ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ  
  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ   
БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ   
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»  
  
ФАКУЛЬТЕТ ИНФОРМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Кафедра вычислительных систем

**ОТЧЕТ  
ПО ЛАБОРАТОРНОЙ РАБОТЕ №2**

Выполнил:  
студент группы ИВ-521  
Прокопенко Р. П.  
  
Проверил:  
доцент кафедры вычислительных систем  
Курносов М. Г.

Оценка – «\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_».

Новосибирск - 2016

**Вариант 12**

**Постановка задачи**

Требуется реализовать на языке C две библиотеки для работы с бинарным деревом поиска (Binary search tree) и хеш-таблицей (Hash table). Провести эксперименты 1 и 5 для модуля с бинарными деревьями поиска и эксперимент 6 (хеш-функции KP, Add) для модуля с хеш-таблицами, в соответствии с вариантом.

**Описание алгоритмов**

1. Бинарное дерево поиска

Бинарное дерево поиска (англ. *binary search tree*, BST) — это двоичное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

* Оба поддерева — левое и правое — являются двоичными деревьями поиска.
* У всех узлов левого поддерева произвольного узла X значения ключей данных меньше, нежели значение ключа данных самого узла X.
* В то время, как значения ключей данных у всех узлов правого поддерева (того же узла X) больше, нежели значение ключа данных узла X.

**Поиск элемента (lookup)**

Дано: дерево Т и ключ K.

Задача: проверить, есть ли узел с ключом K в дереве Т, и если да, то вернуть ссылку на этот узел.

Алгоритм:

* Если дерево пусто, сообщить, что узел не найден, и остановиться.
* Иначе сравнить K со значением ключа корневого узла X.
* Если K=X, выдать ссылку на этот узел и остановиться.
* Если K>X, рекурсивно искать ключ K в правом поддереве Т.
* Если K<X, рекурсивно искать ключ K в левом поддереве Т.

Вычислительная сложность в худшем случае *O*(*n*), где *n* — это количество ключей

Вычислительная сложность в среднем случае *O*(log *n*).

Сложность по памяти O(*n*).

**Добавление элемента (add)**

Дано: дерево Т и пара (K, V).

Задача: вставить пару (K, V) в дерево Т (при совпадении K, заменить V).

Алгоритм:

* Если дерево пусто, заменить его на дерево с одним корневым узлом ((K, V), null, null) и остановиться.
* Иначе сравнить K с ключом корневого узла X.
* Если K>X, циклически добавить (K, V) в правое поддерево Т.
* Если K<X, циклически добавить (K, V) в левое поддерево Т.
* Если K=X, заменить V текущего узла новым значением (хотя можно и организовать список значений V, но это другая тема).

Вычислительная сложность в худшем случае *O*(*n*), где *n* — это количество ключей

Вычислительная сложность в среднем случае *O*(log *n*).

Сложность по памяти O(n).

2. Хеш-табли́ца

Хеш-табли́ца (англ. *Hash table*) — это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Получающееся хеш-значение *i* = hash(*key*) играет роль индекса в массиве H. Затем выполняемая операция (добавление, удаление или поиск) перенаправляется объекту, который хранится в соответствующей ячейке массива H[*i*].

Каждая ячейка массива H является указателем на связный список (цепочку) пар ключ-значение, соответствующих одному и тому же хеш-значению ключа. Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента.

**Поиск элемента (lookup)**

Дано: хеш-таблица H и ключ K.

Задача: проверить, есть ли узел с ключом K в хеш-таблице H, и если да, то вернуть ссылку на этот узел.

Алгоритм:

* Вычислить хеш от ключа K: *i* = hash(*K*).
* Обратиться к элементу массива с индексом *i.*
* Выполнить поиск в связном списке по ключу K.

Вычислительная сложность в худшем случае *O*(*n*), где *n* — это количество ключей

Вычислительная сложность в среднем случае *O*(1 + *n/h*).

Сложность по памяти O(n).

**Добавление элемента (add)**

Дано: хеш-таблица H и пара (K, V).

Задача: вставить пару (K, V) в хеш-таблицу H.

Алгоритм:

* Вычислить хеш от ключа K: i = hash(K).
* Обратиться к элементу массива с индексом *i*.
* Добавить элемент (K, V) в начало связного списка.

Вычислительная сложность в худшем случае *O*(1), где *n* — это количество ключей

Вычислительная сложность в среднем случае *O*(1).

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

**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 «aaaaa», «bbbbb», ...) и замеряем время поиска максимального ключа.

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

**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)

**Организация экспериментов**

Эксперименты проводились на ноутбуке Samsung RC530

(CPU: Intel Core i5-2430M, RAM 6GB);

Операционная система Windows 7 Домашняя базовая;

Среда разработки Dev C++ v5.4.2 (компилятор gcc 4.7.1 64-bit)

Ключи компиляции программы: -Wall -o

**Результаты экспериментов**

Эксперимент 1

Таблица 1

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Количество элементов в словаре** | **Время выполнения функции**  **bstree\_lookup, с** | **Время выполнения функции**  **hashtab\_lookup, с** |
| 1 | 10 000 | 0.00000344 | 0.00000087 |
| 2 | 20 000 | 0.00000349 | 0.00000097 |
| 3 | 30 000 | 0.00000363 | 0.00000103 |
| 4 | 40 000 | 0.00000415 | 0.00000111 |
| 5 | 50 000 | 0.00000432 | 0.00000135 |
| 6 | 60 000 | 0.00000434 | 0.00000141 |
| 7 | 70 000 | 0.00000437 | 0.00000157 |
| 8 | 80 000 | 0.00000454 | 0.00000161 |
| 9 | 90 000 | 0.00000466 | 0.00000164 |
| 10 | 100 000 | 0.00000517 | 0.00000168 |
| 11 | 110 000 | 0.00000493 | 0.00000172 |
| 12 | 120 000 | 0.00000531 | 0.00000181 |
| 13 | 130 000 | 0.00000544 | 0.00000187 |
| 14 | 140 000 | 0.00000539 | 0.00000203 |
| 15 | 150 000 | 0.00000558 | 0.00000255 |
| 16 | 160 000 | 0.00000583 | 0.00000252 |
| 17 | 170 000 | 0.00000589 | 0.00000260 |
| 18 | 180 000 | 0.00000577 | 0.00000277 |
| 19 | 190 000 | 0.00000585 | 0.00000289 |
| 20 | 200000 | 0.00000595 | 0.00000300 |

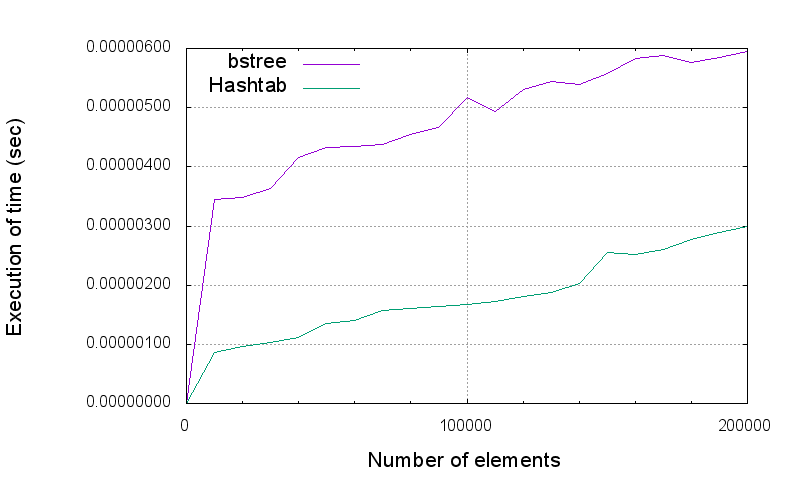


Рисунок - Сравнение эффективности поиска элементов в бинарном девере поиска и хеш-таблице в среднем случае (average case)

Эксперимент 5

Таблица 2

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **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 | 0.00000390 | 0.00000090 |
| 2 | 20 000 | 0.00000395 | 0.00000097 |
| 3 | 30 000 | 0.00000398 | 0.00000104 |
| 4 | 40 000 | 0.00000417 | 0.00000113 |
| 5 | 50 000 | 0.00000431 | 0.00000138 |
| 6 | 60 000 | 0.00000439 | 0.00000143 |
| 7 | 70 000 | 0.00000450 | 0.00000161 |
| 8 | 80 000 | 0.00000494 | 0.00000162 |
| 9 | 90 000 | 0.00000503 | 0.00000164 |
| 10 | 100 000 | 0.00000517 | 0.00000168 |
| 11 | 110 000 | 0.00000525 | 0.00000177 |
| 12 | 120 000 | 0.00000541 | 0.00000187 |
| 13 | 130 000 | 0.00000564 | 0.00000189 |
| 14 | 140 000 | 0.00000581 | 0.00000201 |
| 15 | 150 000 | 0.00000598 | 0.00000252 |
| 16 | 160 000 | 0.00000608 | 0.00000275 |
| 17 | 170 000 | 0.00000628 | 0.00000276 |
| 18 | 180 000 | 0.00000644 | 0.00000287 |
| 19 | 190 000 | 0.00000683 | 0.00000299 |
| 20 | 200 000 | 0.00000695 | 0.00000306 |

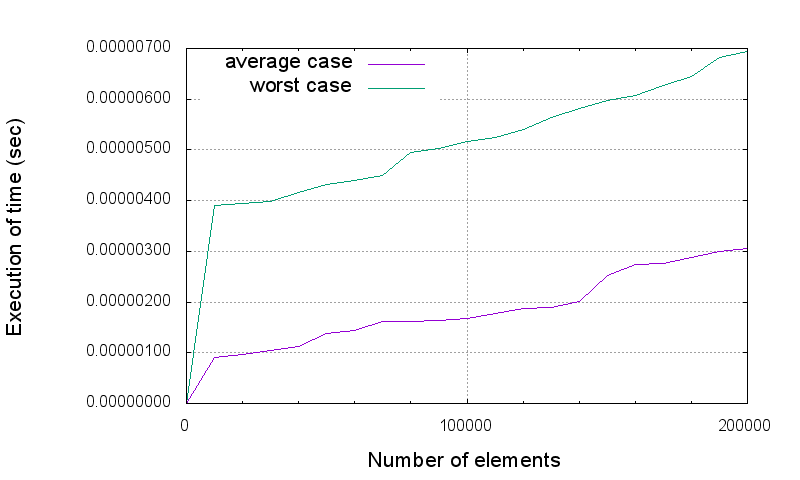


Рисунок - Исследование эффективности поиска максимального элемента в бинарном дереве поиска в худшем и среднем случаях

Эксперимент 6

Таблица 3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **#** | **Количество элементов в словаре** | **Хеш-функция** **KP** | | **Хеш-функция Add** | |
| **Время выполнения**  **функции**  **hashtab\_lookup, с** | **Число**  **коллизий** | **Время выполнения**  **функции**  **hashtab\_lookup, с** | **Число**  **коллизий** |
| **1** | **10 000** | **0.00000093** | **3018** | **0.00000160** | **3240** |
| **2** | **20 000** | **0.00000098** | **9757** | **0.00000170** | **9821** |
| **3** | **30 000** | **0.00000105** | **18231** | **0.00000172** | **18340** |
| **4** | **40 000** | **0.00000112** | **27558** | **0.00000179** | **27670** |
| **5** | **50 000** | **0.00000140** | **37235** | **0.00000226** | **37228** |
| **6** | **60 000** | **0.00000142** | **47104** | **0.00000217** | **47100** |
| **7** | **70 000** | **0.00000157** | **57036** | **0.00000220** | **57075** |
| **8** | **80 000** | **0.00000161** | **67004** | **0.00000229** | **67030** |
| **9** | **90 000** | **0.00000164** | **76988** | **0.00000242** | **77046** |
| **10** | **100 000** | **0.00000169** | **86981** | **0.00000246** | **86999** |
| **11** | **110 000** | **0.00000175** | **96976** | **0.00000253** | **96998** |
| **12** | **120 000** | **0.00000188** | **106974** | **0.00000279** | **107975** |
| **13** | **130 000** | **0.00000189** | **116974** | **0.00000295** | **116973** |
| **14** | **140 000** | **0.00000205** | **126974** | **0.00000297** | **127973** |
| **15** | **150 000** | **0.00000275** | **136973** | **0.00000325** | **136973** |
| **16** | **160 000** | **0.00000262** | **146973** | **0.00000351** | **147973** |
| **17** | **170 000** | **0.00000256** | **156973** | **0.00000355** | **156973** |
| **18** | **180 000** | **0.00000287** | **166973** | **0.00000365** | **167973** |
| **19** | **190 000** | **0.00000299** | **176973** | **0.00000363** | **176973** |
| **20** | **200 000** | **0.00000305** | **186973** | **0.00000369** | **187973** |

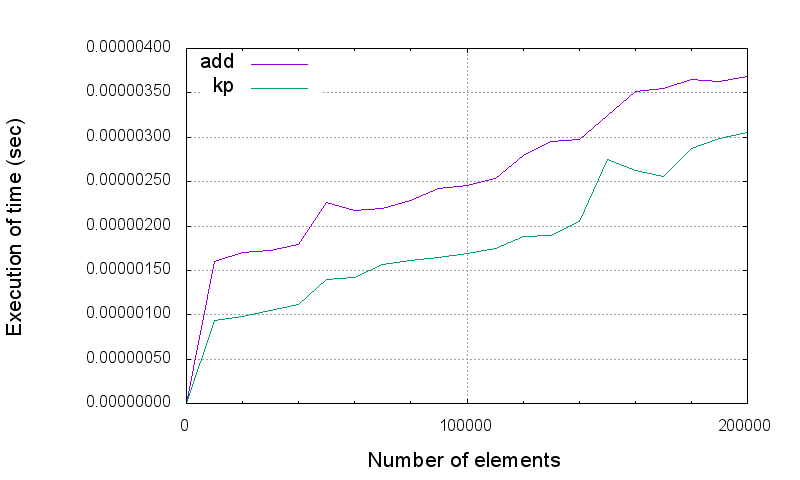


Рисунок 3 - Сравнение эффективности хеш-функций по времени выполнения

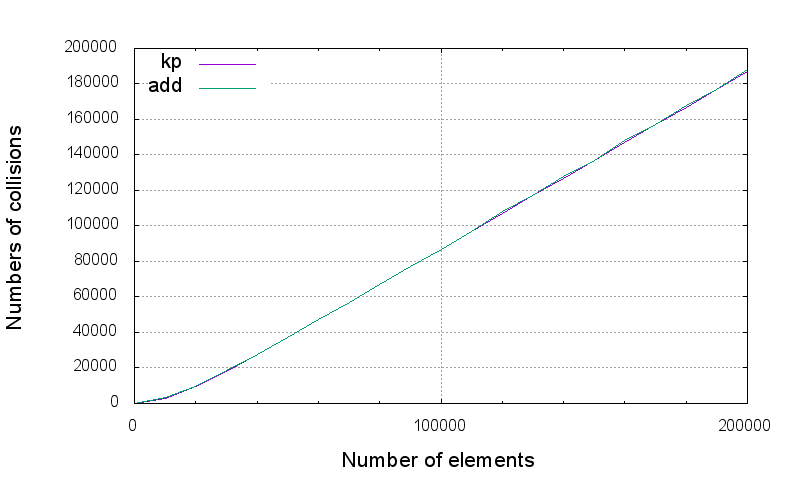


Рисунок 4 - Сравнение эффективности хеш-функций по числу коллизий

**Выводы**

Из результатов экспериментов можно сделать следующие выводы:

1) Сложность добавления элементов в бинарное дерево поиска *Θ*(log *n) –* средний случай, и *O*(*n*)в худшей. Поиск имеет аналогичную сложность с добавлением. Если использовать данную реализацию словаря, то необходимо позаботиться о том, чтобы бинарное дерево поиска было более менее сбалансированным, или чтобы добавление элементов было не по возрастанию(убыванию).

2) Сложность добавления элементов в хеш-таблицу *O*(1) – так как добавление происходит в начало списка. Поиск же происходит со сложностью *O*((*n* + *m*), где *n –* кол-во элементов, а *m* – размер таблицы. Если необходимо только хранить и добавлять элементы, то для увеличения эффективности хеш-таблицы – нужно позаботиться о наименьшем кол-во коллизий, то есть использовать “хорошую” хеш–функцию.

3) В целом, хеш-функция Add работает хуже чем функция KP.

**Ссылки**

1. <https://ru.wikipedia.org/wiki/Хеш-таблица>

2. <https://ru.wikipedia.org/wiki/Двоичное_дерево_поиска>

3. http://www.eternallyconfuzzled.com/tuts/algorithms/jsw\_tut\_hashing.aspx