МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра САПР

ОТЧЕТ

по лабораторной работе № 1

по дисциплине «Алгоритмы и структуры данных»

ТЕМА: АССОЦИАТИВНЫЙ МАССИВ. Вариант: 1

Студент гр. 0302 _____ Савенко Н.С

Преподаватель _____ Тутуева А.В.

Санкт-Петербург

2021

Постановка задачи

Реализовать шаблонный ассоциативный массив (map) на основе красно-черного дерева.

Описание реализуемых алгоритмов

1. insert(ключ, значение)
    a. Удаляем элемент из дерева внутри Map
    b. При вставке новый узел всегда вставляется как красный узел, поскольку это не нарушает черную высоту дерева

    c. После вставки нового узла, если у дерева нарушены свойства красно-черного дерева, выполняются перекрашивание и повороты
2. remove(ключ)
   Удаляем элемент из дерева внутри Map при этом учитываем, что удаление узла может нарушить или не нарушить красно-черные свойства дерева,
   если это произошло, то используется алгоритм восстановления свойств красно-черного дерева
3. find(ключ)
   Проверяем существует ли элемент с таким ключом, находим его в дереве внутри Map, иначе возвращаем null
4. clear
   Заменяем дерево внутри Map на пустое
5. get_keys
   Возвращаем список ключей из дерева внутри Map
6. get_values
   Возвращаем список значений из дерева внутри Map
7. print
   Реализовано в методе ToString(). Производим базовую сериализацию элементов и ключей дерева.

Оценка временной сложности

1. insert
   O(log2(n))
2. remove
   O(log2(n))
3. find
   O(log2(n))
4. clear
   O(1)
5. get_keys

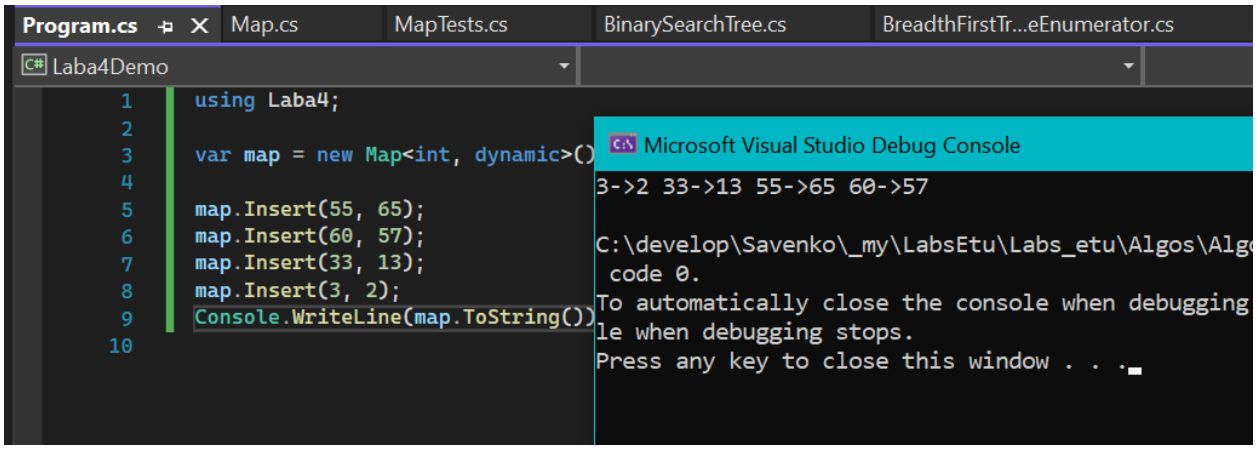
O(n)

6. get_values
   O(n)
7. print
   O(n)

Описание Unit тестов

Во всех тестах создается Map, затем выполняются нужные операции над ним. Проверяется верность элементов после операций.

Примеры работы



Листинг

```
namespace Laba4.RedBlackTree;

internal class RedBlackNode<TKey, TValue>
    where TValue : class
    where TKey : IComparable, IComparable<TKey>, IEquatable<TKey>
{
    public TValue Data { get; set; }

    public TKey Key { get; set; }

    internal NodeColor Color { get; set; }

    public RedBlackNode<TKey, TValue> Left { get; set; }

    public RedBlackNode<TKey, TValue> Right { get; set; }

    public RedBlackNode<TKey, TValue> Parent { get; set; }

    public RedBlackNode()
    {
        Color = NodeColor.Red;

        Right = RedBlackTree<TKey, TValue>.NullObjectNode;
```

```
            Left = RedBlackTree<TKey, TValue>.NullObjectNode;
        }

        public RedBlackNode(TKey key, TValue data)
            : this()
        {
            Key = key;
            Data = data;
        }
}
```

```
using System.Text;

namespace Laba4.RedBlackTree;

public class RedBlackTree<TKey, TValue> where TValue : class
    where TKey : IComparable<TKey>, IComparable, IEquatable<TKey>
{
    private RedBlackNode<TKey, TValue> _baseNode = NullObjectNode;
    private RedBlackNode<TKey, TValue> _lastNodeFound = NullObjectNode;

    internal static readonly RedBlackNode<TKey, TValue> NullObjectNode =
        new() { Left = null, Right = null, Parent = null, Color = NodeColor.Black };

    private int _count;

    public RedBlackTree()
    {
        _count = 0;
    }

    public TValue GetData(TKey key)
    {
        return GetNode(key).Data;
    }

    public void Clear()
    {
        _baseNode = NullObjectNode;
        _count = 0;
    }

    public override bool Equals(object? obj)
    {
        if (obj == null) return false;

        if (!(obj is RedBlackNode<TKey, TValue>)) return false;

        return this == obj || ToString().Equals(obj.ToString());
    }

    public override string ToString()
    {
        var sb = new StringBuilder();
        foreach (var key in Keys) sb.Append($"{key}->{GetData(key)} ");
```

```
    return sb.ToString();
}

public virtual void Add(TKey key, TValue value)
{
    New(key, value);
}

public virtual bool TryRemove(TKey key)
{
    try
    {
        Delete(GetNode(key));
        return true;
    }
    catch (Exception)
    {
        return false;
    }
}

public ICollection<TKey> Keys
{
    get { return GetAll().Select(i => i.Key).ToArray(); }
}

public ICollection<TValue> Values
{
    get { return GetAll().Select(i => i.Data).ToArray(); }
}

private void New(TKey key, TValue data)
{
    if (data == null) throw new Exception("Key and data must not be null");

    var newNode = new RedBlackNode<TKey, TValue>(key, data);

    var workNode = _baseNode;

    while (workNode != NullObjectNode)
    {
        newNode.Parent = workNode;
        var result = key.CompareTo(workNode.Key);
        if (result == 0) throw new Exception("Node with same key already exists");
        workNode = result > 0 ? workNode.Right : workNode.Left;
    }

    if (newNode.Parent != null)
    {
        if (newNode.Key.CompareTo(newNode.Parent.Key) > 0)
            newNode.Parent.Right = newNode;
```

```
            else
                newNode.Parent.Left = newNode;
        }
        else
        {
            _baseNode = newNode;
        }

        BalanceTreeAfterInsert(newNode);

        _lastNodeFound = newNode;

        Interlocked.Increment(ref _count);
    }

    private void Delete(RedBlackNode<TKey, TValue> deleteNode)
    {
        RedBlackNode<TKey, TValue> workNode;

        if (deleteNode.Left == NullObjectNode || deleteNode.Right == NullObjectNode)
        {
            workNode = deleteNode;
        }
        else
        {
            workNode = deleteNode.Right;

            while (workNode.Left != NullObjectNode) workNode = workNode.Left;
        }

        var linkedNode = workNode.Left != NullObjectNode ? workNode.Left : workNode.Right;

        linkedNode.Parent = workNode.Parent;
        if (workNode.Parent != null)
            if (workNode == workNode.Parent.Left)
                workNode.Parent.Left = linkedNode;
            else
                workNode.Parent.Right = linkedNode;
        else
            _baseNode = linkedNode;

        if (workNode != deleteNode)
        {
            deleteNode.Key = workNode.Key;
            deleteNode.Data = workNode.Data;
        }

        if (workNode.Color == NodeColor.Black) BalanceTreeAfterDelete(linkedNode);

        _lastNodeFound = NullObjectNode;

        Interlocked.Decrement(ref _count);
```

```
    }

    private void BalanceTreeAfterDelete(RedBlackNode<TKey, TValue> linkedNode)
    {
        while (linkedNode != _baseNode && linkedNode.Color == NodeColor.Black)
        {
            RedBlackNode<TKey, TValue> workNode;

            if (linkedNode == linkedNode.Parent.Left)
            {
                workNode = linkedNode.Parent.Right;
                if (workNode.Color == NodeColor.Red)
                {
                    linkedNode.Parent.Color = NodeColor.Red;
                    workNode.Color = NodeColor.Black;
                    RotateLeft(linkedNode.Parent);
                    workNode = linkedNode.Parent.Right;
                }

                if (workNode.Left.Color == NodeColor.Black && workNode.Right.Color ==
NodeColor.Black)
                {
                    workNode.Color = NodeColor.Red;

                    linkedNode = linkedNode.Parent;
                }
                else
                {
                    if (workNode.Right.Color == NodeColor.Black)
                    {
                        workNode.Left.Color = NodeColor.Black;
                        workNode.Color = NodeColor.Red;
                        RotateRight(workNode);
                        workNode = linkedNode.Parent.Right;
                    }

                    linkedNode.Parent.Color = NodeColor.Black;
                    workNode.Color = linkedNode.Parent.Color;
                    workNode.Right.Color = NodeColor.Black;
                    RotateLeft(linkedNode.Parent);
                    linkedNode = _baseNode;
                }
            }
            else
            {
                workNode = linkedNode.Parent.Left;
                if (workNode.Color == NodeColor.Red)
                {
                    linkedNode.Parent.Color = NodeColor.Red;
                    workNode.Color = NodeColor.Black;
                    RotateRight(linkedNode.Parent);
                    workNode = linkedNode.Parent.Left;
```

```
            }

            if (workNode.Right.Color == NodeColor.Black && workNode.Left.Color ==
NodeColor.Black)
            {
               workNode.Color = NodeColor.Red;
               linkedNode = linkedNode.Parent;
            }
            else
            {
               if (workNode.Left.Color == NodeColor.Black)
               {
                  workNode.Right.Color = NodeColor.Black;
                  workNode.Color = NodeColor.Red;
                  RotateLeft(workNode);
                  workNode = linkedNode.Parent.Left;
               }

               workNode.Color = linkedNode.Parent.Color;
               linkedNode.Parent.Color = NodeColor.Black;
               workNode.Left.Color = NodeColor.Black;
               RotateRight(linkedNode.Parent);
               linkedNode = _baseNode;
            }
         }
      }

      linkedNode.Color = NodeColor.Black;
   }

   internal Stack<RedBlackNode<TKey, TValue>> GetAll()
   {
      var stack = new Stack<RedBlackNode<TKey, TValue>>();

      if (_baseNode != NullObjectNode) WalkNextLevel(_baseNode, stack);

      return stack;
   }

   private static void WalkNextLevel(RedBlackNode<TKey, TValue> node,
Stack<RedBlackNode<TKey, TValue>> stack)
   {
      if (node.Right != NullObjectNode) WalkNextLevel(node.Right, stack);
      stack.Push(node);
      if (node.Left != NullObjectNode) WalkNextLevel(node.Left, stack);
   }

   private RedBlackNode<TKey, TValue> GetNode(TKey key)
   {
      int result;
      if (_lastNodeFound != NullObjectNode)
      {
```

```
        result = key.CompareTo(_lastNodeFound.Key);
        if (result == 0) return _lastNodeFound;
    }

    var treeNode = _baseNode;

    while (treeNode != NullObjectNode)
    {
        result = key.CompareTo(treeNode.Key);
        if (result == 0)
        {
            _lastNodeFound = treeNode;
            return treeNode;
        }

        treeNode = result < 0 ? treeNode.Left : treeNode.Right;
    }

    return null;
}

private void RotateRight(RedBlackNode<TKey, TValue> rotateNode)
{
    var workNode = rotateNode.Left;

    rotateNode.Left = workNode.Right;

    if (workNode.Right != NullObjectNode) workNode.Right.Parent = rotateNode;

    if (workNode != NullObjectNode) workNode.Parent = rotateNode.Parent;

    if (rotateNode.Parent != null)
    {
        if (rotateNode == rotateNode.Parent.Right)
            rotateNode.Parent.Right = workNode;
        else
            rotateNode.Parent.Left = workNode;
    }
    else
    {
        _baseNode = workNode;
    }

    workNode.Right = rotateNode;

    if (rotateNode != NullObjectNode) rotateNode.Parent = workNode;
}

private void RotateLeft(RedBlackNode<TKey, TValue> rotateNode)
{
    var workNode = rotateNode.Right;
```

```
    rotateNode.Right = workNode.Left;

    if (workNode.Left != NullObjectNode) workNode.Left.Parent = rotateNode;

    if (workNode != NullObjectNode) workNode.Parent = rotateNode.Parent;

    if (rotateNode.Parent != null)
    {
        if (rotateNode == rotateNode.Parent.Left)
            rotateNode.Parent.Left = workNode;
        else
            rotateNode.Parent.Right = workNode;
    }
    else
    {
        _baseNode = workNode;
    }

    workNode.Left = rotateNode;

    if (rotateNode != NullObjectNode) rotateNode.Parent = workNode;
}

private void BalanceTreeAfterInsert(RedBlackNode<TKey, TValue> insertedNode)
{
    while (insertedNode != _baseNode && insertedNode.Parent.Color == NodeColor.Red)
    {
        RedBlackNode<TKey, TValue> workNode;
        if (insertedNode.Parent == insertedNode.Parent.Parent.Left)
        {
            workNode = insertedNode.Parent.Parent.Right;
            if (workNode != null && workNode.Color == NodeColor.Red)
            {
                insertedNode.Parent.Color = NodeColor.Black;
                workNode.Color = NodeColor.Black;

                insertedNode.Parent.Parent.Color = NodeColor.Red;
                insertedNode = insertedNode.Parent.Parent;
            }
            else
            {
                if (insertedNode == insertedNode.Parent.Right)
                {
                    insertedNode = insertedNode.Parent;
                    RotateLeft(insertedNode);
                }

                insertedNode.Parent.Color = NodeColor.Black;
                insertedNode.Parent.Parent.Color = NodeColor.Red;
                RotateRight(insertedNode.Parent.Parent);
            }
        }
```

```
            else
            {
                workNode = insertedNode.Parent.Parent.Left;
                if (workNode != null && workNode.Color == NodeColor.Red)
                {
                    insertedNode.Parent.Color = NodeColor.Black;
                    workNode.Color = NodeColor.Black;
                    insertedNode.Parent.Parent.Color = NodeColor.Red;
                    insertedNode = insertedNode.Parent.Parent;
                }
                else
                {
                    if (insertedNode == insertedNode.Parent.Left)
                    {
                        insertedNode = insertedNode.Parent;
                        RotateRight(insertedNode);
                    }

                    insertedNode.Parent.Color = NodeColor.Black;
                    insertedNode.Parent.Parent.Color = NodeColor.Red;
                    RotateLeft(insertedNode.Parent.Parent);
                }
            }
        }

        _baseNode.Color = NodeColor.Black;
    }
}
```

```
using Laba4.RedBlackTree;

namespace Laba4;

public class Map<TKey, TValue> where TValue : class where TKey : IComparable<TKey>,
IComparable, IEquatable<TKey>
{
    private RedBlackTree<TKey, TValue> _tree { get; set; }

    // Works like get_keys
    // .NET Developers prefer explicit getters instead of old get methods
    public IEnumerable<TKey> Keys
    {
        get => _tree.Keys;
    }

    // Works like get_values
    // .NET Developers prefer explicit getters instead of old get methods
    public IEnumerable<TValue> Values
    {
        get => _tree.Values;
    }

    public Map()
    {
        _tree = new RedBlackTree<TKey, TValue>();
    }

    public void Insert(TKey key, TValue value)
```

```
    {
        _tree.Add(key, value);
    }

    public void Remove(TKey key)
    {
        _tree.TryRemove(key);
    }

    public TValue? Find(TKey key)
    {
        return _tree.Keys.Contains(key) ? _tree.GetData(key) : null;
    }

    public void Clear()
    {
        _tree.Clear();
    }

    // Works like "print()"
    // Please use ToString for map serialization and printing
    public override string ToString()
    {
        return _tree.ToString();
    }
}
```