Лабораторная работа 4. Бинарные деревья поиска. Хеш-таблицы

Постановка задачи

Требуется реализовать две библиотеки для работы с бинарным деревом поиска (*binary search tree*) и хеш-таблицей (*hash table*). В обоих случаях ключом является строка (*char[], string*), а значением — целое число (*uint32_t*). Размер хеш-таблицы: 20000. Длина слова случайная, от 6 до 30 символов, возможен вариант генерации слов с помощью кода, пример кода для генерации текстового файла со словами на языке *Python* можно найти в приложении. Для более точного измерения времени можно измерить время работы функции в цикле `for`, например, 100000 раз, суммируя результат. Тогда для получения среднего времени полученный результат необходимо разделить на 100000. Также можно измерять количество тактов процессора (`rdtsc`).

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_delete(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)`
- `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)`

Целью работы является проведение экспериментального исследования эффективности бинарных деревьев поиска и хеш-таблиц. Результат выполнения работы – реализованные функции для работы с бинарным деревом поиска и хеш-таблицей, выполненные согласно распределению заданий эксперименты, заполненные таблицы и построенные графики. Распределение заданий по вариантам представлено в таблице 1. В задании 3 возможен вариант «хеш-функция КР, криптографическая хеш-функция». Пример криптографической хеш-функции: SHA256.

Таблица 1. Распределение заданий по вариантам

Вариант	Задание 1	Задание 2	Задание 3
1	Эксперимент 1	Эксперимент 2	Эксперимент 6 – хеш-функции КР, Add
2	Эксперимент 1	Эксперимент 3	Эксперимент 6 – хеш-функции КР, XOR
3	Эксперимент 1	Эксперимент 4	Эксперимент 6 – хеш-функции КР, FNV
4	Эксперимент 1	Эксперимент 5	Эксперимент 6 – хеш-функции КР, Jenkins
5	Эксперимент 1	Эксперимент 2	Эксперимент 6 – хеш-функции КР, ELF
6	Эксперимент 1	Эксперимент 3	Эксперимент 6 – хеш-функции КР, DJB
7	Эксперимент 1	Эксперимент 4	Эксперимент 6 – хеш-функции КР, Add
8	Эксперимент 1	Эксперимент 5	Эксперимент 6 – хеш-функции КР, XOR
9	Эксперимент 1	Эксперимент 2	Эксперимент 6 – хеш-функции КР, FNV
10	Эксперимент 1	Эксперимент 3	Эксперимент 6 – хеш-функции КР, Jenkins
11	Эксперимент 1	Эксперимент 4	Эксперимент 6 – хеш-функции КР, ELF
12	Эксперимент 1	Эксперимент 5	Эксперимент 6 – хеш-функции КР, DJB
13	Эксперимент 1	Эксперимент 2	Эксперимент 6 – хеш-функции КР, Add
14	Эксперимент 1	Эксперимент 3	Эксперимент 6 – хеш-функции КР, XOR
15	Эксперимент 1	Эксперимент 4	Эксперимент 6 – хеш-функции КР, FNV
16	Эксперимент 1	Эксперимент 5	Эксперимент 6 – хеш-функции КР, Jenkins
17	Эксперимент 1	Эксперимент 2	Эксперимент 6 – хеш-функции КР, ELF

18	Эксперимент 1	Эксперимент 3	Эксперимент 6 – хеш-функции KP, DJB
19	Эксперимент 1	Эксперимент 4	Эксперимент 6 – хеш-функции KP, Add
20	Эксперимент 1	Эксперимент 5	Эксперимент 6 – хеш-функции KP, XOR

Экспериментальное исследование

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

Требуется заполнить таблицу 2 и построить графики зависимости времени *t* выполнения операции поиска (*lookup*) элемента в бинарном дереве поиска и хеш-таблице от числа *n* элементов, добавленных в словарь. Пример оформления графиков приведён на рисунке 1.

Для создания набора ключей можно использовать любой текстовый файл с большим числом слов. Скрипт, преобразующий текстовый файл в упорядоченный список уникальных слов, представлен в приложении к заданию. В качестве искомого ключа следует выбирать последнее/первое (*BST/Hash table*) слово, которое уже было добавлено в словарь.

Ниже приведен псевдокод одного из вариантов реализации замеров времени операции поиска ключей в бинарном дереве, состоящем из *n* элементов. Для поиска в словаре последнего/первого слова, уже добавленного туда, можно заранее загрузить слова из исходного текстового файла в массив или связный список.

```
// Можно загрузить слова из файла в массив words[] или связный
список
tree = bstree_create(words[0], 0) // Создаём корень дерева
for i = 2 to 400000 do
    tree = bstree_add(words[i - 1], i - 1)
    if i mod 20000 = 0 then
        w = word[num]
        t = wtime()
        node = bstree_lookup(tree, w)
        t = wtime() - t;
        print("n = %d; time = %.6lf", i - 1, t)
    end if
end for
```

Таблица 2. Результаты эксперимента 1

#	Количество элементов в словаре	Время выполнения функции bstree_lookup, с	Время выполнения функции hashtab_lookup, с
1	20000		
2	40000		
3	60000		
...	...		
20	400000		

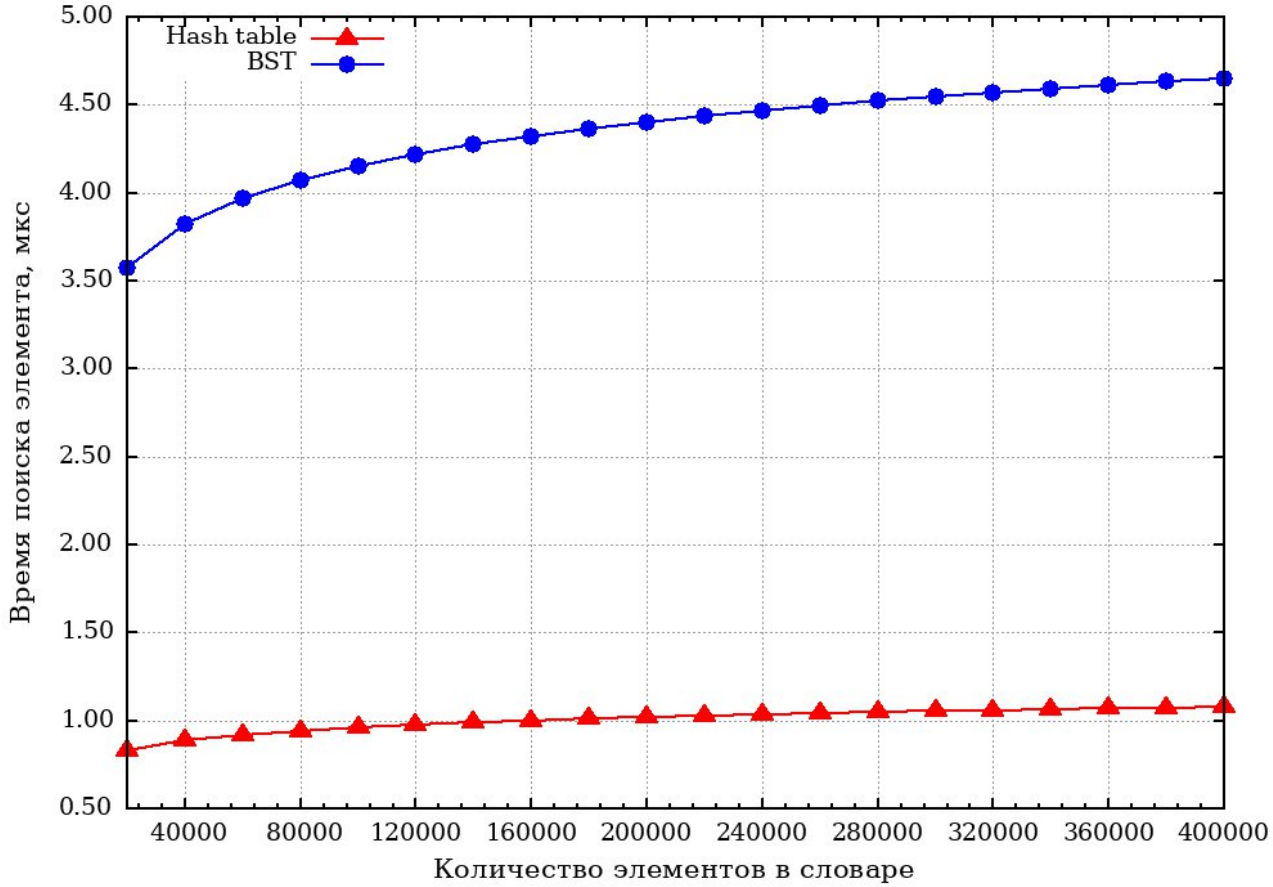

Рисунок 1. Зависимость времени поиска элемента от количества элементов *n* в словаре

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

Требуется заполнить таблицу 3 и построить графики зависимости времени *t* выполнения операции добавления (*add*) элемента в бинарное дерево поиска и хеш-таблицу от числа *n* элементов, добавленных в словарь.

Таблица 3. Результаты эксперимента 2

#	Количество элементов в словаре	Время выполнения функции bstree_add, с	Время выполнения функции hashtab_add, с
1	20000		
2	40000		
3	60000		
...	...		
20	400000		

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

Требуется заполнить таблицу 4 и построить графики зависимости времени *t* выполнения операции поиска (*lookup*) элемента в бинарном дереве поиска и хеш-таблице от числа *n* элементов, уже добавленных в словарь.

Для проведения эксперимента необходимо добавить в словарь *n* слов в порядке неубывания (например, слова «aaaa», «bbbb»). В качестве искомого ключа следует выбрать слово, вставленное последним.

Таблица 4. Результаты эксперимента 3

#	Количество элементов в словаре	Время выполнения функции bstree_lookup, с	Время выполнения функции hashtab_lookup, с
1	20000		
2	40000		
3	60000		
...	...		
20	400000		

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

Требуется заполнить таблицу 5 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: добавить в словарь *n* слов в порядке их невозрастания (например, слова «zzzz», «yyyy», ...), после чего замерить время поиска минимального ключа.

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

Таблица 5. Результаты эксперимента 4

#	Количество элементов в словаре	Время выполнения функции bstree_min в худшем случае, с	Время выполнения функции bstree_min в среднем случае, с
1	20000		
2	40000		
3	60000		
...	...		
20	400000		

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

Требуется заполнить таблицу 6 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: добавить в словарь *n* слов в порядке их неубывания (например, слова «aaaa», «bbbb», ...), затем замерить время поиска максимального ключа.

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

Таблица 6. Результаты эксперимента 5

#	Количество элементов в словаре	Время выполнения функции bstree_max в худшем случае, с	Время выполнения функции bstree_max в среднем случае, с
1	20000		
2	40000		
3	60000		
...	...		
20	400000		

Эксперимент 6. Анализ эффективности различных хеш-функций

Требуется заполнить таблицу 7 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*

Таблица 7. Результаты эксперимента 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	20000				
2	40000				
3	60000				
...	...				
20	400000				

Справочная информация и реализации на языке С хеш-функций *KP*, *Add* и *ELF* приведены в приложении к заданию. Информацию об остальных представленных в лабораторной работе функциях хеширования (*XOR, FNV, Jenkins, DJB*) можно найти в открытых источниках.

Контрольные вопросы

- Что такое словарь, ассоциативный массив?
- Что такое бинарное дерево поиска? Проведите анализ сложности основных операций.
- Что такое хеш-таблица? Проведите анализ сложности основных операций.
- Что такое хеш-функция? Какая хеш-функция является «хорошей»?
- Методы разрешения коллизий в хеш-таблицах.

Приложение

Генерация текстового файла, Python

```
import random
import time

def main():
    print("\n============================\n"
          "This program generate <n> random words with generator's seed <seed>\n"
          "from <min lenght> to <max length>, from <min ascii> to <max ascii>\n"
          "and write it into <filename>;\n"
          "if generator's seed equals to 0, then seed is current time\n"
          "============================\n")

    n, min_length, max_length, min_ascii, max_ascii, seed = \
        map(int, input("Input:\n<n> <min length> <max length> <min ascii> <max ascii>
<seed>\n").split())

    if (max_length < min_length or max_ascii < min_ascii):
        print("<max lenght> should be less than or equal to <min lenght>\n"
              "<max ascii> should be less than or equal to <min ascii>\n")
        exit(1)
    if (max_length < 1 or min_length < 1 or n < 1 or min_ascii < 32 or max_ascii < 32
         or min_ascii > 126 or max_ascii > 126):
        print("all numbers should be greather than 0;\n"
              "<max ascii> and <min ascii> should be less than 126 and greather than
31;\n")
        exit(1)
    if (seed == 0):
        random.seed(time.time())
    else:
        random.seed(seed)

    if (n > 5000000):
        print("n > 5000000, exit\n")
        exit(1)
    elif (n >= 1000000):
        res = input("n = " + str(n) + ", press <Y> to continue\n")
        if (res != "Y"):
            exit(1)

    filename = input("Input <filename>:\n")

    file = open(filename, 'w')
    for i in range(0, n):
        word_len = random.randint(min_length, max_length)
        word = ""
        for j in range(0, word_len):
            word += chr(random.randint(min_ascii, max_ascii))
        word += '\n'
        file.write(word)
    file.close()
    print("Done\n")
main()
```

Преобразование текстового файла в упорядоченный список уникальных слов

Приведённый ниже скрипт командной оболочки позволяет преобразовать текстовый файл в отсортированный по неубыванию список уникальных слов. Использование скрипта: split_by_words.sh <название текстового файла>

```
#!/bin/sh
INFILE=$1
MINCHARS=3
#
# Выводим файл $INFILE | разбиваем поток строк на слова | удаляем из потока
# слова с длиной <= $MINCHARS | преобразуем слова потока в нижний регистр |
# сортируем слова | удаляем повторяющиеся слова
#
cat $INFILE | tr -s '[[:punct:][:space:]]' '\n' | grep -E ".{$MINCHARS}" | \
sed 's/[[:upper:]]*/\L&/' | sort | uniq
```

Аддитивная хеш-функция

Данная хеш-функция является одним из простейших алгоритмов хеширования. Так как в основе её лежит коммутативная операция сложения, такая функция не будет являться эффективной: например, слова «abcd», «cabd» и «cdba» будут обрабатываться аддитивной хеш-функцией одинаково и возвращать одно и то же значение.

Хеш-функция AddHash представлена в учебных целях и не применяется на практике. Здесь и далее HASH_SIZE — количество ячеек в хеш-таблице.

```
unsigned int AddHash(char *s)
{
    unsigned int h = 0;
    while (*s)
        h += (unsigned int)*s++;
    return h % HASH_SIZE;
}
```

Хеш-функция Кернигана–Ричи (KP, BKDR)

Хеш-функция Брайана Кернигана и Денниса Ричи для строкового типа данных из книги «Язык программирования C». Также известна как хеш-функция BKDR (Brian Kernighan, Dennis Ritchie), по принципу работы схожа с функцией DJB.

При реализации хеш-функции KP допускается задание различных значений множителя hash_mul. Как правило, эти значения содержат числа с повторяющимся шаблоном «31» (31, 131, 1313, 13131, ...).

```
unsigned int KRHash(char *s)
{
    unsigned int h = 0, hash_mul = 31;
    while (*s)
        h = h * hash_mul + (unsigned int)*s++;
    return h % HASH_SIZE;
}
```

Хеш-функция ELFHash

Хеш-функция ELFHash широко используется в файлах формата ELF в UNIX-подобных операционных системах. Является вариацией некриптографической хеш-функции PJW.

```
unsigned int ELFHash(char *s)
{
    unsigned int h = 0, g;
    while (*s) {
        h = (h << 4) + (unsigned int)*s++;
        g = h & 0xF0000000L;
        if (g)
            h ^= g >> 24;
        h &= ~g;
    }
    return h % HASH_SIZE;
}
```