

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

```

#### ./Lists/OpenDoublyLinkedListMethods.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Methods.Lists
4  {
5      public abstract class OpenDoublyLinkedListMethods<TElement> :
6          DoublyLinkedListMethodsBase<TElement>
7      {
8          public void AttachBefore(TElement baseElement, TElement newElement)
9          {
10              // Implementation
11          }
12      }
13 }

```

```

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         protected override void DetachCore(IntPtr root, TElement nodeToDetach)
44         {
45             if (ValueEqualToZero(root))
46             {
47                 return;
48             }
49             var currentNode = root;
50             var parent = IntPtr.Zero; /* Изначально зануление, так как родителя может и не быть
51                 ↳ (Корень дерева). */
52             var replacementNode = GetZero();
53             while (!IsEquals(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)curren
54                 ↳ tNode),
55                 ↳ nodeToDetach))
56             {
57                 SetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)currentNode
58                     ↳ ),
59                     ↳ Decrement(GetSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((voi
60                         ↳ d*)currentNode))));
61                 if (FirstIsToTheLeftOfSecond(nodeToDetach,
62                     ↳ System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)currentNode)))
63                 {
64                     parent = currentNode;

```

```

50         currentNode = GetLeftPointer(System.Runtime.CompilerServices.Unsafe.Read<TEl_
    ↪      ement>((void*)currentNode));
51     }
52     else if (FirstIsToTheRightOfSecond(nodeToDetach,
    ↪      System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)currentNode)))
53     {
54         parent = currentNode;
55         currentNode = GetRightPointer(System.Runtime.CompilerServices.Unsafe.Read<TEl_
    ↪      ement>((void*)currentNode));
56     }
57     else
58     {
59         throw new InvalidOperationException("Duplicate link found in the tree.");
60     }
61 }
62 if (!ValueEqualToZero(GetLeftPointer(nodeToDetach)) &&
    ↪      !ValueEqualToZero(GetRightPointer(nodeToDetach)))
63 {
64     var minNode = GetRightValue(nodeToDetach);
65     while (!EqualToZero(GetLeftValue(minNode)))
66     {
67         minNode = GetLeftValue(minNode); /* Передвигаемся до минимума */
68     }
69     DetachCore(GetRightPointer(nodeToDetach), minNode);
70     SetLeft(minNode, GetLeftValue(nodeToDetach));
71     if (!ValueEqualToZero(GetRightPointer(nodeToDetach)))
72     {
73         SetRight(minNode, GetRightValue(nodeToDetach));
74         SetSize(minNode, Increment(Add(GetSize(GetLeftValue(nodeToDetach)),
    ↪      GetSize(GetRightValue(nodeToDetach))));
75     }
76     else
77     {
78         SetSize(minNode, Increment(GetSize(GetLeftValue(nodeToDetach))));
79     }
80     replacementNode = minNode;
81 }
82 else if (!ValueEqualToZero(GetLeftPointer(nodeToDetach)))
83 {
84     replacementNode = GetLeftValue(nodeToDetach);
85 }
86 else if (!ValueEqualToZero(GetRightPointer(nodeToDetach)))
87 {
88     replacementNode = GetRightValue(nodeToDetach);
89 }
90 if (parent == IntPtr.Zero)
91 {
92     System.Runtime.CompilerServices.Unsafe.Write((void*)root, replacementNode);
93 }
94 else if (IsEquals(GetLeftValue(System.Runtime.CompilerServices.Unsafe.Read<TElement>_
    ↪      ((void*)parent)),
    ↪      nodeToDetach))
95 {
96     SetLeft(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)parent),
    ↪      replacementNode);
97 }
98 else if (IsEquals(GetRightValue(System.Runtime.CompilerServices.Unsafe.Read<TElement>_
    ↪      >((void*)parent)),
    ↪      nodeToDetach))
99 {
100     SetRight(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)parent),
    ↪      replacementNode);
101 }
102 ClearNode(nodeToDetach);
103 }
104
105 private void LeftMaintain(IntPtr root)
106 {
107     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 }
140
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 верки *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>(&
48         ↪ (void*)root));
49     root = left;
50 }
51 else // node.Key less than left.Key
52 {
53     var leftLeft = GetLeftValue(System.Runtime.CompilerServices.Unsafe.Read<
54         ↪ TElement>((void*)left));
55     var leftLeftSize = GetSizeOrZero(leftLeft);
56     if (GreaterThan(Increment(leftLeftSize), rightSize))
57     {
58         RightRotate(root);
59     }
60     else
61     {
62         IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>(&
63             ↪ (void*)root));
64         root = left;
65     }
66 }
67 else // node.Key greater than root.Key
68 {
69     if (EqualToZero(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)
70         ↪ )right)))
71     {
72         IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((voi
73             ↪ d*)root));
74         SetSize(node, GetOne());
75         System.Runtime.CompilerServices.Unsafe.Write((void*)right, node);
76         break;
77     }
78     if (FirstIsToTheRightOfSecond(node,
79         ↪ System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right))) //
80         ↪ node.Key greater than right.Key
81     {
82         var rightRight = GetRightValue(System.Runtime.CompilerServices.Unsafe.Re
83             ↪ ad<TElement>((void*)right));
84         var rightRightSize = GetSizeOrZero(rightRight);
85         if (GreaterThan(Increment(rightRightSize), leftSize))
86         {
87             LeftRotate(root);
88         }
89         else
90         {
91             IncrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>(&
92                 ↪ (void*)root));
93             root = right;
94         }
95     }
96     else // node.Key less than right.Key
97     {
98         var rightLeft = GetLeftValue(System.Runtime.CompilerServices.Unsafe.Read
99             ↪ <TElement>((void*)right));
100        var rightLeftSize = GetSizeOrZero(rightLeft);
101        if (GreaterThan(Increment(rightLeftSize), leftSize))
102        {
103            if (EqualToZero(rightLeftSize) && EqualToZero(leftSize))
104            {
105                SetLeft(node, System.Runtime.CompilerServices.Unsafe.Read<Teleme
106                    ↪ nt>((void*)root));
107                SetRight(node, System.Runtime.CompilerServices.Unsafe.Read<TElem
108                    ↪ ent>((void*)right));
109                SetSize(node, Add(GetSize(System.Runtime.CompilerServices.Unsafe
110                    ↪ .Read<TElement>((void*)right)), GetTwo())); // Two (2) -
111                    ↪ размер ветки *root (left) и самого node
112                SetRight(System.Runtime.CompilerServices.Unsafe.Read<TElement>((
113                    ↪ void*)root),
114                    ↪ 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);
153             var leftRight = GetRightValue(System.Runtime.CompilerServices.Unsafe.Read<TE
            ↪ lement>((void*)left));
154             var leftRightSize = GetSizeOrZero(leftRight);
155             if (GreaterThan(leftLeftSize, Decrement(rightSize)))
156             {
157                 RightRotate(root);
158             }
159             else if (GreaterThan(leftRightSize, Decrement(rightSize)))
160             {
161                 LeftRotate(left);

```

```

162         RightRotate(root);
163     }
164     else
165     {
166         DecrementSize(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)
            ↳ d*)root));
            root = right;
167     }
168 }
169
170 else // key equals to root.Key
171 {
172     if (GreaterThanZero(leftSize) && GreaterThanZero(rightSize))
173     {
174         if (GreaterThan(leftSize, rightSize))
175         {
176             var replacement = System.Runtime.CompilerServices.Unsafe.Read<TEleme
                ↳ nt>((void*)left);
                while (!EqualToZero(GetRightValue(replacement)))
177             {
178                 replacement = GetRightValue(replacement);
179             }
180             DetachCore(left, replacement);
181             SetLeft(replacement, System.Runtime.CompilerServices.Unsafe.Read<TEL
                ↳ ement>((void*)left));
182             SetRight(replacement, System.Runtime.CompilerServices.Unsafe.Read<TE
                ↳ lement>((void*)right));
183             FixSize(replacement);
184             System.Runtime.CompilerServices.Unsafe.Write((void*)root,
                ↳ replacement);
185         }
186         else
187         {
188             var replacement = System.Runtime.CompilerServices.Unsafe.Read<TEleme
                ↳ nt>((void*)right);
                while (!EqualToZero(GetLeftValue(replacement)))
189             {
190                 replacement = GetLeftValue(replacement);
191             }
192             DetachCore(right, replacement);
193             SetLeft(replacement, System.Runtime.CompilerServices.Unsafe.Read<TEL
                ↳ ement>((void*)left));
194             SetRight(replacement, System.Runtime.CompilerServices.Unsafe.Read<TE
                ↳ lement>((void*)right));
195             FixSize(replacement);
196             System.Runtime.CompilerServices.Unsafe.Write((void*)root,
                ↳ replacement);
197         }
198     }
199 }
200
201 else if (GreaterThanZero(leftSize))
202 {
203     System.Runtime.CompilerServices.Unsafe.Write((void*)root,
        ↳ System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)left));
204 }
205 else if (GreaterThanZero(rightSize))
206 {
207     System.Runtime.CompilerServices.Unsafe.Write((void*)root,
        ↳ System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)right));
208 }
209 else
210 {
211     System.Runtime.CompilerServices.Unsafe.Write((void*)root, GetZero());
212 }
213 ClearNode(node);
214 break;
215 }
216 }
217 }
218
219 private void EnsureNodeInTheTree(TElement node, IntPtr branch)
220 {
221     if (EqualToZero(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)branch)
        ↳ ))
222     {
223         throw new InvalidOperationException($"Элемент {node} не содержится в дереве.");
224     }
225 }
226

```

# ./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/programmatom/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         protected abstract bool GetLeftIsChild(TElement node);
54         protected abstract void SetLeftIsChild(TElement node, bool value);
55         protected abstract bool GetRightIsChild(TElement node);
56         protected abstract void SetRightIsChild(TElement node, bool value);
57         protected abstract sbyte GetBalance(TElement node);
58         protected abstract void SetBalance(TElement node, sbyte value);
59
60         protected override void AttachCore(IntPtr root, TElement node)
61         {
62             unchecked
63             {
64                 // TODO: Check what is faster to use simple array or array from array pool
65                 // TODO: Try to use stackalloc as an optimization (requires code generation,
66                 ↪ because of generics)
67             }
68         }
69
70 #if USEARRAYPOOL
71         var path = ArrayPool.Allocate<TElement>(MaxPath);
72         var pathPosition = 0;
73         path[pathPosition++] = default;
74 #else
75         var path = new TElement[MaxPath];
76         var pathPosition = 1;
77 #endif
78         var rootPointer = (void*)root;

```

```

68     var currentNode =
69         ↪ System.Runtime.CompilerServices.Unsafe.Read<TElement>(rootPointer);
70     while (true)
71     {
72         if (FirstIsToTheLeftOfSecond(node, currentNode))
73         {
74             if (GetLeftIsChild(currentNode))
75             {
76                 IncrementSize(currentNode);
77                 path[pathPosition++] = currentNode;
78                 currentNode = GetLeftValue(currentNode);
79             }
80             else
81             {
82                 // Threads
83                 SetLeft(node, GetLeftValue(currentNode));
84                 SetRight(node, currentNode);
85                 SetLeft(currentNode, node);
86                 SetLeftIsChild(currentNode, true);
87                 DecrementBalance(currentNode);
88                 SetSize(node, GetOne());
89                 FixSize(currentNode); // Should be incremented already
90                 break;
91             }
92         }
93         else if (FirstIsToTheRightOfSecond(node, currentNode))
94         {
95             if (GetRightIsChild(currentNode))
96             {
97                 IncrementSize(currentNode);
98                 path[pathPosition++] = currentNode;
99                 currentNode = GetRightValue(currentNode);
100             }
101             else
102             {
103                 // Threads
104                 SetRight(node, GetRightValue(currentNode));
105                 SetLeft(node, currentNode);
106                 SetRight(currentNode, node);
107                 SetRightIsChild(currentNode, true);
108                 IncrementBalance(currentNode);
109                 SetSize(node, GetOne());
110                 FixSize(currentNode); // Should be incremented already
111                 break;
112             }
113         }
114         else
115         {
116             throw new InvalidOperationException("Node with the same key already
117                 ↪ attached to a tree.");
118         }
119     }
120     // Restore balance. This is the goodness of a non-recursive
121     // implementation, when we are done with balancing we 'break'
122     // the loop and we are done.
123     while (true)
124     {
125         var parent = path[--pathPosition];
126         var isLeftNode = !IsEquals(parent, default) && IsEquals(currentNode,
127             ↪ GetLeftValue(parent));
128         var currentNodeBalance = GetBalance(currentNode);
129         if (currentNodeBalance < -1 || currentNodeBalance > 1)
130         {
131             currentNode = Balance(currentNode);
132             if (IsEquals(parent, default))
133             {
134                 System.Runtime.CompilerServices.Unsafe.Write((void*)root,
135                     ↪ currentNode);
136             }
137             else if (isLeftNode)
138             {
139                 SetLeft(parent, currentNode);
140                 FixSize(parent);
141             }
142             else
143             {
144                 SetRight(parent, currentNode);
145                 FixSize(parent);
146             }
147         }
148     }

```

```

142     }
143 }
144 currentNodeBalance = GetBalance(currentNode);
145 if (currentNodeBalance == 0 || IsEquals(parent, default))
146 {
147     break;
148 }
149 if (isLeftNode)
150 {
151     DecrementBalance(parent);
152 }
153 else
154 {
155     IncrementBalance(parent);
156 }
157 currentNode = parent;
158 }
159 #if USEARRAYPOOL
160     ArrayPool.Free(path);
161 #endif
162 }
163 }
164
165 private TElement Balance(TElement node)
166 {
167     unchecked
168     {
169         var rootBalance = GetBalance(node);
170         if (rootBalance < -1)
171         {
172             var left = GetLeftValue(node);
173             if (GetBalance(left) > 0)
174             {
175                 SetLeft(node, LeftRotateWithBalance(left));
176                 FixSize(node);
177             }
178             node = RightRotateWithBalance(node);
179         }
180         else if (rootBalance > 1)
181         {
182             var right = GetRightValue(node);
183             if (GetBalance(right) < 0)
184             {
185                 SetRight(node, RightRotateWithBalance(right));
186                 FixSize(node);
187             }
188             node = LeftRotateWithBalance(node);
189         }
190         return node;
191     }
192 }
193
194 protected TElement LeftRotateWithBalance(TElement node)
195 {
196     unchecked
197     {
198         var right = GetRightValue(node);
199         if (GetLeftIsChild(right))
200         {
201             SetRight(node, GetLeftValue(right));
202         }
203         else
204         {
205             SetRightIsChild(node, false);
206             SetLeftIsChild(right, true);
207         }
208         SetLeft(right, node);
209         // Fix size
210         SetSize(right, GetSize(node));
211         FixSize(node);
212         // Fix balance
213         var rootBalance = GetBalance(node);
214         var rightBalance = GetBalance(right);
215         if (rightBalance <= 0)
216         {
217             if (rootBalance >= 1)
218             {
219                 SetBalance(right, (sbyte)(rightBalance - 1));

```

```

220     }
221     else
222     {
223         SetBalance(right, (sbyte)(rootBalance + rightBalance - 2));
224     }
225     SetBalance(node, (sbyte)(rootBalance - 1));
226 }
227 else
228 {
229     if (rootBalance <= rightBalance)
230     {
231         SetBalance(right, (sbyte)(rootBalance - 2));
232     }
233     else
234     {
235         SetBalance(right, (sbyte)(rightBalance - 1));
236     }
237     SetBalance(node, (sbyte)(rootBalance - rightBalance - 1));
238 }
239 return right;
240 }
241 }
242
243 protected TElement RightRotateWithBalance(TElement node)
244 {
245     unchecked
246     {
247         var left = GetLeftValue(node);
248         if (GetRightIsChild(left))
249         {
250             SetLeft(node, GetRightValue(left));
251         }
252         else
253         {
254             SetLeftIsChild(node, false);
255             SetRightIsChild(left, true);
256         }
257         SetRight(left, node);
258         // Fix size
259         SetSize(left, GetSize(node));
260         FixSize(node);
261         // Fix balance
262         var rootBalance = GetBalance(node);
263         var leftBalance = GetBalance(left);
264         if (leftBalance <= 0)
265         {
266             if (leftBalance > rootBalance)
267             {
268                 SetBalance(left, (sbyte)(leftBalance + 1));
269             }
270             else
271             {
272                 SetBalance(left, (sbyte)(rootBalance + 2));
273             }
274             SetBalance(node, (sbyte)(rootBalance - leftBalance + 1));
275         }
276         else
277         {
278             if (rootBalance <= -1)
279             {
280                 SetBalance(left, (sbyte)(leftBalance + 1));
281             }
282             else
283             {
284                 SetBalance(left, (sbyte)(rootBalance + leftBalance + 2));
285             }
286             SetBalance(node, (sbyte)(rootBalance + 1));
287         }
288         return left;
289     }
290 }
291
292 protected TElement GetNext(TElement node)
293 {
294     unchecked
295     {
296         var current = GetRightValue(node);
297         if (GetRightIsChild(node))
298         {

```

```

299         while (GetLeftIsChild(current))
300         {
301             current = GetLeftValue(current);
302         }
303     }
304     return current;
305 }
306 }
307
308 protected TElement GetPrevious(TElement node)
309 {
310     unchecked
311     {
312         var current = GetLeftValue(node);
313         if (GetLeftIsChild(node))
314         {
315             while (GetRightIsChild(current))
316             {
317                 current = GetRightValue(current);
318             }
319         }
320         return current;
321     }
322 }
323
324 protected override void DetachCore(IntPtr root, TElement node)
325 {
326     unchecked
327     {
328         #if USEARRAYPOOL
329             var path = ArrayPool.Allocate<TElement>(MaxPath);
330             var pathPosition = 0;
331             path[pathPosition++] = default;
332         #else
333             var path = new TElement[MaxPath];
334             var pathPosition = 1;
335         #endif
336         var rootPointer = (void*)root;
337         var currentNode =
338             ↪ System.Runtime.CompilerServices.Unsafe.Read<TElement>(rootPointer);
339         while (true)
340         {
341             if (FirstIsToTheLeftOfSecond(node, currentNode))
342             {
343                 if (!GetLeftIsChild(currentNode))
344                 {
345                     throw new InvalidOperationException("Cannot find a node.");
346                 }
347                 DecrementSize(currentNode);
348                 path[pathPosition++] = currentNode;
349                 currentNode = GetLeftValue(currentNode);
350             }
351             else if (FirstIsToTheRightOfSecond(node, currentNode))
352             {
353                 if (!GetRightIsChild(currentNode))
354                 {
355                     throw new InvalidOperationException("Cannot find a node.");
356                 }
357                 DecrementSize(currentNode);
358                 path[pathPosition++] = currentNode;
359                 currentNode = GetRightValue(currentNode);
360             }
361             else
362             {
363                 break;
364             }
365         }
366         var parent = path[--pathPosition];
367         var balanceNode = parent;
368         var isLeftNode = !IsEquals(parent, default) && IsEquals(currentNode,
369             ↪ GetLeftValue(parent));
370         if (!GetLeftIsChild(currentNode))
371         {
372             if (!GetRightIsChild(currentNode)) // node has no children
373             {
374                 if (IsEquals(parent, default))
375                 {
376                     System.Runtime.CompilerServices.Unsafe.Write(rootPointer, GetZero());
377                 }

```



```

376     else if (isLeftNode)
377     {
378         SetLeftIsChild(parent, false);
379         SetLeft(parent, GetLeftValue(currentNode));
380         IncrementBalance(parent);
381     }
382     else
383     {
384         SetRightIsChild(parent, false);
385         SetRight(parent, GetRightValue(currentNode));
386         DecrementBalance(parent);
387     }
388 }
389 else // node has a right child
390 {
391     var successor = GetNext(currentNode);
392     SetLeft(successor, GetLeftValue(currentNode));
393     var right = GetRightValue(currentNode);
394     if (IsEquals(parent, default))
395     {
396         System.Runtime.CompilerServices.Unsafe.Write(rootPointer, right);
397     }
398     else if (isLeftNode)
399     {
400         SetLeft(parent, right);
401         IncrementBalance(parent);
402     }
403     else
404     {
405         SetRight(parent, right);
406         DecrementBalance(parent);
407     }
408 }
409 }
410 else // node has a left child
411 {
412     if (!GetRightIsChild(currentNode))
413     {
414         var predecessor = GetPrevious(currentNode);
415         SetRight(predecessor, GetRightValue(currentNode));
416         var leftValue = GetLeftValue(currentNode);
417         if (IsEquals(parent, default))
418         {
419             System.Runtime.CompilerServices.Unsafe.Write(rootPointer, leftValue);
420         }
421         else if (isLeftNode)
422         {
423             SetLeft(parent, leftValue);
424             IncrementBalance(parent);
425         }
426         else
427         {
428             SetRight(parent, leftValue);
429             DecrementBalance(parent);
430         }
431     }
432     else // node has a both children (left and right)
433     {
434         var predecessor = GetLeftValue(currentNode);
435         var successor = GetRightValue(currentNode);
436         var successorParent = currentNode;
437         int previousPathPosition = ++pathPosition;
438         // find the immediately next node (and its parent)
439         while (GetLeftIsChild(successor))
440         {
441             path[++pathPosition] = successorParent = successor;
442             successor = GetLeftValue(successor);
443             if (!IsEquals(successorParent, currentNode))
444             {
445                 DecrementSize(successorParent);
446             }
447         }
448         path[previousPathPosition] = successor;
449         balanceNode = path[pathPosition];
450         // remove 'successor' from the tree
451         if (!IsEquals(successorParent, currentNode))
452         {
453             if (!GetRightIsChild(successor))

```

```

454         {
455             SetLeftIsChild(successorParent, false);
456         }
457         else
458         {
459             SetLeft(successorParent, GetRightValue(successor));
460         }
461         IncrementBalance(successorParent);
462         SetRightIsChild(successor, true);
463         SetRight(successor, GetRightValue(currentNode));
464     }
465     else
466     {
467         DecrementBalance(currentNode);
468     }
469     // set the predecessor's successor link to point to the right place
470     while (GetRightIsChild(predecessor))
471     {
472         predecessor = GetRightValue(predecessor);
473     }
474     SetRight(predecessor, successor);
475     // prepare 'successor' to replace 'node'
476     var left = GetLeftValue(currentNode);
477     SetLeftIsChild(successor, true);
478     SetLeft(successor, left);
479     SetBalance(successor, GetBalance(currentNode));
480     FixSize(successor);
481     if (IsEquals(parent, default))
482     {
483         System.Runtime.CompilerServices.Unsafe.Write(rootPointer, successor);
484     }
485     else if (isLeftNode)
486     {
487         SetLeft(parent, successor);
488     }
489     else
490     {
491         SetRight(parent, successor);
492     }
493     }
494 }
495 // restore balance
496 if (!IsEquals(balanceNode, default))
497 {
498     while (true)
499     {
500         var balanceParent = path[--pathPosition];
501         isLeftNode = !IsEquals(balanceParent, default) && IsEquals(balanceNode,
502             ↪ GetLeftValue(balanceParent));
503         var currentNodeBalance = GetBalance(balanceNode);
504         if (currentNodeBalance < -1 || currentNodeBalance > 1)
505         {
506             balanceNode = Balance(balanceNode);
507             if (IsEquals(balanceParent, default))
508             {
509                 System.Runtime.CompilerServices.Unsafe.Write(rootPointer,
510                     ↪ balanceNode);
511             }
512             else if (isLeftNode)
513             {
514                 SetLeft(balanceParent, balanceNode);
515             }
516             else
517             {
518                 SetRight(balanceParent, balanceNode);
519             }
520         }
521         currentNodeBalance = GetBalance(balanceNode);
522         if (currentNodeBalance != 0 || IsEquals(balanceParent, default))
523         {
524             break;
525         }
526         if (isLeftNode)
527         {
528             IncrementBalance(balanceParent);
529         }
530     }
531 }

```

```

530         DecrementBalance(balanceParent);
531     }
532     balanceNode = balanceParent;
533 }
534 }
535 ClearNode(node);
536 #if USEARRAYPOOL
537     ArrayPool.Free(path);
538 #endif
539 }
540 }
541
542 [MethodImpl(MethodImplOptions.AggressiveInlining)]
543 protected override void ClearNode(TElement node)
544 {
545     SetLeft(node, GetZero());
546     SetRight(node, GetZero());
547     SetSize(node, GetZero());
548     SetLeftIsChild(node, false);
549     SetRightIsChild(node, false);
550     SetBalance(node, 0);
551 }
552 }
553 }

```

# ./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)]
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 FirstIsToTheLeftOfSecond(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) => GetLeftPointer(node) !=
46             ↳ IntPtr.Zero ? GetLeftValue(node) : default;
47
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         protected virtual TElement GetRightOrDefault(TElement node) => GetRightPointer(node) !=
50             ↳ IntPtr.Zero ? GetRightValue(node) : default;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         protected void IncrementSize(TElement node) => SetSize(node, Increment(GetSize(node)));
54     }
55 }

```

```

52 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53 protected void DecrementSize(TElement node) => SetSize(node, Decrement(GetSize(node)));
54
55 [MethodImpl(MethodImplOptions.AggressiveInlining)]
56 protected TElement GetLeftSize(TElement node) => GetSizeOrZero(GetLeftOrDefault(node));
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 protected TElement GetRightSize(TElement node) => GetSizeOrZero(GetRightOrDefault(node));
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected TElement GetSizeOrZero(TElement node) => EqualToZero(node) ? GetZero() :
    ↳ GetSize(node);
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 protected void FixSize(TElement node) => SetSize(node, Increment(Add(GetLeftSize(node),
    ↳ GetRightSize(node))));
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 protected void LeftRotate(IntPtr root)
69 {
70     var rootPointer = (void*)root;
71     System.Runtime.CompilerServices.Unsafe.Write(rootPointer,
    ↳ LeftRotate(System.Runtime.CompilerServices.Unsafe.Read<TElement>(rootPointer)));
72 }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 protected TElement LeftRotate(TElement root)
76 {
77     var right = GetRightValue(root);
78 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
79     if (EqualToZero(right))
80     {
81         throw new Exception("Right is null.");
82     }
83 #endif
84     SetRight(root, GetLeftValue(right));
85     SetLeft(right, root);
86     SetSize(right, GetSize(root));
87     FixSize(root);
88     return right;
89 }
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 protected void RightRotate(IntPtr root)
93 {
94     var rootPointer = (void*)root;
95     System.Runtime.CompilerServices.Unsafe.Write(rootPointer,
    ↳ RightRotate(System.Runtime.CompilerServices.Unsafe.Read<TElement>(rootPointer)));
96 }
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 protected TElement RightRotate(TElement root)
100 {
101     var left = GetLeftValue(root);
102 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
103     if (EqualToZero(left))
104     {
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 public TElement GetSize(IntPtr root) => root == IntPtr.Zero ? GetZero() :
    ↳ GetSizeOrZero(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
205
206 public void FixSizes(IntPtr root)
207 {
208     if (root != IntPtr.Zero)
209     {
210         FixSizes(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root));
211     }
212 }
213
214 public void FixSizes(TElement node)
215 {
216     if (IsEquals(node, default))
217     {
218         return;
219     }
220     FixSizes(GetLeftOrDefault(node));
221     FixSizes(GetRightOrDefault(node));
222     FixSize(node);
223 }
224
225 public void ValidateSizes(IntPtr root)
226 {
227     if (root != IntPtr.Zero)
228     {
229         ValidateSizes(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root)
    ↳ );
230     }
231 }
232
233 public void ValidateSizes(TElement node)
234 {
235     if (IsEquals(node, default))
236     {
237         return;
238     }
239     var size = GetSize(node);
240     var leftSize = GetLeftSize(node);
241     var rightSize = GetRightSize(node);
242     var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
243     if (!IsEquals(size, expectedSize))
244     {
245         throw new InvalidOperationException($"Size of {node} is not valid. Expected
    ↳ size: {expectedSize}, actual size: {size}.");
246     }
247     ValidateSizes(GetLeftOrDefault(node));
248     ValidateSizes(GetRightOrDefault(node));
249 }
250
251 public void ValidateSize(TElement node)
252 {
253     var size = GetSize(node);
254     var leftSize = GetLeftSize(node);
255     var rightSize = GetRightSize(node);
256     var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
257     if (!IsEquals(size, expectedSize))
258     {
259         throw new InvalidOperationException($"Size of {node} is not valid. Expected
    ↳ size: {expectedSize}, actual size: {size}.");
260     }
261 }
262
263 public string PrintNodes(IntPtr root)
264 {
265     if (root != IntPtr.Zero)
266     {
267         var sb = new StringBuilder();
268         PrintNodes(System.Runtime.CompilerServices.Unsafe.Read<TElement>((void*)root),
    ↳ sb);
269         return sb.ToString();
270     }
271     return "";
272 }
273
274 public string PrintNodes(TElement node)
275 {
276     var sb = new StringBuilder();

```

```

277     PrintNodes(node, sb);
278     return sb.ToString();
279 }
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 }

```

## Index

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