**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»**

## Вариант №1

Студент гр. 9309 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Юшин Е.В.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Тутуева А.В.

Санкт-Петербург

2021

## Цель работы

Научиться использовать итераторы для работы с двоичными деревьями.

## Постановка задачи:

Реализовать для двоичного дерева поиска методы поиск элемента в дереве по ключу, добавление элемента в дерево по ключу, удаление элемента дерева по ключу, создание итератора, реализующего один из методов обхода в глубину, создание итератора, реализующего методы обхода в ширину.

## Описание алгоритмов:

## Описание классов:

Классы BSTREE и Node хранят в себе все необходимые указатели и данные дерева двоичного поиска.

## Описание методов:

**-rec\_ins**  
Рекурсивный алгоритм поиска места для добавления нового элемента с последующим добавлением.  
**-find**  
Ищет в дереве элемент, сокращая с каждой итерацией область поиска.  
**-find\_min**  
Ищет минимальный элемент в поддереве.  
**-insert**  
Удаляет созданные итераторы и вызывает recursive\_ins.  
**-contains**  
Вызывает find и возвращает true, если возвращаемое значение find не равно NULL.  
**-remove**  
Удаляет созданные ранее итераторы, с помощью вызова find находит нужный элемент и удаляет его.  
**-Алгоритмы работы итераторов.**  
Итератор для обхода в глубину использует стэк.   
Итератор для обхода в ширину использует очередь, в которую помещается сначала левый дочерний элемент текущего элемента, а затем — правый.

## Оценка временной сложности методов:

1)rec\_ins O(log(n))

2)find O(log(n))

3)find\_min O(log(n))

4)insert O(log(n))

5)remove O(log(n))

## Текст UnitTest:

#include "pch.h"

#include "CppUnitTest.h"

#include "../Lab3Aistrd\_Ushin/main.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest1

{

TEST\_CLASS(UnitTest1)

{

public:

TEST\_METHOD(insert\_and\_bft)

{

BSTREE T;

T.insert(7);

T.insert(3);

T.insert(1);

T.insert(5);

T.insert(11);

T.insert(12);

T.insert(8);

Iterator\* BFT = T.create\_bft\_iterator();

Assert::AreEqual(BFT->next(), 7);

Assert::AreEqual(BFT->next(), 3);

Assert::AreEqual(BFT->next(), 11);

Assert::AreEqual(BFT->next(), 1);

Assert::AreEqual(BFT->next(), 5);

Assert::AreEqual(BFT->next(), 8);

Assert::AreEqual(BFT->next(), 12);

}

TEST\_METHOD(bft\_except)

{

BSTREE T;

Iterator\* BFT = T.create\_bft\_iterator();

bool indicator = 0;

try

{

BFT->next();

}

catch (exception& exception)

{

indicator = 1;

}

Assert::IsTrue(indicator);

}

TEST\_METHOD(insert\_and\_dft)

{

BSTREE T;

T.insert(9);

T.insert(5);

T.insert(3);

T.insert(7);

T.insert(14);

T.insert(16);

T.insert(10);

Iterator\* BFT = T.create\_dft\_iterator();

Assert::AreEqual(BFT->next(), 9);

Assert::AreEqual(BFT->next(), 5);

Assert::AreEqual(BFT->next(), 3);

Assert::AreEqual(BFT->next(), 7);

Assert::AreEqual(BFT->next(), 14);

Assert::AreEqual(BFT->next(), 10);

Assert::AreEqual(BFT->next(), 16);

}

TEST\_METHOD(dft\_except)

{

BSTREE T;

Iterator\* DFT = T.create\_dft\_iterator();

bool indicator = 0;

try

{

DFT->next();

}

catch (exception& exception)

{

indicator = 1;

}

Assert::IsTrue(indicator);

}

TEST\_METHOD(contains)

{

BSTREE T;

T.insert(7);

T.insert(3);

T.insert(17);

Assert::IsTrue(T.contains(17));

Assert::IsFalse(T.contains(1));

}

TEST\_METHOD(remove)

{

BSTREE T;

T.insert(7);

T.insert(3);

T.insert(1);

T.insert(5);

T.insert(10);

T.insert(12);

T.insert(8);

T.remove(1);

Iterator\* BFT = T.create\_bft\_iterator();

Assert::AreEqual(BFT->next(), 7);

Assert::AreEqual(BFT->next(), 3);

Assert::AreEqual(BFT->next(), 10);

Assert::AreEqual(BFT->next(), 5);

Assert::AreEqual(BFT->next(), 8);

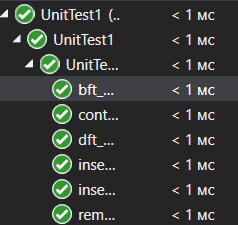
Assert::AreEqual(BFT->next(), 12);

}

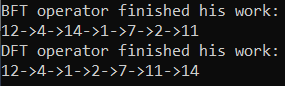
};

}

## Результат выполнения unit-тестов:



## Результат работы программы:



## Текст программы:

**main.cpp**

#include <iostream>

#include <stdexcept>

using namespace std;

class Iterator

{

public:

virtual int next() = 0;

virtual bool has\_next() = 0;

};

class BSTREE

{

class Node

{

public:

int key;

Node\* left;

Node\* right;

Node\* parent;

Node(int val, Node\* prev = NULL)

{

key = val;

parent = prev;

left = NULL;

right = NULL;

}

};

Node\* head;

Iterator\* dft;

Iterator\* bft;

void rec\_ins(Node\* current, int key)

{

if (key >= current->key)

{

if (current->right == NULL)

current->right = new Node(key, current);

else

rec\_ins(current->right, key);

}

else

{

if (current->left == NULL)

current->left = new Node(key, current);

else

rec\_ins(current->left, key);

}

}

Node\* find(Node\* current, int key)

{

if (current == NULL)

return NULL;

else if (current->key == key)

return current;

else if (key > current->key)

return find(current->right, key);

else

return find(current->left, key);

}

Node\* find\_min(Node\* current)

{

if (current->left == NULL)

return current;

else

return find\_min(current->left);

}

public:

BSTREE()

{

head = NULL;

dft = NULL;

bft = NULL;

}

class DFT :public Iterator

{

class Stack

{

public:

Node\* inf;

Stack\* next;

Stack(Node\* Inf, Stack\* Next)

{

inf = Inf;

next = Next;

}

};

Stack\* Head;

Node\* Current;

public:

DFT(Node\* begin)

{

Head = NULL;

Current = begin;

}

int next() override

{

if (!has\_next())

throw out\_of\_range("Tree is over\n");

int returning\_value = Current->key;

if (Current->right != NULL)

{

Stack\* New = new Stack(Current->right, Head);

Head = New;

}

if (Current->left != NULL)

{

Current = Current->left;

}

else

{

if (Head != NULL)

{

Current = Head->inf;

Stack\* Next;

Next = Head->next;

delete Head;

Head = Next;

}

else

Current = NULL;

}

return returning\_value;

}

bool has\_next() override

{

return (Current != NULL);

}

};

Iterator\* create\_dft\_iterator()

{

if (dft != NULL)

delete dft;

dft = new DFT(head);

return dft;

}

class BFT :public Iterator

{

class Tree\_queue

{

public:

Node\* inf;

Tree\_queue\* next;

Tree\_queue(Node\* Inf, Tree\_queue\* Next)

{

inf = Inf;

next = Next;

}

};

Tree\_queue\* First;

Tree\_queue\* Last;

Node\* Current;

public:

BFT(Node\* Start)

{

First = NULL;

Last = NULL;

Current = Start;

}

int next() override

{

if (!has\_next())

throw out\_of\_range("Tree is over\n");

int returning\_value = Current->key;

if (Current->left != NULL)

{

Tree\_queue\* New = new Tree\_queue(Current->left, Last);

if (First == NULL)

{

First = New;

}

Last = New;

}

if (Current->right != NULL)

{

Tree\_queue\* New = new Tree\_queue(Current->right, Last);

if (First == NULL)

{

First = New;

}

Last = New;

}

Tree\_queue\* CURRENT = Last;

if (CURRENT != NULL)

{

if (CURRENT == First)

{

Current = CURRENT->inf;

delete CURRENT;

Last = NULL;

First = NULL;

}

else

{

while (CURRENT->next != First)

{

CURRENT = CURRENT->next;

}

Current = CURRENT->next->inf;

delete First;

First = CURRENT;

CURRENT->next = NULL;

}

}

else

Current = NULL;

return returning\_value;

}

bool has\_next() override

{

return (Current != NULL);

}

};

Iterator\* create\_bft\_iterator()

{

if (bft != NULL)

delete bft;

bft = new BFT(head);

return bft;

}

void insert(int key)

{

if (bft != NULL)

{

delete bft;

bft = NULL;

}

if (dft != NULL)

{

delete dft;

dft = NULL;

}

if (head == NULL)

head = new Node(key);

else

rec\_ins(head, key);

}

bool contains(int key)

{

return (find(head, key) != NULL);

}

void remove(int key)

{

if (bft != NULL)

{

delete bft;

bft = NULL;

}

if (dft != NULL)

{

delete dft;

dft = NULL;

}

Node\* deleter = find(head, key);

if (deleter == NULL)

throw invalid\_argument("Element is not included in tree\n");

if ((deleter->left == NULL) && (deleter->right == NULL))

{

if (deleter->parent == NULL)

head = NULL;

else if (deleter->parent->left == deleter)

deleter->parent->left = NULL;

else

deleter->parent->right = NULL;

delete deleter;

}

else if ((deleter->left == NULL) && (deleter->right != NULL))

{

if (deleter->parent == NULL)

{

head = deleter->right;

deleter->right->parent = NULL;

}

else if (deleter->parent->left == deleter)

{

deleter->parent->left = deleter->right;

deleter->right->parent = deleter->parent;

}

else

{

deleter->parent->right = deleter->right;

deleter->right->parent = deleter->parent;

}

delete deleter;

}

else if ((deleter->left != NULL) && (deleter->right == NULL))

{

if (deleter->parent == NULL)

{

head = deleter->left;

deleter->left->parent = NULL;

}

else if (deleter->parent->left == deleter)

{

deleter->parent->left = deleter->left;

deleter->left->parent = deleter->parent;

}

else

{

deleter->parent->right = deleter->left;

deleter->left->parent = deleter->parent;

}

delete deleter;

}

else if ((deleter->left != NULL) && (deleter->right != NULL))

{

Node\* Min = find\_min(deleter->right);

if (deleter->parent == NULL)

head = Min;

if (Min->parent->left == Min)

Min->parent->left = Min->right;

else

Min->parent->right = Min->right;

if (Min->right != NULL)

Min->right->parent = Min->parent;

Min->parent = deleter->parent;

if (Min->parent != NULL)

{

if (Min->parent->right == deleter)

Min->parent->right = Min;

else

Min->parent->left = Min;

}

Min->left = deleter->left;

Min->right = deleter->right;

Min->left->parent = Min;

if (Min->right != NULL)

Min->right->parent = Min;

delete deleter;

}

}

~BSTREE()

{

if (bft != NULL)

{

delete bft;

bft = NULL;

}

if (dft != NULL)

{

delete dft;

dft = NULL;

}

while (head != NULL)

remove(head->key);

}

};

int main()

{

BSTREE T;

T.insert(12);

T.insert(14);

T.insert(4);

T.insert(1);

T.insert(2);

T.insert(7);

T.insert(11);

Iterator\* BFT = T.create\_bft\_iterator();

Iterator\* DFT = T.create\_dft\_iterator();

cout << "BFT operator finished his work: " << endl;

while (BFT->has\_next())

{

cout << BFT->next();

if (BFT->has\_next())

{

cout << "->";

}

}

cout << endl;

cout << "DFT operator finished his work: " << endl;

while (DFT->has\_next())

{

cout << DFT->next();

if (DFT->has\_next())

{

cout << "->";

}

}

cout << endl;

}