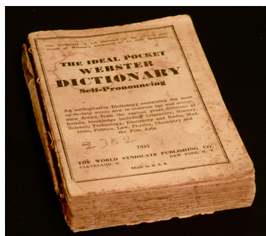
Речници

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** $h : Key \to [0; n)$, която да изчислява индекса съответстващ на всеки ключ, т.е. $i_k := h(k)$

Хеш функции

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

- **инекция**, ако е възможно

Хеш функции

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

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

Хеш функции

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

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

Хеш функции

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

- **инекция**, ако е възможно, т.е. ако $|Key| \leq n$
    - ако $|Key| > n$ 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 $h : Key \to [0; n)$?

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

Хеш функции

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

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

Хеш функции

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

- **инекция**, ако е възможно, т.е. ако $|Key| \leq n$
  - ако $|Key| > n$ със сигурност ще има $k_1 \neq k_2$, така че $h(k_1) = h(k_2)$
  - такава ситуация нарича **колизия**
- **сюрекция**, ако $|Key| > n$ 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 $h : Key \to [0; n)$?

- **инекция**, ако е възможно, т.е. ако $|Key| \leq n$
  - ако $|Key| > n$ със сигурност ще има $k_1 \neq k_2$, така че $h(k_1) = h(k_2)$
  - такава ситуация нарича **колизия**
- **сюрекция**, ако $|Key| > n$ 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 $h : Key \to [0; n)$?

- **инекция**, ако е възможно, т.е. ако $|Key| \leq n$
    - ако $|Key| > n$ със сигурност ще има $k_1 \neq k_2$, така че $h(k_1) = h(k_2)$
    - такава ситуация нарича **колизия**
- **сюрекция**, ако $|Key| > n$ 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 $h : Key \to [0; n)$?

- **инекция**, ако е възможно, т.е. ако $|Key| \leq n$
    - ако $|Key| > n$ със сигурност ще има $k_1 \neq k_2$, така че $h(k_1) = h(k_2)$
    - такава ситуация нарича **колизия**
- **сюрекция**, ако $|Key| > n$ 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 $h : Key \to [0; n)$?

- **инекция**, ако е възможно, т.е. ако $|Key| \leq n$
  - ако $|Key| > n$ със сигурност ще има $k_1 \neq k_2$, така че $h(k_1) = h(k_2)$
  - такава ситуация нарича **колизия**
- **сюрекция**, ако $|Key| > n$ 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 $h : Key \to [0; n)$?

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

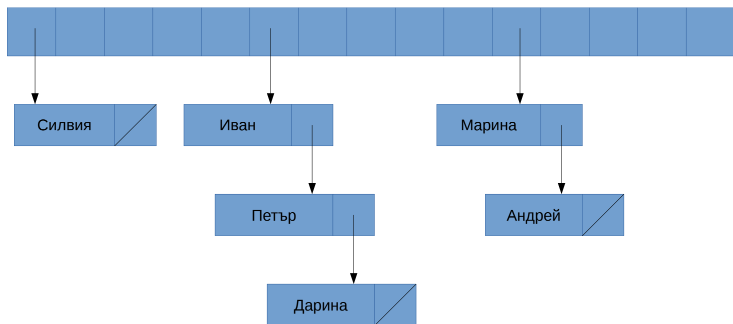
Разрешаване с пряко свързване

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 $h("Силвия") = 0$,
$h("Иван") = h("Петър") = h("Дарина") = 5$,
$h("Марина") = h("Андрей") = 10$.



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.

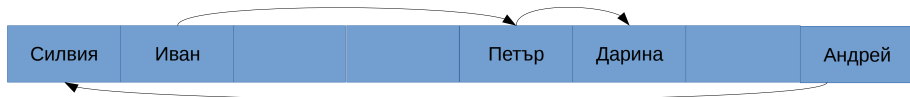
Нарича се още "затворено хеширане".

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

Операцията `lookup` има:

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

Сложност на операциите на хеш таблица

Операцията `lookup` има:

- 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

Операцията `lookup` има:

- 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

Разширение на хеш таблици

- при стратегия с отворено адресиране хеш таблицата може да се препълни

Разширение на хеш таблици

- при стратегия с отворено адресиране хеш таблицата може да се препълни
- при стратегия с пряко адресиране няма такава опасност, но се увеличава вероятността от колизии

Разширение на хеш таблици

- при стратегия с отворено адресиране хеш таблицата може да се препълни
- при стратегия с пряко адресиране няма такава опасност, но се увеличава вероятността от колизии
- при приближаване на максимален капацитет е добре масивът да се преоразмери

Разширение на хеш таблици

- при стратегия с отворено адресиране хеш таблицата може да се препълни
- при стратегия с пряко адресиране няма такава опасност, но се увеличава вероятността от колизии
- при приближаване на максимален капацитет е добре масивът да се преоразмери
- това налага промяна на хеш функцията и преизчисляване на някои или всички хеш кодове

Разширение на хеш таблици

- при стратегия с отворено адресиране хеш таблицата може да се препълни
- при стратегия с пряко адресиране няма такава опасност, но се увеличава вероятността от колизии
- при приближаване на максимален капацитет е добре масивът да се преоразмери
- това налага промяна на хеш функцията и преизчисляване на някои или всички хеш кодове
- стратегии за разширение

Разширение на хеш таблици

- при стратегия с отворено адресиране хеш таблицата може да се препълни
- при стратегия с пряко адресиране няма такава опасност, но се увеличава вероятността от колизии
- при приближаване на максимален капацитет е добре масивът да се преоразмери
- това налага промяна на хеш функцията и преизчисляване на някои или всички хеш кодове
- стратегии за разширение
  - **пълно копиране** — включване на всички елементи в новия масив след ново изчисляване на хеш кодовете им

Разширение на хеш таблици

- при стратегия с отворено адресиране хеш таблицата може да се препълни
- при стратегия с пряко адресиране няма такава опасност, но се увеличава вероятността от колизии
- при приближаване на максимален капацитет е добре масивът да се преоразмери
- това налага промяна на хеш функцията и преизчисляване на някои или всички хеш кодове
- стратегии за разширение
    - **пълно копиране** — включване на всички елементи в новия масив след ново изчисляване на хеш кодовете им
    - **инкременталлно разширение** — елементите се прехвърлят постепенно, на всяка операция за добавяне, докато не се прехвърлят всички елементи. Докато това се случи, се търси и в старата и в новата хеш таблица

Криптографски хеш функции

- в криптографията хеш функциите се използват за пресмятане на цифрови отпечатъци на блокове данни

Криптографски хеш функции

- в криптографията хеш функциите се използват за пресмятане на цифрови отпечатъци на блокове данни
- изискват се допълнителни свойства:

Криптографски хеш функции

- в криптографията хеш функциите се използват за пресмятане на цифрови отпечатъци на блокове данни
- изискват се допълнителни свойства:
- **ефективност** — функцията да може да бъде пресмятана бързо

Криптографски хеш функции

- в криптографията хеш функциите се използват за пресмятане на цифрови отпечатъци на блокове данни
- изискват се допълнителни свойства:
- **ефективност** — функцията да може да бъде пресмятана бързо
- **еднопосочност** — да е практически трудно възстановяването на оригиналните данни по хеш кода

Криптографски хеш функции

- в криптографията хеш функциите се използват за пресмятане на цифрови отпечатъци на блокове данни
- изискват се допълнителни свойства:
- **ефективност** — функцията да може да бъде пресмятана бързо
- **еднопосочност** — да е практически трудно възстановяването на оригиналните данни по хеш кода
    - до момента не е доказано дали такива функции съществуват!

Криптографски хеш функции

- в криптографията хеш функциите се използват за пресмятане на цифрови отпечатъци на блокове данни
- изискват се допълнителни свойства:
- **ефективност** — функцията да може да бъде пресмятана бързо
- **еднопосочност** — да е практически трудно възстановяването на оригиналните данни по хеш кода
  - до момента не е доказано дали такива функции съществуват!
- **устойчивост към обръщане** — да е практически трудно по даден блок данни намирането на друг блок със същия хеш код

Криптографски хеш функции

- в криптографията хеш функциите се използват за пресмятане на цифрови отпечатъци на блокове данни
- изискват се допълнителни свойства:
- **ефективност** — функцията да може да бъде пресмятана бързо
- **еднопосочност** — да е практически трудно възстановяването на оригиналните данни по хеш кода
  - до момента не е доказано дали такива функции съществуват!
- **устойчивост към обръщане** — да е практически трудно по даден блок данни намирането на друг блок със същия хеш код
- **устойчивост към колизии** — да е практически трудно намирането на каквато и да е колизия

Криптографски хеш функции

- в криптографията хеш функциите се използват за пресмятане на цифрови отпечатъци на блокове данни
- изискват се допълнителни свойства:
- **ефективност** — функцията да може да бъде пресмятана бързо
- **еднопосочност** — да е практически трудно възстановяването на оригиналните данни по хеш кода
    - до момента не е доказано дали такива функции съществуват!
- **устойчивост към обръщане** — да е практически трудно по даден блок данни намирането на друг блок със същия хеш код
- **устойчивост към колизии** — да е практически трудно намирането на каквато и да е колизия
- **ефект на лавината** — малки промени в данни да водят до големи промени в хеш кода

Стандартни криптографски функции

- сред популярните съвременни криптографски функции са RIPEMD-160, SHA-256, Whirlpool

Стандартни криптографски функции

- сред популярните съвременни криптографски функции са RIPEMD-160, SHA-256, Whirlpool
- за някои доскоро популярни функции, като MD5 и SHA-1 вече има доказателства, че не са устойчиви към колизии и не се препоръчват за използване

Стандартни криптографски функции

- сред популярните съвременни криптографски функции са RIPEMD-160, SHA-256, Whirlpool
- за някои доскоро популярни функции, като MD5 и SHA-1 вече има доказателства, че не са устойчиви към колизии и не се препоръчват за използване
- популярни приложения:

Стандартни криптографски функции

- сред популярните съвременни криптографски функции са RIPEMD-160, SHA-256, Whirlpool
- за някои доскоро популярни функции, като MD5 и SHA-1 вече има доказателства, че не са устойчиви към колизии и не се препоръчват за използване
- популярни приложения:
  - запазване на пароли и проверка за съвпадение без да се вижда самата парола

Стандартни криптографски функции

- сред популярните съвременни криптографски функции са RIPEMD-160, SHA-256, Whirlpool
- за някои доскоро популярни функции, като MD5 и SHA-1 вече има доказателства, че не са устойчиви към колизии и не се препоръчват за използване
- популярни приложения:
  - запазване на пароли и проверка за съвпадение без да се вижда самата парола
    - речникови атаки: предварително пресметнати хеш кодове на думи

Стандартни криптографски функции

- сред популярните съвременни криптографски функции са RIPEMD-160, SHA-256, Whirlpool
- за някои доскоро популярни функции, като MD5 и SHA-1 вече има доказателства, че не са устойчиви към колизии и не се препоръчват за използване
- популярни приложения:
  - запазване на пароли и проверка за съвпадение без да се вижда самата парола
    - речникови атаки: предварително пресметнати хеш кодове на думи
    - предпазване от атаки: използване на "сол" — не непременно тайна дума, която се добавя към паролата преди хеширане

Стандартни криптографски функции

- сред популярните съвременни криптографски функции са RIPEMD-160, SHA-256, Whirlpool
- за някои доскоро популярни функции, като MD5 и SHA-1 вече има доказателства, че не са устойчиви към колизии и не се препоръчват за използване
- популярни приложения:
  - запазване на пароли и проверка за съвпадение без да се вижда самата парола
    - речникови атаки: предварително пресметнати хеш кодове на думи
    - предпазване от атаки: използване на "сол" — не непременно тайна дума, която се добавя към паролата преди хеширане
  - цифрови отпечатъци (fingerprints) на документи

Стандартни криптографски функции

- сред популярните съвременни криптографски функции са RIPEMD-160, SHA-256, Whirlpool
- за някои доскоро популярни функции, като MD5 и SHA-1 вече има доказателства, че не са устойчиви към колизии и не се препоръчват за използване
- популярни приложения:
  - запазване на пароли и проверка за съвпадение без да се вижда самата парола
    - речникови атаки: предварително пресметнати хеш кодове на думи
    - предпазване от атаки: използване на “сол” — не непременно тайна дума, която се добавя към паролата преди хеширане
  - цифрови отпечатъци (fingerprints) на документи
  - (псевдо)-уникален идентификатор, базиран на съдържание

Стандартни криптографски функции

- сред популярните съвременни криптографски функции са RIPEMD-160, SHA-256, Whirlpool
- за някои доскоро популярни функции, като MD5 и SHA-1 вече има доказателства, че не са устойчиви към колизии и не се препоръчват за използване
- популярни приложения:
  - запазване на пароли и проверка за съвпадение без да се вижда самата парола
    - речникови атаки: предварително пресметнати хеш кодове на думи
    - предпазване от атаки: използване на "сол" — не непременно тайна дума, която се добавя към паролата преди хеширане
  - цифрови отпечатъци (fingerprints) на документи
  - (псевдо)-уникален идентификатор, базиран на съдържание
  - доказателство за извършена работа (Bitcoin)

АТД: множество

Структура, която съдържа уникални елементи без определен ред.
Операции

- `create()` — създаване на празно множество
- `empty()` — проверка за празнота
- `insert(x)` — включване на елемент
- `remove(x)` — изключване на елемент
- `contains(x)` — проверка за съдържане на елемент
- `elements()` — списък от всички елементи

Реализация на множество

Структура от данни множество може да се реализира с използване на речник

- елементите са ключовете
- стойностите са фиктивни
- операциите на множеството наследяват операциите и тяхната сложност от съответната реализация на речник
- ако между елементите има наредба, може да се използва реализация на речник, която разчита на наредба

std::map

Реализация на речник чрез самобалансиращи се двоични дървета за търсене

- обикновено се реализират чрез червено-черни дървета
- дърветата съдържат `std::pair<Key,Value>`
- елементите се пазят сортирани по ключ

Операции

- `find(key)` — намиране на асоциация по ключ
- `insert(pair<...>(key,value))` — вмъкване на асоциация
- `erase(key)` — изтриване на асоциация по ключ
- `operator[key]` — достъп до елемент с даден ключ, може да се използва и за вмъкване
- `begin()`, `end()` — итератор към асоциациите в речника, **сортирани по ключ**

std::unordered_map (C++11)

Реализация на речник чрез хеш таблица

- колизиите се разрешават с пряко свързване
- може да се задават хеш функция и размер на таблицата

Операции

- `find(key)` — намиране на асоциация по ключ
- `insert(pair<...>(key,value))` — вмъкване на асоциация
- `erase(key)` — изтриване на асоциация по ключ
- `operator[key]` — достъп до елемент с даден ключ, може да се използва и за вмъкване
- `begin()`, `end()` — итератор към асоциациите в речника, **в неуточнен ред**
- `rehash(n)` — промяна на размера на хеш таблицата, при увеличаване всички елементи се хешират наново

std::set, std:unordered_set (C++11)

Реализации чрез дървета (std::set) или хеш таблица (std:unordered_set)

Операции

- find(x) — намиране на итератор към елемент
- insert(x) — включване на елемент
- erase(x) — изключване на елемент
- begin(), end() — итератор към елементите в множеството, **сортирани** (при std::set) или **без определен ред** (при std::unordered_set)