

LinksPlatform's Platform.Collections.Methods Class Library

./GenericCollectionMethodsBase.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Numbers;
5  using Platform.Unsafe;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Collections.Methods
10 {
11     public abstract class GenericCollectionMethodsBase<TElement>
12     {
13         private static readonly EqualityComparer<TElement> _equalityComparer =
14             ↪ EqualityComparer<TElement>.Default;
15         private static readonly Comparer<TElement> _comparer = Comparer<TElement>.Default;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected virtual TElement GetZero() => Integer<TElement>.Zero;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected virtual TElement GetOne() => Integer<TElement>.One;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected virtual TElement GetTwo() => Integer<TElement>.Two;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected virtual bool ValueEqualToZero(IntPtr pointer) =>
28             ↪ _equalityComparer.Equals(pointer.GetValue<TElement>(), GetZero());
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected virtual bool EqualToZero(TElement value) => _equalityComparer.Equals(value,
32             ↪ GetZero());
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected virtual bool IsEquals(TElement first, TElement second) =>
36             ↪ _equalityComparer.Equals(first, second);
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected virtual bool GreaterThanZero(TElement value) => _comparer.Compare(value,
40             ↪ GetZero()) > 0;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected virtual bool GreaterThan(TElement first, TElement second) =>
44             ↪ _comparer.Compare(first, second) > 0;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected virtual bool GreaterOrEqualThanZero(TElement value) =>
48             ↪ _comparer.Compare(value, GetZero()) >= 0;
49
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         protected virtual bool GreaterOrEqualThan(TElement first, TElement second) =>
52             ↪ _comparer.Compare(first, second) >= 0;
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected virtual bool LessOrEqualThanZero(TElement value) => _comparer.Compare(value,
56             ↪ GetZero()) <= 0;
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected virtual bool LessOrEqualThan(TElement first, TElement second) =>
60             ↪ _comparer.Compare(first, second) <= 0;
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         protected virtual bool LessThanZero(TElement value) => _comparer.Compare(value,
64             ↪ GetZero()) < 0;
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected virtual bool LessThan(TElement first, TElement second) =>
68             ↪ _comparer.Compare(first, second) < 0;
69
70         [MethodImpl(MethodImplOptions.AggressiveInlining)]
71         protected virtual TElement Increment(TElement value) =>
72             ↪ Arithmetic<TElement>.Increment(value);
73
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         protected virtual TElement Decrement(TElement value) =>
76             ↪ Arithmetic<TElement>.Decrement(value);
77     }
78 }

```

```

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 }

```

./Lists/CircularDoublyLinkedListMethods.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 CircularDoublyLinkedListMethods<TElement> :
        ↪ DoublyLinkedListMethodsBase<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 (IsEquals(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 (IsEquals(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 (IsEquals(elementNext, element))
70     {
71         SetFirst(GetZero());
72         SetLast(GetZero());
73     }
74     else
75     {
76         SetNext(elementPrevious, elementNext);
77         SetPrevious(elementNext, elementPrevious);
78         if (IsEquals(element, GetFirst()))
79         {
80             SetFirst(elementNext);
81         }
82         if (IsEquals(element, GetLast()))
83         {
84             SetLast(elementPrevious);
85         }
86     }
87     SetPrevious(element, GetZero());
88     SetNext(element, GetZero());
89     DecrementSize();
90 }
91 }
92 }

```

./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 GetFirst();
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected abstract TElement GetLast();
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected abstract TElement GetPrevious(TElement element);
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected abstract TElement GetNext(TElement element);
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected abstract TElement GetSize();
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected abstract void SetFirst(TElement element);
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected abstract void SetLast(TElement element);
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected abstract void SetPrevious(TElement element, TElement previous);
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected abstract void SetNext(TElement element, TElement next);
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected abstract void SetSize(TElement size);
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected void IncrementSize() => SetSize(Increment(GetSize()));
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected void DecrementSize() => SetSize(Decrement(GetSize()));
38     }
39 }

```

./Lists/OpenDoublyLinkedListMethods.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 OpenDoublyLinkedListMethods<TElement> :
6      ↪ DoublyLinkedListMethodsBase<TElement>
7      {
8          public void AttachBefore(TElement baseElement, TElement newElement)
9          {

```

```

9         var baseElementPrevious = GetPrevious(baseElement);
10        SetPrevious(newElement, baseElementPrevious);
11        SetNext(newElement, baseElement);
12        if (EqualToZero(baseElementPrevious))
13        {
14            SetFirst(newElement);
15        }
16        else
17        {
18            SetNext(baseElementPrevious, newElement);
19        }
20        SetPrevious(baseElement, newElement);
21        IncrementSize();
22    }
23
24    public void AttachAfter(TElement baseElement, TElement newElement)
25    {
26        var baseElementNext = GetNext(baseElement);
27        SetPrevious(newElement, baseElement);
28        SetNext(newElement, baseElementNext);
29        if (EqualToZero(baseElementNext))
30        {
31            SetLast(newElement);
32        }
33        else
34        {
35            SetPrevious(baseElementNext, newElement);
36        }
37        SetNext(baseElement, newElement);
38        IncrementSize();
39    }
40
41    public void AttachAsFirst(TElement element)
42    {
43        var first = GetFirst();
44        if (EqualToZero(first))
45        {
46            SetFirst(element);
47            SetLast(element);
48            SetPrevious(element, GetZero());
49            SetNext(element, GetZero());
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, GetZero());
92         SetNext(element, GetZero());
93         DecrementSize();
94     }
95 }
96 }

```

./Trees/SizeBalancedTreeMethods2.cs

```

1  using System;
2  using Platform.Unsafe;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Methods.Trees
7  {
8      /// <summary>
9      /// Experimental implementation, don't use it yet.
10     /// </summary>
11     public abstract class SizeBalancedTreeMethods2<TElement> :
12         ↳ SizedBinaryTreeMethodsBase<TElement>
13     {
14         protected override void AttachCore(IntPtr root, TElement newNode)
15         {
16             if (ValueEqualToZero(root))
17             {
18                 root.SetValue(newNode);
19                 IncrementSize(root.GetValue<TElement>());
20             }
21             else
22             {
23                 IncrementSize(root.GetValue<TElement>());
24                 if (FirstIsToTheLeftOfSecond(newNode, root.GetValue<TElement>()))
25                 {
26                     AttachCore(GetLeftPointer(root.GetValue<TElement>()), newNode);
27                     LeftMaintain(root);
28                 }
29                 else
30                 {
31                     AttachCore(GetRightPointer(root.GetValue<TElement>()), newNode);
32                     RightMaintain(root);
33                 }
34             }
35         }
36
37         protected override void DetachCore(IntPtr root, TElement nodeToDetach)
38         {
39             if (ValueEqualToZero(root))
40             {
41                 return;
42             }
43             var currentNode = root;
44             var parent = IntPtr.Zero; /* Изначально зануление, так как родителя может и не быть
45             ↳ (Корень дерева). */
46             var replacementNode = GetZero();
47             while (!IsEquals(currentNode.GetValue<TElement>(), nodeToDetach))
48             {
49                 SetSize(currentNode.GetValue<TElement>(),
50                 ↳ Decrement(GetSize(currentNode.GetValue<TElement>())));
51                 if (FirstIsToTheLeftOfSecond(nodeToDetach, currentNode.GetValue<TElement>()))
52                 {
53                     parent = currentNode;
54                     currentNode = GetLeftPointer(currentNode.GetValue<TElement>());
55                 }
56                 else if (FirstIsToTheRightOfSecond(nodeToDetach,
57                 ↳ currentNode.GetValue<TElement>()))
58                 {
59                     parent = currentNode;
60                     currentNode = GetRightPointer(currentNode.GetValue<TElement>());
61                 }
62                 else
63                 {
64                     throw new InvalidOperationException("Duplicate link found in the tree.");
65                 }
66             }
67         }
68     }
69 }

```

```

63     if (!ValueEqualToZero(GetLeftPointer(nodeToDetach)) &&
64         ↪ !ValueEqualToZero(GetRightPointer(nodeToDetach)))
65     {
66         var minNode = GetRightValue(nodeToDetach);
67         while (!EqualToZero(GetLeftValue(minNode)))
68         {
69             minNode = GetLeftValue(minNode); /* Передвигаемся до минимума */
70         }
71         DetachCore(GetRightPointer(nodeToDetach), minNode);
72         SetLeft(minNode, GetLeftValue(nodeToDetach));
73         if (!ValueEqualToZero(GetRightPointer(nodeToDetach)))
74         {
75             SetRight(minNode, GetRightValue(nodeToDetach));
76             SetSize(minNode, Increment(Add(GetSize(GetLeftValue(nodeToDetach)),
77                 ↪ GetSize(GetRightValue(nodeToDetach)))));
78         }
79         else
80         {
81             SetSize(minNode, Increment(GetSize(GetLeftValue(nodeToDetach))));
82         }
83         replacementNode = minNode;
84     }
85     else if (!ValueEqualToZero(GetLeftPointer(nodeToDetach)))
86     {
87         replacementNode = GetLeftValue(nodeToDetach);
88     }
89     else if (!ValueEqualToZero(GetRightPointer(nodeToDetach)))
90     {
91         replacementNode = GetRightValue(nodeToDetach);
92     }
93     if (parent == IntPtr.Zero)
94     {
95         root.SetValue(replacementNode);
96     }
97     else if (IsEquals(GetLeftValue(parent.GetValue<TElement>()), nodeToDetach))
98     {
99         SetLeft(parent.GetValue<TElement>(), replacementNode);
100     }
101     else if (IsEquals(GetRightValue(parent.GetValue<TElement>()), nodeToDetach))
102     {
103         SetRight(parent.GetValue<TElement>(), replacementNode);
104     }
105     ClearNode(nodeToDetach);
106 }
107
108 private void LeftMaintain(IntPtr root)
109 {
110     if (!ValueEqualToZero(root))
111     {
112         var rootLeftNode = GetLeftPointer(root.GetValue<TElement>());
113         if (!ValueEqualToZero(rootLeftNode))
114         {
115             var rootRightNode = GetRightPointer(root.GetValue<TElement>());
116             var rootLeftNodeLeftNode = GetLeftPointer(rootLeftNode.GetValue<TElement>());
117             if (!ValueEqualToZero(rootLeftNodeLeftNode) &&
118                 (ValueEqualToZero(rootRightNode) ||
119                 ↪ GreaterThan(GetSize(rootLeftNodeLeftNode.GetValue<TElement>()),
120                 ↪ GetSize(rootRightNode.GetValue<TElement>()))))
121             {
122                 RightRotate(root);
123             }
124             else
125             {
126                 var rootLeftNodeRightNode =
127                 ↪ GetRightPointer(rootLeftNode.GetValue<TElement>());
128                 if (!ValueEqualToZero(rootLeftNodeRightNode) &&
129                     (ValueEqualToZero(rootRightNode) ||
130                     ↪ GreaterThan(GetSize(rootLeftNodeRightNode.GetValue<TElement>()),
131                     ↪ GetSize(rootRightNode.GetValue<TElement>()))))
132                 {
133                     LeftRotate(GetLeftPointer(root.GetValue<TElement>()));
134                     RightRotate(root);
135                 }
136             }
137             else
138             {
139                 return;
140             }
141         }
142     }
143 }

```

```

134         LeftMaintain(GetLeftPointer(root.GetValue<TElement>()));
135         RightMaintain(GetRightPointer(root.GetValue<TElement>()));
136         LeftMaintain(root);
137         RightMaintain(root);
138     }
139 }
140 }
141 }
142 private void RightMaintain(IntPtr root)
143 {
144     if (!ValueEqualToZero(root))
145     {
146         var rootRightNode = GetRightPointer(root.GetValue<TElement>());
147         if (!ValueEqualToZero(rootRightNode))
148         {
149             var rootLeftNode = GetLeftPointer(root.GetValue<TElement>());
150             var rootRightNodeRightNode =
151                 ↳ GetRightPointer(rootRightNode.GetValue<TElement>());
152             if (!ValueEqualToZero(rootRightNodeRightNode) &&
153                 (ValueEqualToZero(rootLeftNode) ||
154                 ↳ GreaterThan(GetSize(rootRightNodeRightNode.GetValue<TElement>()),
155                 ↳ GetSize(rootLeftNode.GetValue<TElement>()))))
156             {
157                 LeftRotate(root);
158             }
159             else
160             {
161                 var rootRightNodeLeftNode =
162                 ↳ GetLeftPointer(rootRightNode.GetValue<TElement>());
163                 if (!ValueEqualToZero(rootRightNodeLeftNode) &&
164                     (ValueEqualToZero(rootLeftNode) ||
165                     ↳ GreaterThan(GetSize(rootRightNodeLeftNode.GetValue<TElement>()),
166                     ↳ GetSize(rootLeftNode.GetValue<TElement>()))))
167                 {
168                     RightRotate(GetRightPointer(root.GetValue<TElement>()));
169                     LeftRotate(root);
170                 }
171                 else
172                 {
173                     return;
174                 }
175             }
176             LeftMaintain(GetLeftPointer(root.GetValue<TElement>()));
177             RightMaintain(GetRightPointer(root.GetValue<TElement>()));
178             LeftMaintain(root);
179             RightMaintain(root);
180         }
181     }
182 }
183 }
184 }
185 }
186 }
187 }
188 }

```

./Trees/SizeBalancedTreeMethods.cs

```

1 using System;
2 using Platform.Unsafe;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Methods.Trees
7 {
8     public abstract class SizeBalancedTreeMethods<TElement> :
9         ↳ SizedBinaryTreeMethodsBase<TElement>
10     {
11         protected override void AttachCore(IntPtr root, TElement node)
12         {
13             while (true)
14             {
15                 var left = GetLeftPointer(root.GetValue<TElement>());
16                 var leftSize = GetSizeOrZero(left.GetValue<TElement>());
17                 var right = GetRightPointer(root.GetValue<TElement>());
18                 var rightSize = GetSizeOrZero(right.GetValue<TElement>());
19                 if (FirstIsToTheLeftOfSecond(node, root.GetValue<TElement>())) // node.Key less
20                     ↳ than root.Key
21                 {
22                     if (EqualToZero(left.GetValue<TElement>()))
23                     {
24                         IncrementSize(root.GetValue<TElement>());
25                         SetSize(node, GetOne());
26                     }
27                 }
28             }
29         }
30     }
31 }

```

```

24         left.SetValue(node);
25         break;
26     }
27     if (FirstIsToTheRightOfSecond(node, left.GetValue<TElement>())) // node.Key
    ↪ greater than left.Key
28     {
29         var leftRight = GetRightValue(left.GetValue<TElement>());
30         var leftRightSize = GetSizeOrZero(leftRight);
31         if (GreaterThan(Increment(leftRightSize), rightSize))
32         {
33             if (EqualToZero(leftRightSize) && EqualToZero(rightSize))
34             {
35                 SetLeft(node, left.GetValue<TElement>());
36                 SetRight(node, root.GetValue<TElement>());
37                 SetSize(node, Add(GetSize(left.GetValue<TElement>()),
    ↪ GetTwo())); // Two (2) - размер ветки *root (right) и самого
    ↪ node
38                 SetLeft(root.GetValue<TElement>(), GetZero());
39                 SetSize(root.GetValue<TElement>(), GetOne());
40                 root.SetValue(node);
41                 break;
42             }
43             LeftRotate(left);
44             RightRotate(root);
45         }
46         else
47         {
48             IncrementSize(root.GetValue<TElement>());
49             root = left;
50         }
51     }
52     else // node.Key less than left.Key
53     {
54         var leftLeft = GetLeftValue(left.GetValue<TElement>());
55         var leftLeftSize = GetSizeOrZero(leftLeft);
56         if (GreaterThan(Increment(leftLeftSize), rightSize))
57         {
58             RightRotate(root);
59         }
60         else
61         {
62             IncrementSize(root.GetValue<TElement>());
63             root = left;
64         }
65     }
66 }
67 else // node.Key greater than root.Key
68 {
69     if (EqualToZero(right.GetValue<TElement>()))
70     {
71         IncrementSize(root.GetValue<TElement>());
72         SetSize(node, GetOne());
73         right.SetValue(node);
74         break;
75     }
76     if (FirstIsToTheRightOfSecond(node, right.GetValue<TElement>())) // node.Key
    ↪ greater than right.Key
77     {
78         var rightRight = GetRightValue(right.GetValue<TElement>());
79         var rightRightSize = GetSizeOrZero(rightRight);
80         if (GreaterThan(Increment(rightRightSize), leftSize))
81         {
82             LeftRotate(root);
83         }
84         else
85         {
86             IncrementSize(root.GetValue<TElement>());
87             root = right;
88         }
89     }
90     else // node.Key less than right.Key
91     {
92         var rightLeft = GetLeftValue(right.GetValue<TElement>());
93         var rightLeftSize = GetSizeOrZero(rightLeft);
94         if (GreaterThan(Increment(rightLeftSize), leftSize))
95         {
96             if (EqualToZero(rightLeftSize) && EqualToZero(leftSize))
97             {

```



```

98         SetLeft(node, root.GetValue<TElement>());
99         SetRight(node, right.GetValue<TElement>());
100        SetSize(node, Add(GetSize(right.GetValue<TElement>()),
    ↪ GetTwo())); // Two (2) - размер ветки *root (left) и самого
    ↪ node
101        SetRight(root.GetValue<TElement>(), GetZero());
102        SetSize(root.GetValue<TElement>(), GetOne());
103        root.SetValue(node);
104        break;
105    }
106    RightRotate(right);
107    LeftRotate(root);
108    }
109    else
110    {
111        IncrementSize(root.GetValue<TElement>());
112        root = right;
113    }
114    }
115    }
116    }
117    }
118
119    protected override void DetachCore(IntPtr root, TElement node)
120    {
121        while (true)
122        {
123            var left = GetLeftPointer(root.GetValue<TElement>());
124            var leftSize = GetSizeOrZero(left.GetValue<TElement>());
125            var right = GetRightPointer(root.GetValue<TElement>());
126            var rightSize = GetSizeOrZero(right.GetValue<TElement>());
127            if (FirstIsToTheLeftOfSecond(node, root.GetValue<TElement>())) // node.Key less
    ↪ than root.Key
128            {
129                EnsureNodeInTheTree(node, left);
130                var rightLeft = GetLeftValue(right.GetValue<TElement>());
131                var rightLeftSize = GetSizeOrZero(rightLeft);
132                var rightRight = GetRightValue(right.GetValue<TElement>());
133                var rightRightSize = GetSizeOrZero(rightRight);
134                if (GreaterThan(rightRightSize, Decrement(leftSize)))
135                {
136                    LeftRotate(root);
137                }
138                else if (GreaterThan(rightLeftSize, Decrement(leftSize)))
139                {
140                    RightRotate(right);
141                    LeftRotate(root);
142                }
143                else
144                {
145                    DecrementSize(root.GetValue<TElement>());
146                    root = left;
147                }
148            }
149            else if (FirstIsToTheRightOfSecond(node, root.GetValue<TElement>())) // node.Key
    ↪ greater than root.Key
150            {
151                EnsureNodeInTheTree(node, right);
152                var leftLeft = GetLeftValue(left.GetValue<TElement>());
153                var leftLeftSize = GetSizeOrZero(leftLeft);
154                var leftRight = GetRightValue(left.GetValue<TElement>());
155                var leftRightSize = GetSizeOrZero(leftRight);
156                if (GreaterThan(leftLeftSize, Decrement(rightSize)))
157                {
158                    RightRotate(root);
159                }
160                else if (GreaterThan(leftRightSize, Decrement(rightSize)))
161                {
162                    LeftRotate(left);
163                    RightRotate(root);
164                }
165                else
166                {
167                    DecrementSize(root.GetValue<TElement>());
168                    root = right;
169                }
170            }
171            else // key equals to root.Key

```

```

172     {
173         if (GreaterThanZero(leftSize) && GreaterThanZero(rightSize))
174         {
175             if (GreaterThan(leftSize, rightSize))
176             {
177                 var replacement = left.GetValue<TElement>();
178                 while (!EqualToZero(GetRightValue(replacement)))
179                 {
180                     replacement = GetRightValue(replacement);
181                 }
182                 DetachCore(left, replacement);
183                 SetLeft(replacement, left.GetValue<TElement>());
184                 SetRight(replacement, right.GetValue<TElement>());
185                 FixSize(replacement);
186                 root.SetValue(replacement);
187             }
188             else
189             {
190                 var replacement = right.GetValue<TElement>();
191                 while (!EqualToZero(GetLeftValue(replacement)))
192                 {
193                     replacement = GetLeftValue(replacement);
194                 }
195                 DetachCore(right, replacement);
196                 SetLeft(replacement, left.GetValue<TElement>());
197                 SetRight(replacement, right.GetValue<TElement>());
198                 FixSize(replacement);
199                 root.SetValue(replacement);
200             }
201         }
202         else if (GreaterThanZero(leftSize))
203         {
204             root.SetValue(left.GetValue<TElement>());
205         }
206         else if (GreaterThanZero(rightSize))
207         {
208             root.SetValue(right.GetValue<TElement>());
209         }
210         else
211         {
212             root.SetValue(GetZero());
213         }
214         ClearNode(node);
215         break;
216     }
217 }
218
219 private void EnsureNodeInTheTree(TElement node, IntPtr branch)
220 {
221     if (EqualToZero(branch.GetValue<TElement>()))
222     {
223         throw new InvalidOperationException($"Элемент {node} не содержится в дереве.");
224     }
225 }
226
227 }
228

```

./Trees/SizedAndThreadedAVLBalancedTreeMethods.cs

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using System.Text;
4  using Platform.Unsafe;
5  #if USEARRAYPOOL
6  using Platform.Collections;
7  #endif
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Collections.Methods.Trees
12 {
13     /// <summary>
14     /// Combination of Size, Height (AVL), and threads.
15     /// </summary>
16     /// <remarks>
17     /// Based on: <a href="https://github.com/programmatom/TreeLib/blob/master/TreeLib/TreeLib/G_
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>.

```

```

19  /// </remarks>
20  public abstract class SizedAndThreadedAVLBalancedTreeMethods<TElement> :
    ↳ SizedBinaryTreeMethodsBase<TElement>
21  {
22      // TODO: Link with size of TElement
23      private const int MaxPath = 92;
24
25      protected override void PrintNode(TElement node, StringBuilder sb, int level)
26      {
27          base.PrintNode(node, sb, level);
28          sb.Append(' ');
29          sb.Append(GetLeftIsChild(node) ? 'l' : 'L');
30          sb.Append(GetRightIsChild(node) ? 'r' : 'R');
31          sb.Append(' ');
32          sb.Append(GetBalance(node));
33      }
34
35      [MethodImpl(MethodImplOptions.AggressiveInlining)]
36      protected void IncrementBalance(TElement node) => SetBalance(node,
    ↳ (sbyte)(GetBalance(node) + 1));
37
38      [MethodImpl(MethodImplOptions.AggressiveInlining)]
39      protected void DecrementBalance(TElement node) => SetBalance(node,
    ↳ (sbyte)(GetBalance(node) - 1));
40
41      [MethodImpl(MethodImplOptions.AggressiveInlining)]
42      protected override TElement GetLeftOrDefault(TElement node) => GetLeftIsChild(node) ?
    ↳ base.GetLeftOrDefault(node) : default;
43
44      [MethodImpl(MethodImplOptions.AggressiveInlining)]
45      protected override TElement GetRightOrDefault(TElement node) => GetRightIsChild(node) ?
    ↳ base.GetRightOrDefault(node) : default;
46
47      protected abstract bool GetLeftIsChild(TElement node);
48      protected abstract void SetLeftIsChild(TElement node, bool value);
49      protected abstract bool GetRightIsChild(TElement node);
50      protected abstract void SetRightIsChild(TElement node, bool value);
51      protected abstract sbyte GetBalance(TElement node);
52      protected abstract void SetBalance(TElement node, sbyte value);
53
54      protected override void AttachCore(IntPtr root, TElement node)
55      {
56          unchecked
57          {
58              // TODO: Check what is faster to use simple array or array from array pool
59              // TODO: Try to use stackalloc as an optimization (requires code generation,
    ↳ because of generics)
60  #if USEARRAYPOOL
61              var path = ArrayPool.Allocate<TElement>(MaxPath);
62              var pathPosition = 0;
63              path[pathPosition++] = default;
64  #else
65              var path = new TElement[MaxPath];
66              var pathPosition = 1;
67  #endif
68              var currentNode = root.GetValue<TElement>();
69              while (true)
70              {
71                  if (FirstIsToTheLeftOfSecond(node, currentNode))
72                  {
73                      if (GetLeftIsChild(currentNode))
74                      {
75                          IncrementSize(currentNode);
76                          path[pathPosition++] = currentNode;
77                          currentNode = GetLeftValue(currentNode);
78                      }
79                      else
80                      {
81                          // Threads
82                          SetLeft(node, GetLeftValue(currentNode));
83                          SetRight(node, currentNode);
84                          SetLeft(currentNode, node);
85                          SetLeftIsChild(currentNode, true);
86                          DecrementBalance(currentNode);
87                          SetSize(node, GetOne());
88                          FixSize(currentNode); // Should be incremented already
89                          break;
90                      }
91                  }

```

```

92     else if (FirstIsToTheRightOfSecond(node, currentNode))
93     {
94         if (GetRightIsChild(currentNode))
95         {
96             IncrementSize(currentNode);
97             path[pathPosition++] = currentNode;
98             currentNode = GetRightValue(currentNode);
99         }
100        else
101        {
102            // Threads
103            SetRight(node, GetRightValue(currentNode));
104            SetLeft(node, currentNode);
105            SetRight(currentNode, node);
106            SetRightIsChild(currentNode, true);
107            IncrementBalance(currentNode);
108            SetSize(node, GetOne());
109            FixSize(currentNode); // Should be incremented already
110            break;
111        }
112    }
113    else
114    {
115        throw new InvalidOperationException("Node with the same key already
116        ↳ attached to a tree.");
117    }
118    // Restore balance. This is the goodness of a non-recursive
119    // implementation, when we are done with balancing we 'break'
120    // the loop and we are done.
121    while (true)
122    {
123        var parent = path[--pathPosition];
124        var isLeftNode = !IsEquals(parent, default) && IsEquals(currentNode,
125        ↳ GetLeftValue(parent));
126        var currentNodeBalance = GetBalance(currentNode);
127        if (currentNodeBalance < -1 || currentNodeBalance > 1)
128        {
129            currentNode = Balance(currentNode);
130            if (IsEquals(parent, default))
131            {
132                root.SetValue(currentNode);
133            }
134            else if (isLeftNode)
135            {
136                SetLeft(parent, currentNode);
137                FixSize(parent);
138            }
139            else
140            {
141                SetRight(parent, currentNode);
142                FixSize(parent);
143            }
144            currentNodeBalance = GetBalance(currentNode);
145            if (currentNodeBalance == 0 || IsEquals(parent, default))
146            {
147                break;
148            }
149            if (isLeftNode)
150            {
151                DecrementBalance(parent);
152            }
153            else
154            {
155                IncrementBalance(parent);
156            }
157            currentNode = parent;
158        }
159        #if USEARRAYPOOL
160        ArrayPool.Free(path);
161        #endif
162    }
163 }
164
165 private TElement Balance(TElement node)
166 {
167     unchecked

```

```

168     {
169         var rootBalance = GetBalance(node);
170         if (rootBalance < -1)
171         {
172             var left = GetLeftValue(node);
173             if (GetBalance(left) > 0)
174             {
175                 SetLeft(node, LeftRotateWithBalance(left));
176                 FixSize(node);
177             }
178             node = RightRotateWithBalance(node);
179         }
180         else if (rootBalance > 1)
181         {
182             var right = GetRightValue(node);
183             if (GetBalance(right) < 0)
184             {
185                 SetRight(node, RightRotateWithBalance(right));
186                 FixSize(node);
187             }
188             node = LeftRotateWithBalance(node);
189         }
190         return node;
191     }
192 }
193
194 protected TElement LeftRotateWithBalance(TElement node)
195 {
196     unchecked
197     {
198         var right = GetRightValue(node);
199         if (GetLeftIsChild(right))
200         {
201             SetRight(node, GetLeftValue(right));
202         }
203         else
204         {
205             SetRightIsChild(node, false);
206             SetLeftIsChild(right, true);
207         }
208         SetLeft(right, node);
209         // Fix size
210         SetSize(right, GetSize(node));
211         FixSize(node);
212         // Fix balance
213         var rootBalance = GetBalance(node);
214         var rightBalance = GetBalance(right);
215         if (rightBalance <= 0)
216         {
217             if (rootBalance >= 1)
218             {
219                 SetBalance(right, (sbyte)(rightBalance - 1));
220             }
221             else
222             {
223                 SetBalance(right, (sbyte)(rootBalance + rightBalance - 2));
224             }
225             SetBalance(node, (sbyte)(rootBalance - 1));
226         }
227         else
228         {
229             if (rootBalance <= rightBalance)
230             {
231                 SetBalance(right, (sbyte)(rootBalance - 2));
232             }
233             else
234             {
235                 SetBalance(right, (sbyte)(rightBalance - 1));
236             }
237             SetBalance(node, (sbyte)(rootBalance - rightBalance - 1));
238         }
239         return right;
240     }
241 }
242
243 protected TElement RightRotateWithBalance(TElement node)
244 {
245     unchecked

```

```

246     {
247         var left = GetLeftValue(node);
248         if (GetRightIsChild(left))
249         {
250             SetLeft(node, GetRightValue(left));
251         }
252         else
253         {
254             SetLeftIsChild(node, false);
255             SetRightIsChild(left, true);
256         }
257         SetRight(left, node);
258         // Fix size
259         SetSize(left, GetSize(node));
260         FixSize(node);
261         // Fix balance
262         var rootBalance = GetBalance(node);
263         var leftBalance = GetBalance(left);
264         if (leftBalance <= 0)
265         {
266             if (leftBalance > rootBalance)
267             {
268                 SetBalance(left, (sbyte)(leftBalance + 1));
269             }
270             else
271             {
272                 SetBalance(left, (sbyte)(rootBalance + 2));
273             }
274             SetBalance(node, (sbyte)(rootBalance - leftBalance + 1));
275         }
276         else
277         {
278             if (rootBalance <= -1)
279             {
280                 SetBalance(left, (sbyte)(leftBalance + 1));
281             }
282             else
283             {
284                 SetBalance(left, (sbyte)(rootBalance + leftBalance + 2));
285             }
286             SetBalance(node, (sbyte)(rootBalance + 1));
287         }
288         return left;
289     }
290 }
291
292 protected TElement GetNext(TElement node)
293 {
294     unchecked
295     {
296         var current = GetRightValue(node);
297         if (GetRightIsChild(node))
298         {
299             while (GetLeftIsChild(current))
300             {
301                 current = GetLeftValue(current);
302             }
303         }
304         return current;
305     }
306 }
307
308 protected TElement GetPrevious(TElement node)
309 {
310     unchecked
311     {
312         var current = GetLeftValue(node);
313         if (GetLeftIsChild(node))
314         {
315             while (GetRightIsChild(current))
316             {
317                 current = GetRightValue(current);
318             }
319         }
320         return current;
321     }
322 }
323
324 protected override void DetachCore(IntPtr root, TElement node)

```

```

325     {
326         unchecked
327         {
328             #if USEARRAYPOOL
329                 var path = ArrayPool.Allocate<TElement>(MaxPath);
330                 var pathPosition = 0;
331                 path[pathPosition++] = default;
332             #else
333                 var path = new TElement[MaxPath];
334                 var pathPosition = 1;
335             #endif
336             var currentNode = root.GetValue<TElement>();
337             while (true)
338             {
339                 if (FirstIsToTheLeftOfSecond(node, currentNode))
340                 {
341                     if (!GetLeftIsChild(currentNode))
342                     {
343                         throw new InvalidOperationException("Cannot find a node.");
344                     }
345                     DecrementSize(currentNode);
346                     path[pathPosition++] = currentNode;
347                     currentNode = GetLeftValue(currentNode);
348                 }
349                 else if (FirstIsToTheRightOfSecond(node, currentNode))
350                 {
351                     if (!GetRightIsChild(currentNode))
352                     {
353                         throw new InvalidOperationException("Cannot find a node.");
354                     }
355                     DecrementSize(currentNode);
356                     path[pathPosition++] = currentNode;
357                     currentNode = GetRightValue(currentNode);
358                 }
359                 else
360                 {
361                     break;
362                 }
363             }
364             var parent = path[--pathPosition];
365             balanceNode = parent;
366             var isLeftNode = !IsEquals(parent, default) && IsEquals(currentNode,
367                 ↪ GetLeftValue(parent));
368             if (!GetLeftIsChild(currentNode))
369             {
370                 if (!GetRightIsChild(currentNode)) // node has no children
371                 {
372                     if (IsEquals(parent, default))
373                     {
374                         root.SetValue(GetZero());
375                     }
376                     else if (isLeftNode)
377                     {
378                         SetLeftIsChild(parent, false);
379                         SetLeft(parent, GetLeftValue(currentNode));
380                         IncrementBalance(parent);
381                     }
382                     else
383                     {
384                         SetRightIsChild(parent, false);
385                         SetRight(parent, GetRightValue(currentNode));
386                         DecrementBalance(parent);
387                     }
388                 }
389                 else // node has a right child
390                 {
391                     var successor = GetNext(currentNode);
392                     SetLeft(successor, GetLeftValue(currentNode));
393                     var right = GetRightValue(currentNode);
394                     if (IsEquals(parent, default))
395                     {
396                         root.SetValue(right);
397                     }
398                     else if (isLeftNode)
399                     {
400                         SetLeft(parent, right);
401                         IncrementBalance(parent);
402                     }
403                 }
404             }
405         }
406     }

```

```

402         else
403         {
404             SetRight(parent, right);
405             DecrementBalance(parent);
406         }
407     }
408 }
409 else // node has a left child
410 {
411     if (!GetRightIsChild(currentNode))
412     {
413         var predecessor = GetPrevious(currentNode);
414         SetRight(predecessor, GetRightValue(currentNode));
415         var leftValue = GetLeftValue(currentNode);
416         if (IsEquals(parent, default))
417         {
418             root.SetValue(leftValue);
419         }
420         else if (isLeftNode)
421         {
422             SetLeft(parent, leftValue);
423             IncrementBalance(parent);
424         }
425         else
426         {
427             SetRight(parent, leftValue);
428             DecrementBalance(parent);
429         }
430     }
431     else // node has a both children (left and right)
432     {
433         var predecessor = GetLeftValue(currentNode);
434         var successor = GetRightValue(currentNode);
435         var successorParent = currentNode;
436         int previousPathPosition = ++pathPosition;
437         // find the immediately next node (and its parent)
438         while (GetLeftIsChild(successor))
439         {
440             path[++pathPosition] = successorParent = successor;
441             successor = GetLeftValue(successor);
442             if (!IsEquals(successorParent, currentNode))
443             {
444                 DecrementSize(successorParent);
445             }
446         }
447         path[previousPathPosition] = successor;
448         balanceNode = path[pathPosition];
449         // remove 'successor' from the tree
450         if (!IsEquals(successorParent, currentNode))
451         {
452             if (!GetRightIsChild(successor))
453             {
454                 SetLeftIsChild(successorParent, false);
455             }
456             else
457             {
458                 SetLeft(successorParent, GetRightValue(successor));
459             }
460             IncrementBalance(successorParent);
461             SetRightIsChild(successor, true);
462             SetRight(successor, GetRightValue(currentNode));
463         }
464         else
465         {
466             DecrementBalance(currentNode);
467         }
468         // set the predecessor's successor link to point to the right place
469         while (GetRightIsChild(predecessor))
470         {
471             predecessor = GetRightValue(predecessor);
472         }
473         SetRight(predecessor, successor);
474         // prepare 'successor' to replace 'node'
475         var left = GetLeftValue(currentNode);
476         SetLeftIsChild(successor, true);
477         SetLeft(successor, left);
478         SetBalance(successor, GetBalance(currentNode));
479         FixSize(successor);

```



```

480         if (IsEquals(parent, default))
481         {
482             root.SetValue(successor);
483         }
484         else if (isLeftNode)
485         {
486             SetLeft(parent, successor);
487         }
488         else
489         {
490             SetRight(parent, successor);
491         }
492     }
493 }
494 // restore balance
495 if (!IsEquals(balanceNode, default))
496 {
497     while (true)
498     {
499         var balanceParent = path[--pathPosition];
500         isLeftNode = !IsEquals(balanceParent, default) && IsEquals(balanceNode,
501             ↪ GetLeftValue(balanceParent));
502         var currentNodeBalance = GetBalance(balanceNode);
503         if (currentNodeBalance < -1 || currentNodeBalance > 1)
504         {
505             balanceNode = Balance(balanceNode);
506             if (IsEquals(balanceParent, default))
507             {
508                 root.SetValue(balanceNode);
509             }
510             else if (isLeftNode)
511             {
512                 SetLeft(balanceParent, balanceNode);
513             }
514             else
515             {
516                 SetRight(balanceParent, balanceNode);
517             }
518             currentNodeBalance = GetBalance(balanceNode);
519             if (currentNodeBalance != 0 || IsEquals(balanceParent, default))
520             {
521                 break;
522             }
523             if (isLeftNode)
524             {
525                 IncrementBalance(balanceParent);
526             }
527             else
528             {
529                 DecrementBalance(balanceParent);
530             }
531             balanceNode = balanceParent;
532         }
533     }
534     ClearNode(node);
535 #if USEARRAYPOOL
536     ArrayPool.Free(path);
537 #endif
538 }
539 }
540
541 [MethodImpl(MethodImplOptions.AggressiveInlining)]
542 protected override void ClearNode(TElement node)
543 {
544     SetLeft(node, GetZero());
545     SetRight(node, GetZero());
546     SetSize(node, GetZero());
547     SetLeftIsChild(node, false);
548     SetRightIsChild(node, false);
549     SetBalance(node, 0);
550 }
551 }
552 }

```

./Trees/SizedBinaryTreeMethodsBase.cs

```

1 using System;
2 using System.Runtime.CompilerServices;

```

```

3 using System.Text;
4 using Platform.Numbers;
5 using Platform.Unsafe;
6
7 // #define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Collections.Methods.Trees
11 {
12     public abstract class SizedBinaryTreeMethodsBase<TElement> :
13         ↳ GenericCollectionMethodsBase<TElement>
14     {
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected abstract IntPtr GetLeftPointer(TElement node);
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected abstract IntPtr GetRightPointer(TElement node);
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected abstract TElement GetLeftValue(TElement node);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected abstract TElement GetRightValue(TElement node);
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected abstract TElement GetSize(TElement node);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected abstract void SetLeft(TElement node, TElement left);
32
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) => GetLeftPointer(node) !=
47             ↳ IntPtr.Zero ? GetLeftValue(node) : default;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected virtual TElement GetRightOrDefault(TElement node) => GetRightPointer(node) !=
51             ↳ IntPtr.Zero ? GetRightValue(node) : default;
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) ? GetZero() :
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(IntPtr root) =>
75             ↳ root.SetValue(LeftRotate(root.GetValue<TElement>()));
76
77         [MethodImpl(MethodImplOptions.AggressiveInlining)]
78         protected TElement LeftRotate(TElement root)
79         {
80             var right = GetRightValue(root);
81             #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION

```

```

76         if (EqualToZero(right))
77         {
78             throw new Exception("Right is null.");
79         }
80 #endif
81         SetRight(root, GetLeftValue(right));
82         SetLeft(right, root);
83         SetSize(right, GetSize(root));
84         FixSize(root);
85         return right;
86     }
87
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     protected void RightRotate(IntPtr root) =>
90         ↪ root.SetValue(RightRotate(root.GetValue<TElement>()));
91
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     protected TElement RightRotate(TElement root)
94     {
95         var left = GetLeftValue(root);
96 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
97         if (EqualToZero(left))
98         {
99             throw new Exception("Left is null.");
100         }
101 #endif
102         SetLeft(root, GetRightValue(left));
103         SetRight(left, root);
104         SetSize(left, GetSize(root));
105         FixSize(root);
106         return left;
107     }
108
109     [MethodImpl(MethodImplOptions.AggressiveInlining)]
110     public bool Contains(TElement node, TElement root)
111     {
112         while (!EqualToZero(root))
113         {
114             if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
115             {
116                 root = GetLeftOrDefault(root);
117             }
118             else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
119             {
120                 root = GetRightOrDefault(root);
121             }
122             else // node.Key == root.Key
123             {
124                 return true;
125             }
126         }
127         return false;
128     }
129
130     [MethodImpl(MethodImplOptions.AggressiveInlining)]
131     protected virtual void ClearNode(TElement node)
132     {
133         SetLeft(node, GetZero());
134         SetRight(node, GetZero());
135         SetSize(node, GetZero());
136     }
137
138     public void Attach(IntPtr root, TElement node)
139     {
140 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
141         ValidateSizes(root);
142         Debug.WriteLine("--BeforeAttach--");
143         Debug.WriteLine(PrintNodes(root));
144         Debug.WriteLine("-----");
145         var sizeBefore = GetSize(root);
146 #endif
147         if (ValueEqualToZero(root))
148         {
149             SetSize(node, GetOne());
150             root.SetValue(node);
151             return;
152         }
153         AttachCore(root, node);
154 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION

```

```

154         Debug.WriteLine("--AfterAttach--");
155         Debug.WriteLine(PrintNodes(root));
156         Debug.WriteLine("-----");
157         ValidateSizes(root);
158         var sizeAfter = GetSize(root);
159         if (!IsEquals(MathHelpers.Increment(sizeBefore), sizeAfter))
160         {
161             throw new Exception("Tree was broken after attach.");
162         }
163     #endif
164 }
165
166     protected abstract void AttachCore(IntPtr root, TElement node);
167
168     public void Detach(IntPtr root, TElement node)
169     {
170     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
171         ValidateSizes(root);
172         Debug.WriteLine("--BeforeDetach--");
173         Debug.WriteLine(PrintNodes(root));
174         Debug.WriteLine("-----");
175         var sizeBefore = GetSize(root);
176         if (ValueEqualToZero(root))
177         {
178             throw new Exception($"Элемент с {node} не содержится в дереве.");
179         }
180     #endif
181         DetachCore(root, node);
182     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
183         Debug.WriteLine("--AfterDetach--");
184         Debug.WriteLine(PrintNodes(root));
185         Debug.WriteLine("-----");
186         ValidateSizes(root);
187         var sizeAfter = GetSize(root);
188         if (!IsEquals(MathHelpers.Decrement(sizeBefore), sizeAfter))
189         {
190             throw new Exception("Tree was broken after detach.");
191         }
192     #endif
193     }
194
195     protected abstract void DetachCore(IntPtr root, TElement node);
196
197     public TElement GetSize(IntPtr root) => root == IntPtr.Zero ? GetZero() :
198     ↪ GetSizeOrZero(root.GetValue<TElement>());
199
200     public void FixSizes(IntPtr root)
201     {
202         if (root != IntPtr.Zero)
203         {
204             FixSizes(root.GetValue<TElement>());
205         }
206     }
207
208     public void FixSizes(TElement node)
209     {
210         if (IsEquals(node, default))
211         {
212             return;
213         }
214         FixSizes(GetLeftOrDefault(node));
215         FixSizes(GetRightOrDefault(node));
216         FixSize(node);
217     }
218
219     public void ValidateSizes(IntPtr root)
220     {
221         if (root != IntPtr.Zero)
222         {
223             ValidateSizes(root.GetValue<TElement>());
224         }
225     }
226
227     public void ValidateSizes(TElement node)
228     {
229         if (IsEquals(node, default))
230         {
231             return;
232         }
233     }

```

```

232     var size = GetSize(node);
233     var leftSize = GetLeftSize(node);
234     var rightSize = GetRightSize(node);
235     var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
236     if (!IsEquals(size, expectedSize))
237     {
238         throw new InvalidOperationException($"Size of {node} is not valid. Expected
        ↳ size: {expectedSize}, actual size: {size}.");
239     }
240     ValidateSizes(GetLeftOrDefault(node));
241     ValidateSizes(GetRightOrDefault(node));
242 }
243
244 public void ValidateSize(TElement node)
245 {
246     var size = GetSize(node);
247     var leftSize = GetLeftSize(node);
248     var rightSize = GetRightSize(node);
249     var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
250     if (!IsEquals(size, expectedSize))
251     {
252         throw new InvalidOperationException($"Size of {node} is not valid. Expected
        ↳ size: {expectedSize}, actual size: {size}.");
253     }
254 }
255
256 public string PrintNodes(IntPtr root)
257 {
258     if (root != IntPtr.Zero)
259     {
260         var sb = new StringBuilder();
261         PrintNodes(root.GetValue<TElement>(), sb);
262         return sb.ToString();
263     }
264     return "";
265 }
266
267 public string PrintNodes(TElement node)
268 {
269     var sb = new StringBuilder();
270     PrintNodes(node, sb);
271     return sb.ToString();
272 }
273
274 public void PrintNodes(TElement node, StringBuilder sb) => PrintNodes(node, sb, 0);
275
276 public void PrintNodes(TElement node, StringBuilder sb, int level)
277 {
278     if (IsEquals(node, default))
279     {
280         return;
281     }
282     PrintNodes(GetLeftOrDefault(node), sb, level + 1);
283     PrintNode(node, sb, level);
284     sb.AppendLine();
285     PrintNodes(GetRightOrDefault(node), sb, level + 1);
286 }
287
288 public string PrintNode(TElement node)
289 {
290     var sb = new StringBuilder();
291     PrintNode(node, sb);
292     return sb.ToString();
293 }
294
295 protected void PrintNode(TElement node, StringBuilder sb) => PrintNode(node, sb, 0);
296
297 protected virtual void PrintNode(TElement node, StringBuilder sb, int level)
298 {
299     sb.Append('\t', level);
300     sb.Append(node);
301     PrintNodeValue(node, sb);
302     sb.Append(' ');
303     sb.Append('s');
304     sb.Append(GetSize(node));
305 }
306
307 protected abstract void PrintNodeValue(TElement node, StringBuilder sb);

```

308 }
309 }

Index

- ./GenericCollectionMethodsBase.cs, 1
- ./Lists/CircularDoublyLinkedListMethods.cs, 2
- ./Lists/DoublyLinkedListMethodsBase.cs, 3
- ./Lists/OpenDoublyLinkedListMethods.cs, 3
- ./Trees/SizeBalancedTreeMethods.cs, 7
- ./Trees/SizeBalancedTreeMethods2.cs, 5
- ./Trees/SizedAndThreadedAVLBalancedTreeMethods.cs, 10
- ./Trees/SizedBinaryTreeMethodsBase.cs, 17