

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

ОТЧЕТ
по лабораторной работе № 3
по дисциплине «Алгоритмы и структуры данных»
ТЕМА: ДВОИЧНЫЕ ДЕРЕВЬЯ. Вариант: 1

Студент гр. 0302

Савенко Н.С

Преподаватель

Тутуева А.В.

Санкт-Петербург

2021

Постановка задачи

Задачей является реализация алгоритмов двоичного дерева поиска.

Описание реализуемых алгоритмов

Contains

Идем вниз по дереву сравнивая искомое значение со значением узла и в зависимости от этого идем влево или вправо пока не найдем нужный узел

Создание итератора обхода в ширину

Возвращаем новый экземпляр итератора обхода в ширину

Insert

Сравниваем значение вставляемого элемента с корневым узлом

Если значение меньше или равно, то вставляемый элемент должен быть вставлен в левое поддерево

Если значение больше, то вставляемый элемент должен быть вставлен в правое поддерево

Выбираем нужное поддерево. В нем повторяем шаг 1 пока не дойдем до листового элемента и не добавим вставляемый элемент как новый лист

Создание итератора обхода вглубь

Возвращаем новый экземпляр итератора обхода вглубь

Remove

Если удаляемый элемент – это листовой узел

Просто удаляем лист

Если удаляемый узел имеет единственный дочерний узел

Заменяем этот узел его дочерним узлом

Удаляем дочерний узел из его исходного положения

Если удаляемый узел имеет двух потомков

Находим в правом поддереве относительно удаляемого узла самый левый листовой узел. В нем будет храниться наименьшее значение из этого поддерева, но которое больше корневого узла

Заменяем удаляемый узел на значение из листового узла

Удаляем листовой узел из его исходного положения

То же самое можно выполнить с левым поддеревом и его самым правым листовым узлом

Оценка временной сложности

Contains

$O(\log_2(n))$

Insert

$O(\log_2(n))$

```
Remove  
    0 (1)
```

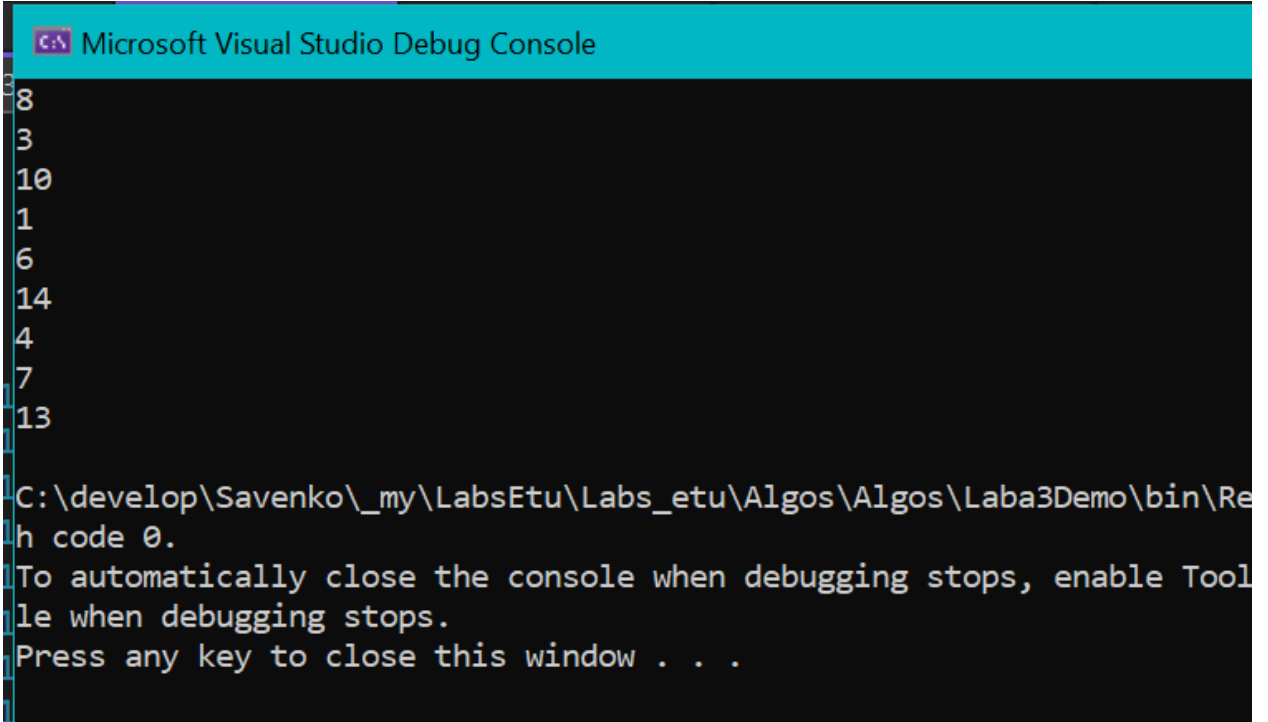
```
CreateBFTEnumerator  
    0 (1)
```

```
CreateDTFEnumerator  
    0 (1)
```

Описание Unit тестов

Во всех тестах создается заполненное двоичное дерево поиска, затем выполняются нужные операции над ним. Верность дерева после операций определяется проходом итератора.

Примеры работы



```
Microsoft Visual Studio Debug Console  
8  
3  
10  
1  
6  
14  
4  
7  
13  
  
C:\develop\Savenko\_my\LabsEtu\Labs_etu\Algos\Algos\Laba3Demo\bin\Re  
h code 0.  
To automatically close the console when debugging stops, enable Tool  
le when debugging stops.  
Press any key to close this window . . .
```

Листинг

```
using Laba3.Enumerators;  
using System;  
using System.Collections;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace Laba3  
{
```

```

public class BinarySearchTree<T> : IEnumerable<BinaryNode<T>> where T :
    IComparable<T>
    {
        public BinaryNode<T>? Root { get; private set; }

        public BinarySearchTree() {
            Root = null;
        }
        public BinarySearchTree(T root)
        {
            Root = new BinaryNode<T>(root);
        }

        public BinaryNode<T> Add(BinaryNode<T> node, BinaryNode<T> currentNode)
        {
            if (Root == null)
            {
                node.Parent = null;
                return Root = node;
            }

            currentNode = currentNode ?? Root;
            node.Parent = currentNode;

            int result;
            return (result = node.Value.CompareTo(currentNode.Value)) == 0
                ? currentNode
                : result < 0
                    ? currentNode.Left == null
                        ? (currentNode.Left = node)
                        : Add(node, currentNode.Left)
                    : currentNode.Right == null
                        ? (currentNode.Right = node)
                        : Add(node, currentNode.Right);
        }

        public void Remove(BinaryNode<T> node)
        {
            if (node == null)
            {
                return;
            }

            var currentNodeSide = node.Side;
            if (node.Left == null && node.Right == null)
            {
                if (currentNodeSide == -1)
                {
                    {
                        node.Parent.Left = null;
                    }
                }
                else if (currentNodeSide == 1)
                {
                    {
                        node.Parent.Right = null;
                    }
                }
                else
                {
                    {
                        // Reset
                        Root = null;
                    }
                }
            }
            else if (node.Left == null)
            {
                if (currentNodeSide == -1)
                {
                    {
                        node.Parent.Left = node.Right;
                    }
                }
            }
        }
    }

```

```

        }
        else
        {
            node.Parent.Right = node.Right;
        }

        node.Right.Parent = node.Parent;
    }
    else if (node.Right == null)
    {
        if (currentNodeSide == -1)
        {
            node.Parent.Left = node.Left;
        }
        else
        {
            node.Parent.Right = node.Left;
        }

        node.Left.Parent = node.Parent;
    }
    else
    {
        switch (currentNodeSide)
        {
            case -1:
                node.Parent.Left = node.Right;
                node.Right.Parent = node.Parent;
                Add(node.Left, node.Right);
                break;
            case 1:
                node.Parent.Right = node.Right;
                node.Right.Parent = node.Parent;
                Add(node.Left, node.Right);
                break;
            default:
                var tmpLeft = node.Left;
                var tmpRightLeft = node.Right.Left;
                var tmpRightRight = node.Right.Right;
                node.Value = node.Right.Value;
                node.Right = tmpRightRight;
                node.Left = tmpRightLeft;
                Add(tmpLeft, node);
                break;
        }
    }
}

public bool Contains(T value)
{
    BinaryNode<T> current = Root;
    while (Root != null)
    {
        if (current.Value.CompareTo(value) == 0)
        {
            return true;
        }
        else if ((value.CompareTo(current.Value) < 0))
        {
            if (current.Left == null)
            {
                return false;
            }
            current = current.Left;
        }
    }
}

```

```

        else
        {
            if (current.Right == null)
            {
                return false;
            }
            current = current.Right;
        }
    }
    return false;
}

public void Insert(T value)
{
    if (Root == null)
    {
        Root = new BinaryNode<T>(value);
        return;
    }
    BinaryNode<T> node = Root;
    while (true)
    {
        if (value.CompareTo(node.Value) <= 0)
        {
            if (node.Left == null)
            {
                node.Left = new BinaryNode<T>(value);
                return;
            }
            node = node.Left;
        }
        else
        {
            if (node.Right == null)
            {
                node.Right = new BinaryNode<T>(value);
                return;
            }
            node = node.Right;
        }
    }
}

public IEnumerator<BinaryNode<T>> GetEnumerator() => CreateBFTEnumerator();

IEnumerator IEnumerable.GetEnumerator() => CreateBFTEnumerator();

public IEnumerator<BinaryNode<T>> CreateDTFEnumerator()
{
    return new DepthFirstTraverseEnumerator<T>(this);
}

public IEnumerator<BinaryNode<T>> CreateBFTEnumerator()
{
    return new BreadthFirstTraverseEnumerator<T>(this);
}
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

```

```

namespace Laba3
{
    public class BinaryNode<T> : IComparable<BinaryNode<T>> where T : IComparable<T>
    {
        public T Value { get; set; }

        public BinaryNode<T> Parent { get; set; }
        public BinaryNode<T> Left { get; set; }
        public BinaryNode<T> Right { get; set; }

        // Left: -1
        // Right: 1
        public int Side
        {
            get
            {
                return Parent != null ? (Parent.Left == this ? -1 : 1) : 0;
            }
        }
        // TODO Add parent
        public BinaryNode(T value)
        {
            Value = value;
        }

        public int CompareTo(BinaryNode<T>? other)
        {
            return Value.CompareTo(other.Value);
        }
    }
}

```

```

using Laba3.Queue;
using Laba3.Stack;
using System;
using System.Collections;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

```

```

namespace Laba3.Enumerators
{
    class BreadthFirstTraverseEnumerator<T> : IEnumerator<BinaryNode<T>> where T :
    IComparable<T>
    {
        public BinarySearchTree<T> Tree { get; private set; }

        private CustomQueue<BinaryNode<T>> _queue;

        private BinaryNode<T> _current;

        public BreadthFirstTraverseEnumerator(BinarySearchTree<T> tree)
        {
            Tree = tree ?? throw new ArgumentNullException(nameof(tree));
            _current = null;
            _queue = new CustomQueue<BinaryNode<T>>();
        }

        object IEnumerator.Current => _queue.Peek();

        BinaryNode<T> IEnumerator<BinaryNode<T>>.Current => _queue.Peek();

        public bool MoveNext()
        {
            if (_current == null)

```

```

        {
            _current = Tree.Root;
            _queue.Enqueue(_current);
            return true;
        }

        if (!HasNext())
        {
            return false;
        }

        _current = _queue.Dequeue();

        if (_current.Left != null)
        {
            _queue.Enqueue(_current.Left);
        }
        if (_current.Right != null)
        {
            _queue.Enqueue(_current.Right);
        }

        return true;
    }

    public void Reset()
    {
        _current = Tree.Root;
    }

    public bool HasNext()
    {
        if(_current == null)
        {
            return Tree.Root != null;
        }
        if (_current.Left != null || _current.Right != null)
        {
            return true;
        }
        return _queue.Count > 1;
    }

    public void Dispose()
    {
        //throw new NotImplementedException();
    }
}

}

using Laba3.Stack;
using System;
using System.Collections;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Laba3.Enumerators
{
    public class DepthFirstTraverseEnumerator<T> : IEnumerator<BinaryNode<T>> where
T : IComparable<T>
    {
        public BinarySearchTree<T> Tree { get; private set; }

        private BinaryNode<T> _current;

```



```

private readonly CustomStack<T> _stack;
public DepthFirstTraverseEnumerator(BinarySearchTree<T> tree)
{
    Tree = tree ?? throw new ArgumentNullException(nameof(tree));
    _current = null;
    _stack = new CustomStack<T>();
}

object IEnumerator.Current => _current;

BinaryNode<T> IEnumerator<BinaryNode<T>>.Current => _current;

public bool MoveNext()
{
    if (_current == null)
    {
        _current = Tree.Root;
        _stack.Push(_current);
        return true;
    }

    if (!HasNext())
    {
        return false;
    }

    if (_current.Right != null)
    {
        _stack.Push(_current.Right);
    }

    if (_current.Left != null)
        _current = _current.Left;
    else
    {
        _current = _stack.Top;
        _stack.Pop();
    }

    return true;
}

public void Reset()
{
    _current = Tree.Root;
}

public bool HasNext()
{
    if (_current == null)
    {
        return Tree.Root != null;
    }
    if (_current.Left != null || _current.Right != null)
    {
        return true;
    }
    return _stack.Count() > 1;
}

public void Dispose()
{
    throw new NotImplementedException();
}

```

} }