# 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 на примере языка С++**

**Вариант 1**

Студент 0301 ————————— Сморжок В. Е.

Преподаватель ————————— Тутуева А.В.

Санкт-Петербург,

2021

Задание на лабораторную работу:

Реализовать класс двоичного дерева в соответствии со своим вариантом. (двоичное дерево поиска)

Список методов, которые реализует каждый вариант:

1. bool contains(int); // поиск элемента в дереве по ключу

2. void insert(int); // добавление элемента в дерево по ключу. Должен

работать за O(logN)

3. void remove(int); // удаление элемента дерева по ключу

4. Iterator create\_dft\_iterator(); // создание итератора, реализующего один из

методов обхода в глубину (depth-first traverse)

5. Iterator create\_bft\_iterator() // создание итератора, реализующего методы

обхода в ширину (breadth-first traverse)

Текст программы:

#include <iostream>

using namespace std;

size\_t get\_size;

int per = 0;

int firstgo = 0;

int sizeofstack = 0;

class Queue

{

private:

class Node

{

public:

int element;

Node\* next;

};

Node\* head;

Node\* tail;

public:

int size\_queue;

Queue()

{

Node\* current = new Node;

head = current;

current->next = NULL;

tail = head = current;

size\_queue = 0;

}

~Queue()

{

while (head != tail)

remove();

}

int getfirst()

{

if (size\_queue != 0)

{

return head->element;

}

else

{

throw out\_of\_range("Empty list");

}

}

void remove() // delete an element

{

head = head->next;

size\_queue--;

}

void insert(int element)

{

if (size\_queue == 0)

{

head = new Node;

head->element = element;

tail = head;

}

else

{

tail->next = new Node;

tail = tail->next;

tail->element = element;

tail->next = NULL;

}

size\_queue++;

}

};

class Stack

{

private:

class Node

{

public:

int element;

Node\* previous;

};

public:

Node\* current;

int size\_stack;

Stack()

{

Node\* buffer = new Node;

buffer->element = NULL;

buffer->previous = NULL;

size\_stack = 0;

}

~Stack()

{

Node\* buffer;

while (current)

{

buffer = current->previous;

delete(current);

current = buffer;

}

}

void add(int element)

{

if (size\_stack == 0)

{

Node\* buffer = new Node;

buffer->element = element;

buffer->previous = nullptr;

current = buffer;

}

else

{

Node\* buffer = current;

current = new Node;

current->element = element;

current->previous = buffer;

}

size\_stack++;

}

void remove()

{

Node\* buffer = current;

current = current->previous;

delete(buffer);

size\_stack--;

}

int getlast()

{

return current->element;

}

int size()

{

if (sizeofstack == 0)

return size\_stack;

else

return 0;

}

};

class Iterator

{

public:

virtual int next() = 0;

virtual bool has\_next() = 0;

};

class Tree

{

private:

class Node

{

public:

Node\* left = NULL;

Node\* right = NULL;

int element = NULL;

int forstack = 0;

};

Node\* parent;

public:

Iterator\* create\_BFT\_iterator()

{

return new ListIteratorBFT(parent);

}

Iterator\* create\_DFT\_iterator()

{

return new ListIteratorDFT(parent);

}

void push\_back(int element)

{

Node\* buffer = new Node;

buffer->element = element;

if (parent->element == NULL)

{

parent = buffer;

parent->left = NULL;

parent->right == NULL;

}

else

{

Node\* root = parent;

while (1)

{

if (root->element >= buffer->element)

{

if (root->left == NULL)

root->left = new Node;

root = root->left;

}

else {

if (root->right == NULL)

root->right = new Node;

root = root->right;

}

if (root->element == NULL)

{

root->element = (int)buffer->element;

break;

}

}

}

get\_size++;

}

Tree() // default constructor

{

Node\* buffer = new Node;

buffer->left = buffer->right = NULL;

buffer->element = NULL;

parent = buffer;

}

class ListIteratorBFT : public Iterator

{

public:

ListIteratorBFT(Node\* buffer)

{

current = buffer;

check = current;

queue.insert(buffer->element);

};

bool has\_next() override;

int next() override;

private:

Queue queue;

Node\* current;

Node\* check;

};

class ListIteratorDFT : public Iterator

{

public:

ListIteratorDFT(Node\* buffer)

{

current = buffer;

checkstack = current;

stack.add(buffer->element);

};

bool has\_next() override;

int next() override;

private:

Stack stack;

Node\* current;

Node\* checkstack;

};

bool Contains(int value)

{

Node\* buffer;

Node\* root = parent;

bool answer = false;

if (root->element == value)

{

answer = true;

}

while (answer == false && (root->left != nullptr || root->right != nullptr))

{

if (value <= root->element)

{

root = root->left;

}

else

{

root = root->right;

}

if (root->element == value)

{

answer = true;

break;

}

}

return answer;

}

void insert(int value)

{

Node\* buffer = parent;

bool answer = false;

Node\* root = parent;

if (root->element >= value)

{

root = root->left;

}

else root = root->right;

while (1)

{

if (root != NULL)

{

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

{

break;

}

}

else

{

if (value <= buffer->element)

{

buffer->left = new Node;

buffer->left->element = value;

}

else

{

buffer->right = new Node;

buffer->right->element = value;

}

answer = true;

break;

}

buffer = root;

if (root->element >= value)

{

root = root->left;

}

else root = root->right;

}

if (answer == false)

if (root->element >= value)

{

root->left = new Node;

root->left->element = value;

}

else

{

root->right = new Node;

root->right->element = value;

}

}

void remove(int value)

{

Node\* root = parent;

Node\* buffer = parent;

Node\* help = parent;

while (root->element != value)

{

if (root->element >= value)

{

buffer = root;

root = root->left;

}

else

{

buffer = root;

root = root->right;

}

}

if (root->left == nullptr && root->right == nullptr)

{

if (buffer->left == root)

{

buffer->left = nullptr;

}

else

buffer->right = nullptr;

delete(root);

}

else

{

if (root->left == nullptr)

{

root->element = root->right->element;

root = root->right;

buffer->right = nullptr;

delete(root);

}

else

if (root->right == nullptr)

{

root->element = root->left->element;

root = root->left;

buffer->left = nullptr;

delete (root);

}

if (root->left != nullptr && root->right != nullptr)

{

buffer = root;

buffer = buffer->right;

while (buffer->left != nullptr)

{

help = buffer;

buffer = buffer->left;

}

root->element = buffer->element;

if (help != parent)

help->left = nullptr;

delete(buffer);

}

}

}

};

int Tree::ListIteratorBFT::next()

{

int temp = queue.getfirst();

if (queue.getfirst() == current->element && per == 0)

{

check = current;

per = 1;

}

else

{

if (check != nullptr)

current = check;

}

queue.remove();

while (current->element != temp)

{

if (current->element > temp)

{

current = current->left;

}

else

{

current = current->right;

}

}

if (current->left != nullptr)

{

queue.insert(current->left->element);

}

if (current->right != nullptr)

{

queue.insert(current->right->element);

}

return temp;

}

bool Tree::ListIteratorBFT::has\_next()

{

if (queue.size\_queue == 0)

{

return false;

}

else

{

return true;

}

}

int Tree::ListIteratorDFT::next()

{

int per = 0;

int temp = 0;

if (current->element == stack.current->element && firstgo == 0)

{

firstgo = 1;

temp = current->element;

return temp;

}

else

{

current = checkstack;

while (1)

{

if (current == checkstack)

while (stack.getlast() != current->element)

{

if (current->element > stack.getlast())

{

current = current->left;

}

else

{

current = current->right;

}

}

if (current->left != nullptr && current->left->forstack == 0)

{

current = current->left;

stack.add(current->element);

temp = current->element;

break;

}

else

if (current->right != nullptr && current->right->forstack == 0)

{

current = current->right;

stack.add(current->element);

temp = current->element;

if (current->left == nullptr && current->right == nullptr)

{

stack.remove();

current->forstack = 1;

}

break;

}

else

{

stack.remove();

current->forstack = 1;

current = checkstack;

}

}

}

if (checkstack->left->forstack == 1 && checkstack->left->forstack == 1)

{

sizeofstack = 1;

return temp;

}

else

return temp;

}

bool Tree::ListIteratorDFT::has\_next()

{

if (stack.size() != 0)

{

return true;

}

else

{

return false;

}

}

int main()

{

int array[6] = {8,10,3,6,1,4};

int\* arraystek;

int n = 6;

arraystek = (int\*)malloc(n \* sizeof(int));

Tree lst;

for (int i = 0; i < n; i++)

{

lst.push\_back(array[i]);

}

bool answer = lst.Contains(3);

if (answer == 1)

{

cout << "this element is contained in a Tree" << endl;

}

else

cout << "this element isn't contained in a Tree" << endl;

lst.insert(7);

lst.remove(4);

cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

cout << "BFT Iterator: " << endl;

Iterator\* lst\_iterator = lst.create\_BFT\_iterator();

while (lst\_iterator->has\_next())

{

cout << lst\_iterator->next() << ' ';

}

cout << endl;

cout << "DFT Iterator: " << endl;

Iterator\* lst\_iterator\_DFT = lst.create\_DFT\_iterator();

while (lst\_iterator\_DFT->has\_next())

{

cout << lst\_iterator\_DFT->next() << ' ';

}

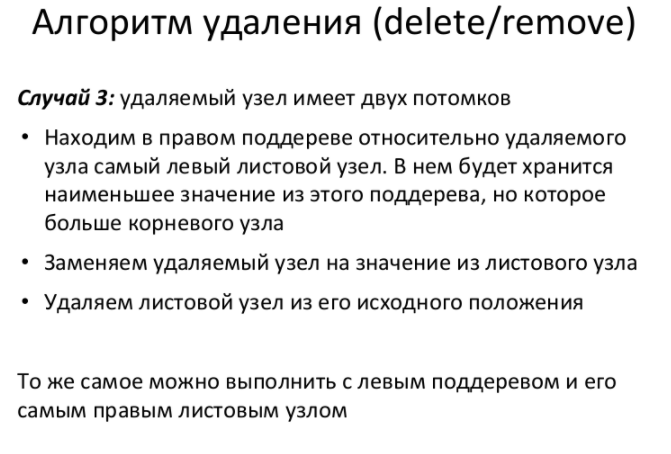
cout << endl;

return 0;

}

Описание реализуемых алгоритмов:

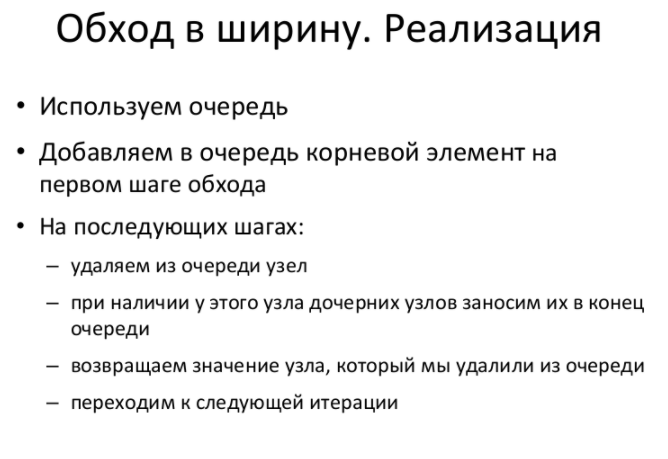
В данной программе реализован класс двоичного бинарного дерева. Класс содержит в себе ссылку на родителя, данные о содержимом узла графа – значение элемента, ссылку на правый и левый элементы дерева. Для этого дерева реализованы конструктор, деструктор, три метода: удаление, вставка и проверка на содержание в дереве элемента. Удаление листа происходит обнулением этого элемента и удалением всех ссылок на него. Удаление узла происходит по алгоритму:



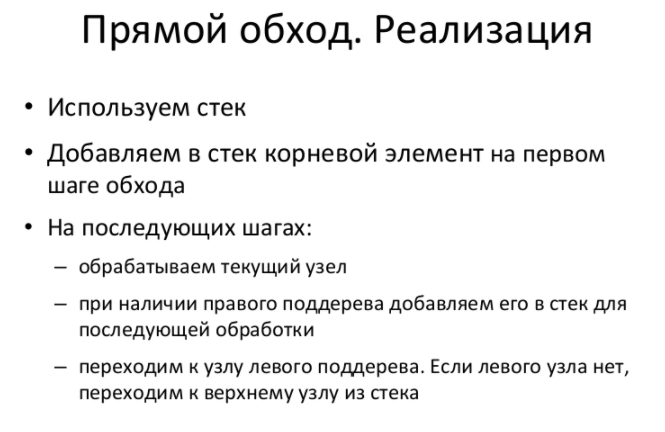
Удаление узла с одним дочерним узлом происходит заменой этого элемента на листовой и удалением листового. Метод проверки на наличие элемента в дереве реализован с помощью обхода всего графа. Метод вставки реализован с помощью обхода всего графа, сравнением всех элементов со вставляемыми и вставка в конец графа, то есть элемент становится листовым.

Для реализации обхода в ширину и глубину были реализованы сект и очередь, как классы. Для них были созданы методы добавления, удаления, конструкторы, деструкторы.

Обход в ширину:



Обход в глубину:



Были созданы итераторы для каждого из обходов, также методы для них, с помощью которых и был выполнен обход. В методе next выполнялись вышеуказанные алгоритмы.

Оценка временной сложности каждого метода:

1. bool contains(int) – O(logN)

2. void insert(int) – O(logN)

3. void remove(int) – O(logN)

4. Iterator create\_dft\_iterator();

5. Iterator create\_bft\_iterator()

Текст Unit – тестов

#include "pch.h"

#include "CppUnitTest.h"

#include "../3 lab.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTestLab3

{

TEST\_CLASS(UnitTestLab3)

{

public:

TEST\_METHOD(ContainsFunction)

{

Tree lst;

int array[] = { 16,6,20,2,12 };

for (int i = 0; i < 5; i++)

{

lst.push\_back(array[i]);

}

int temp = 0;

for (int i = 0; i < 5; i++)

{

temp = lst.Contains(array[i]);

Assert::IsTrue(temp == 1);

}

}

TEST\_METHOD(BFTFunction)

{

Tree lst;

int array[7] = { 12,10,5,19,6,4,7 };

for (int i = 0; i < 7; i++)

{

lst.push\_back(array[i]);

}

int arraytest[] = { 12,10,19,5,4,6,7 };

Iterator\* lst\_iterator = lst.create\_BFT\_iterator();

int i = 0;

int temp;

while (lst\_iterator->has\_next())

{

temp = lst\_iterator->next();

Assert::IsTrue(temp == arraytest[i]);

i++;

}

}

TEST\_METHOD(DFTFunction)

{

Tree lst;

int array[7] = { 12,10,5,19,6,4,7 };

for (int i = 0; i < 7; i++)

{

lst.push\_back(array[i]);

}

int arraytest[] = { 12,10,5,4,6,7,19 };

Iterator\* lst\_iterator = lst.create\_DFT\_iterator();

int i = 0;

int temp;

while (lst\_iterator->has\_next())

{

temp = lst\_iterator->next();

Assert::IsTrue(temp == arraytest[i]);

i++;

}

}

TEST\_METHOD(InsertFunction)

{

Tree lst;

int array[5] = { 16,20,6,2,12 };

for (int i = 0; i < 5; i++)

{

lst.push\_back(array[i]);

}

int arraytest[] = { 16,6,20,2,12,14 };

lst.insert(14);

Iterator\* lst\_iterator = lst.create\_BFT\_iterator();

int i = 0;

int temp;

for (int i = 0; i < 6; i++)

{

temp = lst\_iterator->next();

Assert::IsTrue(temp == arraytest[i]);

}

}

TEST\_METHOD(RemoveFunction)

{

Tree lst;

int array[7] = { 16,20,6,2,12,11,13 };

for (int i = 0; i < 7; i++)

{

lst.push\_back(array[i]);

}

int arraytest[] = { 16,11,20,2,12,13 };

lst.remove(6);

Iterator\* lst\_iterator = lst.create\_BFT\_iterator();

int i = 0;

int temp;

for (int i = 0; i < 6; i++)

{

temp = lst\_iterator->next();

Assert::IsTrue(temp == arraytest[i]);

}

int arraytest1[5] = {16,11,20,2,12};

lst.remove(13);

Iterator\* lst\_iterator1 = lst.create\_BFT\_iterator();

for (int i = 0; i < 4; i++)

{

temp = lst\_iterator1->next();

Assert::IsTrue(temp == arraytest[i]);

}

}

};

}

Описание Unit – тестов

1. Contains – проверяет каждый элемент дерева этим методом. Выполняется, если все результаты будут равны 1.
2. BTF обход в ширину – проверяет, равен ли результат работы итератора и его методов массиву, который отсортировал элементы графа обходом в ширину
3. DFT – проверяет, равен ли результат работы итератора и его методов массиву, который отсортировал элементы графа обходом в глубину
4. Insert – проверяет, вставился ли элемент в дерево, с помощью обхода в ширину
5. Remove - проверяет, удалился ли элемент из дерева, с помощью обхода в ширину. Рассмотрены случаи удаления листа и узлового элемента

Пример работы программы:

