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
«Деревья, хеш-таблицы»

Студент Нисуев Нису Феликсович

Группа ИУ7 – 32Б

Преподаватель Барышникова Марина Юрьевна

2023г.

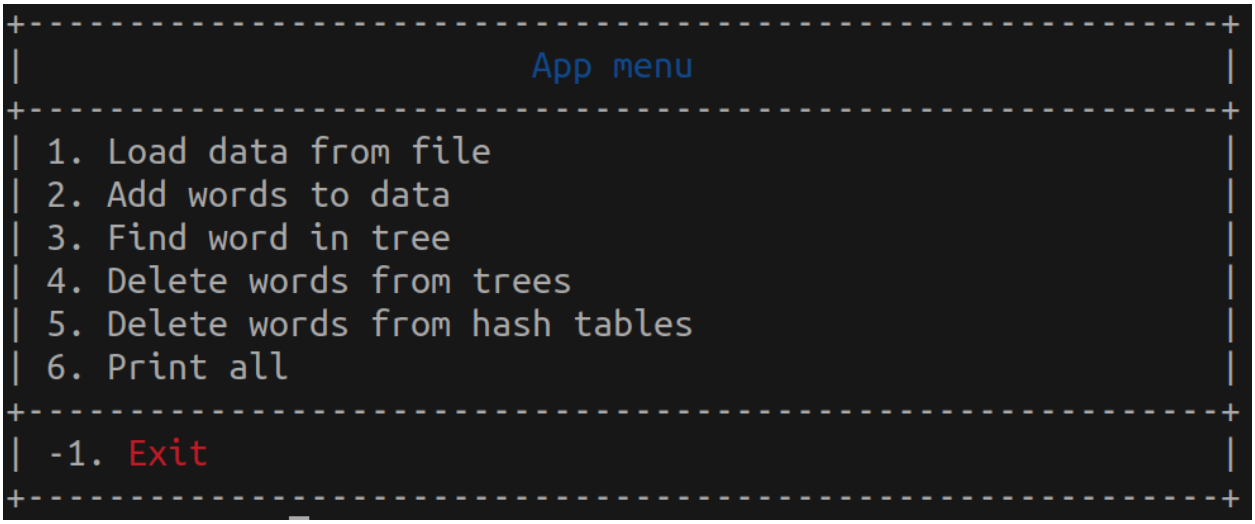
Описание условия задачи

Построить хеш-таблицу 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-таблицу. Подсчитать среднее количество сравнений для поиска данных в указанных структурах. Произвести реструктуризацию хеш-таблицы, если среднее количество сравнений 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.

Описание технического задания

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.

Входные данные:



1. **Номер команды**: целое число в диапазоне {-1} ∪ [1; 6].
2. **Дополнения к таблице:** строковое или целочисленное поле (в зависимости от команды)

Выходные данные:

1. Результат выполнения команды.
2. Сообщение об ошибке.

Обращение к программе:

Запуск через терминал (./target/app.exe | make run)

Аварийные ситуации:

1. Неверная команда
2. Неверный пользовательский ввод
3. Обращение к пустому файлу или дереву

Описание структуры данных

```
/// @brief Строка
typedef char *string;
```

```
/// @brief word_tree_t - дерево двоичного поиска слов
typedef struct leaf{
    string word;             // Слово
    size_t height;           // Высота дерева
    struct leaf *left;       // Левый потомок
    struct leaf *right;      // Правый потомок
} word_tree_t;

/*
 * __Хэш-таблицы__:
 *
 * Изначальный размер 8 - `BEG_SIZE`
 *
 * реструктуризация в хэш таблицах происходит при достижении
 * максимально разрешенного количества коллизий - `MAX_COLLIZION_CNT`
 *
 * при реструктаризации размер хэш-таблицы увеличивает размер вдвое
 * и элементы перехэшируются
 */

#define BEG_SIZE 8 // Начальный размер хеш-таблицы
#define MAX_COLLIZION_CNT 3 /// Максимально возможное кол-во коллизий

/// @brief chash_table_t - закрытая хеш-таблица
typedef struct {
    size_t size;  // Размер таблицы
    string *hash_table;  // Массив строк
    size_t elements_count;  // Кол-во элементов в хеш-таблице
    size_t (*hash_func)(string key, size_t size); // Хеш функция
} chash_table_t;

/// @brief hash_list_t - узел односвязного списка открытой хеш-таблицы
typedef struct hash_node {
    string key;                // Ключ
    size_t index;              // Индекс в односвязном списке
    struct hash_node *next;    // Указатель на следующий элемент
} hash_list_t;

/// @brief ohash_table_t - открытая хеш-таблица
typedef struct {
    size_t size;  // Размер таблицы
    size_t elements_count;  // Кол-во элементов в хеш-таблице
    hash_list_t **hash_table;  // Массив односвязных списков
    size_t (*hash_func)(string key, size_t size);  // Хеш функция
} ohash_table_t;

/// @brief Хэш-функция
static size_t hashing(string key, size_t size) {
    size_t index = 0;
```

```
    for (char *i = key; *i != '\0'; index += *i++);
    index = (index * strlen(key)) % size;
    return index;
}
```

НАБОР ТЕСТОВ

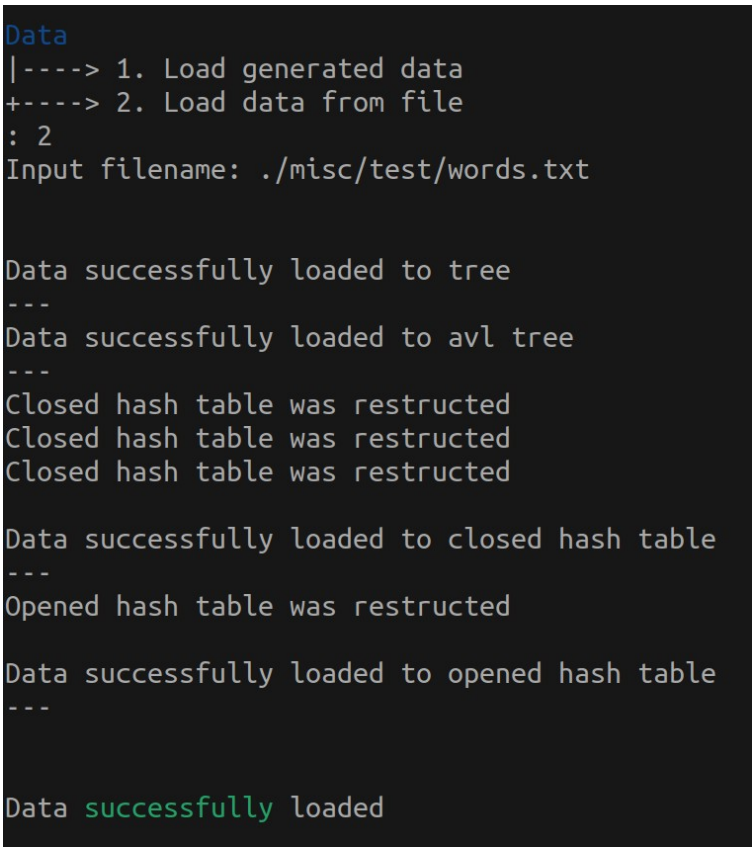
№	Название теста	Пользовательский ввод	Вывод
Негативные тесты			
1	Некорректные пункт меню	99	ERROR: Incorrect action
2	Некорректный пункт подменю 1	1 3	ERROR: Incorrect action
3	Некорректный пункт подменю 2	2 8	ERROR: Incorrect action
4	Некорректный файл	1 2 Notfile.txt	ERROR: File can’t be opened
5	Пустой файл	1 2 Empty.txt	ERROR: File is empty
6	Некорректный ввод слова на добавление	2 7 ‘\n’	ERROR: Tree is empty
7	Не введено слово для поиска	2 ‘\n’	ERROR: Incorrect input
8	Число вместо буквы	4 1	ERROR: Incorrect input

9	Строка вместо буквы	4 vnwpv	ERROR: Incorrect input
10	Генерация недопустимого числа строк	1 2 200000	ERROR: Incorrect input
11	Генерация отрицательного Количества строк	1 2 -1	ERROR: Incorrect input
Позитивные тесты			
1	Загрузка из сгенерированного файла	1 2 10	{Дерево из 10 случайных слов} Data successfully loaded
2	Считывание дерева из нормального файла	1 2 Realfile.txt	{Дерево из слов в файле} Data successfully loaded
3	Поиск слова в дереве (Слово в дереве есть)	3 {word}	Word “{word}” is founded
4	Поиск слова в дереве (Слова в дереве нет)	3 {word}	Word “{word}” is not founded
5	Удаление слов из дерева (Есть слова начинающиеся на введенную букву)	4 n	Successfully deleted <n> words beginning on “n”
6	Удаление слов из дерева (нет слов начинающихся на введенную букву)	4 n	Words beginning on “n” not founded

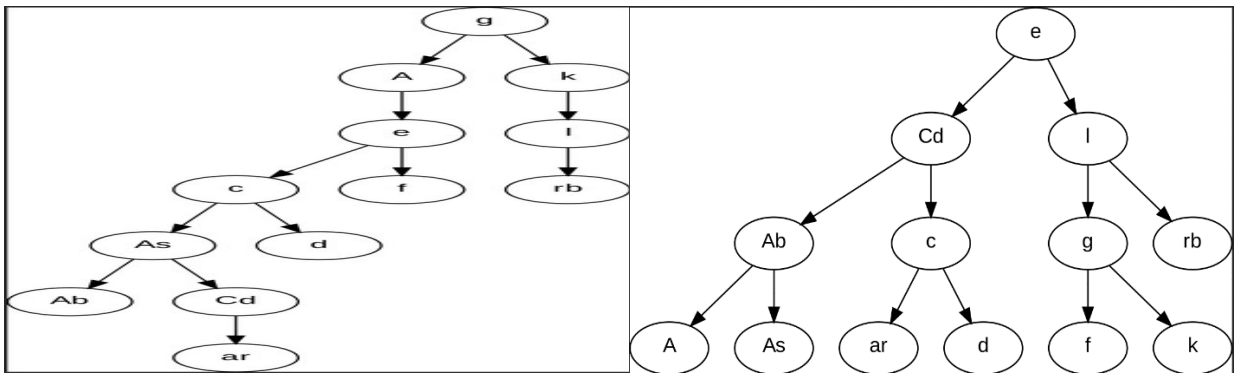
7	Вывод структур данных	6	{Вывод непустых структур}
8	Удаление слова из хэш-таблицы	5 {word}	Word “{word}” successfully deleted

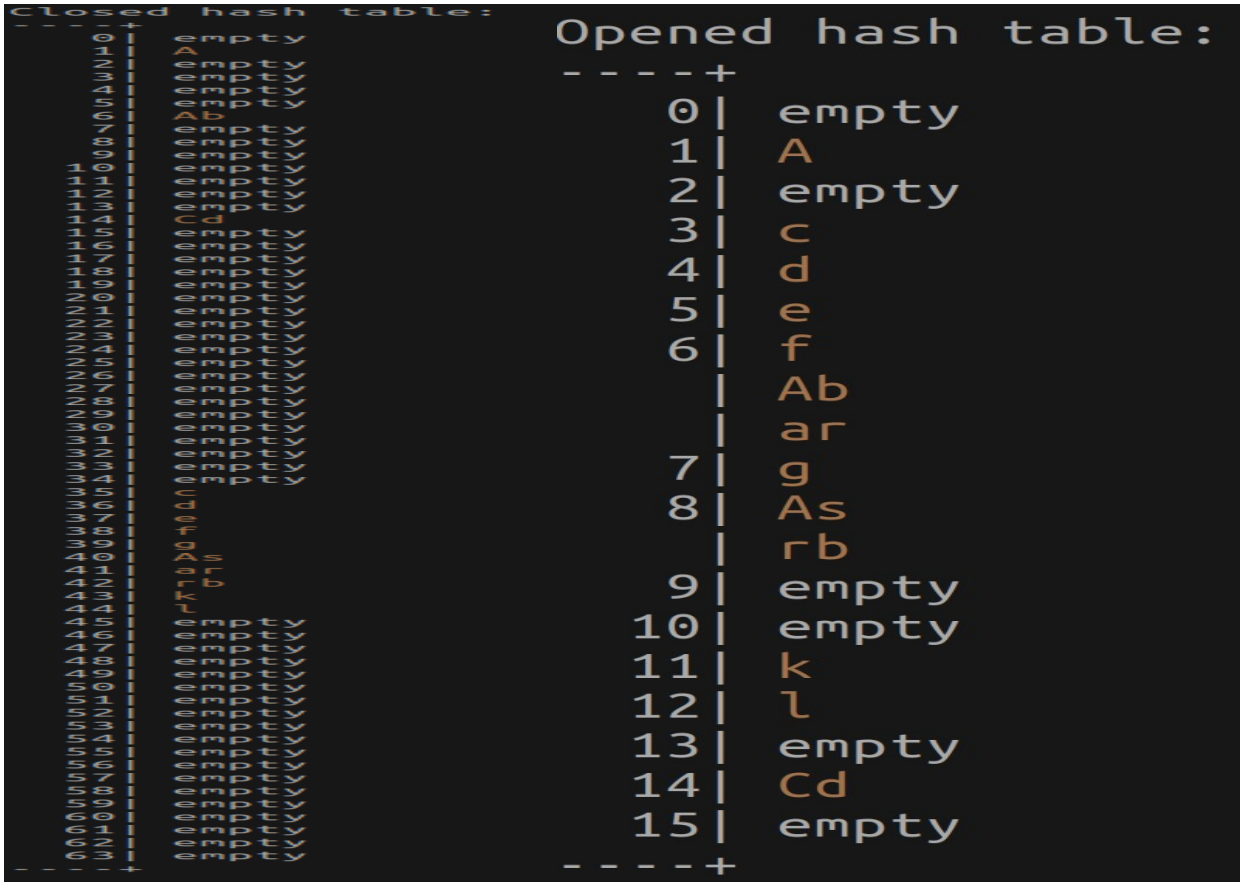
Примеры работы

1. Запись данных из файла



Бинарное дерево Сбалансированное дерево

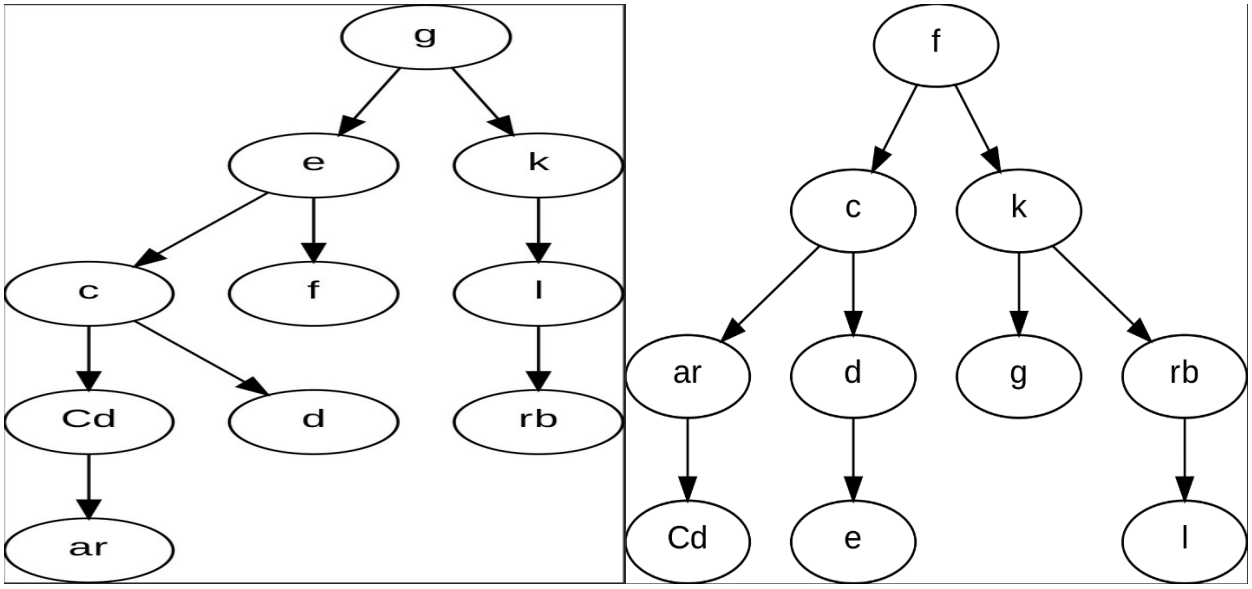




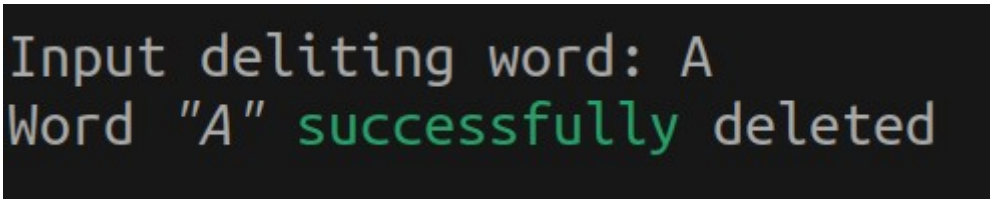
2. Удаление слов из бинарных деревьев

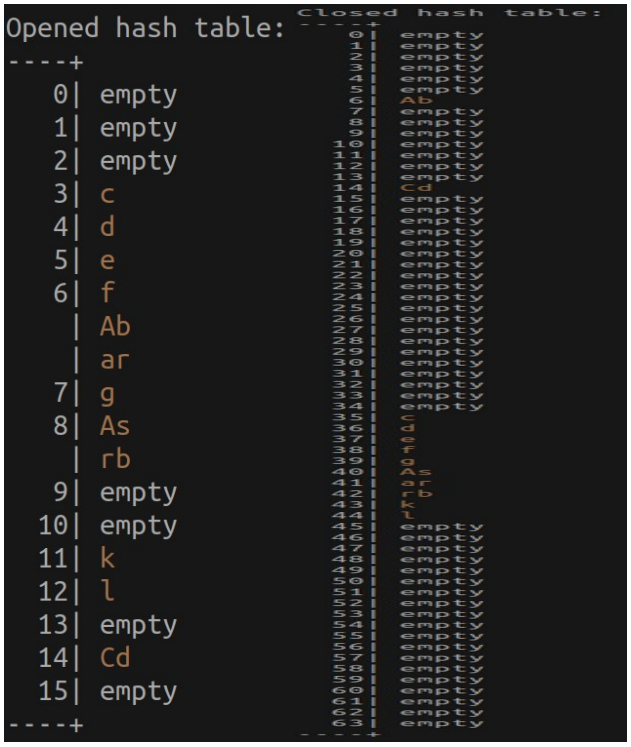
```
Input first letter of deleting words: A
Succesfully deleted 3 words beginnig on "A"
```

Бинарное дерево Сбалансированное дерево



3. Удаление слов из хэш-таблиц





Оценка эффективности

Замеры производятся 1000 раз. На 100 случайных словах (Big data), На 100 словах в которых нет удаляемых (No words for delete), На 100 словах при которых дерево становится несбалансированным (Linear).

Big data					
	Bin Tree	Avl Tree	Hash Table (O)	Hash Table (C)	File
Time, ticks	1237	2221	0	0	274572
Memory, b	3200	3200	2400	800	800
Cmp count	2	8	1	1	8
Linear					
	Bin Tree	Avl Tree	Hash Table (O)	Hash Table (C)	File
Time, ticks	11813	1822	0	0	247950
Memory, b	3200	3200	2400	800	800
Cmp count	99	6	1	1	99
No words for delete					
	Bin Tree	Avl Tree	Hash Table (O)	Hash Table (C)	File
Time, ticks	991	1671	0	0	260832
Memory, b	3200	3200	2400	800	800
Cmp count	5	8	1	4	100

Контрольные вопросы

1.Чем отличается идеально-сбалансированное дерево от АВЛ дерева?

Узлы при добавлении в идеально сбалансированное дерево располагаются равномерно слева и справа. Получается дерево, у которого число вершин в левом и правом поддеревьях отличается не более, чем на единицу. В то время как АВЛ-дерево – сбалансированное двоичное дерево, у каждого узла которого высота двух поддеревьев отличается не более чем на единицу.

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

Временная сложность поиска элемента в АВЛ дереве – O(log2n)

Временная сложность поиска элемента в дереве двоичного поиска –от O(log2n) до O(n).

3.Что такое хеш-таблица, каков принцип ее построения?

Массив, заполненный в порядке, определенном хеш-функцией, называется хеш-таблицей. Функция, по которой можно вычислить этот индекс, называется хеш-функцией. Принято считать, что хорошей является такая функция, которая удовлетворяет следующим условиям:

- функция должна быть простой с вычислительной точки зрения;
- функция должна распределять ключи в хеш-таблице наиболее равномерно.
- функция должна минимизировать число коллизий

4.Что такое коллизии? Каковы методы их устранения.

Коллизия - ситуация, когда разным ключам соответствует одно значение хеш-функции, то есть, когда h(K1) = h(K2), в то время как K1 ≠ K2. Существует два метода разрешения этой проблемы.

Первый метод – внешнее(открытое) хеширование (метод цепочек). В случае, когда элемент таблицы с индексом, который вернула хеш-функция, уже занят, к нему присоединяется связный список. Таким образом, если для нескольких различных значений ключа возвращается одинаковое значение хеш-функции, то по этому адресу находится указатель на связанный список, который содержит все значения.

Второй метод - внутреннее (закрытое) хеширование (открытая адресация). Оно состоит в том, чтобы полностью отказаться от ссылок. В этом случае, если ячейка с вычисленным индексом занята, то можно просто просматривать следующие записи таблицы по порядку (с шагом 1), до тех

пор, пока не будет найден ключ K или пустая позиция в таблице.

5.В каком случае поиск в хеш-таблицах становится неэффективен?

Поиск в хеш-таблицах становится менее эффективен, если наблюдается большое число коллизий. Тогда вместо ожидаемой сложности O(1) получим сложность O(n).

В случае открытого хеширования (цепочки) поиск в списке осуществляется простым перебором, так как при грамотном выборе хеш-функции любой из списков оказывается достаточно коротким. Если же хеширование закрытое, необходимо просматривать все ячейки, если есть много коллизий.

6.Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска и в хеш-таблицах

Хеш-таблица - от O(1) до O(n)

АВЛ-дерево - O(log2n)

Дерево двоичного поиска – от O(log2n) до O(n).

Вывод

Основным преимуществом деревьев является возможная высокая эффективность реализации основных на ней алгоритмов поиска и сортировки. При удалении или добавлении элемента необходимо корректировать балансировку, тем самым это занимает время.

Хеш-таблицы используют меньше памяти, и для них требуется меньшее количество операций сравнения при добавлении и поиске элементов. Так же таблицы требуют качественной хеш-функции, чтобы избежать большого количества коллизий.

Из переведенной выше оценки эффективности можно сделать вывод, что лучше всего и по памяти, и по времени работает хеш-таблица. Это

объясняется тем, что для того, чтобы в сбалансированное бинарное дерево добавить элемент, необходимо так же сделать балансировку, что занимает время. Так же, когда мы храним данные в таблице, мы не используем указатели, как в случае с деревьями, поэтому память у хеш-таблицы меньше.

Так же можно заметить, что сбалансированное дерево не всегда выигрывает у несбалансированного. Проигрывает во времени, так как порядок вершин всегда меняется, но выигрывает в сравнении (по среднему количеству сравнений добавления), так как высота сбалансированного дерева будет меньше или такой же, как и у несбалансированного, поэтому чтобы узнать месторасположение элемента, приходится меньше ходить по дереву.