



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
«СБАЛАНСИРОВАННЫЕ ДЕРЕВЬЯ, ХЕШ-
ТАБЛИЦЫ»

Студент Козырных Александр

Группа ИУ7 – 32Б

Преподаватель Барышникова М. Ю.

Вариант 6

2023 год.

СОДЕРЖАНИЕ

<u>ОПИСАНИЕ УСЛОВИЯ ЗАДАЧИ.....</u>	<u>3</u>
<u>ОПИСАНИЕ ТЕХНИЧЕСКОГО ЗАДАНИЯ.....</u>	<u>3</u>
<u>ОПИСАНИЕ АЛГОРИТМА.....</u>	<u>4</u>
<u>ОПИСАНИЕ СТРУКТУРЫ ДАННЫХ.....</u>	<u>8</u>
<u>ОЦЕНКА ЭФФЕКТИВНОСТИ.....</u>	<u>10</u>
<u>ПАМЯТЬ (БАЙТ).....</u>	<u>10</u>
<u>ОТВЕТЫ НА КОНТРОЛЬНЫЕ ВОПРОСЫ.....</u>	<u>11</u>

ОПИСАНИЕ УСЛОВИЯ ЗАДАЧИ

Используя предыдущую программу (задача №6), построить дерево, например, для следующего выражения: $9+(8*(7+(6*(5+4)-(3-2))+1))$. При постфиксном обходе дерева, вычислить значение каждого узла и результат записать в его вершину. Получить массив, используя инфиксный обход полученного дерева. Построить для этих данных дерево двоичного поиска (ДДП), сбалансировать его. Построить хеш-таблицу для значений этого массива. Осуществить поиск указанного значения. Сравнить время поиска, объем памяти и количество сравнений при использовании ДДП, сбалансированных деревьев и хеш-таблиц

ОПИСАНИЕ ТЕХНИЧЕСКОГО ЗАДАНИЯ

Входные данные:

Пункты меню и целые числа.

Выходные данные:

Дерево двоичного поиска, AVL-дерево, целые числа и их хеш, время нахождения элемента и количество сравнений, которые для этого потребовались.

Обращение к программе:

Запускается через терминал командой: `./app.exe`.

Аварийные ситуации:

1. Некорректные данные переменной.
2. Ошибка выделения памяти.

ОПИСАНИЕ АЛГОРИТМА

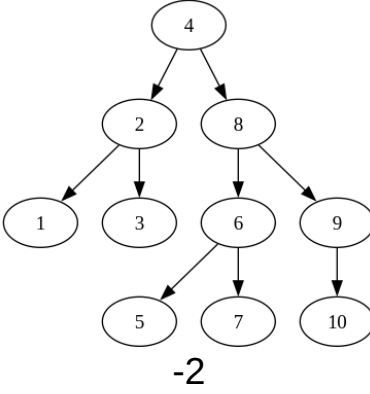
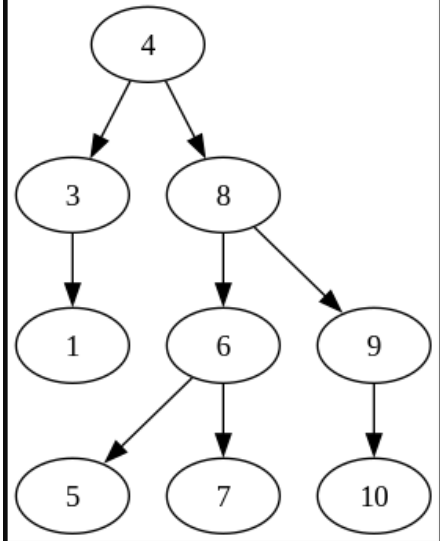
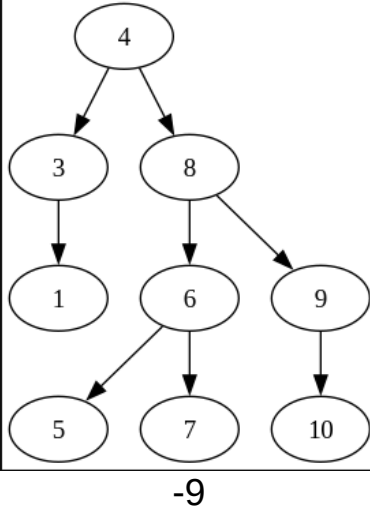
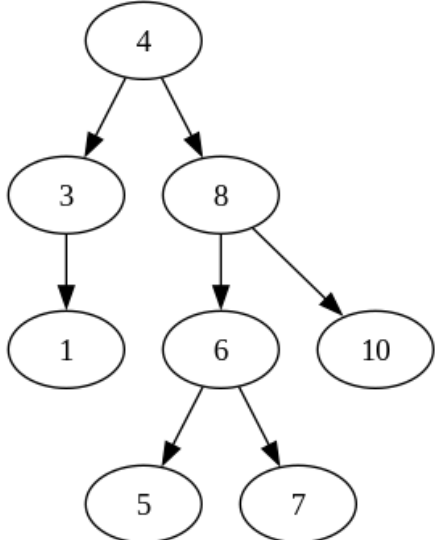
1. Вводится пункт меню
2. В зависимости от пункта меню вводятся или выводятся данные
3. Для решения задачи с набором чисел строится двоичное дерево выражения
4. В операторы этого двоичного дерева ложат результат этих операций
5. Инфиксным обходом получаем результаты операций над операндами
6. Добавляем результаты операций в хеш-таблицы и AVL-дерево
7. Выводим хеш-таблицы и AVL-дерево
8. Если выбран пункт про добавление элементов в хеш-таблицы, то он вставляется. При необходимости хеш-таблицы перестраиваются
9. Если выбран пункт про добавление элементов в AVL-дерево, то он вставляется. При необходимости дерево автоматически балансируется.

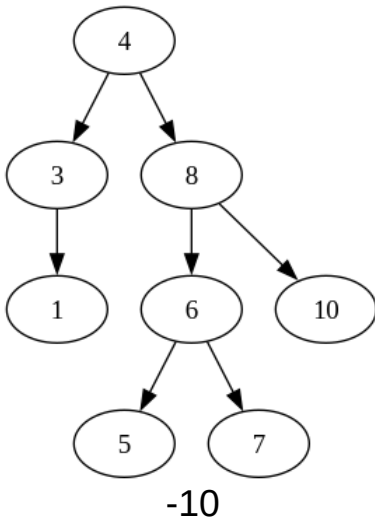
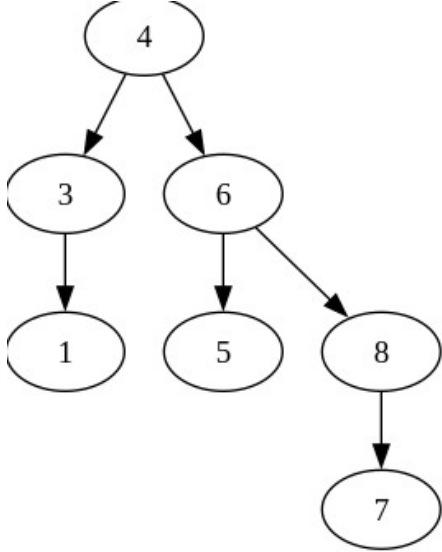
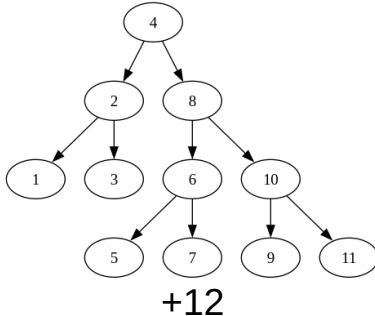
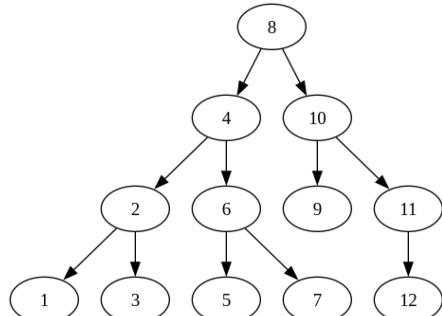
Набор тестов

№	Название теста	Пользовательский ввод	Вывод
1	Корректный ввод переменных	1 1 2 3 4 5 6 7 8 9	
2	Сбалансированное дерево из выражения, составленного из 1 2 3 4 5 6 7 8 9		<pre> graph TD 13((13)) --> 11((11)) 13 --> 114((114)) 11 --> m1((-1)) 11 --> 12((12)) 114 --> 57((57)) 114 --> 115((115)) 57 --> 44((44)) </pre>
3	Хеш таблица с открытым для		

	выражения из 1 2 3 4 5 6 7 8 9		1) Число: -1, хеш: 0 2) 3) 4) 5) 6) 7) 8) 9) 10) 11) 12) 13) Число: 12, хеш: 12 14) Число: 11, хеш: 12 15) Число: 13, хеш: 14 16) 17) 18) 19) 20) 21) 22) 23) 24) 25) 26) 27) 28) Число: 44, хеш: 27 29) Число: 115, хеш: 28 30) Число: 114, хеш: 28 31) Число: 57, хеш: 30 32)
4	Хеш таблица с закрытым для выражения из 1 2 3 4 5 6 7 8 9		1) Число: -1, хеш: 0 -> NULL 2) NULL 3) NULL 4) NULL

			<p>5) NULL 6) NULL 7) NULL 8) NULL 9) NULL 10) NULL 11) NULL 12) Число: 44, хеш: 11 -> NULL 13) Число: 115, хеш: 12 -> Число: 114, хеш: 12 -> Число: 12, хеш: 12 -> Число: 11, хеш: 12 -> NULL 14) NULL 15) Число: 13, хеш: 14 -> Число: 57, хеш: 14 -> NULL 16) NULL</p>
5	Инфиксный обход полученного дерева из выражения, составленного из 1 2 3 4 5 6 7 8 9		<p>115 114 12 13 -1 57 44 11</p>
6	Создание AVL-дерева на 1, 2, 3, 4, 5, 6 ,7 ,8 ,9, 10		<pre> graph TD 4((4)) --> 2((2)) 4 --> 8((8)) 2 --> 1((1)) 2 --> 3((3)) 8 --> 6((6)) 8 --> 9((9)) 6 --> 5((5)) 6 --> 7((7)) 9 --> 10((10)) </pre>

7	Удаление узла с двумя потомками	 <p style="text-align: center;">-2</p>	
8	Удаление узла с 1 потомком	 <p style="text-align: center;">-9</p>	

9	Удаление узла без ПОТОМКОВ		
10	Добавление с балансировкой		
11	Хеш-таблица с открытой адресацией с 10 элементами		<p>1) [Пусто]</p> <p>2) Число: 4, хеш: 1</p> <p>3) Число: 5, хеш: 2</p> <p>4) Число: 7, хеш: 3</p> <p>5) Число: 8, хеш: 3</p> <p>6) Число: 3, хеш: 5</p> <p>7) Число: 10, хеш: 4</p> <p>8) Число: 1, хеш: 7</p> <p>9) Число: 2, хеш: 8</p> <p>10) Число: 6, хеш: 8</p> <p>11) [Пусто]</p> <p>12) Число: 9, хеш: 11</p> <p>13) [Пусто]</p> <p>14) [Пусто]</p>

			15) [Пусто] 16) [Пусто]
12	Хеш-таблица с закрытой адресацией с 10 элементами		1) [NULL] 2) [4, хеш: 1] -> [NULL] 3) [5, хеш: 2] -> [NULL] 4) [7, хеш: 3] -> [8, хеш: 3] -> [NULL] 5) [10, хеш: 4] -> [NULL] 6) [3, хеш: 5] -> [NULL] 7) [NULL] 8) [1, хеш: 7] -> [NULL] 9) [2, хеш: 8] -> [6, хеш: 8] -> [NULL] 10) [NULL] 11) [NULL] 12) [9, хеш: 11] -> [NULL] 13) [NULL] 14) [NULL] 15) [NULL] 16) [NULL]
13	Некорректный ввод переменной	e10	Вы неверно ввели число!
14	Некорректный ввод переменной	a	Вы неверно ввели число!
15	Невозможно выделить память под вершину дерева	----	Не хватает памяти

ОПИСАНИЕ СТРУКТУРЫ ДАННЫХ

```
typedef struct avl_node_t avl_node_t;  
  
struct avl_node_t  
{  
    int num;  
    size_t height;  
  
    avl_node_t *left, *right;  
    /*  
        AVL-дерево.  
        num - информационная часть  
        height - высота данного узла  
        left, right - правый и левый узлы  
    */  
};
```

```
/*typedef на узел дерева*/  
typedef struct treenode tree_node_t;  
  
struct treenode {  
    tree_node_t *left;  
    tree_node_t *right;  
    int value;  
    char option;  
    /*  
        left - Левый потомок текущего узла  
        right - Правый потомок текущего узла  
        value - числовое значение узла  
        option - операция над числом (для вычисления выражения)  
    */  
};
```

```

struct _closed_hash_table
{
    /*
     * Хеш таблица
     * Является массивом списков
     */
    list_node_t **table;
    size_t size, capacity;
};

typedef struct _closed_hash_table *open_hash_t;

```

```

0
7 typedef struct value
8 {
9     int data;
0     bool is_initialized;
1 } value_t;
2
3 typedef value_t *table_t;
4
5 struct hash_set
6 {
7     /*
8     * Хеш-таблица.
9     * Является массивом данных, у которых есть флаг проинициализированности
0     */
1     table_t table;
2     size_t size, capacity;
3 };

```

ОЦЕНКА ЭФФЕКТИВНОСТИ

Тест проводился для 10000 элементов. Добавлялись числа от 0 до 9999. Как можем заметить, в худшем случае обычному несбалансированному ДДП потребовалось значительное количество времени на нахождение

каждого элемента. Для дерева время достаточно невелико, однако скорость хеш-таблиц практически нулевая.

ДДП работает так медленно, потому что при добавлении уже отсортированного набора данных дерево сводится к линейной структуре (список) с лишним количеством памяти (под ненужные боковые указатели). То есть в среднем нужно провести 5000 сравнений, чтобы найти каждый элемент. Для AVL-дерева потребовалась логарифмическая сложность нахождения. Для сравнения $\log_2(10000) \approx 13$. Из тестов можем заметить удивительное сходство. Для хеш-таблиц с закрытой адресацией скорость настолько быстрая потому, что они сразу знают (или практически точно определяют) индекс элемента, отчего в среднем у них 1 сравнение. По той же причине хеш-таблица с открытой адресацией работает быстро.

	ДДП (худший случай)	AVL-дерево	Хеш-таблица (открытая)	Хеш-таблица (закрытая)
Время поиска, мс	13.60	0.04	0.02	0.03
Кол-во сравнений	5000	12.36	1.21	1.31

ПАМЯТЬ (БАЙТ)

Самое эффективное по памяти оказалась открытая. Самое неэффективное по памяти могут разделить AVL-дерево и закрытая.

ДДП	AVL-дерево	Хеш с открытой	Хеш с закрытой
240000 байт	320000 байт	262144 байт	291072 байт

РЕСТРУКТУРИЗАЦИЯ

Для открытой

Для открытой адресации:
 1) Число: 2, хеш: 0
 2) Число: 1, хеш: 1

После добавления 3-ки:

```
Для открытой адресации:  
1) Число: 2, хеш: 0  
2) Число: 3, хеш: 1  
3) [ Пусто ]  
4) Число: 1, хеш: 3
```

Для ЗАКРЫТОЙ

```
Для закрытой адресации:  
1) [2, хеш: 0] -> [NULL]  
2) [1, хеш: 1] -> [NULL]
```

После добавления 3-ки :

```
Для закрытой адресации:  
1) [2, хеш: 0] -> [NULL]  
2) [3, хеш: 1] -> [NULL]  
3) [NULL]  
4) [1, хеш: 3] -> [NULL]
```

Реструктурирование хеш-таблицы с открытой адресацией происходит при достижении ее заполненности 80% (то есть отношение всех элементов, включая коллизии, ко всем выделенным ячейкам).

Реструктурирование хеш-таблицы с закрытой адресацией происходит только в том случае, если невозможно вставить элемент (достигнут конец массива и элемент не вставлен).

При реструктуризации в обоих случаях размер хеш-таблицы удваивается.

ОТВЕТЫ НА КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Чем отличается идеально сбалансированное дерево от AVL дерева?

В идеально сбалансированном дереве кол-во элементов в правом и левом поддереве отличается не более чем на единицу. В AVL дереве высоты правого и левого поддеревя отличается не более чем на единицу

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

Алгоритм одинаков.

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

Структура данных, позволяющая получать по ключу элемент массива, называется хеш-таблицей. Для доступа по ключу используется хеш-функция, которая получает нужный индекс массива. Хеш-функция должна возвращать одинаковые значения для одного ключа и использовать все индексы с одинаковой вероятностью (желательно!)

4. Что такое коллизии? Каковы методы их устранения.

Ситуация, когда из разных ключей хеш-функция выдаёт одни и тот же индекс, называется коллизией.

Метод цепочек – при коллизии элемент добавляется в список элементов этого индекса.

Линейная адресация – при коллизии ищется следующая незаполненная ячейка.

Произвольная адресация - используется заранее сгенерированный список случайных чисел для получения последовательности.

Двойное хеширование – использовать разность 2 разных хеш-функций.

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

При большом количестве коллизий.

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

Скорость поиска в хеш-таблице зависит от числа коллизий. При небольшом числе коллизий для поиска элемента совершается мало сравнений и поиск получается быстрее чем в деревьях (с хорошей хеш-функцией).

АВЛ дерево быстрее при поиске за счёт более равномерного распределения элементов, чем в ДДП.

Вывод

Самое эффективное решение – хеш-таблица с открытой адресацией. По скорости доступа — АВЛ-дерево (из деревьев) и две хеш-таблицы. Самое затратное по памяти оказалось АВЛ-дерево. Самое медленное решение оказывается у ДДП. К тому же ДДП подвергнут разному поведению из-за специфики добавления в него элементов, отчего его использование сильно замедляет работу приложения.

Скорее всего, с ростом количества элементов разница между хеш-функцией и АВЛ-деревом будет расти в пользу хеш-таблицы, если у нее хорошая хеш-функция. Связано это с тем, что доступ к элементу хеш-таблицы считается с помощью хеш-функции и сводится практически к $O(1)$, когда у АВЛ-дерева это будет $O(\log_2(n))$ всегда.