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

ОТЧЕТ

по лабораторной работе №5
по дисциплине «Алгоритмы и структуры данных»
Тема: «Бинарное дерево поиска»

Студент гр. 7381 _____ Тарасенко Е. А.
Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2018

Цель работы.

Ознакомиться с такой структурой данных, как бинарное дерево поиска, и научиться применять БДП на практике.

Основные теоретические положения.

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

Декартово дерево или дерамида (Treap) — это структура данных, объединяющая в себе бинарное дерево поиска и бинарную кучу (отсюда и второе её название: treap (tree + heap) и дерамида (дерево + пирамида), также существует название курево (куча + дерево).

Бинарная пирамида поиска (дерамида) – это бинарное дерево, в узлах которого хранятся пары (x, y), где x — это ключ, а y — это приоритет. Также оно является двоичным деревом поиска по x и пирамидой по y. Предполагая, что все x и все y являются различными, получаем, что если некоторый элемент дерева содержит (x0, y0), то у всех элементов в левом поддереве $x < x0$, у всех элементов в правом поддереве $x > x0$, а также и в левом, и в правом поддереве имеем: $y < y0$.

Задание.

Вариант 14:

1) По заданному файлу F (типа file of Elem), все элементы которого различны, построить БДП определённого типа;
2) Для построенного БДП проверить, входит ли в него элемент e типа Elem, и если входит, то удалить элемент e из дерева поиска.

Описание алгоритма.

После получения данных начинается постройка дерева, реализованного на базе списка. Каждый новый элемент добавляется в качестве листа, позже двигается вверх к корню, пока его приоритет не станет меньше приоритета родителя. После построения дерево выводится на экран по средствам консоли.

На следующем этапе происходит поиск элемента ‘e’. Учитывается тот факт, что, если вес корня больше веса элемента для удаления, то искомый элемент будет содержаться в левом поддереве и наоборот, если вес корня меньше, то – в правом. Если элемент найден, то он удаляется из пирамиды, поддеревья перестраиваются по необходимости. Опять же результат изображается в консоли.

Функции и структуры данных.

`class Elem` – класс, в котором хранится информация об элементе БДП.

`int main()` – головная функция программы, в которой происходит инициализация переменных, ввод данных и вызовы функций, необходимых для работы, на всех этапах программы. Там же находится обработка ошибок.

`void CreatePiramid(vector <class Elem>& array, int i)` – функция создания дерамиды. В ней поэтапно связываются между собой элементы списка. Каждый элемент добавляется в качестве листа по своему весу, позже переносится наверх с учетом своего приоритета.

`void DrawPiramid(vector <class Elem>& array)` – функция, рисующая дерамиду. Сначала каждому элементу присваивается его уровень вложенности,

потом формируется сетка из элементов БДП (с учетом пустых). Позже при помощи этой сетки в консоли рисуется дерево.

`void SetLevel(class Elem* head)` – назначает уровни вложенности каждому элементу дерева.

`void Grid(vector <string>& grid, class Elem* head, int maxlvl)` – составляет сетку из элементов.

`void Processing(vector <class Elem>& array)` – ищет элемент ‘e’.

`void Destroy(class Elem* el)` – удаляет элемент ‘e’.

`void Locate(class Elem* par, class Elem* son)` – рекурсивная функция. Производит перестройку поддеревьев после удаления элемента, опираясь на их веса и приоритеты.

Тестирование.

Примеры условий и ответов.

1) Enter a line: abc

   Make random priorities for elements:

   a priority: 2

   b priority: 6

   c priority: 3

   Creating the treap:

   a sons: # # level: 1

   b sons: a c level: 0

   c sons: # # level: 1

   Drawing the treap:

   --------b-------

   ----a--- ----c---

   Searching the 'e' element:

   There isn't 'e' element in a treap!

   a sons: # # level: 1

   b sons: a c level: 0

   c sons: # # level: 1

   Drawing the treap:

--------b-------

----a--- ----c---

2) Enter a line: abcdef

Make random priorities for elements:

a priority: 4

b priority: 7

c priority: 1

d priority: 9

e priority: 11

f priority: 5

Creating the treap:

a sons: # # level: 3

b sons: a c level: 2

c sons: # # level: 3

d sons: b # level: 1

e sons: d f level: 0

f sons: # # level: 1

Drawing the treap:

------------------------e----------------------

---------- d---------- ----------f----------

------b-----

---a-- ---c--

Searching the 'e' element:

Element has been found!

a sons: # # level: 2

b sons: a c level: 1

c sons: # # level: 2

d sons: b f level: 0

sons: d f level: 0

f sons: # # level: 1

Drawing the treap:

----------d----------

--------b------- --------f-------

----a--- ----c---

3) Enter a line:
   ERROR: The data line is empty!
4) Enter a line: aavb
   ERROR: there are too much (> 1) 'a' symbols!
5) Enter a line: abc@
   ERROR: unexpected symbol '@'!

Вывод.

В ходе работы были получены необходимые теоретические знания по работе с БДП. Была составлена программа, которая строит БДП (рандомизированную пирамиду поиска), производит в ней поиск элемента ‘e’ и удаляет его, если таковой был найден.

Приложение А. Код программы.

```
#include "stdafx.h"

#include <iostream>
#include <string>
#include <vector>
#include <cctype>
#include <ctime>
#include <cmath>
#include <cstdlib>

#define LENGTH 8

using namespace std;

class Elem {
public:
	int level;
	char data;
	int priority;
	class Elem* lson;
	class Elem* rson;
	class Elem* parent;

	void StartInit(char d) {
		level = 0;
		data = d;
		priority = 0;
		lson = NULL;
		rson = NULL;
		parent = NULL;
	}
};

void Locate(class Elem* par, class Elem* son) {
	son->parent = par;
	if (par->data > son->data) {
		if (!par->lson) par->lson = son;
		else {
			if (par->lson->priority < son->priority) Locate(son, par->lson);
			else Locate(par->lson, son);
			//par->lson = son;
		}
	}
	if (par->data < son->data) {
		if (!par->rson) par->rson = son;
		else {
			if (par->rson->priority < son->priority) Locate(son, par->rson);
			else Locate(par->rson, son);
			//par->rson = son;
		}
	}
}

void Destroy(class Elem* el) {
	class Elem* tmp = NULL; // will be root instead the "el";
	class Elem* tmp1 = NULL; // another el's son;
	if (!el->lson && !el->rson) {
		if (!el->parent) return;
		if (el->parent->data < el->data) el->parent->rson = NULL;
		else el->parent->lson = NULL;
```

```
			return;
		}
		if (!el->lson) {
			if (!el->parent) return;
			if (el->parent->data < el->data) el->parent->rson = el->rson;
			else el->parent->lson = el->rson;
			return;
		}
		if (!el->rson) {
			if (!el->parent) return;
			if (el->parent->data < el->data) el->parent->rson = el->lson;
			else el->parent->lson = el->lson;
			return;
		}

		if (el->lson->priority > el->rson->priority) {
			tmp = el->lson;
			tmp1 = el->rson;
		}
		else {
			tmp = el->rson;
			tmp1 = el->lson;
		}
		if (el->parent) {
			if (el->parent->data > tmp->data) el->parent->lson = tmp;
			else el->parent->rson = tmp;
		}
		tmp->parent = el->parent;
		Locate(tmp, tmp1);
	}

	void Processing(vector <class Elem>& array) {
		class Elem* tmp = NULL; // head;
		for (int i = 0; i < array.size(); i++) {
			if (array[i].parent == NULL) {
				tmp = &array[i];   // the head has been found;
				break;
			}
		}
		while (tmp) { // searching the element;
			if (tmp->data == 'e') {
				cout << "Element has been found!" << endl;
				Destroy(tmp);
				tmp->data = '#';
				break;
			}
			if (tmp->lson && tmp->data > 'e') {
				tmp = tmp->lson;
				continue;
			}
			if (tmp->rson && tmp->data < 'e') {
				tmp = tmp->rson;
				continue;
			}
			cout << "There isn't 'e' element in a treap!" << endl;
			break;
		}
	}

	void Grid(vector <string>& grid, class Elem* head, int maxlvl) {
		if (head->data != '#') grid[head->level] += head->data;
		else grid[head->level] += " ";
```

```
	if (head->lson) Grid(grid, head->lson, maxlvl);
	else {
		for (int i = 1; i < (grid.size() - head->level); i++)
			for (int j = 0; j < pow(2, i - 1); j++)
				grid[head->level + i] += " ";
	}
	if (head->rson) Grid(grid, head->rson, maxlvl);
	else {
		for (int i = 1; i < (grid.size() - head->level); i++)
			for (int j = 0; j < pow(2, i - 1); j++)
				grid[head->level + i] += " ";
	}
}

void SetLevel(class Elem* head) {
	if (!head->parent) head->level = 0;
	else head->level = head->parent->level + 1;
	if (head->lson) SetLevel(head->lson);
	if (head->rson) SetLevel(head->rson);
}

void DrawPiramid(vector <class Elem>& array) {
	// make levels;
	class Elem* head = NULL;
	int maxlvl = 0;
	for (int i = 0; i < array.size(); i++) {
		if (array[i].parent == NULL && array[i].data != '#') {
			head = &array[i];  // the head has been found;
			SetLevel(head);
			break;
		}
	}
	for (int i = 0; i < array.size(); i++) {
		if (array[i].level > maxlvl)
			maxlvl = array[i].level;
	}
	for (int i = 0; i < array.size(); i++) {
		cout << array[i].data << "\tsons: ";
		if (array[i].lson) cout << array[i].lson->data << " ";
		else cout << "#" << " ";
		if (array[i].rson) cout << array[i].rson->data << "\tlevel: " <<
array[i].level << endl;
		else cout << "#" << "\tlevel: " << array[i].level << endl;
	}
	cout << endl;

	// make a grid;
	vector <string> grid(maxlvl + 1);
	Grid(grid, head, maxlvl);

	// drawing;
	cout << "Drawing the treap:" << endl;
	int length = LENGTH * maxlvl;
	int lvl = 0;
	while (lvl <= maxlvl) {
		for (int j = 0; j < grid[lvl].size(); j++) {
			for (int i = 0; i < length; i++) cout << " ";
			for (int i = 0; i < length; i++) {
				if (grid[lvl][j] != ' ') cout << "-";
				else cout << " ";
			}
			cout << grid[lvl][j];
```

```
                        for (int i = 0; i < length - 1; i++) {
                                if (grid[lvl][j] != ' ') cout << "-";
                                else cout << " ";
                        }
                        for (int i = 0; i < length; i++) cout << " ";
                }
                cout << endl;
                lvl++;
                length /= 2;
        }
        cout << endl;

        // free memory;
        grid.clear();
}

void CreatePiramid(vector <class Elem>& array, int i) {
        //cout << "Creating the treap:" << endl;
        if (i != 0) {
                class Elem* tmp1 = &array[0]; // head;
                for (int j = 0; j < i; j++) {
                        if (array[j].parent == NULL)
                                tmp1 = &array[j]; // found a head;
                }
                // start location of i-element;
                while (1) {
                        if (array[i].data < tmp1->data) {
                                if (!(tmp1->lson)) {
                                        tmp1->lson = &array[i];
                                        array[i].parent = tmp1;
                                        break;
                                }
                                else tmp1 = tmp1->lson;
                        }
                        else {
                                if (!(tmp1->rson)) {
                                        tmp1->rson = &array[i];
                                        array[i].parent = tmp1;
                                        break;
                                }
                                else tmp1 = tmp1->rson;
                        }
                }
                // final location of i-element;
                while (array[i].parent && (array[i].priority > array[i].parent->priority))
{
                        class Elem* tmp = array[i].parent;
                        class Elem* tmpson = NULL;
                        array[i].parent = tmp->parent;
                        if (array[i].parent) {
                                if (array[i].data < array[i].parent->data) array[i].parent-
>lson = &array[i];
                                else array[i].parent->rson = &array[i];
                        }
                        tmp->parent = &array[i];
                        if (array[i].data < tmp->data) {
                                tmp->lson = NULL;
                                if (array[i].rson) tmpson = array[i].rson;
                                array[i].rson = tmp;
                        }
                        else {
                                tmp->rson = NULL;
```

```
                        if (array[i].lson) tmpson = array[i].lson;
                        array[i].lson = tmp;
                    }
                    if (tmpson) {
                        tmpson->parent = tmp;
                        if (tmpson->data < tmp->data) tmp->lson = tmpson;
                        else tmp->rson = tmpson;
                    }
                }
            }
            // the next one;
            if((i + 1) < array.size()) CreatePiramid(array, i + 1);
}

int main() {
            // entering the data;
            string str;
            cout << "Enter a line: ";
            getline(cin, str);
            cout << endl;
            // checking line;
            if (str[0] == '\0') {
                cout << "ERROR: The data line is empty!" << endl;
                return 0;
            }
            for (int i = 0; i < str.size(); i++) {
                if (!isalpha(str[i])) {
                    cout << "ERROR: unexpected symbol '" << str[i] << "'!" << endl;
                    return 0;
                }
                else {
                    str[i] = tolower(str[i]);
                    for (int j = 0; j < i; j++) {
                        if ((str[j] == str[i]) && (j != i)) {
                            cout << "ERROR: there are too much (> 1) '" << str[i]
<< "' symbols!" << endl;
                            return 0;
                        }
                    }
                }
            }

            // creating a data array;
            vector <class Elem> array(str.size());
            // start initialization of elements;
            for (int i = 0; i < str.size(); i++)
                array[i].StartInit(str[i]);
            // creating random priorities;
            srand((unsigned int)time(NULL));
            cout << "Make random priorities for elements:" << endl;
            for (int i = 0; i < str.size(); i++) {
                array[i].priority = 1 + rand() % (str.size() * 2);
                int j = 0;
                while (1) {
                    if ((array[i].priority == array[j].priority) && (i != j)) {
                        array[i].priority = 1 + rand() % (str.size() * 2);
                        j = 0;
                        continue;
                    }
                    j++;
                    if (j >= i) break;
                }
```

```
            cout << array[i].data << "\tpriority: " << array[i].priority << endl;
        }
        cout << endl;

        // creating a treap;
        CreatePiramid(array, 0);
        cout << "Creating the treap:" << endl;
        // drawing a treap;
        DrawPiramid(array);
        // processing;
        cout << "Searching the 'e' element:" << endl;
        Processing(array);
        DrawPiramid(array);
        cout << endl;

        // free memory, ending program;
        array.clear();
    return 0;
}
```