => EqualToZero(node) ? Zero :
67         ↪ GetSize(node);
68
69     [MethodImpl(MethodImplOptions.AggressiveInlining)]
70     protected void FixSize(TElement node) => SetSize(node, Increment(Add(GetLeftSize(node),
71         ↪ GetRightSize(node))));
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     protected void LeftRotate(ref TElement root) => root = LeftRotate(root);
75
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     protected TElement LeftRotate(TElement root)
78     {
79         var right = GetRight(root);
80         #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
81         if (EqualToZero(right))
82         {
83             throw new InvalidOperationException("Right is null.");
84         }
85         #endif
86         SetRight(root, GetLeft(right));
87         SetLeft(right, root);
88         SetSize(right, GetSize(root));
89         FixSize(root);
90         return right;
91     }
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     protected void RightRotate(ref TElement root) => root = RightRotate(root);
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     protected TElement RightRotate(TElement root)
98     {
99         var left = GetLeft(root);
100         #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
101         if (EqualToZero(left))
102         {
103             throw new InvalidOperationException("Left is null.");
104         }
105         #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 GetRightest(TElement current)
111 {
112     var currentRight = GetRight(current);
113     while (!EqualToZero(currentRight))
114     {
115         current = currentRight;
116         currentRight = GetRight(current);
117     }
118     return current;
119 }
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 protected virtual TElement GetLeftest(TElement current)
123 {
124     var currentLeft = GetLeft(current);
125     while (!EqualToZero(currentLeft))
126     {
127         current = currentLeft;
128         currentLeft = GetLeft(current);
129     }
130     return current;
131 }
132
133 [MethodImpl(MethodImplOptions.AggressiveInlining)]
134 protected virtual TElement GetNext(TElement node) => GetLeftest(GetRight(node));
135
136 [MethodImpl(MethodImplOptions.AggressiveInlining)]
137 protected virtual TElement GetPrevious(TElement node) => GetRightest(GetLeft(node));
138
139 [MethodImpl(MethodImplOptions.AggressiveInlining)]
140 public virtual bool Contains(TElement node, TElement root)
141 {
142     while (!EqualToZero(root))
143     {
144         if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
145         {
146             root = GetLeft(root);
147         }
148         else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
149         {
150             root = GetRight(root);
151         }
152         else // node.Key == root.Key
153         {
154             return true;
155         }
156     }
157     return false;
158 }
159
160 [MethodImpl(MethodImplOptions.AggressiveInlining)]
161 protected virtual void ClearNode(TElement node)
162 {
163     SetLeft(node, Zero);
164     SetRight(node, Zero);
165     SetSize(node, Zero);
166 }
167
168 [MethodImpl(MethodImplOptions.AggressiveInlining)]
169 public void Attach(ref TElement root, TElement node)
170 {
171 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
172     ValidateSizes(root);
173     Debug.WriteLine("--BeforeAttach--");
174     Debug.WriteLine(PrintNodes(root));
175     Debug.WriteLine("-----");
176     var sizeBefore = GetSize(root);
177 #endif
178     if (EqualToZero(root))
179     {
180         SetSize(node, One);

```



```

181         root = node;
182         return;
183     }
184     AttachCore(ref root, node);
185 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
186     Debug.WriteLine("--AfterAttach--");
187     Debug.WriteLine(PrintNodes(root));
188     Debug.WriteLine("-----");
189     ValidateSizes(root);
190     var sizeAfter = GetSize(root);
191     if (!AreEqual(Arithmetic.Increment(sizeBefore), sizeAfter))
192     {
193         throw new InvalidOperationException("Tree was broken after attach.");
194     }
195 #endif
196 }
197
198 protected abstract void AttachCore(ref TElement root, TElement node);
199
200 [MethodImpl(MethodImplOptions.AggressiveInlining)]
201 public void Detach(ref TElement root, TElement node)
202 {
203 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
204     ValidateSizes(root);
205     Debug.WriteLine("--BeforeDetach--");
206     Debug.WriteLine(PrintNodes(root));
207     Debug.WriteLine("-----");
208     var sizeBefore = GetSize(root);
209     if (EqualToZero(root))
210     {
211         throw new InvalidOperationException($"Элемент с {node} не содержится в
212             ↳ дереве.");
213     }
214 #endif
215     DetachCore(ref root, node);
216 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
217     Debug.WriteLine("--AfterDetach--");
218     Debug.WriteLine(PrintNodes(root));
219     Debug.WriteLine("-----");
220     ValidateSizes(root);
221     var sizeAfter = GetSize(root);
222     if (!AreEqual(Arithmetic.Decrement(sizeBefore), sizeAfter))
223     {
224         throw new InvalidOperationException("Tree was broken after detach.");
225     }
226 #endif
227 }
228
229 protected abstract void DetachCore(ref TElement root, TElement node);
230
231 public void FixSizes(TElement node)
232 {
233     if (AreEqual(node, default))
234     {
235         return;
236     }
237     FixSizes(GetLeft(node));
238     FixSizes(GetRight(node));
239     FixSize(node);
240 }
241
242 public void ValidateSizes(TElement node)
243 {
244     if (AreEqual(node, default))
245     {
246         return;
247     }
248     var size = GetSize(node);
249     var leftSize = GetLeftSize(node);
250     var rightSize = GetRightSize(node);
251     var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
252     if (!AreEqual(size, expectedSize))
253     {
254         throw new InvalidOperationException($"Size of {node} is not valid. Expected
255             ↳ size: {expectedSize}, actual size: {size}.");
256     }
257     ValidateSizes(GetLeft(node));
258     ValidateSizes(GetRight(node));

```

```

257     }
258
259     public void ValidateSize(TElement node)
260     {
261         var size = GetSize(node);
262         var leftSize = GetLeftSize(node);
263         var rightSize = GetRightSize(node);
264         var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
265         if (!AreEqual(size, expectedSize))
266         {
267             throw new InvalidOperationException($"Size of {node} is not valid. Expected
                ↳ size: {expectedSize}, actual size: {size}.");
268         }
269     }
270
271     public string PrintNodes(TElement node)
272     {
273         var sb = new StringBuilder();
274         PrintNodes(node, sb);
275         return sb.ToString();
276     }
277
278     [MethodImpl(MethodImplOptions.AggressiveInlining)]
279     public void PrintNodes(TElement node, StringBuilder sb) => PrintNodes(node, sb, 0);
280
281     public void PrintNodes(TElement node, StringBuilder sb, int level)
282     {
283         if (AreEqual(node, default))
284         {
285             return;
286         }
287         PrintNodes(GetLeft(node), sb, level + 1);
288         PrintNode(node, sb, level);
289         sb.AppendLine();
290         PrintNodes(GetRight(node), sb, level + 1);
291     }
292
293     public string PrintNode(TElement node)
294     {
295         var sb = new StringBuilder();
296         PrintNode(node, sb);
297         return sb.ToString();
298     }
299
300     [MethodImpl(MethodImplOptions.AggressiveInlining)]
301     protected void PrintNode(TElement node, StringBuilder sb) => PrintNode(node, sb, 0);
302
303     protected virtual void PrintNode(TElement node, StringBuilder sb, int level)
304     {
305         sb.Append('\t', level);
306         sb.Append(node);
307         PrintNodeValue(node, sb);
308         sb.Append(' ');
309         sb.Append('s');
310         sb.Append(GetSize(node));
311     }
312
313     protected abstract void PrintNodeValue(TElement node, StringBuilder sb);
314 }
315 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using Platform.Numbers;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Converters;
7
8  namespace Platform.Collections.Methods.Tests
9  {
10     public class RecursionlessSizeBalancedTree<TElement> :
        ↳ RecursionlessSizeBalancedTreeMethods<TElement>
11     {
12         private struct TreeElement
13         {
14             public TElement Size;
15             public TElement Left;
16             public TElement Right;

```

```

17     }
18
19     private readonly TreeElement[] _elements;
20     private TElement _allocated;
21
22     public TElement Root;
23
24     public TElement Count => GetSizeOrZero(Root);
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;
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
27             ↪ TreeElement[capacity], One);
28
29         public TElement Allocate()
30         {
31             var newNode = _allocated;
32             if (IsEmpty(newNode))
33             {
34                 _allocated = Arithmetic.Increment(_allocated);
35                 return newNode;
36             }
37             else
38             {
39                 throw new InvalidOperationException("Allocated tree element is not empty.");
40             }
41         }
42
43         public void Free(TElement node)
44         {
45             while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
46             {
47                 var lastNode = Arithmetic.Decrement(_allocated);
48                 if (EqualityComparer.Equals(lastNode, node))
49                 {
50                     _allocated = lastNode;
51                     node = Arithmetic.Decrement(node);
52                 }
53                 else
54                 {
55                     return;
56                 }
57             }
58         }
59
60         public bool IsEmpty(TElement node) =>
61             ↪ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
62
63         protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
64             ↪ Comparer.Compare(first, second) < 0;
65
66         protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
67             ↪ Comparer.Compare(first, second) > 0;
68
69         protected override ref TElement GetLeftReference(TElement node) => ref
70             ↪ GetElement(node).Left;
71
72         protected override TElement GetLeft(TElement node) => GetElement(node).Left;
73
74         protected override ref TElement GetRightReference(TElement node) => ref
75             ↪ GetElement(node).Right;
76
77         protected override TElement GetRight(TElement node) => GetElement(node).Right;
```

```

72     protected override TElement GetSize(TElement node) => GetElement(node).Size;
73
74     protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
75         ↪ sb.Append(node);
76
77     protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
78         ↪ left;
79
80     protected override void SetRight(TElement node, TElement right) =>
81         ↪ GetElement(node).Right = right;
82
83     protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
84         ↪ size;
85
86     private ref TreeElement GetElement(TElement node) => ref
87         ↪ _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
88 }
89 }

```

### 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
14         {
15             public TElement Size;
16             public TElement Left;
17             public TElement Right;
18             public sbyte Balance;
19             public bool LeftIsChild;
20             public bool RightIsChild;
21         }
22
23         private readonly TreeElement[] _elements;
24         private TElement _allocated;
25
26         public TElement Root;
27
28         public TElement Count => GetSizeOrZero(Root);
29
30         public SizedAndThreadedAVLBalancedTree(int capacity) => (_elements, _allocated) = (new
31             ↪ TreeElement[capacity], One);
32
33         public TElement Allocate()
34         {
35             var newNode = _allocated;
36             if (IsEmpty(newNode))
37             {
38                 _allocated = Arithmetic.Increment(_allocated);
39                 return newNode;
40             }
41             else
42             {
43                 throw new InvalidOperationException("Allocated tree element is not empty.");
44             }
45         }
46
47         public void Free(TElement node)
48         {
49             while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
50             {
51                 var lastNode = Arithmetic.Decrement(_allocated);
52                 if (EqualityComparer.Equals(lastNode, node))
53                 {
54                     _allocated = lastNode;
55                     node = Arithmetic.Decrement(node);
56                 }
57                 else
58                 {
59                     return;
60                 }
61             }
62         }
63     }
64 }

```

```

58     }
59 }
60 }
61
62 public bool IsEmpty(TElement node) =>
63     ↳ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
64
65 protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
66     ↳ Comparer.Compare(first, second) < 0;
67
68 protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
69     ↳ Comparer.Compare(first, second) > 0;
70
71 protected override sbyte GetBalance(TElement node) => GetElement(node).Balance;
72
73 protected override bool GetLeftIsChild(TElement node) => GetElement(node).LeftIsChild;
74
75 protected override ref TElement GetLeftReference(TElement node) => ref
76     ↳ GetElement(node).Left;
77
78 protected override TElement GetLeft(TElement node) => GetElement(node).Left;
79
80 protected override bool GetRightIsChild(TElement node) => GetElement(node).RightIsChild;
81
82 protected override ref TElement GetRightReference(TElement node) => ref
83     ↳ GetElement(node).Right;
84
85 protected override TElement GetRight(TElement node) => GetElement(node).Right;
86
87 protected override TElement GetSize(TElement node) => GetElement(node).Size;
88
89 protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
90     ↳ sb.Append(node);
91
92 protected override void SetBalance(TElement node, sbyte value) =>
93     ↳ GetElement(node).Balance = value;
94
95 protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
96     ↳ left;
97
98 protected override void SetLeftIsChild(TElement node, bool value) =>
99     ↳ GetElement(node).LeftIsChild = value;
100
101 protected override void SetRight(TElement node, TElement right) =>
102     ↳ GetElement(node).Right = right;
103
104 protected override void SetRightIsChild(TElement node, bool value) =>
105     ↳ GetElement(node).RightIsChild = value;
106
107 protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
108     ↳ size;
109
110 private ref TreeElement GetElement(TElement node) => ref
111     ↳ _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
112
113 }
114 }

```

## 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);

```

```

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
38 public static void TestMultipleRandomCreationsAndDeletions<TElement>(this
    ↪ SizedBinaryTreeMethodsBase<TElement> tree, ref TElement root, Func<TElement>
    ↪ treeCount, int maximumOperationsPerCycle)
39 {
40     var random = new System.Random(0);
41     var added = new HashSet<TElement>();
42     var currentCount = 0;
43     for (var N = 1; N < maximumOperationsPerCycle; N++)
44     {
45         for (var i = 0; i < N; i++)
46         {
47             var node = UncheckedConverter<int, TElement>.Default.Convert(random.Next(1,
        ↪ N));
48             if (added.Add(node))
49             {
50                 tree.Attach(ref root, node);
51                 currentCount++;
52                 Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
        ↪ int>.Default.Convert(treeCount()));
53             }
54         }
55         for (var i = 1; i <= N; i++)
56         {
57             TElement node = UncheckedConverter<int,
        ↪ TElement>.Default.Convert(random.Next(1, N));
58             if (tree.Contains(node, root))
59             {
60                 tree.Detach(ref root, node);
61                 currentCount--;
62                 Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
        ↪ int>.Default.Convert(treeCount()));
63                 added.Remove(node);
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);
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

- ./csharp/Platform.Collections.Methods.Tests/RecursionlessSizeBalancedTree.cs, 26
- ./csharp/Platform.Collections.Methods.Tests/SizeBalancedTree.cs, 28
- ./csharp/Platform.Collections.Methods.Tests/SizedAndThreadedAVLBalancedTree.cs, 29
- ./csharp/Platform.Collections.Methods.Tests/TestExtensions.cs, 30
- ./csharp/Platform.Collections.Methods.Tests/TreesTests.cs, 31
- ./csharp/Platform.Collections.Methods/GenericCollectionMethodsBase.cs, 1
- ./csharp/Platform.Collections.Methods/Lists/AbsoluteCircularDoublyLinkedListMethods.cs, 3
- ./csharp/Platform.Collections.Methods/Lists/AbsoluteDoublyLinkedListMethodsBase.cs, 5
- ./csharp/Platform.Collections.Methods/Lists/AbsoluteOpenDoublyLinkedListMethods.cs, 5
- ./csharp/Platform.Collections.Methods/Lists/DoublyLinkedListMethodsBase.cs, 6
- ./csharp/Platform.Collections.Methods/Lists/RelativeCircularDoublyLinkedListMethods.cs, 7
- ./csharp/Platform.Collections.Methods/Lists/RelativeDoublyLinkedListMethodsBase.cs, 8
- ./csharp/Platform.Collections.Methods/Lists/RelativeOpenDoublyLinkedListMethods.cs, 8
- ./csharp/Platform.Collections.Methods/Trees/RecursionlessSizeBalancedTreeMethods.cs, 10
- ./csharp/Platform.Collections.Methods/Trees/SizeBalancedTreeMethods.cs, 12
- ./csharp/Platform.Collections.Methods/Trees/SizedAndThreadedAVLBalancedTreeMethods.cs, 14
- ./csharp/Platform.Collections.Methods/Trees/SizedBinaryTreeMethodsBase.cs, 22