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) Comparison 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

TABLE HASH KEY: 17

Index	Value	Count
0	f	2
0	w	1
1	–	–
2	y	1
3	–	–
4	j	1
5	k	1
6	–	–
7	–	–
8	–	–
9	–	–
10	–	–
11	–	–
12	a	1
13	b	1
14	c	1
15	d	1
16	e	1

Table row filled count: 10
Collision count: 1

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

TABLE HASH KEY: 5

Index	Value	Count
0	d	1
1	y	1
1	j	1
1	e	1
2	f	2
2	k	1
2	a	1
3	b	1
4	w	1
4	c	1

Table row filled count: 10
Collision count: 5

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

```
TABLE HASH KEY: 17
```

Index	Value	Count
0	f	1
1	g	1
2	x	1
3	y	1
4	j	1
5	–	–
6	l	1
7	m	1
8	–	–
9	–	–
10	p	1
11	–	–
12	a	1
13	s	1
14	–	–
15	u	1
16	v	1

```
Table row filled count: 12
Collision count: 2
```

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

```
5
YOUR TABLE NEED RESTRUCT

TABLE HASH KEY: 5
```

Index	Value	Count
0	x	1
1	y	1
2	f	1
3	g	1
4	j	1
5	l	1
6	m	1
7	p	1
8	a	1
9	s	1
10	u	1
11	v	1

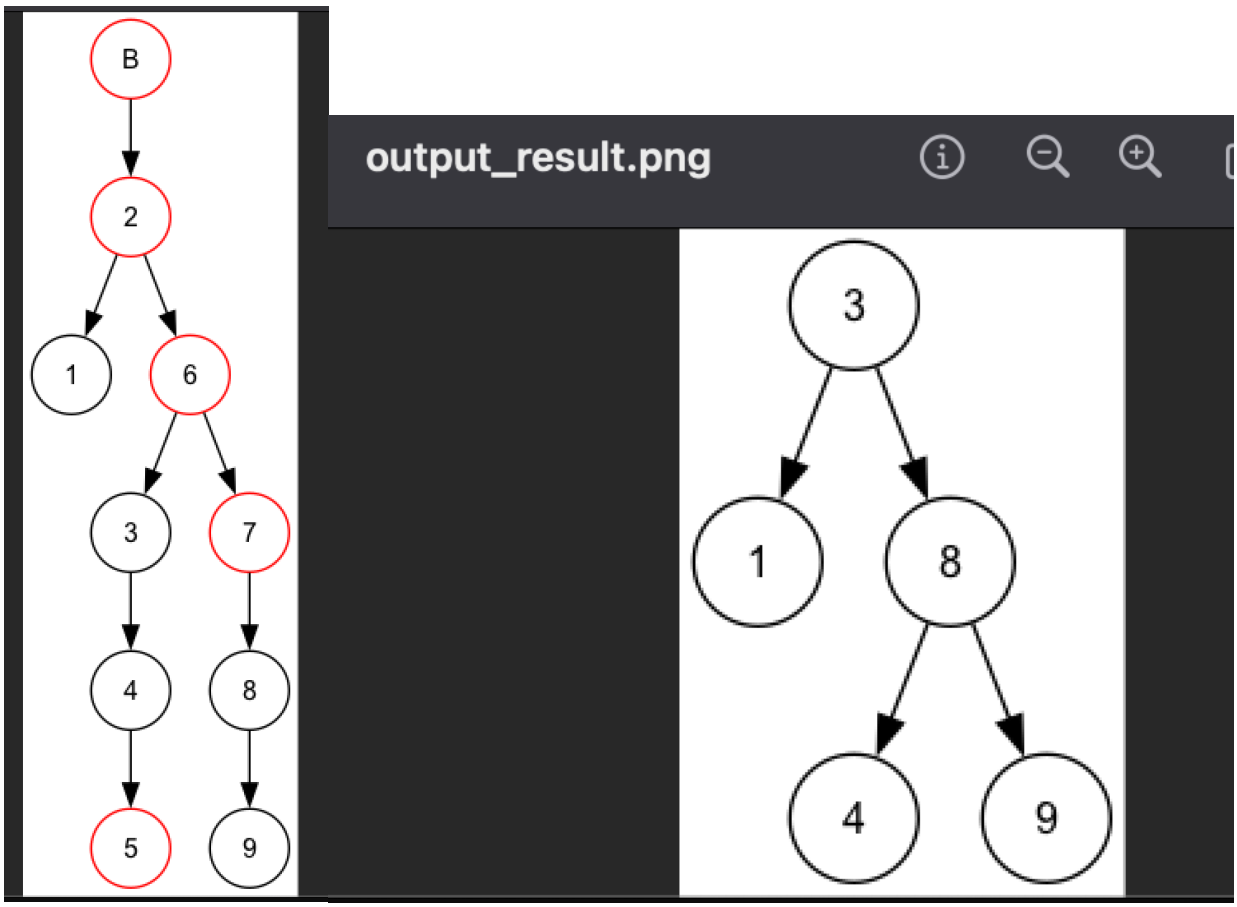
```
Table row filled count: 12
Collision count: 8
------------------
```

Пример работы

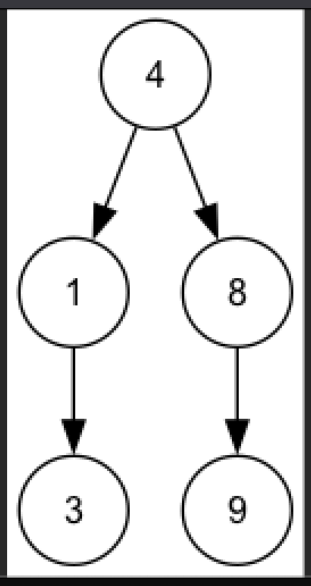
```
------------------
Menu:
    1) Add in hash table (opened)
    2) Print hash table (opened)
    3) Restruct hash table (opened)
    4) Add in hash table (closed)
    5) Print hash table (closed)
    6) Restruct hash table (closed)
    7) Process string
    8) Comparison of search types
    0) Quit program
------------------
Choose menu item (0-8):
7
Enter string:
B21634557689728
1
4
9
8
3

TABLE HASH KEY: 17
```

Index	Value	Count
0	3	1
1	4	1
2	5	2
3	6	2
4	7	2
5	8	1
6	9	1
7	-	-
8	-	-
9	-	-
10	-	-
11	-	-
12	-	-
13	-	-
14	-	-
15	B	2
15	1	1
16	2	2

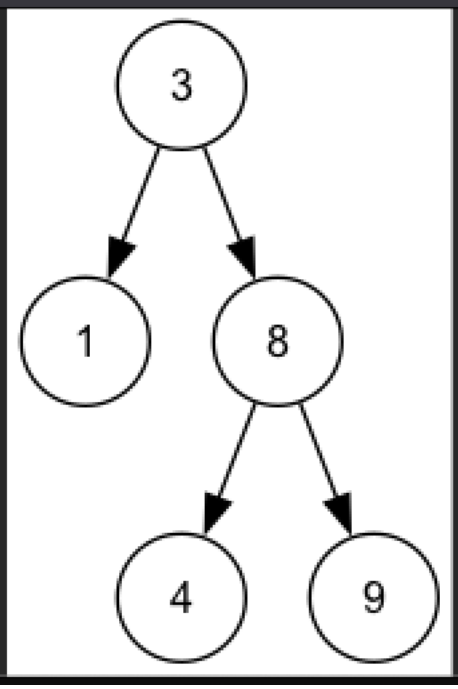
```
Table row filled count: 10
Collision count: 1
```


B
2
1
6
3
7
4
8
5
9
output_result.png
3
1
8
4
9

output_result_balanced...



output_result_avl.png



Замеры

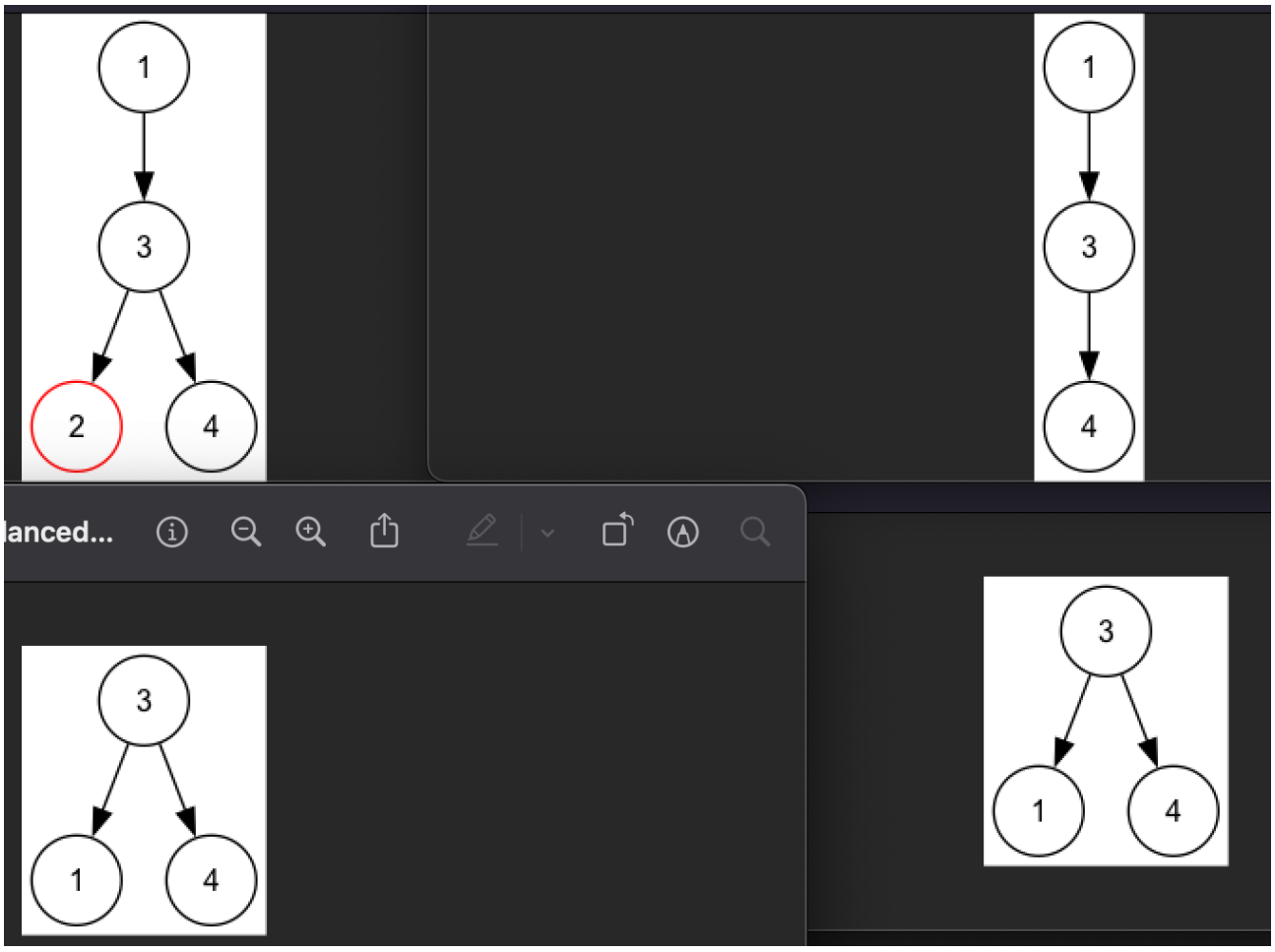
Производится 2000 итераций при замерах, на основе деревьев представленных ниже

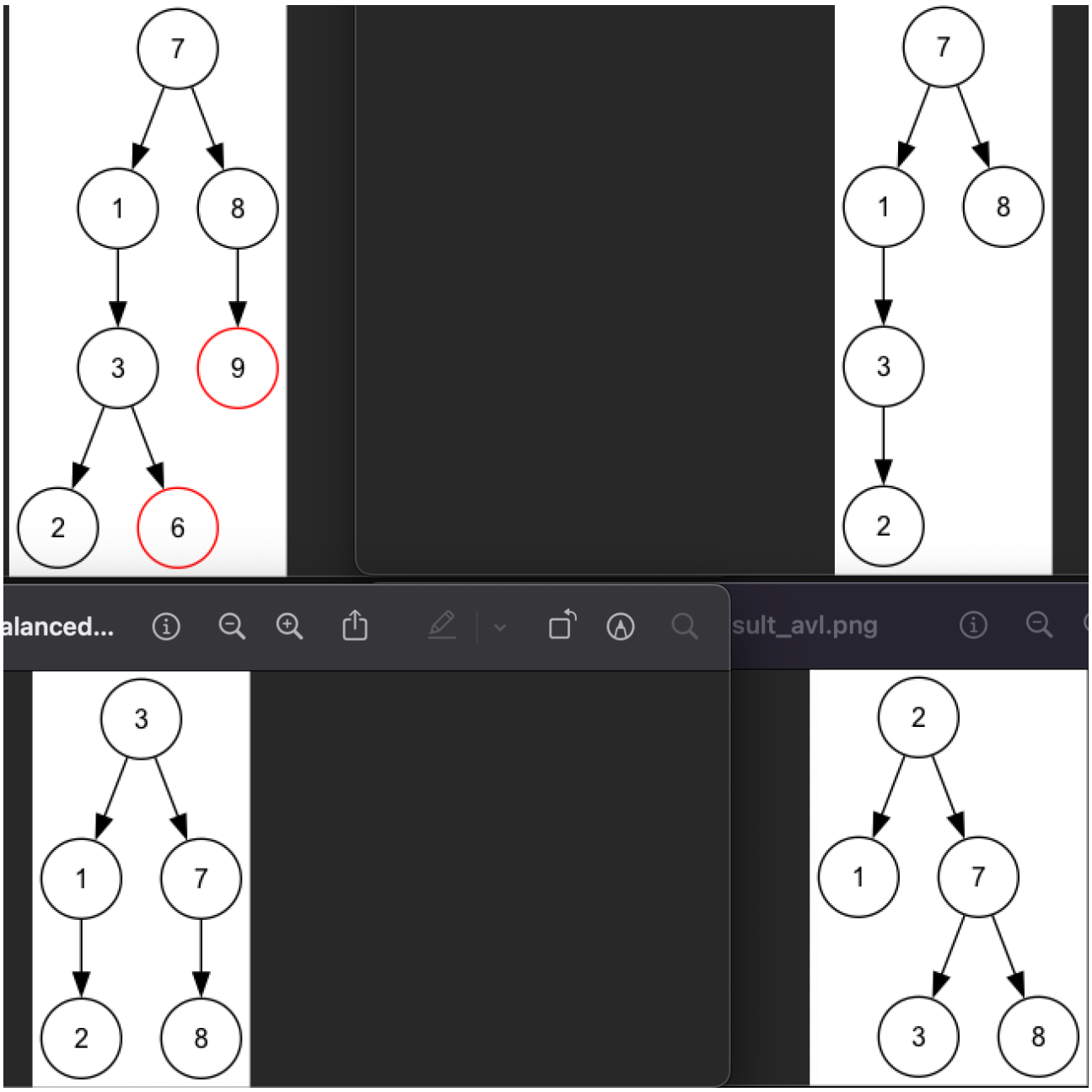
8
SEARCH ANALYZE

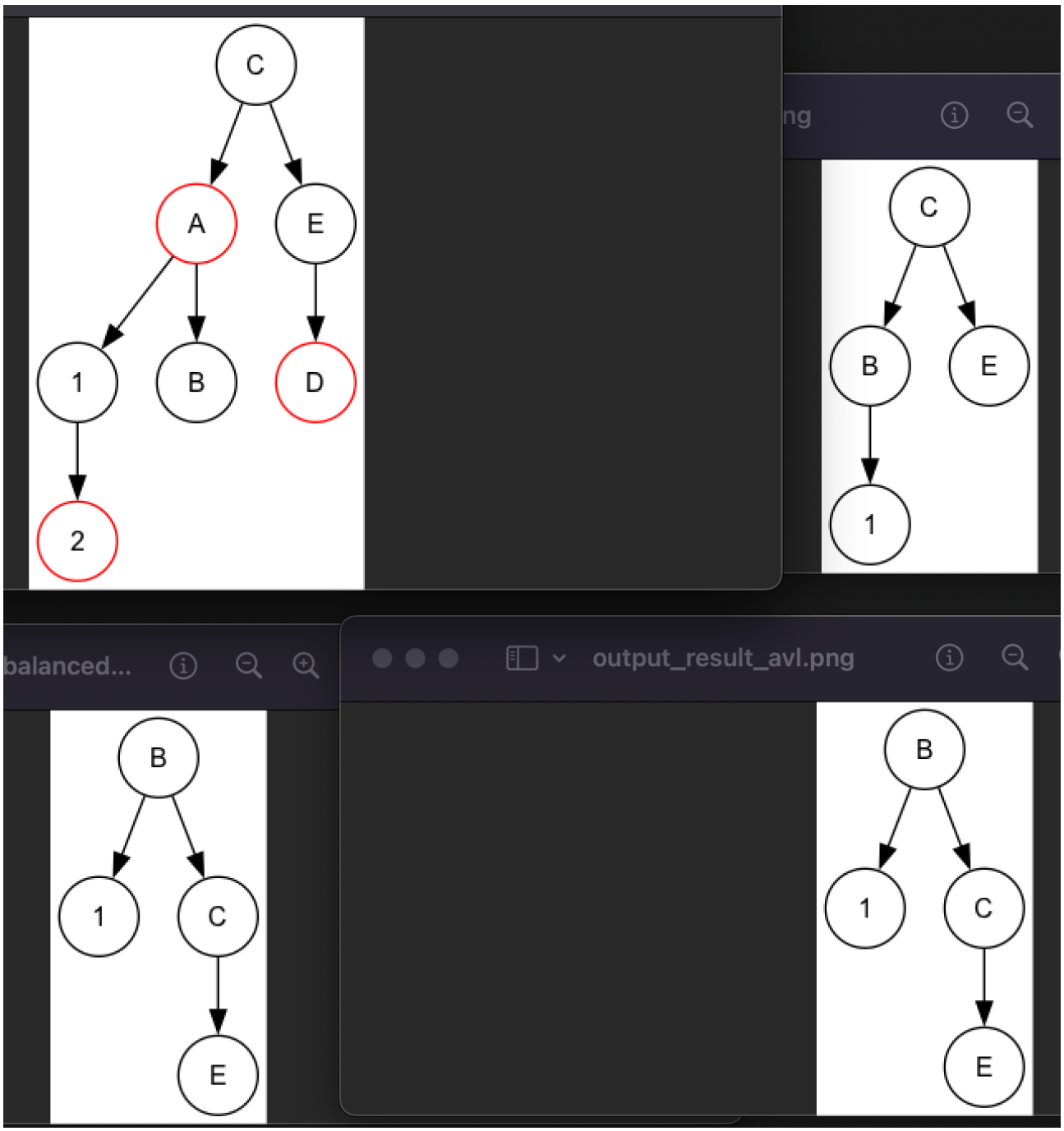
Len	Tree time(ns)	Balanced tree time(ns)	AVL tree time(ns)	Hash table opened time(ns)	Hash table closed time(ns)
5	36	35	35	37	39
9	47	49	45	39	48
10	36	43	37	42	51
15	49	56	43	39	38
13	69	86	71	40	40
26	318	300	269	33	35

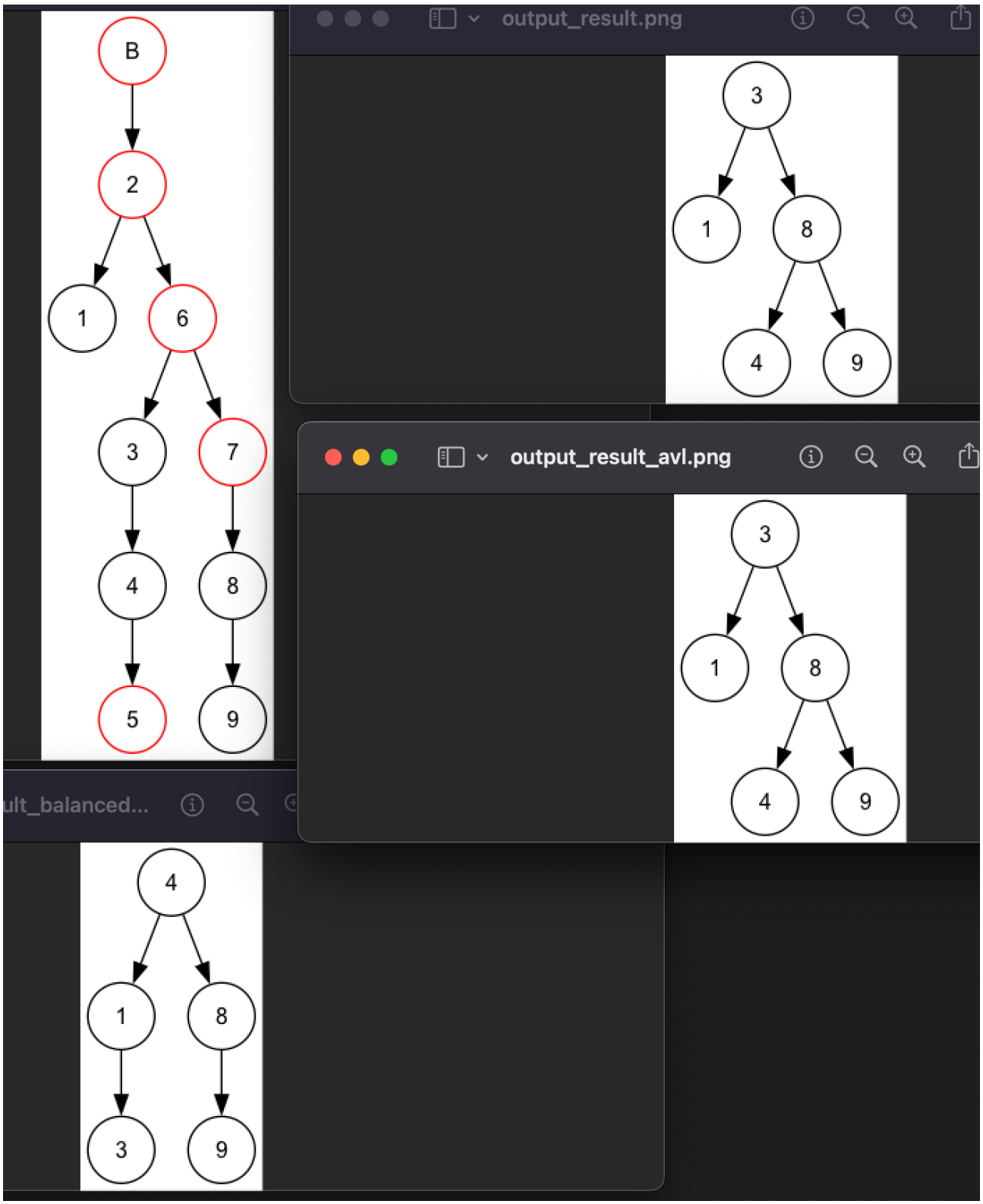
Len	Tree size(bytes)	Balanced tree size(bytes)	AVL tree size(bytes)	Hash table opened size(bytes)	Hash table closed size(bytes)
5	144	144	144	96	96
9	240	240	240	168	168
10	192	192	192	168	168
15	240	240	240	240	240
13	624	624	624	312	312
26	1248	1248	1248	624	624

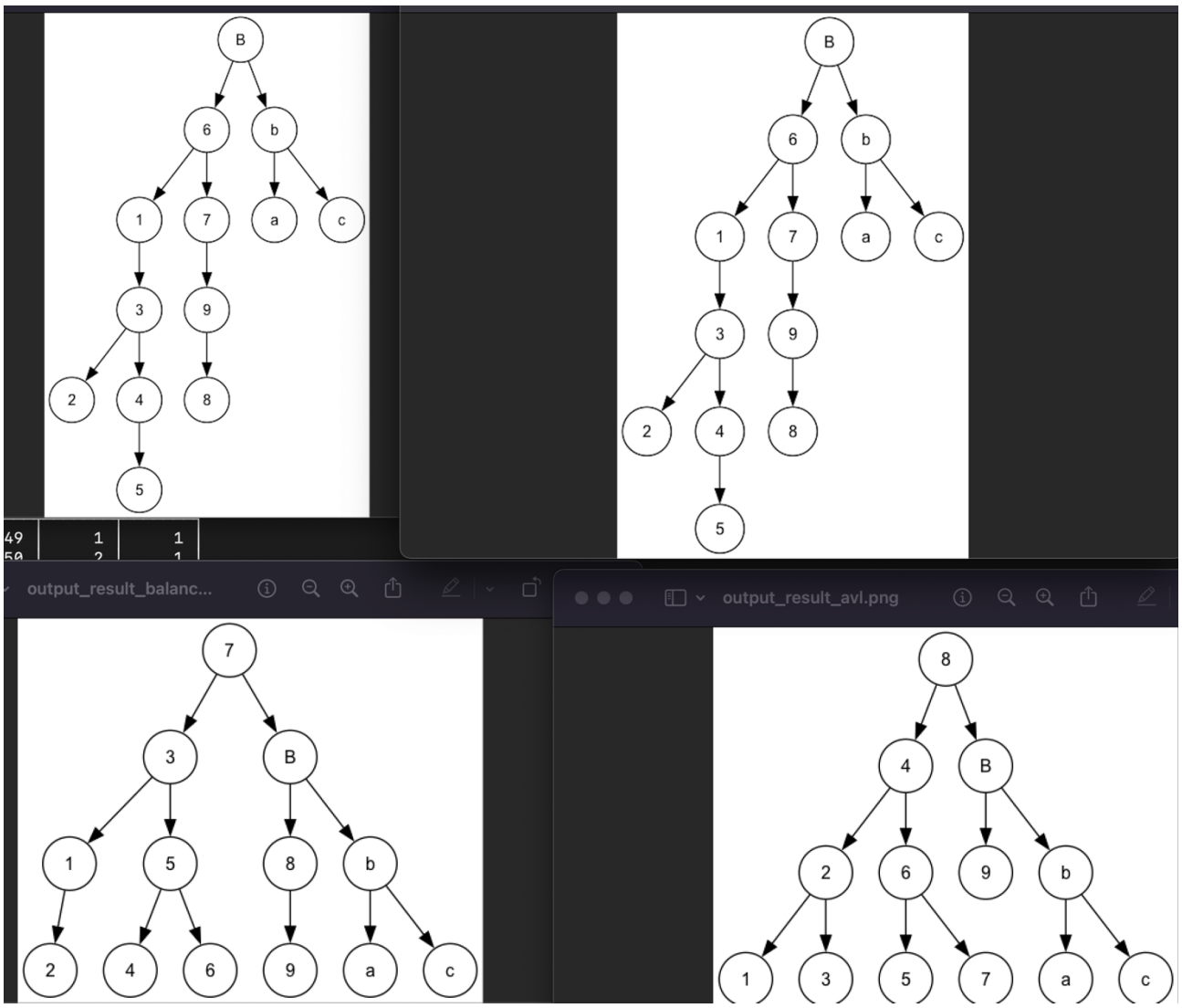
Len	Tree cmp count	Balanced cmp count	AVL cmp count	Hash cmp opened count	Hash cmp closed count
5	1	2	2	1	1
9	4	3	1	1	1
10	2	1	1	2	3
15	4	5	4	1	1
13	8	1	8	1	1
26	26	26	11	2	1

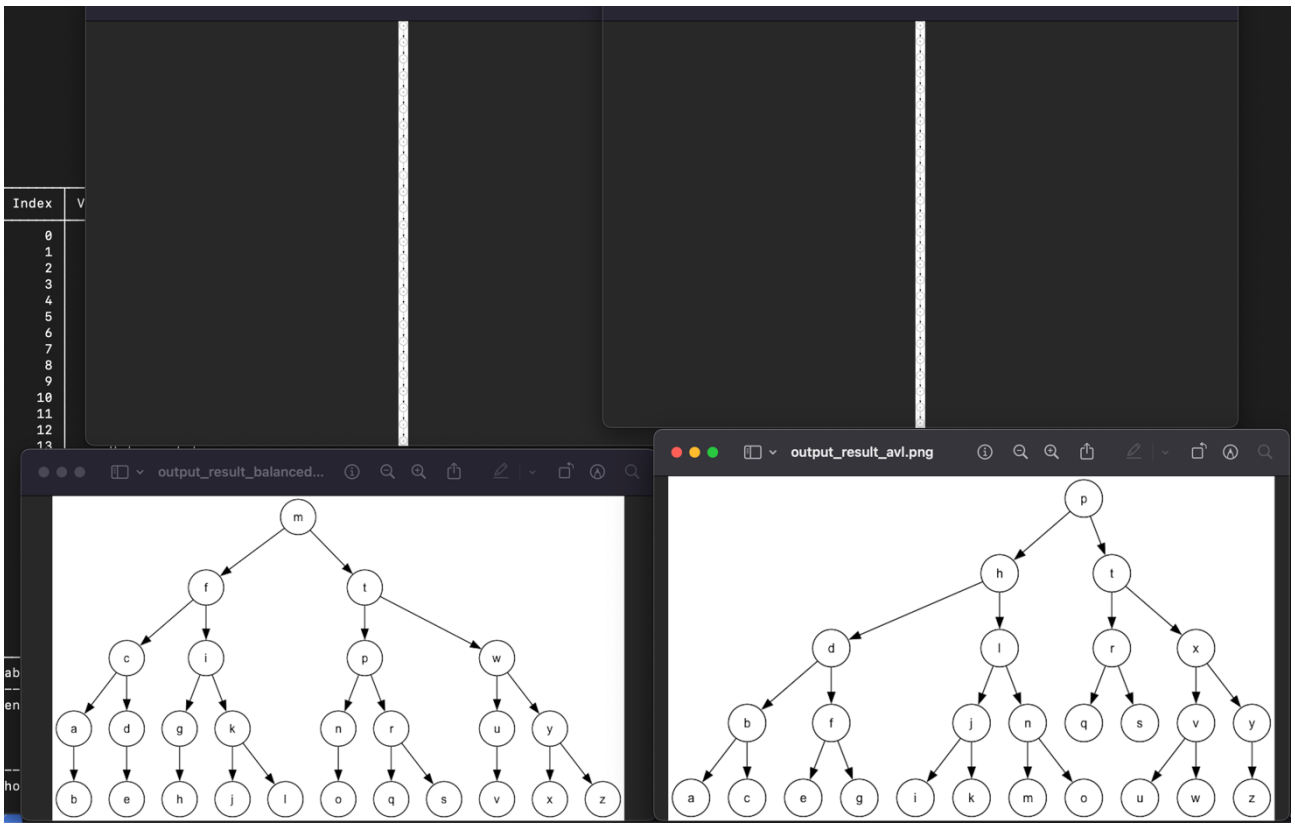



7
1
8
3
9
2
6
7
1
8
3
2
alanced...
sult_avl.png
3
1
7
2
8
2
1
7
3
8


C
A
E
1
B
D
2
C
B
E
1
B
1
C
E
B
1
C
E


B
2
1
6
3
7
4
8
5
9
output_result.png
3
1
8
4
9
output_result_avl.png
3
1
8
4
9
ult_balanced...
4
1
8
3
9


B
6
b
1
7
a
c
3
9
2
4
8
5
49 1 1
output_result_balanc...
output_result_avl.png
7
3
B
1
5
8
b
2
4
6
9
a
c
8
4
B
2
6
9
b
1
3
5
7
a
c



Выводы по проделанной работе

Среди рассматриваемых структур в большинстве случаев выигрывает хеш-таблица, так как поиск в ней происходит за O(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 O(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: O(1).

В АВЛ: O(log2n).

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