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

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Алгоритмы и структуры данных»

Тема: «Деревья»

Студент гр. 7381 _____ Тарасенко Е. А.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2018

Цель работы.

Ознакомиться с такой структурой данных, как дерево, и научиться применять ее на практике.

Основные теоретические положения.

Дерево – конечное множество T, состоящее из одного или более узлов, таких, что:

а) имеется один специально обозначенный узел, называемый *корнем* данного дерева;

б) остальные узлы (исключая корень) содержатся в $m \geq 0$ попарно не пересекающихся множествах T_1, T_2, ..., T_m, каждое из которых, в свою очередь, является деревом. Деревья T_1, T_2, ..., T_m называются *поддеревьями* данного дерева.

При программировании и разработке вычислительных алгоритмов удобно использовать именно такое *рекурсивное* определение, поскольку рекурсивность является естественной характеристикой этой структуры данных.

Каждый узел дерева является корнем некоторого поддерева. В том случае, когда множество поддеревьев такого корня пусто, этот узел называется *концевым узлом*, или *листом*. *Уровень* узла определяется рекурсивно следующим образом: 1) корень имеет уровень 1; 2) другие узлы имеют уровень, на единицу больший их уровня в содержащем их поддереве этого корня. Используя для уровня узла a дерева T обозначение *уровень* (a, T), можно записать это определение в виде

$$\textit{уровень}(a,T) = \begin{cases} 1, & \text{если } a - \text{корень дерева } T \\ \textit{уровень}(a,T_i)+1, & \text{если } a - \text{не корень дерева } T \end{cases}$$

где T_i – поддерево корня дерева T, такое, что $a \in T_i$.

Говорят, что каждый корень является *отцом* корней своих поддеревьев и что последние являются *сыновьями* своего отца и *братьями* между собой. Также говорят, что узел n – *предок* узла m (а узел m – *потомок* узла n), если n – либо отец m, либо отец некоторого предка m.

Если в определении дерева существен порядок перечисления поддеревьев T_1, T_2, ..., T_m, то дерево называют *упорядоченным* и говорят о «первом» (T_1), «втором» (T_2) и т. д. поддеревьях данного корня. Далее будем считать, что все рассматриваемые деревья являются упорядоченными, если явно

не оговорено противное. Отметим также, что в терминологии теории графов определенное ранее упорядоченное дерево более полно называлось бы «конечным ориентированным (корневым) упорядоченным деревом».

Лес – это множество (обычно упорядоченное), состоящее из некоторого (быть может, равного нулю) числа непересекающихся деревьев. Используя понятие леса, пункт *б* в определении дерева можно было бы сформулировать так: *узлы дерева, за исключением корня, образуют лес*.

Традиционно дерево изображают графически, например, так, как на рис. 1.

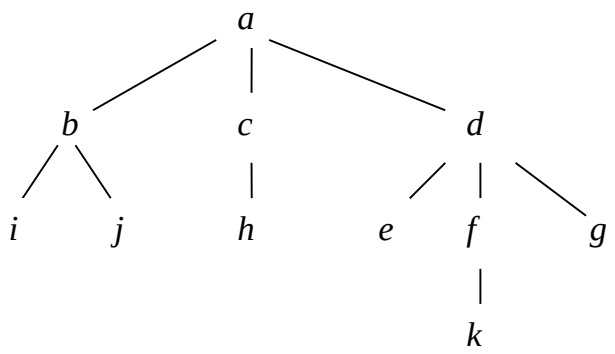

Рис. 1. Графическое изображение дерева

В скобочном представлении дерево, изображенное на рис. 1, запишется как

$$(a\ (b\ (i)\ (j))\ (c\ (h))\ (d\ (e)\ (f\ (k))\ (g))).$$

Наиболее важным типом деревьев являются *бинарные деревья*. Удобно дать следующее формальное определение. *Бинарное дерево* – конечное множество узлов, которое либо пусто, либо состоит из корня и двух непересекающихся бинарных деревьев, называемых правым поддеревом и левым поддеревом. Так определенное бинарное дерево *не* является частным случаем дерева. Например, бинарные деревья, изображенные на рис. 2, различны между собой, так как в одном случае корень имеет пустое правое поддерево, а в другом случае правое поддерево непусто. Если же их рассматривать как деревья, то они идентичны.

Рис. 2. Бинарные деревья из двух узлов

Определим скобочное представление бинарного дерева (БД):

< БД > ::= < пусто > | < непустое БД >,

< пусто > ::= Λ,

< непустое БД > ::= (< корень > < БД > < БД >).

Здесь пустое дерево имеет специальное обозначение – Λ.

Например, бинарное дерево, изображенное на рис. 3, имеет скобочное представление

(*a* (*b* (*d* Λ (*h* Λ Λ)) (*e* Λ Λ)) (*c* (*f* (*i* Λ Λ) (*j* Λ Λ)) (*g* Λ (*k* (*l* Λ Λ) Λ)))).

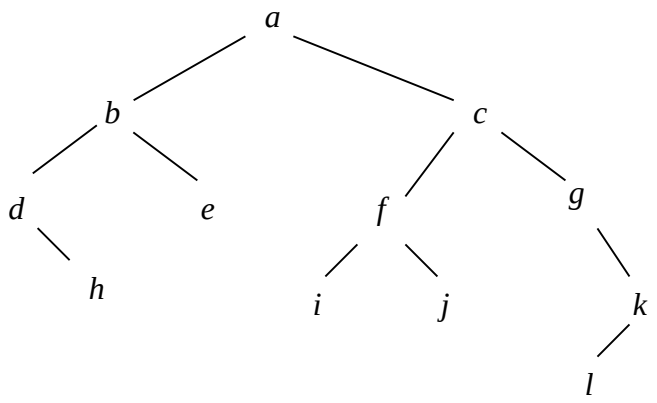

Рис. 3. Бинарное дерево

Можно упростить скобочную запись бинарного дерева, исключив «лишние» знаки Λ по правилам:

1) (< корень > < непустое БД > Λ) ≡ (< корень > < непустое БД >),

2) (< корень > Λ Λ) ≡ (< корень >).

Тогда, например, скобочная запись бинарного дерева, изображенного на рис. 3, будет иметь вид

(*a* (*b* (*d* Λ (*h*) (*e*)) (*c* (*f* (*i*) (*j*)) (*g* Λ (*k* (*l*))))).

Задание.

Вариант 6 - в:

Для заданного бинарного дерева с произвольным типом элементов:

а) получить лес, естественно представленный этим бинарным деревом;

б) вывести изображение бинарного дерева и леса;

в) перечислить элементы леса в горизонтальном порядке (в ширину).

Описание алгоритма.

В ходе алгоритма введенное бинарное дерево записывается в массив, состоящий из элементов класса, которые содержат поля данных, глубины (уровня) и методов для их получения и изменения. Выводится изображение бинарного дерева. Для этого создается массив строк, в каждую из которых записываются все символы, которые должны находиться на уровне, соответствующем номеру строки. Количество строк равно максимальному уровню в дереве. Если каких-то символов нет (пустые символы), то в текущей строке они помечаются как ‘#’; эти же символы записываются во все остальные строки вместо их детей (соответственно в прогрессии). Позже, при помощи скобочной записи бинарного дерева, оно перестраивается в лес с последующим изменением глубин соответствующих элементов. Выводится ответ, позже – последовательность символов, соответствующих горизонтальному обходу полученного леса.

Функции и структуры данных.

`class Atom` – класс, элемент массива узлов дерева и, соответствующего ему, леса. Имеет поля данных и глубины. В нем содержатся конструктор, деструктор, методы получения уровня и данных, их изменения. Для построения элементов класса для любого типа данных используется шаблон.

`int main()` – головная функция программы, в которой происходит инициализация переменных, ввод данных, процесс перестройки дерева в лес и вывод ответов. Там же находится обработка ошибок.

`void push(char data, int level, vector <Atom <char>>& array)` – функция добавления элемента в массив.

`void PrintForest(vector <Atom <char>>& array)` – функция вывода скобочного представления получившегося леса.

`void DrawTree(vector <Atom <char>>& array, int length, int maxlvl)` – функция вывода консольного изображения бинарного дерева.

Тестирование.

Примеры условий и ответов.

1) Please, enter the bin. tree:

   (a#(b(c)#))

   Bin. tree:

   a----------

   ------b

   c

   Processing tree ...

   New level of element 'a': 0

   New level of element 'b': 0

   New level of element 'c': 1

   Forest: (a)(b(c))

   Horizontal detour of the forest: abc

2) Please, enter the bin. tree:

   (a(b#(d))(c(e(g)(h))(f(i)#)))

   Bin. tree:

   ----------a----------

   b-------- --------c--------

   d ----e---- ----f

   g h i

   Processing tree ...

   New level of element 'a': 0

   New level of element 'b': 1

   New level of element 'd': 1

   New level of element 'c': 0

   New level of element 'e': 1

   New level of element 'g': 2

   New level of element 'h': 1

   New level of element 'f': 0

   New level of element 'i': 1

   Forest: (a(b)(d))(c(e(g))(h))(f(i))

   Horizontal detour of the forest: acfbdehig

3) Please, enter the bin. tree:

   (a)

   Bin. tree:

   a

   Processing tree ...

   New level of element 'a': 0

   Forest: (a)

   Horizontal detour of the forest: a

4) Please, enter the bin. tree:

   ERROR: Empty line was entered!

5) Please, enter the bin. tree:

   (a(b)(c))))

   ERROR: Numbers of '(' and ')' aren't equal!

Вывод.

В ходе работы были получены необходимые теоретические знания по работе с деревьями и лесами. Была составлена программа, которая преобразует бинарное дерево в соответствующий ему лес и совершает обход полученного леса в горизонтальном порядке.

Приложение А. Код программы.

```
#include <iostream>
#include <string>
#include <vector>
#include <cmath>

#define L_STD 16  // length standart for drowing;

using namespace std;

template <typename T>
class Atom {      // an element of the array;
public:
      Atom<T>::Atom(T atom, int lvl) {    // constructor;
            data = atom;
            level = lvl;
      }
      Atom<T>::~Atom() {      // destructor;
            data = NULL;
            level = 0;
      }
      // methods;
      void ChgLvl(int lvl) {
            level = lvl;
      }
      int GetLevel() {
            return level;
      }
      T GetData() {
            return data;
      }
private:
      T data;
      int level;
};

void push(char data, int level, vector <Atom <char>>& array) {
      Atom <char> element(data, level);
      array.push_back(element);
}

void PrintForest(vector <Atom <char>>& array) {
      cout << "Forest: ";
      int level = 0;
      for (int i = 0; i < array.size(); i++) {
            if (array[i].GetData() == '#') continue;
            level = array[i].GetLevel();
            cout << "(" << array[i].GetData();
            if ((i + 1) < array.size()) {
                  if(array[i + 1].GetData() != '#') for (int lvl =
array[i].GetLevel(); lvl >= array[i + 1].GetLevel(); lvl--)
                        cout << ")";
                  else if((i + 2) < array.size()) for (int lvl =
array[i].GetLevel(); lvl >= array[i + 2].GetLevel(); lvl--)
                        cout << ")";
```

```
            }
      }
      for (int lvl = level; lvl >= 0; lvl--) cout << ")";
      cout << endl << endl;
}

void DrawTree(vector <Atom <char>>& array, int length, int maxlvl) {
      string grid[100]; // a grid with elements for drawing;
      for (int i = 0; i < array.size(); i++) {
            grid[array[i].GetLevel()] += array[i].GetData();
            if ((i + 1) >= array.size() || ((i + 1) < array.size() && (array[i
+ 1].GetLevel() <= array[i].GetLevel())))
                  for (int n = 1; n <= (maxlvl - array[i].GetLevel()); n++)
                        for (int k = 0; k < pow(2, n); k++)
                              grid[array[i].GetLevel() + n] += '#';
      }
      for (int lvl = 0; lvl <= maxlvl; lvl++, length /= 2){
            for (int n = 0; n < grid[lvl].size(); n++) {
                  for (int j = 0; j < length; j++) cout << " ";
                  if (!(maxlvl - lvl) || grid[lvl][n] == '#' || grid[lvl + 1][n
* 2] == '#')
                        for (int j = 0; j < length; j++) cout << " ";
                  else for (int j = 0; j < length; j++) cout << "-";
                  if (grid[lvl][n] != '#') cout << grid[lvl][n];
                  else cout << " ";
                  if (!(maxlvl - lvl) || grid[lvl][n] == '#' || grid[lvl + 1][n
* 2 + 1] == '#')
                        for (int j = 0; j < length; j++) cout << " ";
                  else for (int j = 0; j < length; j++) cout << "-";
                  for (int j = 0; j < length - 1; j++) cout << " ";
            }
            cout << endl;
      }
      cout << endl;
}

int main() {
      cout << "Please, enter the bin. tree:" << endl;      // initialization and
entering the data;
      string str;
      int level = -1;
      int maxLevel = -1;
      getline(cin, str);
      if (str[0] == '\0') {   // checking an error;
            cout << "\nERROR: Empty line was entered!\n";
            return 0;
      }
      vector <Atom <char>> array;

      for (int i = 0; i < str.size(); i++) {   // reading the line; building
the bin. tree;
            if (str[0] != '(') {    // checking an error;
                  cout << "\nERROR: The symbol '(' was expected!\n";
                  array.clear();
                  return 0;
            }
```

```
            if (str[i] == '(') {
                  level++;
                  continue;
            }
            if (str[i] == ')') {
                  level--;
                  continue;
            }
            if (str[i] == ' ' || str[i] == '\t' || str[i] == '\n') continue;
            if (str[i] == '#') {
                  push(str[i], level + 1, array);
                  continue;
            }
            push(str[i], level, array);
            if (level > maxLevel) maxLevel = level;
            if (str[i + 1] && str[i + 1] != ' ' && str[i + 1] != '(' && str[i +
1] != ')'
                  && str[i + 1] != '#' && str[i + 1] != '\n') {  // checking an
error;
                  cout << "\nERROR: The symbol ')' was expected!\n";
                  array.clear();
                  return 0;
            }
      }

      if (level != -1) {      // checking an errors;
            cout << "\nERROR: Numbers of '(' and ')' aren't equal!\n";
            array.clear();
            return 0;
      }
      if (array.size() == 0) {
            cout << "\nERROR: Empty tree was entered!\n";
            return 0;
      }

      cout << "Bin. tree:\n" << endl;
      DrawTree(array, (maxLevel + 1) * L_STD / 4, maxLevel);     // drawing the
bin. tree;

      int n_arr = 0;    // a number of array element;
      level = -1, maxLevel = 0;
      cout << "Processing tree ..." << endl;
      for (int i = 0; i < str.size(); i++) {   // processing the line; building
the forest;
            if (str[i] == '(') {
                  level++;
                  continue;
            }
            if (str[i] == ')') {
                  if (str[i + 1] && str[i + 1] == '(') level -= 2;
                  continue;
            }
            if (str[i] == ' ' || str[i] == '\n') continue;
            if (str[i] == '#') {
                  level--;
                  continue;
```

```
            }
            if (array[n_arr].GetData() == '#') n_arr++;
            cout << "New level of element '" << array[n_arr].GetData() << "': "
<< level << endl;
            array[n_arr++].ChgLvl(level);
            if (level > maxLevel) maxLevel = level;
      }
      cout << endl;
      PrintForest(array);     // printing a result;

      cout << "Horizontal detour of the forest: ";
      for (level = 0; level <= maxLevel; level++) {
            for (int i = 0; i < array.size(); i++)
                  if (array[i].GetLevel() == level && array[i].GetData() != '#')
                        cout << array[i].GetData();
      }
      cout << endl << endl;

    return 0;
}
```