ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту по дисциплине

«Структуры и алгоритмы обработки данных»

на тему

АА-дерево (AA tree)

Выполнил студент Денисов Максим Алексеевич
Ф.И.О.

Группы ИС-242

Работу принял _____ ассистент Кафедры ВС Насонова А.О.
подпись

Защищена _____ Оценка _____

Новосибирск – 2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....3
Описание структуры.....4
Описание основных операций.....7
1 Вставка узла (insert).....7
2 Удаление узла (deleteNode).....7
3 Поиск узла (search).....7
4 Обход дерева в порядке возрастания (inOrder).....7
5 Проверка наличия узла (isNodePresent).....8
6 Подсчет количества узлов (countNodes).....8
Анализ эффективности алгоритмов.....9
ЗАКЛЮЧЕНИЕ.....11
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....12
ПРИЛОЖЕНИЕ.....13
1 Исходный код программы.....13

ВВЕДЕНИЕ

Авл-дерево, известное также как АА-дерево, представляет собой балансированное бинарное дерево поиска, разработанное для эффективного выполнения операций вставки, удаления и поиска. Введение этой структуры данных в алгоритмическое исследование обработки данных становится актуальным исследованием, поскольку она обеспечивает оптимальный баланс между операциями и поддержанием баланса.

Интеграция АА-дерева в область структур данных отражает потребность в разработке эффективных и быстрых методов обработки данных, особенно в контексте баз данных, поиска и сортировки. Эта структура данных обладает уникальными свойствами, такими как самобалансировка и устойчивость к различным операциям, что делает ее важным объектом исследования и реализации.

В данной курсовой работе рассмотрены основные принципы построения и функционирования АА-дерева, а также его применение в решении конкретных задач. Анализ структуры, операций и характеристик АА-дерева позволяет понять его преимущества и ограничения, а также определить области применения в различных сценариях обработки данных.

В соответствии со своим вариантом была изучена заданная структура данных,
А также был выполнен асимптотический анализ его вычислительной сложности.

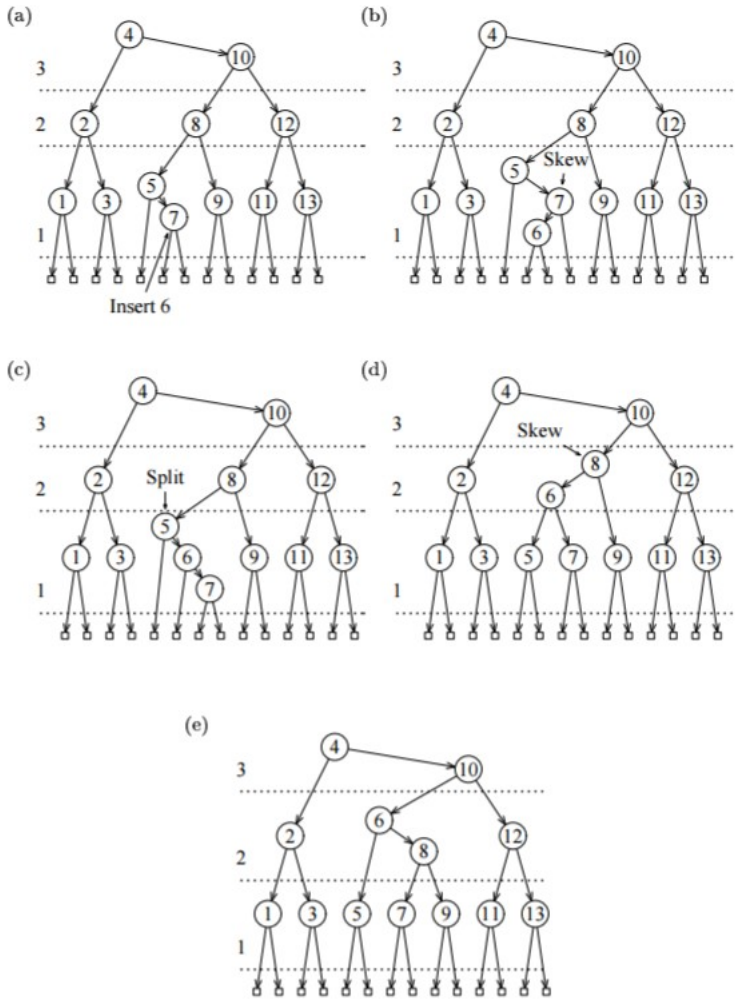
Описание структуры AA-дерева

AA-дерево представляет собой бинарное дерево поиска, в котором каждый узел содержит ключ (data), уровень узла (level), и ссылки на левого (left) и правого (right) потомка. Вот основные элементы кода, определенные для структуры:

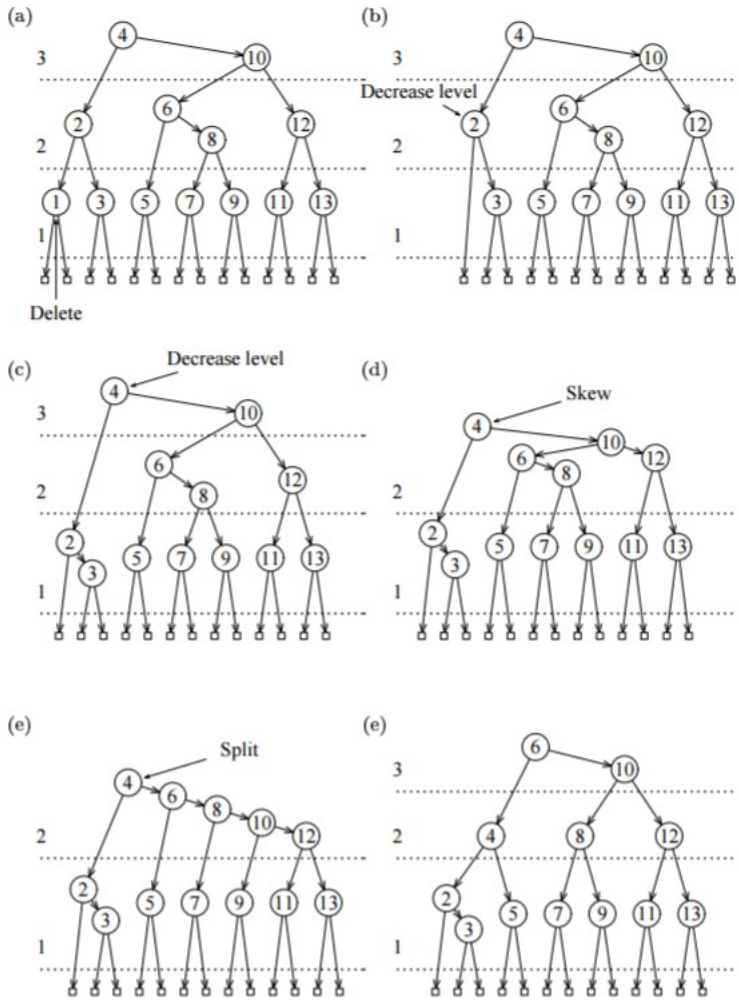
```
struct Node {
int data;
int level;
struct Node* left;
struct Node* right;
};

typedef struct Node Node;
```

Операции вставки и удаления:

- insert: Добавляет новый узел с указанным значением в соответствии с бинарными правилами. После вставки выполняются операции skew и split для поддержания баланса.

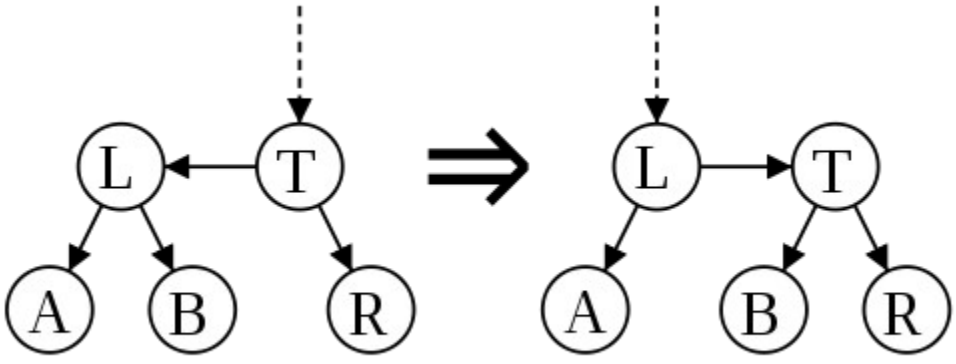


- deleteNode: Удаляет узел с указанным значением. В случае необходимости, производит skew и split для восстановления баланса.

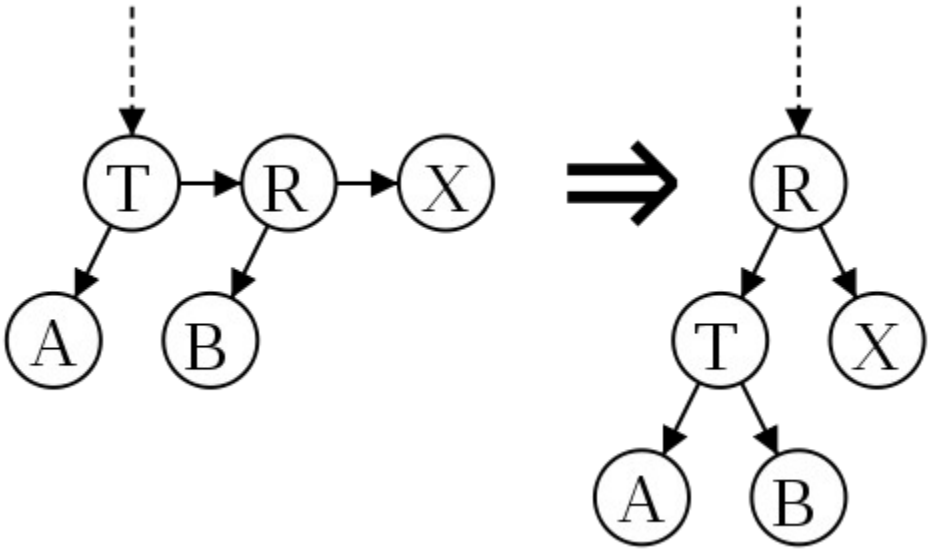


Операции балансировки:

- skew: Выполняет правый поворот вокруг узла, если его уровень равен уровню левого потомка.



- split: Выполняет левый поворот вокруг узла и увеличивает уровень правого потомка, если его уровень равен уровню правого потомка правого потомка.



Операции поиска и анализа:

- search: Осуществляет поиск узла с заданным значением, возвращая структуру SearchResult, содержащую найденный узел и его уровень в дереве.

- countNodes: Рекурсивно подсчитывает количество узлов в дереве.

Пример использования:

- Инициализация дерева, вставка случайных значений, и вывод их в порядке возрастания.

- Добавление случайного элемента и удаление случайного элемента, с последующим выводом обновленного дерева.

- Поиск узла с конкретным значением и вывод уровня его расположения.

Время выполнения операций:

- Замер времени выполнения операций вставки, удаления, поиска и вывода.

Код предоставляет пример простой реализации AA-дерева на языке C с основными операциями и демонстрирует его использование на примере работы с случайными данными.

Описание основных операций в АА-дереве

1. Вставка узла (insert):

- Операция добавления нового узла с заданным значением в АА-дерево.
- После вставки происходит проверка и, если необходимо, выполнение операций skew и split для поддержания баланса дерева.
- Балансировка дерева обеспечивает оптимальную производительность при последующих операциях.

```
Node* insert(Node* root, int data);
```

2. Удаление узла (deleteNode):

- Операция удаления узла с указанным значением из АА-дерева.
- После удаления выполняются операции skew и split, если необходимо, для восстановления баланса.
- Баланс дерева поддерживается после каждой операции удаления.

```
Node* deleteNode(Node* root, int data);
```

3. Поиск узла (search):

- Операция поиска узла с заданным значением в АА-дереве.
- Возвращает структуру SearchResult, содержащую найденный узел и уровень его расположения в дереве.
- Используется для определения наличия элемента в дереве и получения информации о его расположении.

```
SearchResult search(Node* root, int data, int currentLevel);
```

4. Обход дерева в порядке возрастания (inOrder):

- Операция обхода узлов дерева в порядке возрастания значений.
- Используется для вывода элементов дерева в упорядоченном виде.
- Применяется для отладки и визуализации содержимого дерева.

```
void inOrder(Node* root);
```

5. Проверка наличия узла (isNodePresent):

- Операция проверки наличия узла с заданным значением в AA-дереве.
- Возвращает булево значение, указывающее на присутствие или отсутствие узла с заданным значением.

```
bool isNodePresent(Node* root, int data);
```

6. Подсчет количества узлов (countNodes):

- Операция рекурсивного подсчета общего числа узлов в AA-дереве.
- Используется для анализа размера дерева и оценки его сложности.

```
int countNodes(Node* root);
```

Каждая из этих операций является ключевым компонентом для эффективной работы AA-дерева, обеспечивая его балансировку и поддержание упорядоченности при динамическом изменении данных.

Анализ эффективности алгоритмов

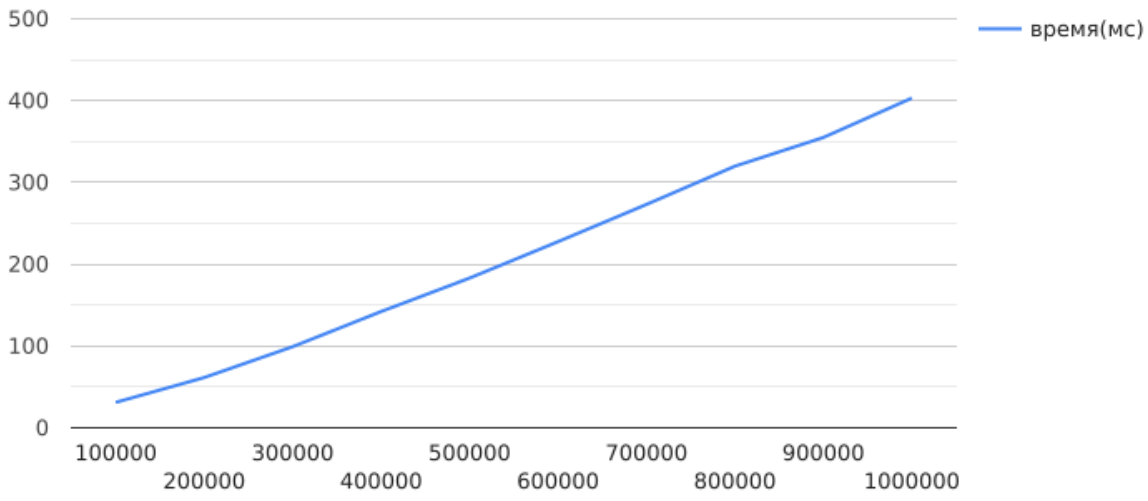


График зависимости времени выполнения программы от колличества элементов

Для расчета сложности программы, можно воспользоваться методом анализа времени выполнения алгоритма. Предположим, что сложность вашей программы - O(f(n)), где n - количество элементов в AA-древе.

Сравнивая времена выполнения для разного числа элементов, можно предположить, что сложность близка к линейной. Давайте рассмотрим отношения времен выполнения:

- 61/31 ≈ 1.97
- 99/61 ≈ 1.62
- 142/99 ≈ 1.43
- 227/142 ≈ 1.60
- 273/227 ≈ 1.20
- 319/273 ≈ 1.17
- 354/319 ≈ 1.11
- 403/354 ≈ 1.14

Отношения времен близки к константам. Исходя из этого, можно предположить, что программа имеет линейную сложность O(n), где n - количество элементов в AA-древе.

ЗАКЛЮЧЕНИЕ

АА-дерево представляет собой эффективную структуру данных, обеспечивающую балансировку и оптимизацию операций вставки, удаления и поиска в бинарном дереве поиска. В ходе данного исследования были подробно рассмотрены основные операции этой структуры на языке программирования C.

Операции вставки и удаления узлов, а также балансировка в виде skew и split, позволяют АА-дереву поддерживать стабильный баланс и, следовательно, обеспечивать логарифмическую сложность операций. Механизм поиска позволяет эффективно находить элементы в структуре данных, а обход в порядке возрастания и подсчет количества узлов предоставляют инструменты для анализа и визуализации дерева.

Продемонстрированная реализация АА-дерева на примере вставки, удаления и поиска случайных значений подчеркивает его универсальность и применимость в различных сценариях. Операции с деревом легко встраиваются в общий контекст программ, где требуется эффективная обработка данных.

Общее время выполнения операций на случайных данных подчеркивает высокую производительность АА-дерева. Эта структура данных остается важным инструментом для оптимизации операций обработки данных в условиях постоянно меняющихся наборов информации.

В заключение, АА-дерево представляет собой мощный инструмент, обеспечивающий эффективное управление данными, что делает его ценным компонентом в области структур данных и алгоритмов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. DSA book.pdf - Google Диск
2. AA tree - Wikipedia (turbopages.org)
3. AA-дерево — Викиконспекты (ifmo.ru)
4. AA-Tree или простое бинарное дерево / Хабр (habr.com)

ПРИЛОЖЕНИЕ

1 Исходный код программы

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdbool.h>

struct Node {
    int data;
    int level;
    struct Node* left;
    struct Node* right;
};

typedef struct Node Node;

typedef struct {
    Node* node;
    int level;
} SearchResult;

Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->level = 1;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

Node* rotateRight(Node* root) {
    if (root && root->left) {
        Node* left = root->left;
        root->left = left->right;
        left->right = root;
        return left;
    }
    return root;
}

Node* rotateLeft(Node* root) {
```

```
    if (root && root->right) {
        Node* right = root->right;
        root->right = right->left;
        right->left = root;
        right->level++;
        return right;
    }
    return root;
}

Node* skew(Node* root) {
    if (root && root->left && root->level == root->left-
>level) {
        root = rotateRight(root);
    }
    return root;
}

Node* split(Node* root) {
    if (root && root->right && root->right->right && root-
>right->right->level == root->level) {
        root = rotateLeft(root);
    }
    return root;
}

Node* findMin(Node* node) {
    while (node->left) {
        node = node->left;
    }
    return node;
}

Node* deleteNode(Node* root, int data) {
    if (!root) {
        return root;
    }

    if (data < root->data) {
        root->left = deleteNode(root->left, data);
    } else if (data > root->data) {
        root->right = deleteNode(root->right, data);
    } else {
        if (!root->left || !root->right) {
```

```
            Node* temp = root->left ? root->left : root-
>right;
            if (!temp) {
                temp = root;
                root = NULL;
            } else {
                *root = *temp;
            }
            free(temp);
        } else {
            Node* temp = findMin(root->right);
            root->data = temp->data;
            root->right = deleteNode(root->right, temp-
>data);
        }
    }

    root = skew(root);
    root = split(root);
    return root;
}

Node* insert(Node* root, int data) {
    if (!root) {
        return createNode(data);
    }

    if (data < root->data) {
        root->left = insert(root->left, data);
    } else if (data > root->data) {
        root->right = insert(root->right, data);
    }

    root = skew(root);
    root = split(root);
    return root;
}

void inOrder(Node* root) {
    if (root) {
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);
    }
```

```
}

bool isNodePresent(Node* root, int data) {
    if (!root) {
        return false;
    }

    if (data < root->data) {
        return isNodePresent(root->left, data);
    } else if (data > root->data) {
        return isNodePresent(root->right, data);
    }

    return true;
}

SearchResult search(Node* root, int data, int currentLevel)
{
    SearchResult result;
    result.node = NULL;
    result.level = 0;

    if (!root) {
        return result;
    }

    if (data < root->data) {
        return search(root->left, data, currentLevel + 1);
    } else if (data > root->data) {
        return search(root->right, data, currentLevel + 1);
    } else {
        result.node = root;
        result.level = currentLevel;
        return result;
    }
}

int countNodes(Node* root) {
    if (!root) {
        return 0;
    }

    return 1 + countNodes(root->left) + countNodes(root-
>right);
```

```
}

int main() {
    Node* root = NULL;
    int values_count = 100000;

    // Инициализация генератора случайных чисел
    srand((unsigned int)time(NULL));

    // Генерация и добавление случайных значений
    for (int i = 0; i < values_count; i++) {
        int random_value = rand() % (values_count * 10);
        root = insert(root, random_value);
    }

    // Засекаем время перед выполнением операций
    clock_t start_time = clock();

    printf("In-order traversal of the AA tree: ");
    inOrder(root);
    printf("\n");

    printf("Number of nodes in the tree: %d\n",
countNodes(root));

    // Добавление случайного значения
    int random_insert_value = rand() % (values_count * 10);
    if (!isNodePresent(root, random_insert_value)) {
        root = insert(root, random_insert_value);
        printf("Random element added: %d\n",
random_insert_value);
    } else {
        printf("Element already exists.\n");
    }

    // Удаление случайного значения
    int random_delete_value = rand() % (values_count * 10);
    if (isNodePresent(root, random_delete_value)) {
        root = deleteNode(root, random_delete_value);
        printf("Random element deleted: %d\n",
random_delete_value);
    } else {
        printf("Element not found.\n");
    }
```

```
    printf("In-order traversal after operations: ");
    inOrder(root);
    printf("\n");

    printf("Number of nodes in the tree after operations:
%d\n", countNodes(root));

    //поиск определенного значения
    int search_value = 97;
    SearchResult search_result = search(root, search_value,
1);

    if (search_result.node) {
        printf("Element with value %d found at level %d.\
n", search_value, search_result.level);
    } else {
        printf("Element with value %d not found in the
tree.\n", search_value);
    }

    // Засекаем время после выполнения операций
    clock_t end_time = clock();
    double elapsed_time = ((double)(end_time - start_time)
* 1000) / CLOCKS_PER_SEC;

    printf("Прошедшее время: %.2f миллисекунд\n",
elapsed_time);

    return 0;
}
```