***Министерство образования Республики Беларусь***

***Учреждение образования***

***«Брестский государственный технический университет»***

***Кафедра ИИТ***

**Лабораторная работа №6**

**По дисциплине МОИС за III семестр**

**Тема: «Кодирование деревьев»**

**Выполнил:**

Студент группы ИИ-15 (1)

2-го курса

Волк И. А.

**Проверил:**

Шуть В. Н.

Брест 2018

Цель: научиться кодировать и раскодировать деревья.

Задание.

1. Написать программу кодировки/декодировки деревьев кодами Прюфера.

2. Рассмотреть указанное дерево как бинарное дерево с корнем в вершине 0. Проверить является ли дерево сбалансированным. В случае отрицательного ответа, построить сбалансированное бинарное дерево поиска по заданному через таблицу списку вершин дерева.

3. Все построения проиллюстрировать графически.

|  |  |
| --- | --- |
| 7. | *(0,1),(0,2),(1,3),(1,4),(2,5), (2,6),(3,7),*  *(3,8),(6,9),(8,10),(8,11),(10,12),(11,13)* |

Код программы:

**//main.cpp**

#include "pch.h"

#include <iostream>

#include "lab6\_src.h"

int **main**()

{

typedef int Type;

Lab6::BTree<Type> bt;

bt.add(0);

bt.add(1);

bt.add(2);

bt.add(3);

bt.add(4);

bt.add(5);

bt.add(6);

bt.add(7);

bt.add(8);

bt.add(9);

bt.add(10);

bt.add(11);

bt.add(12);

bt.add(13);

bt.view();

std::cout << std::endl;

auto code = bt.getCode();

auto valueList = bt.getValueList();

auto root = bt.getRoot();

Lab6::BTree<Type> bt\_copy(code, valueList, root);

bt\_copy.view();

std::cin.get();

}

**//lab6\_src.h**

#pragma once

#include "graph.h"

#include <vector>

#include <queue>

#define **max**(a, b) ((a>b)?(a):(b))

#define **erase\_from\_vector**(a, b) a.erase(std::find(a.begin(), a.end(), b))

#define **makeIndent**(a) for(int i = 0; i < a; i++) std::cout << " ";

namespace Lab6

{

int **sumAllBefore**(int a);

template<typename ValueType = int, typename Compare = std::less<ValueType>>

class **BTree**

{

std::vector<ValueType> valueList;

enum **SonType**

{

ROOT = 0,

LEFT,

RIGHT

};

struct **Element**

{

Element \*parent;

Element \*left, \*right;

ValueType value;

SonType type;

Element()

: parent(nullptr), left(nullptr), right(nullptr), value(), type(ROOT) {}

Element(const ValueType& value)

: parent(nullptr), left(nullptr), right(nullptr), value(value), type(ROOT) {}

};

Element \*root;

void **copy\_constructor**(Element \*&el1, Element \*el2)

{

if (el2 == nullptr)

return;

el1 = new Element(el2->value);

copy\_constructor(el1->left, el2->left);

copy\_constructor(el1->right, el2->right);

}

void **destructor**(Element \*&el)

{

if (el == nullptr)

return;

destructor(el->left);

destructor(el->right);

delete el;

}

int **balancing**(Element\*& root)

{

bool OK = true;

int l = -1, r = -1;

do

{

if (root == nullptr)

break;

OK = true;

l = balancing(root->left);

r = balancing(root->right);

if (abs(r - l) > 1)

{

OK = false;

if (r - l > 1)

{

rLeft(root);

}

else

{

rRight(root);

}

}

} while (!OK);

return max(r, l) + 1;

}

void **rLeft**(Element\* root)

{

if (root->type == RIGHT)

{

root->parent->right = root->right;

root->right->parent = root->parent;

}

else if (root->type == LEFT)

{

root->parent->left = root->right;

root->right->parent = root->parent;

root->right->type = LEFT;

}

else if (root->type == ROOT)

{

this->root = root->right;

this->root->parent = nullptr;

this->root->type = ROOT;

}

if (root->left != nullptr)

{

Element\* temp = root->right;

while (temp->left != nullptr)

temp = temp->left;

temp->left = root->left;

root->left->parent = temp;

}

erase\_from\_vector(valueList, root->value);

add(root->value, false);

delete root;

}

void **rRight**(Element\* root)

{

if (root->type == RIGHT)

{

root->parent->right = root->left;

root->left->parent = root->parent;

root->left->type = RIGHT;

}

else if (root->type == LEFT)

{

root->parent->left = root->left;

root->left->parent = root->parent;

}

else if (root->type == ROOT)

{

this->root = root->left;

this->root->parent = nullptr;

this->root->type = ROOT;

}

if (root->right != nullptr)

{

Element\* temp = root->left;

while (temp->right != nullptr)

temp = temp->right;

temp->right = root->right;

root->right->parent = temp;

}

erase\_from\_vector(valueList, root->value);

add(root->value, false);

delete root;

}

void **fillCode**(std::vector<ValueType>& code, Element \* el)

{

if (el != nullptr)

{

fillCode(code, el->left);

fillCode(code, el->right);

code.push\_back(el->parent->value);

}

}

public:

**BTree**()

: root(nullptr) {}

**BTree**(const ValueType& root)

: root(new Element(root)) {}

**BTree**(std::vector<ValueType> code, std::vector<ValueType> valueList, const ValueType& root)

{

class **Added**

{

std::vector<bool> mark;

std::vector<ValueType> values;

public:

**Added**(const std::vector<ValueType>& valueList)

: mark(valueList.size()), values(valueList.size())

{

std::fill(mark.begin(), mark.end(), false);

for (int i = 0; i < values.size(); i++)

values[i] = valueList[i];

}

bool **isAdded**(const ValueType& el)

{

bool res = false;

for (int i = 0; i < values.size(); i++)

if (values[i] == el)

{

res = mark[i];

break;

}

return res;

}

void **markAdded**(const ValueType& el)

{

for (int i = 0; i < values.size(); i++)

if (values[i] == el)

{

mark[i] = true;

}

}

bool **notAllTrue**()

{

bool res = false;

for (bool x : mark)

if (x == false)

{

res = true;

break;

}

return res;

}

};

Added added(valueList);

this->root = new Element(root);

this->valueList.push\_back(root);

added.markAdded(root);

typedef std::pair<ValueType, ValueType> Edge;

std::vector<Edge> edges;

ValueType parent, child;

while (!code.empty())

{

parent = code[0];

for (int i = 0; i < valueList.size(); i++)

{

bool isInCode = false;

for (int j = 0; j < code.size(); j++)

if (valueList[i] == code[j])

{

isInCode = true;

break;

}

if (!isInCode)

child = valueList[i];

}

for (int i = 0; i < valueList.size(); i++)

{

bool isInCode = false;

for (int j = 0; j < code.size(); j++)

if (valueList[i] == code[j])

{

isInCode = true;

break;

}

if (!isInCode)

if (valueList[i] < child)

child = valueList[i];

}

edges.push\_back({ parent, child });

code.erase(code.begin());

erase\_from\_vector(valueList, child);

}

edges.push\_back({ valueList[0], valueList[1] });

while (added.notAllTrue())

{

for (int i = 0; i < edges.size(); i++)

{

if (added.isAdded(edges[i].first) && !added.isAdded(edges[i].second))

{

add(edges[i].second, false);

added.markAdded(edges[i].second);

}

else if (!added.isAdded(edges[i].first) && added.isAdded(edges[i].second))

{

add(edges[i].first, false);

added.markAdded(edges[i].first);

}

}

}

}

**BTree**(const BTree<ValueType, Compare>& other)

{

copy\_constructor(root, other.root);

}

**~BTree**()

{

destructor(root);

}

void **add**(const ValueType& newElement, bool balanceOn = true)

{

if (root == nullptr)

{

root = new Element(newElement);

valueList.push\_back(newElement);

return;

}

else if (newElement == root->value)

return;

Element \* currEl = root;

while (true)

{

if (newElement > currEl->value)

{

if (currEl->right == nullptr)

{

currEl->right = new Element(newElement);

currEl->right->parent = currEl;

currEl->right->type = RIGHT;

valueList.push\_back(newElement);

break;

}

else if (currEl->right->value == newElement)

break;

currEl = currEl->right;

}

else if (newElement < currEl->value)

{

if (currEl->left == nullptr)

{

currEl->left = new Element(newElement);

currEl->left->parent = currEl;

currEl->left->type = LEFT;

valueList.push\_back(newElement);

break;

}

else if (currEl->left->value == newElement)

return;

currEl = currEl->left;

}

}

if (balanceOn)

balancing(this->root);

}

int **search**(const ValueType& value)

{

Element \* currEl = root;

int height = 0;

while (true)

{

if (currEl == nullptr)

{

height = -1;

break;

}

if (currEl->value == value)

break;

if (value > currEl->value)

{

currEl = currEl->right;

height++;

}

else if (value < currEl->value)

{

currEl = currEl->left;

height++;

}

}

return height;

}

void **remove**(const ValueType& remElement)

{

bool isRemoved = false;

if (root->value == remElement)

{

isRemoved = true;

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

{

delete root;

break;

}

else if (root->left == nullptr)

{

Element \* root\_copy = root;

root = root->right;

root->parent = nullptr;

root->type = ROOT;

delete root\_copy;

break;

}

else if (root->right == nullptr)

{

Element \* root\_copy = root;

root = root->left;

root->parent = nullptr;

root->type = ROOT;

delete root\_copy;

break;

}

else

{

Element \* root\_copy = root;

root = root->left;

root->parent = nullptr;

root->type = ROOT;

while (root->right != nullptr)

root = root->right;

root->right = root\_copy->right;

root->right->parent = root;

root = root\_copy->left;

delete root\_copy;

break;

}

}

Element \* currEl = root;

while (!isRemoved)

{

if (remElement > currEl->value)

{

if (currEl->right == nullptr)

return;

if (currEl->right == remElement)

{

Element \*right\_copy = currEl->right;

currEl->right = currEl->right->right;

currEl->right->parent = currEl;

currEl = currEl->right;

while (currEl->left != nullptr)

currEl = currEl->left;

currEl->left = right\_copy->left;

currEl->left->parent = currEl;

break;

}

currEl = currEl->right;

}

else if (remElement < currEl->value)

{

if (currEl->left == nullptr)

return;

if (currEl->left == remElement)

{

Element \*left\_copy = currEl->left;

currEl->left = currEl->left->left;

currEl->left->parent = currEl;

currEl = currEl->left;

while (currEl->right != nullptr)

currEl = currEl->right;

currEl->right = left\_copy->right;

currEl->right->parent = currEl;

break;

}

currEl = currEl->left;

}

}

valueList.erase(std::find(valueList.begin(), valueList.end(), remElement));

balancing(this->root);

}

std::vector<ValueType> **getCode**()

{

std::vector<ValueType> code;

fillCode(code, root->left);

if (root->right != nullptr)

code.push\_back(root->right->value);

fillCode(code, root->right);

code.erase(code.end() - 1);

code.erase(code.end() - 1);

return code;

}

std::vector<ValueType> **getValueList**()

{

return valueList;

}

ValueType **getRoot**() const

{

return root->value;

}

int **numOfLayers**(Element \* el, int layer = 0)

{

int deepestLeft = layer;

int deepestRight = layer;

if (el->left != nullptr)

deepestLeft = numOfLayers(el->left, layer + 1);

if (el->right != nullptr)

deepestRight = numOfLayers(el->right, layer + 1);

return max(deepestLeft, deepestRight);

}

void **view**()

{

struct ElementToShow

{

int layer;

Element \*element;

ElementToShow()

: layer(0), element(nullptr)

{}

ElementToShow(int layer, Element \*element)

: layer(layer), element(element)

{}

};

int layNum = numOfLayers(root);

std::vector<bool> isLayerVisited(layNum + 1);

std::queue<ElementToShow> tree;

tree.push({ 0, root });

while (!tree.empty())

{

ElementToShow front = tree.front();

tree.pop();

if (isLayerVisited[front.layer] == false)

{

std::cout << std::endl;

}

int indent = sumAllBefore(layNum - front.layer);

if (isLayerVisited[front.layer])

indent = sumAllBefore(layNum - front.layer + 1);

makeIndent(indent);

ElementToShow leftChild;

ElementToShow rightChild;

if (front.element == nullptr)

{

if (front.layer < layNum)

{

leftChild.element = nullptr;

leftChild.layer = front.layer + 1;

rightChild.element = nullptr;

rightChild.layer = front.layer + 1;

tree.push(leftChild);

tree.push(rightChild);

}

std::cout << " ";

}

else

{

leftChild.element = front.element->left;

leftChild.layer=(front.layer< layNum)?front.layer+1:front.layer;

rightChild.element = front.element->right;

rightChild.layer = (front.layer<layNum)?front.layer+1: front.layer;

tree.push(leftChild);

tree.push(rightChild);

std::cout << front.element->value;

}

if (isLayerVisited[front.layer] == false)

{

isLayerVisited[front.layer] = true;

}

}

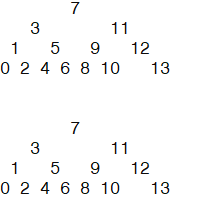
std::cout << std::endl;

}

};

} // Lab6

Результат выполнения:



Вывод: научился кодировать и раскодировать деревья.