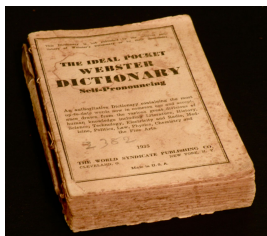


Речници

Трифон Трифонов

Структури от данни и програмиране,
спец. Компютърни науки, 2 поток, 2015/16 г.

8 януари 2016 г.



АТД: Речник

Структура, представяща функционална връзка между две множества от елементи (ключове и стойности).

АТД: Речник

Структура, представяща функционална връзка между две множества от елементи (ключове и стойности).

Още известна като: асоциативен списък (associative list) и карта (map)

АТД: Речник

Структура, представяща функционална връзка между две множества от елементи (ключове и стойности).

Още известна като: асоциативен списък (associative list) и карта (map)

Операции

- `create()` — създаване на празен речник
- `lookup(key)` — търсене на стойност по ключ
- `add(key, value)` — добавяне на връзка между ключ и стойност
- `remove(key)` — изтриване на ключ и свързаната с него стойност
- `keys()` — списък от всички ключове
- `values()` — списък от всички стойности

Реализация: списък

Свързан списък от двойки (ключ, стойност)

Реализация: списък

Свързан списък от двойки (ключ, стойност)

- `lookup(key)`
- `add(key, value)`
- `remove(key)`

Реализация: списък

Свързан списък от двойки (ключ, стойност)

- `lookup(key)` — $O(n)$, обхождането е в реда на включването
- `add(key, value)`
- `remove(key)`

Реализация: списък

Свързан списък от двойки (ключ, стойност)

- `lookup(key)` — $O(n)$, обхождането е в реда на включването
- `add(key, value)` — $O(1)$, винаги в края на списъка
- `remove(key)`

Реализация: списък

Свързан списък от двойки (ключ, стойност)

- `lookup(key)` — $O(n)$, обхождането е в реда на включването
- `add(key, value)` — $O(1)$, винаги в края на списъка
- `remove(key)` — $O(n)$, първо ключът трябва да бъде намерен

Реализация: сортиран списък

Свързан списък от двойки (ключ, стойност), сортиран по ключове (считаме, че имаме наредба по ключовете)

Реализация: сортиран списък

Свързан списък от двойки (ключ, стойност), сортиран по ключове (считаме, че имаме наредба по ключовете)

- `lookup(key)`
- `add(key, value)`
- `remove(key)`

Реализация: сортиран списък

Свързан списък от двойки (ключ, стойност), сортиран по ключове (считаме, че имаме наредба по ключовете)

- `lookup(key)` — $O(n)$, отново може да се наложи да обходим целия списък!
- `add(key, value)`
- `remove(key)`

Реализация: сортиран списък

Свързан списък от двойки (ключ, стойност), сортиран по ключове (считаме, че имаме наредба по ключовете)

- `lookup(key)` — $O(n)$, отново може да се наложи да обходим целия списък!
- `add(key, value)` — $O(n)$, търсим мястото на новия ключ
- `remove(key)`

Реализация: сортиран списък

Свързан списък от двойки (ключ, стойност), сортиран по ключове (считаме, че имаме наредба по ключовете)

- `lookup(key)` — $O(n)$, отново може да се наложи да обходим целия списък!
- `add(key, value)` — $O(n)$, търсим мястото на новия ключ
- `remove(key)` — $O(n)$, първо ключът трябва да бъде намерен

Реализация: сортиран масив

Динамичен масив от двойки (ключ, стойност), сортиран по ключове

Реализация: сортиран масив

Динамичен масив от двойки (ключ, стойност), сортиран по ключове
Алгоритъм за двоично търсене:

- 1 търсим ключ Y в сортиран масив в интервала $[left; right]$
- 2 първоначално $left = 0, right = n - 1$
- 3 намираме средата на масива $mid = (left + right) / 2$
- 4 сравняваме търсения ключ Y с ключа X на позиция mid
- 5 ако $Y == X$ — успех
- 6 ако $Y < X$ — търсим Y отляво, $right = mid - 1$ и към 3
- 7 ако $Y > X$ — търсим X отдясно, $left = mid + 1$ и към 3

Реализация: сортиран масив

Динамичен масив от двойки (ключ, стойност), сортиран по ключове
Алгоритъм за двоично търсене:

- 1 търсим ключ Y в сортиран масив в интервала $[left; right]$
 - 2 първоначално $left = 0, right = n - 1$
 - 3 намираме средата на масива $mid = (left + right) / 2$
 - 4 сравняваме търсения ключ Y с ключа X на позиция mid
 - 5 ако $Y == X$ — успех
 - 6 ако $Y < X$ — търсим Y отляво, $right = mid - 1$ и към 3
 - 7 ако $Y > X$ — търсим X отдясно, $left = mid + 1$ и към 3
- `lookup(key)`
 - `add(key, value)`
 - `remove(key)`

Реализация: сортиран масив

Динамичен масив от двойки (ключ, стойност), сортиран по ключове
 Алгоритъм за двоично търсене:

- 1 търсим ключ Y в сортиран масив в интервала $[left; right]$
 - 2 първоначално $left = 0, right = n - 1$
 - 3 намираме средата на масива $mid = (left + right) / 2$
 - 4 сравняваме търсения ключ Y с ключа X на позиция mid
 - 5 ако $Y == X$ — успех
 - 6 ако $Y < X$ — търсим Y отляво, $right = mid - 1$ и към 3
 - 7 ако $Y > X$ — търсим X отдясно, $left = mid + 1$ и към 3
- $lookup(key) — O(\log n)$, можем да използваме двоично търсене
 - $add(key, value)$
 - $remove(key)$

Реализация: сортиран масив

Динамичен масив от двойки (ключ, стойност), сортиран по ключове
 Алгоритъм за двоично търсене:

- ① търсим ключ Y в сортиран масив в интервала $[left; right]$
 - ② първоначално $left = 0, right = n - 1$
 - ③ намираме средата на масива $mid = (left + right) / 2$
 - ④ сравняваме търсения ключ Y с ключа X на позиция mid
 - ⑤ ако $Y == X$ — успех
 - ⑥ ако $Y < X$ — търсим Y отляво, $right = mid - 1$ и към 3
 - ⑦ ако $Y > X$ — търсим X отдясно, $left = mid + 1$ и към 3
- $lookup(key)$ — $O(\log n)$, можем да използваме двоично търсене
 - $add(key, value)$ — $O(n)$, може да разместим всички елементи
 - $remove(key)$

Реализация: сортиран масив

Динамичен масив от двойки (ключ, стойност), сортиран по ключове
 Алгоритъм за двоично търсене:

- ① търсим ключ Y в сортиран масив в интервала $[left; right]$
 - ② първоначално $left = 0, right = n - 1$
 - ③ намираме средата на масива $mid = (left + right) / 2$
 - ④ сравняваме търсения ключ Y с ключа X на позиция mid
 - ⑤ ако $Y == X$ — успех
 - ⑥ ако $Y < X$ — търсим Y отляво, $right = mid - 1$ и към 3
 - ⑦ ако $Y > X$ — търсим X отдясно, $left = mid + 1$ и към 3
- $lookup(key) — O(\log n)$, можем да използваме двоично търсене
 - $add(key, value) — O(n)$, може да разместим всички елементи
 - $remove(key) — O(n)$, може да разместим всички елементи

Реализация: двоично дърво за търсене

Двоично дърво за търсене с двойки (ключ, стойност), като се използва наредбата между ключовете

Реализация: двоично дърво за търсене

Двоично дърво за търсене с двойки (ключ, стойност), като се използва наредбата между ключовете

- обикновено двоично дърво за търсене

Реализация: двоично дърво за търсене

Двоично дърво за търсене с двойки (ключ, стойност), като се използва наредбата между ключовете

- обикновено двоично дърво за търсене
 - lookup, add, remove са със средна сложност $O(\log n)$...

Реализация: двоично дърво за търсене

Двоично дърво за търсене с двойки (ключ, стойност), като се използва наредбата между ключовете

- обикновено двоично дърво за търсене
 - lookup, add, remove са със средна сложност $O(\log n)$...
 - ...но са с най-лоша сложност $O(n)$

Реализация: двоично дърво за търсене

Двоично дърво за търсене с двойки (ключ, стойност), като се използва наредбата между ключовете

- обикновено двоично дърво за търсене
 - lookup, add, remove са със средна сложност $O(\log n)$...
 - ...но са с най-лоша сложност $O(n)$
- самобалансиращо се дърво за търсене (AVL, червено-черно, ...)

Реализация: двоично дърво за търсене

Двоично дърво за търсене с двойки (ключ, стойност), като се използва наредбата между ключовете

- обикновено двоично дърво за търсене
 - lookup, add, remove са със средна сложност $O(\log n)$...
 - ...но са с най-лоша сложност $O(n)$
- самобалансиращо се дърво за търсене (AVL, червено-черно, ...)
 - lookup, add, remove са със най-лоша и средна сложност $O(\log n)$

Реализация: двоично дърво за търсене

Двоично дърво за търсене с двойки (ключ, стойност), като се използва наредбата между ключовете

- обикновено двоично дърво за търсене
 - lookup, add, remove са със средна сложност $O(\log n)$...
 - ...но са с най-лоша сложност $O(n)$
- самобалансиращо се дърво за търсене (AVL, червено-черно, ...)
 - lookup, add, remove са със най-лоша и средна сложност $O(\log n)$
- B-дърво

Реализация: двоично дърво за търсене

Двоично дърво за търсене с двойки (ключ, стойност), като се използва наредбата между ключовете

- обикновено двоично дърво за търсене
 - lookup, add, remove са със средна сложност $O(\log n)$...
 - ...но са с най-лоша сложност $O(n)$
- самобалансиращо се дърво за търсене (AVL, червено-черно, ...)
 - lookup, add, remove са със най-лоша и средна сложност $O(\log n)$
- B-дърво
 - същата сложност като при самобалансиращи се дървета

Реализация: двоично дърво за търсене

Двоично дърво за търсене с двойки (ключ, стойност), като се използва наредбата между ключовете

- обикновено двоично дърво за търсене
 - lookup, add, remove са със средна сложност $O(\log n)$...
 - ...но са с най-лоша сложност $O(n)$
- самобалансиращо се дърво за търсене (AVL, червено-черно, ...)
 - lookup, add, remove са със най-лоша и средна сложност $O(\log n)$
- B-дърво
 - същата сложност като при самобалансиращи се дървета
 - допълнително предимство: всеки възел е с фиксиран размер
 - може да се избере да съвпада с физически/логически сектор на външно записващо устройство
 - това може да доведе до увеличаване на производителността

Хеш таблица

Масив с фиксиран размер n от двойки (ключ, стойност).

- **Основна идея:** На базата на ключ k бързо да определяме индекс в масива i_k , където той трябва да бъде търсен, добавен или изтрит

Хеш таблица

Масив с фиксиран размер n от двойки (ключ, стойност).

- **Основна идея:** На базата на ключ k бързо да определяме индекс в масива i_k , където той трябва да бъде търсен, добавен или изтрит
- **Частен случай:** Допустимите ключове са последователните цели числа от 0 до $m < n$

Хеш таблица

Масив с фиксиран размер n от двойки (ключ, стойност).

- **Основна идея:** На базата на ключ k бързо да определяме индекс в масива i_k , където той трябва да бъде търсен, добавен или изтрит
- **Частен случай:** Допустимите ключове са последователните цели числа от 0 до $m < n$
- В този случай съпоставяме на всеки ключ индекс със същия номер $i_k := k$

Хеш таблица

Масив с фиксиран размер n от двойки (ключ, стойност).

- **Основна идея:** На базата на ключ k бързо да определяме индекс в масива i_k , където той трябва да бъде търсен, добавен или изтрит
- **Частен случай:** Допустимите ключове са последователните цели числа от 0 до $m < n$
- В този случай съпоставяме на всеки ключ индекс със същия номер $i_k := k$
- Какво да правим в останалите случаи?

Хеш таблица

Масив с фиксиран размер n от двойки (ключ, стойност).

- **Основна идея:** На базата на ключ k бързо да определяме индекс в масива i_k , където той трябва да бъде търсен, добавен или изтрит
- **Частен случай:** Допустимите ключове са последователните цели числа от 0 до $m < n$
- В този случай съпоставяме на всеки ключ индекс със същия номер $i_k := k$
- Какво да правим в останалите случаи?
- Търсим **функция** $h : \text{Key} \rightarrow [0; n)$, която да изчислява индекса съответстващ на всеки ключ, т.е. $i_k := h(k)$

Хеш функции

Каква трябва да бъде функцията $h : \text{Key} \rightarrow [0; n)$?

- **инекция**, ако е възможно

Хеш функции

Каква трябва да бъде функцията $h : Key \rightarrow [0; n)$?

- **инекция**, ако е възможно, т.е. ако $|Key| \leq n$

Хеш функции

Каква трябва да бъде функцията $h : \text{Key} \rightarrow [0; n)$?

- **инекция**, ако е възможно, т.е. ако $|\text{Key}| \leq n$
 - ако $|\text{Key}| > n$ със сигурност ще има $k_1 \neq k_2$, така че $h(k_1) = h(k_2)$

Хеш функции

Каква трябва да бъде функцията $h : \text{Key} \rightarrow [0; n)$?

- **инекция**, ако е възможно, т.е. ако $|\text{Key}| \leq n$
 - ако $|\text{Key}| > n$ със сигурност ще има $k_1 \neq k_2$, така че $h(k_1) = h(k_2)$
 - такава ситуация нарича **КОЛИЗИЯ**

Хеш функции

Каква трябва да бъде функцията $h : \text{Key} \rightarrow [0; n)$?

- **инекция**, ако е възможно, т.е. ако $|\text{Key}| \leq n$
 - ако $|\text{Key}| > n$ със сигурност ще има $k_1 \neq k_2$, така че $h(k_1) = h(k_2)$
 - такава ситуация нарича **КОЛИЗИЯ**
- **сюрекция**, ако $|\text{Key}| > n$ за да може да се изпълни целият масив

Хеш функции

Каква трябва да бъде функцията $h : \text{Key} \rightarrow [0; n)$?

- **инекция**, ако е възможно, т.е. ако $|\text{Key}| \leq n$
 - ако $|\text{Key}| > n$ със сигурност ще има $k_1 \neq k_2$, така че $h(k_1) = h(k_2)$
 - такава ситуация нарича **колизия**
- **сюрекция**, ако $|\text{Key}| > n$ за да може да се изпълни целият масив
- **равномерна**, т.е. с минимална вероятност за колизии

Хеш функции

Каква трябва да бъде функцията $h : \text{Key} \rightarrow [0; n)$?

- **инекция**, ако е възможно, т.е. ако $|\text{Key}| \leq n$
 - ако $|\text{Key}| > n$ със сигурност ще има $k_1 \neq k_2$, така че $h(k_1) = h(k_2)$
 - такава ситуация нарича **колизия**
- **сюрекция**, ако $|\text{Key}| > n$ за да може да се изпълни целият масив
- **равномерна**, т.е. с минимална вероятност за колизии
 - може да се постигне, ако стойностите на h (индексите) са колкото се може по-равномерно разпределени

Хеш функции

Каква трябва да бъде функцията $h : \text{Key} \rightarrow [0; n)$?

- **инекция**, ако е възможно, т.е. ако $|\text{Key}| \leq n$
 - ако $|\text{Key}| > n$ със сигурност ще има $k_1 \neq k_2$, така че $h(k_1) = h(k_2)$
 - такава ситуация нарича **колизия**
- **сюрекция**, ако $|\text{Key}| > n$ за да може да се изпълни целият масив
- **равномерна**, т.е. с минимална вероятност за колизии
 - може да се постигне, ако стойностите на h (индексите) са колкото се може по-равномерно разпределени
 - т.е. всеки индекс да се получава с приблизително равна вероятност при случаен избор на ключове

Хеш функции

Каква трябва да бъде функцията $h : \text{Key} \rightarrow [0; n)$?

- **инекция**, ако е възможно, т.е. ако $|\text{Key}| \leq n$
 - ако $|\text{Key}| > n$ със сигурност ще има $k_1 \neq k_2$, така че $h(k_1) = h(k_2)$
 - такава ситуация нарича **колизия**
- **сюрекция**, ако $|\text{Key}| > n$ за да може да се изпълни целият масив
- **равномерна**, т.е. с минимална вероятност за колизии
 - може да се постигне, ако стойностите на h (индексите) са колкото се може по-равномерно разпределени
 - т.е. всеки индекс да се получава с приблизително равна вероятност при случаен избор на ключове
 - т.е. всички множества $h^{-1}(i)$ са с приблизително еднаква големина

Хеш функции

Каква трябва да бъде функцията $h : \text{Key} \rightarrow [0; n)$?

- **инекция**, ако е възможно, т.е. ако $|\text{Key}| \leq n$
 - ако $|\text{Key}| > n$ със сигурност ще има $k_1 \neq k_2$, така че $h(k_1) = h(k_2)$
 - такава ситуация нарича **колизия**
- **сюрекция**, ако $|\text{Key}| > n$ за да може да се изпълни целият масив
- **равномерна**, т.е. с минимална вероятност за колизии
 - може да се постигне, ако стойностите на h (индексите) са колкото се може по-равномерно разпределени
 - т.е. всеки индекс да се получава с приблизително равна вероятност при случаен избор на ключове
 - т.е. всички множества $h^{-1}(i)$ са с приблизително еднаква големина
 - тази характеристика се мери със статистически методи (χ^2 тест)

Хеш функции

Каква трябва да бъде функцията $h : \text{Key} \rightarrow [0; n)$?

- **инекция**, ако е възможно, т.е. ако $|\text{Key}| \leq n$
 - ако $|\text{Key}| > n$ със сигурност ще има $k_1 \neq k_2$, така че $h(k_1) = h(k_2)$
 - такава ситуация нарича **колизия**
- **сюрекция**, ако $|\text{Key}| > n$ за да може да се изпълни целият масив
- **равномерна**, т.е. с минимална вероятност за колизии
 - може да се постигне, ако стойностите на h (индексите) са колкото се може по-равномерно разпределени
 - т.е. всеки индекс да се получава с приблизително равна вероятност при случаен избор на ключове
 - т.е. всички множества $h^{-1}(i)$ са с приблизително еднаква големина
 - тази характеристика се мери със статистически методи (χ^2 тест)
- функциите с горните свойства, се наричат **хеш функции**

Хеш функции

Каква трябва да бъде функцията $h : \text{Key} \rightarrow [0; n)$?

- **инекция**, ако е възможно, т.е. ако $|\text{Key}| \leq n$
 - ако $|\text{Key}| > n$ със сигурност ще има $k_1 \neq k_2$, така че $h(k_1) = h(k_2)$
 - такава ситуация нарича **колизия**
- **сюрекция**, ако $|\text{Key}| > n$ за да може да се изпълни целият масив
- **равномерна**, т.е. с минимална вероятност за колизии
 - може да се постигне, ако стойностите на h (индексите) са колкото се може по-равномерно разпределени
 - т.е. всеки индекс да се получава с приблизително равна вероятност при случаен избор на ключове
 - т.е. всички множества $h^{-1}(i)$ са с приблизително еднаква големина
 - тази характеристика се мери със статистически методи (χ^2 тест)
- функциите с горните свойства, се наричат **хеш функции**
- стойностите $h(k)$ се наричат **хеш кодове**

Разрешаване на колизии

В практическите случаи колизиите са неизбежни и трябва да имаме стратегия за справяне с тях.

Разрешаване на колизии

В практическите случаи колизиите са неизбежни и трябва да имаме стратегия за справяне с тях.

Има две основни стратегии за разрешаване на колизии:

- разрешаване чрез пряко свързване
- разрешаване чрез отворено адресиране

Разрешаване с пряко свързване

Идея: на всяка позиция в хеш таблицата съпоставяме “кофа” (bucket), която съдържа всички данни, чиито ключове имат еднакъв хеш код.

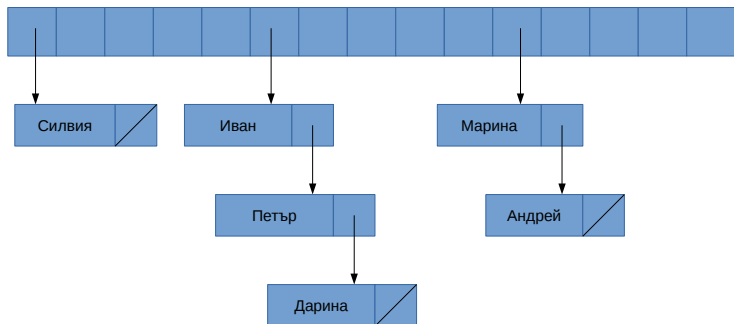
Разрешаване с пряко свързване

Идея: на всяка позиция в хеш таблицата съпоставяме “кофа” (bucket), която съдържа всички данни, чиито ключове имат еднакъв хеш код. Нарича се още “отворено хеширане” или “затворено адресиране”.

Разрешаване с пряко свързване

Идея: на всяка позиция в хеш таблицата съпоставяме “кофа” (bucket), която съдържа всички данни, чиито ключове имат еднакъв хеш код. Нарича се още “отворено хеширане” или “затворено адресиране”.

Пример: Нека $h(\text{"Силвия"}) = 0$,
 $h(\text{"Иван"}) = h(\text{"Петър"}) = h(\text{"Дарина"}) = 5$,
 $h(\text{"Марина"}) = h(\text{"Андрей"}) = 10$.



Разрешаване с отворено адресиране

Идея: използваме предварително фиксирана пермутация на индексите. При колизия изпробваме последователно индексите в реда, зададен в пермутацията.

Разрешаване с отворено адресиране

Идея: използваме предварително фиксирана пермутация на индексите. При колизия изпробваме последователно индексите в реда, зададен в пермутацията.

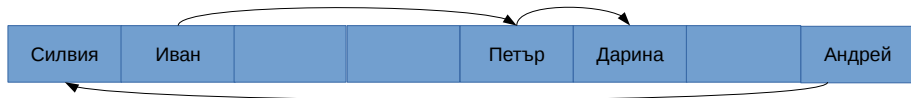
Нарича се още “затворено хеширане”.

Разрешаване с отворено адресиране

Идея: използваме предварително фиксирана пермутация на индексите. При колизия изпробваме последователно индексите в реда, зададен в пермутацията.

Нарича се още “затворено хеширане”.

Пример: Нека сме фиксирали пермутацията 1, 4, 5, 2, 6, 7, 0, 3. Нека $h(\text{"Иван"}) = h(\text{"Петър"}) = 1$, $h(\text{"Дарина"}) = 4$, $h(\text{"Силвия"}) = h(\text{"Андрей"}) = 7$



Сравнение на двете стратегии

Пряко свързване

- лесно за реализация

Сравнение на двете стратегии

Пряко свързване

- лесно за реализация
- при добре избрана хеш функция кофите остават малки

Сравнение на двете стратегии

Пряко свързване

- лесно за реализация
- при добре избрана хеш функция кофите остават малки
- разхищение на памет при малки данни

Сравнение на двете стратегии

Пряко свързване

- лесно за реализация
- при добре избрана хеш функция кофите остават малки
- разхищение на памет при малки данни

Отворено адресиране

Сравнение на двете стратегии

Пряко свързване

- лесно за реализация
- при добре избрана хеш функция кофите остават малки
- разхищение на памет при малки данни

Отворено адресиране

- пести памет при малки данни

Сравнение на двете стратегии

Пряко свързване

- лесно за реализация
- при добре избрана хеш функция кофите остават малки
- разхищение на памет при малки данни

Отворено адресиране

- пести памет при малки данни
- пермутацията може да се генерира от вторична хеш функция

Сравнение на двете стратегии

Пряко свързване

- лесно за реализация
- при добре избрана хеш функция кофите остават малки
- разхищение на памет при малки данни

Отворено адресиране

- пести памет при малки данни
- пермутацията може да се генерира от вторична хеш функция
- при запълване на хеш-таблицата до около 70% производителността на търсенето рязко пада и се налага преоразмеряване на масива и прехеширане на всички стойности

Сложност на операциите на хеш таблица

Операциите lookup, add, remove имат:

- сложност $O(n)$ в най-лошия случай, когато много елементи попаднат в една кофа

Сложност на операциите на хеш таблица

Операциите lookup, add, remove имат:

- сложност $O(n)$ в най-лошия случай, когато много елементи попаднат в една кофа
- амортизирана средна сложност $O(1)$

Сложност на операциите на хеш таблица

Операциите lookup, add, remove имат:

- сложност $O(n)$ в най-лошия случай, когато много елементи попаднат в една кофа
- амортизирана средна сложност $O(1)$
- амортизираната сложност се пресмята като се вземе предвид средния брой операции при продължително използване на хеш таблицата