

LinksPlatform's Platform.Collections.Methods Class Library

./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      public abstract class GenericCollectionMethodsBase<TElement>
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         protected virtual TElement GetZero() => Integer<TElement>.Zero;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected virtual bool EqualToZero(TElement value) => EqualityComparer.Equals(value,
16             ↪ Zero);
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected virtual bool AreEqual(TElement first, TElement second) =>
20             ↪ EqualityComparer.Equals(first, second);
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected virtual bool GreaterThanZero(TElement value) => Comparer.Compare(value, Zero)
24             ↪ > 0;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected virtual bool GreaterThan(TElement first, TElement second) =>
28             ↪ Comparer.Compare(first, second) > 0;
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected virtual bool GreaterOrEqualThanZero(TElement value) => Comparer.Compare(value,
32             ↪ Zero) >= 0;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         protected virtual bool GreaterOrEqualThan(TElement first, TElement second) =>
36             ↪ Comparer.Compare(first, second) >= 0;
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected virtual bool LessOrEqualThanZero(TElement value) => Comparer.Compare(value,
40             ↪ Zero) <= 0;
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected virtual bool LessOrEqualThan(TElement first, TElement second) =>
44             ↪ Comparer.Compare(first, second) <= 0;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         protected virtual bool LessThanZero(TElement value) => Comparer.Compare(value, Zero) < 0;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected virtual bool LessThan(TElement first, TElement second) =>
51             ↪ Comparer.Compare(first, second) < 0;
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected virtual TElement Increment(TElement value) =>
55             ↪ Arithmetic<TElement>.Increment(value);
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected virtual TElement Decrement(TElement value) =>
59             ↪ Arithmetic<TElement>.Decrement(value);
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         protected virtual TElement Add(TElement first, TElement second) =>
63             ↪ Arithmetic<TElement>.Add(first, second);
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         protected virtual TElement Subtract(TElement first, TElement second) =>
67             ↪ Arithmetic<TElement>.Subtract(first, second);
68
69         protected readonly TElement Zero;
70         protected readonly TElement One;
71         protected readonly TElement Two;
72         protected readonly EqualityComparer<TElement> EqualityComparer;
73         protected readonly Comparer<TElement> Comparer;
74
75         protected GenericCollectionMethodsBase()
76         {
77             EqualityComparer = EqualityComparer<TElement>.Default;
78         }
79     }
80 }

```

```

65         Comparer = Comparer<TElement>.Default;
66         Zero = GetZero(); //-V3068
67         One = Increment(Zero); //-V3068
68         Two = Increment(One); //-V3068
69     }
70 }
71 }

```

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

```

./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 GetFirst();
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected abstract TElement GetLast();
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected abstract TElement GetPrevious(TElement element);
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected abstract TElement GetNext(TElement element);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected abstract TElement GetSize();
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected abstract void SetFirst(TElement element);
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected abstract void SetLast(TElement element);
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected abstract void SetPrevious(TElement element, TElement previous);
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected abstract void SetNext(TElement element, TElement next);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected abstract void SetSize(TElement size);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected void IncrementSize() => SetSize(Increment(GetSize()));
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected void DecrementSize() => SetSize(Decrement(GetSize()));
49     }
50 }

```

./Platform.Collections.Methods/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> :
        ↳ 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 (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, 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 }

```

./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                      }
57                  }
58              }
59          }
60      }
61  }

```

```

56         IncrementSize(root);
57         root = ref left;
58     }
59 }
60 }
61 else // node.Key greater than root.Key
62 {
63     if (EqualToZero(right))
64     {
65         IncrementSize(root);
66         SetSize(node, One);
67         right = node;
68         return;
69     }
70     if (FirstIsToTheRightOfSecond(node, right)) // node.Key greater than
        ↪ 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
                    ↪ of root and a node itself
92                 SetRight(root, Zero);
93                 SetSize(root, One);
94                 root = node;
95                 return;
96             }
97             RightRotate(ref right);
98             LeftRotate(ref root);
99         }
100         else
101         {
102             IncrementSize(root);
103             root = ref right;
104         }
105     }
106 }
107 }
108 }
109
110 protected override void DetachCore(ref TElement root, TElement node)
111 {
112     while (true)
113     {
114         ref var left = ref GetLeftReference(root);
115         var leftSize = GetSizeOrZero(left);
116         ref var right = ref GetRightReference(root);
117         var rightSize = GetSizeOrZero(right);
118         if (FirstIsToTheLeftOfSecond(node, root)) // node.Key less than root.Key
119         {
120             var decrementedLeftSize = Decrement(leftSize);
121             if (GreaterThan(GetSizeOrZero(GetRight(right)), decrementedLeftSize))
122             {
123                 LeftRotate(ref root);
124             }
125             else if (GreaterThan(GetSizeOrZero(GetLeft(right)), decrementedLeftSize))
126             {
127                 RightRotate(ref right);
128                 LeftRotate(ref root);
129             }
130             else
131             {

```

```

132         DecrementSize(root);
133         root = ref left;
134     }
135 }
136 else if (FirstIsToTheRightOfSecond(node, root)) // node.Key greater than root.Key
137 {
138     var decrementedRightSize = Decrement(rightSize);
139     if (GreaterThan(GetSizeOrZero(GetLeft(left)), decrementedRightSize))
140     {
141         RightRotate(ref root);
142     }
143     else if (GreaterThan(GetSizeOrZero(GetRight(left)), decrementedRightSize))
144     {
145         LeftRotate(ref left);
146         RightRotate(ref root);
147     }
148     else
149     {
150         DecrementSize(root);
151         root = ref right;
152     }
153 }
154 else // key equals to root.Key
155 {
156     if (GreaterThanZero(leftSize) && GreaterThanZero(rightSize))
157     {
158         TElement replacement;
159         if (GreaterThan(leftSize, rightSize))
160         {
161             replacement = 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 }

```

./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         }
18     }
19 }

```

```

16     else
17     {
18         IncrementSize(root);
19         if (FirstIsToTheLeftOfSecond(node, root))
20         {
21             AttachCore(ref GetLeftReference(root), node);
22             LeftMaintain(ref root);
23         }
24         else
25         {
26             AttachCore(ref GetRightReference(root), node);
27             RightMaintain(ref root);
28         }
29     }
30 }
31
32 protected override void DetachCore(ref TElement root, TElement nodeToDetach)
33 {
34     ref var currentNode = ref root;
35     ref var parent = ref root;
36     var replacementNode = Zero;
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 }

```



```

94     ClearNode(nodeToDetach);
95 }
96
97 private void LeftMaintain(ref TElement root)
98 {
99     if (!EqualToZero(root))
100     {
101         var rootLeftNode = GetLeft(root);
102         if (!EqualToZero(rootLeftNode))
103         {
104             var rootRightNode = GetRight(root);
105             var rootRightNodeSize = GetSize(rootRightNode);
106             var rootLeftNodeLeftNode = GetLeft(rootLeftNode);
107             if (!EqualToZero(rootLeftNodeLeftNode) &&
108                 (EqualToZero(rootRightNode) ||
109                  ⇨ GreaterThan(GetSize(rootLeftNodeLeftNode), rootRightNodeSize)))
110             {
111                 RightRotate(ref root);
112             }
113             else
114             {
115                 var rootLeftNodeRightNode = GetRight(rootLeftNode);
116                 if (!EqualToZero(rootLeftNodeRightNode) &&
117                     (EqualToZero(rootRightNode) ||
118                      ⇨ GreaterThan(GetSize(rootLeftNodeRightNode), rootRightNodeSize)))
119                 {
120                     LeftRotate(ref GetLeftReference(root));
121                     RightRotate(ref root);
122                 }
123                 else
124                 {
125                     return;
126                 }
127             }
128             LeftMaintain(ref GetLeftReference(root));
129             RightMaintain(ref GetRightReference(root));
130             LeftMaintain(ref root);
131             RightMaintain(ref root);
132         }
133     }
134 }
135
136 private void RightMaintain(ref TElement root)
137 {
138     if (!EqualToZero(root))
139     {
140         var rootRightNode = GetRight(root);
141         if (!EqualToZero(rootRightNode))
142         {
143             var rootLeftNode = GetLeft(root);
144             var rootLeftNodeSize = GetSize(rootLeftNode);
145             var rootRightNodeRightNode = GetRight(rootRightNode);
146             if (!EqualToZero(rootRightNodeRightNode) &&
147                 (EqualToZero(rootLeftNode) ||
148                  ⇨ GreaterThan(GetSize(rootRightNodeRightNode), rootLeftNodeSize)))
149             {
150                 LeftRotate(ref root);
151             }
152             else
153             {
154                 var rootRightNodeLeftNode = GetLeft(rootRightNode);
155                 if (!EqualToZero(rootRightNodeLeftNode) &&
156                     (EqualToZero(rootLeftNode) ||
157                      ⇨ GreaterThan(GetSize(rootRightNodeLeftNode), rootLeftNodeSize)))
158                 {
159                     RightRotate(ref GetRightReference(root));
160                     LeftRotate(ref root);
161                 }
162                 else
163                 {
164                     return;
165                 }
166             }
167             LeftMaintain(ref GetLeftReference(root));
168             RightMaintain(ref GetRightReference(root));
169             LeftMaintain(ref root);
170             RightMaintain(ref root);
171         }
172     }
173 }

```

```

168     }
169 }
170 }
171 }

```

./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
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Collections.Methods.Trees
11 {
12     /// <summary>
13     /// Combination of Size, Height (AVL), and threads.
14     /// </summary>
15     /// <remarks>
16     /// Based on: <a href="https://github.com/programatom/TreeLib/blob/master/TreeLib/TreeLib/G_
17     ↪ enerated/AVLTreeList.cs">TreeLib.AVLTreeList</a>.
18     /// Which itself based on: <a
19     ↪ href="https://github.com/GNOME/glib/blob/master/glib/gtree.c">GNOME/glib/gtree</a>.
20     /// </remarks>
21     public abstract class SizedAndThreadedAVLBalancedTreeMethods<TElement> :
22     ↪ SizedBinaryTreeMethodsBase<TElement>
23     {
24         private const int MaxPath = 92;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override TElement GetRighttest(TElement current)
28         {
29             var currentRight = GetRightOrDefault(current);
30             while (!EqualToZero(currentRight))
31             {
32                 current = currentRight;
33                 currentRight = GetRightOrDefault(current);
34             }
35             return current;
36         }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected override TElement GetLefttest(TElement current)
40         {
41             var currentLeft = GetLeftOrDefault(current);
42             while (!EqualToZero(currentLeft))
43             {
44                 current = currentLeft;
45                 currentLeft = GetLeftOrDefault(current);
46             }
47             return current;
48         }
49
50         public override bool Contains(TElement node, TElement root)
51         {
52             while (!EqualToZero(root))
53             {
54                 if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
55                 {
56                     root = GetLeftOrDefault(root);
57                 }
58                 else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
59                 {
60                     root = GetRightOrDefault(root);
61                 }
62                 else // node.Key == root.Key
63                 {
64                     return true;
65                 }
66             }
67             return false;
68         }
69
70         protected override void PrintNode(TElement node, StringBuilder sb, int level)
71         {
72             base.PrintNode(node, sb, level);
73             sb.Append(' ');
74         }
75     }
76 }

```

```

71         sb.Append(GetLeftIsChild(node) ? 'l' : 'L');
72         sb.Append(GetRightIsChild(node) ? 'r' : 'R');
73         sb.Append(' ');
74         sb.Append(GetBalance(node));
75     }
76
77     [MethodImpl(MethodImplOptions.AggressiveInlining)]
78     protected void IncrementBalance(TElement node) => SetBalance(node,
79         ↪ (sbyte)(GetBalance(node) + 1));
80
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     protected void DecrementBalance(TElement node) => SetBalance(node,
83         ↪ (sbyte)(GetBalance(node) - 1));
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     protected override TElement GetLeftOrDefault(TElement node) => GetLeftIsChild(node) ?
87         ↪ GetLeft(node) : default;
88
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     protected override TElement GetRightOrDefault(TElement node) => GetRightIsChild(node) ?
91         ↪ GetRight(node) : default;
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     protected abstract bool GetLeftIsChild(TElement node);
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     protected abstract void SetLeftIsChild(TElement node, bool value);
98
99     [MethodImpl(MethodImplOptions.AggressiveInlining)]
100    protected abstract bool GetRightIsChild(TElement node);
101
102    [MethodImpl(MethodImplOptions.AggressiveInlining)]
103    protected abstract void SetRightIsChild(TElement node, bool value);
104
105    [MethodImpl(MethodImplOptions.AggressiveInlining)]
106    protected abstract sbyte GetBalance(TElement node);
107
108    [MethodImpl(MethodImplOptions.AggressiveInlining)]
109    protected abstract void SetBalance(TElement node, sbyte value);
110
111    protected override void AttachCore(ref TElement root, TElement node)
112    {
113        unchecked
114        {
115            // TODO: Check what is faster to use simple array or array from array pool
116            // TODO: Try to use stackalloc as an optimization (requires code generation,
117            ↪ because of generics)
118
119            #if USEARRAYPOOL
120                var path = ArrayPool.Allocate<TElement>(MaxPath);
121                var pathPosition = 0;
122                path[pathPosition++] = default;
123            #else
124                var path = new TElement[MaxPath];
125                var pathPosition = 1;
126            #endif
127
128            var currentNode = root;
129            while (true)
130            {
131                if (FirstIsToTheLeftOfSecond(node, currentNode))
132                {
133                    if (GetLeftIsChild(currentNode))
134                    {
135                        IncrementSize(currentNode);
136                        path[pathPosition++] = currentNode;
137                        currentNode = GetLeft(currentNode);
138                    }
139                    else
140                    {
141                        // Threads
142                        SetLeft(node, GetLeft(currentNode));
143                        SetRight(node, currentNode);
144                        SetLeft(currentNode, node);
145                        SetLeftIsChild(currentNode, true);
146                        DecrementBalance(currentNode);
147                        SetSize(node, One);
148                        FixSize(currentNode); // Should be incremented already
149                        break;
150                    }
151                }
152            }
153        }
154    }

```

```

145     else if (FirstIsToTheRightOfSecond(node, currentNode))
146     {
147         if (GetRightIsChild(currentNode))
148         {
149             IncrementSize(currentNode);
150             path[pathPosition++] = currentNode;
151             currentNode = GetRight(currentNode);
152         }
153         else
154         {
155             // Threads
156             SetRight(node, GetRight(currentNode));
157             SetLeft(node, currentNode);
158             SetRight(currentNode, node);
159             SetRightIsChild(currentNode, true);
160             IncrementBalance(currentNode);
161             SetSize(node, One);
162             FixSize(currentNode); // Should be incremented already
163             break;
164         }
165     }
166     else
167     {
168         throw new InvalidOperationException("Node with the same key already
169         ↳ attached to a tree.");
170     }
171     // Restore balance. This is the goodness of a non-recursive
172     // implementation, when we are done with balancing we 'break'
173     // the loop and we are done.
174     while (true)
175     {
176         var parent = path[--pathPosition];
177         var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
178         ↳ GetLeft(parent));
179         var currentNodeBalance = GetBalance(currentNode);
180         if (currentNodeBalance < -1 || currentNodeBalance > 1)
181         {
182             currentNode = Balance(currentNode);
183             if (AreEqual(parent, default))
184             {
185                 root = currentNode;
186             }
187             else if (isLeftNode)
188             {
189                 SetLeft(parent, currentNode);
190                 FixSize(parent);
191             }
192             else
193             {
194                 SetRight(parent, currentNode);
195                 FixSize(parent);
196             }
197             currentNodeBalance = GetBalance(currentNode);
198             if (currentNodeBalance == 0 || AreEqual(parent, default))
199             {
200                 break;
201             }
202             if (isLeftNode)
203             {
204                 DecrementBalance(parent);
205             }
206             else
207             {
208                 IncrementBalance(parent);
209             }
210             currentNode = parent;
211         }
212     }
213     #if USEARRAYPOOL
214     ArrayPool.Free(path);
215     #endif
216 }
217
218 private TElement Balance(TElement node)
219 {
220     unchecked

```

```

221     {
222         var rootBalance = GetBalance(node);
223         if (rootBalance < -1)
224         {
225             var left = GetLeft(node);
226             if (GetBalance(left) > 0)
227             {
228                 SetLeft(node, LeftRotateWithBalance(left));
229                 FixSize(node);
230             }
231             node = RightRotateWithBalance(node);
232         }
233         else if (rootBalance > 1)
234         {
235             var right = GetRight(node);
236             if (GetBalance(right) < 0)
237             {
238                 SetRight(node, RightRotateWithBalance(right));
239                 FixSize(node);
240             }
241             node = LeftRotateWithBalance(node);
242         }
243         return node;
244     }
245 }
246
247 protected TElement LeftRotateWithBalance(TElement node)
248 {
249     unchecked
250     {
251         var right = GetRight(node);
252         if (GetLeftIsChild(right))
253         {
254             SetRight(node, GetLeft(right));
255         }
256         else
257         {
258             SetRightIsChild(node, false);
259             SetLeftIsChild(right, true);
260         }
261         SetLeft(right, node);
262         // Fix size
263         SetSize(right, GetSize(node));
264         FixSize(node);
265         // Fix balance
266         var rootBalance = GetBalance(node);
267         var rightBalance = GetBalance(right);
268         if (rightBalance <= 0)
269         {
270             if (rootBalance >= 1)
271             {
272                 SetBalance(right, (sbyte)(rightBalance - 1));
273             }
274             else
275             {
276                 SetBalance(right, (sbyte)(rootBalance + rightBalance - 2));
277             }
278             SetBalance(node, (sbyte)(rootBalance - 1));
279         }
280         else
281         {
282             if (rootBalance <= rightBalance)
283             {
284                 SetBalance(right, (sbyte)(rootBalance - 2));
285             }
286             else
287             {
288                 SetBalance(right, (sbyte)(rightBalance - 1));
289             }
290             SetBalance(node, (sbyte)(rootBalance - rightBalance - 1));
291         }
292         return right;
293     }
294 }
295
296 protected TElement RightRotateWithBalance(TElement node)
297 {
298     unchecked

```

```

299     {
300         var left = GetLeft(node);
301         if (GetRightIsChild(left))
302         {
303             SetLeft(node, GetRight(left));
304         }
305         else
306         {
307             SetLeftIsChild(node, false);
308             SetRightIsChild(left, true);
309         }
310         SetRight(left, node);
311         // Fix size
312         SetSize(left, GetSize(node));
313         FixSize(node);
314         // Fix balance
315         var rootBalance = GetBalance(node);
316         var leftBalance = GetBalance(left);
317         if (leftBalance <= 0)
318         {
319             if (leftBalance > rootBalance)
320             {
321                 SetBalance(left, (sbyte)(leftBalance + 1));
322             }
323             else
324             {
325                 SetBalance(left, (sbyte)(rootBalance + 2));
326             }
327             SetBalance(node, (sbyte)(rootBalance - leftBalance + 1));
328         }
329         else
330         {
331             if (rootBalance <= -1)
332             {
333                 SetBalance(left, (sbyte)(leftBalance + 1));
334             }
335             else
336             {
337                 SetBalance(left, (sbyte)(rootBalance + leftBalance + 2));
338             }
339             SetBalance(node, (sbyte)(rootBalance + 1));
340         }
341         return left;
342     }
343 }
344
345 [MethodImpl(MethodImplOptions.AggressiveInlining)]
346 protected override TElement GetNext(TElement node)
347 {
348     var current = GetRight(node);
349     if (GetRightIsChild(node))
350     {
351         return GetLefttest(current);
352     }
353     return current;
354 }
355
356 [MethodImpl(MethodImplOptions.AggressiveInlining)]
357 protected override TElement GetPrevious(TElement node)
358 {
359     var current = GetLeft(node);
360     if (GetLeftIsChild(node))
361     {
362         return GetRighttest(current);
363     }
364     return current;
365 }
366
367 protected override void DetachCore(ref TElement root, TElement node)
368 {
369     unchecked
370     {
371         #if USEARRAYPOOL
372             var path = ArrayPool.Allocate<TElement>(MaxPath);
373             var pathPosition = 0;
374             path[pathPosition++] = default;
375         #else
376             var path = new TElement[MaxPath];
377             var pathPosition = 1;

```

```

378 #endif
379
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))

```

```

455 {
456     var predecessor = GetPrevious(currentNode);
457     SetRight(predecessor, GetRight(currentNode));
458     var leftValue = GetLeft(currentNode);
459     if (AreEqual(parent, default))
460     {
461         root = leftValue;
462     }
463     else if (isLeftNode)
464     {
465         SetLeft(parent, leftValue);
466         IncrementBalance(parent);
467     }
468     else
469     {
470         SetRight(parent, leftValue);
471         DecrementBalance(parent);
472     }
473 }
474 else // node has a both children (left and right)
475 {
476     var predecessor = GetLeft(currentNode);
477     var successor = GetRight(currentNode);
478     var successorParent = currentNode;
479     int previousPathPosition = ++pathPosition;
480     // find the immediately next node (and its parent)
481     while (GetLeftIsChild(successor))
482     {
483         path[++pathPosition] = successorParent = successor;
484         successor = GetLeft(successor);
485         if (!AreEqual(successorParent, currentNode))
486         {
487             DecrementSize(successorParent);
488         }
489     }
490     path[previousPathPosition] = successor;
491     balanceNode = path[pathPosition];
492     // remove 'successor' from the tree
493     if (!AreEqual(successorParent, currentNode))
494     {
495         if (!GetRightIsChild(successor))
496         {
497             SetLeftIsChild(successorParent, false);
498         }
499         else
500         {
501             SetLeft(successorParent, GetRight(successor));
502         }
503         IncrementBalance(successorParent);
504         SetRightIsChild(successor, true);
505         SetRight(successor, GetRight(currentNode));
506     }
507     else
508     {
509         DecrementBalance(currentNode);
510     }
511     // set the predecessor's successor link to point to the right place
512     while (GetRightIsChild(predecessor))
513     {
514         predecessor = GetRight(predecessor);
515     }
516     SetRight(predecessor, successor);
517     // prepare 'successor' to replace 'node'
518     var left = GetLeft(currentNode);
519     SetLeftIsChild(successor, true);
520     SetLeft(successor, left);
521     SetBalance(successor, GetBalance(currentNode));
522     FixSize(successor);
523     if (AreEqual(parent, default))
524     {
525         root = successor;
526     }
527     else if (isLeftNode)
528     {
529         SetLeft(parent, successor);
530     }
531     else
532     {

```



```

533         SetRight(parent, successor);
534     }
535 }
536
537 // restore balance
538 if (!AreEqual(balanceNode, default))
539 {
540     while (true)
541     {
542         var balanceParent = path[--pathPosition];
543         isLeftNode = !AreEqual(balanceParent, default) && AreEqual(balanceNode,
544             ↪ GetLeft(balanceParent));
545         var currentNodeBalance = GetBalance(balanceNode);
546         if (currentNodeBalance < -1 || currentNodeBalance > 1)
547         {
548             balanceNode = Balance(balanceNode);
549             if (AreEqual(balanceParent, default))
550             {
551                 root = balanceNode;
552             }
553             else if (isLeftNode)
554             {
555                 SetLeft(balanceParent, balanceNode);
556             }
557             else
558             {
559                 SetRight(balanceParent, balanceNode);
560             }
561             currentNodeBalance = GetBalance(balanceNode);
562             if (currentNodeBalance != 0 || AreEqual(balanceParent, default))
563             {
564                 break;
565             }
566             if (isLeftNode)
567             {
568                 IncrementBalance(balanceParent);
569             }
570             else
571             {
572                 DecrementBalance(balanceParent);
573             }
574             balanceNode = balanceParent;
575         }
576     }
577     ClearNode(node);
578 #if USEARRAYPOOL
579     ArrayPool.Free(path);
580 #endif
581 }
582
583 [MethodImpl(MethodImplOptions.AggressiveInlining)]
584 protected override void ClearNode(TElement node)
585 {
586     SetLeft(node, Zero);
587     SetRight(node, Zero);
588     SetSize(node, Zero);
589     SetLeftIsChild(node, false);
590     SetRightIsChild(node, false);
591     SetBalance(node, 0);
592 }
593
594 }
595 }

```

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

```

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

```

```

13 [MethodImpl(MethodImplOptions.AggressiveInlining)]
14 protected abstract ref TElement GetLeftReference(TElement node);
15
16 [MethodImpl(MethodImplOptions.AggressiveInlining)]
17 protected abstract ref TElement GetRightReference(TElement node);
18
19 [MethodImpl(MethodImplOptions.AggressiveInlining)]
20 protected abstract TElement GetLeft(TElement node);
21
22 [MethodImpl(MethodImplOptions.AggressiveInlining)]
23 protected abstract TElement GetRight(TElement node);
24
25 [MethodImpl(MethodImplOptions.AggressiveInlining)]
26 protected abstract TElement GetSize(TElement node);
27
28 [MethodImpl(MethodImplOptions.AggressiveInlining)]
29 protected abstract void SetLeft(TElement node, TElement left);
30
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 protected abstract void SetRight(TElement node, TElement right);
33
34 [MethodImpl(MethodImplOptions.AggressiveInlining)]
35 protected abstract void SetSize(TElement node, TElement size);
36
37 [MethodImpl(MethodImplOptions.AggressiveInlining)]
38 protected abstract bool FirstIsToTheLeftOfSecond(TElement first, TElement second);
39
40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 protected abstract bool FirstIsToTheRightOfSecond(TElement first, TElement second);
42
43 [MethodImpl(MethodImplOptions.AggressiveInlining)]
44 protected virtual TElement GetLeftOrDefault(TElement node) => AreEqual(node, default) ?
    ↳ default : GetLeft(node);
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 protected virtual TElement GetRightOrDefault(TElement node) => AreEqual(node, default) ?
    ↳ default : GetRight(node);
48
49 [MethodImpl(MethodImplOptions.AggressiveInlining)]
50 protected void IncrementSize(TElement node) => SetSize(node, Increment(GetSize(node)));
51
52 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53 protected void DecrementSize(TElement node) => SetSize(node, Decrement(GetSize(node)));
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 protected TElement GetLeftSize(TElement node) => GetSizeOrZero(GetLeftOrDefault(node));
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 protected TElement GetRightSize(TElement node) => GetSizeOrZero(GetRightOrDefault(node));
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected TElement GetSizeOrZero(TElement node) => EqualToZero(node) ? Zero :
    ↳ GetSize(node);
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 protected void FixSize(TElement node) => SetSize(node, Increment(Add(GetLeftSize(node),
    ↳ GetRightSize(node))));
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 protected void LeftRotate(ref TElement root) => root = LeftRotate(root);
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 protected TElement LeftRotate(TElement root)
72 {
73     var right = GetRight(root);
74 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
75     if (EqualToZero(right))
76     {
77         throw new Exception("Right is null.");
78     }
79 #endif
80     SetRight(root, GetLeft(right));
81     SetLeft(right, root);
82     SetSize(right, GetSize(root));
83     FixSize(root);
84     return right;
85 }
86
87 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

88     protected void RightRotate(ref TElement root) => root = RightRotate(root);
89
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     protected TElement RightRotate(TElement root)
92     {
93         var left = GetLeft(root);
94 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
95         if (EqualToZero(left))
96         {
97             throw new Exception("Left is null.");
98         }
99 #endif
100         SetLeft(root, GetRight(left));
101         SetRight(left, root);
102         SetSize(left, GetSize(root));
103         FixSize(root);
104         return left;
105     }
106
107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
108     protected virtual TElement GetRightest(TElement current)
109     {
110         var currentRight = GetRight(current);
111         while (!EqualToZero(currentRight))
112         {
113             current = currentRight;
114             currentRight = GetRight(current);
115         }
116         return current;
117     }
118
119     [MethodImpl(MethodImplOptions.AggressiveInlining)]
120     protected virtual TElement GetLeftest(TElement current)
121     {
122         var currentLeft = GetLeft(current);
123         while (!EqualToZero(currentLeft))
124         {
125             current = currentLeft;
126             currentLeft = GetLeft(current);
127         }
128         return current;
129     }
130
131     [MethodImpl(MethodImplOptions.AggressiveInlining)]
132     protected virtual TElement GetNext(TElement node) => GetLeftest(GetRight(node));
133
134     [MethodImpl(MethodImplOptions.AggressiveInlining)]
135     protected virtual TElement GetPrevious(TElement node) => GetRightest(GetLeft(node));
136
137     [MethodImpl(MethodImplOptions.AggressiveInlining)]
138     public virtual bool Contains(TElement node, TElement root)
139     {
140         while (!EqualToZero(root))
141         {
142             if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
143             {
144                 root = GetLeft(root);
145             }
146             else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
147             {
148                 root = GetRight(root);
149             }
150             else // node.Key == root.Key
151             {
152                 return true;
153             }
154         }
155         return false;
156     }
157
158     [MethodImpl(MethodImplOptions.AggressiveInlining)]
159     protected virtual void ClearNode(TElement node)
160     {
161         SetLeft(node, Zero);
162         SetRight(node, Zero);
163         SetSize(node, Zero);
164     }
165
166     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

167         public void Attach(ref TElement root, TElement node)
168         {
169             #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
170                 ValidateSizes(root);
171                 Debug.WriteLine("--BeforeAttach--");
172                 Debug.WriteLine(PrintNodes(root));
173                 Debug.WriteLine("-----");
174                 var sizeBefore = GetSize(root);
175             #endif
176
177             if (EqualToZero(root))
178             {
179                 SetSize(node, One);
180                 root = node;
181                 return;
182             }
183             AttachCore(ref root, node);
184             #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
185                 Debug.WriteLine("--AfterAttach--");
186                 Debug.WriteLine(PrintNodes(root));
187                 Debug.WriteLine("-----");
188                 ValidateSizes(root);
189                 var sizeAfter = GetSize(root);
190                 if (!IsEquals(MathHelpers.Increment(sizeBefore), sizeAfter))
191                 {
192                     throw new Exception("Tree was broken after attach.");
193                 }
194             #endif
195         }
196
197         protected abstract void AttachCore(ref TElement root, TElement node);
198
199         [MethodImpl(MethodImplOptions.AggressiveInlining)]
200         public void Detach(ref TElement root, TElement node)
201         {
202             #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
203                 ValidateSizes(root);
204                 Debug.WriteLine("--BeforeDetach--");
205                 Debug.WriteLine(PrintNodes(root));
206                 Debug.WriteLine("-----");
207                 var sizeBefore = GetSize(root);
208                 if (ValueEqualToZero(root))
209                 {
210                     throw new Exception($"Элемент с {node} не содержится в дереве.");
211                 }
212             #endif
213             DetachCore(ref root, node);
214             #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
215                 Debug.WriteLine("--AfterDetach--");
216                 Debug.WriteLine(PrintNodes(root));
217                 Debug.WriteLine("-----");
218                 ValidateSizes(root);
219                 var sizeAfter = GetSize(root);
220                 if (!IsEquals(MathHelpers.Decrement(sizeBefore), sizeAfter))
221                 {
222                     throw new Exception("Tree was broken after detach.");
223                 }
224             #endif
225         }
226
227         protected abstract void DetachCore(ref TElement root, TElement node);
228
229         public void FixSizes(TElement node)
230         {
231             if (AreEqual(node, default))
232             {
233                 return;
234             }
235             FixSizes(GetLeft(node));
236             FixSizes(GetRight(node));
237             FixSize(node);
238         }
239
240         public void ValidateSizes(TElement node)
241         {
242             if (AreEqual(node, default))
243             {
244                 return;
245             }
246             var size = GetSize(node);

```

```

246     var leftSize = GetLeftSize(node);
247     var rightSize = GetRightSize(node);
248     var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
249     if (!AreEqual(size, expectedSize))
250     {
251         throw new InvalidOperationException($"Size of {node} is not valid. Expected
        ↳ size: {expectedSize}, actual size: {size}.");
252     }
253     ValidateSizes(GetLeft(node));
254     ValidateSizes(GetRight(node));
255 }
256
257 public void ValidateSize(TElement node)
258 {
259     var size = GetSize(node);
260     var leftSize = GetLeftSize(node);
261     var rightSize = GetRightSize(node);
262     var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
263     if (!AreEqual(size, expectedSize))
264     {
265         throw new InvalidOperationException($"Size of {node} is not valid. Expected
        ↳ size: {expectedSize}, actual size: {size}.");
266     }
267 }
268
269 public string PrintNodes(TElement node)
270 {
271     var sb = new StringBuilder();
272     PrintNodes(node, sb);
273     return sb.ToString();
274 }
275
276 [MethodImpl(MethodImplOptions.AggressiveInlining)]
277 public void PrintNodes(TElement node, StringBuilder sb) => PrintNodes(node, sb, 0);
278
279 public void PrintNodes(TElement node, StringBuilder sb, int level)
280 {
281     if (AreEqual(node, default))
282     {
283         return;
284     }
285     PrintNodes(GetLeft(node), sb, level + 1);
286     PrintNode(node, sb, level);
287     sb.AppendLine();
288     PrintNodes(GetRight(node), sb, level + 1);
289 }
290
291 public string PrintNode(TElement node)
292 {
293     var sb = new StringBuilder();
294     PrintNode(node, sb);
295     return sb.ToString();
296 }
297
298 [MethodImpl(MethodImplOptions.AggressiveInlining)]
299 protected void PrintNode(TElement node, StringBuilder sb) => PrintNode(node, sb, 0);
300
301 protected virtual void PrintNode(TElement node, StringBuilder sb, int level)
302 {
303     sb.Append('\t', level);
304     sb.Append(node);
305     PrintNodeValue(node, sb);
306     sb.Append(' ');
307     sb.Append('s');
308     sb.Append(GetSize(node));
309 }
310
311 protected abstract void PrintNodeValue(TElement node, StringBuilder sb);
312 }
313 }

```

./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
7 namespace Platform.Collections.Methods.Tests

```

```

8 {
9     public class RecursionlessSizeBalancedTree<TElement> :
        ↳ RecursionlessSizeBalancedTreeMethods<TElement>
10 {
11     private struct TreeElement
12     {
13         public TElement Size;
14         public TElement Left;
15         public TElement Right;
16     }
17
18     private readonly TreeElement[] _elements;
19     private TElement _allocated;
20
21     public TElement Root;
22
23     public TElement Count => GetSizeOrZero(Root);
24
25     public RecursionlessSizeBalancedTree(int capacity) => (_elements, _allocated) = (new
        ↳ TreeElement[capacity], Integer<TElement>.One);
26
27     public TElement Allocate()
28     {
29         var newNode = _allocated;
30         if (IsEmpty(newNode))
31         {
32             _allocated = Arithmetic.Increment(_allocated);
33             return newNode;
34         }
35         else
36         {
37             throw new InvalidOperationException("Allocated tree element is not empty.");
38         }
39     }
40
41     public void Free(TElement node)
42     {
43         while (!EqualityComparer.Equals(_allocated, Integer<TElement>.One) && IsEmpty(node))
44         {
45             var lastNode = Arithmetic.Decrement(_allocated);
46             if (EqualityComparer.Equals(lastNode, node))
47             {
48                 _allocated = lastNode;
49                 node = Arithmetic.Decrement(node);
50             }
51             else
52             {
53                 return;
54             }
55         }
56     }
57
58     public bool IsEmpty(TElement node) =>
        ↳ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
59
60     protected override bool FirstIsToLeftOfSecond(TElement first, TElement second) =>
        ↳ Comparer.Compare(first, second) < 0;
61
62     protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
        ↳ Comparer.Compare(first, second) > 0;
63
64     protected override ref TElement GetLeftReference(TElement node) => ref
        ↳ GetElement(node).Left;
65
66     protected override TElement GetLeft(TElement node) => GetElement(node).Left;
67
68     protected override ref TElement GetRightReference(TElement node) => ref
        ↳ GetElement(node).Right;
69
70     protected override TElement GetRight(TElement node) => GetElement(node).Right;
71
72     protected override TElement GetSize(TElement node) => GetElement(node).Size;
73
74     protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
        ↳ sb.Append(node);
75
76     protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
        ↳ left;
77

```

```

78     protected override void SetRight(TElement node, TElement right) =>
79         ↪ GetElement(node).Right = right;
80
81     protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
82         ↪ size;
83
84     private ref TreeElement GetElement(TElement node) => ref
85         ↪ _elements[(Integer<TElement>)node];
86 }
87 }

```

./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
7  namespace Platform.Collections.Methods.Tests
8  {
9      public class SizeBalancedTree<TElement> : SizeBalancedTreeMethods<TElement>
10     {
11         private struct TreeElement
12         {
13             public TElement Size;
14             public TElement Left;
15             public TElement Right;
16         }
17
18         private readonly TreeElement[] _elements;
19         private TElement _allocated;
20
21         public TElement Root;
22
23         public TElement Count => GetSizeOrZero(Root);
24
25         public SizeBalancedTree(int capacity) => (_elements, _allocated) = (new
26             ↪ TreeElement[capacity], Integer<TElement>.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, Integer<TElement>.One) && IsEmpty(node))
45             {
46                 var lastNode = Arithmetic.Decrement(_allocated);
47                 if (EqualityComparer.Equals(lastNode, node))
48                 {
49                     _allocated = lastNode;
50                     node = Arithmetic.Decrement(node);
51                 }
52                 else
53                 {
54                     return;
55                 }
56             }
57         }
58
59         public bool IsEmpty(TElement node) =>
60             ↪ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
61
62         protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
63             ↪ Comparer.Compare(first, second) < 0;
64
65         protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
66             ↪ Comparer.Compare(first, second) > 0;
67
68     }
69 }

```

```

64     protected override ref TElement GetLeftReference(TElement node) => ref
        ↳ GetElement(node).Left;
65
66     protected override TElement GetLeft(TElement node) => GetElement(node).Left;
67
68     protected override ref TElement GetRightReference(TElement node) => ref
        ↳ GetElement(node).Right;
69
70     protected override TElement GetRight(TElement node) => GetElement(node).Right;
71
72     protected override TElement GetSize(TElement node) => GetElement(node).Size;
73
74     protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
        ↳ sb.Append(node);
75
76     protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
        ↳ left;
77
78     protected override void SetRight(TElement node, TElement right) =>
        ↳ GetElement(node).Right = right;
79
80     protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
        ↳ size;
81
82     private ref TreeElement GetElement(TElement node) => ref
        ↳ _elements[(Integer<TElement>)node];
83 }
84 }

```

./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
7  namespace Platform.Collections.Methods.Tests
8  {
9      public class SizedAndThreadedAVLBalancedTree<TElement> :
        ↳ SizedAndThreadedAVLBalancedTreeMethods<TElement>
10     {
11         private struct TreeElement
12         {
13             public TElement Size;
14             public TElement Left;
15             public TElement Right;
16             public sbyte Balance;
17             public bool LeftIsChild;
18             public bool RightIsChild;
19         }
20
21         private readonly TreeElement[] _elements;
22         private TElement _allocated;
23
24         public TElement Root;
25
26         public TElement Count => GetSizeOrZero(Root);
27
28         public SizedAndThreadedAVLBalancedTree(int capacity) => (_elements, _allocated) = (new
        ↳ TreeElement[capacity], Integer<TElement>.One);
29
30         public TElement Allocate()
31         {
32             var newNode = _allocated;
33             if (IsEmpty(newNode))
34             {
35                 _allocated = Arithmetic.Increment(_allocated);
36                 return newNode;
37             }
38             else
39             {
40                 throw new InvalidOperationException("Allocated tree element is not empty.");
41             }
42         }
43
44         public void Free(TElement node)
45         {
46             while (!EqualityComparer.Equals(_allocated, Integer<TElement>.One) && IsEmpty(node))
47             {
48                 var lastNode = Arithmetic.Decrement(_allocated);

```



```

49         if (EqualityComparer.Equals(lastNode, node))
50         {
51             _allocated = lastNode;
52             node = Arithmetic.Decrement(node);
53         }
54         else
55         {
56             return;
57         }
58     }
59 }
60
61 public bool IsEmpty(TElement node) =>
62     ↳ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
63
64 protected override bool FirstIsToLeftOfSecond(TElement first, TElement second) =>
65     ↳ Comparer.Compare(first, second) < 0;
66
67 protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
68     ↳ Comparer.Compare(first, second) > 0;
69
70 protected override sbyte GetBalance(TElement node) => GetElement(node).Balance;
71
72 protected override bool GetLeftIsChild(TElement node) => GetElement(node).LeftIsChild;
73
74 protected override ref TElement GetLeftReference(TElement node) => ref
75     ↳ GetElement(node).Left;
76
77 protected override TElement GetLeft(TElement node) => GetElement(node).Left;
78
79 protected override bool GetRightIsChild(TElement node) => GetElement(node).RightIsChild;
80
81 protected override ref TElement GetRightReference(TElement node) => ref
82     ↳ GetElement(node).Right;
83
84 protected override TElement GetRight(TElement node) => GetElement(node).Right;
85
86 protected override TElement GetSize(TElement node) => GetElement(node).Size;
87
88 protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
89     ↳ sb.Append(node);
90
91 protected override void SetBalance(TElement node, sbyte value) =>
92     ↳ GetElement(node).Balance = value;
93
94 protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
95     ↳ left;
96
97 protected override void SetLeftIsChild(TElement node, bool value) =>
98     ↳ GetElement(node).LeftIsChild = value;
99
100 protected override void SetRight(TElement node, TElement right) =>
101     ↳ GetElement(node).Right = right;
102
103 protected override void SetRightIsChild(TElement node, bool value) =>
104     ↳ GetElement(node).RightIsChild = value;
105
106 protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
107     ↳ size;
108
109 private ref TreeElement GetElement(TElement node) => ref
110     ↳ _elements[(Integer<TElement>)node];
111
112 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Xunit;
4  using Platform.Numbers;
5  using Platform.Collections.Methods.Trees;
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)

```

```

12 {
13     for (var N = 1; N < maximumOperationsPerCycle; N++)
14     {
15         var currentCount = 0;
16         for (var i = 0; i < N; i++)
17         {
18             var node = allocate();
19             tree.Attach(ref root, node);
20             currentCount++;
21             Assert.Equal(currentCount, (int)(Integer<TElement>)treeCount());
22         }
23         for (var i = 1; i <= N; i++)
24         {
25             TElement node = (Integer<TElement>)i;
26             if (tree.Contains(node, root))
27             {
28                 tree.Detach(ref root, node);
29                 free(node);
30                 currentCount--;
31                 Assert.Equal(currentCount, (int)(Integer<TElement>)treeCount());
32             }
33         }
34     }
35 }
36
37 public static void TestMultipleRandomCreationsAndDeletions<TElement>(this
    ↳ SizedBinaryTreeMethodsBase<TElement> tree, ref TElement root, Func<TElement>
    ↳ treeCount, int maximumOperationsPerCycle)
38 {
39     var random = new System.Random(0);
40     var added = new HashSet<TElement>();
41     var currentCount = 0;
42     for (var N = 1; N < maximumOperationsPerCycle; N++)
43     {
44         for (var i = 0; i < N; i++)
45         {
46             var node = (Integer<TElement>)random.Next(1, N);
47             if (added.Add(node))
48             {
49                 tree.Attach(ref root, node);
50                 currentCount++;
51                 Assert.Equal(currentCount, (int)(Integer<TElement>)treeCount());
52             }
53         }
54         for (var i = 1; i <= N; i++)
55         {
56             TElement node = (Integer<TElement>)random.Next(1, N);
57             if (tree.Contains(node, root))
58             {
59                 tree.Detach(ref root, node);
60                 currentCount--;
61                 Assert.Equal(currentCount, (int)(Integer<TElement>)treeCount());
62                 added.Remove(node);
63             }
64         }
65     }
66 }
67 }
68 }

```

./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

./Platform.Collections.Methods.Tests/RecursionlessSizeBalancedTree.cs, 21
./Platform.Collections.Methods.Tests/SizeBalancedTree.cs, 23
./Platform.Collections.Methods.Tests/SizedAndThreadedAVLBalancedTree.cs, 24
./Platform.Collections.Methods.Tests/TestExtensions.cs, 25
./Platform.Collections.Methods.Tests/TreesTests.cs, 26
./Platform.Collections.Methods/GenericCollectionMethodsBase.cs, 1
./Platform.Collections.Methods/Lists/CircularDoublyLinkedListMethods.cs, 2
./Platform.Collections.Methods/Lists/DoublyLinkedListMethodsBase.cs, 3
./Platform.Collections.Methods/Lists/OpenDoublyLinkedListMethods.cs, 3
./Platform.Collections.Methods/Trees/RecursionlessSizeBalancedTreeMethods.cs, 5
./Platform.Collections.Methods/Trees/SizeBalancedTreeMethods.cs, 7
./Platform.Collections.Methods/Trees/SizedAndThreadedAVLBalancedTreeMethods.cs, 10
./Platform.Collections.Methods/Trees/SizedBinaryTreeMethodsBase.cs, 17