Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Институт информационных технологий

Факультет компьютерных технологий

**ЛАБОРАТОРНАЯ РАБОТА №4**

**Тема: Построение В-деревьев и реализация операций с данными**

по учебной дисциплине «Алгоритмы и структуры данных»

Выполнил:

студент гр. 381574

Жгуновский О.Б.

Минск — 2024

**Интерпретация задачи**

Необходимо ввести 10 целых чисел и построить из них B-дерево порядка 5. А также, необходимо реализовать операции поиска, вставки и удаления элементов B-дерева с последующим отображением дерева. Ввод должен производиться как в ручном режиме, так и автоматически. Необходимо учесть, что правильность вводимых данных не гарантируется.

**Код программы**

**using System;**

**using System.Collections;**

**using System.Collections.Generic;**

**namespace CodeExMachina**

**{**

**public class FreeList<T> where T : class**

**{**

**private const int DefaultFreeListSize = 32;**

**private readonly object \_mu;**

**private readonly List<Node<T>> \_freelist;**

**private readonly Comparer<T> \_comparer;**

**public FreeList(Comparer<T> comparer)**

**: this(DefaultFreeListSize, comparer)**

**{ }**

**public FreeList(int size, Comparer<T> comparer)**

**{**

**\_mu = new object();**

**\_freelist = new List<Node<T>>(size);**

**\_comparer = comparer;**

**}**

**internal Node<T> NewNode()**

**{**

**lock (\_mu)**

**{**

**int index = \_freelist.Count - 1;**

**if (index < 0)**

**{**

**return new Node<T>(\_comparer);**

**}**

**Node<T> n = \_freelist[index];**

**\_freelist[index] = null;**

**\_freelist.RemoveAt(index);**

**return n;**

**}**

**}**

**internal bool FreeNode(Node<T> n)**

**{**

**bool success = false;**

**lock (\_mu)**

**{**

**if (\_freelist.Count < \_freelist.Capacity)**

**{**

**\_freelist.Add(n);**

**success = true;**

**}**

**}**

**return success;**

**}**

**}**

**public delegate bool ItemIterator<T>(T i) where T : class;**

**internal class Items<T> : IEnumerable<T> where T : class**

**{**

**private readonly List<T> \_items = new List<T>();**

**private readonly Comparer<T> \_comparer;**

**public int Length => \_items.Count;**

**public int Capacity => \_items.Capacity;**

**public Items(Comparer<T> comparer)**

**{**

**\_comparer = comparer;**

**}**

**public void InsertAt(int index, T item)**

**{**

**\_items.Insert(index, item);**

**}**

**public T RemoveAt(int index)**

**{**

**T item = \_items[index];**

**\_items.RemoveAt(index);**

**return item;**

**}**

**public T Pop()**

**{**

**int index = \_items.Count - 1;**

**T item = \_items[index];**

**\_items[index] = null;**

**\_items.RemoveAt(index);**

**return item;**

**}**

**public void Truncate(int index)**

**{**

**int count = \_items.Count - index;**

**if (count > 0)**

**{**

**\_items.RemoveRange(index, count);**

**}**

**}**

**public (int, bool) Find(T item)**

**{**

**int index = \_items.BinarySearch(0, \_items.Count, item, \_comparer);**

**bool found = index >= 0;**

**if (!found)**

**{**

**index = ~index;**

**}**

**return index > 0 && !Less(\_items[index - 1], item) ? (index - 1, true) : (index, found);**

**}**

**public T this[int i]**

**{**

**get => \_items[i];**

**set => \_items[i] = value;**

**}**

**public void Append(T item)**

**{**

**\_items.Add(item);**

**}**

**public void Append(IEnumerable<T> items)**

**{**

**\_items.AddRange(items);**

**}**

**public List<T> GetRange(int index, int count)**

**{**

**return \_items.GetRange(index, count);**

**}**

**IEnumerator<T> IEnumerable<T>.GetEnumerator()**

**{**

**return \_items.GetEnumerator();**

**}**

**IEnumerator IEnumerable.GetEnumerator()**

**{**

**return \_items.GetEnumerator();**

**}**

**public override string ToString()**

**{**

**return string.Join(" ", \_items);**

**}**

**private bool Less(T x, T y)**

**{**

**return \_comparer.Compare(x, y) == -1;**

**}**

**}**

**internal class Children<T> : IEnumerable<Node<T>> where T : class**

**{**

**private readonly List<Node<T>> \_children = new List<Node<T>>();**

**public int Length => \_children.Count;**

**public int Capacity => \_children.Capacity;**

**public void InsertAt(int index, Node<T> item)**

**{**

**\_children.Insert(index, item);**

**}**

**// back.**

**public Node<T> RemoveAt(int index)**

**{**

**Node<T> n = \_children[index];**

**\_children.RemoveAt(index);**

**return n;**

**}**

**public Node<T> Pop()**

**{**

**int index = \_children.Count - 1;**

**Node<T> child = \_children[index];**

**\_children[index] = null;**

**\_children.RemoveAt(index);**

**return child;**

**}**

**public void Truncate(int index)**

**{**

**int count = \_children.Count - index;**

**if (count > 0)**

**{**

**\_children.RemoveRange(index, count);**

**}**

**}**

**public Node<T> this[int i]**

**{**

**get => \_children[i];**

**set => \_children[i] = value;**

**}**

**public void Append(Node<T> node)**

**{**

**\_children.Add(node);**

**}**

**public void Append(IEnumerable<Node<T>> range)**

**{**

**\_children.AddRange(range);**

**}**

**public List<Node<T>> GetRange(int index, int count)**

**{**

**return \_children.GetRange(index, count);**

**}**

**IEnumerator<Node<T>> IEnumerable<Node<T>>.GetEnumerator()**

**{**

**return \_children.GetEnumerator();**

**}**

**IEnumerator IEnumerable.GetEnumerator()**

**{**

**return \_children.GetEnumerator();**

**}**

**}**

**// Details what item to remove in a node.remove call.**

**internal enum ToRemove**

**{**

**RemoveItem,**

**RemoveMin,**

**RemoveMax**

**}**

**internal enum Direction**

**{**

**Descend = -1,**

**Ascend = 1**

**}**

**internal class Node<T> where T : class**

**{**

**internal Items<T> Items { get; set; }**

**internal Children<T> Children { get; set; }**

**internal CopyOnWriteContext<T> Cow { get; set; }**

**internal Comparer<T> Comparer { get; set; }**

**public Node(Comparer<T> comparer)**

**{**

**Comparer = comparer;**

**Items = new Items<T>(comparer);**

**Children = new Children<T>();**

**}**

**public Node<T> MutableFor(CopyOnWriteContext<T> cow)**

**{**

**if (ReferenceEquals(Cow, cow))**

**{**

**return this;**

**}**

**Node<T> node = Cow.NewNode();**

**node.Items.Append(Items);**

**node.Children.Append(Children);**

**return node;**

**}**

**public Node<T> MutableChild(int i)**

**{**

**Node<T> c = Children[i].MutableFor(Cow);**

**Children[i] = c;**

**return c;**

**}**

**public (T item, Node<T> node) Split(int i)**

**{**

**T item = Items[i];**

**Node<T> next = Cow.NewNode();**

**next.Items.Append(Items.GetRange(i + 1, Items.Length - (i + 1)));**

**Items.Truncate(i);**

**if (Children.Length > 0)**

**{**

**next.Children.Append(Children.GetRange(i + 1, Children.Length - (i + 1)));**

**Children.Truncate(i + 1);**

**}**

**return (item, next);**

**}**

**public bool MaybeSplitChild(int i, int maxItems)**

**{**

**if (Children[i].Items.Length < maxItems)**

**{**

**return false;**

**}**

**Node<T> first = MutableChild(i);**

**(T item, Node<T> second) = first.Split(maxItems / 2);**

**Items.InsertAt(i, item);**

**Children.InsertAt(i + 1, second);**

**return true;**

**}**

**public T Insert(T item, int maxItems)**

**{**

**(int i, bool found) = Items.Find(item);**

**if (found)**

**{**

**T n = Items[i];**

**Items[i] = item;**

**return n;**

**}**

**if (Children.Length == 0)**

**{**

**Items.InsertAt(i, item);**

**return null;**

**}**

**if (MaybeSplitChild(i, maxItems))**

**{**

**T inTree = Items[i];**

**if (Less(item, inTree))**

**{**

**// no change, we want first split node**

**}**

**else if (Less(inTree, item))**

**{**

**i++; // we want second split node**

**}**

**else**

**{**

**T n = Items[i];**

**Items[i] = item;**

**return n;**

**}**

**}**

**return MutableChild(i).Insert(item, maxItems);**

**}**

**public T Get(T key)**

**{**

**(int i, bool found) = Items.Find(key);**

**if (found)**

**{**

**return Items[i];**

**}**

**else if (Children.Length > 0)**

**{**

**return Children[i].Get(key);**

**}**

**return null;**

**}**

**public static T Min(Node<T> n)**

**{**

**if (n == null)**

**{**

**return null;**

**}**

**while (n.Children.Length > 0)**

**{**

**n = n.Children[0];**

**}**

**return n.Items.Length == 0 ? null : n.Items[0];**

**}**

**public static T Max(Node<T> n)**

**{**

**if (n == null)**

**{**

**return null;**

**}**

**while (n.Children.Length > 0)**

**{**

**n = n.Children[n.Children.Length - 1];**

**}**

**return n.Items.Length == 0 ? null : n.Items[n.Items.Length - 1];**

**}**

**public T Remove(T item, int minItems, ToRemove typ)**

**{**

**int i = 0;**

**bool found = false;**

**switch (typ)**

**{**

**case ToRemove.RemoveMax:**

**{**

**if (Children.Length == 0)**

**{**

**return Items.Pop();**

**}**

**i = Items.Length;**

**}**

**break;**

**case ToRemove.RemoveMin:**

**{**

**if (Children.Length == 0)**

**{**

**return Items.RemoveAt(0);**

**}**

**i = 0;**

**}**

**break;**

**case ToRemove.RemoveItem:**

**{**

**(i, found) = Items.Find(item);**

**if (Children.Length == 0)**

**{**

**return found ? Items.RemoveAt(i) : null;**

**}**

**}**

**break;**

**default:**

**Environment.FailFast("invalid type");**

**break;**

**}**

**// If we get to here, we have children.**

**if (Children[i].Items.Length <= minItems)**

**{**

**return GrowChildAndRemove(i, item, minItems, typ);**

**}**

**Node<T> child = MutableChild(i);**

**if (found)**

**{**

**T n = Items[i];**

**Items[i] = child.Remove(null, minItems, ToRemove.RemoveMax);**

**return n;**

**}**

**return child.Remove(item, minItems, typ);**

**}**

**public T GrowChildAndRemove(int i, T item, int minItems, ToRemove typ)**

**{**

**if (i > 0 && Children[i - 1].Items.Length > minItems)**

**{**

**// Steal from left child**

**Node<T> child = MutableChild(i);**

**Node<T> stealFrom = MutableChild(i - 1);**

**T stolenItem = stealFrom.Items.Pop();**

**child.Items.InsertAt(0, Items[i - 1]);**

**Items[i - 1] = stolenItem;**

**if (stealFrom.Children.Length > 0)**

**{**

**child.Children.InsertAt(0, stealFrom.Children.Pop());**

**}**

**}**

**else if (i < Items.Length && Children[i + 1].Items.Length > minItems)**

**{**

**// steal from right child**

**Node<T> child = MutableChild(i);**

**Node<T> stealFrom = MutableChild(i + 1);**

**T stolenItem = stealFrom.Items.RemoveAt(0);**

**child.Items.Append(Items[i]);**

**Items[i] = stolenItem;**

**if (stealFrom.Children.Length > 0)**

**{**

**child.Children.Append(stealFrom.Children.RemoveAt(0));**

**}**

**}**

**else**

**{**

**if (i >= Items.Length)**

**{**

**i--;**

**}**

**Node<T> child = MutableChild(i);**

**// merge with right child**

**T mergeItem = Items.RemoveAt(i);**

**Node<T> mergeChild = Children.RemoveAt(i + 1);**

**child.Items.Append(mergeItem);**

**child.Items.Append(mergeChild.Items);**

**child.Children.Append(mergeChild.Children);**

**\_ = Cow.FreeNode(mergeChild);**

**}**

**return Remove(item, minItems, typ);**

**}**

**public (bool, bool) Iterate(Direction dir, T start, T stop, bool includeStart, bool hit, ItemIterator<T> iter)**

**{**

**bool ok, found;**

**int index = 0;**

**switch (dir)**

**{**

**case Direction.Ascend:**

**{**

**if (start != null)**

**{**

**(index, \_) = Items.Find(start);**

**}**

**for (int i = index; i < Items.Length; i++)**

**{**

**if (Children.Length > 0)**

**{**

**(hit, ok) = Children[i].Iterate(dir, start, stop, includeStart, hit, iter);**

**if (!ok)**

**{**

**return (hit, false);**

**}**

**}**

**if (!includeStart && !hit && start != null && !Less(start, Items[i]))**

**{**

**hit = true;**

**continue;**

**}**

**hit = true;**

**if (stop != null && !Less(Items[i], stop))**

**{**

**return (hit, false);**

**}**

**if (!iter(Items[i]))**

**{**

**return (hit, false);**

**}**

**}**

**if (Children.Length > 0)**

**{**

**(hit, ok) = Children[Children.Length - 1].Iterate(dir, start, stop, includeStart, hit, iter);**

**if (!ok)**

**{**

**return (hit, false);**

**}**

**}**

**}**

**break;**

**case Direction.Descend:**

**{**

**if (start != null)**

**{**

**(index, found) = Items.Find(start);**

**if (!found)**

**{**

**index -= 1;**

**}**

**}**

**else**

**{**

**index = Items.Length - 1;**

**}**

**for (int i = index; i >= 0; i--)**

**{**

**if (start != null && !Less(Items[i], start))**

**{**

**if (!includeStart || hit || Less(start, Items[i]))**

**{**

**continue;**

**}**

**}**

**if (Children.Length > 0)**

**{**

**(hit, ok) = Children[i + 1].Iterate(dir, start, stop, includeStart, hit, iter);**

**if (!ok)**

**{**

**return (hit, false);**

**}**

**}**

**if (stop != null && !Less(stop, Items[i]))**

**{**

**return (hit, false);**

**}**

**hit = true;**

**if (!iter(Items[i]))**

**{**

**return (hit, false);**

**}**

**}**

**if (Children.Length > 0)**

**{**

**(hit, ok) = Children[0].Iterate(dir, start, stop, includeStart, hit, iter);**

**if (!ok)**

**{**

**return (hit, false);**

**}**

**}**

**}**

**break;**

**default:**

**break;**

**}**

**return (hit, true);**

**}**

**public bool Reset(CopyOnWriteContext<T> c)**

**{**

**foreach (Node<T> child in Children)**

**{**

**if (!child.Reset(c))**

**{**

**return false;**

**}**

**}**

**return c.FreeNode(this) != FreeType.ftFreeListFull;**

**}**

**public void Print(System.IO.TextWriter w, int level)**

**{**

**string repeat = new string(' ', level);**

**w.Write($"{repeat}NODE:{Items}\n");**

**foreach (Node<T> c in Children)**

**{**

**c.Print(w, level + 1);**

**}**

**}**

**private bool Less(T x, T y)**

**{**

**return Comparer.Compare(x, y) == -1;**

**}**

**}**

**public class BTree<T> where T : class**

**{**

**private int Degree { get; set; }**

**public int Length { get; private set; }**

**private Node<T> Root { get; set; }**

**private CopyOnWriteContext<T> Cow { get; set; }**

**private BTree()**

**{ }**

**public BTree(int degree, Comparer<T> comparer)**

**: this(degree, new FreeList<T>(comparer))**

**{ }**

**public BTree(int degree, FreeList<T> f)**

**{**

**if (degree <= 1)**

**{**

**Environment.FailFast("bad degree");**

**}**

**Degree = degree;**

**Cow = new CopyOnWriteContext<T> { FreeList = f };**

**}**

**public BTree<T> Clone()**

**{**

**CopyOnWriteContext<T> cow1 = new CopyOnWriteContext<T> { FreeList = Cow.FreeList };**

**CopyOnWriteContext<T> cow2 = new CopyOnWriteContext<T> { FreeList = Cow.FreeList };**

**BTree<T> tree = new BTree<T>**

**{**

**Degree = Degree,**

**Length = Length,**

**Root = Root,**

**Cow = Cow**

**};**

**Cow = cow1;**

**tree.Cow = cow2;**

**return tree;**

**}**

**private int MaxItems()**

**{**

**return (Degree \* 2) - 1;**

**}**

**private int MinItems()**

**{**

**return Degree - 1;**

**}**

**public T ReplaceOrInsert(T item)**

**{**

**if (item == null)**

**{**

**Environment.FailFast("null item being added to BTree");**

**}**

**if (Root == null)**

**{**

**Root = Cow.NewNode();**

**Root.Items.Append(item);**

**Length++;**

**return null;**

**}**

**else**

**{**

**Root = Root.MutableFor(Cow);**

**if (Root.Items.Length >= MaxItems())**

**{**

**(T item2, Node<T> second) = Root.Split(MaxItems() / 2);**

**Node<T> oldRoot = Root;**

**Root = Cow.NewNode();**

**Root.Items.Append(item2);**

**Root.Children.Append(oldRoot);**

**Root.Children.Append(second);**

**}**

**}**

**T result = Root.Insert(item, MaxItems());**

**if (result == null)**

**{**

**Length++;**

**}**

**return result;**

**}**

**public T Delete(T item)**

**{**

**return DeleteItem(item, ToRemove.RemoveItem);**

**}**

**public T DeleteMin()**

**{**

**return DeleteItem(null, ToRemove.RemoveMin);**

**}**

**public T DeleteMax()**

**{**

**return DeleteItem(null, ToRemove.RemoveMax);**

**}**

**private T DeleteItem(T item, ToRemove typ)**

**{**

**if (Root == null || Root.Items.Length == 0)**

**{**

**return null;**

**}**

**Root = Root.MutableFor(Cow);**

**T result = Root.Remove(item, MinItems(), typ);**

**if (Root.Items.Length == 0 && Root.Children.Length > 0)**

**{**

**Node<T> oldRoot = Root;**

**Root = Root.Children[0];**

**\_ = Cow.FreeNode(oldRoot);**

**}**

**if (result != null)**

**{**

**Length--;**

**}**

**return result;**

**}**

**public void AscendRange(T greaterOrEqual, T lessThan, ItemIterator<T> iterator)**

**{**

**if (Root == null)**

**{**

**return;**

**}**

**\_ = Root.Iterate(Direction.Ascend, greaterOrEqual, lessThan, true, false, iterator);**

**}**

**public void AscendLessThan(T pivot, ItemIterator<T> iterator)**

**{**

**if (Root == null)**

**{**

**return;**

**}**

**\_ = Root.Iterate(Direction.Ascend, null, pivot, false, false, iterator);**

**}**

**public void AscendGreaterOrEqual(T pivot, ItemIterator<T> iterator)**

**{**

**if (Root == null)**

**{**

**return;**

**}**

**\_ = Root.Iterate(Direction.Ascend, pivot, null, true, false, iterator);**

**}**

**public void Ascend(ItemIterator<T> iterator)**

**{**

**if (Root == null)**

**{**

**return;**

**}**

**\_ = Root.Iterate(Direction.Ascend, null, null, false, false, iterator);**

**}**

**public void DescendRange(T lessOrEqual, T greaterThan, ItemIterator<T> iterator)**

**{**

**if (Root == null)**

**{**

**return;**

**}**

**\_ = Root.Iterate(Direction.Descend, lessOrEqual, greaterThan, true, false, iterator);**

**}**

**public void DescendLessOrEqual(T pivot, ItemIterator<T> iterator)**

**{**

**if (Root == null)**

**{**

**return;**

**}**

**\_ = Root.Iterate(Direction.Descend, pivot, null, true, false, iterator);**

**}**

**/// <param name="iterator"></param>**

**public void DescendGreaterThan(T pivot, ItemIterator<T> iterator)**

**{**

**if (Root == null)**

**{**

**return;**

**}**

**\_ = Root.Iterate(Direction.Descend, null, pivot, false, false, iterator);**

**}**

**public void Descend(ItemIterator<T> iterator)**

**{**

**if (Root == null)**

**{**

**return;**

**}**

**\_ = Root.Iterate(Direction.Descend, null, null, false, false, iterator);**

**}**

**public T Get(T key)**

**{**

**return Root?.Get(key);**

**}**

**public T Min()**

**{**

**return Node<T>.Min(Root);**

**}**

**public T Max()**

**{**

**return Node<T>.Max(Root);**

**}**

**public bool Has(T key)**

**{**

**return Get(key) != null;**

**}**

**public void Clear(bool addNodesToFreeList)**

**{**

**if (Root != null && addNodesToFreeList)**

**{**

**\_ = Root.Reset(Cow);**

**}**

**Root = null;**

**Length = 0;**

**}**

**}**

**internal enum FreeType**

**{**

**ftFreeListFull,**

**ftStored,**

**ftNotOwned**

**}**

**internal class CopyOnWriteContext<T> where T : class**

**{**

**public FreeList<T> FreeList { get; internal set; }**

**public Node<T> NewNode()**

**{**

**Node<T> n = FreeList.NewNode();**

**n.Cow = this;**

**return n;**

**}**

**public FreeType FreeNode(Node<T> n)**

**{**

**if (ReferenceEquals(n.Cow, this))**

**{**

**// clear to allow GC**

**n.Items.Truncate(0);**

**n.Children.Truncate(0);**

**n.Cow = null;**

**return FreeList.FreeNode(n) ? FreeType.ftStored : FreeType.ftFreeListFull;**

**}**

**else**

**{**

**return FreeType.ftNotOwned;**

**}**

**}**

**}**

**public class Int : IComparable, IComparable<int>**

**{**

**private readonly int \_v;**

**public Int(int v)**

**{**

**\_v = v;**

**}**

**public override string ToString()**

**{**

**return \_v.ToString();**

**}**

**public int CompareTo(int other)**

**{**

**return \_v.CompareTo(other);**

**}**

**public int CompareTo(object obj)**

**{**

**Int v = (Int)obj;**

**return \_v.CompareTo(v.\_v);**

**}**

**public override bool Equals(object obj)**

**{**

**return Equals(obj as Int);**

**}**

**public override int GetHashCode()**

**{**

**return \_v.GetHashCode();**

**}**

**public bool Equals(Int other)**

**{**

**if (other is null) return false;**

**return \_v == other.\_v;**

**}**

**public static bool operator <(Int a, Int b)**

**{**

**return a.\_v < b.\_v;**

**}**

**public static bool operator >(Int a, Int b)**

**{**

**return a.\_v > b.\_v;**

**}**

**public static bool operator !=(Int a, Int b)**

**{**

**return !(a == b);**

**}**

**public static bool operator ==(Int a, Int b)**

**{**

**return object.Equals(a, b);**

**}**

**}**

**public class IntComparer : Comparer<Int>**

**{**

**public override int Compare(Int x, Int y)**

**{**

**return x.CompareTo(y);**

**}**

**}**

**class Program**

**{**

**static void Main(string[] args)**

**{**

**BTree<Int> btree = new BTree<Int>(5, new IntComparer()); // создаем B-дерево порядка 5**

**bool exit = false;**

**while (!exit)**

**{**

**Console.Clear();**

**Console.WriteLine("Меню операций с B-деревом:");**

**Console.WriteLine("1) Добавить элемент");**

**Console.WriteLine("2) Удалить элемент");**

**Console.WriteLine("3) Найти элемент");**

**Console.WriteLine("4) Показать все элементы по возрастанию");**

**Console.WriteLine("5) Автоматический ввод нескольких элементов");**

**Console.WriteLine("0) Выход");**

**Console.Write("Выберите действие: ");**

**if (!int.TryParse(Console.ReadLine(), out int choice))**

**{**

**Console.WriteLine("Ошибка: Введите корректный номер операции.");**

**Console.ReadKey();**

**continue;**

**}**

**switch (choice)**

**{**

**case 1:**

**AddElement(btree);**

**break;**

**case 2:**

**RemoveElement(btree);**

**break;**

**case 3:**

**FindElement(btree);**

**break;**

**case 4:**

**ShowAllElements(btree);**

**break;**

**case 5:**

**AutoInsertElements(btree);**

**break;**

**case 0:**

**exit = true;**

**break;**

**default:**

**Console.WriteLine("Ошибка: выберите номер операции из списка.");**

**Console.ReadKey();**

**break;**

**}**

**}**

**}**

**static void AddElement(BTree<Int> btree)**

**{**

**Console.Write("Введите значение для добавления: ");**

**if (int.TryParse(Console.ReadLine(), out int value))**

**{**

**btree.ReplaceOrInsert(new Int(value));**

**Console.WriteLine($"Элемент {value} добавлен.");**

**}**

**else**

**{**

**Console.WriteLine("Ошибка: введите корректное целое число.");**

**}**

**Console.ReadKey();**

**}**

**static void RemoveElement(BTree<Int> btree)**

**{**

**Console.Write("Введите значение для удаления: ");**

**if (int.TryParse(Console.ReadLine(), out int value))**

**{**

**var removed = btree.Delete(new Int(value));**

**if (removed != null)**

**{**

**Console.WriteLine($"Элемент {value} удален.");**

**}**

**else**

**{**

**Console.WriteLine($"Элемент {value} не найден в дереве.");**

**}**

**}**

**else**

**{**

**Console.WriteLine("Ошибка: введите корректное целое число.");**

**}**

**Console.ReadKey();**

**}**

**static void FindElement(BTree<Int> btree)**

**{**

**Console.Write("Введите значение для поиска: ");**

**if (int.TryParse(Console.ReadLine(), out int value))**

**{**

**var found = btree.Get(new Int(value));**

**if (found != null)**

**{**

**Console.WriteLine($"Элемент {value} найден.");**

**}**

**else**

**{**

**Console.WriteLine($"Элемент {value} не найден в дереве.");**

**}**

**}**

**else**

**{**

**Console.WriteLine("Ошибка: введите корректное целое число.");**

**}**

**Console.ReadKey();**

**}**

**static void ShowAllElements(BTree<Int> btree)**

**{**

**Console.WriteLine("Элементы дерева по возрастанию:");**

**btree.Ascend(item => {**

**Console.Write($"{item} ");**

**return true; // возвращаем true, чтобы продолжить обход**

**});**

**Console.WriteLine();**

**Console.ReadKey();**

**}**

**static void AutoInsertElements(BTree<Int> btree)**

**{**

**int[] values = { 15, 10, 20, 5, 13, 17, 25 };**

**foreach (var value in values)**

**{**

**btree.ReplaceOrInsert(new Int(value));**

**}**

**Console.WriteLine("Элементы [15, 10, 20, 5, 13, 17, 25] добавлены автоматически.");**

**Console.ReadKey();**

**}**

**}**

**}**

**Демонстрация работы программы**

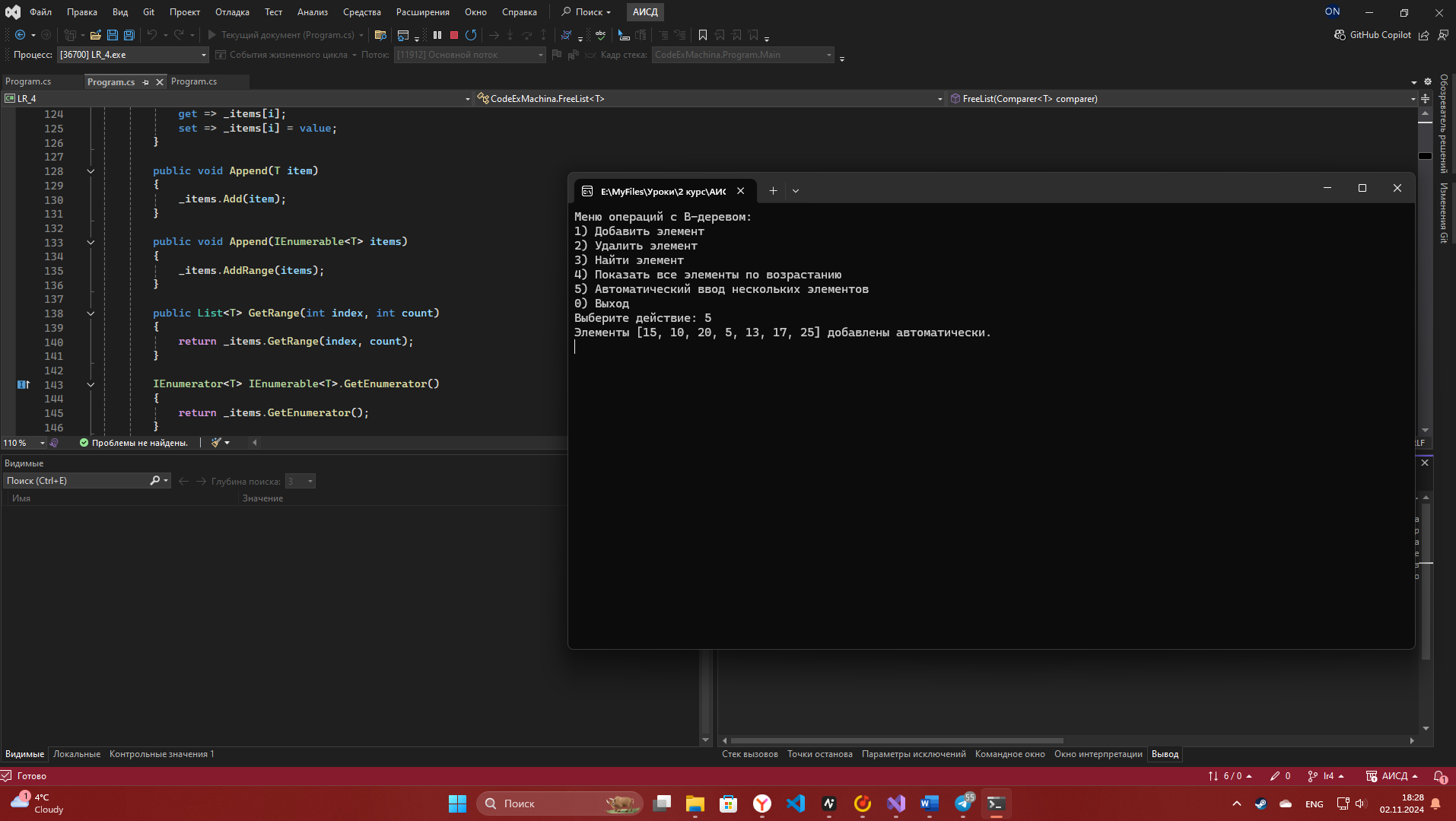


Рисунок 1 – автоматическое заполнение

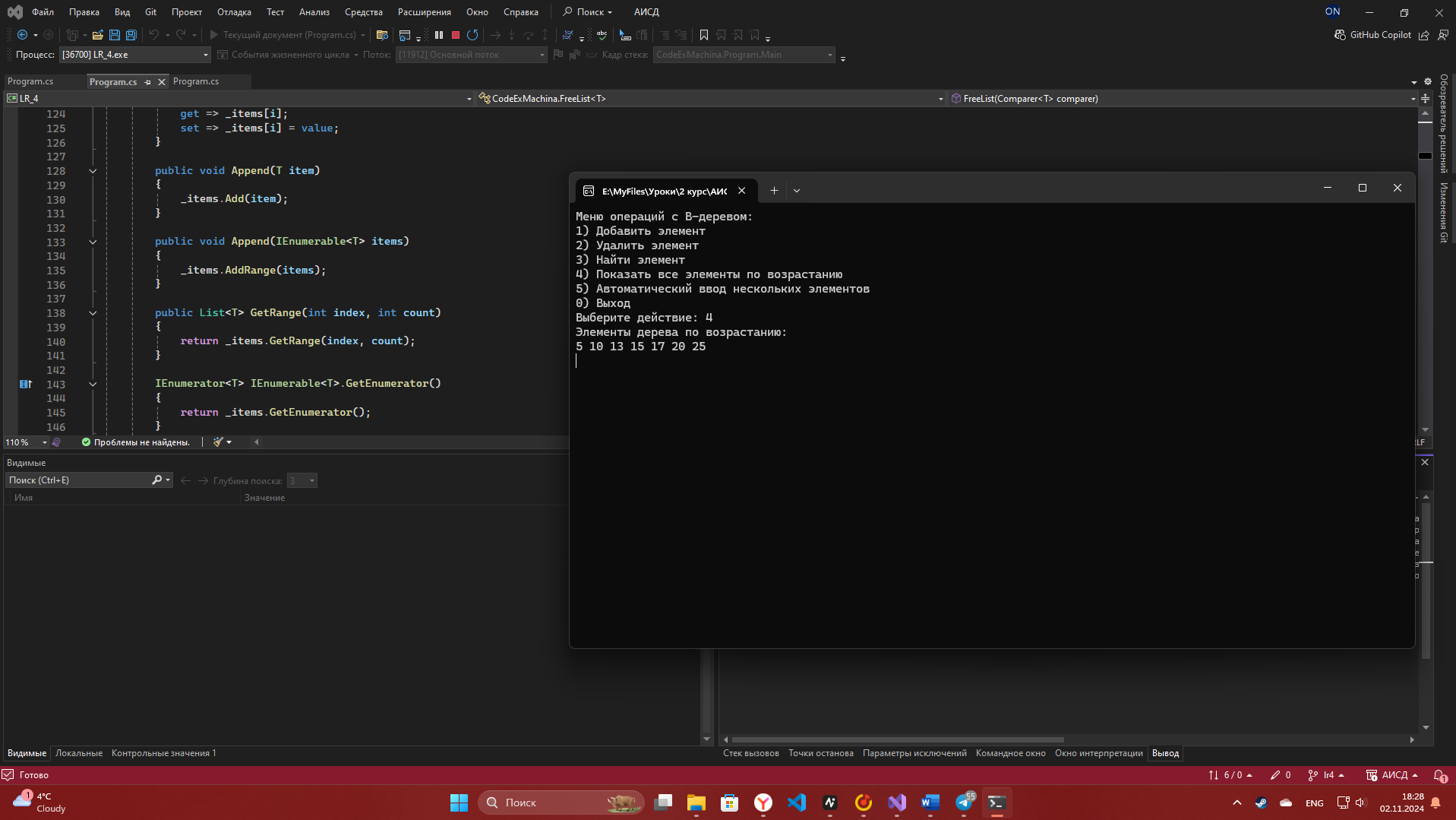


Рисунок 2 – демонстрация элементов

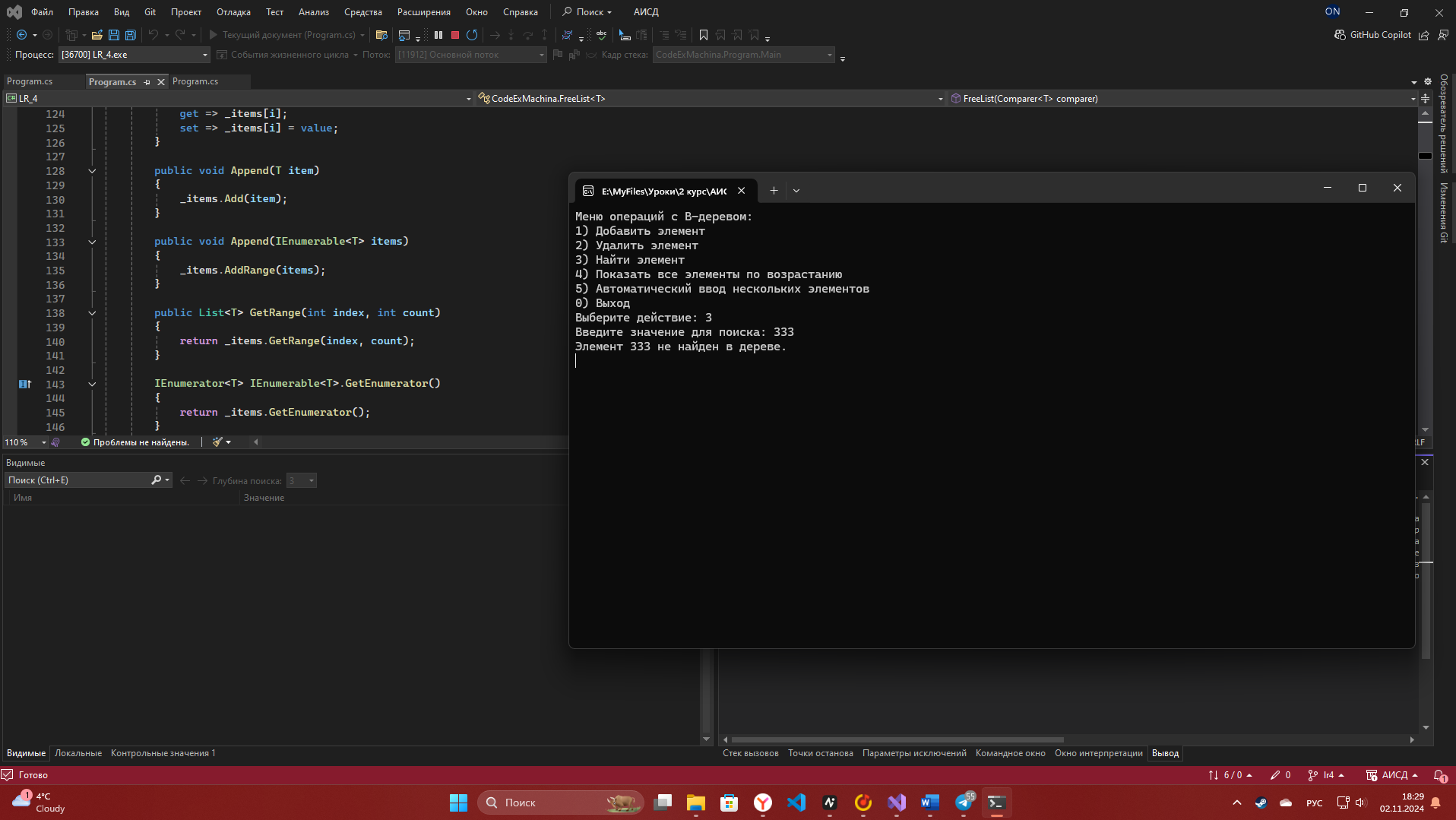


Рисунок 3 – неудачный поиск

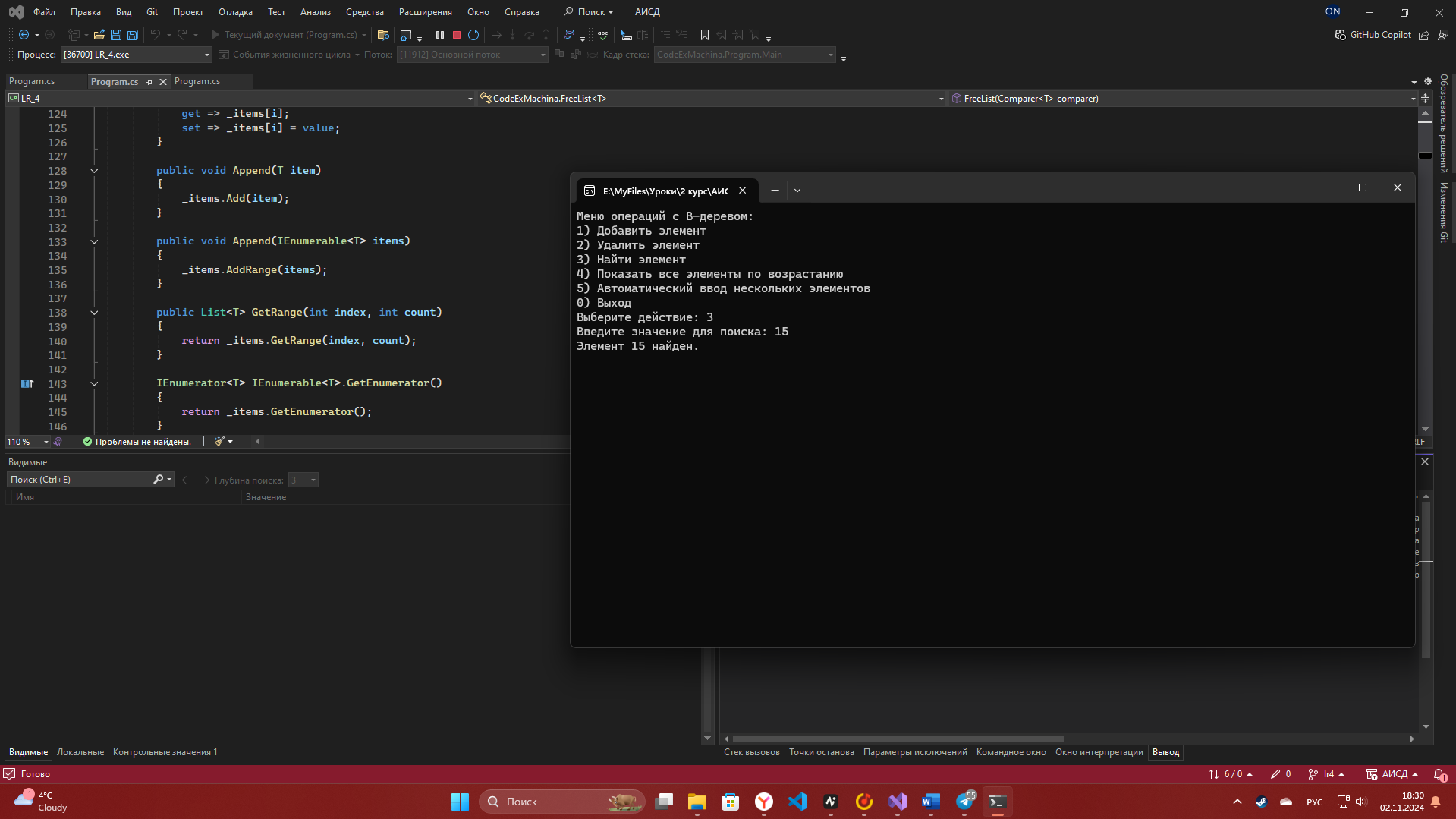


Рисунок 4 – удачный поиск

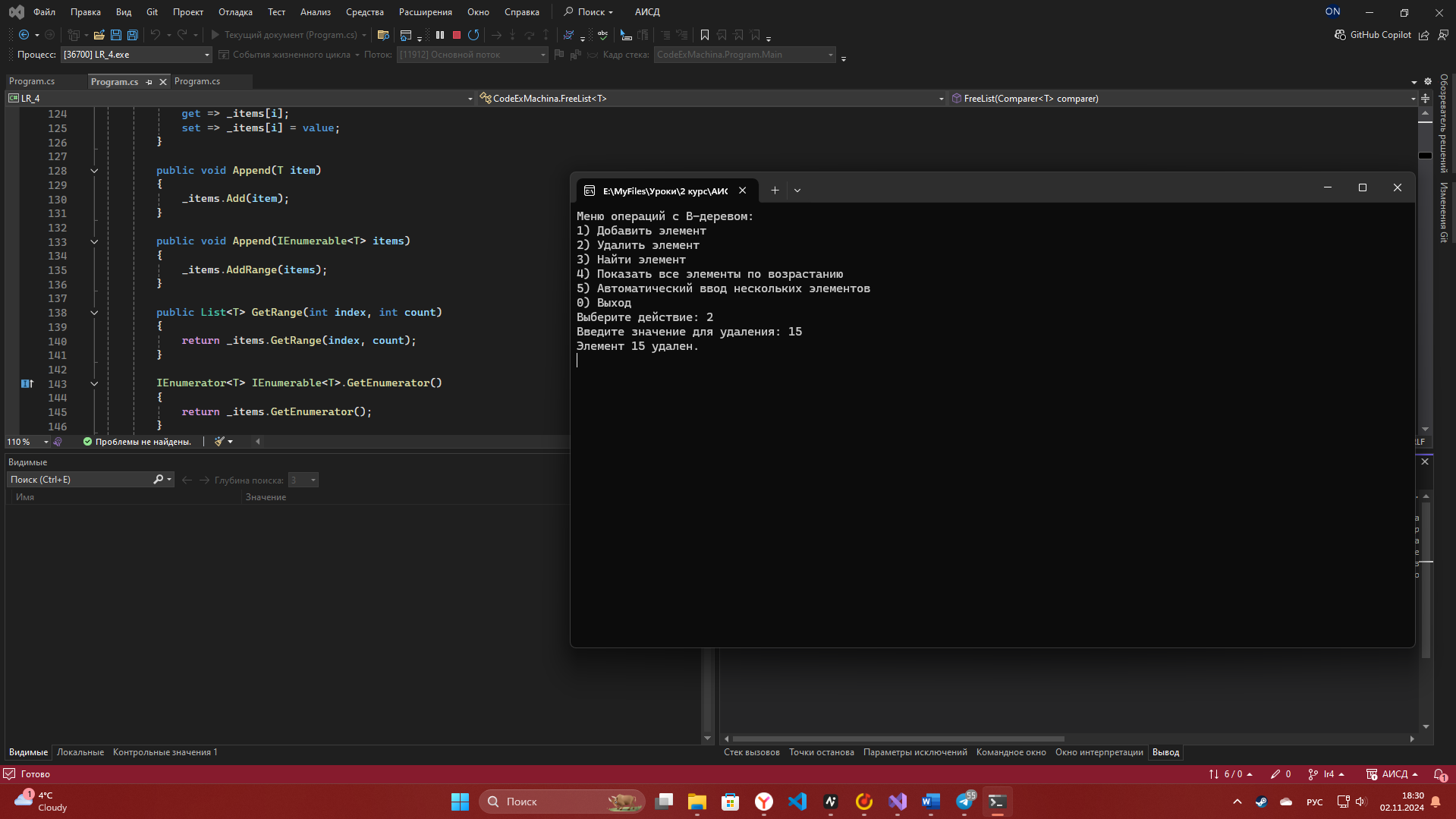


Рисунок 5 – успешное удаление

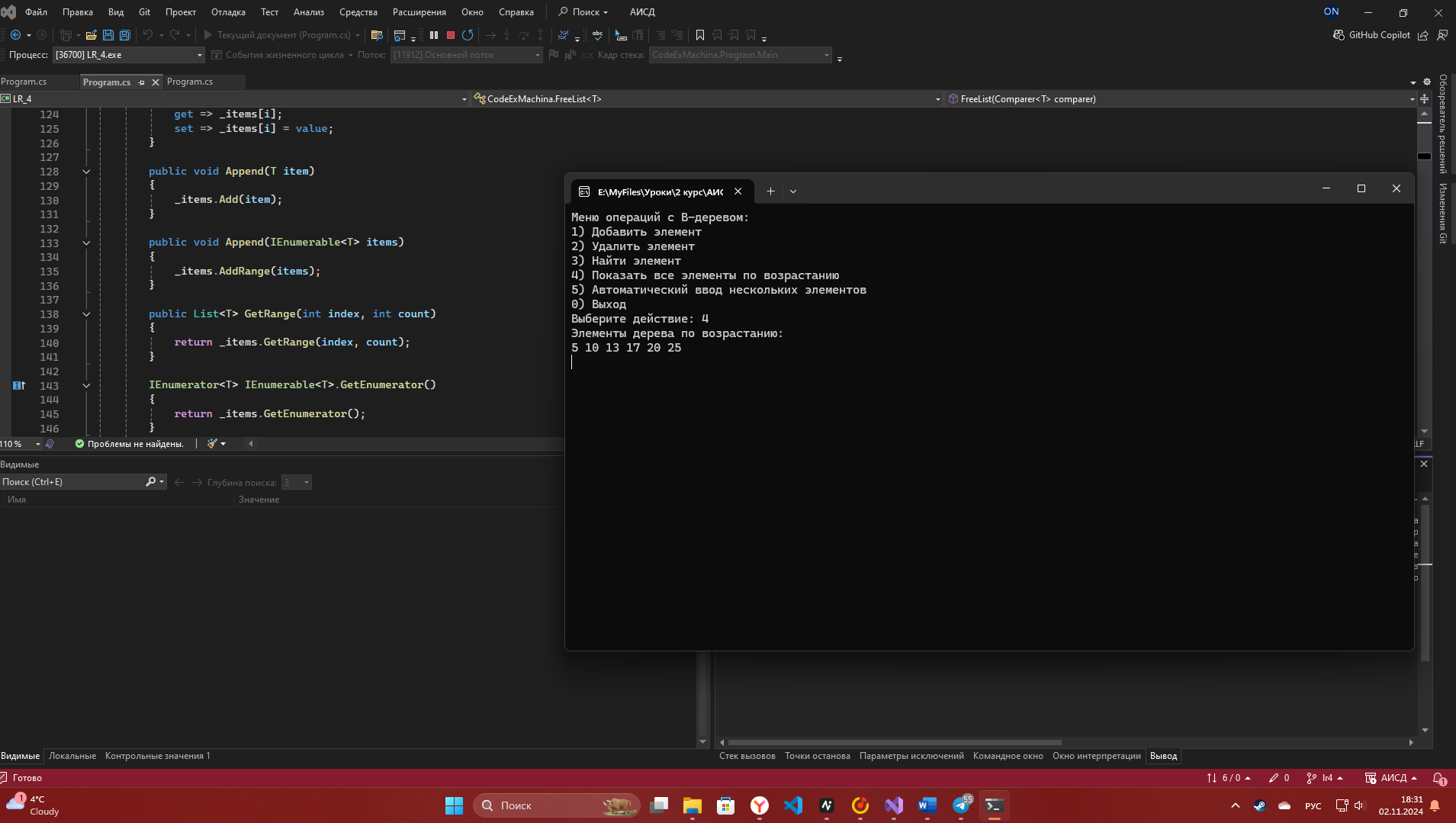


Рисунок 6 – вывод после удаления

**Контрольные вопросы**

*Чем B-дерево отличается от бинарного дерева?*

В бинарном дереве каждый узел имеет не более двух дочерних узлов. В свою очередь, количество узлов у B-дерева зависит от его ранга.

*На основании чего выбирается арность B-дерева?*

Вообще – по размеру блока памяти. Но чем больше делаешь арность, тем дерево быстрее работает (меньше глубина дерева)

*Почему удается ускорить обработку данных при использовании B-*

*деревьев?*

У В-деревьев равномерное время доступа, каждый узел может содержать большое количество элементов (сокращает количество операций)