

LinksPlatform's Platform.Collections.Methods Class Library

./GenericCollectionMethodsBase.cs

```

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

```

```

62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected virtual TElement Add(TElement first, TElement second) =>
64         ↪ Arithmetic<TElement>.Add(first, second);
65
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     protected virtual TElement Subtract(TElement first, TElement second) =>
68         ↪ Arithmetic<TElement>.Subtract(first, second);
69 }

```

./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 (IsEquals(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 (IsEquals(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)

```

```

66     {
67         var elementPrevious = GetPrevious(element);
68         var elementNext = GetNext(element);
69         if (IsEquals(elementNext, element))
70         {
71             SetFirst(GetZero());
72             SetLast(GetZero());
73         }
74         else
75         {
76             SetNext(elementPrevious, elementNext);
77             SetPrevious(elementNext, elementPrevious);
78             if (IsEquals(element, GetFirst()))
79             {
80                 SetFirst(elementNext);
81             }
82             if (IsEquals(element, GetLast()))
83             {
84                 SetLast(elementPrevious);
85             }
86         }
87         SetPrevious(element, GetZero());
88         SetNext(element, GetZero());
89         DecrementSize();
90     }
91 }
92 }

```

./Lists/DoublyLinkedListMethodsBase.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Methods.Lists
6  {
7      /// <remarks>
8      /// Based on <a href="https://en.wikipedia.org/wiki/Doubly_linked_list">doubly linked
9      ↪ list</a> implementation.
10     /// </remarks>
11     public abstract class DoublyLinkedListMethodsBase<TElement> :
12     ↪ GenericCollectionMethodsBase<TElement>
13     {
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected abstract TElement GetFirst();
16
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 }

```

./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, GetZero());
49                 SetNext(element, GetZero());
50                 IncrementSize();
51             }
52             else
53             {
54                 AttachBefore(first, element);
55             }
56         }
57
58         public void AttachAsLast(TElement element)
59         {
60             var last = GetLast();
61             if (EqualToZero(last))
62             {
63                 AttachAsFirst(element);
64             }
65             else
66             {
67                 AttachAfter(last, element);
68             }
69         }
70
71         public void Detach(TElement element)
72         {
73             var elementPrevious = GetPrevious(element);
74             var elementNext = GetNext(element);
75             if (EqualToZero(elementPrevious))
76             {
```

```

77         SetFirst(elementNext);
78     }
79     else
80     {
81         SetNext(elementPrevious, elementNext);
82     }
83     if (EqualToZero(elementNext))
84     {
85         SetLast(elementPrevious);
86     }
87     else
88     {
89         SetPrevious(elementNext, elementPrevious);
90     }
91     SetPrevious(element, GetZero());
92     SetNext(element, GetZero());
93     DecrementSize();
94 }
95 }
96 }

```

./Trees/SizeBalancedTreeMethods2.cs

```

1  using System;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Methods.Trees
6  {
7      /// <summary>
8      /// Experimental implementation, don't use it yet.
9      /// </summary>
10     public unsafe abstract class SizeBalancedTreeMethods2<TElement> :
11         ↳ SizedBinaryTreeMethodsBase<TElement>
12     {
13         protected override void AttachCore(IntPtr root, TElement newNode)
14         {
15             if (ValueEqualToZero(root))
16             {
17                 System.Runtime.CompilerServices.Unsafe.Write((void*)root, newNode);
18                 IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root)
19                     ↳ );
20             }
21             else
22             {
23                 IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root)
24                     ↳ );
25                 if (FirstIsToTheLeftOfSecond(newNode,
26                     ↳ System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root)))
27                 {
28                     AttachCore(GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TEleme
29                         ↳ nt>((void*)root)),
30                         ↳ newNode);
31                     LeftMaintain(root);
32                 }
33                 else
34                 {
35                     AttachCore(GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TElem
36                         ↳ ent>((void*)root)),
37                         ↳ newNode);
38                     RightMaintain(root);
39                 }
40             }
41         }
42     }
43
44     protected override void DetachCore(IntPtr root, TElement nodeToDetach)
45     {
46         if (ValueEqualToZero(root))
47         {
48             return;
49         }
50         var currentNode = root;
51         var parent = IntPtr.Zero; /* Изначально зануление, так как родителя может и не быть
52             ↳ (Корень дерева). */
53         var replacementNode = GetZero();
54         while (!IsEquals(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)curren
55             ↳ tNode),
56             ↳ nodeToDetach))
57         {

```

```

46         SetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)currentNode
47         ↪ ),
48         ↪ Decrement(GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)
49         ↪ currentNode)))));
50         if (FirstIsToTheLeftOfSecond(nodeToDetach,
51         ↪ System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)currentNode)))
52         {
53             parent = currentNode;
54             currentNode = GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TEl
55             ↪ ement>((void*)currentNode));
56         }
57         else if (FirstIsToTheRightOfSecond(nodeToDetach,
58         ↪ System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)currentNode)))
59         {
60             parent = currentNode;
61             currentNode = GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TEl
62             ↪ ement>((void*)currentNode));
63         }
64         else
65         {
66             throw new InvalidOperationException("Duplicate link found in the tree.");
67         }
68     }
69     if (!ValueEqualToZero(GetLeftPointer(nodeToDetach)) &&
70     ↪ !ValueEqualToZero(GetRightPointer(nodeToDetach)))
71     {
72         var minNode = GetRightValue(nodeToDetach);
73         while (!EqualToZero(GetLeftValue(minNode)))
74         {
75             minNode = GetLeftValue(minNode); /* Передвигаемся до минимума */
76         }
77         DetachCore(GetRightPointer(nodeToDetach), minNode);
78         SetLeft(minNode, GetLeftValue(nodeToDetach));
79         if (!ValueEqualToZero(GetRightPointer(nodeToDetach)))
80         {
81             SetRight(minNode, GetRightValue(nodeToDetach));
82             SetSize(minNode, Increment(Add(GetSize(GetLeftValue(nodeToDetach)),
83             ↪ GetSize(GetRightValue(nodeToDetach))));
84         }
85         else
86         {
87             SetSize(minNode, Increment(GetSize(GetLeftValue(nodeToDetach))));
88         }
89         replacementNode = minNode;
90     }
91     else if (!ValueEqualToZero(GetLeftPointer(nodeToDetach)))
92     {
93         replacementNode = GetLeftValue(nodeToDetach);
94     }
95     else if (!ValueEqualToZero(GetRightPointer(nodeToDetach)))
96     {
97         replacementNode = GetRightValue(nodeToDetach);
98     }
99     if (parent == IntPtr.Zero)
100     {
101         System.Runtime.CompilerServices.Unsafe.Write((void*)root, replacementNode);
102     }
103     else if (IsEquals(GetLeftValue(System.Runtime.CompilerServices.Unsafe.Read<TElement>
104     ↪ ((void*)parent)),
105     ↪ nodeToDetach))
106     {
107         SetLeft(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)parent),
108         ↪ replacementNode);
109     }
110     else if (IsEquals(GetRightValue(System.Runtime.CompilerServices.Unsafe.Read<TElement>
111     ↪ >((void*)parent)),
112     ↪ nodeToDetach))
113     {
114         SetRight(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)parent),
115         ↪ replacementNode);
116     }
117     ClearNode(nodeToDetach);
118 }
119
120 private void LeftMaintain(IntPtr root)
121 {
122     if (!ValueEqualToZero(root))

```

```

108     {
109         var rootLeftNode = GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TEl
    ↪      ement>((void*)root));
110         if (!ValueEqualToZero(rootLeftNode))
111         {
112             var rootRightNode = GetRightPointer(System.Runtime.CompilerServices.Unsafe.R
    ↪      ead<TElement>((void*)root));
113             var rootLeftNodeLeftNode = GetLeftPointer(System.Runtime.CompilerServices.Un
    ↪      safe.Read<TElement>((void*)rootLeftNode));
114             if (!ValueEqualToZero(rootLeftNodeLeftNode) &&
115                 (ValueEqualToZero(rootRightNode) || GreaterThan(GetSize(System.Runtime.C
    ↪      ompilerServices.Unsafe.Read<TElement>((void*)rootLeftNodeLeftNode)),
    ↪      GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*
    ↪      )rootRightNode)))))
116             {
117                 RightRotate(root);
118             }
119             else
120             {
121                 var rootLeftNodeRightNode = GetRightPointer(System.Runtime.CompilerServi
    ↪      ces.Unsafe.Read<TElement>((void*)rootLeftNode));
122                 if (!ValueEqualToZero(rootLeftNodeRightNode) &&
123                     (ValueEqualToZero(rootRightNode) ||
    ↪      GreaterThan(GetSize(System.Runtime.CompilerServices.Unsafe.Read<
    ↪      TElement>((void*)rootLeftNodeRightNode)),
    ↪      GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((v
    ↪      oid*)rootRightNode)))))
124                 {
125                     LeftRotate(GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Rea
    ↪      d<TElement>((void*)root)));
126                     RightRotate(root);
127                 }
128                 else
129                 {
130                     return;
131                 }
132             }
133             LeftMaintain(GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TEle
    ↪      ment>((void*)root)));
134             RightMaintain(GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TE
    ↪      lement>((void*)root)));
135             LeftMaintain(root);
136             RightMaintain(root);
137         }
138     }
139 }

```

```

141 private void RightMaintain(IntPtr root)

```

```

142 {
143     if (!ValueEqualToZero(root))
144     {
145         var rootRightNode = GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<
    ↪      TElement>((void*)root));
146         if (!ValueEqualToZero(rootRightNode))
147         {
148             var rootLeftNode = GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Rea
    ↪      d<TElement>((void*)root));
149             var rootRightNodeRightNode = GetRightPointer(System.Runtime.CompilerServices
    ↪      .Unsafe.Read<TElement>((void*)rootRightNode));
150             if (!ValueEqualToZero(rootRightNodeRightNode) &&
151                 (ValueEqualToZero(rootLeftNode) ||
    ↪      GreaterThan(GetSize(System.Runtime.CompilerServices.Unsafe.Read<TEle
    ↪      ment>((void*)rootRightNodeRightNode)),
    ↪      GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*
    ↪      )rootLeftNode)))))
152             {
153                 LeftRotate(root);
154             }
155             else
156             {
157                 var rootRightNodeLeftNode = GetLeftPointer(System.Runtime.CompilerServic
    ↪      es.Unsafe.Read<TElement>((void*)rootRightNode));
158                 if (!ValueEqualToZero(rootRightNodeLeftNode) &&

```

```

159         (ValueEqualToZero(rootLeftNode) ||
        ↪      GreaterThan(GetSize(System.Runtime.CompilerServices.Unsafe.Read<
        ↪      TElement>((void*)rootRightNodeLeftNode)),
        ↪      GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((v
        ↪      oid*)rootLeftNode))))))
160     {
161         RightRotate(GetRightPointer(System.Runtime.CompilerServices.Unsafe.R
        ↪      ead<TElement>((void*)root)));
162         LeftRotate(root);
163     }
164     else
165     {
166         return;
167     }
168 }
169 LeftMaintain(GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<Tele
    ↪      ment>((void*)root)));
170 RightMaintain(GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TE
    ↪      lement>((void*)root)));
171 LeftMaintain(root);
172 RightMaintain(root);
173 }
174 }
175 }
176 }
177 }

```

./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 unsafe abstract class SizeBalancedTreeMethods<TElement> :
        ↪      SizedBinaryTreeMethodsBase<TElement>
8      {
9          protected override void AttachCore(IntPtr root, TElement node)
10         {
11             while (true)
12             {
13                 var left = GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((
        ↪      void*)root));
14                 var leftSize = GetSizeOrZero(System.Runtime.CompilerServices.Unsafe.Read<TElemen
        ↪      t>((void*)left));
15                 var right = GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement
        ↪      >((void*)root));
16                 var rightSize = GetSizeOrZero(System.Runtime.CompilerServices.Unsafe.Read<TEleme
        ↪      nt>((void*)right));
17                 if (FirstIsToTheLeftOfSecond(node,
        ↪      System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root))) //
        ↪      node.Key less than root.Key
18                 {
19                     if (EqualToZero(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*
        ↪      )left)))
20                     {
21                         IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((voi
        ↪      d*)root));
22                         SetSize(node, GetOne());
23                         System.Runtime.CompilerServices.Unsafe.Write((void*)left, node);
24                         break;
25                     }
26                     if (FirstIsToTheRightOfSecond(node,
        ↪      System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left))) //
        ↪      node.Key greater than left.Key
27                     {
28                         var leftRight = GetRightValue(System.Runtime.CompilerServices.Unsafe.Re
        ↪      d<TElement>((void*)left));
29                         var leftRightSize = GetSizeOrZero(leftRight);
30                         if (GreaterThan(Increment(leftRightSize), rightSize))
31                         {
32                             if (EqualToZero(leftRightSize) && EqualToZero(rightSize))
33                             {
34                                 SetLeft(node, System.Runtime.CompilerServices.Unsafe.Read<TEleme
        ↪      nt>((void*)left));
35                                 SetRight(node, System.Runtime.CompilerServices.Unsafe.Read<TElem
        ↪      ent>((void*)root));

```



```

36         SetSize(node, Add(GetSize(System.Runtime.CompilerServices.Unsafe
    ↪      .Read<TElement>((void*)left)), GetTwo())); // Two (2) -
    ↪      размер ветки *root (right) и самого node
37     SetLeft(System.Runtime.CompilerServices.Unsafe.Read<TElement>((v
    ↪      oid*)root),
    ↪      GetZero());
38     SetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((v
    ↪      oid*)root),
    ↪      GetOne());
39     System.Runtime.CompilerServices.Unsafe.Write((void*)root, node);
40     break;
41 }
42 LeftRotate(left);
43 RightRotate(root);
44 }
45 else
46 {
47     IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>(
    ↪      (void*)root));
48     root = left;
49 }
50 }
51 else // node.Key less than left.Key
52 {
53     var leftLeft = GetLeftValue(System.Runtime.CompilerServices.Unsafe.Read<
    ↪      TElement>((void*)left));
54     var leftLeftSize = GetSizeOrZero(leftLeft);
55     if (GreaterThan(Increment(leftLeftSize), rightSize))
56     {
57         RightRotate(root);
58     }
59     else
60     {
61         IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>(
    ↪      (void*)root));
62         root = left;
63     }
64 }
65 }
66 else // node.Key greater than root.Key
67 {
68     if (EqualToZero(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*
    ↪      )right)))
69     {
70         IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((voi
    ↪      d*)root));
71         SetSize(node, GetOne());
72         System.Runtime.CompilerServices.Unsafe.Write((void*)right, node);
73         break;
74     }
75     if (FirstIsToTheRightOfSecond(node,
    ↪      System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right))) //
    ↪      node.Key greater than right.Key
76     {
77         var rightRight = GetRightValue(System.Runtime.CompilerServices.Unsafe.Re
    ↪      ad<TElement>((void*)right));
78         var rightRightSize = GetSizeOrZero(rightRight);
79         if (GreaterThan(Increment(rightRightSize), leftSize))
80         {
81             LeftRotate(root);
82         }
83         else
84         {
85             IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>(
    ↪      (void*)root));
86             root = right;
87         }
88     }
89     else // node.Key less than right.Key
90     {
91         var rightLeft = GetLeftValue(System.Runtime.CompilerServices.Unsafe.Read
    ↪      <TElement>((void*)right));
92         var rightLeftSize = GetSizeOrZero(rightLeft);
93         if (GreaterThan(Increment(rightLeftSize), leftSize))
94         {
95             if (EqualToZero(rightLeftSize) && EqualToZero(leftSize))
96             {

```

```

97         SetLeft(node, System.Runtime.CompilerServices.Unsafe.Read<TEleme
    ↪ nt>((void*)root));
98         SetRight(node, System.Runtime.CompilerServices.Unsafe.Read<TElem
    ↪ ent>((void*)right));
99         SetSize(node, Add(GetSize(System.Runtime.CompilerServices.Unsafe
    ↪ .Read<TElement>((void*)right)), GetTwo())); // Two (2) -
    ↪ размер верки *root (left) и самого node
100         SetRight(System.Runtime.CompilerServices.Unsafe.Read<TElement>((
    ↪ void*)root),
    ↪ GetZero());
101         SetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((v
    ↪ oid*)root),
    ↪ GetOne());
102         System.Runtime.CompilerServices.Unsafe.Write((void*)root, node);
103         break;
104     }
105     RightRotate(right);
106     LeftRotate(root);
107 }
108 else
109 {
110     IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((
    ↪ void*)root));
111     root = right;
112 }
113 }
114 }
115 }
116 }
117 }
118 protected override void DetachCore(IntPtr root, TElement node)
119 {
120     while (true)
121     {
122         var left = GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement>((
    ↪ void*)root));
123         var leftSize = GetSizeOrZero(System.Runtime.CompilerServices.Unsafe.Read<TElemen
    ↪ t>((void*)left));
124         var right = GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TElement
    ↪ >((void*)root));
125         var rightSize = GetSizeOrZero(System.Runtime.CompilerServices.Unsafe.Read<TEleme
    ↪ nt>((void*)right));
126         if (FirstIsToTheLeftOfSecond(node,
    ↪ System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root))) //
    ↪ node.Key less than root.Key
127         {
128             EnsureNodeInTheTree(node, left);
129             var rightLeft = GetLeftValue(System.Runtime.CompilerServices.Unsafe.Read<TEL
    ↪ ement>((void*)right));
130             var rightLeftSize = GetSizeOrZero(rightLeft);
131             var rightRight = GetRightValue(System.Runtime.CompilerServices.Unsafe.Read<T
    ↪ Element>((void*)right));
132             var rightRightSize = GetSizeOrZero(rightRight);
133             if (GreaterThan(rightRightSize, Decrement(leftSize)))
134             {
135                 LeftRotate(root);
136             }
137             else if (GreaterThan(rightLeftSize, Decrement(leftSize)))
138             {
139                 RightRotate(right);
140                 LeftRotate(root);
141             }
142             else
143             {
144                 DecrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((voi
    ↪ d*)root));
145                 root = left;
146             }
147         }
148         else if (FirstIsToTheRightOfSecond(node,
    ↪ System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root))) //
    ↪ node.Key greater than root.Key
149         {
150             EnsureNodeInTheTree(node, right);
151             var leftLeft = GetLeftValue(System.Runtime.CompilerServices.Unsafe.Read<TEle
    ↪ ment>((void*)left));
152             var leftLeftSize = GetSizeOrZero(leftLeft);

```

```
var leftRight = GetRightValue(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left));
var leftRightSize = GetSizeOrZero(leftRight);
if (GreaterThan(leftLeftSize, Decrement(rightSize)))
{
    RightRotate(root);
}
else if (GreaterThan(leftRightSize, Decrement(rightSize)))
{
    LeftRotate(left);
    RightRotate(root);
}
else
{
    DecrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
    root = right;
}
}
else // key equals to root.Key
{
    if (GreaterThanZero(leftSize) && GreaterThanZero(rightSize))
    {
        if (GreaterThan(leftSize, rightSize))
        {
            var replacement = System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left);
            while (!EqualToZero(GetRightValue(replacement)))
            {
                replacement = GetRightValue(replacement);
            }
            DetachCore(left, replacement);
            SetLeft(replacement, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left));
            SetRight(replacement, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right));
            FixSize(replacement);
            System.Runtime.CompilerServices.Unsafe.Write((void*)root, replacement);
        }
        else
        {
            var replacement = System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right);
            while (!EqualToZero(GetLeftValue(replacement)))
            {
                replacement = GetLeftValue(replacement);
            }
            DetachCore(right, replacement);
            SetLeft(replacement, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left));
            SetRight(replacement, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right));
            FixSize(replacement);
            System.Runtime.CompilerServices.Unsafe.Write((void*)root, replacement);
        }
    }
    else if (GreaterThanZero(leftSize))
    {
        System.Runtime.CompilerServices.Unsafe.Write((void*)root, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left));
    }
    else if (GreaterThanZero(rightSize))
    {
        System.Runtime.CompilerServices.Unsafe.Write((void*)root, System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right));
    }
    else
    {
        System.Runtime.CompilerServices.Unsafe.Write((void*)root, GetZero());
    }
    ClearNode(node);
    break;
}
}
```

```

218
219     private void EnsureNodeInTheTree(TElement node, IntPtr branch)
220     {
221         if (EqualZero(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)branch)
222             ↪ ))
223         {
224             throw new InvalidOperationException($"Элемент {node} не содержится в дереве.");
225         }
226     }
227 }

```

./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 unsafe abstract class SizedAndThreadedAVLBalancedTreeMethods<TElement> :
22     ↪ SizedBinaryTreeMethodsBase<TElement>
23     {
24         // TODO: Link with size of TElement
25         private const int MaxPath = 92;
26
27         protected override void PrintNode(TElement node, StringBuilder sb, int level)
28         {
29             base.PrintNode(node, sb, level);
30             sb.Append(' ');
31             sb.Append(GetLeftIsChild(node) ? 'l' : 'L');
32             sb.Append(GetRightIsChild(node) ? 'r' : 'R');
33             sb.Append(' ');
34             sb.Append(GetBalance(node));
35         }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected void IncrementBalance(TElement node) => SetBalance(node,
39             ↪ (sbyte)(GetBalance(node) + 1));
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         protected void DecrementBalance(TElement node) => SetBalance(node,
43             ↪ (sbyte)(GetBalance(node) - 1));
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected override TElement GetLeftOrDefault(TElement node) => GetLeftIsChild(node) ?
47             ↪ base.GetLeftOrDefault(node) : default;
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         protected override TElement GetRightOrDefault(TElement node) => GetRightIsChild(node) ?
51             ↪ base.GetRightOrDefault(node) : default;
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         protected abstract bool GetLeftIsChild(TElement node);
55
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected abstract void SetLeftIsChild(TElement node, bool value);
58
59         [MethodImpl(MethodImplOptions.AggressiveInlining)]
60         protected abstract bool GetRightIsChild(TElement node);
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         protected abstract void SetRightIsChild(TElement node, bool value);
64
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         protected abstract sbyte GetBalance(TElement node);
67     }
68 }

```

```

60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 protected abstract void SetBalance(TElement node, sbyte value);
62
63
64 protected override void AttachCore(IntPtr root, TElement node)
65 {
66     unchecked
67     {
68         // TODO: Check what is faster to use simple array or array from array pool
69         // TODO: Try to use stackalloc as an optimization (requires code generation,
70         // ↳ because of generics)
71 #if USEARRAYPOOL
72         var path = ArrayPool.Allocate<TElement>(MaxPath);
73         var pathPosition = 0;
74         path[pathPosition++] = default;
75 #else
76         var path = new TElement[MaxPath];
77         var pathPosition = 1;
78 #endif
79         var rootPointer = (void*)root;
80         var currentNode =
81         ↳ System.Runtime.CompilerServices.Unsafe.Read<TElement>(rootPointer);
82         while (true)
83         {
84             if (FirstIsToTheLeftOfSecond(node, currentNode))
85             {
86                 if (GetLeftIsChild(currentNode))
87                 {
88                     IncrementSize(currentNode);
89                     path[pathPosition++] = currentNode;
90                     currentNode = GetLeftValue(currentNode);
91                 }
92                 else
93                 {
94                     // Threads
95                     SetLeft(node, GetLeftValue(currentNode));
96                     SetRight(node, currentNode);
97                     SetLeft(currentNode, node);
98                     SetLeftIsChild(currentNode, true);
99                     DecrementBalance(currentNode);
100                     SetSize(node, GetOne());
101                     FixSize(currentNode); // Should be incremented already
102                     break;
103                 }
104             }
105             else if (FirstIsToTheRightOfSecond(node, currentNode))
106             {
107                 if (GetRightIsChild(currentNode))
108                 {
109                     IncrementSize(currentNode);
110                     path[pathPosition++] = currentNode;
111                     currentNode = GetRightValue(currentNode);
112                 }
113                 else
114                 {
115                     // Threads
116                     SetRight(node, GetRightValue(currentNode));
117                     SetLeft(node, currentNode);
118                     SetRight(currentNode, node);
119                     SetRightIsChild(currentNode, true);
120                     IncrementBalance(currentNode);
121                     SetSize(node, GetOne());
122                     FixSize(currentNode); // Should be incremented already
123                     break;
124                 }
125             }
126             else
127             {
128                 throw new InvalidOperationException("Node with the same key already
129                 ↳ attached to a tree.");
130             }
131         }
132         // Restore balance. This is the goodness of a non-recursive
133         // implementation, when we are done with balancing we 'break'
134         // the loop and we are done.
135         while (true)
136         {
137             var parent = path[--pathPosition];

```

```

135     var isLeftNode = !IsEquals(parent, default) && IsEquals(currentNode,
136         ↪ GetLeftValue(parent));
137     var currentNodeBalance = GetBalance(currentNode);
138     if (currentNodeBalance < -1 || currentNodeBalance > 1)
139     {
140         currentNode = Balance(currentNode);
141         if (IsEquals(parent, default))
142         {
143             System.Runtime.CompilerServices.Unsafe.Write((void*)root,
144                 ↪ currentNode);
145         }
146         else if (isLeftNode)
147         {
148             SetLeft(parent, currentNode);
149             FixSize(parent);
150         }
151         else
152         {
153             SetRight(parent, currentNode);
154             FixSize(parent);
155         }
156     }
157     currentNodeBalance = GetBalance(currentNode);
158     if (currentNodeBalance == 0 || IsEquals(parent, default))
159     {
160         break;
161     }
162     if (isLeftNode)
163     {
164         DecrementBalance(parent);
165     }
166     else
167     {
168         IncrementBalance(parent);
169     }
170     currentNode = parent;
171 }
172 #if USEARRAYPOOL
173     ArrayPool.Free(path);
174 #endif
175 }
176
177 private TElement Balance(TElement node)
178 {
179     unchecked
180     {
181         var rootBalance = GetBalance(node);
182         if (rootBalance < -1)
183         {
184             var left = GetLeftValue(node);
185             if (GetBalance(left) > 0)
186             {
187                 SetLeft(node, LeftRotateWithBalance(left));
188                 FixSize(node);
189             }
190             node = RightRotateWithBalance(node);
191         }
192         else if (rootBalance > 1)
193         {
194             var right = GetRightValue(node);
195             if (GetBalance(right) < 0)
196             {
197                 SetRight(node, RightRotateWithBalance(right));
198                 FixSize(node);
199             }
200             node = LeftRotateWithBalance(node);
201         }
202         return node;
203     }
204 }
205
206 protected TElement LeftRotateWithBalance(TElement node)
207 {
208     unchecked
209     {
210         var right = GetRightValue(node);
211         if (GetLeftIsChild(right))

```

```

211     {
212         SetRight(node, GetLeftValue(right));
213     }
214     else
215     {
216         SetRightIsChild(node, false);
217         SetLeftIsChild(right, true);
218     }
219     SetLeft(right, node);
220     // Fix size
221     SetSize(right, GetSize(node));
222     FixSize(node);
223     // Fix balance
224     var rootBalance = GetBalance(node);
225     var rightBalance = GetBalance(right);
226     if (rightBalance <= 0)
227     {
228         if (rootBalance >= 1)
229         {
230             SetBalance(right, (sbyte)(rightBalance - 1));
231         }
232         else
233         {
234             SetBalance(right, (sbyte)(rootBalance + rightBalance - 2));
235         }
236         SetBalance(node, (sbyte)(rootBalance - 1));
237     }
238     else
239     {
240         if (rootBalance <= rightBalance)
241         {
242             SetBalance(right, (sbyte)(rootBalance - 2));
243         }
244         else
245         {
246             SetBalance(right, (sbyte)(rightBalance - 1));
247         }
248         SetBalance(node, (sbyte)(rootBalance - rightBalance - 1));
249     }
250     return right;
251 }
252
253
254 protected TElement RightRotateWithBalance(TElement node)
255 {
256     unchecked
257     {
258         var left = GetLeftValue(node);
259         if (GetRightIsChild(left))
260         {
261             SetLeft(node, GetRightValue(left));
262         }
263         else
264         {
265             SetLeftIsChild(node, false);
266             SetRightIsChild(left, true);
267         }
268         SetRight(left, node);
269         // Fix size
270         SetSize(left, GetSize(node));
271         FixSize(node);
272         // Fix balance
273         var rootBalance = GetBalance(node);
274         var leftBalance = GetBalance(left);
275         if (leftBalance <= 0)
276         {
277             if (leftBalance > rootBalance)
278             {
279                 SetBalance(left, (sbyte)(leftBalance + 1));
280             }
281             else
282             {
283                 SetBalance(left, (sbyte)(rootBalance + 2));
284             }
285             SetBalance(node, (sbyte)(rootBalance - leftBalance + 1));
286         }
287         else
288         {

```

```

289         if (rootBalance <= -1)
290         {
291             SetBalance(left, (sbyte)(leftBalance + 1));
292         }
293         else
294         {
295             SetBalance(left, (sbyte)(rootBalance + leftBalance + 2));
296         }
297         SetBalance(node, (sbyte)(rootBalance + 1));
298     }
299     return left;
300 }
301 }
302
303 protected TElement GetNext(TElement node)
304 {
305     unchecked
306     {
307         var current = GetRightValue(node);
308         if (GetRightIsChild(node))
309         {
310             while (GetLeftIsChild(current))
311             {
312                 current = GetLeftValue(current);
313             }
314         }
315         return current;
316     }
317 }
318
319 protected TElement GetPrevious(TElement node)
320 {
321     unchecked
322     {
323         var current = GetLeftValue(node);
324         if (GetLeftIsChild(node))
325         {
326             while (GetRightIsChild(current))
327             {
328                 current = GetRightValue(current);
329             }
330         }
331         return current;
332     }
333 }
334
335 protected override void DetachCore(IntPtr root, TElement node)
336 {
337     unchecked
338     {
339 #if USEARRAYPOOL
340         var path = ArrayPool.Allocate<TElement>(MaxPath);
341         var pathPosition = 0;
342         path[pathPosition++] = default;
343 #else
344         var path = new TElement[MaxPath];
345         var pathPosition = 1;
346 #endif
347         var rootPointer = (void*)root;
348         var currentNode =
349             ↪ System.Runtime.CompilerServices.Unsafe.Read<TElement>(rootPointer);
350         while (true)
351         {
352             if (FirstIsToTheLeftOfSecond(node, currentNode))
353             {
354                 if (!GetLeftIsChild(currentNode))
355                 {
356                     throw new InvalidOperationException("Cannot find a node.");
357                 }
358                 DecrementSize(currentNode);
359                 path[pathPosition++] = currentNode;
360                 currentNode = GetLeftValue(currentNode);
361             }
362             else if (FirstIsToTheRightOfSecond(node, currentNode))
363             {
364                 if (!GetRightIsChild(currentNode))
365                 {
366                     throw new InvalidOperationException("Cannot find a node.");
367                 }
368             }
369         }
370     }
371 }

```



```

367         DecrementSize(currentNode);
368         path[pathPosition++] = currentNode;
369         currentNode = GetRightValue(currentNode);
370     }
371     else
372     {
373         break;
374     }
375 }
376 var parent = path[--pathPosition];
377 var balanceNode = parent;
378 var isLeftNode = !IsEquals(parent, default) && IsEquals(currentNode,
    ↪ GetLeftValue(parent));
379 if (!GetLeftIsChild(currentNode))
380 {
381     if (!GetRightIsChild(currentNode)) // node has no children
382     {
383         if (IsEquals(parent, default))
384         {
385             System.Runtime.CompilerServices.Unsafe.Write(rootPointer, GetZero());
386         }
387         else if (isLeftNode)
388         {
389             SetLeftIsChild(parent, false);
390             SetLeft(parent, GetLeftValue(currentNode));
391             IncrementBalance(parent);
392         }
393         else
394         {
395             SetRightIsChild(parent, false);
396             SetRight(parent, GetRightValue(currentNode));
397             DecrementBalance(parent);
398         }
399     }
400     else // node has a right child
401     {
402         var successor = GetNext(currentNode);
403         SetLeft(successor, GetLeftValue(currentNode));
404         var right = GetRightValue(currentNode);
405         if (IsEquals(parent, default))
406         {
407             System.Runtime.CompilerServices.Unsafe.Write(rootPointer, right);
408         }
409         else if (isLeftNode)
410         {
411             SetLeft(parent, right);
412             IncrementBalance(parent);
413         }
414         else
415         {
416             SetRight(parent, right);
417             DecrementBalance(parent);
418         }
419     }
420 }
421 else // node has a left child
422 {
423     if (!GetRightIsChild(currentNode))
424     {
425         var predecessor = GetPrevious(currentNode);
426         SetRight(predecessor, GetRightValue(currentNode));
427         var leftValue = GetLeftValue(currentNode);
428         if (IsEquals(parent, default))
429         {
430             System.Runtime.CompilerServices.Unsafe.Write(rootPointer, leftValue);
431         }
432         else if (isLeftNode)
433         {
434             SetLeft(parent, leftValue);
435             IncrementBalance(parent);
436         }
437         else
438         {
439             SetRight(parent, leftValue);
440             DecrementBalance(parent);
441         }
442     }
443     else // node has a both children (left and right)

```

```

444 {
445     var predecessor = GetLeftValue(currentNode);
446     var successor = GetRightValue(currentNode);
447     var successorParent = currentNode;
448     int previousPathPosition = ++pathPosition;
449     // find the immediately next node (and its parent)
450     while (GetLeftIsChild(successor))
451     {
452         path[++pathPosition] = successorParent = successor;
453         successor = GetLeftValue(successor);
454         if (!IsEquals(successorParent, currentNode))
455         {
456             DecrementSize(successorParent);
457         }
458     }
459     path[previousPathPosition] = successor;
460     balanceNode = path[pathPosition];
461     // remove 'successor' from the tree
462     if (!IsEquals(successorParent, currentNode))
463     {
464         if (!GetRightIsChild(successor))
465         {
466             SetLeftIsChild(successorParent, false);
467         }
468         else
469         {
470             SetLeft(successorParent, GetRightValue(successor));
471         }
472         IncrementBalance(successorParent);
473         SetRightIsChild(successor, true);
474         SetRight(successor, GetRightValue(currentNode));
475     }
476     else
477     {
478         DecrementBalance(currentNode);
479     }
480     // set the predecessor's successor link to point to the right place
481     while (GetRightIsChild(predecessor))
482     {
483         predecessor = GetRightValue(predecessor);
484     }
485     SetRight(predecessor, successor);
486     // prepare 'successor' to replace 'node'
487     var left = GetLeftValue(currentNode);
488     SetLeftIsChild(successor, true);
489     SetLeft(successor, left);
490     SetBalance(successor, GetBalance(currentNode));
491     FixSize(successor);
492     if (IsEquals(parent, default))
493     {
494         System.Runtime.CompilerServices.Unsafe.Write(rootPointer, successor);
495     }
496     else if (isLeftNode)
497     {
498         SetLeft(parent, successor);
499     }
500     else
501     {
502         SetRight(parent, successor);
503     }
504 }
505 }
506 // restore balance
507 if (!IsEquals(balanceNode, default))
508 {
509     while (true)
510     {
511         var balanceParent = path[--pathPosition];
512         isLeftNode = !IsEquals(balanceParent, default) && IsEquals(balanceNode,
513             ↪ GetLeftValue(balanceParent));
514         var currentNodeBalance = GetBalance(balanceNode);
515         if (currentNodeBalance < -1 || currentNodeBalance > 1)
516         {
517             balanceNode = Balance(balanceNode);
518             if (IsEquals(balanceParent, default))
519             {
520                 System.Runtime.CompilerServices.Unsafe.Write(rootPointer,
521                     ↪ balanceNode);

```

```

520     }
521     else if (isLeftNode)
522     {
523         SetLeft(balanceParent, balanceNode);
524     }
525     else
526     {
527         SetRight(balanceParent, balanceNode);
528     }
529 }
530 currentNodeBalance = GetBalance(balanceNode);
531 if (currentNodeBalance != 0 || IsEquals(balanceParent, default))
532 {
533     break;
534 }
535 if (isLeftNode)
536 {
537     IncrementBalance(balanceParent);
538 }
539 else
540 {
541     DecrementBalance(balanceParent);
542 }
543 balanceNode = balanceParent;
544 }
545 }
546 ClearNode(node);
547 #if USEARRAYPOOL
548     ArrayPool.Free(path);
549 #endif
550 }
551 }
552
553 [MethodImpl(MethodImplOptions.AggressiveInlining)]
554 protected override void ClearNode(TElement node)
555 {
556     SetLeft(node, GetZero());
557     SetRight(node, GetZero());
558     SetSize(node, GetZero());
559     SetLeftIsChild(node, false);
560     SetRightIsChild(node, false);
561     SetBalance(node, 0);
562 }
563 }
564 }

```

./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 unsafe abstract class SizedBinaryTreeMethodsBase<TElement> :
12         ↳ GenericCollectionMethodsBase<TElement>
13     {
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected abstract IntPtr GetLeftPointer(TElement node);
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected abstract IntPtr GetRightPointer(TElement node);
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected abstract TElement GetLeftValue(TElement node);
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         protected abstract TElement GetRightValue(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)]

```

```

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) => GetLeftPointer(node) !=
45         ↪ IntPtr.Zero ? GetLeftValue(node) : default;
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected virtual TElement GetRightOrDefault(TElement node) => GetRightPointer(node) !=
49         ↪ IntPtr.Zero ? GetRightValue(node) : default;
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected void IncrementSize(TElement node) => SetSize(node, Increment(GetSize(node)));
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected void DecrementSize(TElement node) => SetSize(node, Decrement(GetSize(node)));
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     protected TElement GetLeftSize(TElement node) => GetSizeOrZero(GetLeftOrDefault(node));
59
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     protected TElement GetRightSize(TElement node) => GetSizeOrZero(GetRightOrDefault(node));
62
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     protected TElement GetSizeOrZero(TElement node) => EqualToZero(node) ? GetZero() :
65         ↪ GetSize(node);
66
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     protected void FixSize(TElement node) => SetSize(node, Increment(Add(GetLeftSize(node),
69         ↪ GetRightSize(node))));
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     protected void LeftRotate(IntPtr root)
73     {
74         var rootPointer = (void*)root;
75         System.Runtime.CompilerServices.Unsafe.Write(rootPointer,
76             ↪ LeftRotate(System.Runtime.CompilerServices.Unsafe.Read<TElement>(rootPointer)));
77     }
78
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     protected TElement LeftRotate(TElement root)
81     {
82         var right = GetRightValue(root);
83         #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
84             if (EqualToZero(right))
85             {
86                 throw new Exception("Right is null.");
87             }
88         #endif
89
90         SetRight(root, GetLeftValue(right));
91         SetLeft(right, root);
92         SetSize(right, GetSize(root));
93         FixSize(root);
94         return right;
95     }
96
97     [MethodImpl(MethodImplOptions.AggressiveInlining)]
98     protected void RightRotate(IntPtr root)
99     {
100         var rootPointer = (void*)root;
101         System.Runtime.CompilerServices.Unsafe.Write(rootPointer,
102             ↪ RightRotate(System.Runtime.CompilerServices.Unsafe.Read<TElement>(rootPointer)));
103     }
104
105     [MethodImpl(MethodImplOptions.AggressiveInlining)]
106     protected TElement RightRotate(TElement root)
107     {
108         var left = GetLeftValue(root);
109         #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
110             if (EqualToZero(left))
111             {

```

```

105         throw new Exception("Left is null.");
106     }
107 #endif
108     SetLeft(root, GetRightValue(left));
109     SetRight(left, root);
110     SetSize(left, GetSize(root));
111     FixSize(root);
112     return left;
113 }
114
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 public bool Contains(TElement node, TElement root)
117 {
118     while (!EqualToZero(root))
119     {
120         if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
121         {
122             root = GetLeftOrDefault(root);
123         }
124         else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
125         {
126             root = GetRightOrDefault(root);
127         }
128         else // node.Key == root.Key
129         {
130             return true;
131         }
132     }
133     return false;
134 }
135
136 [MethodImpl(MethodImplOptions.AggressiveInlining)]
137 protected virtual void ClearNode(TElement node)
138 {
139     SetLeft(node, GetZero());
140     SetRight(node, GetZero());
141     SetSize(node, GetZero());
142 }
143
144 public void Attach(IntPtr root, TElement node)
145 {
146 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
147     ValidateSizes(root);
148     Debug.WriteLine("--BeforeAttach--");
149     Debug.WriteLine(PrintNodes(root));
150     Debug.WriteLine("-----");
151     var sizeBefore = GetSize(root);
152 #endif
153     if (ValueEqualToZero(root))
154     {
155         SetSize(node, GetOne());
156         System.Runtime.CompilerServices.Unsafe.Write((void*)root, node);
157         return;
158     }
159     AttachCore(root, node);
160 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
161     Debug.WriteLine("--AfterAttach--");
162     Debug.WriteLine(PrintNodes(root));
163     Debug.WriteLine("-----");
164     ValidateSizes(root);
165     var sizeAfter = GetSize(root);
166     if (!IsEquals(MathHelpers.Increment(sizeBefore), sizeAfter))
167     {
168         throw new Exception("Tree was broken after attach.");
169     }
170 #endif
171 }
172
173 protected abstract void AttachCore(IntPtr root, TElement node);
174
175 public void Detach(IntPtr root, TElement node)
176 {
177 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
178     ValidateSizes(root);
179     Debug.WriteLine("--BeforeDetach--");
180     Debug.WriteLine(PrintNodes(root));
181     Debug.WriteLine("-----");
182     var sizeBefore = GetSize(root);
183     if (ValueEqualToZero(root))

```

```

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

```

```

259     {
260         throw new InvalidOperationException($"Size of {node} is not valid. Expected
261         ↪ size: {expectedSize}, actual size: {size}.");
262     }
263 }
264 public string PrintNodes(IntPtr root)
265 {
266     if (root != IntPtr.Zero)
267     {
268         var sb = new StringBuilder();
269         PrintNodes(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root),
270         ↪ sb);
271         return sb.ToString();
272     }
273     return "";
274 }
275 public string PrintNodes(TElement node)
276 {
277     var sb = new StringBuilder();
278     PrintNodes(node, sb);
279     return sb.ToString();
280 }
281 public void PrintNodes(TElement node, StringBuilder sb) => PrintNodes(node, sb, 0);
282
283 public void PrintNodes(TElement node, StringBuilder sb, int level)
284 {
285     if (IsEquals(node, default))
286     {
287         return;
288     }
289     PrintNodes(GetLeftOrDefault(node), sb, level + 1);
290     PrintNode(node, sb, level);
291     sb.AppendLine();
292     PrintNodes(GetRightOrDefault(node), sb, level + 1);
293 }
294
295 public string PrintNode(TElement node)
296 {
297     var sb = new StringBuilder();
298     PrintNode(node, sb);
299     return sb.ToString();
300 }
301
302 protected void PrintNode(TElement node, StringBuilder sb) => PrintNode(node, sb, 0);
303
304 protected virtual void PrintNode(TElement node, StringBuilder sb, int level)
305 {
306     sb.Append('\t', level);
307     sb.Append(node);
308     PrintNodeValue(node, sb);
309     sb.Append(' ');
310     sb.Append('s');
311     sb.Append(GetSize(node));
312 }
313
314 protected abstract void PrintNodeValue(TElement node, StringBuilder sb);
315 }
316 }
317

```

Index

- ./GenericCollectionMethodsBase.cs, 1
- ./Lists/CircularDoublyLinkedListMethods.cs, 2
- ./Lists/DoublyLinkedListMethodsBase.cs, 3
- ./Lists/OpenDoublyLinkedListMethods.cs, 4
- ./Trees/SizeBalancedTreeMethods.cs, 8
- ./Trees/SizeBalancedTreeMethods2.cs, 5
- ./Trees/SizedAndThreadedAVLBalancedTreeMethods.cs, 12
- ./Trees/SizedBinaryTreeMethodsBase.cs, 19