



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, ХЕШ- ТАБЛИЦЫ

Студент Жаринов М. А.

Вариант 5

Группа ИУ7-32Б

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

Студент _____ Жаринов М. А.

Преподаватель _____ Барышникова М. Ю.

Описание условия задачи

Построить хеш-таблицу для зарезервированных слов языка C++ (не менее 20 слов), содержащую HELP для каждого слова. Выдать на экран подсказку по введенному слову. Выполнить программу для различных размерностей таблицы и сравнить время поиска и количество сравнений. Для указанных данных создать сбалансированное дерево. Добавить в таблицы и сбалансированное дерево подсказку по вновь введенному слову, используя при необходимости реструктуризацию таблицы. Сравнить эффективность добавления ключа в сбалансированное дерево и таблицу (с учетом ее реструктуризации).

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.

Техническое задание

Исходные данные:

Зарезервированные слова языка C++: Текстовый файл, содержащий некоторое количество слов и подсказок к ним, слово и подсказка на разных строках, слово не длиннее 15 символов, количество слов в первой строке файла.

Выходные данные:

Двоичное дерево поиска и AVL-дерево по данным словам.

Хеш-таблицы с закрытым и открытым хешированием по данным словам

Результаты сравнения всех четырех структур данных для добавления и поиска элементов по времени и количеству сравнений.

Реализуемые задачи:

Пункт меню 0 – Выход из программы

Пункт меню 1 – Инициализировать двоичное дерево поиска

Пункт меню 2 – Инициализировать AVL-дерево

Пункт меню 3 – Инициализировать хеш-таблицу с открытым хешированием

Пункт меню 4 – Инициализировать хеш-таблицу с закрытым хешированием

Пункт меню 5 – Считать файл в выбранную структуру данных

Пункт меню 6 – Добавить элемент в выбранную СД

Пункт меню 7 – Найти элемент в выбранной СД

Пункт меню 8 – Удалить элемент из выбранной СД

Пункт меню 9 – Вывести выбранную СД

Пункт меню 10 – Изменить максимальное количество сравнений до реструктуризации (только для хеш-таблиц)

Пункт меню 11 – Провести сравнение эффективности поиска и добавления элементов в ДДП, AVL-дерева, хеш-таблицах с различным хешированием

Способ обращения к программе:

Строка запуска программы ./app.exe

Аварийные ситуации:

1. Недостаток свободного места в оперативной памяти. Код ошибки 2
2. Некорректный ввод в консоль. Код ошибки 1.

Все остальные ошибочные ситуации (некорректный пункт меню, несуществующий файл, отсутствующий элемент в СД, пустая СД) не являются аварийными и вызывают возвращение в меню.

Внутренние структуры данных

Информация о зарезервированном слове хранится в структуре

reserved_word_info_t.

```
#define MAX_WORD_LEN 16
typedef char reserved_word_t[MAX_WORD_LEN];
typedef struct
{
    char *help;
    reserved_word_t word;
} reserved_word_info_t;
```

Поле word- зарезервированное слово

Поле help- подсказка

Элементы ДДП хранятся в структуре bin_tree_node_t.

```
typedef struct bin_tree_node
{
    reserved_word_info_t reserved_word_info;
    struct bin_tree_node *left;
    struct bin_tree_node *right;
} bst_tree_node_t;
```

Поле reserved_word_info -данные о слове

Поле left -указатель на левого потомка

Поле right -указатель на правого потомка

Элементы AVL-дерева хранятся в структуре `avl_tree_node_t`.

```
typedef struct avl_tree_node
{
    reserved_word_info_t reserved_word_info;
    int height;
    struct avl_tree_node *left;
    struct avl_tree_node *right;
} avl_tree_node_t;
```

Поле `reserved_word_info` – данные о слове

Поле `height` – высота поддерева

Поле `left` – указатель на левого потомка

Поле `right` – указатель на правого потомка

Элементы хеш-таблицы с открытым хешированием хранятся в структуре `open_ht_node_t`.

```
typedef struct open_ht_node
{
    reserved_word_info_t reserved_word_info;
    struct open_ht_node *next;
} open_ht_node_t;
```

Поле `reserved_word_info` – данные о слове

Поле `next` – указатель на следующий элемент

Хеш-таблица с открытым хешированием хранится в структуре `open_hashtable_t`.

```
typedef struct
{
    open_ht_node_t **array;
    int size;
    int max_comparisons;
} open_hashtable_t;
```

Поле `array` – массив указателей на вершины списков

Поле `size` – размер хеш-таблицы

Поле `max_comparisons` – максимальное количество сравнений до реструктуризации

Элементы хеш-таблицы с закрытым хешированием хранятся в структуре `closed_ht_elem_t`.

```
typedef struct
{
    reserved_word_info_t reserved_word_info;
    enum
    {
        FREE,
        USED,
        DELETED
    } state;
} closed_ht_elem_t;
```

Поле `reserved_word_info` – данные о слове

Поле `state` – состояние элемента

Хеш-таблица с закрытым хешированием хранится в структуре `closed_hashtable_t`.

```
typedef struct
{
    closed_ht_elem_t *array;
    int size;
    int max_comparisons;
} closed_hashtable_t;
```

Поле `array` – массив элементов таблицы

Поле `size` – размер хеш-таблицы

Поле `max_comparisons` – максимальное количество сравнений до реструктуризации

Все СД хранятся в структуре ассоциативного массива `assoc_array_t`.

```
typedef struct
{
    void *data;
    bool (*insert)(void *data, reserved_word_info_t *word_info, int
*n_comps);
    bool (*remove)(void *data, reserved_word_t word, int *n_comps);
    char *(*find)(void *data, reserved_word_t word, int *n_comps);
    void (*print)(void *data);
    bool (*set_comps)(void *data, int comps);
    void (*clean_and_set_size)(void *data, int new_size);
    void (*restruct)(void *data, int comps);
    void (*free)(void *data);
} assoc_array_t;
```

Поле `data` – указатель на основную СД

Поле `insert` – указатель на функцию вставки в СД

Поле `remove` – указатель на функцию удаления из СД

Поле `find` – указатель на функцию поиска в СД

Поле `print` – указатель на функцию вывода на экран СД

Поле `set_comps` – указатель на функцию изменения порога сравнений
для хеш-таблиц

Поле `clean_and_set_size` – указатель на функцию очистки и
изменения размера для хеш-таблиц

Поле `restruct` – указатель на функцию реструктуризации хеш-таблиц

Поле `free` – указатель на функцию освобождения СД

Описание алгоритма

Программа запрашивает номер элемента в меню, считывает его и выполняет соответствующее действие:

Пункт меню 6 – Добавление элемента в СД

Программа запрашивает и считывает информацию о зарезервированном слове

ДДП:

1. Если текущий узел пуст, то вставить в него
2. Если текущий узел равен по ключу, то вернуть ошибку
3. Если текущий узел меньше по ключу вставляемого, то вставить в

правое поддерево, иначе в левое

Для AVL дерева выполняется балансировка

Для открытого хеширования

1. Вычисляется хеш слова
2. Слово вставляется в цепочку по соответствующему индексу

Для закрытого хеширования:

1. Вычисляется хеш слова
2. Выполняется поиск в массиве начиная с рассчитанного индекса

вплоть до первого пустого или удаленного элемента или до полного прохода массива

Пункт меню 7 – Поиск элемента в СД

Программа запрашивает и считывает слово

ДДП и AVL дерево:

1. Если текущий элемент больше по ключу, то искать в левом

поддереве, если меньше, то в правом

2. Если элемент равен, то вернуть его

Для открытого хеширования

1. Вычисляется хеш слова
2. Слово ищется в цепочке по соответствующему индексу

Для закрытого хеширования:

1. Вычисляется хеш слова
2. Выполняется поиск в массиве начиная с рассчитанного индекса

вплоть до первого пустого элемента или до полного прохода массива

Пункт меню 8 – Удаление элемента из СД

Программа запрашивает и считывает слово

ДДП:

1. Если элемент больше по ключу удаляемого, то удалить в левом поддереве, если меньше, то в правом
2. Если элемент равен по ключу, то
3. Если правый потомок пуст, то записать на место текущего узла левый потомок
4. Иначе если левый пуст, то записать правого потомка
5. Иначе найти самый левый элемент в правом поддереве, записать его данные в текущий элемент, и вызывать функцию удаления этого элемента в правом поддереве

Для AVL – дерева выполняется балансировка

Для открытого хеширования

1. Вычисляется хеш слова
2. Слово удаляется из цепочки по соответствующему индексу

Для закрытого хеширования:

1. Вычисляется хеш слова
2. Выполняется поиск в массиве начиная с рассчитанного индекса вплоть до первого пустого элемента или до полного прохода массива, найденный элемент помечается как удаленный

Реструктуризация выполняется, если при вставке или поиске количество сравнений было больше чем заданный порог (по умолчанию 3). При реструктуризации создается новая хеш-таблица размером в 2 раза больше исходной, и в нее переносятся все элементы исходной хеш-таблицы с пересчитыванием хешей

Основные функции

Функция для обработки выбранного пункта меню. Принимает номер выбранного пункта меню и указатель на ассоциативный массив. Возвращает код ошибки

```
int process_menu(menu_item_t menu_item, assoc_array_t *assoc_array)
```

Функция вставки элемента в ДДП. Принимает указатель на дерево, слово для вставки, указатель на число сравнений. Возвращает флаг успешности вставки

```
bool bst_tree_insert(bst_tree_node_t **tree, reserved_word_info_t  
*word_info, int *n_comps)
```

Функция удаления элемента из ДДП. Принимает указатель на дерево, слово к удалению, указатель на число сравнений. Возвращает флаг успешности удаления

```
bool bst_tree_remove(bst_tree_node_t **tree, reserved_word_t word, int  
*n_comps)
```

Функция поиска элемента в ДДП. Принимает указатель на дерево, слово к поиску, указатель на число сравнений. Возвращает строку подсказки.

```
char *abs_bst_tree_find(void *tree, reserved_word_t word, int *n_comps)
```

Функция освобождения ДДП. Принимает указатель на указатель на корень.

```
static void bst_tree_free(bst_tree_node_t *tree)
```

Функция вставки элемента в AVL-дерево. Принимает указатель на дерево, слово для вставки, указатель на число сравнений. Возвращает флаг успешности вставки

```
bool avl_tree_insert(avl_tree_node_t **node, reserved_word_info_t  
*word_info, int *n_comps)
```

Функция удаления элемента из AVL-дерева. Принимает указатель на дерево, слово к удалению, указатель на число сравнений. Возвращает флаг успешности удаления

```
bool avl_tree_remove(avl_tree_node_t **node, reserved_word_t word, int  
*n_comps)
```

Функция поиска элемента в AVL-дереве. Принимает указатель на дерево, слово к поиску , указатель на число сравнений. Возвращает строку подсказки.

```
char *abs_avl_tree_find(void *tree, reserved_word_t word, int *n_comps)
```

Функция освобождения AVL-деревя. Принимает указатель на указатель на корень.

```
void abs_avl_tree_free(void *tree)
```

Функция вставки элемента в хеш-таблицу с открытым хешированием.

Принимает указатель на таблицу, слово для вставки, указатель на число сравнений. Возвращает флаг успешности вставки

```
bool abs_open_hashtable_insert(void *data, reserved_word_info_t  
*reserved_word_info, int *n_comps)
```

Функция удаления элемента из хеш-таблицы с открытым хешированием.

Принимает указатель на таблицу, слово к удалению, указатель на число сравнений. Возвращает флаг успешности удаления

```
bool abs_open_hashtable_remove(void *data, reserved_word_t word, int  
*n_comps)
```

Функция поиска элемента в хеш-таблице с открытым хешированием.

Принимает указатель на таблицу, слово к поиску , указатель на число сравнений. Возвращает строку подсказки.

```
char *abs_open_hashtable_find(void *data, reserved_word_t reserved_word,  
int *n_comps)
```

Функция освобождения хеш-таблицы с открытым хешированием.

Принимает указатель на указатель на таблицу.

```
void abs_open_hashtable_free(void *data)
```

Функция реструктуризации хеш-таблицы с открытым хешированием.

Принимает указатель на таблицу и новый размер.

```
void open_hashtable_restruct(open_hashtable_t *open_hashtable, int  
new_size)
```

Функция вставки элемента в хеш-таблицу с закрытым хешированием.

Принимает указатель на таблицу, слово для вставки, указатель на число сравнений. Возвращает флаг успешности вставки

```
bool abs_closed_hashtable_insert(void *data, reserved_word_info_t
*reserved_word_info, int *n_comps)
```

Функция удаления элемента из хеш-таблицы с закрытым хешированием.

Принимает указатель на таблицу, слово к удалению, указатель на число сравнений. Возвращает флаг успешности удаления

```
bool abs_closed_hashtable_remove(void *data, reserved_word_t word, int
*n_comps)
```

Функция поиска элемента в хеш-таблице с закрытым хешированием.

Принимает указатель на таблицу, слово к поиску, указатель на число сравнений. Возвращает строку подсказки.

```
char *abs_closed_hashtable_find(void *data, reserved_word_t
reserved_word, int *n_comps)
```

Функция освобождения хеш-таблицы с закрытым хешированием.

Принимает указатель на таблицу.

```
void abs_closed_hashtable_free(void *data)
```

Функция реструктуризации хеш-таблицы с закрытым хешированием.

Принимает указатель на таблицу и новый размер.

```
void closed_hashtable_restruct(closed_hashtable_t *closed_hashtable, int
new_size)
```

Функция анализа разных СД.

```
void run_analysis(void);
```

Функция вычисления хеш-функции от строки. Принимает указатель на строку и размер хеш-таблицы.

```
int hash_function(char *word, int size)
{
    unsigned long long hash = 5381;
    while (*word != '\0')
    {
        hash = hash * 33 + *word;
        word++;
    }
    return (int)(hash % (unsigned long long)size);
}
```


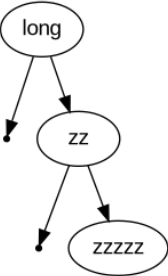
Функция начинает вычисление с большого простого числа. Далее она умножает хеш на простое число 33 и прибавляет к нему код символа в цикле.

Набор тестов

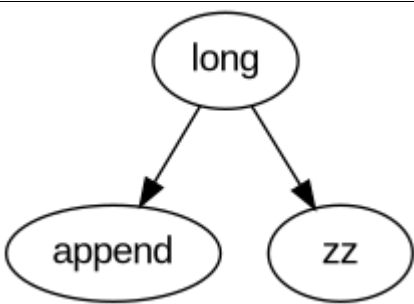
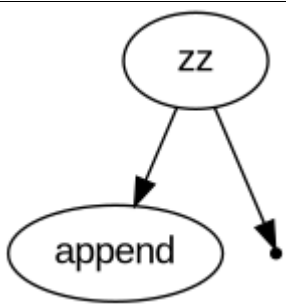
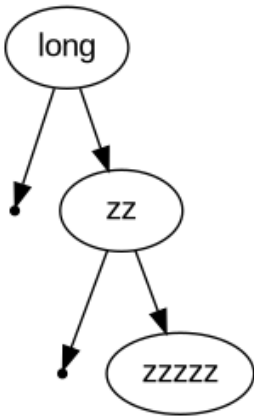
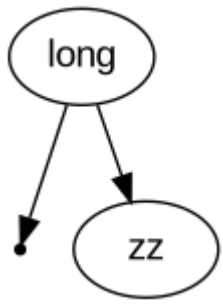

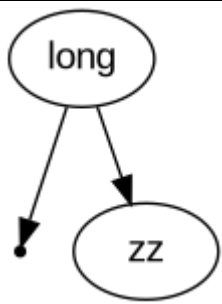
Обработка пунктов меню

Описание теста	Вход	Результат
Отрицательное число	-1	Для выбора пункта меню введите целое число от 0 до 11
Число больше 11	12	Для выбора пункта меню введите целое число от 0 до 11
Пустой ввод		Для выбора пункта меню введите целое число от 0 до 11
Максимальный пункт меню	11	[сведения о скорости поиска и вставки]
Минимальный пункт меню	0	[выход из программы]
Символ вместо числа	a	Для выбора пункта меню введите целое число от 0 до 11
Выбор пунктов меню, которые требуют инициализированную СД при неинициализированной СД	5	Нельзя производить данное действие над неинициализированной СД!


Добавление элемента в ДДП

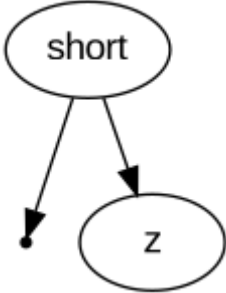
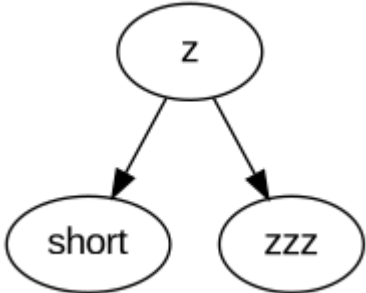
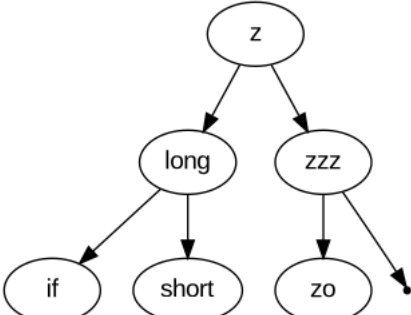
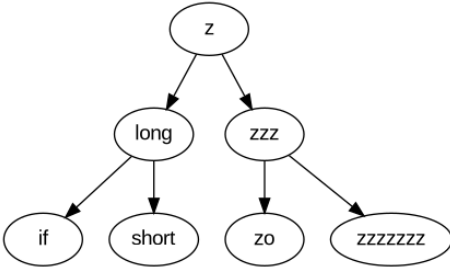
Описание теста	Слово	Входное дерево	Выходное дерево
Добавление в пустое дерево	long	Пустое	
Добавление существующего элемента	zz		Слово уже есть в СД!

Удаление элемента из ДДП


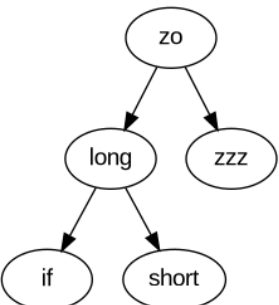
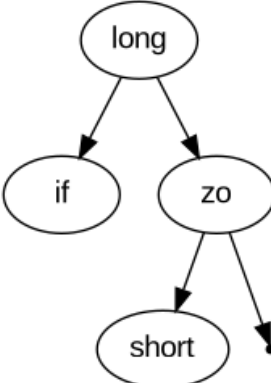
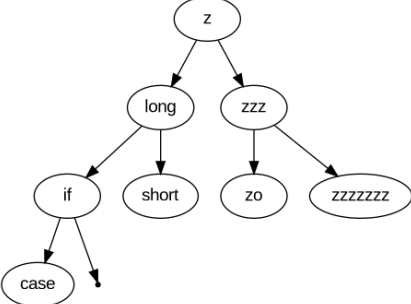
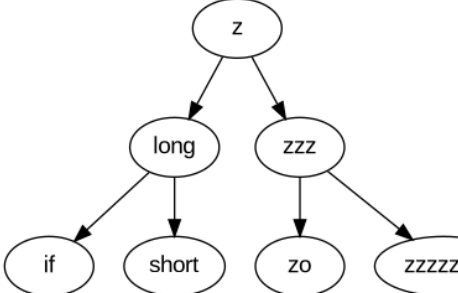
Описание теста	Слово	Входное дерево	Выходное дерево
Удаление элемента с обоими предками	long	 <pre> graph TD long((long)) --> append1((append)) long --> zz1((zz)) </pre>	 <pre> graph TD zz2((zz)) --> append2((append)) zz2 --> null1(()) </pre>
Удаление элемента без предков	zzzzz	 <pre> graph TD long3((long)) --> null3(()) long3 --> zz3((zz)) zz3 --> null4(()) zz3 --> zzzzz((zzzzz)) </pre>	 <pre> graph TD long4((long)) --> null5(()) long4 --> zz4((zz)) </pre>
Удаление единственного узла	append	 <pre> graph TD append3((append)) </pre>	Пустое дерево
Удаление отсутствующего элемента	zzzzz	 <pre> graph TD long5((long)) --> null6(()) long5 --> zz5((zz)) </pre>	Зарезервированное слово не найдено!

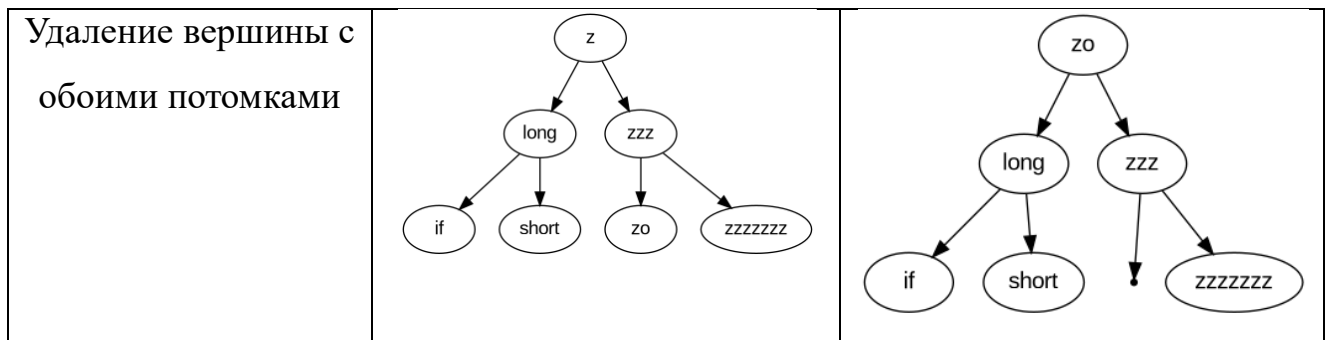
Добавление элемента в AVL-дерево

Описание теста	Входное дерево и элемент	Выходное дерево
Добавление в пустое дерево	short	 <pre> graph TD short1((short)) </pre>

Добавление нарушит сбалансированность	<p>zzz</p> 	
Добавление не нарушит сбалансированность	<p>zzzzzzzz</p> 	

Удаление элемента из AVL-дерева

Описание теста	Входное дерево	Выходное дерево
Удаление последнего элемента	<p>long</p> 	<p>Пустое дерево</p>
Удаление нарушит сбалансированность	<p>zzz</p> 	
Удаление не нарушит сбалансированность		



Добавление элемента хеш-таблицу с открытым хешированием

Описание теста	Вход и элемент	Выход
Добавление в пустую таблицу	void	... NULL void -> NULL ...
Добавление в конец цепочки	NULL continue -> static -> void -> long	NULL continue -> long -> static -> void ->
Добавление существующего элемента	NULL continue -> long -> static -> void -> long	Слово уже есть в СД!

Удаление элемента из хеш-таблицы с открытым хешированием

Описание теста	Вход	Выход
Удаление элемента без коллизий	NULL char -> NULL char	NULL NULL NULL

Удаление несуществующего элемента		Зарезервированное слово не найдено!
Удаление элемента из цепочки	else -> int -> ghr -> uikuty -> auto -> ghr	else -> int -> uikuty -> auto ->

Добавление элемента хеш-таблицу с закрытым хешированием

Описание теста	Вход и элемент	Выход
Добавление в пустую таблицу	int	20 - - 21 21 int 22 - -
Добавление на место удаленного	28 27 long 29 29 short 30 28 Элемент удален 31 31 while 32 28 void static	28 27 long 29 29 short 30 28 static 31 31 while
Добавление с коллизией	30 28 static 31 31 while 32 28 void 33 - - 34 - -	30 28 static 31 31 while 32 28 void 33 32 switch 34 - -

Удаление элемента из хеш-таблицы с закрытым хешированием

Описание теста	Вход	Выход
Удаление элемента без коллизий	24 - - 25 25 if 26 - - if	24 - - 25 25 Элемент удален 26 - -
Удаление несуществующего элемента		Зарезервированное слово не найдено!
Удаление элемента с коллизией	29 29 short 30 28 static 31 31 while	29 29 short 30 28 Элемент удален 31 31 while

Реструктуризация хеш-таблиц

Описание теста	Вход	Выход
Реструктуризация хеш-таблицы с открытым хешированием	NULL while -> if -> char -> for -> NULL int -> NULL NULL NULL NULL	NULL NULL NULL NULL NULL for -> NULL ... NULL if -> NULL NULL char -> while -> int -> NULL

Реструктуризация	Индекс Хеш Слово	Индекс Хеш Слово
хеш-таблицы с закрытым хешированием	0 - -	0 - -
	1 - -	1 - -
	2 2 while	2 - -
	3 3 if	3 - -
	4 4 char	4 - -
	5 4 for	5 5 for
	6 6 int	6 - -
	7 - -	...
	8 - -	16 - -
	9 - -	17 17 if
	10 - -	18 - -
		19 - -
		20 20 char
		21 21 while
		22 21 int

Результаты сравнения

Для анализа я измерил среднее время и среднее количество сравнений при добавлении и поиске элементов в ассоциативном массиве из 1000 элементов, реализованном с помощью ДДП, AVL-дерева, хеш-таблицы с 2 видами хеширования 20 элементов.

Результаты сравнения (в наносекундах)

Двоичное дерево поиска		
Операция	Время	Сравнения
Добавление	4064	12.55
Поиск	2789	13.55
Занимаемый объем памяти: 40000 байт		
AVL-дерево		
Операция	Время	Сравнения
Добавление	7415	10.30
Поиск	2107	10.85
Занимаемый объем памяти: 48000 байт		
Хеш-таблица с открытым хешированием		
Операция	Время	Сравнения
Добавление	1156	1.45
Поиск	369	1.45
Занимаемый объем памяти: 40000 байт		
Хеш-таблица с закрытым хешированием		
Операция	Время	Сравнения
Добавление	704	2.20
Поиск	655	2.20
Занимаемый объем памяти: 32000 байт		

Таким образом, можно заметить, что данные согласуются с теорией. Так, среднее количество сравнений в AVL-дерево составило ровно $\log_2(1000)$, количество сравнений в ДДП несколько больше, так как оно не сбалансировано.

Количество сравнений в хеш-таблицах не превышает 2.2 (а для открытого хеширования 1.5), что обеспечивает в разы более быстрый доступ к элементам.

Время добавления в AVL-дерево несколько больше чем время добавления в обычное ДДП, так как добавление в AVL требует дополнительных действий по перебалансировке.

Время добавления в хеш-таблицу с открытым хешированием значительно больше времени поиска, так как добавление требует затрат времени на выделение памяти под элемент, в то время как при закрытом хешировании этого не требуется, и в нем время обеих операций равно.

Вывод

При решении задач, которые требуют хранения данных типа ключ-значение, целесообразно, использовать такую абстрактную структуру данных, как ассоциативный массив, реализованных с помощью таких СД, как ДДП, AVL-дерево или хеш-таблицы.

Если данные поддаются хешированию, то лучше использовать хеш-таблицу.

Если важна скорость добавления, то лучше использовать закрытое хеширование, так как в нем вставка не выделяет память, иначе оптимальнее открытое хеширование.

Если же данные плохо поддаются хешированию, то имеет смысл использовать деревья поиска. Если важна простота написания алгоритма, или критически важно время добавления(при этом гарантируется что данные будут более-менее случайны) или объем памяти то можно использовать обычное ДДП, а иначе лучше использовать сбалансированное AVL-дерево.

Ответы на контрольные вопросы

1. Чем отличается идеально сбалансированное дерево от AVL дерева?

В идеально сбалансированном дереве число вершин в левом и правом поддеревьях каждой вершины отличается не более, чем на единицу. В AVL-дереве у каждого узла дерева высота двух поддеревьев отличается не более чем на единицу. ИСД также не является деревом поиска

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

Ничем, так как AVL-дерево изменяет только алгоритмы вставки и удаления.

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

Хеш-таблица – это массив, индекс данных в котором определяется хеш-функцией.

4. Что такое коллизии? Каковы методы их устранения.

Коллизии – ситуации, в которых разным ключам соответствует одинаковых хеш, и соответственно одинаковых индекс в массиве. Основные методы устранения – закрытое хеширование (элементы сдвигаются не на свой индекс) и метод цепочек (элементы располагаются в списке)

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

Поиск в хеш-таблицах становится неэффективен, если в хеш-таблице большое количество коллизий, что приводит к большому количеству сравнений при поиске

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

Поиск в AVL дереве $O(\log_2 n)$

В ДДП – в среднем случае $O(\log_2 n)$, в худшем случае $O(n)$

В хеш таблице – в среднем случае $O(1)$, в худшем случае $O(n)$

В файле – $O(n)$