**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра математического обеспечения и применения ЭВМ**

отчет

**по лабораторной работе №4**

**по дисциплине «Алгоритмы и структуры данных»  
Тема: Бинарные деревья**

|  |  |  |
| --- | --- | --- |
| Студент гр. 7381 |  | Судакова П.С. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2018

**Задание.**

**Вариант 3-в.**

Заданы два бинарных дерева b1 и b2 типа BT с произвольным типом элементов. Проверить:

а) подобны ли они (два бинарных дерева подобны, если они оба пусты либо они оба не пусты и их левые поддеревья подобны и правые поддеревья подобны);

б) равны ли они (два бинарных дерева равны, если они подобны и их

соответствующие элементы равны);

в) зеркально подобны ли они (два бинарных дерева зеркально подобны, если они оба пусты либо они оба не пусты и для каждого из них левое поддерево одного подобно правому поддереву другого);

г) симметричны ли они (два бинарных дерева симметричны, если они зеркально подобны и их соответствующие элементы равны).

**Пояснение к заданию.**

Нужно написать программу, принимающую на вход два скобочных представления бинарных деревьев (в сокращенной форме), сравнить их и сделать вывод об их подобности, равенстве, зеркальности и симметричности. Правила сокращенной формы скобочного представления деревьев: (<корень> <непустное БД> <пустое БД>) = (<корень> <непустое БД>); ( <корень> <пустое БД> <пустое БД>) = (<корень>); ( <пустое БД>) = (#)

**Описание алгоритма.**

* Алгоритм создания бинарного деревья.

Последовательно рассматриваем каждый символ скобочного представления бинарного дерева. Если текущий элемент содержит в себе непустые поддеревья, то заполняем соответствующими данными ячейку в массиве и рекурсивно вызываем функцию создания БД для детей этого элемента. Если текущий элемент не имеет детей, то заполняем соответствующими данными ячейку в массиве и либо переходим на правого брата этого элемента (если он есть), либо текущий вызов рекурсии завершается.

* Алгоритм сравнения деревьев.

На подобность: сравнение начинается с первого элемента массивов БД. Если левый и правый ребенок первого дерева равен левому и правому ребенку второго, то происходит рекурсивный вызов сначала для левых детей первого и второго дерева, затем для правых детей, иначе рассматриваемые деревья не являются подобными.

На зеркальность: сравнение начинается с первого элемента массивов БД. Если первое дерево имеет левого (правого) ребенка, а второе дерево — правого (левого), то рекурсивно вызываем алгоритм для левого (правого) ребенка первого и правого (левого) ребенка второго. Иначе деревья не являются зеркальными.

**Тестирование.**

Для демонстрации работы программы были написаны тесты и bash-скрипт, последовательно выводящий содержимое очередного теста и результат работы программы для этого теста. В табл. 1 представлены входные и выходные данные всех тестов

Таблица 1 — входные и выходные данные

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| (a(b)(c))  (a(b)(c)) | are similar  are equal  are mirrored  aren't symmetrical |
| (a(b)(c))  (a(c)(b)) | are similar  aren't equal  are mirrored  are symmetrical |
| (a(b(c(d))))  (d#(c#(b#(a)))) | aren't similar  aren't equal  are mirrored  aren't symmetrical |
| (a(b#(c(d#(e)))))  (a#(b(c#(d(e))))) | aren't similar  aren't equal  are mirrored  are symmetrical |
| (bbbb)(R)(U)  (a) | 1 tree: Expression isn't a Binary Tree! |
| (a(b)  (a(b)) | 1 tree: Wrong brackets! |

**Выводы.**

В ходе лабораторной работы была рассмотрена тема бинарных деревьев более детально. Изучено представление бинарного дерева в виде массива, а также как с ним работать. Были получены значения о том какие деревья подобны, равны, зеркально подобны и симметричны.

**ПРИЛОЖЕНИЕ А**

**ИСХОДНЫЙ КОД**

#include <stdio.h>

#include <string.h>

#include <ctype.h>

#include "API.h"

#define MAX\_SIZE 1024

void printSpaces(int deep) { // printSpacess to show recursion's calls

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

printf(" ");

return;

}

int isSimilar(BinTree\* firstTree, BinTree\* secTree, int level, int\* isEqual) {

if (level == 0)

printf("\nCheking for similarity and equality...\n\n");

printSpaces(level);

printf("Comparing: [%d]<-[%c]->[%d] [%d]<-[%c]->[%d]\n", firstTree[0].left,

firstTree[0].data, firstTree[0].right, secTree[0].left, secTree[0].data, secTree[0].right);

if (firstTree[0].data != secTree[0].data) // if elems are equal

\*isEqual = 0;

if (firstTree[0].left == secTree[0].left && firstTree[0].left != -1) { // if left children of elems are similar

if (isSimilar(firstTree + firstTree[0].left - firstTree[0].size,

secTree + secTree[0].left - secTree[0].size, level + 1, isEqual) // if left subtrees aren't similar - return 0

== 0)

return 0;

}

else if (firstTree[0].left != secTree[0].left) { // if left children of elems aren't similar

printf("\naren't similar\naren't equal\n");

return 0;

}

if (firstTree[0].right == secTree[0].right && firstTree[0].right != -1) { // if right children of elems are similar

if (isSimilar(firstTree + firstTree[0].right - firstTree[0].size,

secTree + secTree[0].right - firstTree[0].size, level + 1, isEqual) // / if right subtrees aren't similar - return 0

== 0)

return 0;

}

else if (firstTree[0].right != secTree[0].right) { // if right children of elems aren't similar

printf("\naren't similar\naren't equal\n");

return 0;

}

if (level == 0 && \*isEqual == 0) // if trees are similar and not equal

printf("\nare similar\naren't equal\n");

else if (level == 0 && \*isEqual == 1) // if trees are similar and equal

printf("\nare similar\nare equal\n");

return 1;

}

int isMirrored(BinTree\* firstTree, BinTree\* secTree, int level, int\* isSymm) {

int areMir = 0;

if (level == 0)

printf("\nCheking for mirror similarity and symmetry...\n\n");

printSpaces(level);

printf("Comparing: [%d]<-[%c]->[%d] [%d]<-[%c]->[%d]\n", firstTree[0].left,

firstTree[0].data, firstTree[0].right, secTree[0].left, secTree[0].data, secTree[0].right);

if (firstTree[0].data != secTree[0].data) // if elems is equal

\*isSymm = 0;

if (firstTree[0].left != -1 && secTree[0].right != -1 && firstTree[0].right == -1 && secTree[0].left == -1) { // if left child of tree1 and right child of tree2 aren't empty

areMir = 1;

if (isMirrored(firstTree + firstTree[0].left - firstTree[0].size,

secTree + secTree[0].right - secTree[0].size, level + 1, isSymm) // if left subtree of tree1 and right subtree of tree2 aren't mirrored - return 0

== 0)

return 0;

}

if (firstTree[0].left == -1 && secTree[0].right == -1 && firstTree[0].right != -1 && secTree[0].left != -1) { // if right child of tree1 and left child of tree2 aren't empty

areMir = 1;

if (isMirrored(firstTree + firstTree[0].right - firstTree[0].size,

secTree + secTree[0].left - secTree[0].size, level + 1, isSymm) // if left subtree of tree2 and right subtree of tree1 aren't mirrored - return 0

== 0)

return 0;

}

if (firstTree[0].left != -1 && secTree[0].right != -1 && firstTree[0].right != -1 && secTree[0].left != -1) { // if all children of both trees aren't empty

areMir = 1;

if (isMirrored(firstTree + firstTree[0].left - firstTree[0].size,

secTree + secTree[0].right - secTree[0].size, level + 1, isSymm) // if left subtree of tree1 and right subtree of tree2 aren't mirrored - return 0

== 0)

return 0;

if (isMirrored(firstTree + firstTree[0].right - firstTree[0].size,

secTree + secTree[0].left - secTree[0].size, level + 1, isSymm) // if left subtree of tree2 and right subtree of tree1 aren't mirrored - return 0

== 0)

return 0;

}

if ((firstTree[0].left == -1 && secTree[0].right == -1 && firstTree[0].right == -1 && secTree[0].left == -1)) { // if both trees have no children

if (level == 0 && \*isSymm == 0)

printf("\nare mirrored\naren't symmetrical\n");

if (level == 0 && \*isSymm == 1)

printf("\nare mirrored\nare symmetrical\n");

return 1;

}

if (areMir == 0) {

printf("\naren't mirrored\naren't symmetrical\n");

return 0;

}

if (level == 0 && \*isSymm == 0)

printf("\nare mirrored\naren't symmetrical\n");

if (level == 0 && \*isSymm == 1)

printf("\nare mirrored\nare symmetrical\n");

return 1;

}

int createElem(BinTree\* tree, char data, int size, int isAtom) {

if (size >= MAX\_SIZE) { // if size is more then max size of array - return 0

printf("Max depth of binary tree is 10!\n");

return 0;

}

printf("created: %c\n", data);

tree[size].data = data;

if (isAtom == 1) { // if elem is atom

tree[size].left = -1;

tree[size].right = -1;

tree[size].size = size;

return -1;

}

tree[size].left = size \* 2 + 1;

tree[size].right = size \* 2 + 2;

tree[size].size = size;

return 1;

}

int createTree(BinTree\* tree, char\* str, int level, int size) {

if (level == 0)

printf("\n\nCreating tree...\n");

int LeftRight = 1; // LeftRight = 1, if elem is left; = 2, if elem is right

int isEmpty = 0;

if (level == 0)

LeftRight = 0;

for (int i = 0; i < strlen(str); i++) {

if (str[i] == '#') { // if left elem is empty

LeftRight = 2;

isEmpty = 1;

}

if (str[i] == '(' && (str[i + 2] == '(' || str[i + 2] == '#')) { // if current elem isn't atom

printSpaces(level);

printf("[levele %d] ", level);

if (createElem(tree, str[i + 1], size \* 2 + LeftRight, 0) == 0) // fill array with current elem

return 0;

createTree(tree, str + i + 2, level + 1, size \* 2 + LeftRight); // call function for children of current elem

if (LeftRight == 1)

LeftRight = 2;

else

LeftRight = 1;

i++;

int opened = 1;

int closed = 0;

while (opened != closed) { // skip elems, that are already in array

if (str[i] == '(')

opened++;

if (str[i] == ')')

closed++;

if (opened != closed)

i++;

}

}

if (str[i] == '(' && str[i + 2] == ')') { // if current elem is atom

printSpaces(level);

printf("[level %d] ", level);

if (createElem(tree, str[i + 1], size \* 2 + LeftRight, 1) == 0) // fill array with current elem

return 0;

if (LeftRight == 1)

LeftRight = 2;

else

LeftRight = 1;

}

if (str[i] == ')' && str[i + 1] == ')') { // end of current recursion call

if (isEmpty == 1)

tree[size].left = -1;

if (LeftRight == 2)

tree[size].right = -1;

return 0;

}

}

}

int isCorrect(char\* str) {

int opened = 1;

int closed = 1;

int items = 0;

if (str[0] != '(' && str[strlen(str) - 1] != ')') {

printf("Expression isn't a Tree!\n");

return 0;

}

else {

opened += 1;

closed += 1;

}

for (int i = 1; i < strlen(str) - 1; i++) {

if ((str[i] != '(' && str[i] != ')' && str[i] != '#' && isalpha(str[i]) == 0) || (str[i] == ' ')) { // chek for unacceptable symbols

printf("Wrong symbol: [%c]\n", str[i]);

return 0;

}

if (isalpha(str[i]) || str[i] == '#')

items += 1;

if (str[i] == '(')

opened += 1;

if (str[i] == ')') {

closed += 1;

if (closed > opened) {

printf("Brackets are wrong!\n");

return 0;

}

}

if (isalpha(str[i]) && isalpha(str[i + 1]) || str[i] == '#' && (str[i + 1] == '#' || str[i + 1] != '(' && i != 1)) { // check for correct form of elements

printf("Expression isn't a Binary Tree!\n");

return 0;

}

}

if (opened != closed) { // check for correctness of bracket form of BT

printf("Wrong brackets!\n");

return 0;

}

if (items == 0) {

printf("Tree is empty!\n");

return 0;

}

return items;

}

**ПРИЛОЖЕНИЕ Б**

**КОД BASH-СКРИПТА**

g++ ./Source/main.cpp -std=c++11 -o Lab4

echo -e '\_\_\_\_\_\_\_\nTest 1:'

cat ./Tests/Test1.txt

echo -e '\_\_\_\_\_\_\_\nTesting:\n'

./Lab4 < ./Tests/Test1.txt

echo -e ''

echo -e '\_\_\_\_\_\_\_\nTest 2:'

cat ./Tests/Test2.txt

echo -e '\_\_\_\_\_\_\_\nTesting:\n'

./Lab4 < ./Tests/Test2.txt

echo -e ''

echo -e '\_\_\_\_\_\_\_\nTest 3:'

cat ./Tests/Test3.txt

echo -e '\_\_\_\_\_\_\_\nTesting:\n'

./Lab4 < ./Tests/Test3.txt

echo -e ''

echo -e '\_\_\_\_\_\_\_\nTest 4:'

cat ./Tests/Test4.txt

echo -e '\_\_\_\_\_\_\_\nTesting:\n'

./Lab4 < ./Tests/Test4.txt

echo -e ''

echo -e '\_\_\_\_\_\_\_\nTest 5:'

cat ./Tests/Test5.txt

echo -e '\_\_\_\_\_\_\_\nTesting:\n'

./Lab4 < ./Tests/Test5.txt

echo -e ''

echo -e '\_\_\_\_\_\_\_\nTest 6:'

cat ./Tests/Test6.txt

echo -e '\_\_\_\_\_\_\_\nTesting:\n'

./Lab4 < ./Tests/Test6.txt