Задание 2
Бинарные деревья поиска и хеш-таблицы

Постановка задачи

Требуется реализовать на языке C две библиотеки для работы с бинарным деревом поиска (Binary search tree) и хеш-таблицей (Hash table). Ключом в обоих случаях является строка (char []), а значением целое число (int).

Функции для работы с бинарным деревом поиска должны быть помещены в файлы bstree.c (реализация функций) и bstree.h (объявление функций). В файлах необходимо реализовать следующие функции:

- `struct bstree *bstree_create(char *key, int value)`
- `void bstree_add(struct bstree *tree, char *key, int value)`
- `struct bstree *bstree_lookup(struct bstree *tree, char *key)`
- `struct bstree *bstree_min(struct bstree *tree)`
- `struct bstree *bstree_max(struct bstree *tree)`

Функции для работы с хеш-таблицей должны быть помещены в файлы hashtab.c (реализация функций) и hashtab.h (объявление функций). В файлах необходимо реализовать следующие функции:

- `unsigned int hashtab_hash(char *key)  /* Реализация из лекции 6 */`
- `void hashtab_init(struct listnode **hashtab)`
- `void hashtab_add(struct listnode **hashtab, char *key, int value)`
- `struct listnode *hashtab_lookup(struct listnode **hashtab, char *key)`
- `void hashtab_delete(struct listnode **hashtab, char *key)`

Цель работы — провести экспериментальное исследование эффективности бинарных деревьев поиска и хеш-таблиц.

В отчет следует включить:

- общее описание бинарных деревьев поиска и хеш-таблиц (основные операции и их вычислительные сложности)
- таблицы и графики с результатами экспериментов и выводы по каждому эксперименту

Распределение заданий по вариантам

Вариант (см. на сайте)	**1**	**2**	**3**
1	Эксперимент 1	Эксперимент 2	Эксперимент 6 - хеш-функции KP, Add
2	Эксперимент 1	Эксперимент 3	Эксперимент 6 - хеш-функции KP, XOR
3	Эксперимент 1	Эксперимент 4	Эксперимент 6 - хеш-функции KP, FNV
4	Эксперимент 1	Эксперимент 5	Эксперимент 6 - хеш-функции KP, Jenkins
5	Эксперимент 1	Эксперимент 2	Эксперимент 6 - хеш-функции KP, ELF
6	Эксперимент 1	Эксперимент 3	Эксперимент 6 - хеш-функции KP, DJB
7	Эксперимент 1	Эксперимент 4	Эксперимент 6 - хеш-функции KP, DJB
8	Эксперимент 1	Эксперимент 5	Эксперимент 6 - хеш-функции KP, ELF
9	Эксперимент 1	Эксперимент 2	Эксперимент 6 - хеш-функции KP, Jenkins
10	Эксперимент 1	Эксперимент 3	Эксперимент 6 - хеш-функции KP, FNV
11	Эксперимент 1	Эксперимент 4	Эксперимент 6 - хеш-функции KP, XOR
12	Эксперимент 1	Эксперимент 5	Эксперимент 6 - хеш-функции KP, Add
13	Эксперимент 1	Эксперимент 2	Эксперимент 6 - хеш-функции KP, Jenkins
14	Эксперимент 1	Эксперимент 3	Эксперимент 6 - хеш-функции KP, ELF
15	Эксперимент 1	Эксперимент 4	Эксперимент 6 - хеш-функции KP, FNV
16	Эксперимент 1	Эксперимент 5	Эксперимент 6 - хеш-функции KP, XOR
17	Эксперимент 1	Эксперимент 2	Эксперимент 6 - хеш-функции KP, DJB

Экспериментальное исследование

Эксперимент 1 Сравнение эффективности поиска элементов в бинарном дереве поиска и хеш-таблице в среднем случае (average case)

Требуется заполнить таблицу 1 и построить графики зависимости времени t выполнения операции поиска (lookup) элемента в бинарном дереве поиска и хеш-таблице от числа n элементов уже вставленных в словарь.

В качестве ключей использовать слова из романа Л.Н. Толстого «Война и Мир» (можно использовать любой текстовый файл с большим числом слов). Файл включен в архив.

В качестве искомого ключа следует выбрать случайное слово, которое уже было добавлено в словарь.

Таблица 1

#	Количество элементов в словаре	Время выполнения функции bstree_lookup, с	Время выполнения функции hashtab_lookup, с
1	10 000		
2	20 000		
3	30 000		
...			
20	200 000		

Пример оформления графиков приведен на рис. 1 (в архив включен пример Excel-файла для построения графиков такого вида).

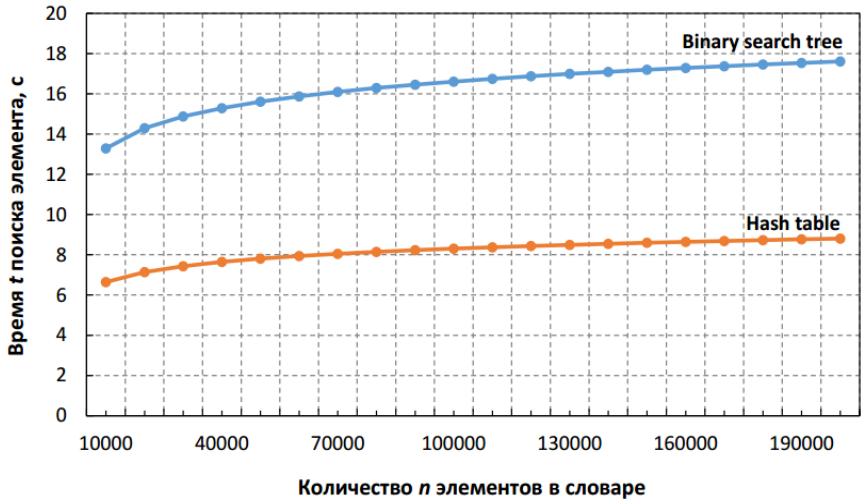

Рис. 1. Зависимость времени *t* поиска элемента в словаре от числа *n* ключей уже вставленных в него

Ниже приведен псевдокод одного из вариантов реализации замеров времени операции поиска ключей в бинарном дереве с *n* элементами.

```
// Можно загрузить слова из файла в массив words[] или связный список
tree = bstree_create(words[0], 0);  // Создаем корень дерева
for (i = 2; i <= 200000; i++) do
    tree = bstree_add(word[i - 1], i - 1)
    if i % 10000 == 0 then
        w = word[getRand(0, i – 1)] // Выбрать случайное слово
        t = wtime()
        node = bstree_lookup(tree, w)
        t = wtime() - t
        print(«n = %d; time = %.6f», i – 1, t)
    end if
end for
```

Эксперимент 2 Сравнение эффективности добавления элементов в бинарное дерево поиска и хеш-таблицу

Требуется заполнить таблицу 2 и построить графики зависимости времени *t* выполнения операции добавления (add) элемента в бинарное дерево поиска и хеш-таблицу от числа *n* элементов уже вставленных в словарь.

Пример оформления графиков приведен на рис. 2 (в архив включен пример Excel-файла для построения графиков такого вида).

Таблица 2

#	Количество элементов в словаре	Время выполнения функции bstree_add, с	Время выполнения функции hashtab_add, с
1	10 000		
2	20 000		
3	30 000		
...			
20	200 000		

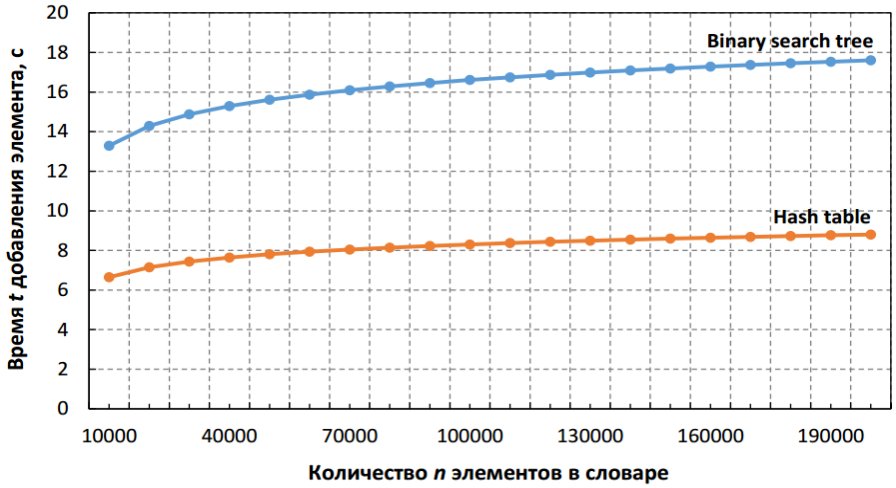

Рис. 2. Зависимость времени *t* добавления элемента в словарь от числа *n* ключей уже вставленных в него

Эксперимент 3 Сравнение эффективности поиска элементов в бинарном дереве поиска и хеш-таблице в худшем случае (worst case)

Требуется заполнить таблицу 3 и построить графики зависимости времени *t* выполнения операции поиска (lookup) элемента в бинарном дереве поиска и хеш-таблице от числа *n* элементов уже вставленных в словарь.

Добавляем в словарь *n* слов - от меньших к большим (например, слова «ааааа», «bbbbb», ...).

В качестве искомого ключа следует выбрать слово, которое вставлено последним.

Таблица 3

#	Количество элементов в словаре	Время выполнения функции bstree_lookup, с	Время выполнения функции hashtab_lookup, с
1	10 000		
2	20 000		
3	30 000		
...			
20	200 000		

Эксперимент 4 Исследование эффективности поиска минимального элемента в бинарном дереве поиска в худшем и среднем случаях

Требуется заполнить таблицу 4 и построить графики зависимости времени *t* выполнения операции поиска минимального элемента в бинарном дереве поиска в худшем и среднем случаях.

Анализ поведения в худшем случае: добавляем в словарь *n* слов - от больших к меньшим (например, слова «zzzzzzzz», «yyyyyy», ...) и замеряем время поиска минимального ключа.

Анализ поведения в среднем случае: добавляем в словарь *n* слов и замеряем время поиска минимального ключа.

Таблица 4

#	Количество элементов в словаре	Время выполнения функции bstree_min в худшем случае, с	Время выполнения функции bstree_min в среднем случае, с
1	10 000		
2	20 000		
3	30 000		
...			
20	200 000		

Эксперимент 5 Исследование эффективности поиска максимального элемента в бинарном дереве поиска в худшем и среднем случаях

Требуется заполнить таблицу 5 и построить графики зависимости времени *t* выполнения операции поиска максимального элемента в бинарном дереве поиска в худшем и среднем случаях.

Анализ поведения в худшем случае: добавляем в словарь *n* слов - от меньших к большим (например, слова «ааааа», «bbbbb», ...) и замеряем время поиска максимального ключа.

Анализ поведения в среднем случае: добавляем в словарь *n* слов и замеряем время поиска максимального ключа.

Таблица 5

#	Количество элементов в словаре	Время выполнения функции bstree_max в худшем случае, с	Время выполнения функции bstree_max в среднем случае, с
1	10 000		
2	20 000		
3	30 000		
...			
20	200 000		

Эксперимент 6 Анализ эффективности хеш-функций

Требуется заполнить таблицу 6 и построить:

- графики зависимости времени *t* выполнения операции поиска элемента в хеш-таблице от числа *n* элементов в ней для заданных хеш-функций X и Y (см. распределение вариантов)
- графики зависимости числа *q* коллизий от количества *n* элементов в хеш-таблице для заданных хеш-функций X и Y (см. распределение вариантов)

Таблица 6

#	Количество элементов в словаре	Хеш-функция X		Хеш-функция Y	
		Время выполнения функции hashtab_lookup, с	Число коллизий	Время выполнения функции hashtab_lookup, с	Число коллизий
1	10 000				
2	20 000				
3	30 000				
...					
20	200 000				

Справочная информация о хеш-функциях (см. распределение вариантов):

- **KP** — хеш-функция из «Практики программирования» (лекция 6, слайд 7)
- **Add** — аддитивная хеш-функция add_hash // http://www.eternallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx
- **XOR** — хеш-функция xor_hash // http://www.eternallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx
- **FNV** — хеш-функция fnv_hash (Fowler/Noll/Vo) // http://www.eternallyconfuzzled.com/tuts/algorithms/jsw_tut_hashing.aspx
- **Jenkins** — хеш-функция jenkins_one_at_a_time_hash // http://en.wikipedia.org/wiki/Jenkins_hash_function
- **ELF** — хеш-функция ELFHash (лекция 6, слайд 17)
- **DJB** — хеш-функция djb_hash (Dan Bernstein) // http://en.wikipedia.org/wiki/Jenkins_hash_function

Контрольные вопросы

- Что такое словарь, ассоциативный массив?
- Что такое бинарное дерево поиска (анализ сложности основных операций)?
- Что такое хеш-таблица (анализ сложности основных операций)?
- Что такое хеш-функция? Что такое «хорошая» хеш-функция?
- Методы разрешения коллизий в хеш-таблицах