Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «ФУНДАМЕНТАЛЬНЫЕ НАУКИ»

КАФЕДРА «ПРИКЛАДНАЯ МАТЕМАТИКА»

Лабораторная работа № 6
по дисциплине «Типы и структуры данных»

Тема АВЛ-дерево

Студент Лямин И. С.

Группа ФН12-31Б

Преподаватели Волкова Л.Л.

Москва, 2024

Содержание

ВВЕДЕНИЕ

Цель работы — реализация структуры данных(АВЛ-дерева), реализация алгоритма поиска по структуре.

Осуществление поставленных целей требует выполнения следующих задач:

1) описание алгоритмов для корректной работы АВЛ-дерева;
2) реализация алгоритмов;
3) тестирование полученных реализаций.

1 Аналитическая часть

1.1 АВЛ-дерево

АВЛ-дерево — это один из видов бинарного дерева. Его особенностью заключается в том, что разница высот каждого поддерева не больше двух, такое дерево называется сбалансированным. Это позволяет гарантировать, что операции поиска и добавления элемента будут выполнены за логарифмическое время $O(log\ n)$, где n — высота дерева.

1.2 Балансировка дерева

1.2.1 Малое левое вращение

Малое левое вращение выполняется, когда разница высот между правым (b) и левым (L) поддеревом узла (a) равна 2, и высота левого поддерева узла b (C) меньше либо равна высоте его правого поддерева (R). Для восстановления баланса узел b становится новым корнем, узел a перемещается в левое поддерево b, а поддерево C становится правым поддеревом a. После вращения необходимо обновить высоты поддеревьев следующим образом: высота a становится $max(height(L), height(C)) + 1$, а высота b — $max(height(a), height(R)) + 1$. После вращения баланс поддерева восстанавливается. На рисунке 1.1 изображено графическое пояснение малого левого вращения.

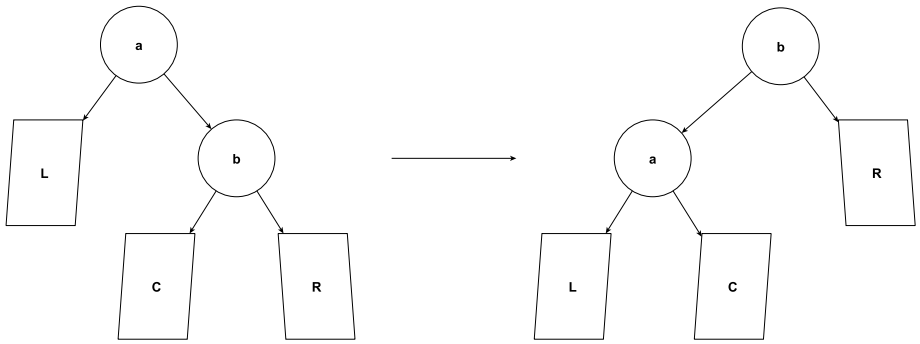

Рисунок 1.1 — Малое левое вращение

1.2.2 Малое правое вращение

Малое правое вращение выполняется, когда разница высот между правым (b) и левым (L) поддеревом узла (a) равна 2, и высота правого поддерева узла b (C) меньше либо равна высоте его левого поддерева (R). Для восстановления баланса узел b становится новым корнем, узел a перемещается в правое поддерево b, а поддерево C становится левым поддеревом a.После вращения необходимо обновить высоты поддеревьев следующим образом: высота a становится $max(height(C), height(R)) + 1$, а высота b — $max(height(a), height(L)) + 1$. После вращения баланс поддерева восстанавливается. На рисунке 1.2 изображено графическое пояснение малого правого вращения.

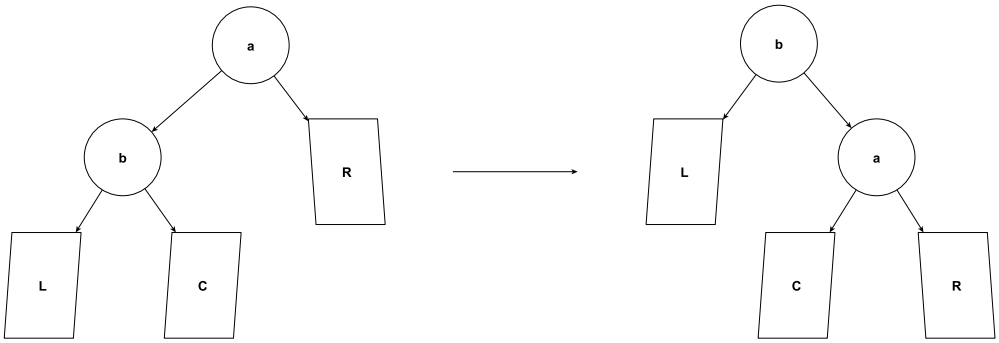

Рисунок 1.2 — Малое правое вращение

1.2.3 Большое левое вращение

Большое левое вращение выполняется, когда разница высот между правым (b) и левым (L) поддеревом узла (a) равна 2, а высота левого поддерева узла b (c) больше высоты его правого поддерева (R). Для восстановления баланса сначала выполняется малое правое вращение относительно узла b: узел c становится корнем правого поддерева узла a, узел b перемещается в правое поддерево c, а поддерево N становится левым поддеревом b. Затем выполняется малое левое вращение относительно узла a: узел c становится новым корнем текущего поддерева, узел a перемещается в левое поддерево c, а узел b остаётся правым поддеревом c. Поддеревья M и N распределяются соответственно. Высоты узлов a, b и c пересчитываются, восстанавливая баланс. На рисунке 1.3 изображено графическое пояснение большого левого вращения.

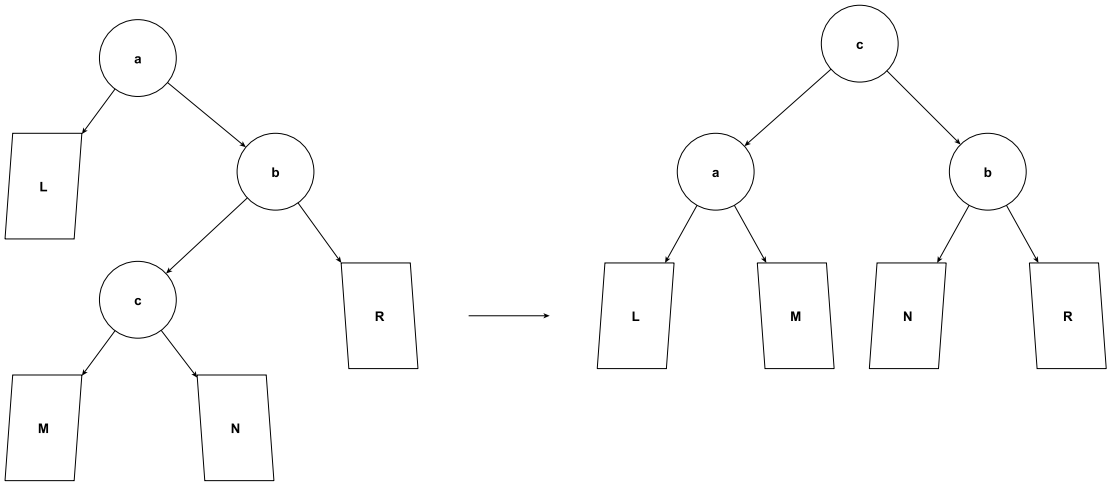

Рисунок 1.3 — Большое левое вращение

1.2.4 Большое правое вращение

Большое правое вращение выполняется, когда разница высот между левым (b) и правым (R) поддеревом узла (a) равна 2, а высота правого поддерева узла b (c) больше высоты его левого поддерева (L). Для восстановления баланса сначала выполняется малое левое вращение относительно узла b: узел c становится корнем левого поддерева узла a, узел b перемещается в левое поддерево c, а поддерево M становится правым поддеревом b. Затем выполняется малое правое вращение относительно узла a: узел c становится новым корнем текущего поддерева, узел a перемещается в правое поддерево c, а узел b остаётся левым поддеревом c. Поддеревья L и N распределяются соответственно. Высоты узлов a, b и c пересчитываются, восстанавливая баланс. На рисунке 1.4 изображено графическое пояснение большого правого вращения.

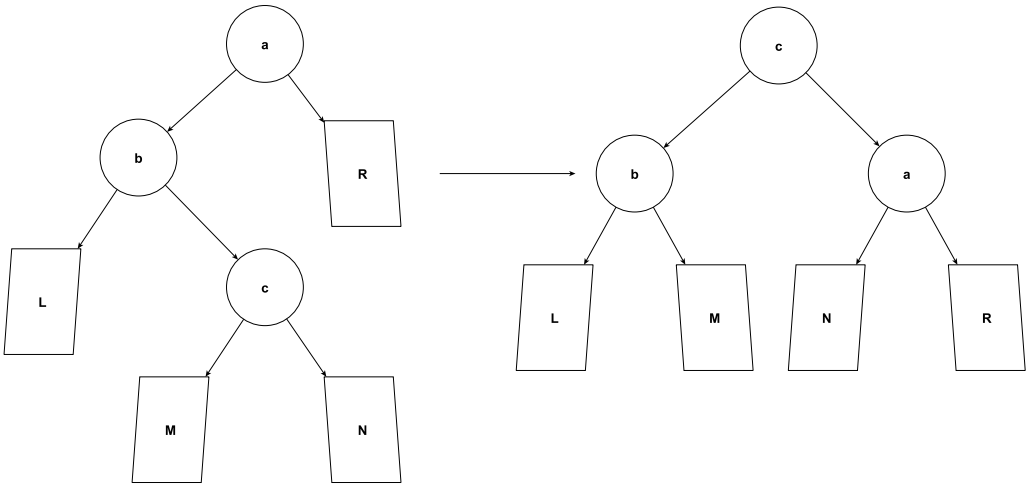

Рисунок 1.4 — Большое правое вращение

2 Конструкторская часть

2.1 Структура узла АВЛ-дерева

Для хранения узлов АВЛ-дерева используется структура `Node`, которая содержит следующие поля:

— `char content` — массив типа `char`, хранящий содержимое узла;

— `int height` — высота поддерева с корнем в данном узле;

— `Node* L, Node* R` — указатели на левое и правое поддерево узла соответственно.

Конструктор структуры `Node`, не принимающий параметров, инициализирует поля следующим образом:

— `content` присваивает `nullptr`;

— `height` инициализируется значением 1;

— указатели `L` и `R` инициализируются `nullptr`.

Конструктор структуры `Node`, принимающий указатель на переменную `content` типа `char` в качестве параметра, инициализируя поля следующим образом:

— `content` присваивается значение параметра;

— `height` инициализируется значением 1;

— указатели `L` и `R` инициализируются `new Node`.

Функция проверки баланса дерева `check_balance(Node* a, Node* b)` принимающая на вход два аргумента и возвращающая разность между высотами этих деревьев.

Функция `compare(char* str1, char* str2)` для сравнения объектов типа `char*`. Функция сравнивает строки с помощью функции `strcmp` библиотеки `cstring`, и возвращает соответствующие значения:

— если `str1 > str2` — возвращает 1;

— если `str1 < str2` — возвращает -1;

— если `str1 == str2` — возвращает 0.

2.2 Функции для работы с высотой и балансом узлов

— `get_peight(Node* node)` — возвращает высоту узла или 0, если узел отсутствует.

— `fix_height(Node* node)` — обновляет высоту узла на основании высот его потомков.

2.3 Вращения

Малое левое вращение

Функция `rot_left(Node* a)` выполняет малое левое вращение узла `a`. Обозначим `root` — корень дерева, `p` ссылка на правое поддерево корня. Алгоритм работы следующий:

1) `root->R = p->L` заменяем правое поддерево корня с `p` на `p->L`;

2) `p->L = root` ставим корень на место левого поддерева узла `p`;

3) обновляется высота узла `root` с учетом новых потомков;

4) обновляется высота узла `p`;

5) возвращается новый корень поддерева `root`.

Малое правое вращение

Функция `rot_left(Node* a)` выполняет малое правое вращение узла `a`. Обозначим `root` — корень дерева, `p` ссылка на левое поддерево корня. Алгоритм работы следующий:

1) `root->L = p->R` заменяем левое поддерево корня с `p` на `p->R`;

2) `p->R = root` ставим корень на место правого поддерева узла `p`;

3) обновляется высота узла `root` с учетом новых потомков;

4) обновляется высота узла `p`;

5) возвращается новый корень поддерева `root`.

Большие вращения

Для больших вращений была реализована функция `balance(Node* root)`, возвращающая указатель на переданный корень, работающая по следующему алгоритму:

1) если `root->L` $\neq$ `nullptr` или `root->R` $\neq$ `nullptr` то вызываем функцию `fix_height(root)`;

2) если высота правого поддерева больше высоты левого на два, то;1

   — если `(root->R->L - root->R->R) > 0` значит совершаем малое правое вращение;

   — совершаем малое левое вращение;

   — возвращает `root`;

3) если высота левого поддерева больше высоты правого на два;

   — если `(root->R->L - root->R->R) > 0` значит совершаем малое правое вращение;

   — совершаем малое левое вращение;

   — возвращает `root`;

4) возвращает `root`.

2.4 Добавление элемента в дерево

Рекурсивная функция `add(char* content, Node* node)` выполняет добавление нового узла в соответствии со следующим алгоритмом:

1) с начала происходит поиск узла для нового элемента;

2) если текущее значение `node->content` равно `nullptr` то присваиваем `node = new`

`Node(content);`

3) после добавления вызывается `balance(node)` для поддержания сбалансированности.

2.5 Поиск элемента в дереве

Функция `search_element(char* element, Node* node)` осуществляет поиск узла со значением `element`:

— если значение текущего узла совпадает с `element`, поиск завершается;

— если `element < node->content`, поиск продолжается в левом поддереве;

— если `element > node->content`, поиск продолжается в правом поддереве.

В процессе поиска выводится путь до найденного узла или сообщение об отсутствии элемента.

2.6 Вывод дерева на экран

Функция `printTree(Node* root)` рекурсивно выводит дерево в консоль.

3 Технологическая часть

3.1 Выбор средств реализации

Для программной реализации алгоритма использовалась среда разработки Visual Studio 2022, язык программирования, на котором была выполнена реализации алгоритмов — C++. Для компиляции кода использовался компилятор MSVC. Исследование проводилось на ноут буке (64–разрядная операционная система, процессор x64, частота процессора 3.1 ГГц, модель процессора 12th Gen Intel(R) Core(TM) i5-12500H, оперативная память 16 ГБ)

3.2 Реализация дерева

В листинге 3.1 представлена программная реализация **АВЛ-дерева**.

Листинг 3.1 — Программная реализация АВЛ-дерева с интерфейсом для пользователя

```
#include <cstring>
#include <iostream>
#include <queue>
#include <iomanip>
using namespace std;


struct Node {
  Node* L = nullptr;
  Node* R = nullptr;
  char* content = nullptr;
  int height;

  Node(char* el) {
    Node* L = new Node;
    Node* R = new Node;
    content = el;
    height = 1;
  }

  Node() {
    content = nullptr;
    height = 1;
  }

  Node& operator=(const Node& rhs) {
    L = rhs.L;
```

```
    R = rhs.R;
    content = rhs.content;
    height = rhs.height;
    return *this;
  }
};

int compare(char* str1, char* str2) {
  if (str1 == nullptr || str2 == nullptr) {
    if (str1 == nullptr && str2 == nullptr) return 0;
    return (str1 == nullptr) ? -1 : 1;
  }
  return strcmp(str1, str2);// -1 --- <; 0 -- ==;  1 --- >;
}

int get_height(Node* p) {
  return p ? p->height : 0;
}

int check_balance(Node* p) {// -2 - left; 0 - norm; 2 - right;
  return (get_height(p->L) - get_height(p->R)) == 0 ? 0 : (get_height(p
     ->L) - get_height(p->R));
}

void fix_height(Node* p) {
  int a = get_height(p->L);
  int b = get_height(p->R);
  p->height = (a > b ? a : b) + 1;
}

Node* rot_right(Node* p) {
  Node* q = p->L;
  p->L = q->R;
  q->R = p;
  fix_height(p);
  fix_height(q);
  return q;
}

Node* rot_left(Node* root) {
  Node* p = root->R;
```

```
  root->R = p->L;
  p->L = root;
  fix_height(root);
  fix_height(p);
  return p;
}

Node* balance(Node* root) {
  if ((root->L) or (root->R)) fix_height(root);
  if (check_balance(root) == -2) {
    if (check_balance(root->R) > 0) root->R = rot_right(root->R);
    return rot_left(root);
  }
  if (check_balance(root) == 2) {
    if (check_balance(root->L) < 0)
    root->L = rot_left(root->L);
    return rot_right(root);
  }
  return root;
}

Node* ADD_ELEMENT(Node* root, char* content) {
  if (root == nullptr) return new Node(content);
  if (compare(content, root->content) == -1) root->L = ADD_ELEMENT(root
     ->L, content);
  else if (compare(content, root->content) == 1) root->R = ADD_ELEMENT(
     root->R, content);
  else cout << "ERROR! This element have already exist.\n";
  return balance(root);
}

void printTree(Node* p) {
  cout << endl;
  if (p == nullptr) {
    cout << "Tree is empty..." << endl;
    return;
  }
  queue<Node*> q;
  q.push(p);
  int k = 128;
  int levelNodes = 1;
```

```
  Node* current = nullptr;
  while (!q.empty()) {
    k = k / 2;
    levelNodes = q.size();
    for (int i = 0; i < levelNodes; ++i) {
      current = q.front();
      q.pop();
      if (current != nullptr) {
        cout << setw(k) << current->content;

        q.push(current->L);
        q.push(current->R);
      }
      else {
        cout << setw(k) << "-";
      }
    }
    cout << endl;
  }
}

bool search_element(char* elem, Node* root) {
  cout << "Searching:\n";
  Node* cur = root;
  while (cur != nullptr) {
    if (compare(elem, cur->content) == -1) {
      cur = cur->L;
      cout << "left ";
    }
    else if (compare(elem, cur->content) == 1) {
      cur = cur->R;
      cout << "right ";
    }
    else if (compare(elem, cur->content) == 0){
      cout << "\nSearched: ";
      char loc = cur->content[0];
      int ind = 0;
      while (loc != '\0') {
        cout << loc;
        ind++;
        loc = cur->content[ind];
```

```
      }
      cout << "\n";
      return true;
    }
  }
  cout << "Element not found!!!!!\n";
  return false;
}

void MENU(Node* root) {
  cout << "Menu: " << endl;
  cout << "1-Exit" << endl;
  cout << "2-Insert element" << endl;
  cout << "3-Search element" << endl;
  cout << "4-Print tree" << endl;
  string el;
  int mode = 0;
  while (mode == 0) {
    cout << "Enter the number: \n";
    cin >> mode;
  }
  char* element = nullptr;
  if (mode == 1) exit(1);
  else if (mode == 2) {
    cout << "Enter the element:\n";
    cin >> el;
    char* element = new char[el.size()];
    for (int i = 0; i < el.size(); i++) {
      element[i] = el[i];
    }
    element[el.size()] = '\0';
    root = ADD_ELEMENT(root, element);
    printTree(root);
    MENU(root);
  }
  else if (mode == 3) {
    cout << "Enter the element for searching:\n";
    cin >> el;
    char* element = new char[el.size()];
    for (int i = 0; i < el.size(); i++) {
      element[i] = el[i];
```

```
    }
    element[el.size()] = '\0';
    search_element(element, root);
    MENU(root);
  }
  else if (mode == 4) {
    printTree(root);
    MENU(root);
  }
}

int main()
{
  setlocale(LC_ALL, "rus");
  Node* root = nullptr;
  MENU(root);
}
```

3.3 Тестирование программы

В таблице 3.1 представлены описания тестов по методологии чёрного ящика, все тесты пройдены успешно.

Таблица 3.1 — Тестирование программы

Описание тестирования	Входные данные	Ожидаемый результат	Полученный результат
Проверка на корректность добавления элементов	a s d g h j abc	h d s a g j - - abc - - - - - -	h d s a g j - - abc - - - - - -
Проверка на поиск элемента в дереве	3 a	left left Searched: a	left left Searched: a
Проверка на поиск несуществующего элемента элемента	3 l	right left right Element not found!!!!!	right left right Element not found!!!!!

Вывод

Было реализовано **АВЛ-дерево**, а также было проведено тестирование работы программы. Все тесты были пройдены успешно.

ЗАКЛЮЧЕНИЕ

В лабораторной работе были описано и реализовано АВЛ-дерево с операциями удаления и вставки. Также было проведено тестирование работы программы.

Для достижения поставленной цели были успешно выполнены основные задачи:

1) описаны алгоритмы для работы АВЛ-дерева;
2) реализованы описанные алгоритмы и операции;
3) протестированы полученные реализации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Сергеев М.И. — АВЛ-деревья, выполнение операций над ними. / Научная статья 2016. С 15-25.