

# 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/SizeBalancedTreeMethods2.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 SizeBalancedTreeMethods2<TElement> :
8          ↳ SizedBinaryTreeMethodsBase<TElement>
9      {
10         protected override void AttachCore(ref TElement root, TElement node)
11         {
12             if (EqualToZero(root))
13             {
14                 root = node;
15                 IncrementSize(root);
16             }
17             else
18             {
19                 IncrementSize(root);
20                 if (FirstIsToTheLeftOfSecond(node, root))
21                 {
22                     AttachCore(ref GetLeftReference(root), node);
23                     LeftMaintain(ref root);
24                 }
25                 else
26                 {
27                     AttachCore(ref GetRightReference(root), node);
28                     RightMaintain(ref root);
29                 }
30             }
31         }
32
33         protected override void DetachCore(ref TElement root, TElement nodeToDetach)
34         {
35             ref var currentNode = ref root;
36             ref var parent = ref root;
37             var replacementNode = Zero;
38             while (!AreEqual(currentNode, nodeToDetach))
39             {
40                 SetSize(currentNode, Decrement(GetSize(currentNode)));
41                 if (FirstIsToTheLeftOfSecond(nodeToDetach, currentNode))
42                 {
43                     parent = ref currentNode;
44                     currentNode = ref GetLeftReference(currentNode);
45                 }
46                 else if (FirstIsToTheRightOfSecond(nodeToDetach, currentNode))
47                 {
48                     parent = ref currentNode;
49                     currentNode = ref GetRightReference(currentNode);
50                 }
51                 else
52                 {
53                     throw new InvalidOperationException("Duplicate link found in the tree.");
54                 }
55             }
56             var nodeToDetachLeft = GetLeft(nodeToDetach);
57             var node = GetRight(nodeToDetach);

```

```

57     if (!EqualToZero(nodeToDetachLeft) && !EqualToZero(node))
58     {
59         var minNode = node;
60         var minNodeLeft = GetLeft(minNode);
61         while (!EqualToZero(minNodeLeft))
62         {
63             minNode = minNodeLeft;
64             minNodeLeft = GetLeft(minNode);
65         }
66         DetachCore(ref GetRightReference(nodeToDetach), minNode);
67         SetLeft(minNode, nodeToDetachLeft);
68         node = GetRight(nodeToDetach);
69         if (!EqualToZero(node))
70         {
71             SetRight(minNode, node);
72             SetSize(minNode, Increment(Add(GetSize(nodeToDetachLeft), GetSize(node))));
73         }
74         else
75         {
76             SetSize(minNode, Increment(GetSize(nodeToDetachLeft)));
77         }
78         replacementNode = minNode;
79     }
80     else if (!EqualToZero(nodeToDetachLeft))
81     {
82         replacementNode = nodeToDetachLeft;
83     }
84     else if (!EqualToZero(node))
85     {
86         replacementNode = node;
87     }
88     if (AreEqual(root, nodeToDetach))
89     {
90         root = replacementNode;
91     }
92     else if (AreEqual(GetLeft(parent), nodeToDetach))
93     {
94         SetLeft(parent, replacementNode);
95     }
96     else if (AreEqual(GetRight(parent), nodeToDetach))
97     {
98         SetRight(parent, replacementNode);
99     }
100    ClearNode(nodeToDetach);
101 }
102
103 private void LeftMaintain(ref TElement root)
104 {
105     if (!EqualToZero(root))
106     {
107         var rootLeftNode = GetLeft(root);
108         if (!EqualToZero(rootLeftNode))
109         {
110             var rootRightNode = GetRight(root);
111             var rootLeftNodeLeftNode = GetLeft(rootLeftNode);
112             if (!EqualToZero(rootLeftNodeLeftNode) &&
113                 (EqualToZero(rootRightNode) ||
114                  ⇨ GreaterThan(GetSize(rootLeftNodeLeftNode), GetSize(rootRightNode))))
115             {
116                 RightRotate(ref root);
117             }
118             else
119             {
120                 var rootLeftNodeRightNode = GetRight(rootLeftNode);
121                 if (!EqualToZero(rootLeftNodeRightNode) &&
122                     (EqualToZero(rootRightNode) ||
123                      ⇨ GreaterThan(GetSize(rootLeftNodeRightNode),
124                      ⇨ GetSize(rootRightNode))))
125                 {
126                     LeftRotate(ref GetLeftReference(root));
127                     RightRotate(ref root);
128                 }
129                 else
130                 {
131                     return;
132                 }
133             }
134             LeftMaintain(ref GetLeftReference(root));
135         }
136     }
137 }

```

```

132         RightMaintain(ref GetRightReference(root));
133         LeftMaintain(ref root);
134         RightMaintain(ref root);
135     }
136 }
137
138
139 private void RightMaintain(ref TElement root)
140 {
141     if (!EqualToZero(root))
142     {
143         var rootRightNode = GetRight(root);
144         if (!EqualToZero(rootRightNode))
145         {
146             var rootLeftNode = GetLeft(root);
147             var rootRightNodeRightNode = GetRight(rootRightNode);
148             if (!EqualToZero(rootRightNodeRightNode) &&
149                 (EqualToZero(rootLeftNode) ||
150                  ⇨ GreaterThan(GetSize(rootRightNodeRightNode), GetSize(rootLeftNode))))
151             {
152                 LeftRotate(ref root);
153             }
154             else
155             {
156                 var rootRightNodeLeftNode = GetLeft(rootRightNode);
157                 if (!EqualToZero(rootRightNodeLeftNode) &&
158                     (EqualToZero(rootLeftNode) ||
159                      ⇨ GreaterThan(GetSize(rootRightNodeLeftNode),
160                      ⇨ GetSize(rootLeftNode))))
161                 {
162                     RightRotate(ref GetRightReference(root));
163                     LeftRotate(ref root);
164                 }
165                 else
166                 {
167                     return;
168                 }
169             }
170             LeftMaintain(ref GetLeftReference(root));
171             RightMaintain(ref GetRightReference(root));
172             LeftMaintain(ref root);
173             RightMaintain(ref root);
174         }
175     }
176 }
177
178 }
179
180 }

```

./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             while (true)
13             {
14                 ref var left = ref GetLeftReference(root);
15                 var leftSize = GetSizeOrZero(left);
16                 ref var right = ref GetRightReference(root);
17                 var rightSize = GetSizeOrZero(right);
18                 if (FirstIsToTheLeftOfSecond(node, root)) // node.Key less than root.Key
19                 {
20                     if (EqualToZero(left))
21                     {
22                         IncrementSize(root);
23                         SetSize(node, One);
24                         left = node;
25                         return;
26                     }
27                     if (FirstIsToTheLeftOfSecond(node, left)) // node.Key less than left.Key
28                     {
29                         if (GreaterThan(Increment(leftSize), rightSize))

```

```

29         {
30             RightRotate(ref root);
31         }
32         else
33         {
34             IncrementSize(root);
35             root = ref left;
36         }
37     }
38     else // node.Key greater than left.Key
39     {
40         var leftRightSize = GetSizeOrZero(GetRight(left));
41         if (GreaterThan(Increment(leftRightSize), rightSize))
42         {
43             if (EqualToZero(leftRightSize) && EqualToZero(rightSize))
44             {
45                 SetLeft(node, left);
46                 SetRight(node, root);
47                 SetSize(node, Add(leftSize, Two)); // Two (2) - node the size of
48                 ↪ root and a node itself
49                 SetLeft(root, Zero);
50                 SetSize(root, One);
51                 root = node;
52                 return;
53             }
54             LeftRotate(ref left);
55             RightRotate(ref root);
56         }
57         else
58         {
59             IncrementSize(root);
60             root = ref left;
61         }
62     }
63     else // node.Key greater than root.Key
64     {
65         if (EqualToZero(right))
66         {
67             IncrementSize(root);
68             SetSize(node, One);
69             right = node;
70             return;
71         }
72         if (FirstIsToTheRightOfSecond(node, right)) // node.Key greater than
73         ↪ right.Key
74         {
75             if (GreaterThan(Increment(rightSize), leftSize))
76             {
77                 LeftRotate(ref root);
78             }
79             else
80             {
81                 IncrementSize(root);
82                 root = ref right;
83             }
84         }
85         else // node.Key less than right.Key
86         {
87             var rightLeftSize = GetSizeOrZero(GetLeft(right));
88             if (GreaterThan(Increment(rightLeftSize), leftSize))
89             {
90                 if (EqualToZero(rightLeftSize) && EqualToZero(leftSize))
91                 {
92                     SetLeft(node, root);
93                     SetRight(node, right);
94                     SetSize(node, Add(rightSize, Two)); // Two (2) - node the size
95                     ↪ of root and a node itself
96                     SetRight(root, Zero);
97                     SetSize(root, One);
98                     root = node;
99                     return;
100                 }
101                 RightRotate(ref right);
102                 LeftRotate(ref root);
103             }
104             else
105             {
106                 IncrementSize(root);

```



```

105         root = ref right;
106     }
107 }
108 }
109 }
110 }
111
112 protected override void DetachCore(ref TElement root, TElement node)
113 {
114     while (true)
115     {
116         ref var left = ref GetLeftReference(root);
117         var leftSize = GetSizeOrZero(left);
118         ref var right = ref GetRightReference(root);
119         var rightSize = GetSizeOrZero(right);
120         if (FirstIsToTheLeftOfSecond(node, root)) // node.Key less than root.Key
121         {
122             var decrementedLeftSize = Decrement(leftSize);
123             if (GreaterThan(GetSizeOrZero(GetRight(right)), decrementedLeftSize))
124             {
125                 LeftRotate(ref root);
126             }
127             else if (GreaterThan(GetSizeOrZero(GetLeft(right)), decrementedLeftSize))
128             {
129                 RightRotate(ref right);
130                 LeftRotate(ref root);
131             }
132             else
133             {
134                 DecrementSize(root);
135                 root = ref left;
136             }
137         }
138         else if (FirstIsToTheRightOfSecond(node, root)) // node.Key greater than root.Key
139         {
140             var decrementedRightSize = Decrement(rightSize);
141             if (GreaterThan(GetSizeOrZero(GetLeft(left)), decrementedRightSize))
142             {
143                 RightRotate(ref root);
144             }
145             else if (GreaterThan(GetSizeOrZero(GetRight(left)), decrementedRightSize))
146             {
147                 LeftRotate(ref left);
148                 RightRotate(ref root);
149             }
150             else
151             {
152                 DecrementSize(root);
153                 root = ref right;
154             }
155         }
156         else // key equals to root.Key
157         {
158             if (GreaterThanZero(leftSize) && GreaterThanZero(rightSize))
159             {
160                 TElement replacement;
161                 if (GreaterThan(leftSize, rightSize))
162                 {
163                     replacement = left;
164                     var replacementRight = GetRight(replacement);
165                     while (!EqualToZero(replacementRight))
166                     {
167                         replacement = replacementRight;
168                         replacementRight = GetRight(replacement);
169                     }
170                     DetachCore(ref left, replacement);
171                 }
172                 else
173                 {
174                     replacement = right;
175                     var replacementLeft = GetLeft(replacement);
176                     while (!EqualToZero(replacementLeft))
177                     {
178                         replacement = replacementLeft;
179                         replacementLeft = GetLeft(replacement);
180                     }
181                     DetachCore(ref right, replacement);
182                 }
183             }
184         }
185     }
186 }

```

```

183         SetLeft(replacement, left);
184         SetRight(replacement, right);
185         SetSize(replacement, Add(leftSize, rightSize));
186         root = replacement;
187     }
188     else if (GreaterThanZero(leftSize))
189     {
190         root = left;
191     }
192     else if (GreaterThanZero(rightSize))
193     {
194         root = right;
195     }
196     else
197     {
198         root = Zero;
199     }
200     ClearNode(node);
201     return;
202 }
203 }
204 }
205 }
206 }

```

./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         // TODO: Link with size of TElement
25         private const int MaxPath = 92;
26
27         public override bool Contains(TElement node, TElement root)
28         {
29             while (!EqualToZero(root))
30             {
31                 if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
32                 {
33                     root = GetLeftOrDefault(root);
34                 }
35                 else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
36                 {
37                     root = GetRightOrDefault(root);
38                 }
39                 else // node.Key == root.Key
40                 {
41                     return true;
42                 }
43             }
44             return false;
45         }
46
47         protected override void PrintNode(TElement node, StringBuilder sb, int level)
48         {
49             base.PrintNode(node, sb, level);
50             sb.Append(' ');
51             sb.Append(GetLeftIsChild(node) ? 'l' : 'L');
52             sb.Append(GetRightIsChild(node) ? 'r' : 'R');
53             sb.Append(' ');
54         }
55     }
56 }

```

```

51     sb.Append(GetBalance(node));
52 }
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected void IncrementBalance(TElement node) => SetBalance(node,
56     ↪ (sbyte)(GetBalance(node) + 1));
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 protected void DecrementBalance(TElement node) => SetBalance(node,
60     ↪ (sbyte)(GetBalance(node) - 1));
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected override TElement GetLeftOrDefault(TElement node) => GetLeftIsChild(node) ?
64     ↪ GetLeft(node) : default;
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 protected override TElement GetRightOrDefault(TElement node) => GetRightIsChild(node) ?
68     ↪ GetRight(node) : default;
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 protected abstract bool GetLeftIsChild(TElement node);
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 protected abstract void SetLeftIsChild(TElement node, bool value);
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 protected abstract bool GetRightIsChild(TElement node);
78
79 [MethodImpl(MethodImplOptions.AggressiveInlining)]
80 protected abstract void SetRightIsChild(TElement node, bool value);
81
82 [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 protected abstract sbyte GetBalance(TElement node);
84
85 [MethodImpl(MethodImplOptions.AggressiveInlining)]
86 protected abstract void SetBalance(TElement node, sbyte value);
87
88 protected override void AttachCore(ref TElement root, TElement node)
89 {
90     unchecked
91     {
92         // TODO: Check what is faster to use simple array or array from array pool
93         // TODO: Try to use stackalloc as an optimization (requires code generation,
94         ↪ because of generics)
95
96 #if USEARRAYPOOL
97     var path = ArrayPool.Allocate<TElement>(MaxPath);
98     var pathPosition = 0;
99     path[pathPosition++] = default;
100 #else
101     var path = new TElement[MaxPath];
102     var pathPosition = 1;
103 #endif
104     var currentNode = root;
105     while (true)
106     {
107         if (FirstIsToTheLeftOfSecond(node, currentNode))
108         {
109             if (GetLeftIsChild(currentNode))
110             {
111                 IncrementSize(currentNode);
112                 path[pathPosition++] = currentNode;
113                 currentNode = GetLeft(currentNode);
114             }
115             else
116             {
117                 // Threads
118                 SetLeft(node, GetLeft(currentNode));
119                 SetRight(node, currentNode);
120                 SetLeft(currentNode, node);
121                 SetLeftIsChild(currentNode, true);
122                 DecrementBalance(currentNode);
123                 SetSize(node, One);
124                 FixSize(currentNode); // Should be incremented already
125                 break;
126             }
127         }
128         else if (FirstIsToTheRightOfSecond(node, currentNode))
129         {
130             if (GetRightIsChild(currentNode))

```

```

125         {
126             IncrementSize(currentNode);
127             path[pathPosition++] = currentNode;
128             currentNode = GetRight(currentNode);
129         }
130         else
131         {
132             // Threads
133             SetRight(node, GetRight(currentNode));
134             SetLeft(node, currentNode);
135             SetRight(currentNode, node);
136             SetRightIsChild(currentNode, true);
137             IncrementBalance(currentNode);
138             SetSize(node, One);
139             FixSize(currentNode); // Should be incremented already
140             break;
141         }
142     }
143     else
144     {
145         throw new InvalidOperationException("Node with the same key already
146             ↳ attached to a tree.");
147     }
148 }
149 // Restore balance. This is the goodness of a non-recursive
150 // implementation, when we are done with balancing we 'break'
151 // the loop and we are done.
152 while (true)
153 {
154     var parent = path[--pathPosition];
155     var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
156         ↳ GetLeft(parent));
157     var currentNodeBalance = GetBalance(currentNode);
158     if (currentNodeBalance < -1 || currentNodeBalance > 1)
159     {
160         currentNode = Balance(currentNode);
161         if (AreEqual(parent, default))
162         {
163             root = currentNode;
164         }
165         else if (isLeftNode)
166         {
167             SetLeft(parent, currentNode);
168             FixSize(parent);
169         }
170         else
171         {
172             SetRight(parent, currentNode);
173             FixSize(parent);
174         }
175     }
176     currentNodeBalance = GetBalance(currentNode);
177     if (currentNodeBalance == 0 || AreEqual(parent, default))
178     {
179         break;
180     }
181     if (isLeftNode)
182     {
183         DecrementBalance(parent);
184     }
185     else
186     {
187         IncrementBalance(parent);
188     }
189     currentNode = parent;
190 }
191 #if USEARRAYPOOL
192     ArrayPool.Free(path);
193 #endif
194 }
195 private TElement Balance(TElement node)
196 {
197     unchecked
198     {
199         var rootBalance = GetBalance(node);
200         if (rootBalance < -1)

```

```

201     {
202         var left = GetLeft(node);
203         if (GetBalance(left) > 0)
204         {
205             SetLeft(node, LeftRotateWithBalance(left));
206             FixSize(node);
207         }
208         node = RightRotateWithBalance(node);
209     }
210     else if (rootBalance > 1)
211     {
212         var right = GetRight(node);
213         if (GetBalance(right) < 0)
214         {
215             SetRight(node, RightRotateWithBalance(right));
216             FixSize(node);
217         }
218         node = LeftRotateWithBalance(node);
219     }
220     return node;
221 }
222
223
224 protected TElement LeftRotateWithBalance(TElement node)
225 {
226     unchecked
227     {
228         var right = GetRight(node);
229         if (GetLeftIsChild(right))
230         {
231             SetRight(node, GetLeft(right));
232         }
233         else
234         {
235             SetRightIsChild(node, false);
236             SetLeftIsChild(right, true);
237         }
238         SetLeft(right, node);
239         // Fix size
240         SetSize(right, GetSize(node));
241         FixSize(node);
242         // Fix balance
243         var rootBalance = GetBalance(node);
244         var rightBalance = GetBalance(right);
245         if (rightBalance <= 0)
246         {
247             if (rootBalance >= 1)
248             {
249                 SetBalance(right, (sbyte)(rightBalance - 1));
250             }
251             else
252             {
253                 SetBalance(right, (sbyte)(rootBalance + rightBalance - 2));
254             }
255             SetBalance(node, (sbyte)(rootBalance - 1));
256         }
257         else
258         {
259             if (rootBalance <= rightBalance)
260             {
261                 SetBalance(right, (sbyte)(rootBalance - 2));
262             }
263             else
264             {
265                 SetBalance(right, (sbyte)(rightBalance - 1));
266             }
267             SetBalance(node, (sbyte)(rootBalance - rightBalance - 1));
268         }
269         return right;
270     }
271 }
272
273 protected TElement RightRotateWithBalance(TElement node)
274 {
275     unchecked
276     {
277         var left = GetLeft(node);
278         if (GetRightIsChild(left))

```

```

279     {
280         SetLeft(node, GetRight(left));
281     }
282     else
283     {
284         SetLeftIsChild(node, false);
285         SetRightIsChild(left, true);
286     }
287     SetRight(left, node);
288     // Fix size
289     SetSize(left, GetSize(node));
290     FixSize(node);
291     // Fix balance
292     var rootBalance = GetBalance(node);
293     var leftBalance = GetBalance(left);
294     if (leftBalance <= 0)
295     {
296         if (leftBalance > rootBalance)
297         {
298             SetBalance(left, (sbyte)(leftBalance + 1));
299         }
300         else
301         {
302             SetBalance(left, (sbyte)(rootBalance + 2));
303         }
304         SetBalance(node, (sbyte)(rootBalance - leftBalance + 1));
305     }
306     else
307     {
308         if (rootBalance <= -1)
309         {
310             SetBalance(left, (sbyte)(leftBalance + 1));
311         }
312         else
313         {
314             SetBalance(left, (sbyte)(rootBalance + leftBalance + 2));
315         }
316         SetBalance(node, (sbyte)(rootBalance + 1));
317     }
318     return left;
319 }
320 }
321
322 protected TElement GetNext(TElement node)
323 {
324     unchecked
325     {
326         var current = GetRight(node);
327         if (GetRightIsChild(node))
328         {
329             while (GetLeftIsChild(current))
330             {
331                 current = GetLeft(current);
332             }
333         }
334         return current;
335     }
336 }
337
338 protected TElement GetPrevious(TElement node)
339 {
340     unchecked
341     {
342         var current = GetLeft(node);
343         if (GetLeftIsChild(node))
344         {
345             while (GetRightIsChild(current))
346             {
347                 current = GetRight(current);
348             }
349         }
350         return current;
351     }
352 }
353
354 protected override void DetachCore(ref TElement root, TElement node)
355 {
356     unchecked
357     {

```

```

358 #if USEARRAYPOOL
359     var path = ArrayPool.Allocate<TElement>(MaxPath);
360     var pathPosition = 0;
361     path[pathPosition++] = default;
362 #else
363     var path = new TElement[MaxPath];
364     var pathPosition = 1;
365 #endif
366     var currentNode = root;
367     while (true)
368     {
369         if (FirstIsToTheLeftOfSecond(node, currentNode))
370         {
371             if (!GetLeftIsChild(currentNode))
372             {
373                 throw new InvalidOperationException("Cannot find a node.");
374             }
375             DecrementSize(currentNode);
376             path[pathPosition++] = currentNode;
377             currentNode = GetLeft(currentNode);
378         }
379         else if (FirstIsToTheRightOfSecond(node, currentNode))
380         {
381             if (!GetRightIsChild(currentNode))
382             {
383                 throw new InvalidOperationException("Cannot find a node.");
384             }
385             DecrementSize(currentNode);
386             path[pathPosition++] = currentNode;
387             currentNode = GetRight(currentNode);
388         }
389         else
390         {
391             break;
392         }
393     }
394     var parent = path[--pathPosition];
395     var balanceNode = parent;
396     var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
397     ↪ GetLeft(parent));
398     if (!GetLeftIsChild(currentNode))
399     {
400         if (!GetRightIsChild(currentNode)) // node has no children
401         {
402             if (AreEqual(parent, default))
403             {
404                 root = Zero;
405             }
406             else if (isLeftNode)
407             {
408                 SetLeftIsChild(parent, false);
409                 SetLeft(parent, GetLeft(currentNode));
410                 IncrementBalance(parent);
411             }
412             else
413             {
414                 SetRightIsChild(parent, false);
415                 SetRight(parent, GetRight(currentNode));
416                 DecrementBalance(parent);
417             }
418         }
419         else // node has a right child
420         {
421             var successor = GetNext(currentNode);
422             SetLeft(successor, GetLeft(currentNode));
423             var right = GetRight(currentNode);
424             if (AreEqual(parent, default))
425             {
426                 root = right;
427             }
428             else if (isLeftNode)
429             {
430                 SetLeft(parent, right);
431                 IncrementBalance(parent);
432             }
433             else
434             {
435                 SetRight(parent, right);

```

```

435         DecrementBalance(parent);
436     }
437 }
438 }
439 else // node has a left child
440 {
441     if (!GetRightIsChild(currentNode))
442     {
443         var predecessor = GetPrevious(currentNode);
444         SetRight(predecessor, GetRight(currentNode));
445         var leftValue = GetLeft(currentNode);
446         if (AreEqual(parent, default))
447         {
448             root = leftValue;
449         }
450         else if (isLeftNode)
451         {
452             SetLeft(parent, leftValue);
453             IncrementBalance(parent);
454         }
455         else
456         {
457             SetRight(parent, leftValue);
458             DecrementBalance(parent);
459         }
460     }
461     else // node has a both children (left and right)
462     {
463         var predecessor = GetLeft(currentNode);
464         var successor = GetRight(currentNode);
465         var successorParent = currentNode;
466         int previousPathPosition = ++pathPosition;
467         // find the immediately next node (and its parent)
468         while (GetLeftIsChild(successor))
469         {
470             path[++pathPosition] = successorParent = successor;
471             successor = GetLeft(successor);
472             if (!AreEqual(successorParent, currentNode))
473             {
474                 DecrementSize(successorParent);
475             }
476         }
477         path[previousPathPosition] = successor;
478         balanceNode = path[pathPosition];
479         // remove 'successor' from the tree
480         if (!AreEqual(successorParent, currentNode))
481         {
482             if (!GetRightIsChild(successor))
483             {
484                 SetLeftIsChild(successorParent, false);
485             }
486             else
487             {
488                 SetLeft(successorParent, GetRight(successor));
489             }
490             IncrementBalance(successorParent);
491             SetRightIsChild(successor, true);
492             SetRight(successor, GetRight(currentNode));
493         }
494         else
495         {
496             DecrementBalance(currentNode);
497         }
498         // set the predecessor's successor link to point to the right place
499         while (GetRightIsChild(predecessor))
500         {
501             predecessor = GetRight(predecessor);
502         }
503         SetRight(predecessor, successor);
504         // prepare 'successor' to replace 'node'
505         var left = GetLeft(currentNode);
506         SetLeftIsChild(successor, true);
507         SetLeft(successor, left);
508         SetBalance(successor, GetBalance(currentNode));
509         FixSize(successor);
510         if (AreEqual(parent, default))
511         {
512             root = successor;

```



```

513     }
514     else if (isLeftNode)
515     {
516         SetLeft(parent, successor);
517     }
518     else
519     {
520         SetRight(parent, successor);
521     }
522 }
523 }
524 // restore balance
525 if (!AreEqual(balanceNode, default))
526 {
527     while (true)
528     {
529         var balanceParent = path[--pathPosition];
530         isLeftNode = !AreEqual(balanceParent, default) && AreEqual(balanceNode,
531             ↪ GetLeft(balanceParent));
532         var currentNodeBalance = GetBalance(balanceNode);
533         if (currentNodeBalance < -1 || currentNodeBalance > 1)
534         {
535             balanceNode = Balance(balanceNode);
536             if (AreEqual(balanceParent, default))
537             {
538                 root = balanceNode;
539             }
540             else if (isLeftNode)
541             {
542                 SetLeft(balanceParent, balanceNode);
543             }
544             else
545             {
546                 SetRight(balanceParent, balanceNode);
547             }
548             currentNodeBalance = GetBalance(balanceNode);
549             if (currentNodeBalance != 0 || AreEqual(balanceParent, default))
550             {
551                 break;
552             }
553             if (isLeftNode)
554             {
555                 IncrementBalance(balanceParent);
556             }
557             else
558             {
559                 DecrementBalance(balanceParent);
560             }
561             balanceNode = balanceParent;
562         }
563     }
564     ClearNode(node);
565 #if USEARRAYPOOL
566     ArrayPool.Free(path);
567 #endif
568 }
569 }
570
571 [MethodImpl(MethodImplOptions.AggressiveInlining)]
572 protected override void ClearNode(TElement node)
573 {
574     SetLeft(node, Zero);
575     SetRight(node, Zero);
576     SetSize(node, Zero);
577     SetLeftIsChild(node, false);
578     SetRightIsChild(node, false);
579     SetBalance(node, 0);
580 }
581 }
582 }

```

./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     {
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected abstract ref TElement GetLeftReference(TElement node);
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected abstract ref TElement GetRightReference(TElement node);
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected abstract TElement GetLeft(TElement node);
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected abstract TElement GetRight(TElement node);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected abstract TElement GetSize(TElement node);
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected abstract void SetLeft(TElement node, TElement left);
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected abstract void SetRight(TElement node, TElement right);
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected abstract void SetSize(TElement node, TElement size);
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         protected abstract bool FirstIsToLeftOfSecond(TElement first, TElement second);
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected abstract bool FirstIsToTheRightOfSecond(TElement first, TElement second);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         protected virtual TElement GetLeftOrDefault(TElement node) => AreEqual(node, default) ?
46             ↳ default : GetLeft(node);
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected virtual TElement GetRightOrDefault(TElement node) => AreEqual(node, default) ?
50             ↳ default : GetRight(node);
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected void IncrementSize(TElement node) => SetSize(node, Increment(GetSize(node)));
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         protected void DecrementSize(TElement node) => SetSize(node, Decrement(GetSize(node)));
57
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         protected TElement GetLeftSize(TElement node) => GetSizeOrZero(GetLeftOrDefault(node));
60
61         [MethodImpl(MethodImplOptions.AggressiveInlining)]
62         protected TElement GetRightSize(TElement node) => GetSizeOrZero(GetRightOrDefault(node));
63
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         protected TElement GetSizeOrZero(TElement node) => EqualToZero(node) ? Zero :
66             ↳ GetSize(node);
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         protected void FixSize(TElement node) => SetSize(node, Increment(Add(GetLeftSize(node),
70             ↳ GetRightSize(node))));
71
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         protected void LeftRotate(ref TElement root) => root = LeftRotate(root);
74
75         [MethodImpl(MethodImplOptions.AggressiveInlining)]
76         protected TElement LeftRotate(TElement root)
77         {
78             var right = GetRight(root);
79             #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
80                 if (EqualToZero(right))
81                 {
82                     throw new Exception("Right is null.");
83                 }
84             #endif
85             SetRight(root, GetLeft(right));
86         }
87     }
88 }

```

```

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     public virtual bool Contains(TElement node, TElement root)
109     {
110         while (!EqualToZero(root))
111         {
112             if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
113             {
114                 root = GetLeft(root);
115             }
116             else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
117             {
118                 root = GetRight(root);
119             }
120             else // node.Key == root.Key
121             {
122                 return true;
123             }
124         }
125         return false;
126     }
127
128     [MethodImpl(MethodImplOptions.AggressiveInlining)]
129     protected virtual void ClearNode(TElement node)
130     {
131         SetLeft(node, Zero);
132         SetRight(node, Zero);
133         SetSize(node, Zero);
134     }
135
136     [MethodImpl(MethodImplOptions.AggressiveInlining)]
137     public void Attach(ref TElement root, TElement node)
138     {
139     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
140         ValidateSizes(root);
141         Debug.WriteLine("--BeforeAttach--");
142         Debug.WriteLine(PrintNodes(root));
143         Debug.WriteLine("-----");
144         var sizeBefore = GetSize(root);
145     #endif
146         if (EqualToZero(root))
147         {
148             SetSize(node, One);
149             root = node;
150             return;
151         }
152         AttachCore(ref root, node);
153     #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(ref TElement root, TElement node);
167
168 [MethodImpl(MethodImplOptions.AggressiveInlining)]
169 public void Detach(ref TElement root, TElement node)
170 {
171     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
172         ValidateSizes(root);
173         Debug.WriteLine("---BeforeDetach---");
174         Debug.WriteLine(PrintNodes(root));
175         Debug.WriteLine("-----");
176         var sizeBefore = GetSize(root);
177         if (ValueEqualToZero(root))
178         {
179             throw new Exception($"Элемент с {node} не содержится в дереве.");
180         }
181     #endif
182     DetachCore(ref root, node);
183     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
184         Debug.WriteLine("---AfterDetach---");
185         Debug.WriteLine(PrintNodes(root));
186         Debug.WriteLine("-----");
187         ValidateSizes(root);
188         var sizeAfter = GetSize(root);
189         if (!IsEquals(MathHelpers.Decrement(sizeBefore), sizeAfter))
190         {
191             throw new Exception("Tree was broken after detach.");
192         }
193     #endif
194 }
195
196 protected abstract void DetachCore(ref TElement root, TElement node);
197
198 public void FixSizes(TElement node)
199 {
200     if (AreEqual(node, default))
201     {
202         return;
203     }
204     FixSizes(GetLeft(node));
205     FixSizes(GetRight(node));
206     FixSize(node);
207 }
208
209 public void ValidateSizes(TElement node)
210 {
211     if (AreEqual(node, default))
212     {
213         return;
214     }
215     var size = GetSize(node);
216     var leftSize = GetLeftSize(node);
217     var rightSize = GetRightSize(node);
218     var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
219     if (!AreEqual(size, expectedSize))
220     {
221         throw new InvalidOperationException($"Size of {node} is not valid. Expected
222             ↳ size: {expectedSize}, actual size: {size}.");
223     }
224     ValidateSizes(GetLeft(node));
225     ValidateSizes(GetRight(node));
226 }
227
228 public void ValidateSize(TElement node)
229 {
230     var size = GetSize(node);
231     var leftSize = GetLeftSize(node);
232     var rightSize = GetRightSize(node);
233     var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
234     if (!AreEqual(size, expectedSize))
235     {
236         throw new InvalidOperationException($"Size of {node} is not valid. Expected
237             ↳ size: {expectedSize}, actual size: {size}.");
238     }
239 }

```

```

236     }
237 }
238
239 public string PrintNodes(TElement node)
240 {
241     var sb = new StringBuilder();
242     PrintNodes(node, sb);
243     return sb.ToString();
244 }
245
246 [MethodImpl(MethodImplOptions.AggressiveInlining)]
247 public void PrintNodes(TElement node, StringBuilder sb) => PrintNodes(node, sb, 0);
248
249 public void PrintNodes(TElement node, StringBuilder sb, int level)
250 {
251     if (AreEqual(node, default))
252     {
253         return;
254     }
255     PrintNodes(GetLeft(node), sb, level + 1);
256     PrintNode(node, sb, level);
257     sb.AppendLine();
258     PrintNodes(GetRight(node), sb, level + 1);
259 }
260
261 public string PrintNode(TElement node)
262 {
263     var sb = new StringBuilder();
264     PrintNode(node, sb);
265     return sb.ToString();
266 }
267
268 [MethodImpl(MethodImplOptions.AggressiveInlining)]
269 protected void PrintNode(TElement node, StringBuilder sb) => PrintNode(node, sb, 0);
270
271 protected virtual void PrintNode(TElement node, StringBuilder sb, int level)
272 {
273     sb.Append('\t', level);
274     sb.Append(node);
275     PrintNodeValue(node, sb);
276     sb.Append(' ');
277     sb.Append('s');
278     sb.Append(GetSize(node));
279 }
280
281 protected abstract void PrintNodeValue(TElement node, StringBuilder sb);
282 }
283 }

```

./Platform.Collections.Methods.Tests/SizeBalancedTree2.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 SizeBalancedTree2<TElement> : SizeBalancedTreeMethods2<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 SizeBalancedTree2(int capacity) => (_elements, _allocated) = (new
29             ↪ TreeElement[capacity], Integer<TElement>.One);

```

```

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 FirstIsToTheLeftOfSecond(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 ref TElement GetLeftReference(TElement node) => ref
71     ↳ GetElement(node).Left;
72
73 protected override TElement GetLeft(TElement node) => GetElement(node).Left;
74
75 protected override ref TElement GetRightReference(TElement node) => ref
76     ↳ GetElement(node).Right;
77
78 protected override TElement GetRight(TElement node) => GetElement(node).Right;
79
80 protected override TElement GetSize(TElement node) => GetElement(node).Size;
81
82 protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
83     ↳ sb.Append(node);
84
85 protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
86     ↳ left;
87
88 protected override void SetRight(TElement node, TElement right) =>
89     ↳ GetElement(node).Right = right;
90
91 protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
92     ↳ size;
93
94 private ref TreeElement GetElement(TElement node) => ref
95     ↳ _elements[(Integer<TElement>)node];
96
97 }

```

./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         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 SizeBalancedTree(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) =>
    ↪ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
62
63     protected override bool FirstIsToLeftOfSecond(TElement first, TElement second) =>
    ↪ Comparer.Compare(first, second) < 0;
64
65     protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
    ↪ Comparer.Compare(first, second) > 0;
66
67     protected override ref TElement GetLeftReference(TElement node) => ref
    ↪ GetElement(node).Left;
68
69     protected override TElement GetLeft(TElement node) => GetElement(node).Left;
70
71     protected override ref TElement GetRightReference(TElement node) => ref
    ↪ GetElement(node).Right;
72
73     protected override TElement GetRight(TElement node) => GetElement(node).Right;
74
75     protected override TElement GetSize(TElement node) => GetElement(node).Size;
76
77     protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
    ↪ sb.Append(node);
78
79     protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
    ↪ left;
80

```

```

81     protected override void SetRight(TElement node, TElement right) =>
82         ↪ GetElement(node).Right = right;
83
84     protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
85         ↪ size;
86
87     private ref TreeElement GetElement(TElement node) => ref
88         ↪ _elements[(Integer<TElement>)node];
89 }
90 }

```

./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> :
10         ↪ SizedAndThreadedAVLBalancedTreeMethods<TElement>
11     {
12         private struct TreeElement
13         {
14             public TElement Size;
15             public TElement Left;
16             public TElement Right;
17             public sbyte Balance;
18             public bool LeftIsChild;
19             public bool RightIsChild;
20         }
21
22         private readonly TreeElement[] _elements;
23         private TElement _allocated;
24
25         public TElement Root;
26
27         public TElement Count => GetSizeOrZero(Root);
28
29         public SizedAndThreadedAVLBalancedTree(int capacity) => (_elements, _allocated) = (new
30             ↪ TreeElement[capacity], Integer<TElement>.One);
31
32         public TElement Allocate()
33         {
34             var newNode = _allocated;
35             if (IsEmpty(newNode))
36             {
37                 _allocated = Arithmetic.Increment(_allocated);
38                 return newNode;
39             }
40             else
41             {
42                 throw new InvalidOperationException("Allocated tree element is not empty.");
43             }
44         }
45
46         public void Free(TElement node)
47         {
48             while (!EqualityComparer.Equals(_allocated, Integer<TElement>.One) && IsEmpty(node))
49             {
50                 var lastNode = Arithmetic.Decrement(_allocated);
51                 if (EqualityComparer.Equals(lastNode, node))
52                 {
53                     _allocated = lastNode;
54                     node = Arithmetic.Decrement(node);
55                 }
56                 else
57                 {
58                     return;
59                 }
60             }
61         }
62
63         public bool IsEmpty(TElement node) =>
64             ↪ EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);
65
66         protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
67             ↪ Comparer.Compare(first, second) < 0;
68     }
69 }

```



```

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

```

./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
            ↳ SizedBinaryTreeMethodsBase<TElement> tree, Func<TElement> allocate, Action<TElement>
            ↳ free, ref TElement root, Func<TElement> treeCount, int maximumOperationsPerCycle)
12         {
13             for (var N = 1; N < maximumOperationsPerCycle; N++)
14             {
15                 var currentCount = 0;
16                 for (var i = 0; i < N; i++)
17                 {
18                     var node = allocate();
19                     tree.Attach(ref root, node);
20                     currentCount++;
21                     Assert.Equal(currentCount, (int)(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 SizeBalancedTreeMultipleAttachAndDetachTest()
11         {
12             var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
13             sizeBalancedTree.TestMultipleCreationsAndDeletions(sizeBalancedTree.Allocate,
    ↳ sizeBalancedTree.Free, ref sizeBalancedTree.Root, () => sizeBalancedTree.Count,
    ↳ _n);
14         }
15
16         [Fact]
17         public static void SizeBalancedTree2MultipleAttachAndDetachTest()
18         {
19             var sizeBalancedTree2 = new SizeBalancedTree2<uint>(10000);
20             sizeBalancedTree2.TestMultipleCreationsAndDeletions(sizeBalancedTree2.Allocate,
    ↳ sizeBalancedTree2.Free, ref sizeBalancedTree2.Root, () =>
    ↳ sizeBalancedTree2.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 SizeBalancedTreeMultipleRandomAttachAndDetachTest()

```

```

32     {
33         var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
34         sizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref sizeBalancedTree.Root,
35             ↪ () => sizeBalancedTree.Count, _n);
36     }
37
38     [Fact]
39     public static void SizeBalancedTree2MultipleRandomAttachAndDetachTest()
40     {
41         var sizeBalancedTree2 = new SizeBalancedTree2<uint>(10000);
42         sizeBalancedTree2.TestMultipleRandomCreationsAndDeletions(ref
43             ↪ sizeBalancedTree2.Root, () => sizeBalancedTree2.Count, _n);
44     }
45
46     [Fact]
47     public static void SizedAndThreadedAVLBalancedTreeMultipleRandomAttachAndDetachTest()
48     {
49         var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
50         avlTree.TestMultipleRandomCreationsAndDeletions(ref avlTree.Root, () =>
51             ↪ avlTree.Count, _n);
52     }
53 }

```

## Index

- ./Platform.Collections.Methods.Tests/SizeBalancedTree.cs, 22
- ./Platform.Collections.Methods.Tests/SizeBalancedTree2.cs, 21
- ./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/SizeBalancedTreeMethods.cs, 7
- ./Platform.Collections.Methods/Trees/SizeBalancedTreeMethods2.cs, 5
- ./Platform.Collections.Methods/Trees/SizedAndThreadedAVLBalancedTreeMethods.cs, 10
- ./Platform.Collections.Methods/Trees/SizedBinaryTreeMethodsBase.cs, 17