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 «ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ»

КАФЕДРА «ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 7
ПО ДИСЦИПЛИНЕ «ТИПЫ И СТРУКТУРЫ ДАННЫХ»

<Сбалансированные деревья, хэш таблицы>

Вариант № 4

Студент: ***<Ермаков И.Г>***

Группа: ***<ИУ7-32Б>***

Преподаватель: ***<Барышникова М.Ю>***

2024 г.

Содержание

Цель работы

Цель работы – освоение работы с хеш-таблицами, сравнение эффективности поиска в сбалансированных (AVL) деревьях, в двоичных деревьях поиска и в хештаблицах. Сравнение эффективности устранения коллизий при внешнем и внутреннем хешировании.

Общее задание

Задание: Построить хеш-таблицу и AVL-дерево по указанным данным. Сравнить эффективность поиска в сбалансированном двоичном дереве, в двоичном дереве поиска и в хеш-таблице (используя открытую и закрытую адресацию). Вывести на экран деревья и хеш-таблицы. Подсчитать среднее количество сравнений для поиска данных в указанных структурах. Произвести реструктуризацию хеш-таблицы, если количество сравнений при поиске/добавлении больше указанного. Оценить эффективность использования этих структур (по времени и по памяти) для поставленной задачи. Оценить эффективность поиска в хеш-таблице при различном количестве коллизий и при различных методах их разрешения

Задание по варианту

Построить дерево поиска из слов текстового файла, сбалансировать полученное дерево. Вывести его на экран в виде дерева. Удалить все слова, начинающиеся на указанную букву, в исходном и сбалансированном дереве. Сравнить время удаления, объем памяти. Построить хеш-таблицу из слов текстового файла. Вывести построенную таблицу слов на экран. Осуществить поиск и удаление введенного слова, вывести таблицу. Выполнить программу для различных размерностей таблицы и сравнить время удаления, объем памяти и количество сравнений при использовании сбалансированных деревьев и хеш-таблиц

Описание технического задания

Входные данные

Пользователь вводит название входного файла, затем указывает его размер, файл предназначен для хранения данных для структур данных, описанных в задании, каждый ввод сопровождается приглашением к выводу

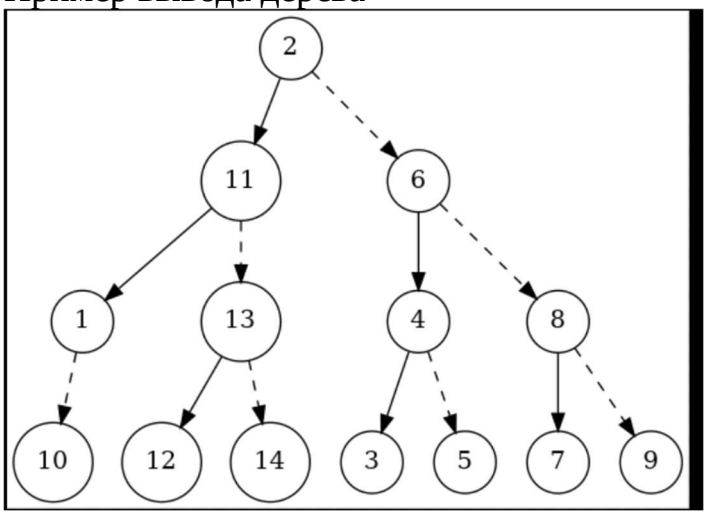
После ввода размера и названия файла, пользователя встречает описание задания(аналогично вышенаписанному) и набор доступных опций

```
Доступные опции меню:
1 - Построить BST и AVL дерево из файла
2 - Вставить слово в BST и AVL деревья
3 - Удалить все слова в деревьях начинающихся на введенный символ
4 - Поиск введенного слова в деревьях
5 - Графический вывод деревьев
6 - Построить хэш таблицы
7 - Добавление в хэш таблицы
8 - Удаление из хэш таблицы
9 - Поиск в хэш таблице
10 - Замеры по времени и памяти
0 - Выход из программы
```

После выбора определенной опции пользователя встречает соответстветствующее приглашение к вводу, например при выборе опции 3 Пользователь должен ввести символ для удаления, для поиска - слово, и тд.

Выходные данные

Тк в качестве визуализации деревьев была выбрана утилита graphgviz, то большиство вывода содержится в файлах, в то же время программа выводит названия файлов в которые были визуализированы деревья после того или иного шага.
Пример вывода дерева



В качестве вывода хэш таблиц был использован стандартный поток вывода, в котором расписывается та или иная хэш таблица и различные характеристики

```
=== СТАТИСТИКА ХЕШ-ТАБЛИЦ ===

Хеш-таблица с ОТКРЫТОЙ адресацией:
Размер таблицы: 21
Содержимое таблицы:
```

```
  [0]: NULL
  [1]: NULL
  [2]: NULL
  [3]: NULL
  [4]: NULL
  [5]: NULL
  [6]: NULL
  [7]: 1
….
Количество сравнений в последней операции: 0

Хеш-таблица с ЗАКРЫТОЙ адресацией:
Размер таблицы: 21
Содержимое таблицы:
  [0]: NULL
  [1]: NULL
  [2]: NULL
  [3]: NULL
  [4]: NULL
  [5]: NULL
  [6]: NULL
  [7]: -> 1
….
Количество сравнений в последней операции: 0
=== КОНЕЦ СТАТИСТИКИ ===
```

Действие программы:
Программа работает до тех пор пока пользователь не введет 0, в случае некорректного ввода программа будет предлагать пользователю ввести еще раз.

Обращение к программе

./app.exe

Аварийные ситуации

Некорректный ввод опции
Удаление из пустого дерева
Удаление из пустой хэш таблицы
Поиск в пустом дереве
Поиск в пустой хэш таблице
Ввод слова вместо символа для удаления

Описание структур данных

```
typedef struct bst
{
   char *value;
   struct bst *left;
   struct bst *right;
} bst_t;
```

```
typedef struct avl
{
   char *value;
   int height;
   struct avl *left;
   struct avl *right;
} avl_t;
```

Данные структуры данных являются определением для BST и AVL деревьев, поле value отвечает за слово в каждом узле, указатели left и right соответственно указывают на потомков родителя, поле height в АВЛ дереве отвечает за высоту каждого узла, данное поле используется для проверки сбалансированности дерева

Так же был реализован интерфейс для работы с деревьями

```
struct tree_interface
{
   void *root;
   void *(*init)(const char *filename);
   void *(*insert)(void *root, const char *value);
   void *(*delete)(void *node, char value, int *comparisons);
   void *(*search)(void *node, const char *value, int *comparisons);
   void (*traversal)(void *root, bool is_measuring);
   avl_t *(*balance)(bst_t *root);
   void (*vusualise)(void *head, const char *filename, bool is_bst);
   void (*destroy)(void *head);
};
```

Данный интерфейс содержит весь базовый функционал для работы с деревьями, и представлен в виде указателей на функции.
Поле root отвечает за корень дерева
Указатель на функцию init содержит подразумевает инициализацию БСТ дерева из файла. Insert – вставка узла в дерево, delete – удаление всех узлов,

начинающихся на заданный пользователем символ, search – поиск узла в дереве, traversal – префиксный обход дерева, balance – функция для балансировки дерева, доступна только в экземпляре АВЛ дерева и балансируется на основе инициализированного БСТ дерева. Vusualise – графическое предствление дерева с использованием утилиты graphviz, destroy – очистка дерева.

Для работы с деревьями так же представлен “конструктор каждого экземпляра”

```
struct tree_interface init_tree_as_bst()
{
    struct tree_interface bst_interface =
    {
        .root = NULL,
        .init = build_tree_from_file,
        .insert = (void *(*)(void *, const char *))insert_bst,
        .delete = (void *(*)(void *, char, int
*))delete_nodes_starting_with_non_balanced,
        .search = (void *(*)(void *, const char *, int *))search_node_bst,
        .traversal = (void (*)(void *, bool))prefix_traversal_bst,
        .balance = NULL,
        .vusualise = (void (*)(void *, const char *, bool))export_to_dot,
        .destroy = (void (*)(void *))free_tree_bst
    };
    return bst_interface;
}
```

```
struct tree_interface init_tree_as_avl()
{
    struct tree_interface avl_interface =
    {
        .root = NULL,
        .init = NULL,
        .insert = (void *(*)(void *, const char *))insert_avl,
        .delete = (void *(*)(void *, char, int *))delete_nodes_starting_with_balanced,
        .search = (void *(*)(void *, const char *, int *))search_node_avl,
        .traversal = (void (*)(void *, bool))prefix_traversal_avl,
        .balance = (avl_t *(*)(bst_t *))balance_tree,
        .vusualise = (void (*)(void *, const char *, bool))export_to_dot,
        .destroy = (void (*)(void *))free_tree_avl

    };
    return avl_interface;
```

```
}
```

Эти два конструктора инициализируют интерфейс для каждого экземпляра дерева.

Далее представлены структуры хэш таблиц

```
typedef struct hash_node
{
   char word[MAX_WORD_LENGTH];
   struct hash_node *next;
} hash_node_t;
```

Отдельный экземпляр узла хэш таблицы, содержит слово и указатель на следующий элемент (для закрытой адресации)
MAX_WORD_LENGTH = 100

Структура для хэш таблицы с открытой адресацией

```
typedef struct opened
{
   int size;
   char** open_table;
   int* status;
   int comparisons;
} open_ht;
```

Содержит размер таблицы, массив слов, так же содержит целочисленный массив status в котором хранится информация о каждом элементе хэш таблицы
0 – не занят
1 – занят
2 – удален

Последним параметром является поле comparisons отвечающее за кол-во сравнений для той или иной операции.

Структура для хэш таблицы с закрытой адресацией

```
typedef struct closed
{
   int size;
   hash_node_t **closed_table;
```

```
    int comparisons;
} closed_ht;
```

Так же содержит размер таблицы, поле closed_table является предствлением метода “цепочек”. И аналогичное поле для учета кол-ва сравнений для той или иной операции

Описание основных сигнатур функций

Функции для операций с деревьями

```
bst_t *create_node_bst(char *value);
void *insert_bst(void *head, char *value);
bst_t *delete_node_bst(bst_t *node);
void *search_node_bst(void *head, const char *value, int *comparisons);
void prefix_traversal_bst(void *head, bool is_measuring);


avl_t *balance_tree(bst_t *root);
avl_t *create_node_avl(const char *value);
void *insert_avl(void *head, const char *value);
void *search_node_avl(void *head, const char *value, int *comparisons);
void *delete_node_avl(void *head);
void prefix_traversal_avl(void *head, bool is_measuring);
```

Создание узла, вставка узла, удаление одного узла, поиск, обход

Функции для операций с хэш таблицами

```
open_ht* create_open_table(int size);
void insert_open(open_ht *table, const char *word);
char* search_open(open_ht* table, const char* word);
void delete_open(open_ht* table, const char* word);


closed_ht* create_closed_table(int size);
void insert_closed(closed_ht* table, const char* word);
hash_node_t* search_closed(closed_ht* table, const char* word);
void delete_closed(closed_ht* table, const char* word);
void free_closed_table(closed_ht* table);
void free_open_table(open_ht* table);

void read_file_to_hts(const char* filename, closed_ht* closed_table, open_ht*
open_table);
```

Создание таблицы, вставка, поиск, удаление, заполнение таблиц из файла

Описание алгоритма

Деревья

Удаление узла в БСТ:
1. Если узел - лист, просто удаляем его.
2. Если у узла один потомок, заменяем узел на его потомка.
3. Если два потомка, ищем минимум в правом поддереве (или максимум в левом), заменяем значение удаляемого узла на найденное и удаляем узел-замену рекурсивно.

Удаление узла в АВЛ:
1. Удаление такое же как в БСТ.
2. После удаления пересчитываем баланс-фактор для всех узлов на пути вверх от удаленного. 3. Если баланс-фактор узла становится -2 или 2, выполняем вращение для восстановления баланса.

Перебалансировка АВЛ:
1. LL (левый левый): правое вращение.
2. RR (правый правый): левое вращение.
3. LR (левый правый): сначала левое вращение для левого поддерева, затем правое вращение. 4. RL (правый левый): сначала правое вращение для правого поддерева, затем левое вращение. После вращения пересчитываются высоты узлов.

Хэш таблица

Реализация хэш функции

```
unsigned hash(const char *str, int size)
{
    unsigned hash_value = 0;
    while (*str)
        hash_value = (hash_value * 31 + *str++) % size;
    return hash_value;
}
```

Хэш-функция: функция берет строку и преобразует её в хэш-значение. Для этого она начинает с нуля и проходит по каждому символу строки. На каждой итерации текущее значение хэша умножается на 31, прибавляется код символа, после чего берется остаток от деления на размер таблицы. Это помогает распределить строки по таблице более равномерно, минимизируя коллизии.

Использование 31 как множителя стандартно для текстовых данных, так как это простое число и помогает в распределении.

Открытая адресация. В реализации таблица создается с полями для хранения данных, статуса ячеек (0 — свободна, 1 — занята, 2 — удалена) и счётчика сравнений. Для вставки сначала вычисляется индекс через хэш-функцию, затем проверяется, свободна ли ячейка. Если занята, используется линейное пробирование: идём к следующей ячейке, пока не найдём свободную. При поиске аналогично: проверяем индекс от хэш-функции и продолжаем искать, если не нашли совпадение. Для удаления ищем нужное слово, освобождаем память, помечаем ячейку как удалённую (статус 2). Это сделано, чтобы не нарушить поиск других элементов.

Закрытая адресация. В этой реализации таблица хранит массив указателей на связанные списки (цепочки). Для вставки сначала вычисляется индекс через хэш-функцию, после чего создаётся новый узел и добавляется в начало списка в соответствующей ячейке. При поиске идём по списку в ячейке, сравниваем каждое слово с искомым. Для удаления идём по списку и ищем нужный узел. Если нашли, перенастраиваем указатели так, чтобы исключить этот узел, и освобождаем память. Если удаляется первый узел в списке, просто перемещаем указатель ячейки на следующий элемент.

Сравнение эффективности работы алгоритмов

Методология исследования – для каждого замера времени на каждый размер проводится по 1000 измерений, затем значение усредняется.

Таблица замеров по времени для удаления в АВЛ и БСТ деревьях

Размер	Время для удаления в БСТ дереве (такты)	Время для удаления в АВЛ дереве (такты)
100	1366	1092
250	4223	4218
500	7257	6579
1000	15598	11959
2000	56795	36070

Таблица замеров по времени для поиска в АВЛ и БСТ деревьях

Размер	Время для поиска в БСТ дереве (такты)	Время для поиска в АВЛ дереве (такты)
100	447	224
250	552	233
500	637	302
1000	699	349
2000	969	399

Таблица замеров по времени для удаления в хэш таблицах с открытой и закрытой адресацией

Размер	Время для удаления в хэш таблице с открытой адресацией (такты)	Время для удаления в хэш таблице с закрытой адресацией (такты)
100	145	70
250	185	54
500	373	40
1000	435	66
2000	150	53

Таблица замеров по времени для поиска в хэш таблицах с открытой и закрытой адресацией

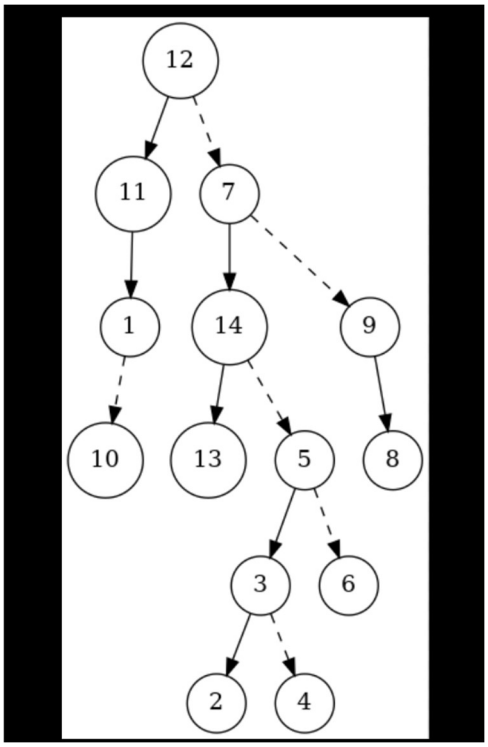
Размер	Время для поиска в хэш таблице с открытой адресацией (такты)	Время для поиска в хэш таблице с закрытой адресацией (такты)
100	235	47
250	197	44
500	216	28
1000	341	51
2000	133	45

Таблица замеров для кол-ва сравнений для поиска в АВЛ дереве и хэш таблицах

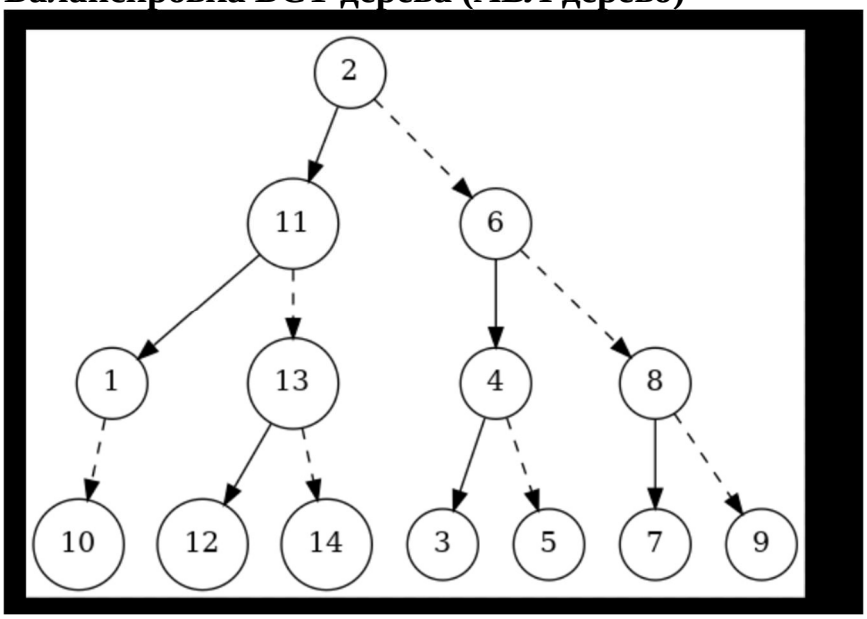
Размер	Кол-во сравнений в АВЛ дереве	Кол-во сравнений в хэш таблице с открытой адресацией	Кол-во сравнений в хэш таблице с закрытой адресацией
100	7	1	1
250	7	1	1
500	9	9	1
1000	9	7	3
2000	17	1	1

Тесты

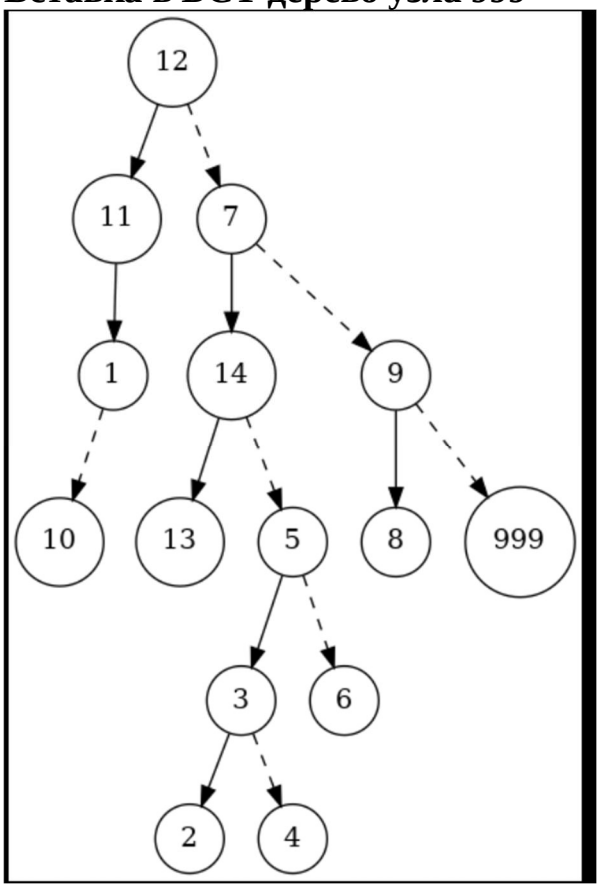
Генерация БСТ дерева



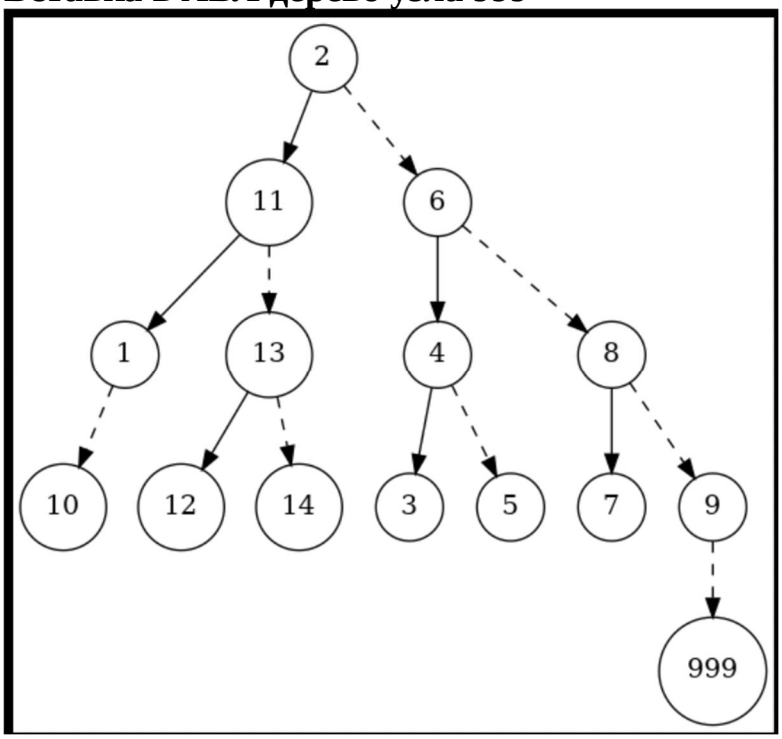
Балансировка БСТ дерева (АВЛ дерево)



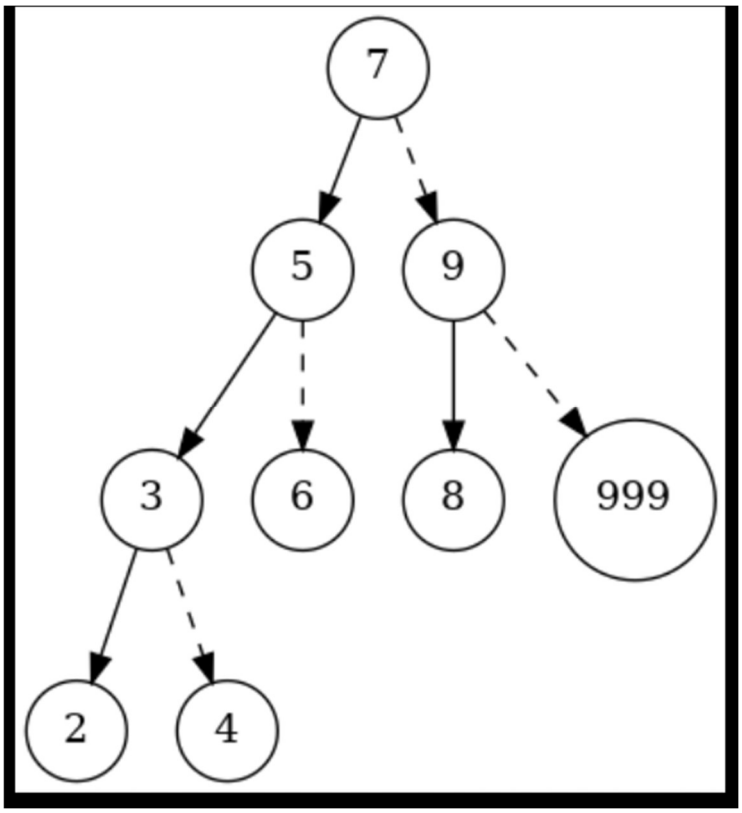
Вставка в БСТ дерево узла 999



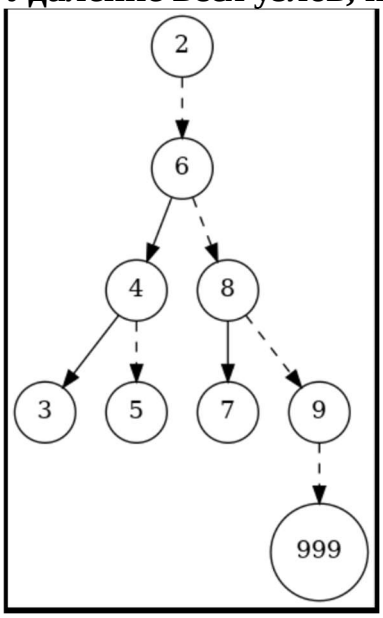
Вставка в АВЛ дерево узла 999



Удаление всех узлов, начинающихся на символ ‘1’ в БСТ дереве



Удаление всех узлов, начинающихся на символ ‘1’ в АВЛ дереве



Поиск элемента ‘5’ в АВЛ и БСТ деревьях

```
Кол-во сравнений для BST : 3
Кол-во сравнений для AVL : 3
Слово найдено в BST!
Слово найдено в AVL!
```

Поиск элемента ‘777’ в АВЛ и БСТ деревьях

```
Кол-во сравнений для BST : 2
Кол-во сравнений для AVL : 2
Слово не найдено в BST!
Слово не найдено в AVL!
```

Построение хэш таблиц

```
=== СТАТИСТИКА ХЕШ-ТАБЛИЦ ===

Хеш-таблица с ОТКРЫТОЙ адресацией:
Размер таблицы: 15
Содержимое таблицы:
  [0]: NULL
  [1]: NULL
  [2]: NULL
  [3]: NULL
  [4]: 1
  [5]: 2
  [6]: 3
  [7]: 4
  [8]: 5
  [9]: 10
  [10]: 7
  [11]: 8
  [12]: 6
  [13]: 9
  [14]: NULL
Количество сравнений в последней операции: 0

Хеш-таблица с ЗАКРЫТОЙ адресацией:
Размер таблицы: 15
Содержимое таблицы:
  [0]: NULL
  [1]: NULL
  [2]: NULL
  [3]: NULL
  [4]: -> 1
  [5]: -> 2
  [6]: -> 3
  [7]: -> 10 -> 4
  [8]: -> 5
  [9]: -> 6
  [10]: -> 7
  [11]: -> 8
  [12]: -> 9
  [13]: NULL
  [14]: NULL
Количество сравнений в последней операции: 0
=== КОНЕЦ СТАТИСТИКИ ===
```

Вставка узла 999 в хэш таблицы
=== СТАТИСТИКА ХЕШ-ТАБЛИЦ ===

Хеш-таблица с ОТКРЫТОЙ адресацией:
Размер таблицы: 15
Содержимое таблицы:
  [0]: NULL
  [1]: NULL
  [2]: NULL
  [3]: NULL
  [4]: 1
  [5]: 2
  [6]: 3
  [7]: 4
  [8]: 5
  [9]: 10
  [10]: 7
  [11]: 8
  [12]: 6
  [13]: 9
  [14]: 999
Количество сравнений в последней операции: 0

Хеш-таблица с ЗАКРЫТОЙ адресацией:
Размер таблицы: 15
Содержимое таблицы:
  [0]: NULL
  [1]: NULL
  [2]: NULL
  [3]: NULL
  [4]: -> 1
  [5]: -> 2
  [6]: -> 999 -> 3
  [7]: -> 10 -> 4
  [8]: -> 5
  [9]: -> 6
  [10]: -> 7
  [11]: -> 8
  [12]: -> 9
  [13]: NULL
  [14]: NULL
Количество сравнений в последней операции: 0
=== КОНЕЦ СТАТИСТИКИ ===

Удаление элемента ‘3’ из таблиц
=== СТАТИСТИКА ХЕШ-ТАБЛИЦ ===

Хеш-таблица с ОТКРЫТОЙ адресацией:
Размер таблицы: 15
Содержимое таблицы:
  [0]: NULL
  [1]: NULL
  [2]: NULL
  [3]: NULL
  [4]: 1
  [5]: 2
  [6]: NULL
  [7]: 4
  [8]: 5
  [9]: 10
  [10]: 7
  [11]: 8
  [12]: 6
  [13]: 9
  [14]: 999
Количество сравнений в последней операции: 1

Хеш-таблица с ЗАКРЫТОЙ адресацией:
Размер таблицы: 15
Содержимое таблицы:
  [0]: NULL
  [1]: NULL
  [2]: NULL
  [3]: NULL
  [4]: -> 1
  [5]: -> 2
  [6]: -> 999
  [7]: -> 10 -> 4
  [8]: -> 5
  [9]: -> 6
  [10]: -> 7
  [11]: -> 8
  [12]: -> 9
  [13]: NULL
  [14]: NULL
Количество сравнений в последней операции: 2
=== КОНЕЦ СТАТИСТИКИ ===

Поиск элемента ‘3’ в хэш таблице

```
Слово не найдено в открытой хэш-таблице!
Слово не найдено в закрытой хэш-таблице!
```

Поиск элемента ‘999’ в хэш таблице

```
Слово найдено в открытой хэш-таблице!
Слово найдено в закрытой хэш-таблице!
```

Выводы

Структуры АВЛ и БСТ:
В БСТ каждое значение хранится в узле, а левый потомок содержит значения меньше текущего узла, правый — больше. В АВЛ дереве структура та же, но дополнительно хранится поле высоты, чтобы контролировать баланс.

Времена выполнения операций:
Поиск: БСТ — O(h), где h — высота дерева. Если дерево несбалансированное, высота может быть O(n).
В АВЛ поиск O(log n), так как дерево всегда сбалансировано.

Вставка: БСТ — O(h). В АВЛ тоже O(log n), но включает дополнительные шаги для перебалансировки, что немного увеличивает время.

Удаление: БСТ — O(h). В АВЛ O(log n), так как после удаления может потребоваться перебалансировка.

Память: БСТ требует памяти только для хранения узлов (value, left, right). АВЛ дополнительно хранит высоту каждого узла, что увеличивает использование памяти. Однако, это увеличение минимально (примерно на 4 байта на узел).

Плюсы и минусы:

БСТ:
Плюсы: Простая реализация, эффективна, если данные случайно распределены.
Минусы: Без балансировки может превратиться в линейный список, и тогда операции будут работать медленно.

АВЛ:
Плюсы: Гарантированная сбалансированность, всегда обеспечивает O(log n) для всех операций.
Минусы: Сложнее реализация, немного увеличенные накладные расходы по времени и памяти из-за контроля баланса.
Вывод: БСТ подходит для случаев, когда данные добавляются случайным образом, и нет необходимости в гарантированной производительности. АВЛ деревья нужны для задач, где важно, чтобы операции выполнялись быстро, независимо от порядка данных.

Хэш-таблицы:
Открытая адресация и закрытая адресация — это методы обработки коллизий в хэш-таблицах. Основное различие в том, что при открытой адресации все

элементы хранятся в массиве, а при закрытой — элементы с одним и тем же хэшем хранятся в списках.

Времена выполнения операций:
Поиск: В идеале O(1) для обеих реализаций. Однако при открытой адресации поиск может занять O(n) в случае высокой заполненности таблицы. В закрытой адресации — O(1 + k), где k — длина цепочки (обычно малая).

Вставка: В открытой адресации — O(1) в идеале, но O(n) при переполнении. В закрытой — O(1), так как элемент добавляется в начало списка.

Удаление: Открытая адресация сложнее, так как ячейки помечаются как удалённые, чтобы не нарушить работу поиска. Это O(1) в идеале. В закрытой удаление проще и занимает O(1 + k).

Память: Открытая адресация требует меньше памяти, так как использует только массив для хранения данных. Закрытая требует дополнительной памяти для списков, но это упрощает управление коллизиями.

Плюсы и минусы:

Открытая адресация:
Плюсы: Экономит память, простая структура.

Минусы: Плохо работает при заполненности таблицы более 70%. Удаление сложнее реализовать.

Закрытая адресация:
Плюсы: Удобнее работать с коллизиями, легко удалять элементы, производительность меньше зависит от заполненности.

Минусы: Использует больше памяти из-за списков.
Вывод: Открытая адресация подходит для таблиц с низкой заполненностью и ограниченным размером. Закрытая адресация эффективнее для больших таблиц с частыми коллизиями.

Ответы на контрольные вопросы

Чем отличается идеально сбалансированное дерево от АВЛ дерева?

Идеально сбалансированное дерево — это дерево, в котором высота левого и правого поддерева каждого узла не превышает 1, что обеспечивает минимальную высоту дерева. АВЛ дерево — это сбалансированное дерево, где разница высот поддеревьев любого узла не более 1. Идеально сбалансированное дерево всегда сбалансировано по всем уровням, в то время как АВЛ дерево поддерживает баланс только при добавлении или удалении узлов, и для этого могут понадобиться дополнительные операции перебалансировки.

Чем отличается поиск в АВЛ-дереве от поиска в дереве двоичного поиска?

Поиск в АВЛ-дереве аналогичен поиску в дереве двоичного поиска, но в АВЛ дереве дополнительно гарантируется сбалансированность. В БСТ поиск может быть неэффективным, если дерево сильно несбалансировано, что приводит к линейной структуре (в худшем случае). В АВЛ дереве, благодаря поддержанию баланса, поиск всегда имеет логарифмическую сложность, даже в худшем случае.

Что такое хеш-таблица, каков принцип ее построения?

Хеш-таблица — это структура данных, которая использует хеш-функцию для преобразования ключа в индекс массива, где хранится соответствующее значение. Принцип построения заключается в том, что для каждого ключа вычисляется хеш-значение, и этот индекс указывает на место хранения данных в таблице. Хеш-таблица позволяет выполнять операции вставки, поиска и удаления за амортизированное время O(1), если коллизии минимальны.

Что такое коллизии? Каковы методы их устранения.

Коллизии — это ситуации, когда два различных элемента хешируются в одну и ту же ячейку хеш-таблицы. Методы устранения коллизий включают открытую и закрытую адресацию. В открытой адресации элементы размещаются в следующей свободной ячейке при столкновении, а в закрытой адресации для каждого индекса создается список, и элементы с одинаковым хешем добавляются в этот список.

В каком случае поиск в хеш-таблицах становится неэффективен?

Поиск в хеш-таблицах становится неэффективным, когда таблица сильно заполнена или хеш-функция неудачна, что приводит к большому числу коллизий. Это

может значительно увеличить время поиска, так как приходится искать в длинных цепочках (в случае закрытой адресации) или использовать методы линейного поиска (в случае открытой адресации).

Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска, в хештаблицах и в файле.

Эффективность поиска: в АВЛ-деревьях поиск имеет сложность O(log n), в дереве двоичного поиска — O(h), где h — высота дерева, которая в худшем случае может быть O(n) (если дерево несбалансировано). В хеш-таблицах поиск в идеале выполняется за O(1), но может быть O(n) в случае сильных коллизий. В файле поиск обычно выполняется за O(n), так как приходится проверять все элементы последовательности.