Отчет по лабораторной работе № 23 по курсу “Алгоритмы и структуры данных ”

Студент группы М80-101Б-22 Шляхтуров Александр Викторович, № по списку 27

Контакты email: shliakhturov@gmail.com

Работа выполнена: «16» марта 202г.

Преподаватель: каф. 806 Крылов Сергей Сергеевич

Входной контроль знаний с оценкой _____

Отчет сдан « » _____202 __ г., итоговая оценка _____

Подпись преподавателя _____

1. **Тема:** Динамические структуры данных. Обработка деревьев.
2. **Цель работы:** Научиться работать с динамическими структурами данных и обрабатывать деревья
3. **Задание** Найти глубину дерева

4. **Оборудование**:

Оборудование ПЭВМ студента, если использовалось:

Процессор **AMD Ryzen 5 5500U 2.10 GHz, 6 ядер** с ОП **8192 Мб**, ТТН **512000 Мб**. Мониторы **Lenovo.**

5. **Программное обеспечение:**

Программное обеспечение ЭВМ студента, если использовалось:

Операционная система семейства **Linux**, наименование **Ubuntu** версия **20.04.5,** интерпретатор команд bash версия 5.0.17(1).

Система программирования CLion версия 2021.1.3

Редактор текстов nano версия 6.2

Утилиты операционной системы WinRar, Microsoft Word.

Прикладные системы и программы Ubuntu wsl, Clion, Google Chrome

Местонахождение и имена файлов программ и данных на домашнем компьютере /home/artur

6. **Идея, метод, алгоритм** решения задачи (в формах: словесной, псевдокода, графической [блок-схема, диаграмма, рисунок, таблица] или формальные спецификации с пред- и постусловиями)

Опишем следующие структуры:

```
struct Node {    int value;    Node*
left;
   Node* right;

   Node(int   val)   {
value = val;       left
= nullptr;         right
= nullptr;
   }

   ~Node() {
      //cout << "DESTROYED " << this->value << endl;
   }
```

}; Структура узла дерева. Хранит значение узла, указатель на левого ребенка и указатель на правого ребенка.

```
struct      BinaryTree      {
Node* root;
   BinaryTree()        {
root = nullptr;
   }
```

```
   void insert(int value) {
if (root == nullptr) {
        root = new Node(value);
      }
else {
        Node* obj = root;
while(true) {
            if (value == obj->value) return;
if (value < obj->value) {
               if (obj->left == nullptr) {
                  obj->left = new Node(value);
return;                    }                    else {
obj = obj->left;
               }
            }
            if (value > obj->value) {
if (obj->right == nullptr) {
                  obj->right = new Node(value);
return;                    }                    else {
obj = obj->right;
               }
            }
         }
      }
   }

   void get_deph(int n, int &mx, Node* obj) {
mx = max(mx, n);        if (obj == nullptr) {
obj = root;
      }          if (obj->left !=
nullptr) {
         get_deph(n + 1, mx, obj->left);
      }
      if (obj->right != nullptr) {
         get_deph(n + 1, mx, obj->right);
      }
   }

   int exist_node(int val, Node* obj) {
if (root == nullptr) return 0;
      if (obj->value == val) return 1;



      if (val < obj->value) {

         if (obj->left != nullptr) {
            return exist_node(val, obj->left);
         }
return 0;

      }                  else {
if (obj->right != nullptr) {
            return exist_node(val, obj->right);
         }
return 0;
      }
   }

   void show(Node* root, int tabs) {
if (root == nullptr) return;
show(root->right, tabs + 1);        for
```

```
(int _ = 0; _ < tabs; _ ++) {          cout
<< '\t';
        }
        cout << root->value << endl;
        show(root->left, tabs + 1);
    }

    void remove(int val) {

        if (this->exist_node(val, this->root) == 0) return;

        Node* obj = root;
int k = -1;       int levels
= 1;
        this->get_deph(1, levels, nullptr);

        // Checking exsisting of obj
        if (obj) {

            // If deleting root
if (obj->value == val) {
if (obj->left == nullptr &&
obj->right == nullptr) {
delete root;                    root =
nullptr;                    return;
            }

            // 1 child
            if (obj->left != nullptr && obj->right == nullptr) {
root = obj->left;                    delete obj;                    return;

            }

            // 1 child
            if (obj->left == nullptr && obj->right != nullptr) {
root = obj->right;                    delete obj;                    return;
            }

            //2 child
            if (obj->left != nullptr && obj->right != nullptr) {

                root = obj->right;

                Node* left_node;
                left_node = root;

                while(true) {
                    if (left_node->left != nullptr) {
                        left_node = left_node->left;
                    }
else {
break;
                    }
                }

                left_node->left = obj->left;
delete obj;                    return;
            }
        }

        while (true) {
```

```
            if    (val    <    obj->value)    {
if (val == obj->left->value) {
                    k      =      1;
break;                              }
obj          =          obj->left;
continue;
            }

            if (val > obj->value) {
               if (val == obj->right->value) {
k = 2;                      break;                  }
obj = obj->right;                   continue;
            }

         }

         if (k == 1) {

            // No child
            if (obj->left->left == nullptr && obj->left->right == nullptr) {
delete obj->left;                   obj->left = nullptr;                    return;
            }

            // 1 child
            if (obj->left->left != nullptr && obj->left->right == nullptr) {
Node* temp;                 temp = obj->left;
               obj->left = obj->left->left;
delete temp;                return;

            }

            // 1 child
            if (obj->left->left == nullptr && obj->left->right != nullptr) {
Node* temp;                 temp = obj->left;                   obj->left = obj-
>left->right;
               delete           temp;
return;
            }

            // 2 child
            if (obj->left->left != nullptr && obj->left->right != nullptr) {
Node* temp;                 temp = obj->left;                   obj->left =
obj->left->right;

               Node* left_node;
               left_node = obj->left;

               while(true) {
                  if (left_node->left != nullptr) {
                     left_node = left_node->left;
                  }
else {
break;
                  }
               }

               left_node->left = temp->left;
delete temp;                return;
            }
         }

         if (k == 2) {
```

```
            // No child
            if (obj->right->left == nullptr && obj->right->right == nullptr) {
delete obj->right;                  obj->right = nullptr;                  return;
            }

            // 1 child
            if (obj->right->left != nullptr && obj->right->right == nullptr) {
Node* temp;                  temp = obj->right;                  obj->right = obj-
>right->left;                delete temp;                  return;
            }

            // 1 child
            if (obj->right->left == nullptr && obj->right->right != nullptr) {
Node* temp;                  temp = obj->right;                  obj->right = obj-
>right->right;                delete temp;                  return;
            }

            // 2 child
            if (obj->right->left != nullptr && obj->right->right != nullptr) {
Node* temp;                  temp = obj->right;                  obj->right = obj-
>right->right;

                Node* left_node;
                left_node = obj->right;

                while(true) {
                   if (left_node->left != nullptr) {
                      left_node = left_node->left;
                   }
else {
break;
                   }
                }

                left_node->left = temp->left;
delete temp;                  return;
            }
         }
      }
   }

   int check_B(Node* root) {

      if ((root->left == nullptr && root->right != nullptr) || (root->left != nullptr && root->right == nullptr)) {
return 0;
      }

      if (root->right != nullptr) {          if
(check_B(root->right)     ==     0)     {
return 0;
         }
      }

      if (root->left != nullptr) {          if
(check_B(root->left) == 0) {
            return 0;
         }
}
      return 1;
   } };
```

Структура самого дерева. Хранит указатель на корень. Внутри
дерева реализованы следующие методы:

- void insert(int value)

Функция добавления узла в дерево.

- void get_deph(int n, int &mx, Node* obj)

Функция получения глубины дерева. Является вспомогательным методом, используется в методе remove.

- int exist_node(int val, Node* obj)

Проверка существования узла. Если узел существует, возвращается единица, иначе, 0. Так же является вспомогательным методом, используется в remove, чтобы проверить, существует ли узел, который мы пытаемся удалить.

- void show(Node* root, int tabs)

Метод для вывода дерева на экран. Дерево обходится рекурсивно, сначала выводится самое большое значение дерева с определенным количеством табов, и далее, все меньшие и меньшие значения. Самым последним будет выведено наименьшее значение (оно будет в самом низу дерева). Дерево выводится не вертикально, а горизонтально

- void remove(int val)

Удаление узла по его значение. Т.е. по атрибуту value объекта класса Node.

int check_B(Node* root)

Метод для определения того, является ли дерево B-деревом.

В основной части программы будем использовать меню, в котором есть 7 опций:

1. Создание дерева (mktree) Создает объект класса BinaryTree.
2. Добавление узла в дерево (insert <значение>)

   Добавляет переданное значение в дерево (если оно создано), иначе выводится сообщение, что дерева не существует
3. Удаление узла дерева (remove <значение>)

   Удаляется вершина, если существует дерево и существует вершина с соответсвующем значением. Иначе, выводится сообщение об ошибке
4. Выполнение задания, проверка, является ли текущее дерево B-деревом (Btree).

   Выводится соответствующее сообщение, об этом дереве

   Вызывает функцию task от корня с максимальным значением 0, а затем выводит ответ.
5. Печать дерева (show)

   Выводи дерева на экран
6. Вывод меню помощи (h)

   Выводятся все доступные в меню команды и их синтаксис
   7. Depth – печатает глубину дерева
   8. Выход (q)

   Выход из меню

7. Сценарий выполнения работы [план работы, первоначальный текст программы в черновике (можно на отдельном листе) и тесты либо соображения по тестированию].

1. **Создать дерево с некоторыми вершинами**
2. **Найти его глубину**

8. Распечатка протокола (подклеить листинг окончательного варианта программы с тестовыми примерами, подписанный преподавателем).

```
#include <bits/stdc++.h>

using namespace std;


struct Node {     int value;     Node* left;

   Node* right;


   Node(int val) {        value = val;        left = nullptr;

      right = nullptr;

   }


   ~Node() {

      //cout << "DESTROYED " << this->value << endl;

   }

};


struct BinaryTree {     Node* root;

   BinaryTree() {

      root = nullptr;

   }


   void insert(int value) {        if (root == nullptr) {           root = new Node(value);

      }       else {

         Node* obj = root;           while(true) {

            if (value == obj->value) return;              if (value < obj->value) {                 if (obj->left ==
nullptr) {                    obj->left = new Node(value);

               return;

            }                else {

               obj = obj->left;

            }

         }
```

```
            if (value > obj->value) {                if (obj->right == nullptr) {                    obj->right = new
Node(value);                    return;
                }                else {
                    obj = obj->right;
                }
            }
        }
    }
}

void get_deph(int n, int &mx, Node* obj) {        mx = max(mx, n);        if (obj == nullptr) {            obj =
root;
    }
    if (obj->left != nullptr) {            get_deph(n + 1, mx, obj->left);
    }
    if (obj->right != nullptr) {            get_deph(n + 1, mx, obj->right);        }
}

int exist_node(int val, Node* obj) {        if (root == nullptr) return 0;        if (obj->value == val) return 1;



    if (val < obj->value) {

        if (obj->left != nullptr) {
            return exist_node(val, obj->left);
        }            return 0;


    }


    else {
        if (obj->right != nullptr) {
            return exist_node(val, obj->right);
        }            return 0;
```

```
    }
  }

  void show(Node* root, int tabs) {       if (root == nullptr) return;       show(root->right, tabs + 1);
for (int _ = 0; _ < tabs; _ ++) {           cout << '\t';
    }
    cout << root->value << endl;       show(root->left, tabs + 1);
  }

  void remove(int val) {

    if (this->exist_node(val, this->root) == 0) return;

    Node* obj = root;       int k = -1;       int levels = 1;
    this->get_deph(1, levels, nullptr);

    // Checking exsisting of obj       if (obj) {

      // If deleting root
      if (obj->value == val) {

        if (obj->left == nullptr && obj->right == nullptr) {                delete root;                root =
nullptr;                return;
        }

        // 1 child
        if (obj->left != nullptr && obj->right == nullptr) {                root = obj->left;                delete
obj;                return;

        }

        // 1 child
```

```
        if (obj->left == nullptr && obj->right != nullptr) {            root = obj->right;            delete
obj;            return;
        }

        //2 child
        if (obj->left != nullptr && obj->right != nullptr) {

          root = obj->right;

          Node* left_node;
          left_node = root;

          while(true) {
            if (left_node->left != nullptr) {                left_node = left_node->left;
            }            else {            break;
            }
          }

          left_node->left = obj->left;            delete obj;            return;
        }
      }

      while (true) {

        if (val < obj->value) {            if (val == obj->left->value) {
          k = 1;            break;
        }            obj = obj->left;            continue;
        }

        if (val > obj->value) {            if (val == obj->right->value) {            k = 2;
break;
          }
          obj = obj->right;            continue;
```

```
            }


        }

        if (k == 1) {

            // No child
            if (obj->left->left == nullptr && obj->left->right == nullptr) {                delete obj->left;
obj->left = nullptr;                return;
            }

            // 1 child
            if (obj->left->left != nullptr && obj->left->right == nullptr) {                Node* temp;
temp = obj->left;                obj->left = obj->left->left;                delete temp;                return;

            }

            // 1 child
            if (obj->left->left == nullptr && obj->left->right != nullptr) {                Node* temp;
               temp = obj->left;                obj->left = obj->left->right;                delete temp;
return;
            }

            // 2 child
            if (obj->left->left != nullptr && obj->left->right != nullptr) {                Node* temp;
temp = obj->left;                obj->left = obj->left->right;

               Node* left_node;
               left_node = obj->left;

               while(true) {
                  if (left_node->left != nullptr) {
                     left_node = left_node->left;
```

```
            }                   else {                      break;

            }

          }


            left_node->left = temp->left;               delete temp;                return;

        }

      }


      if (k == 2) {


        // No child

        if (obj->right->left == nullptr && obj->right->right == nullptr) {              delete obj->right;
obj->right = nullptr;               return;

        }


        // 1 child

        if (obj->right->left != nullptr && obj->right->right == nullptr) {              Node* temp;
temp = obj->right;              obj->right = obj->right->left;              delete temp;                return;

        }


        // 1 child

        if (obj->right->left == nullptr && obj->right->right != nullptr) {              Node* temp;
temp = obj->right;              obj->right = obj->right->right;             delete temp;                return;

        }


        // 2 child

        if (obj->right->left != nullptr && obj->right->right != nullptr) {              Node* temp;
temp = obj->right;

          obj->right = obj->right->right;


          Node* left_node;              left_node = obj->right;


          while(true) {
```

```
                if (left_node->left != nullptr) {                    left_node = left_node->left;                    }
else {                    break;
                }
            }

            left_node->left = temp->left;                delete temp;                return;
        }
      }
    }
  }

  int check_B(Node* root) {

    if ((root->left == nullptr && root->right != nullptr) || (root->left != nullptr && root->right ==
nullptr)) {          return 0;
    }

    if (root->right != nullptr) {          if (check_B(root->right) == 0) {
        return 0;
      }
    }

    if (root->left != nullptr) {          if (check_B(root->left) == 0) {              return 0;
      }
    }
    return 1;
  }
};

int str_validate(string s) {    int key = 1;

  for (char i : s) {
    if (('0' > i || i > '9') && i != '-') {          key = 0;          break;
```

```
        }
    }
    return key;
}


void menu() {
    cout << "h - справка" << endl;
    cout << "q - выход" << endl;
    cout << "mktree - создание дерева" << endl;
    cout << "insert <> - добавление элемента" << endl;
    cout << "remove <> - удаление элемента" << endl;
   // cout << "Btree - проверка, является ли дерево B-деревом" << endl;
    cout << "show - вывод" << endl ;
    cout << "depth - показать глубину дерева" << endl << endl;
    string s;
    BinaryTree bt;
    string value;
    int tree_exists = 0;
    int val;
    const vector<string> CORRECT_INPUT = {"h", "q", "mktree", "insert", "remove", "depth", "show"};
    const int CNT_CORRECT_INPUTS = CORRECT_INPUT.size();


    while(true) {

        cin >> s;

        if (tree_exists && s == "depth"){
            int a;
            bt.get_depth(1, a, nullptr);
            cout << "Глубина дерева равна " << a << endl;
```

```
    continue;
}
if (tree_exists == 0 && s == "depth"){
    cout << "Дерева не существует" << endl;
    continue;
}

// Пользователь захотел выйти
if (s == "q") {
    break;
}

if (s == "h" ) {
    cout << "h - справка" << endl;
    cout << "q - выход" << endl;
    cout << "mktree - создание дерева" << endl;
    cout << "insert <> - добавление элемента" << endl;
    cout << "remove <> - удаление элемента" << endl;
   // cout << "Btree - проверка, является ли дерево B-деревом" << endl;
    cout << "depth - показать глубину дерева" << endl;
    cout << "show - вывод" << endl << endl;
    continue;
}

// Создание дерева
if (tree_exists == 0 && s == "mktree") {
    tree_exists = 1;
    cout << "Дерево создано" << endl;
    continue;
}
if (tree_exists && s == "mktree") {
    cout << "Дерево уже создано" << endl;
```

```
    continue;
}

// Добавление значения
if (tree_exists && s == "insert") {
    cin >> value;

    if (str_validate(value)) {
        val = stoi(value);
    }
    else {
        cout << "Введено не число" << endl;
        continue;
    }

    if (bt.exist_node(val, bt.root) == 1) {
        cout << "Вершина " << val <<  " уже в дереве" << endl;
        continue;
    }

    bt.insert(val);
    cout << "Добавлено значение " << val << endl;
    continue;
}
if (tree_exists == 0 && s == "insert") {
    cin >> value;
    cout << "Дерева не существует, не возможно добавить элемент" << endl;
    continue;
}

// Удаление вершины
if (tree_exists && s == "remove") {
```

```
    cin >> value;
    if (str_validate(value)) {
        val = stoi(value);
    }
    else {
        cout << "Введено не число" << endl;
        continue;
    }

    if (bt.exist_node(val, bt.root) == 0) {
        cout << "Вершины " << val <<  " не существует" << endl;
        continue;
    }

    bt.remove(val);
    cout << "Удалена вершина со значением  " << val << endl;
    if (bt.root == nullptr) {
        tree_exists = 0;
        cout << "Дерево удалено" << endl;
    }
    continue;
}
if (tree_exists == 0 && s == "remove") {
    cin >> value;
    cout << "Дерева не существует, невозможно удалить вершину" << endl;
    continue;
}

if (tree_exists && s == "show") {
    bt.show(bt.root, 0);
    continue;
}
```

```
        if (tree_exists == 0 && s == "show") {
            cout << "Дерева не существует" << endl;
            continue;
        }

        if (tree_exists && s == "Btree") {
            if (bt.check_B(bt.root)) {
                cout << "Это дерево является B-деревом" << endl;
            }
            else {
                cout << "Это дерево является не B-деревом" << endl;
            }
            continue;
        }

        if (tree_exists == 0 && s == "Btree") {
            cout << "Дерева не существует" << endl;
            continue;
        }

        cout << "Ошибка, неверная команда!!!" << endl;
        break;

    }


}

int main() {
    menu();
}
```

```
 alexander@DESKTOP-KNBCFCI:~/LR23$ ./a.out
h - справка
q - выход
mktree - создание дерева
insert <> - добавление элемента
remove <> - удаление элемента
show - вывод
depth - показать степень дерева

insert 5
Дерева не существует, не возможно добавить элемент
mktree
Дерево создано
insert 10
Добавлено значение 10
insert 5
Добавлено значение 5
insert 1
Добавлено значение 1
insert 12
Добавлено значение 12
show
      12
10
      5
            1
remove 5
Удалена вершина со значением  5
show
      12
10
      1
```

depth

Глубина дерева равна 2

q

alexander@DESKTOP-KNBCFCI:~/LR23$ g++ lr23.cpp

alexander@DESKTOP-KNBCFCI:~/LR23$ ./a.out

h - справка

q - выход

mktree - создание дерева

insert <> - добавление элемента

remove <> - удаление элемента

show - вывод

depth - показать глубину дерева

h

h - справка

q - выход

mktree - создание дерева

insert <> - добавление элемента

remove <> - удаление элемента

depth - показать глубину дерева

show - вывод

mktree

Дерево создано

insert 1

Добавлено значение 1

insert 5

Добавлено значение 5

insert 4

Добавлено значение 4

insert 10

Добавлено значение 10

insert 15

Добавлено значение 15

insert 50

Добавлено значение 50

insert 2

Добавлено значение 2

insert 7

Добавлено значение 7

show

```
                50
            15
        10
            7
    5
        4
            2
1
```

remove 5

Удалена вершина со значением 5

show

```
            50
        15
    10
        7
            4
                2
1
```

depth

Глубина дерева равна 5

9. **Дневник отладки** должен содержать дату и время сеансов отладки и основные события (ошибки в сценарии и программе, нестандартные ситуации) и краткие комментарии к ним. В дневнике отладки приводятся сведения об использовании других ЭВМ, существенном участии преподавателя и других лиц в написании и отладке программы.

10. **Замечания автора** по существу работы

11. **Выводы**

Я научился работать с динамическими структурами и обрабатывать деревья

Недочёты при выполнении задания могут быть устранены следующим образом: --

Подпись студента

_____