



Алгоритмы и структуры данных

Лекция 6. Таблицы.

(с) Глухих Михаил Игоревич, glukhikh@mail.ru

Таблица

- Элемент = Ключ + Данные
- Ключи у разных элементов не совпадают
- Операции
 - Search (Key)
 - Insert (Key, Value)
 - Remove (Key)

Виды таблиц

- ▶ С прямой адресацией

Таблицы с прямой адресацией

- Ключ = Целое неотрицательное число или может быть к нему преобразован (взаимно-однозначно)
- Ключи не слишком большие

Таблицы с прямой адресацией

- Ключ = Целое неотрицательное число или может быть к нему преобразован (взаимно-однозначно)
- Ключи не слишком большие
- => Используем ключ как индекс массива, а данные храним в этом массиве
- Достоинства, Недостатки?

Таблицы с прямой адресацией

- Ключ = Целое неотрицательное число или может быть к нему преобразован (взаимно-однозначно)
- Ключи не слишком большие
- => Используем ключ как индекс массива, а данные храним в этом массиве
- Достоинства: трудоёмкость всюду $O(1)$
- Недостатки?

Таблицы с прямой адресацией

- Ключ = Целое неотрицательное число или может быть к нему преобразован (взаимно-однозначно)
- Ключи не слишком большие
- => Используем ключ как индекс массива, а данные храним в этом массиве
- Достоинства: трудоёмкость всюду $O(1)$
- Недостатки: ресурсоёмкость $O(\text{MaxKey})$

Виды таблиц

- С прямой адресацией
- Хэш-таблицы
 - Используем для индексации $\text{hash}(\text{key})$ или $\text{hash}(\text{key}) \% 2^N$

Виды таблиц

- С прямой адресацией
- Хэш-таблицы
 - Используем для индексации $\text{hash}(\text{key})$ или $\text{hash}(\text{key}) \% 2^N$
 - В отличие от случая с прямой адресацией, здесь не требуется взаимной однозначности
 - Можем управлять размером нашего массива
 - НО! Коллизии

Виды таблиц

- С прямой адресацией
- Хэш-таблицы: как бороться с коллизиями?

Виды таблиц

- С прямой адресацией
- Хэш-таблицы: как бороться с коллизиями?
 - С помощью цепочек
 - Каждому значению кода ставится в соответствие не одна пара Ключ + Значение, а несколько, объединённых в связанный список
 - Достоинства = Размерность исходного массива ограничивается
 - Недостатки = ?

Виды таблиц

- С прямой адресацией
- Хэш-таблицы: как бороться с коллизиями?
 - С помощью цепочек
 - Каждому значению кода ставится в соответствие не одна пара Ключ + Значение, а несколько, объединённых в связанный список
 - Достоинства = Размерность исходного массива ограничивается
 - Недостатки = При фиксированной размерности массива трудоёмкость операций $O(N)$ – в общем случае $O(1 + N / m)$, где m – размерность массива
 - Как с этим бороться?

Виды таблиц

- С прямой адресацией
- Хэш-таблицы: как бороться с коллизиями?
 - С помощью цепочек
 - Каждому значению кода ставится в соответствие не одна пара Ключ + Значение, а несколько, объединённых в связанный список
 - Достоинства = Размерность исходного массива ограничивается
 - Недостатки = При фиксированной размерности массива трудоёмкость операций $O(N)$ -- в общем случае $O(1 + N / m)$, где m – размерность массива
 - Как с этим бороться? Понятно как – изменять размерность массива динамически

Таблица с открытой адресацией

- Нет никаких цепочек и связанных списков
- Всё хранится в едином массиве
- Если есть возможность, хэш-код (по модулю) используем как индекс в массиве
- Если возникает коллизия, то сдвигаем

HASH-INSERT(array[m], key)

- `i = hash(key) % m`
- `start = i`
- `do:`
 - `if (array[i] == null):`
 - `array[i] = key`
 - `return i`
 - `i = (i + 1) % m`
- `while (i != start)`
- `Error("Overflow")`

HASH-INSERT: пример

- Используем 3 бита хэша и массив из 8 элементов
- В качестве ключа используем символ

NN NN NN NN NN NN NN NN

HASH-INSERT: пример

- Используем 3 бита хэша и массив из 8 элементов
- В качестве ключа используем символ

```
NN NN NN NN NN NN NN NN  
NN NN 2A NN NN 5B NN 7C
```

HASH-INSERT: пример

- Используем 3 бита хэша и массив из 8 элементов
- В качестве ключа используем символ

NN NN NN NN NN NN NN NN

NN NN 2A NN NN 5B NN 7C

NN NN 2A 2D NN 5B NN 7C

HASH-SEARCH(array[m], key)

- `i = hash(key) % m`
- `start = i`
- `do:`
 - `if (array[i] == key):`
 - `return i`
 - `if (array[i] == null):`
 - `return null`
 - `i = (i + 1) % m`
- `while (i != start)`
- `return null`

HASH-DELETE

- Реализация «в лоб» не проходит

NN NN NN NN NN NN NN NN

NN NN 2A NN NN 5B NN 7C

NN NN 2A 2D NN 5B NN 7C

HASH-DELETE

- Реализация «в лоб» не проходит

NN NN NN NN NN NN NN NN

NN NN 2A NN NN 5B NN 7C

NN NN 2A 2D NN 5B NN 7C

// Теперь стираем 2A

NN NN NN 2D NN 5B NN 7C

HASH-DELETE

- Реализация «в лоб» не проходит

NN NN NN NN NN NN NN NN

NN NN 2A NN NN 5B NN 7C

NN NN 2A 2D NN 5B NN 7C

// Теперь стираем 2A

NN NN NN 2D NN 5B NN 7C

// И ищем 2D – NOT FOUND!!!

HASH-DELETE

- А как «не в лоб»? Отдельный вариант “DD” -- DELETED

NN NN NN NN NN NN NN NN

NN NN 2A NN NN 5B NN 7C

NN NN 2A 2D NN 5B NN 7C

// Теперь стираем 2A

NN NN DD 2D NN 5B NN 7C

HASH-DELETE

- А как «не в лоб»?
Отдельный вариант “DD” – DELETED
- И при поиске DD не считается null

NN NN NN NN NN NN NN NN

NN NN 2A NN NN 5B NN 7C

NN NN 2A 2D NN 5B NN 7C

// Теперь стираем 2A

NN NN DD 2D NN 5B NN 7C

Таблица с открытой адресацией

- ▶ Трудоёмкость операций зависит от коэффициента заполнения $A = n / m$ (n = число элементов, m = размерность массива)

Таблица с открытой адресацией

- ▶ Трудоёмкость операций зависит от коэффициента заполнения $A = n / m$ (n = число элементов, m = размерность массива)
- ▶ И равна $\sim 1 / (1 - A)$ для большинства операций

Таблица с открытой адресацией

- ▶ Трудоёмкость операций зависит от коэффициента заполнения $A = n / m$ (n = число элементов, m = размерность массива)
- ▶ И равна $\sim 1 / (1 - A)$ для большинства операций
- ▶ Например, при таблице, заполненной на четверть, мы имеем трудоёмкость $4 / 3$

Таблица с открытой адресацией

- ▶ Трудоёмкость операций зависит от коэффициента заполнения $A = n / m$ (n = число элементов, m = размерность массива)
- ▶ И равна $\sim 1 / (1 - A)$ для большинства операций
- ▶ Например, при таблице, заполненной на четверть, мы имеем трудоёмкость $4 / 3$
- ▶ А наполовину – уже 2

Таблица с открытой адресацией

- ▶ Трудоёмкость операций зависит от коэффициента заполнения $A = n / m$ (n = число элементов, m = размерность массива)
- ▶ И равна $\sim 1 / (1 - A)$ для большинства операций
- ▶ Например, при таблице, заполненной на четверть, мы имеем трудоёмкость $4 / 3$
- ▶ А наполовину – уже 2
- ▶ А на три четверти – уже 4
- ▶ И наконец, если заполнены все элементы, кроме одного, то требуется m операций

Таблица с открытой адресацией

- ▶ Последовательности проб: как мы переходим от ячейки к ячейке, если не нашли то, что нужно, в ячейке, соответствующей хэш-коду

Таблица с открытой адресацией

- ▶ Последовательности проб
 - ▶ $H = \text{hashCode} \% N$, $N = \text{размер таблицы}$
 - ▶ Линейная: $H, H+1, H+2, \dots, N-1, 0, 1, 2, \dots, H-1$
 - ▶ Рассмотренный нами вариант

Таблица с открытой адресацией

- ▶ Последовательности проб
 - ▶ $H = \text{hashCode} \% N$, $N = \text{размер таблицы}$
 - ▶ Линейная: $H, H+1, H+2, \dots, N-1, 0, 1, 2, \dots, H-1$
 - ▶ Двойное хэширование: $H, H+s, H+2s, \dots (\% N)$,
 N и s – взаимно простые
(обычно достигается выбором простого N или $N=2^P$),
 $s = \text{anotherHashCode}(\text{key})$

Преимущества хэш-таблиц

- Универсальность
- Очень быстрое время работы в «нормальном», среднем случае
 - Невысокая заполняемость
 - Количество элементов в таблице \leq Количество ячеек
 - Метод коллизий
 - Количество элементов в таблице $< \frac{1}{2}$ Количество ячеек
 - Открытая адресации
 - Качественная хэш-функция

Недостатки хэш-таблиц

- Плохое время работы в худшем случае
 - Как правило $O(N)$ вместо $O(1)$
- Сравнительно высокий коэффициент при $O(1)$
 - Таблица с открытой адресацией здесь выигрывает у стандартного Java HashMap (метод цепочек). Но она плохо поддерживает удаление
 - Таблица с прямой адресацией ещё лучше
- Зависимость от качества хэш-функции и от «везения»
 - Такие методы, как двойное хэширование, придуманы как раз для уменьшения роли везения

Кукушкино хэширование

- Открытая адресация
- Две хэш-функции, определяющие **две** возможные ячейки для элемента
- Алгоритм поиска/удаления
 - Считаем первую хэш-функцию, ищем ключ в ней
 - Считаем вторую хэш-функцию, ищем ключ в ней

Кукушкино хэширование

- Открытая адресация
- Две хэш-функции, определяющие **две** возможные ячейки для элемента
- Алгоритм вставки примерно такой
 - Считаем первую хэш-функцию, если ячейка свободна, вставляем
 - Делаем то же со второй хэш-функцией
 - Если обе ячейки заняты, выталкиваем ключ из одной из них (кукушка), вставляем наш ключ в эту ячейку, а для старого ключа повторяем тот же алгоритм вставки
 - С чем боремся?

Кукушкино хэширование

- Открытая адресация
- Две хэш-функции, определяющие **две** возможные ячейки для элемента
- Время поиска / удаления $O(1)$
- Время вставки $O(1)$ в среднем случае, в худшем по-прежнему $O(N)$

Хэш-таблица VS бинарное дерево поиска

➤ Что лучше?

Хэш-таблица VS бинарное дерево поиска

- Перфоманс
- Порядок элементов

Итоги

- Рассмотрено
 - Таблицы и хэш-таблицы
 - На основе цепочек
 - С открытой адресацией
- Далее
 - Графовые алгоритмы