```csharp
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Numbers;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Methods
{
    public abstract class GenericCollectionMethodsBase<TElement>
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetZero() => default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool EqualToZero(TElement value) => EqualityComparer.Equals(value,
        ↪  Zero);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool AreEqual(TElement first, TElement second) =>
        ↪  EqualityComparer.Equals(first, second);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterThanZero(TElement value) => Comparer.Compare(value, Zero)
        ↪  > 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterThan(TElement first, TElement second) =>
        ↪  Comparer.Compare(first, second) > 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterOrEqualThanZero(TElement value) => Comparer.Compare(value,
        ↪  Zero) >= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool GreaterOrEqualThan(TElement first, TElement second) =>
        ↪  Comparer.Compare(first, second) >= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessOrEqualThanZero(TElement value) => Comparer.Compare(value,
        ↪  Zero) <= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessOrEqualThan(TElement first, TElement second) =>
        ↪  Comparer.Compare(first, second) <= 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessThanZero(TElement value) => Comparer.Compare(value, Zero) < 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool LessThan(TElement first, TElement second) =>
        ↪  Comparer.Compare(first, second) < 0;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement Increment(TElement value) =>
        ↪  Arithmetic<TElement>.Increment(value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement Decrement(TElement value) =>
        ↪  Arithmetic<TElement>.Decrement(value);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement Add(TElement first, TElement second) =>
        ↪  Arithmetic<TElement>.Add(first, second);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement Subtract(TElement first, TElement second) =>
        ↪  Arithmetic<TElement>.Subtract(first, second);

        protected readonly TElement Zero;
        protected readonly TElement One;
        protected readonly TElement Two;
        protected readonly EqualityComparer<TElement> EqualityComparer;
        protected readonly Comparer<TElement> Comparer;

        protected GenericCollectionMethodsBase()
        {
            EqualityComparer = EqualityComparer<TElement>.Default;
```

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

1.2 ./csharp/Platform.Collections.Methods/Lists/CircularDoublyLinkedListMethods.cs

```csharp
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Methods.Lists
4  {
5      public abstract class CircularDoublyLinkedListMethods<TElement> :
     ↪   DoublyLinkedListMethodsBase<TElement>
6      {
7          public void AttachBefore(TElement baseElement, TElement newElement)
8          {
9              var baseElementPrevious = GetPrevious(baseElement);
10             SetPrevious(newElement, baseElementPrevious);
11             SetNext(newElement, baseElement);
12             if (AreEqual(baseElement, GetFirst()))
13             {
14                 SetFirst(newElement);
15             }
16             SetNext(baseElementPrevious, newElement);
17             SetPrevious(baseElement, newElement);
18             IncrementSize();
19         }
20
21         public void AttachAfter(TElement baseElement, TElement newElement)
22         {
23             var baseElementNext = GetNext(baseElement);
24             SetPrevious(newElement, baseElement);
25             SetNext(newElement, baseElementNext);
26             if (AreEqual(baseElement, GetLast()))
27             {
28                 SetLast(newElement);
29             }
30             SetPrevious(baseElementNext, newElement);
31             SetNext(baseElement, newElement);
32             IncrementSize();
33         }
34
35         public void AttachAsFirst(TElement element)
36         {
37             var first = GetFirst();
38             if (EqualToZero(first))
39             {
40                 SetFirst(element);
41                 SetLast(element);
42                 SetPrevious(element, element);
43                 SetNext(element, element);
44                 IncrementSize();
45             }
46             else
47             {
48                 AttachBefore(first, element);
49             }
50         }
51
52         public void AttachAsLast(TElement element)
53         {
54             var last = GetLast();
55             if (EqualToZero(last))
56             {
57                 AttachAsFirst(element);
58             }
59             else
60             {
61                 AttachAfter(last, element);
62             }
63         }
64
65         public void Detach(TElement element)
66         {
67             var elementPrevious = GetPrevious(element);
68             var elementNext = GetNext(element);
```

```
69          if (AreEqual(elementNext, element))
70          {
71              SetFirst(Zero);
72              SetLast(Zero);
73          }
74          else
75          {
76              SetNext(elementPrevious, elementNext);
77              SetPrevious(elementNext, elementPrevious);
78              if (AreEqual(element, GetFirst()))
79              {
80                  SetFirst(elementNext);
81              }
82              if (AreEqual(element, GetLast()))
83              {
84                  SetLast(elementPrevious);
85              }
86          }
87          SetPrevious(element, Zero);
88          SetNext(element, Zero);
89          DecrementSize();
90      }
91    }
92 }
```

1.3 ./csharp/Platform.Collections.Methods/Lists/DoublyLinkedListMethodsBase.cs

```csharp
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
    ↪   list</a> implementation.
9      /// </remarks>
10     public abstract class DoublyLinkedListMethodsBase<TElement> :
    ↪   GenericCollectionMethodsBase<TElement>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         protected abstract TElement GetFirst();
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected abstract TElement GetLast();
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected abstract TElement GetPrevious(TElement element);
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         protected abstract TElement GetNext(TElement element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected abstract TElement GetSize();
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected abstract void SetFirst(TElement element);
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected abstract void SetLast(TElement element);
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected abstract void SetPrevious(TElement element, TElement previous);
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         protected abstract void SetNext(TElement element, TElement next);
38
39         [MethodImpl(MethodImplOptions.AggressiveInlining)]
40         protected abstract void SetSize(TElement size);
41
42         [MethodImpl(MethodImplOptions.AggressiveInlining)]
43         protected void IncrementSize() => SetSize(Increment(GetSize()));
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         protected void DecrementSize() => SetSize(Decrement(GetSize()));
47     }
48 }
```

```csharp
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Methods.Lists
{
    public abstract class OpenDoublyLinkedListMethods<TElement> :
    ↪    DoublyLinkedListMethodsBase<TElement>
    {
        public void AttachBefore(TElement baseElement, TElement newElement)
        {
            var baseElementPrevious = GetPrevious(baseElement);
            SetPrevious(newElement, baseElementPrevious);
            SetNext(newElement, baseElement);
            if (EqualToZero(baseElementPrevious))
            {
                SetFirst(newElement);
            }
            else
            {
                SetNext(baseElementPrevious, newElement);
            }
            SetPrevious(baseElement, newElement);
            IncrementSize();
        }

        public void AttachAfter(TElement baseElement, TElement newElement)
        {
            var baseElementNext = GetNext(baseElement);
            SetPrevious(newElement, baseElement);
            SetNext(newElement, baseElementNext);
            if (EqualToZero(baseElementNext))
            {
                SetLast(newElement);
            }
            else
            {
                SetPrevious(baseElementNext, newElement);
            }
            SetNext(baseElement, newElement);
            IncrementSize();
        }

        public void AttachAsFirst(TElement element)
        {
            var first = GetFirst();
            if (EqualToZero(first))
            {
                SetFirst(element);
                SetLast(element);
                SetPrevious(element, Zero);
                SetNext(element, Zero);
                IncrementSize();
            }
            else
            {
                AttachBefore(first, element);
            }
        }

        public void AttachAsLast(TElement element)
        {
            var last = GetLast();
            if (EqualToZero(last))
            {
                AttachAsFirst(element);
            }
            else
            {
                AttachAfter(last, element);
            }
        }

        public void Detach(TElement element)
        {
            var elementPrevious = GetPrevious(element);
            var elementNext = GetNext(element);
            if (EqualToZero(elementPrevious))
            {
```

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

1.5 ./csharp/Platform.Collections.Methods/Trees/RecursionlessSizeBalancedTreeMethods.cs

```
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Methods.Trees
4  {
5      public abstract class RecursionlessSizeBalancedTreeMethods<TElement> :
      ↪   SizedBinaryTreeMethodsBase<TElement>
6      {
7          protected override void AttachCore(ref TElement root, TElement node)
8          {
9              while (true)
10             {
11                 ref var left = ref GetLeftReference(root);
12                 var leftSize = GetSizeOrZero(left);
13                 ref var right = ref GetRightReference(root);
14                 var rightSize = GetSizeOrZero(right);
15                 if (FirstIsToTheLeftOfSecond(node, root)) // node.Key less than root.Key
16                 {
17                     if (EqualToZero(left))
18                     {
19                         IncrementSize(root);
20                         SetSize(node, One);
21                         left = node;
22                         return;
23                     }
24                     if (FirstIsToTheLeftOfSecond(node, left)) // node.Key less than left.Key
25                     {
26                         if (GreaterThan(Increment(leftSize), rightSize))
27                         {
28                             RightRotate(ref root);
29                         }
30                         else
31                         {
32                             IncrementSize(root);
33                             root = ref left;
34                         }
35                     }
36                     else  // node.Key greater than left.Key
37                     {
38                         var leftRightSize = GetSizeOrZero(GetRight(left));
39                         if (GreaterThan(Increment(leftRightSize), rightSize))
40                         {
41                             if (EqualToZero(leftRightSize) && EqualToZero(rightSize))
42                             {
43                                 SetLeft(node, left);
44                                 SetRight(node, root);
45                                 SetSize(node, Add(leftSize, Two)); // Two (2) - node the size of
                                 ↪   root and a node itself
46                                 SetLeft(root, Zero);
47                                 SetSize(root, One);
48                                 root = node;
49                                 return;
50                             }
51                             LeftRotate(ref left);
52                             RightRotate(ref root);
53                         }
54                         else
55                         {
```

```
56                        IncrementSize(root);
57                        root = ref left;
58                    }
59                }
60            }
61            else // node.Key greater than root.Key
62            {
63                if (EqualToZero(right))
64                {
65                    IncrementSize(root);
66                    SetSize(node, One);
67                    right = node;
68                    return;
69                }
70                if (FirstIsToTheRightOfSecond(node, right)) // node.Key greater than
   ↪  right.Key
71                {
72                    if (GreaterThan(Increment(rightSize), leftSize))
73                    {
74                        LeftRotate(ref root);
75                    }
76                    else
77                    {
78                        IncrementSize(root);
79                        root = ref right;
80                    }
81                }
82                else // node.Key less than right.Key
83                {
84                    var rightLeftSize = GetSizeOrZero(GetLeft(right));
85                    if (GreaterThan(Increment(rightLeftSize), leftSize))
86                    {
87                        if (EqualToZero(rightLeftSize) && EqualToZero(leftSize))
88                        {
89                            SetLeft(node, root);
90                            SetRight(node, right);
91                            SetSize(node, Add(rightSize, Two)); // Two (2) - node the size
   ↪   of root and a node itself
92                            SetRight(root, Zero);
93                            SetSize(root, One);
94                            root = node;
95                            return;
96                        }
97                        RightRotate(ref right);
98                        LeftRotate(ref root);
99                    }
100                   else
101                   {
102                       IncrementSize(root);
103                       root = ref right;
104                   }
105               }
106           }
107       }
108   }
109
110   protected override void DetachCore(ref TElement root, TElement node)
111   {
112       while (true)
113       {
114           ref var left = ref GetLeftReference(root);
115           var leftSize = GetSizeOrZero(left);
116           ref var right = ref GetRightReference(root);
117           var rightSize = GetSizeOrZero(right);
118           if (FirstIsToTheLeftOfSecond(node, root)) // node.Key less than root.Key
119           {
120               var decrementedLeftSize = Decrement(leftSize);
121               if (GreaterThan(GetSizeOrZero(GetRight(right)), decrementedLeftSize))
122               {
123                   LeftRotate(ref root);
124               }
125               else if (GreaterThan(GetSizeOrZero(GetLeft(right)), decrementedLeftSize))
126               {
127                   RightRotate(ref right);
128                   LeftRotate(ref root);
129               }
130               else
131               {
```

```
132                     DecrementSize(root);
133                     root = ref left;
134                 }
135             }
136             else if (FirstIsToTheRightOfSecond(node, root)) // node.Key greater than root.Key
137             {
138                 var decrementedRightSize = Decrement(rightSize);
139                 if (GreaterThan(GetSizeOrZero(GetLeft(left)), decrementedRightSize))
140                 {
141                     RightRotate(ref root);
142                 }
143                 else if (GreaterThan(GetSizeOrZero(GetRight(left)), decrementedRightSize))
144                 {
145                     LeftRotate(ref left);
146                     RightRotate(ref root);
147                 }
148                 else
149                 {
150                     DecrementSize(root);
151                     root = ref right;
152                 }
153             }
154             else // key equals to root.Key
155             {
156                 if (GreaterThanZero(leftSize) && GreaterThanZero(rightSize))
157                 {
158                     TElement replacement;
159                     if (GreaterThan(leftSize, rightSize))
160                     {
161                         replacement = GetRightest(left);
162                         DetachCore(ref left, replacement);
163                     }
164                     else
165                     {
166                         replacement = GetLeftest(right);
167                         DetachCore(ref right, replacement);
168                     }
169                     SetLeft(replacement, left);
170                     SetRight(replacement, right);
171                     SetSize(replacement, Add(leftSize, rightSize));
172                     root = replacement;
173                 }
174                 else if (GreaterThanZero(leftSize))
175                 {
176                     root = left;
177                 }
178                 else if (GreaterThanZero(rightSize))
179                 {
180                     root = right;
181                 }
182                 else
183                 {
184                     root = Zero;
185                 }
186                 ClearNode(node);
187                 return;
188             }
189         }
190     }
191     }
192 }
```

1.6 ./csharp/Platform.Collections.Methods/Trees/SizeBalancedTreeMethods.cs

```
1  using System;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Methods.Trees
6  {
7      public abstract class SizeBalancedTreeMethods<TElement> :
         ↪   SizedBinaryTreeMethodsBase<TElement>
8      {
9          protected override void AttachCore(ref TElement root, TElement node)
10         {
11             if (EqualToZero(root))
12             {
13                 root = node;
14                 IncrementSize(root);
15             }
```

```csharp
            else
            {
                IncrementSize(root);
                if (FirstIsToTheLeftOfSecond(node, root))
                {
                    AttachCore(ref GetLeftReference(root), node);
                    LeftMaintain(ref root);
                }
                else
                {
                    AttachCore(ref GetRightReference(root), node);
                    RightMaintain(ref root);
                }
            }
        }

        protected override void DetachCore(ref TElement root, TElement nodeToDetach)
        {
            ref var currentNode = ref root;
            ref var parent = ref root;
            var replacementNode = Zero;
            while (!AreEqual(currentNode, nodeToDetach))
            {
                DecrementSize(currentNode);
                if (FirstIsToTheLeftOfSecond(nodeToDetach, currentNode))
                {
                    parent = ref currentNode;
                    currentNode = ref GetLeftReference(currentNode);
                }
                else if (FirstIsToTheRightOfSecond(nodeToDetach, currentNode))
                {
                    parent = ref currentNode;
                    currentNode = ref GetRightReference(currentNode);
                }
                else
                {
                    throw new InvalidOperationException("Duplicate link found in the tree.");
                }
            }
            var nodeToDetachLeft = GetLeft(nodeToDetach);
            var node = GetRight(nodeToDetach);
            if (!EqualToZero(nodeToDetachLeft) && !EqualToZero(node))
            {
                var leftestNode = GetLeftest(node);
                DetachCore(ref GetRightReference(nodeToDetach), leftestNode);
                SetLeft(leftestNode, nodeToDetachLeft);
                node = GetRight(nodeToDetach);
                if (!EqualToZero(node))
                {
                    SetRight(leftestNode, node);
                    SetSize(leftestNode, Increment(Add(GetSize(nodeToDetachLeft),
                        ↪ GetSize(node))));
                }
                else
                {
                    SetSize(leftestNode, Increment(GetSize(nodeToDetachLeft)));
                }
                replacementNode = leftestNode;
            }
            else if (!EqualToZero(nodeToDetachLeft))
            {
                replacementNode = nodeToDetachLeft;
            }
            else if (!EqualToZero(node))
            {
                replacementNode = node;
            }
            if (AreEqual(root, nodeToDetach))
            {
                root = replacementNode;
            }
            else if (AreEqual(GetLeft(parent), nodeToDetach))
            {
                SetLeft(parent, replacementNode);
            }
            else if (AreEqual(GetRight(parent), nodeToDetach))
            {
                SetRight(parent, replacementNode);
            }
```

```csharp
                    ClearNode(nodeToDetach);
        }

        private void LeftMaintain(ref TElement root)
        {
            if (!EqualToZero(root))
            {
                var rootLeftNode = GetLeft(root);
                if (!EqualToZero(rootLeftNode))
                {
                    var rootRightNode = GetRight(root);
                    var rootRightNodeSize = GetSize(rootRightNode);
                    var rootLeftNodeLeftNode = GetLeft(rootLeftNode);
                    if (!EqualToZero(rootLeftNodeLeftNode) &&
                        (EqualToZero(rootRightNode) ||
                        ↪ GreaterThan(GetSize(rootLeftNodeLeftNode), rootRightNodeSize)))
                    {
                        RightRotate(ref root);
                    }
                    else
                    {
                        var rootLeftNodeRightNode = GetRight(rootLeftNode);
                        if (!EqualToZero(rootLeftNodeRightNode) &&
                            (EqualToZero(rootRightNode) ||
                            ↪ GreaterThan(GetSize(rootLeftNodeRightNode), rootRightNodeSize)))
                        {
                            LeftRotate(ref GetLeftReference(root));
                            RightRotate(ref root);
                        }
                        else
                        {
                            return;
                        }
                    }
                    LeftMaintain(ref GetLeftReference(root));
                    RightMaintain(ref GetRightReference(root));
                    LeftMaintain(ref root);
                    RightMaintain(ref root);
                }
            }
        }

        private void RightMaintain(ref TElement root)
        {
            if (!EqualToZero(root))
            {
                var rootRightNode = GetRight(root);
                if (!EqualToZero(rootRightNode))
                {
                    var rootLeftNode = GetLeft(root);
                    var rootLeftNodeSize = GetSize(rootLeftNode);
                    var rootRightNodeRightNode = GetRight(rootRightNode);
                    if (!EqualToZero(rootRightNodeRightNode) &&
                        (EqualToZero(rootLeftNode) ||
                        ↪ GreaterThan(GetSize(rootRightNodeRightNode), rootLeftNodeSize)))
                    {
                        LeftRotate(ref root);
                    }
                    else
                    {
                        var rootRightNodeLeftNode = GetLeft(rootRightNode);
                        if (!EqualToZero(rootRightNodeLeftNode) &&
                            (EqualToZero(rootLeftNode) ||
                            ↪ GreaterThan(GetSize(rootRightNodeLeftNode), rootLeftNodeSize)))
                        {
                            RightRotate(ref GetRightReference(root));
                            LeftRotate(ref root);
                        }
                        else
                        {
                            return;
                        }
                    }
                    LeftMaintain(ref GetLeftReference(root));
                    RightMaintain(ref GetRightReference(root));
                    LeftMaintain(ref root);
                    RightMaintain(ref root);
                }
```

```
168                 }
169             }
170         }
171 }
```

1.7 ./csharp/Platform.Collections.Methods/Trees/SizedAndThreadedAVLBalancedTreeMethods.cs

```csharp
 1  using System;
 2  using System.Runtime.CompilerServices;
 3  using System.Text;
 4  #if USEARRAYPOOL
 5  using Platform.Collections;
 6  #endif
 7  using Platform.Reflection;
 8
 9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11  namespace Platform.Collections.Methods.Trees
12  {
13      /// <summary>
14      /// Combination of Size, Height (AVL), and threads.
15      /// </summary>
16      /// <remarks>
17      /// Based on: <a href="https://github.com/programmatom/TreeLib/blob/master/TreeLib/TreeLib/G↵
         ↪  enerated/AVLTreeList.cs">TreeLib.AVLTreeList</a>.
18      /// Which itself based on: <a
         ↪  href="https://github.com/GNOME/glib/blob/master/glib/gtree.c">GNOME/glib/gtree</a>.
19      /// </remarks>
20      public abstract class SizedAndThreadedAVLBalancedTreeMethods<TElement> :
         ↪  SizedBinaryTreeMethodsBase<TElement>
21      {
22          private static readonly int _maxPath = 11 * NumericType<TElement>.BytesSize + 4;
23
24          [MethodImpl(MethodImplOptions.AggressiveInlining)]
25          protected override TElement GetRightest(TElement current)
26          {
27              var currentRight = GetRightOrDefault(current);
28              while (!EqualToZero(currentRight))
29              {
30                  current = currentRight;
31                  currentRight = GetRightOrDefault(current);
32              }
33              return current;
34          }
35
36          [MethodImpl(MethodImplOptions.AggressiveInlining)]
37          protected override TElement GetLeftest(TElement current)
38          {
39              var currentLeft = GetLeftOrDefault(current);
40              while (!EqualToZero(currentLeft))
41              {
42                  current = currentLeft;
43                  currentLeft = GetLeftOrDefault(current);
44              }
45              return current;
46          }
47
48          public override bool Contains(TElement node, TElement root)
49          {
50              while (!EqualToZero(root))
51              {
52                  if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
53                  {
54                      root = GetLeftOrDefault(root);
55                  }
56                  else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
57                  {
58                      root = GetRightOrDefault(root);
59                  }
60                  else // node.Key == root.Key
61                  {
62                      return true;
63                  }
64              }
65              return false;
66          }
67
68          protected override void PrintNode(TElement node, StringBuilder sb, int level)
69          {
70              base.PrintNode(node, sb, level);
```

```csharp
                    sb.Append(' ');
                    sb.Append(GetLeftIsChild(node) ? 'l' : 'L');
                    sb.Append(GetRightIsChild(node) ? 'r' : 'R');
                    sb.Append(' ');
                    sb.Append(GetBalance(node));
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected void IncrementBalance(TElement node) => SetBalance(node,
            ↪  (sbyte)(GetBalance(node) + 1));

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected void DecrementBalance(TElement node) => SetBalance(node,
            ↪  (sbyte)(GetBalance(node) - 1));

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override TElement GetLeftOrDefault(TElement node) => GetLeftIsChild(node) ?
            ↪  GetLeft(node) : default;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override TElement GetRightOrDefault(TElement node) => GetRightIsChild(node) ?
            ↪  GetRight(node) : default;

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract bool GetLeftIsChild(TElement node);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract void SetLeftIsChild(TElement node, bool value);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract bool GetRightIsChild(TElement node);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract void SetRightIsChild(TElement node, bool value);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract sbyte GetBalance(TElement node);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract void SetBalance(TElement node, sbyte value);

            protected override void AttachCore(ref TElement root, TElement node)
            {
                unchecked
                {
                    // TODO: Check what is faster to use simple array or array from array pool
                    // TODO: Try to use stackalloc as an optimization (requires code generation,
                    ↪  because of generics)
#if USEARRAYPOOL
                    var path = ArrayPool.Allocate<TElement>(MaxPath);
                    var pathPosition = 0;
                    path[pathPosition++] = default;
#else
                    var path = new TElement[_maxPath];
                    var pathPosition = 1;
#endif
                    var currentNode = root;
                    while (true)
                    {
                        if (FirstIsToTheLeftOfSecond(node, currentNode))
                        {
                            if (GetLeftIsChild(currentNode))
                            {
                                IncrementSize(currentNode);
                                path[pathPosition++] = currentNode;
                                currentNode = GetLeft(currentNode);
                            }
                            else
                            {
                                // Threads
                                SetLeft(node, GetLeft(currentNode));
                                SetRight(node, currentNode);
                                SetLeft(currentNode, node);
                                SetLeftIsChild(currentNode, true);
                                DecrementBalance(currentNode);
                                SetSize(node, One);
                                FixSize(currentNode); // Should be incremented already
                                break;
                            }
                        }
```

```csharp
                    }
                    else if (FirstIsToTheRightOfSecond(node, currentNode))
                    {
                        if (GetRightIsChild(currentNode))
                        {
                            IncrementSize(currentNode);
                            path[pathPosition++] = currentNode;
                            currentNode = GetRight(currentNode);
                        }
                        else
                        {
                            // Threads
                            SetRight(node, GetRight(currentNode));
                            SetLeft(node, currentNode);
                            SetRight(currentNode, node);
                            SetRightIsChild(currentNode, true);
                            IncrementBalance(currentNode);
                            SetSize(node, One);
                            FixSize(currentNode); // Should be incremented already
                            break;
                        }
                    }
                    else
                    {
                        throw new InvalidOperationException("Node with the same key already
                        ↪  attached to a tree.");
                    }
                }
                // Restore balance. This is the goodness of a non-recursive
                // implementation, when we are done with balancing we 'break'
                // the loop and we are done.
                while (true)
                {
                    var parent = path[--pathPosition];
                    var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
                    ↪  GetLeft(parent));
                    var currentNodeBalance = GetBalance(currentNode);
                    if (currentNodeBalance < -1 || currentNodeBalance > 1)
                    {
                        currentNode = Balance(currentNode);
                        if (AreEqual(parent, default))
                        {
                            root = currentNode;
                        }
                        else if (isLeftNode)
                        {
                            SetLeft(parent, currentNode);
                            FixSize(parent);
                        }
                        else
                        {
                            SetRight(parent, currentNode);
                            FixSize(parent);
                        }
                    }
                    currentNodeBalance = GetBalance(currentNode);
                    if (currentNodeBalance == 0 || AreEqual(parent, default))
                    {
                        break;
                    }
                    if (isLeftNode)
                    {
                        DecrementBalance(parent);
                    }
                    else
                    {
                        IncrementBalance(parent);
                    }
                    currentNode = parent;
                }
#if USEARRAYPOOL
                ArrayPool.Free(path);
#endif
            }
        }

        private TElement Balance(TElement node)
        {
```

```csharp
            unchecked
            {
                var rootBalance = GetBalance(node);
                if (rootBalance < -1)
                {
                    var left = GetLeft(node);
                    if (GetBalance(left) > 0)
                    {
                        SetLeft(node, LeftRotateWithBalance(left));
                        FixSize(node);
                    }
                    node = RightRotateWithBalance(node);
                }
                else if (rootBalance > 1)
                {
                    var right = GetRight(node);
                    if (GetBalance(right) < 0)
                    {
                        SetRight(node, RightRotateWithBalance(right));
                        FixSize(node);
                    }
                    node = LeftRotateWithBalance(node);
                }
                return node;
            }
        }

        protected TElement LeftRotateWithBalance(TElement node)
        {
            unchecked
            {
                var right = GetRight(node);
                if (GetLeftIsChild(right))
                {
                    SetRight(node, GetLeft(right));
                }
                else
                {
                    SetRightIsChild(node, false);
                    SetLeftIsChild(right, true);
                }
                SetLeft(right, node);
                // Fix size
                SetSize(right, GetSize(node));
                FixSize(node);
                // Fix balance
                var rootBalance = GetBalance(node);
                var rightBalance = GetBalance(right);
                if (rightBalance <= 0)
                {
                    if (rootBalance >= 1)
                    {
                        SetBalance(right, (sbyte)(rightBalance - 1));
                    }
                    else
                    {
                        SetBalance(right, (sbyte)(rootBalance + rightBalance - 2));
                    }
                    SetBalance(node, (sbyte)(rootBalance - 1));
                }
                else
                {
                    if (rootBalance <= rightBalance)
                    {
                        SetBalance(right, (sbyte)(rootBalance - 2));
                    }
                    else
                    {
                        SetBalance(right, (sbyte)(rightBalance - 1));
                    }
                    SetBalance(node, (sbyte)(rootBalance - rightBalance - 1));
                }
                return right;
            }
        }

        protected TElement RightRotateWithBalance(TElement node)
        {
```

```csharp
                    unchecked
                    {
                        var left = GetLeft(node);
                        if (GetRightIsChild(left))
                        {
                            SetLeft(node, GetRight(left));
                        }
                        else
                        {
                            SetLeftIsChild(node, false);
                            SetRightIsChild(left, true);
                        }
                        SetRight(left, node);
                        // Fix size
                        SetSize(left, GetSize(node));
                        FixSize(node);
                        // Fix balance
                        var rootBalance = GetBalance(node);
                        var leftBalance = GetBalance(left);
                        if (leftBalance <= 0)
                        {
                            if (leftBalance > rootBalance)
                            {
                                SetBalance(left, (sbyte)(leftBalance + 1));
                            }
                            else
                            {
                                SetBalance(left, (sbyte)(rootBalance + 2));
                            }
                            SetBalance(node, (sbyte)(rootBalance - leftBalance + 1));
                        }
                        else
                        {
                            if (rootBalance <= -1)
                            {
                                SetBalance(left, (sbyte)(leftBalance + 1));
                            }
                            else
                            {
                                SetBalance(left, (sbyte)(rootBalance + leftBalance + 2));
                            }
                            SetBalance(node, (sbyte)(rootBalance + 1));
                        }
                        return left;
                    }
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override TElement GetNext(TElement node)
            {
                var current = GetRight(node);
                if (GetRightIsChild(node))
                {
                    return GetLeftest(current);
                }
                return current;
            }

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected override TElement GetPrevious(TElement node)
            {
                var current = GetLeft(node);
                if (GetLeftIsChild(node))
                {
                    return GetRightest(current);
                }
                return current;
            }

            protected override void DetachCore(ref TElement root, TElement node)
            {
                unchecked
                {
#if USEARRAYPOOL
                    var path = ArrayPool.Allocate<TElement>(MaxPath);
                    var pathPosition = 0;
                    path[pathPosition++] = default;
#else
                    var path = new TElement[_maxPath];
```

```
378         var pathPosition = 1;
379  #endif
380         var currentNode = root;
381         while (true)
382         {
383             if (FirstIsToTheLeftOfSecond(node, currentNode))
384             {
385                 if (!GetLeftIsChild(currentNode))
386                 {
387                     throw new InvalidOperationException("Cannot find a node.");
388                 }
389                 DecrementSize(currentNode);
390                 path[pathPosition++] = currentNode;
391                 currentNode = GetLeft(currentNode);
392             }
393             else if (FirstIsToTheRightOfSecond(node, currentNode))
394             {
395                 if (!GetRightIsChild(currentNode))
396                 {
397                     throw new InvalidOperationException("Cannot find a node.");
398                 }
399                 DecrementSize(currentNode);
400                 path[pathPosition++] = currentNode;
401                 currentNode = GetRight(currentNode);
402             }
403             else
404             {
405                 break;
406             }
407         }
408         var parent = path[--pathPosition];
409         var balanceNode = parent;
410         var isLeftNode = !AreEqual(parent, default) && AreEqual(currentNode,
             ↪  GetLeft(parent));
411         if (!GetLeftIsChild(currentNode))
412         {
413             if (!GetRightIsChild(currentNode)) // node has no children
414             {
415                 if (AreEqual(parent, default))
416                 {
417                     root = Zero;
418                 }
419                 else if (isLeftNode)
420                 {
421                     SetLeftIsChild(parent, false);
422                     SetLeft(parent, GetLeft(currentNode));
423                     IncrementBalance(parent);
424                 }
425                 else
426                 {
427                     SetRightIsChild(parent, false);
428                     SetRight(parent, GetRight(currentNode));
429                     DecrementBalance(parent);
430                 }
431             }
432             else // node has a right child
433             {
434                 var successor = GetNext(currentNode);
435                 SetLeft(successor, GetLeft(currentNode));
436                 var right = GetRight(currentNode);
437                 if (AreEqual(parent, default))
438                 {
439                     root = right;
440                 }
441                 else if (isLeftNode)
442                 {
443                     SetLeft(parent, right);
444                     IncrementBalance(parent);
445                 }
446                 else
447                 {
448                     SetRight(parent, right);
449                     DecrementBalance(parent);
450                 }
451             }
452         }
453         else // node has a left child
454         {
```

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

```
533                                    {
534                                        SetRight(parent, successor);
535                                    }
536                                }
537                            }
538                            // restore balance
539                            if (!AreEqual(balanceNode, default))
540                            {
541                                while (true)
542                                {
543                                    var balanceParent = path[--pathPosition];
544                                    isLeftNode = !AreEqual(balanceParent, default) && AreEqual(balanceNode,
                                    ↪   GetLeft(balanceParent));
545                                    var currentNodeBalance = GetBalance(balanceNode);
546                                    if (currentNodeBalance < -1 || currentNodeBalance > 1)
547                                    {
548                                        balanceNode = Balance(balanceNode);
549                                        if (AreEqual(balanceParent, default))
550                                        {
551                                            root = balanceNode;
552                                        }
553                                        else if (isLeftNode)
554                                        {
555                                            SetLeft(balanceParent, balanceNode);
556                                        }
557                                        else
558                                        {
559                                            SetRight(balanceParent, balanceNode);
560                                        }
561                                    }
562                                    currentNodeBalance = GetBalance(balanceNode);
563                                    if (currentNodeBalance != 0 || AreEqual(balanceParent, default))
564                                    {
565                                        break;
566                                    }
567                                    if (isLeftNode)
568                                    {
569                                        IncrementBalance(balanceParent);
570                                    }
571                                    else
572                                    {
573                                        DecrementBalance(balanceParent);
574                                    }
575                                    balanceNode = balanceParent;
576                                }
577                            }
578                            ClearNode(node);
579    #if USEARRAYPOOL
580                            ArrayPool.Free(path);
581    #endif
582                    }
583            }
584
585            [MethodImpl(MethodImplOptions.AggressiveInlining)]
586            protected override void ClearNode(TElement node)
587            {
588                SetLeft(node, Zero);
589                SetRight(node, Zero);
590                SetSize(node, Zero);
591                SetLeftIsChild(node, false);
592                SetRightIsChild(node, false);
593                SetBalance(node, 0);
594            }
595        }
596    }
```

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

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

```csharp
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract ref TElement GetLeftReference(TElement node);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract ref TElement GetRightReference(TElement node);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract TElement GetLeft(TElement node);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract TElement GetRight(TElement node);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract TElement GetSize(TElement node);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract void SetLeft(TElement node, TElement left);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract void SetRight(TElement node, TElement right);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract void SetSize(TElement node, TElement size);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract bool FirstIsToTheLeftOfSecond(TElement first, TElement second);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected abstract bool FirstIsToTheRightOfSecond(TElement first, TElement second);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected virtual TElement GetLeftOrDefault(TElement node) => AreEqual(node, default) ?
                default : GetLeft(node);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected virtual TElement GetRightOrDefault(TElement node) => AreEqual(node, default) ?
                default : GetRight(node);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected void IncrementSize(TElement node) => SetSize(node, Increment(GetSize(node)));

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected void DecrementSize(TElement node) => SetSize(node, Decrement(GetSize(node)));

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected TElement GetLeftSize(TElement node) => GetSizeOrZero(GetLeftOrDefault(node));

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected TElement GetRightSize(TElement node) => GetSizeOrZero(GetRightOrDefault(node));

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected TElement GetSizeOrZero(TElement node) => EqualToZero(node) ? Zero :
                GetSize(node);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected void FixSize(TElement node) => SetSize(node, Increment(Add(GetLeftSize(node),
                GetRightSize(node))));

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected void LeftRotate(ref TElement root) => root = LeftRotate(root);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            protected TElement LeftRotate(TElement root)
            {
                var right = GetRight(root);
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
                if (EqualToZero(right))
                {
                    throw new Exception("Right is null.");
                }
#endif
                SetRight(root, GetLeft(right));
                SetLeft(right, root);
                SetSize(right, GetSize(root));
                FixSize(root);
                return right;
            }
```

```csharp
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void RightRotate(ref TElement root) => root = RightRotate(root);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected TElement RightRotate(TElement root)
        {
            var left = GetLeft(root);
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
            if (EqualToZero(left))
            {
                throw new Exception("Left is null.");
            }
#endif
            SetLeft(root, GetRight(left));
            SetRight(left, root);
            SetSize(left, GetSize(root));
            FixSize(root);
            return left;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetRightest(TElement current)
        {
            var currentRight = GetRight(current);
            while (!EqualToZero(currentRight))
            {
                current = currentRight;
                currentRight = GetRight(current);
            }
            return current;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetLeftest(TElement current)
        {
            var currentLeft = GetLeft(current);
            while (!EqualToZero(currentLeft))
            {
                current = currentLeft;
                currentLeft = GetLeft(current);
            }
            return current;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetNext(TElement node) => GetLeftest(GetRight(node));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual TElement GetPrevious(TElement node) => GetRightest(GetLeft(node));

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Contains(TElement node, TElement root)
        {
            while (!EqualToZero(root))
            {
                if (FirstIsToTheLeftOfSecond(node, root)) // node.Key < root.Key
                {
                    root = GetLeft(root);
                }
                else if (FirstIsToTheRightOfSecond(node, root)) // node.Key > root.Key
                {
                    root = GetRight(root);
                }
                else // node.Key == root.Key
                {
                    return true;
                }
            }
            return false;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual void ClearNode(TElement node)
        {
            SetLeft(node, Zero);
            SetRight(node, Zero);
            SetSize(node, Zero);
        }
```

```csharp
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void Attach(ref TElement root, TElement node)
            {
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
                ValidateSizes(root);
                Debug.WriteLine("--BeforeAttach--");
                Debug.WriteLine(PrintNodes(root));
                Debug.WriteLine("----------------");
                var sizeBefore = GetSize(root);
#endif
                if (EqualToZero(root))
                {
                    SetSize(node, One);
                    root = node;
                    return;
                }
                AttachCore(ref root, node);
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
                Debug.WriteLine("--AfterAttach--");
                Debug.WriteLine(PrintNodes(root));
                Debug.WriteLine("----------------");
                ValidateSizes(root);
                var sizeAfter = GetSize(root);
                if (!IsEquals(MathHelpers.Increment(sizeBefore), sizeAfter))
                {
                    throw new Exception("Tree was broken after attach.");
                }
#endif
            }

            protected abstract void AttachCore(ref TElement root, TElement node);

            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void Detach(ref TElement root, TElement node)
            {
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
                ValidateSizes(root);
                Debug.WriteLine("--BeforeDetach--");
                Debug.WriteLine(PrintNodes(root));
                Debug.WriteLine("----------------");
                var sizeBefore = GetSize(root);
                if (ValueEqualToZero(root))
                {
                    throw new Exception($"Элемент с {node} не содержится в дереве.");
                }
#endif
                DetachCore(ref root, node);
#if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
                Debug.WriteLine("--AfterDetach--");
                Debug.WriteLine(PrintNodes(root));
                Debug.WriteLine("----------------");
                ValidateSizes(root);
                var sizeAfter = GetSize(root);
                if (!IsEquals(MathHelpers.Decrement(sizeBefore), sizeAfter))
                {
                    throw new Exception("Tree was broken after detach.");
                }
#endif
            }

            protected abstract void DetachCore(ref TElement root, TElement node);

            public void FixSizes(TElement node)
            {
                if (AreEqual(node, default))
                {
                    return;
                }
                FixSizes(GetLeft(node));
                FixSizes(GetRight(node));
                FixSize(node);
            }

            public void ValidateSizes(TElement node)
            {
                if (AreEqual(node, default))
                {
                    return;
                }
```

```csharp
                var size = GetSize(node);
                var leftSize = GetLeftSize(node);
                var rightSize = GetRightSize(node);
                var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
                if (!AreEqual(size, expectedSize))
                {
                    throw new InvalidOperationException($"Size of {node} is not valid. Expected
                    ↪  size: {expectedSize}, actual size: {size}.");
                }
                ValidateSizes(GetLeft(node));
                ValidateSizes(GetRight(node));
            }

        public void ValidateSize(TElement node)
        {
                var size = GetSize(node);
                var leftSize = GetLeftSize(node);
                var rightSize = GetRightSize(node);
                var expectedSize = Arithmetic.Increment(Arithmetic.Add(leftSize, rightSize));
                if (!AreEqual(size, expectedSize))
                {
                    throw new InvalidOperationException($"Size of {node} is not valid. Expected
                    ↪  size: {expectedSize}, actual size: {size}.");
                }
        }

        public string PrintNodes(TElement node)
        {
                var sb = new StringBuilder();
                PrintNodes(node, sb);
                return sb.ToString();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void PrintNodes(TElement node, StringBuilder sb) => PrintNodes(node, sb, 0);

        public void PrintNodes(TElement node, StringBuilder sb, int level)
        {
                if (AreEqual(node, default))
                {
                    return;
                }
                PrintNodes(GetLeft(node), sb, level + 1);
                PrintNode(node, sb, level);
                sb.AppendLine();
                PrintNodes(GetRight(node), sb, level + 1);
        }

        public string PrintNode(TElement node)
        {
                var sb = new StringBuilder();
                PrintNode(node, sb);
                return sb.ToString();
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected void PrintNode(TElement node, StringBuilder sb) => PrintNode(node, sb, 0);

        protected virtual void PrintNode(TElement node, StringBuilder sb, int level)
        {
                sb.Append('\t', level);
                sb.Append(node);
                PrintNodeValue(node, sb);
                sb.Append(' ');
                sb.Append('s');
                sb.Append(GetSize(node));
        }

        protected abstract void PrintNodeValue(TElement node, StringBuilder sb);
    }
}
```

1.9 ./csharp/Platform.Collections.Methods.Tests/RecursionlessSizeBalancedTree.cs

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using Platform.Numbers;
using Platform.Collections.Methods.Trees;
using Platform.Converters;
```

```csharp
namespace Platform.Collections.Methods.Tests
{
    public class RecursionlessSizeBalancedTree<TElement> :
        RecursionlessSizeBalancedTreeMethods<TElement>
    {
        private struct TreeElement
        {
            public TElement Size;
            public TElement Left;
            public TElement Right;
        }

        private readonly TreeElement[] _elements;
        private TElement _allocated;

        public TElement Root;

        public TElement Count => GetSizeOrZero(Root);

        public RecursionlessSizeBalancedTree(int capacity) => (_elements, _allocated) = (new
            TreeElement[capacity], One);

        public TElement Allocate()
        {
            var newNode = _allocated;
            if (IsEmpty(newNode))
            {
                _allocated = Arithmetic.Increment(_allocated);
                return newNode;
            }
            else
            {
                throw new InvalidOperationException("Allocated tree element is not empty.");
            }
        }

        public void Free(TElement node)
        {
            while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
            {
                var lastNode = Arithmetic.Decrement(_allocated);
                if (EqualityComparer.Equals(lastNode, node))
                {
                    _allocated = lastNode;
                    node = Arithmetic.Decrement(node);
                }
                else
                {
                    return;
                }
            }
        }

        public bool IsEmpty(TElement node) =>
            EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);

        protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
            Comparer.Compare(first, second) < 0;

        protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
            Comparer.Compare(first, second) > 0;

        protected override ref TElement GetLeftReference(TElement node) => ref
            GetElement(node).Left;

        protected override TElement GetLeft(TElement node) => GetElement(node).Left;

        protected override ref TElement GetRightReference(TElement node) => ref
            GetElement(node).Right;

        protected override TElement GetRight(TElement node) => GetElement(node).Right;

        protected override TElement GetSize(TElement node) => GetElement(node).Size;

        protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
            sb.Append(node);

        protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
            left;
```

```csharp
            protected override void SetRight(TElement node, TElement right) =>
            ↪   GetElement(node).Right = right;

            protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
            ↪   size;

            private ref TreeElement GetElement(TElement node) => ref
            ↪   _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
        }
    }
```

1.10 ./csharp/Platform.Collections.Methods.Tests/SizeBalancedTree.cs

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using Platform.Numbers;
using Platform.Collections.Methods.Trees;
using Platform.Converters;

namespace Platform.Collections.Methods.Tests
{
    public class SizeBalancedTree<TElement> : SizeBalancedTreeMethods<TElement>
    {
        private struct TreeElement
        {
            public TElement Size;
            public TElement Left;
            public TElement Right;
        }

        private readonly TreeElement[] _elements;
        private TElement _allocated;

        public TElement Root;

        public TElement Count => GetSizeOrZero(Root);

        public SizeBalancedTree(int capacity) => (_elements, _allocated) = (new
        ↪   TreeElement[capacity], One);

        public TElement Allocate()
        {
            var newNode = _allocated;
            if (IsEmpty(newNode))
            {
                _allocated = Arithmetic.Increment(_allocated);
                return newNode;
            }
            else
            {
                throw new InvalidOperationException("Allocated tree element is not empty.");
            }
        }

        public void Free(TElement node)
        {
            while (!EqualityComparer.Equals(_allocated, One) && IsEmpty(node))
            {
                var lastNode = Arithmetic.Decrement(_allocated);
                if (EqualityComparer.Equals(lastNode, node))
                {
                    _allocated = lastNode;
                    node = Arithmetic.Decrement(node);
                }
                else
                {
                    return;
                }
            }
        }

        public bool IsEmpty(TElement node) =>
        ↪   EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);

        protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
        ↪   Comparer.Compare(first, second) < 0;

        protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
        ↪   Comparer.Compare(first, second) > 0;
```

```
65        protected override ref TElement GetLeftReference(TElement node) => ref
      ↪   GetElement(node).Left;
66
67        protected override TElement GetLeft(TElement node) => GetElement(node).Left;
68
69        protected override ref TElement GetRightReference(TElement node) => ref
      ↪   GetElement(node).Right;
70
71        protected override TElement GetRight(TElement node) => GetElement(node).Right;
72
73        protected override TElement GetSize(TElement node) => GetElement(node).Size;
74
75        protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
      ↪   sb.Append(node);
76
77        protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
      ↪   left;
78
79        protected override void SetRight(TElement node, TElement right) =>
      ↪   GetElement(node).Right = right;
80
81        protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
      ↪   size;
82
83        private ref TreeElement GetElement(TElement node) => ref
      ↪   _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
84      }
85  }
```

1.11 ./csharp/Platform.Collections.Methods.Tests/SizedAndThreadedAVLBalancedTree.cs

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

```csharp
            {
                var lastNode = Arithmetic.Decrement(_allocated);
                if (EqualityComparer.Equals(lastNode, node))
                {
                    _allocated = lastNode;
                    node = Arithmetic.Decrement(node);
                }
                else
                {
                    return;
                }
            }
        }

        public bool IsEmpty(TElement node) =>
            EqualityComparer<TreeElement>.Default.Equals(GetElement(node), default);

        protected override bool FirstIsToTheLeftOfSecond(TElement first, TElement second) =>
            Comparer.Compare(first, second) < 0;

        protected override bool FirstIsToTheRightOfSecond(TElement first, TElement second) =>
            Comparer.Compare(first, second) > 0;

        protected override sbyte GetBalance(TElement node) => GetElement(node).Balance;

        protected override bool GetLeftIsChild(TElement node) => GetElement(node).LeftIsChild;

        protected override ref TElement GetLeftReference(TElement node) => ref
            GetElement(node).Left;

        protected override TElement GetLeft(TElement node) => GetElement(node).Left;

        protected override bool GetRightIsChild(TElement node) => GetElement(node).RightIsChild;

        protected override ref TElement GetRightReference(TElement node) => ref
            GetElement(node).Right;

        protected override TElement GetRight(TElement node) => GetElement(node).Right;

        protected override TElement GetSize(TElement node) => GetElement(node).Size;

        protected override void PrintNodeValue(TElement node, StringBuilder sb) =>
            sb.Append(node);

        protected override void SetBalance(TElement node, sbyte value) =>
            GetElement(node).Balance = value;

        protected override void SetLeft(TElement node, TElement left) => GetElement(node).Left =
            left;

        protected override void SetLeftIsChild(TElement node, bool value) =>
            GetElement(node).LeftIsChild = value;

        protected override void SetRight(TElement node, TElement right) =>
            GetElement(node).Right = right;

        protected override void SetRightIsChild(TElement node, bool value) =>
            GetElement(node).RightIsChild = value;

        protected override void SetSize(TElement node, TElement size) => GetElement(node).Size =
            size;

        private ref TreeElement GetElement(TElement node) => ref
            _elements[UncheckedConverter<TElement, long>.Default.Convert(node)];
    }
}
```

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

```csharp
using System;
using System.Collections.Generic;
using Xunit;
using Platform.Collections.Methods.Trees;
using Platform.Converters;

namespace Platform.Collections.Methods.Tests
{
    public static class TestExtensions
    {
```

```csharp
        public static void TestMultipleCreationsAndDeletions<TElement>(this
        ↪   SizedBinaryTreeMethodsBase<TElement> tree, Func<TElement> allocate, Action<TElement>
        ↪   free, ref TElement root, Func<TElement> treeCount, int maximumOperationsPerCycle)
        {
            for (var N = 1; N < maximumOperationsPerCycle; N++)
            {
                var currentCount = 0;
                for (var i = 0; i < N; i++)
                {
                    var node = allocate();
                    tree.Attach(ref root, node);
                    currentCount++;
                    Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                    ↪   int>.Default.Convert(treeCount()));
                }
                for (var i = 1; i <= N; i++)
                {
                    TElement node = UncheckedConverter<int, TElement>.Default.Convert(i);
                    if (tree.Contains(node, root))
                    {
                        tree.Detach(ref root, node);
                        free(node);
                        currentCount--;
                        Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                        ↪   int>.Default.Convert(treeCount()));
                    }
                }
            }
        }

        public static void TestMultipleRandomCreationsAndDeletions<TElement>(this
        ↪   SizedBinaryTreeMethodsBase<TElement> tree, ref TElement root, Func<TElement>
        ↪   treeCount, int maximumOperationsPerCycle)
        {
            var random = new System.Random(0);
            var added = new HashSet<TElement>();
            var currentCount = 0;
            for (var N = 1; N < maximumOperationsPerCycle; N++)
            {
                for (var i = 0; i < N; i++)
                {
                    var node = UncheckedConverter<int, TElement>.Default.Convert(random.Next(1,
                    ↪   N));
                    if (added.Add(node))
                    {
                        tree.Attach(ref root, node);
                        currentCount++;
                        Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                        ↪   int>.Default.Convert(treeCount()));
                    }
                }
                for (var i = 1; i <= N; i++)
                {
                    TElement node = UncheckedConverter<int,
                    ↪   TElement>.Default.Convert(random.Next(1, N));
                    if (tree.Contains(node, root))
                    {
                        tree.Detach(ref root, node);
                        currentCount--;
                        Assert.Equal(currentCount, (int)UncheckedConverter<TElement,
                        ↪   int>.Default.Convert(treeCount()));
                        added.Remove(node);
                    }
                }
            }
        }
    }
}
```

1.13 ./csharp/Platform.Collections.Methods.Tests/TreesTests.cs

```csharp
using Xunit;

namespace Platform.Collections.Methods.Tests
{
    public static class TreesTests
    {
        private const int _n = 500;

```

```csharp
        [Fact]
        public static void RecursionlessSizeBalancedTreeMultipleAttachAndDetachTest()
        {
            var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
            recursionlessSizeBalancedTree.TestMultipleCreationsAndDeletions(recursionlessSizeBal
            ↪  ancedTree.Allocate, recursionlessSizeBalancedTree.Free, ref
            ↪  recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
            ↪  _n);
        }

        [Fact]
        public static void SizeBalancedTreeMultipleAttachAndDetachTest()
        {
            var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
            sizeBalancedTree.TestMultipleCreationsAndDeletions(sizeBalancedTree.Allocate,
            ↪  sizeBalancedTree.Free, ref sizeBalancedTree.Root, () => sizeBalancedTree.Count,
            ↪  _n);
        }

        [Fact]
        public static void SizedAndThreadedAVLBalancedTreeMultipleAttachAndDetachTest()
        {
            var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
            avlTree.TestMultipleCreationsAndDeletions(avlTree.Allocate, avlTree.Free, ref
            ↪  avlTree.Root, () => avlTree.Count, _n);
        }

        [Fact]
        public static void RecursionlessSizeBalancedTreeMultipleRandomAttachAndDetachTest()
        {
            var recursionlessSizeBalancedTree = new RecursionlessSizeBalancedTree<uint>(10000);
            recursionlessSizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref
            ↪  recursionlessSizeBalancedTree.Root, () => recursionlessSizeBalancedTree.Count,
            ↪  _n);
        }

        [Fact]
        public static void SizeBalancedTreeMultipleRandomAttachAndDetachTest()
        {
            var sizeBalancedTree = new SizeBalancedTree<uint>(10000);
            sizeBalancedTree.TestMultipleRandomCreationsAndDeletions(ref sizeBalancedTree.Root,
            ↪  () => sizeBalancedTree.Count, _n);
        }

        [Fact]
        public static void SizedAndThreadedAVLBalancedTreeMultipleRandomAttachAndDetachTest()
        {
            var avlTree = new SizedAndThreadedAVLBalancedTree<uint>(10000);
            avlTree.TestMultipleRandomCreationsAndDeletions(ref avlTree.Root, () =>
            ↪  avlTree.Count, _n);
        }
    }
}
```

Index