Адрианов Н.М.
Иванов А.Б.

Алгоритмы и структуры данных

B-деревья. Хеш-таблицы.

B-деревья

(Bayer-McCreight, 1972)

- Данные читаются поблочно (постранично)
- Время чтения блока относительно велико

Применение:

- Файловые системы:
  NTFS (Windows), HFS, HFS+ (Mac),
  ReiserFS, XFS, Ext3FS, JFS (Linux)
- Базы данных:
  Oracle, MS SQL, DB2, PostgreSQL

B-деревья

B-деревья обобщают 2-3 деревья: максимально разрешается хранить до $M - 1$ ключа в вершине

- Во всех вершинах кроме корневой – не менее $M/2$ ключей
- Внешние узлы хранят ключи (+данные)
- Внутренние узлы хранят ключи для обеспечения поиска

B-деревья

Для поиска или вставки в дерево с N ключами требуется количество чтений между $\log_{M-1} N$ и $\log_{M/2} N$.

На практике: не более 4 ($M = 1024 \Rightarrow N = 64 \cdot 10^9$)

Хинт: всегда держать корневую страницу загруженной в память.

Хеш-таблица (Hash Table)

Структура данных, реализующая ассоциативный массив.

Соглашение по сложности (контракт):

- Вставка: $O(1)^*$
- Поиск по ключу: $O(1)^*$
- Удаление по ключу: $O(1)^*$

Память - $O(n)$

Применение:

- Анализ потока данных (уникальные пользователи, etc)
- Сетевое оборудование (firewall, балансировщик, IPS)
- 3-SUM (vs Subset sum)
- ...

Хеш-функция

$$h : U \to \{0, \ldots, m-1\}$$

Коллизия:

$$x, y \in U : h(x) = h(y)$$

Хеш-функция

Парадокс дней рождения. В группе, состоящей из 23 или более человек, вероятность совпадения дней рождения (число и месяц) хотя бы у двух людей превышает 50%.

U – группа людей, $h : U \to \{1, \ldots 365\}$.

`https://en.wikipedia.org/wiki/Birthday_problem`
Количество n необходимое для получения коллизий с вероятностью p:

$$n(p, m) \approx \sqrt{2m \cdot \ln\left(\frac{1}{1-p}\right)}$$

Пример реализации в Java (когда-то)

```
/** Cache the hash code for the string */
private int hash; // Default to 0

public int hashCode(char[] value) {
    int h = hash;
    if (h == 0 && value.length > 0) {
        for (int i = 0; i < value.length; i++) {
            h = 31 * h + value[i];
        }
        hash = h;
    }
    return h;
}
```

Пример реализации в Java: почему 31?

Joshua Bloch “Effective Java”

- `Always override hashcode when you override equals`

The value 31 was chosen because it is an odd prime. If it were even and the multiplication overflowed, information would be lost, as multiplication by 2 is equivalent to shifting. The advantage of using a prime is less clear, but it is traditional. A nice property of 31 is that the multiplication can be replaced by a shift and a subtraction for better performance: `31 * i == (i << 5) - i`. Modern VMs do this sort of optimization automatically.

Реализация: метод цепочек (chaining)

- Массив размера m
- В каждой ячейке храним список (key, value)

```
buckets = array of length m
function find(key):
  h = hash(key)
  node = buckets[h] // linked list
  while node != null:
    if node.key == key:
      return node.value
    node = node.next
  return null
end
```

Реализация: Открытая адресация

- Массив размера m
- В каждой ячейке храним (key, value) (не список!)
- Приведена одна из возможных реализаций открытой адресации (линейное исследование)
  probing = опробывание, исследование

```
buckets = array of length m
function find(key):
  h = hash(key)
  node = buckets[h] // array element
  if node.key == key:
    return node.value
  i = (h + 1) % m
  while node != null && i != h:
    if node.key == key:
      return node.data
  return null
end
```

Варианты открытой адресации

- Линейное исследование:

$$h(i, x) = h_1(x) + i \pmod m$$

- Квадратичное исследование:

$$h(i, x) = h_1(x) + c_2 i^2 + c_1 i \pmod m$$

Например:

$$h(i, x) = h_1(x) + 1 + 2 + \ldots + i = h_1(x) + i(i + 1)/2$$

- Двойное хэширование:

$$h(i, x) = h_1(x) + i h_2(x) \pmod m$$

- Хэширование Робин Гуда ("Robin Hood hashing")
- Кукушкино хеширование (Cuckoo hashing)

Сложность: метод цепочек

Утверждение: Математическое ожидание сложности неудачного поиска при условии простого равномерного хеширования равна $\Theta(1+\alpha)$, где α - коэффициент заполнения таблицы. ($\alpha = \frac{n}{m}$, где m - количество ячеек)

Сложность: открытая адресация

Утверждение: Математическое ожидание сложности неудачного поиска при условии простого равномерного хеширования равна $O(\frac{1}{1-\alpha})$, где $\alpha < 1$ - коэффициент заполнения таблицы. ($\alpha = \frac{n}{m}$, где m - количество ячеек)

Сложность: открытая адресация

Утверждение: Математическое ожидание сложности неудачного поиска при условии простого равномерного хеширования равна $O(\frac{1}{1-\alpha})$, где $\alpha < 1$ - коэффициент заполнения таблицы. ($\alpha = \frac{n}{m}$, где m - количество ячеек)
Если X =количество проверок при неуспешном поиске, то

$$E[X] = \sum_{i=1}^{\infty} P\{X \geq i\} \leq \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

т.к.

$$\sum_{i=0}^{\infty} iP\{X = i\} = \sum_{i=0}^{\infty} i(P\{X \geq i\} - P\{X \geq i+1\}) = \sum_{i=0}^{\infty} P\{X \geq i\}$$

Открытая адресация VS метод цепочек

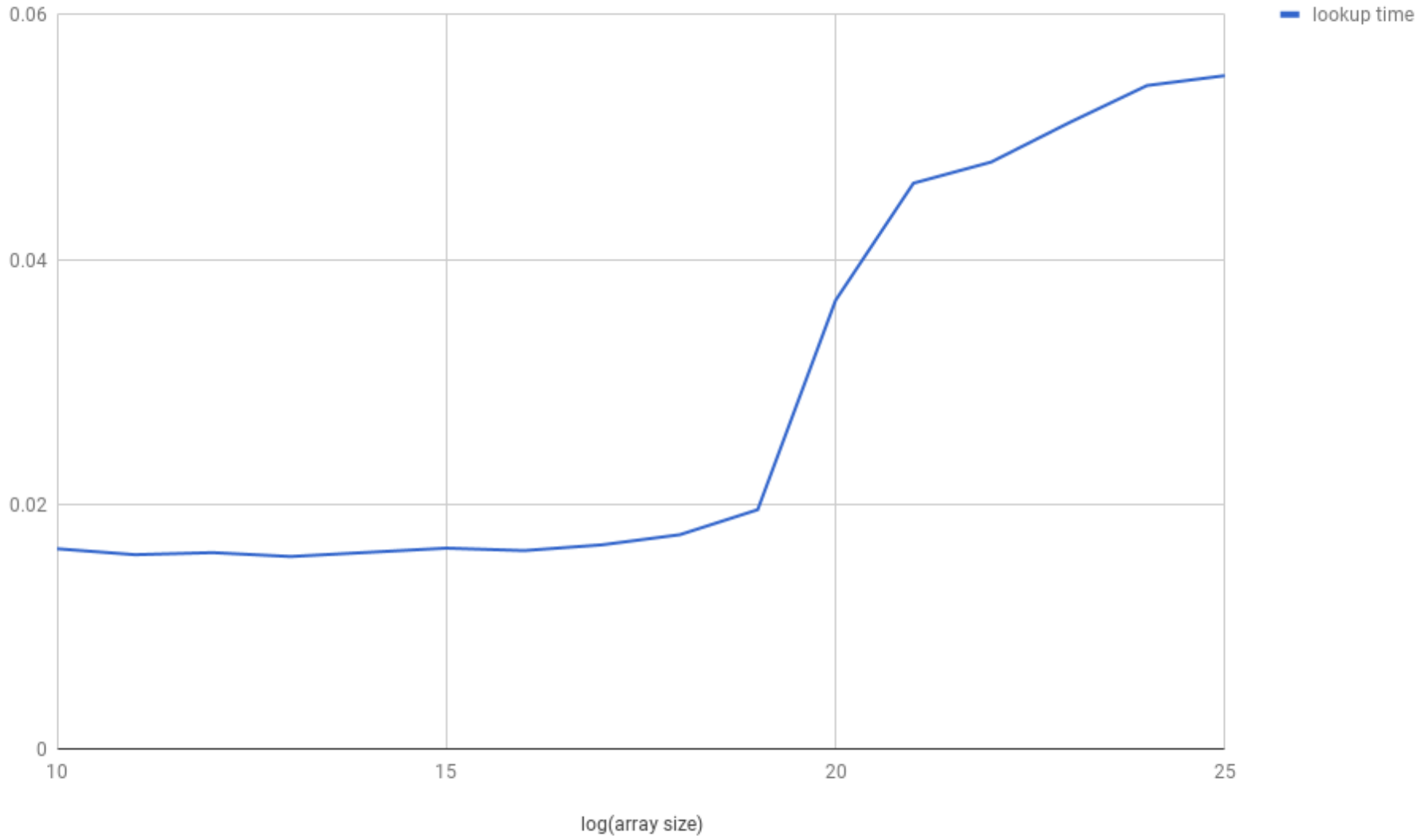
Недостатки:

- Сложность реализации (относительная)
- Сложность удаления
- Большая чувствительность к коэффициенту заполнения

Преимущества:

- Локальность!!!

Локальность



Кукушкино хеширование

- Используются 2 массива A_1, A_2 каждый размером $\frac{m}{2}$
- 2 хеш функции. $h_1, h_2 : U \to \left\{0, \ldots, \frac{m}{2} - 1\right\}$
- Каждый ключ $x \in U$ Может находится в одном из двух мест: в ячейке $h_1(x)$ в первом массиве или в ячейке $h_2(x)$ во втором массиве
- Если вставляем ключ x и место $h_1(x)$ занято ключом y, то "выкидываем"y, кладем x, а y перемещаем в $h_2(y)$ и далее по цепочке
- Если чепочка перемещений длиннее порого, выбираем новые хэш функции и перестраиваем всю таблицу
- Сложность поиска и удаления в худшем случае - $O(1)$
- Сложность вставки при достаточно низком коэффициенте заполнения в среднем - $O(1)$