|  |  |
| --- | --- |
|  | **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**

Студент **Паншин Сергей Константинович**

Группа **ИУ7-33Б**

Название предприятия **НУК ИУ МГТУ им. Н. Э. Баумана**

|  |  |
| --- | --- |
| Студент | **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Паншин С. К.** |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Никульшина Т. А.** |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **Барышникова М. Ю.** |

Оценка \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*2023 г.*

**Условие задачи**

Построить и обработать хеш-таблицы, сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска, в хеш-таблицах и в файлах. Сравнить эффективность реструктуризации таблицы для устранения коллизий и поиска в ней с эффективностью поиска в исходной таблице.

**Техническое задание**

Сбалансировать дерево (задача №6) после удаления повторяющихся букв. Вывести его на экран в виде дерева. Составить хеш-таблицу, содержащую буквы и количество их вхождений во введенной строке. Вывести таблицу на экран. Осуществить поиск введенной буквы в двоичном дереве поиска, в сбалансированном дереве и в хеш-таблице. Сравнить время поиска, объем памяти и количество сравнений при использовании различных структур данных.

**Входные данные:**

Пункт меню (число от 0 до 8):

Menu:

1) Add in hash table (opened)

2) Print hash table (opened)

3) Restruct hash table (opened)

4) Add in hash table (closed)

5) Print hash table (closed)

6) Restruct hash table (closed)

7) Process string

8) Сomparison of search types

0) Quit program

Также обрабатываемая строка и символ

**Выходные данные:**

Картинка изначального дерева, конечного, сбалансированного, авл дерева, а также конечного дерева в постфиксном обходе и хеш таблица.

**Возможные аварийные ситуации:**

Некорректный ввод: пункта меню, пустая строка, в строке не буква/цифра.

**Способ обращения к программе**

В папке с программой запустить команду *make run*.

**Структуры данных**

// узел дерева

typedef struct tree\_node

{

char value;

size\_t count;

struct tree\_node \*parent;

struct tree\_node \*left;

struct tree\_node \*right;

int height;

} tree\_node\_t;

// данные хеш-таблицы

struct data

{

char value;

size\_t count;

data\_t \*next;

};

// хэш-таблица

typedef struct hash\_table

{

hash\_table\_type type;

data\_t \*data[MAX\_ELEMENT\_COUNT];

size\_t size;

int max\_ind;

size\_t collision\_count;

int hash\_key;

} hash\_table\_t;

// Создания узла дерева

// Вход: указатель на указатель по которому хранится узел, значение

// Выход: код возврата

int node\_init(tree\_node\_t \*\*node, char value);

// Освобождение памяти из-под узла дерева

// Вход: указатель, по которому хранится узел

void node\_free(tree\_node\_t \*node);

// Освобождение памяти из-под дерева

// Вход: указатель, по которому хранится узел

void tree\_free(tree\_node\_t \*\*tree);

// Создания дерева из элементов строки

// Вход: указатель на строку, размер строки

// Выход: указатель на корень дерева

tree\_node\_t \*tree\_create\_from\_str(char \*str, size\_t len);

// Добавление узла в дерево

// Вход: указатель на указатель по которому хранится корень, значение

// Выход: код возврата

int tree\_add\_el(tree\_node\_t \*\*tree, char value);

// Вывод дерева при постфиксном обходе

// Вход: указатель на корень

void tree\_print\_post\_order(tree\_node\_t \*tree);

// Удаление не уникальных узлов

// Вход: указатель на указатель по которому хранится корень

void tree\_del\_not\_unique\_nodes(tree\_node\_t \*\*tree);

// Применение функции к дереву при префиксном обходе

// Вход: указатель на указатель на корень, указатель на функцию, указатель на аргументы

void tree\_apply\_pre(tree\_node\_t \*tree, void (\*f)(tree\_node\_t \*, void \*), void \*arg);

// Экспорт дерева в формат dot для отрисовки в виде картинки

// Вход: указатель на файл, имя для дерева, указатель на корень,

void tree\_export\_to\_dot(FILE \*f, const char \*tree\_name, tree\_node\_t \*tree);

// Перевод дерева в ИСД

// Вход: указатель по которому хранится корень

// Выход: ссылка на новый корень

tree\_node\_t \*tree\_balance\_BST(tree\_node\_t \*tree);

// Перевод дерева в АВЛ дерево

// Вход: указатель, по которому хранится корень

// Выход: ссылка на новый корень

tree\_node\_t \*AVL\_tree\_from\_tree(tree\_node\_t \*tree);

// Инициализация хэш-таблицы

// Вход: тип открытая/закрытая

// Выход: ссылка на таблицу

hash\_table\_t \*table\_init(int type);

// Очистка хэш-таблицы

// Вход: указатель на таблицу

void table\_free(hash\_table\_t \*table);

// Добавление в хэш-таблицу

// Вход: указатель на таблицу, значение

// Выход: код возврата

int table\_add(hash\_table\_t \*\*table, char value);

// Создание хэш-таблицы из строки

// Вход: указатель на таблицу, строка

// Выход: код возврата

int table\_create\_from\_str(hash\_table\_t \*\*table, char \*str)

// Печать хэш-таблицы

// Вход: указатель на таблицу

void table\_print(hash\_table\_t \*table);

// Создание хэш-таблицы из строки

// Вход: указатель на таблицу, значение, указатель на счетчик

// Выход: код возврата

int table\_search(hash\_table\_t \*table, char value, int \*cmp\_count);

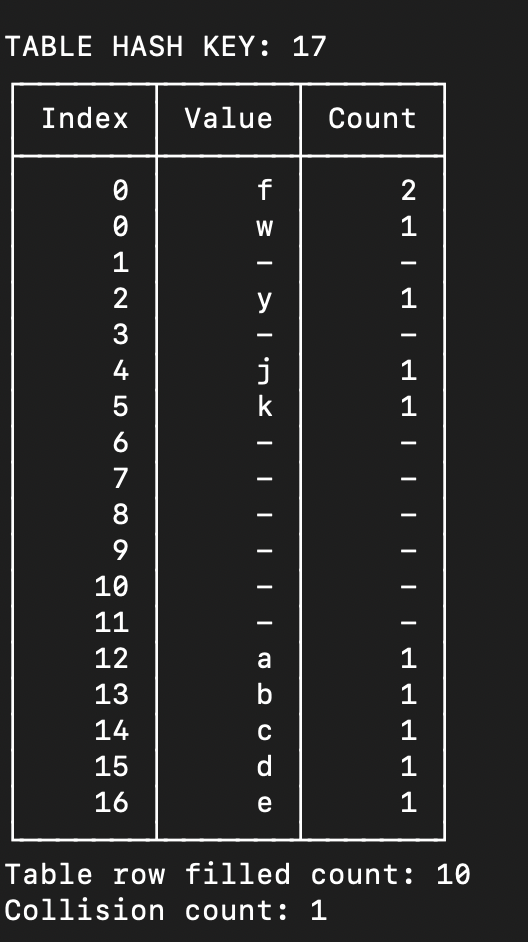
**Реализация хэш-таблицы**

При создании таблица имеет хэш-ключ 17.  
Функция хэширования — это взятие остатка от кода символа по числу (хэш-ключ).

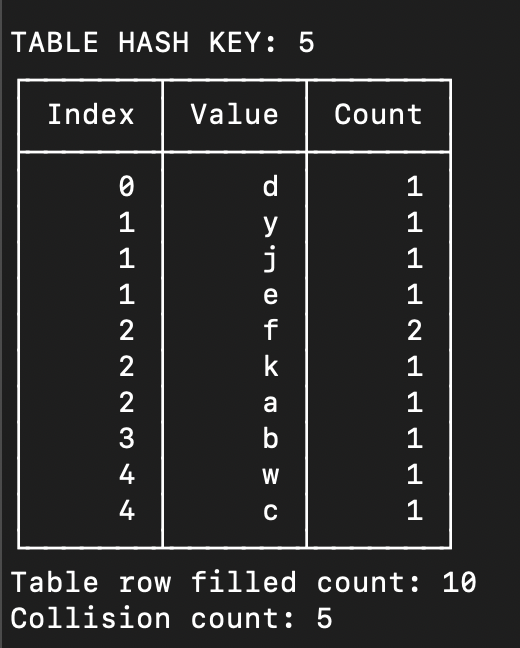
При решении коллизий мы проводим реструктуризацию таблицы меняя ей хэш-ключ на следующее простое число (в ручном режиме можно поменять на любое число, даже меньшее, например, для демонстрации коллизий).

Пример работы хэш-таблицы с открытым решением коллизий (создание списка в индексе коллизий):

Таблица в начале



Уменьшим хэш-ключ чтобы появилось большее количество коллизий, для того чтобы посмотреть, как происходит их решение.



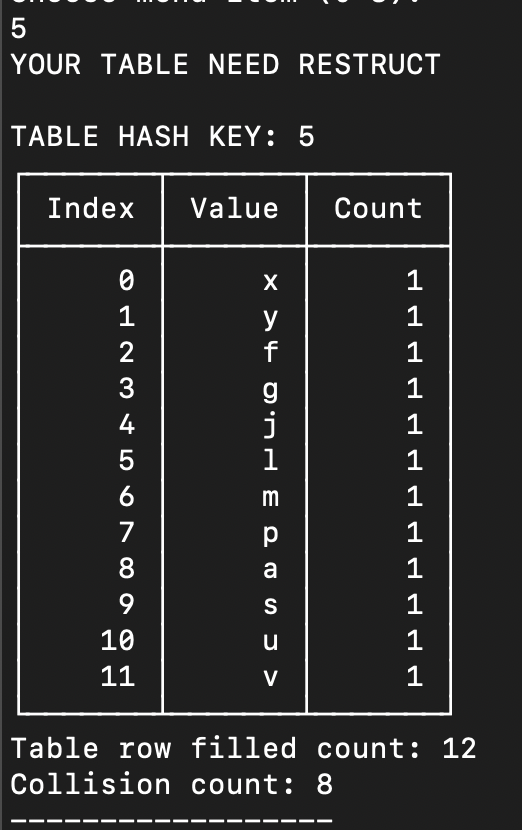
Эти коллизии можно решить, проведя реструктуризацию таблицы задав ей больший хэш-ключ (например, на изначальный равный 17).

Пример работы хэш-таблицы с закрытым решением коллизий (поиск ближайшего свободного места за тем местом, которое нам вернула хэш-функция):

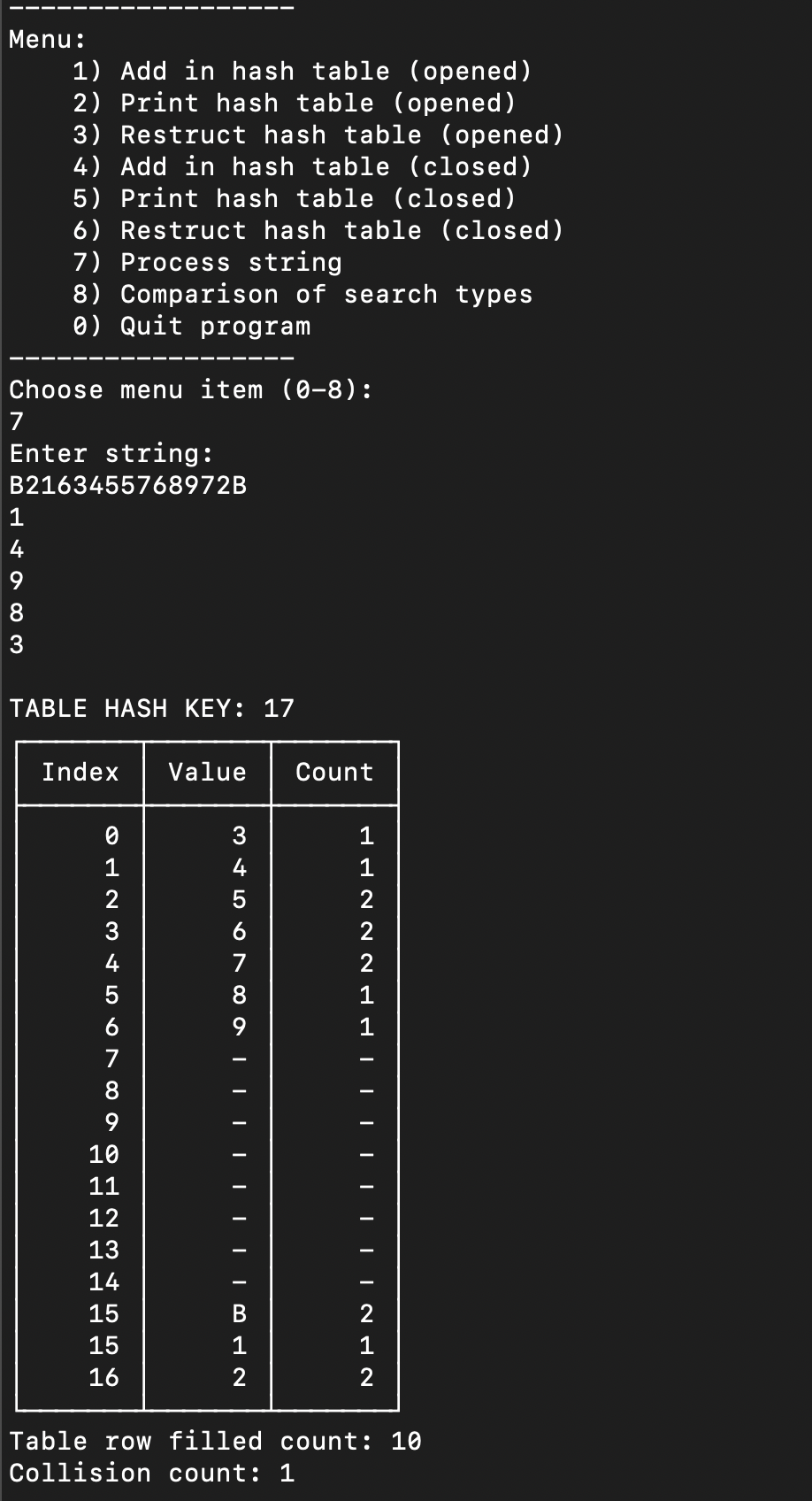
Таблица в начале

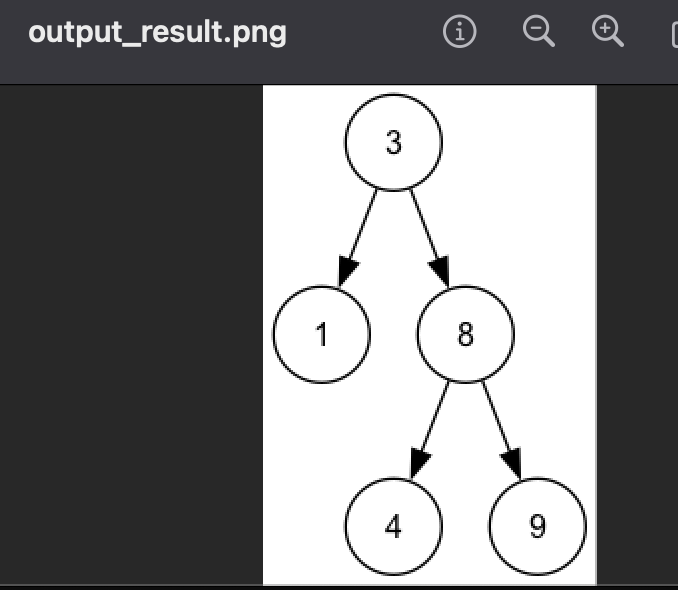
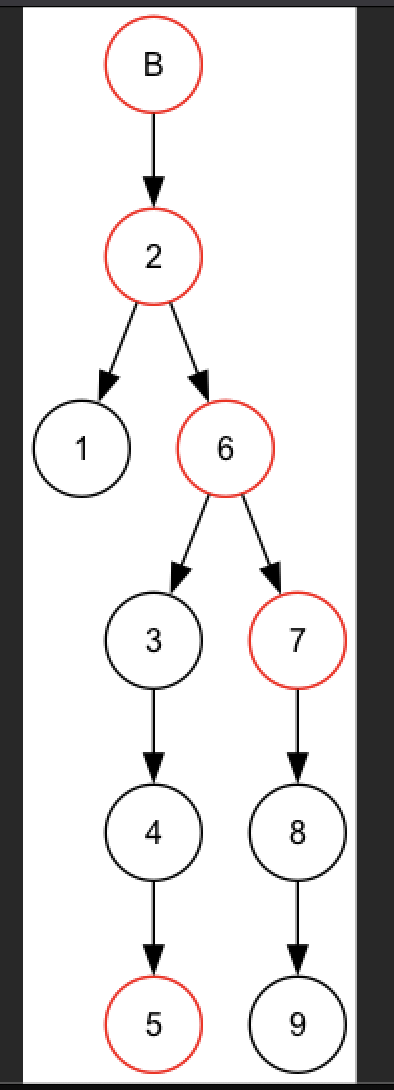


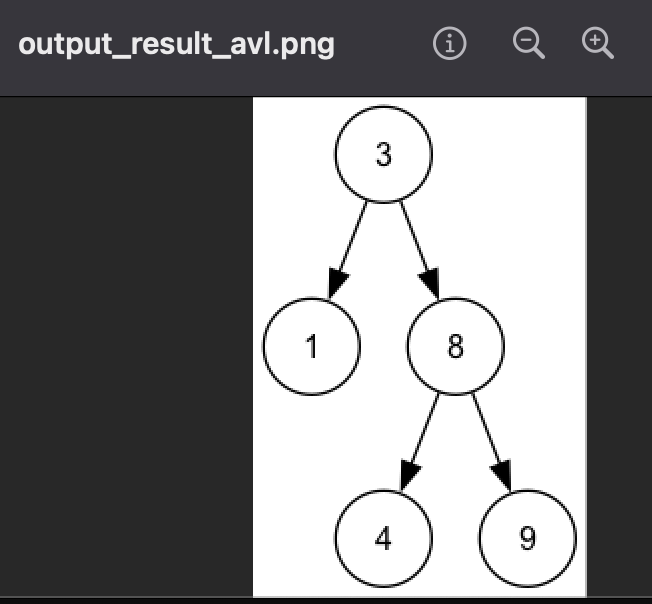
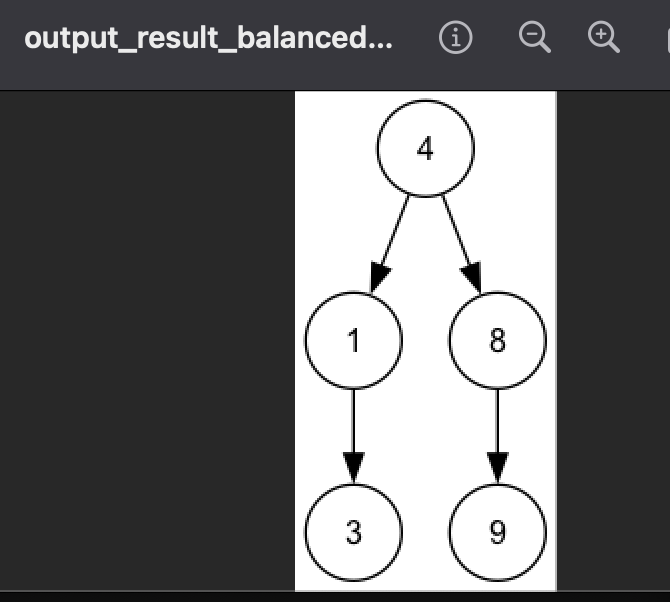
Уменьшим хэш-ключ чтобы появилось большее количество коллизий, для того чтобы посмотреть, как происходит их решение.



**Пример работы**

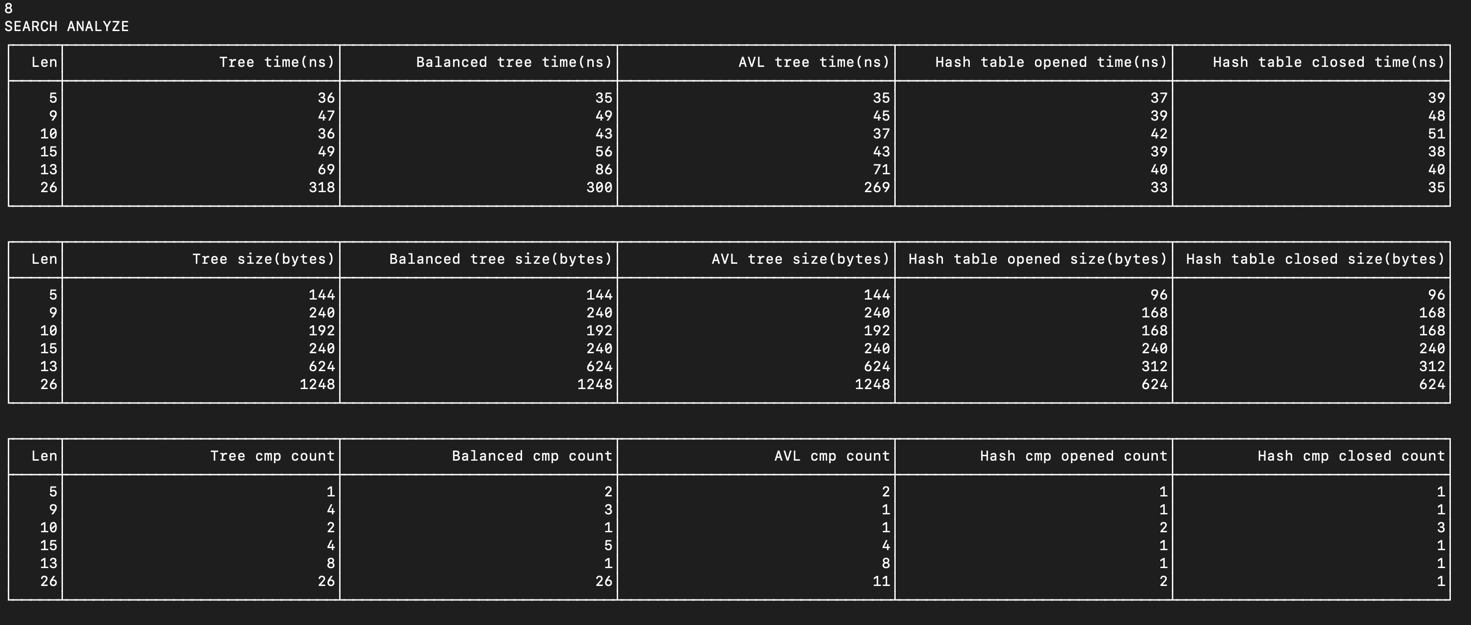
****

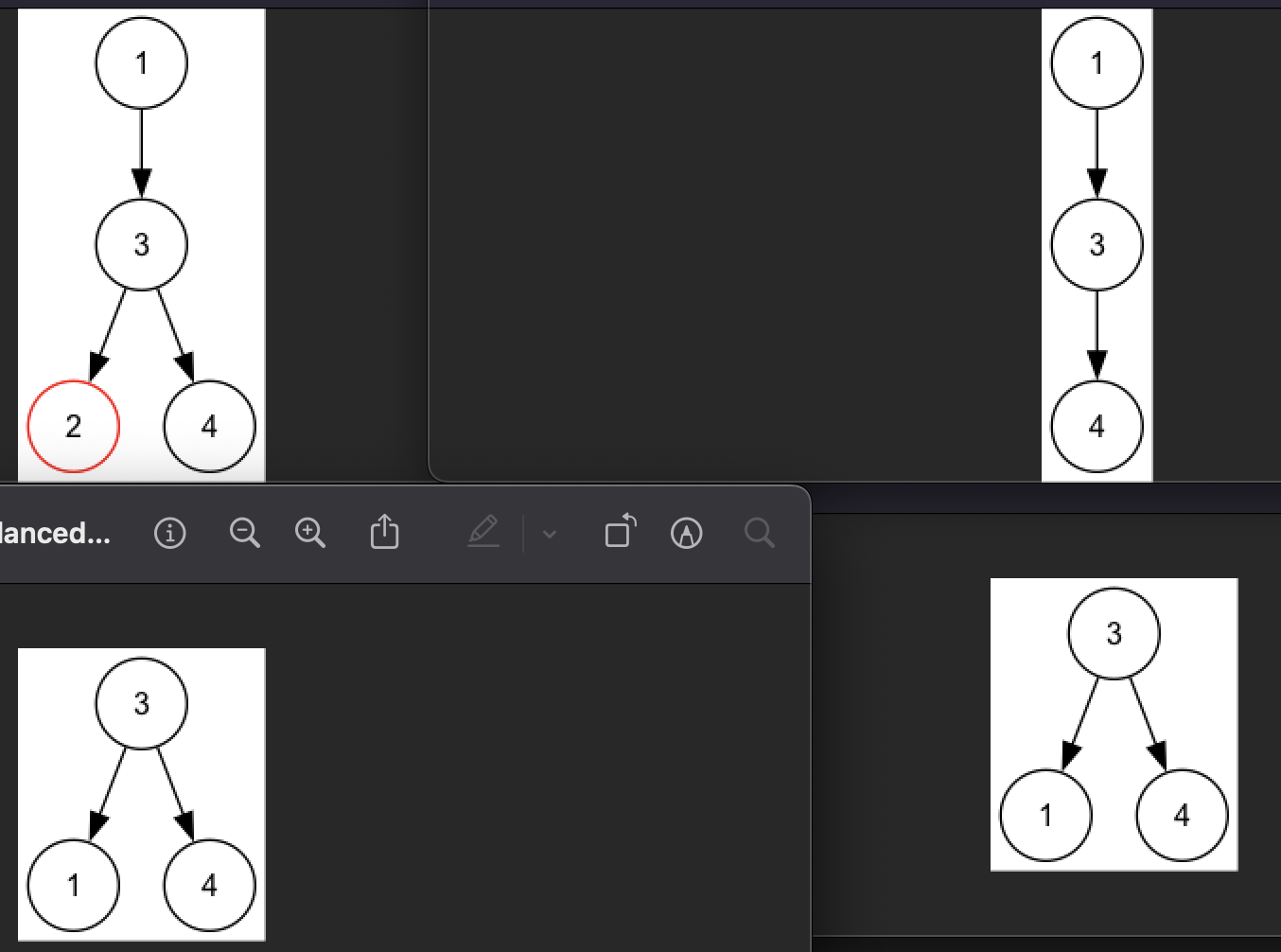
****

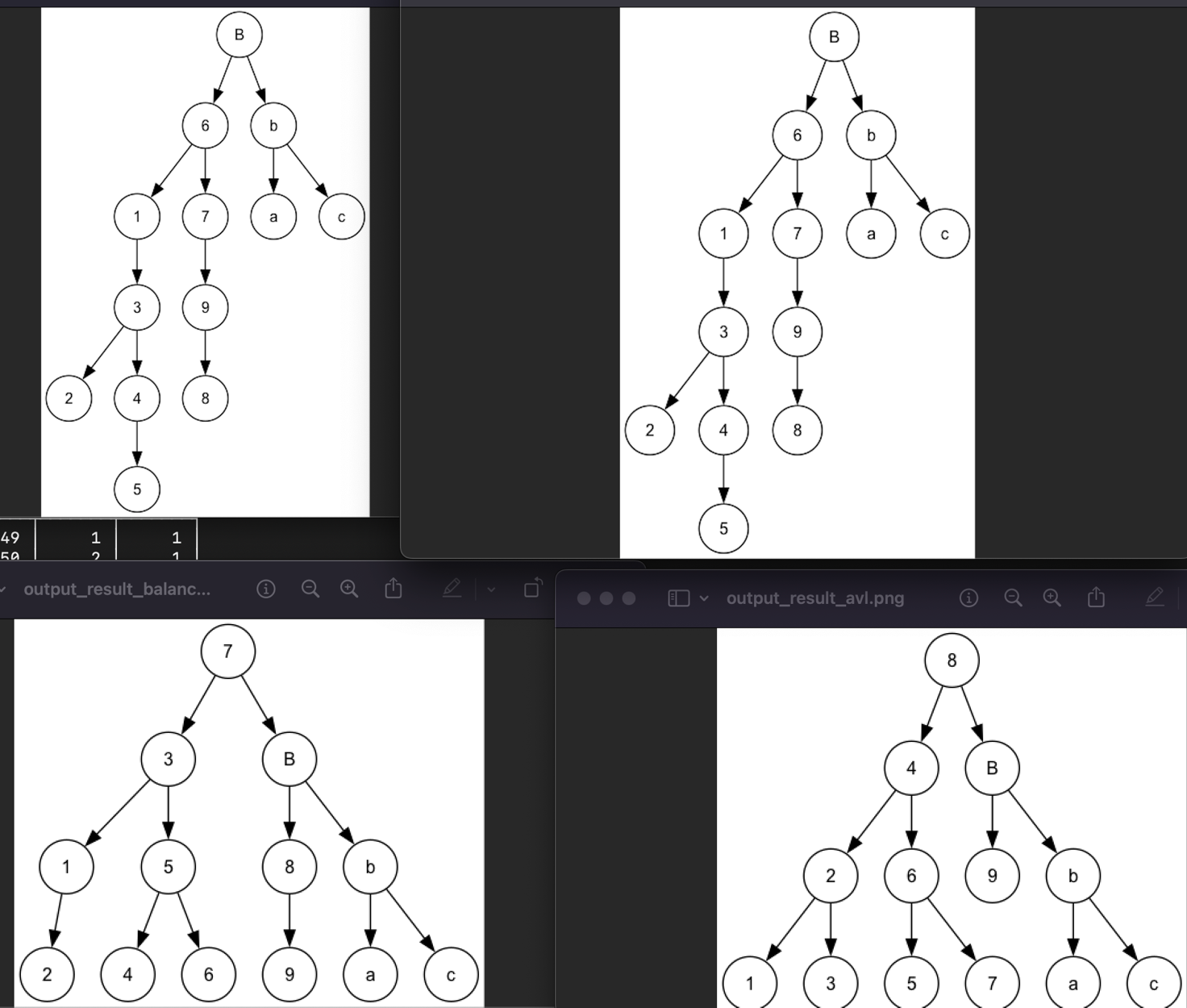
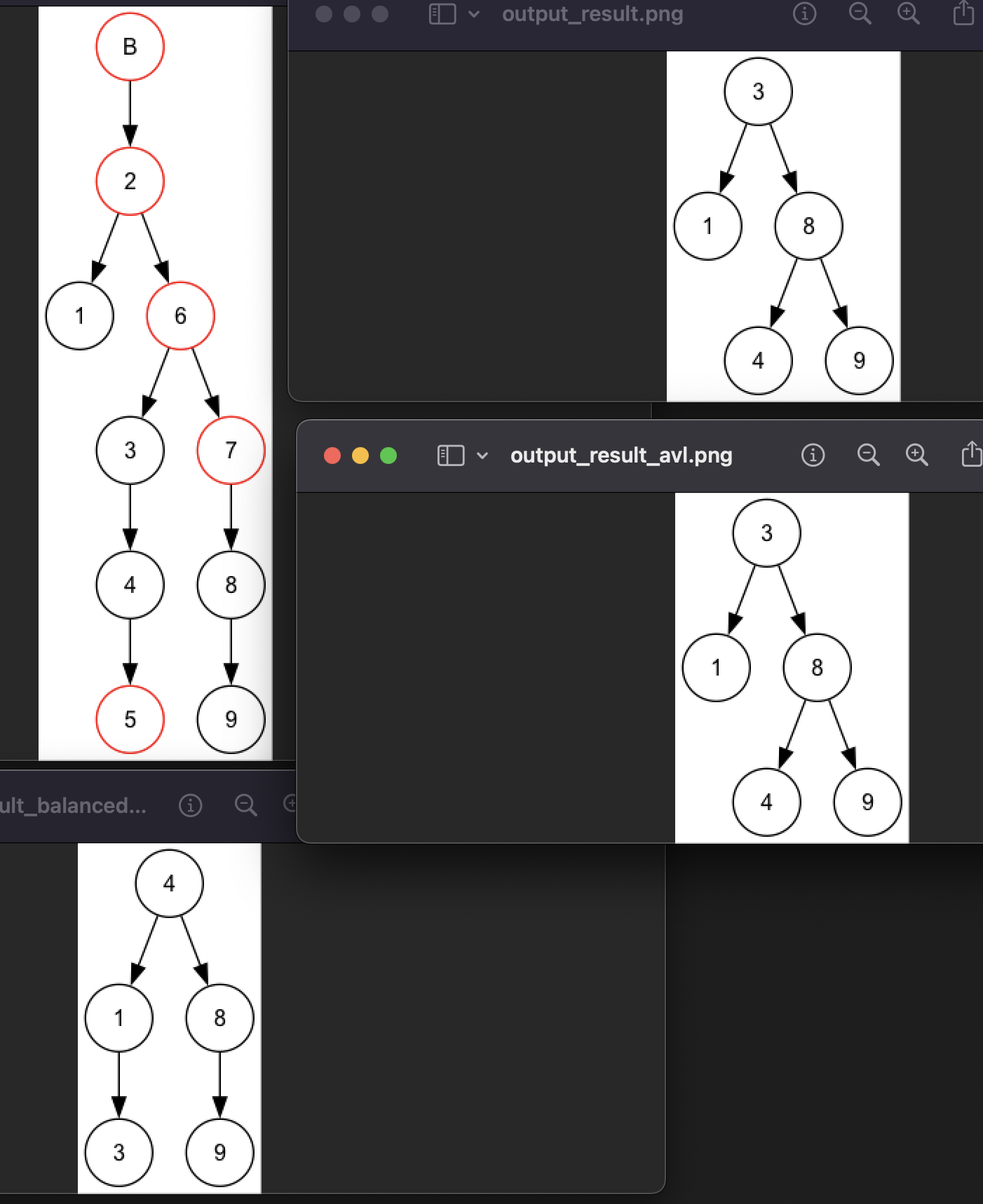
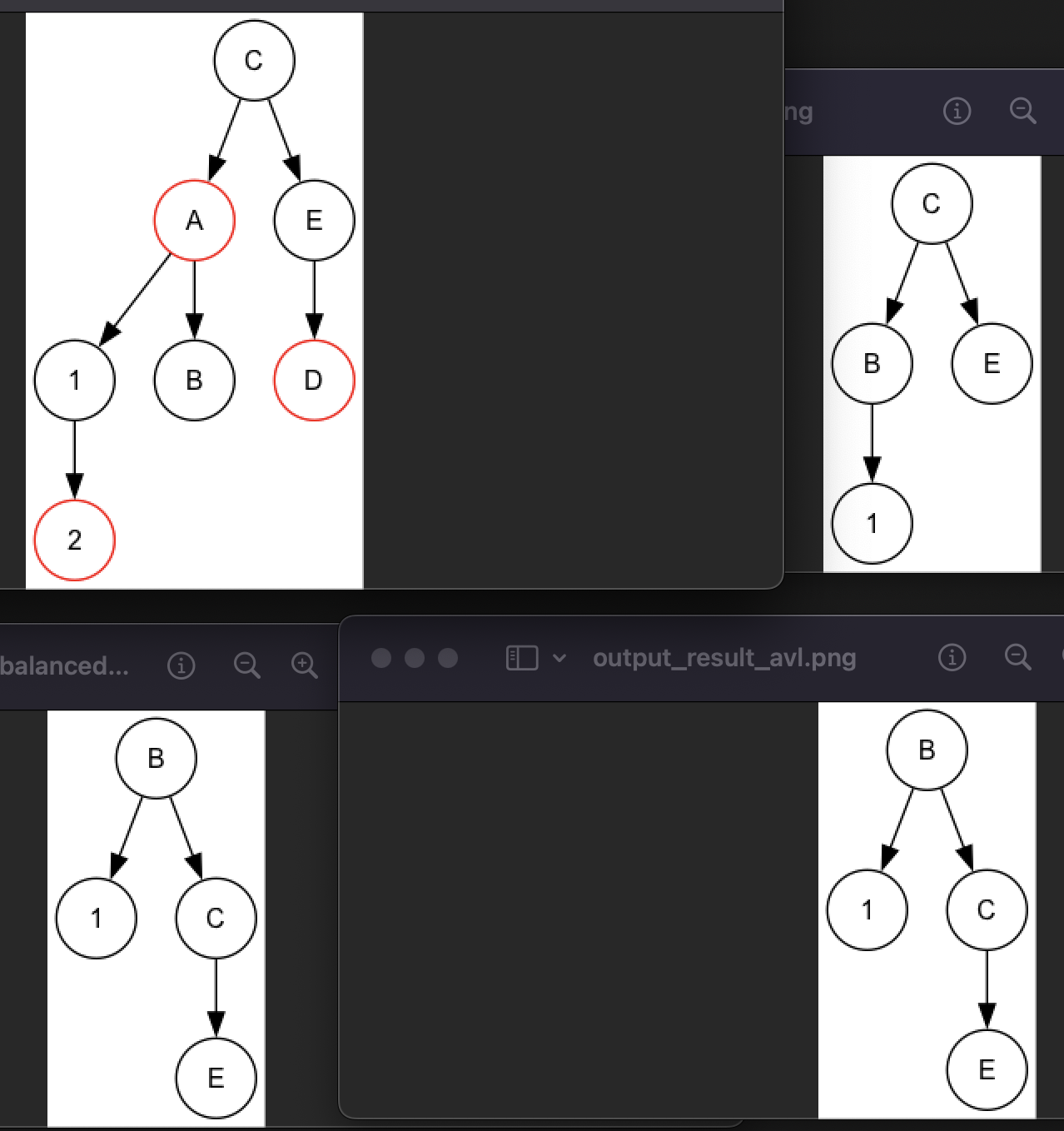
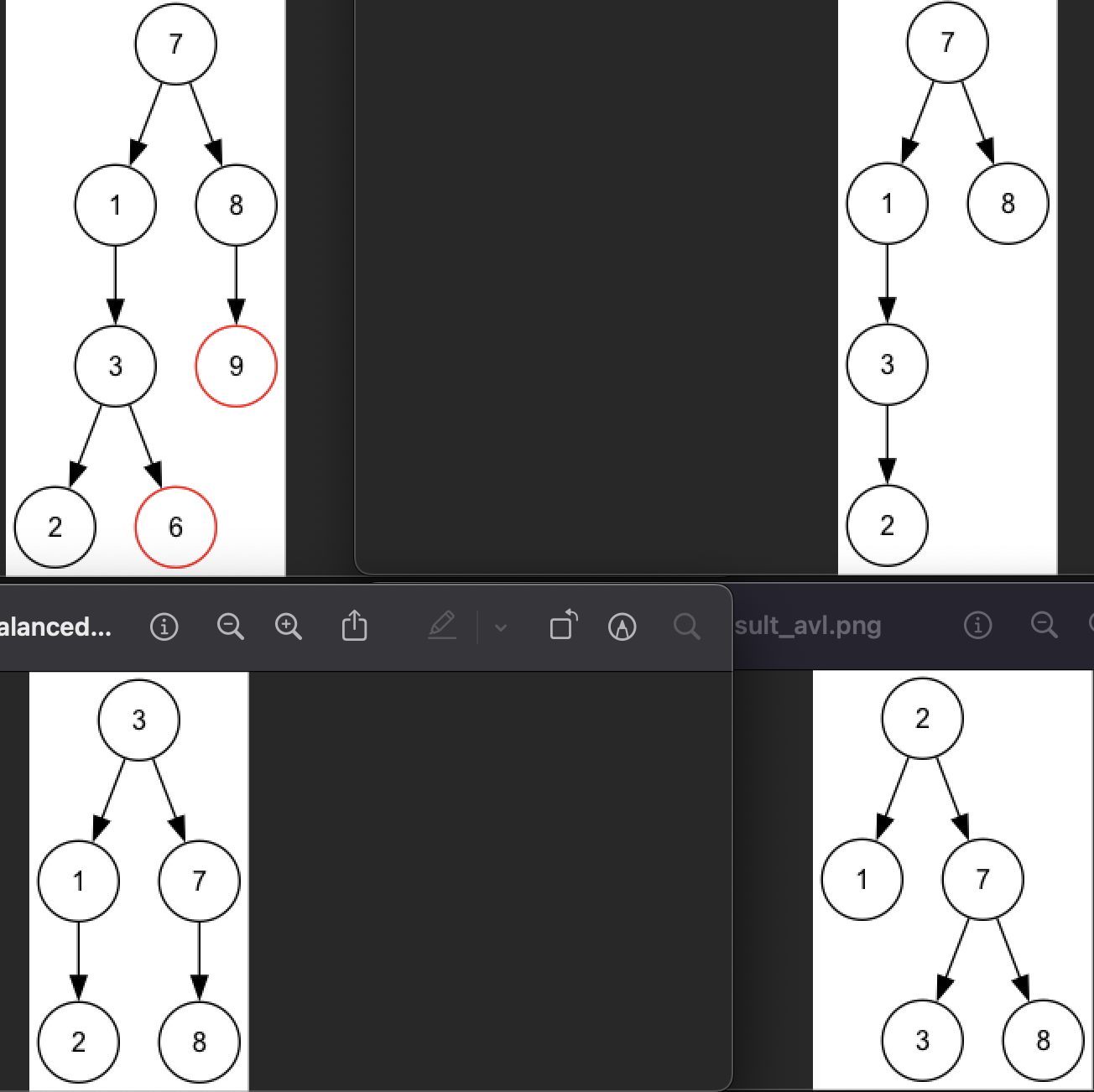
****

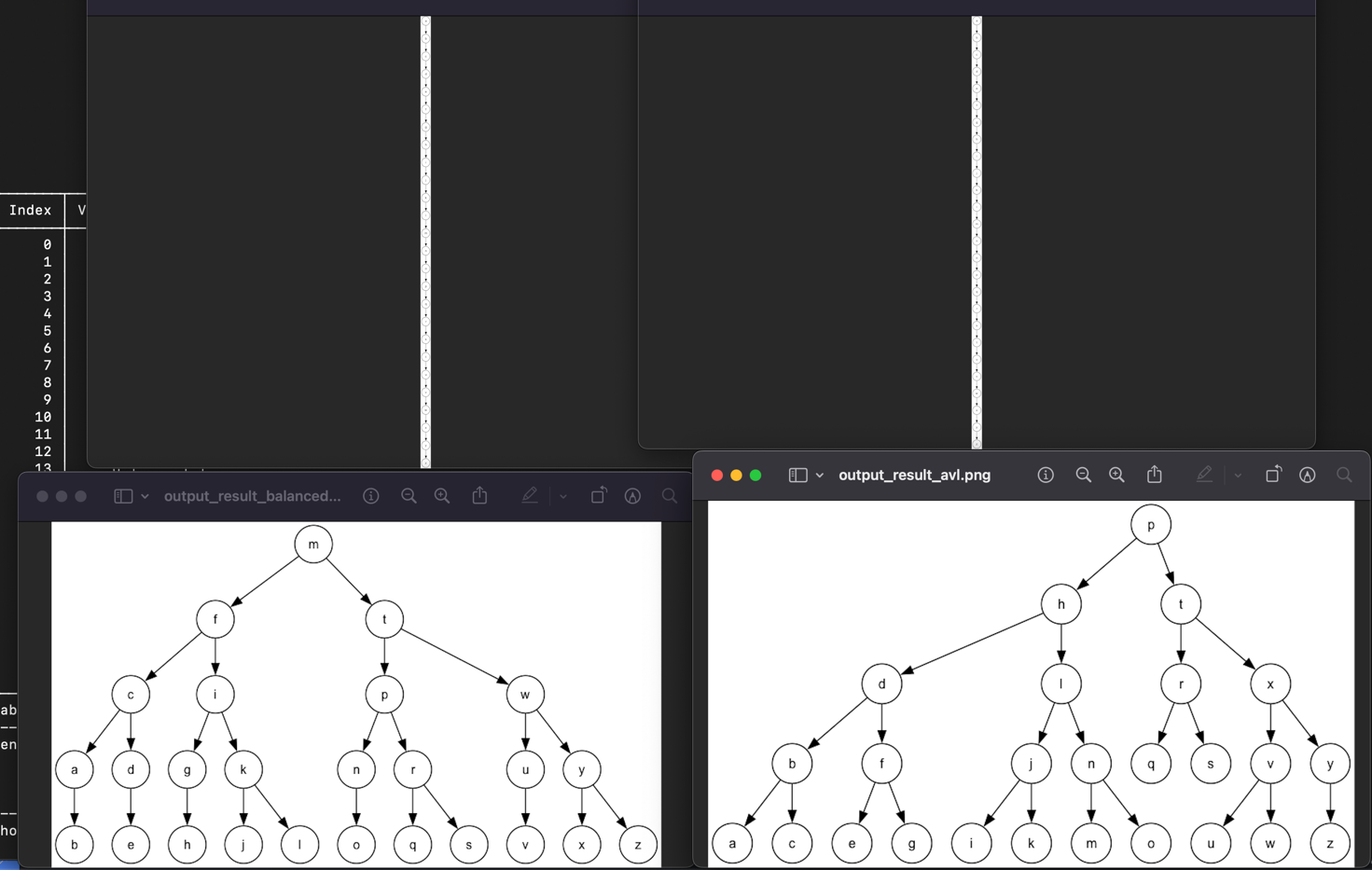
**Замеры**

Производится 2000 итераций при замерах, на основе деревьев представленных ниже









**Выводы по проделанной работе**

Среди рассматриваемых структур в большинстве случаев выигрывает хеш-таблица, так как поиск в ней происходит за О(1). Так же при большом объеме данных, для хранения хеш-таблицы нужно меньше памяти, чем для хранения деревьев. Но, у АВЛ и ДДП деревьев есть по крайней мере одно заметное преимущество по сравнению с хеш-таблицей: в них можно выполнить проход по возрастанию или убыванию ключей и сделать это быстро. Также скорость поиска в ИСД напрямую зависит от того, как подавать узлы на вход при его построении.

**Контрольные вопросы**

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

У АВЛ дерева для каждой его вершины высота двух её поддеревьев различается не более чем на 1, а у идеально сбалансированного дерева различается количество вершин в каждом поддереве не более чем на 1.

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

Поиск в АВЛ дереве происходит быстрее, чем в ДДП.

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

Хеш-таблицей называется массив, заполненный элементами в порядке, определяемом хеш-функцией. Хеш-функция каждому элементу таблицы ставит в соответствие некоторый индекс. Функция должна быть простой для вычисления, распределять ключи в таблице равномерно и давать минимум коллизий.

*4. Что такое коллизии? Каковы методы их устранения.*

Коллизия – ситуация, когда разным ключам хеш-функция ставит в соответствие один и тот же индекс. Основные методы устранения коллизий: открытое и закрытое хеширование. При открытом хешировании к ячейке по данному ключу прибавляется связанный список, при закрытом – новый элемент кладется в ближайшую свободную ячейку после данной.

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

Поиск в хеш-таблице становится неэффективен при большом числе коллизий – сложность поиска возрастает по сравнению с О(1). В этом случае требуется реструктуризация таблицы – заполнение её с использованием новой хеш-функции.

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

В хеш-таблице минимальное время поиска: О(1).

В АВЛ: О(log2n).

В дереве двоичного поиска О(h), где h - высота дерева (от log2n до n).