Hash dan ADT Map

IF2110/IF2111 – Algoritma dan Struktur Data
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

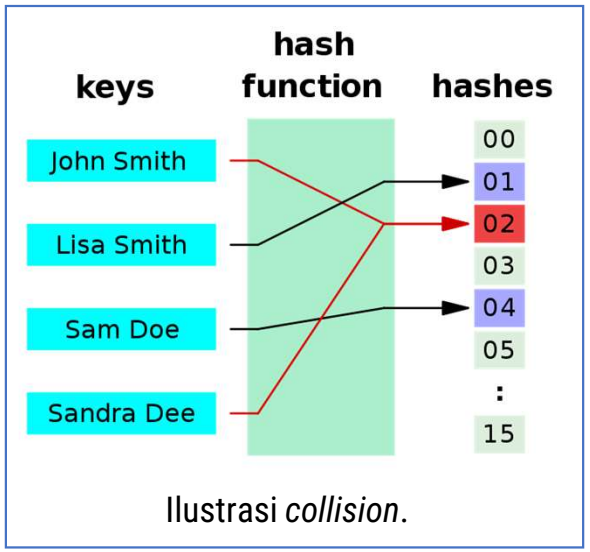
Hash dan Fungsi Hash

- **fungsi hash:** fungsi yang dapat digunakan untuk memetakan data berukuran "berapa pun" menjadi nilai berukuran tetap/tertentu.
  - Misal, string dengan panjang berbeda-beda dipetakan ke satu byte.
- **key:** data yang menjadi masukan **fungsi hash**.
- **hash** atau **digest:** nilai hasil perhitungan **fungsi hash**.

Karena nilai **hash** berukuran tetap (dan umumnya berukuran lebih kecil dari pada **key**), dapat terjadi **collision**. ☞

Fungsi hash yang baik:

1) Komputasinya cepat,
2) Meminimalisir terjadinya *collision*.

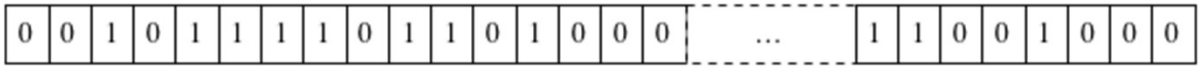


Ilustrasi *collision*.

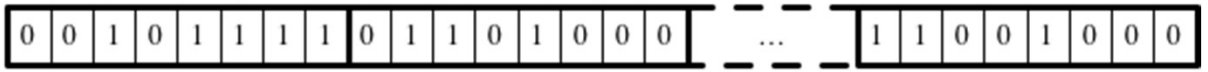
Contoh Fungsi Hash

Berikut contoh **fungsi hash** sangat sederhana menggunakan operasi XOR. (Tidak *secure*, rentan *collision*.)

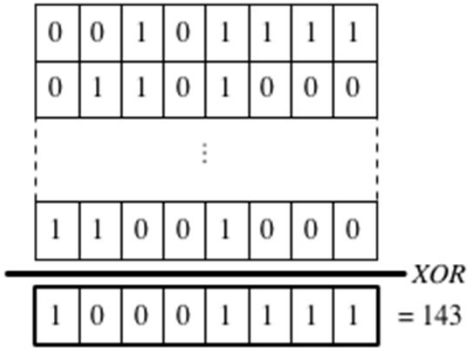
① Untuk memetakan *key* sepanjang apapun menjadi *hash* sepanjang 8 bit…



② Bagi *key* menjadi potongan-potongan dengan panjang 8 bit…



③ Kemudian lakukan operasi bitwise XOR secara beruntun terhadap setiap potongan tersebut.



④ Didapatlah nilai *hash* = 143.

Hash Table

Dengan memanfaatkan **fungsi hash** untuk menentukan di mana data disimpan dan bagaimana menemukannya kembali:

1) **Fungsi hash** mengubah **key** menjadi sebuah **hash** yang digunakan sebagai **indeks**.

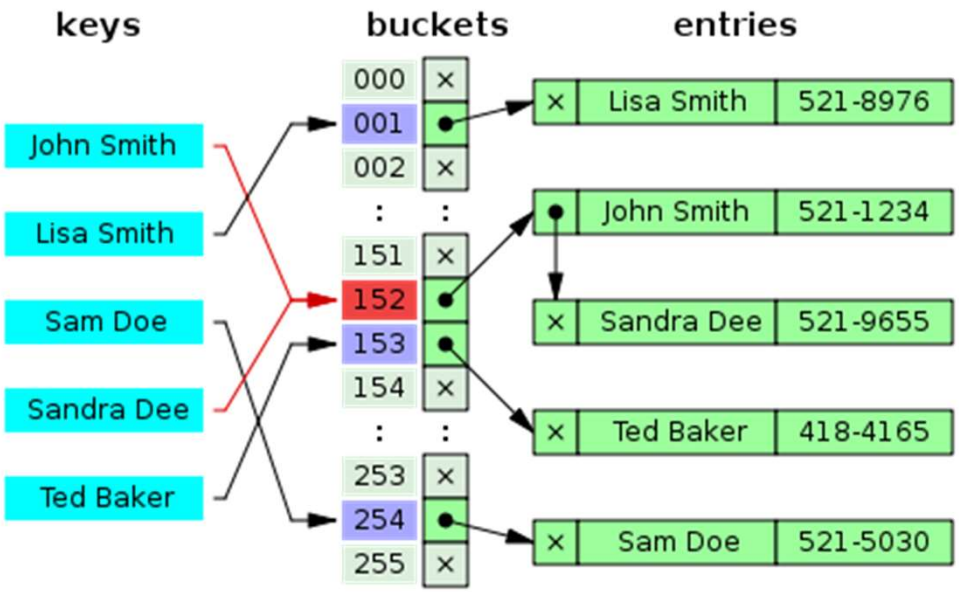
2) **Key** disimpan pada **slot** sesuai **indeks** tersebut.

3) Pada saat hendak menyimpan, **slot** bisa *kosong* atau *terisi*.
Slot *terisi* artinya sudah ada data lain yang **hash**-nya sama (terjadi **collision**).

   Sejumlah ***collision resolution strategies*** dikembangkan, biasanya berbasis:
   - *a)* *Hash chaining*: menyimpan sebuah *association list* berukuran kecil pada setiap sel *hash table*.
   - *b)* *Open addressing*.

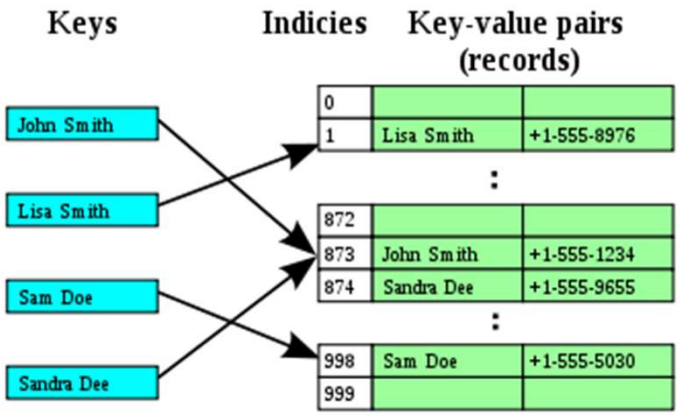
4) **Key** dan **value** disimpan pada **slot** *kosong* yang ditemukan.

a. Hash Chaining



b. Open Addressing/Closed hashing

Collision diselesaikan dengan cara mencari pada lokasi alternatif hingga pasangan yang diinginkan ditemukan atau ditemukan slot array yang belum terpakai, yang berarti belum ada pasangan dengan nilai *key* yang dicari.



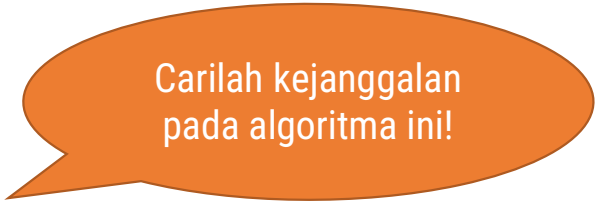
Metode pencarian yang dikenal:

1) *Linear probing*: interval pencarian tetap – biasanya 1.
2) *Quadratic probing*: interval pencarian bertambah secara linier – indeks dideskripsikan dengan fungsi kuadratik.
3) *Double hashing*: interval pencarian berikutnya ditentukan menggunakan fungsi *hash* yang lain.

Kembali ke ADT Map...

Contoh Algoritma *set* dengan Hash

```
procedure set(input/output m: Map, input k: KeyType, input v: ElType)
{ I.S. m terdefinisi, tidak penuh. }
{ F.S. Terdapat sebuah entri dengan key k pada m, dengan value=v. }
KAMUS LOKAL
  idx: integer
  found: boolean
ALGORITMA
  found ← false
  idx ← hash(k)
  while m.buffer[idx] ≠ NIL or not found do
    if m.buffer[idx].key = k then
      found ← true
    else
      idx ← idx+1
  if found then
    m.buffer[idx].value ← v
  else
    m.buffer[idx] ← <k,v>
    m.length ← m.length+1
```



Asumsi: fungsi hash(*k*:keytype) → address terdefinisi.
Collision ditangani dengan *linear probing*.
Slot kosong ditandai dengan nilai NIL.

Contoh Algoritma *find* dengan Hash

```
function find(m: Map, k: KeyType) → infotype
{ Mengembalikan value yang terasosiasi dengan key k pada m, atau
  VAL_UNDEF jika tidak ada key k di dalam m. }
KAMUS LOKAL
  idx: integer
  found: boolean
ALGORITMA
  found ← false
  idx ← hash(k)
  while m.buffer[idx] ≠ NIL or not found do
    if m.buffer[idx].key = k then
      found ← true
    else
      idx ← idx+1
  if found then
    → m.buffer[idx].value
  else
    → VAL_UNDEF
```

Asumsi: fungsi hash(*k*:keytype) → address terdefinisi.
Collision ditangani dengan *linear probing*.
Slot kosong ditandai dengan nilai NIL.

Load factor

Pada algoritma *set* tadi, tidak ada kondisi berhenti jika pencarian slot kosong sudah mencapai ujung tabel.

Alternatif:

1) Pencarian berhenti → padahal mungkin masih ada slot kosong di indeks kecil?
2) Pencarian lanjut ke indeks 0 lagi → menyulitkan operasi *unset* (bisa jadi harus memeriksa semua isi array)

Load factor: jumlah slot terisi dibagi total slot yang tersedia.

Hash table bekerja dengan baik saat *load factor* < 1. Maka alternatif berikutnya:

3) Ketika mencapai suatu batas *load factor* tertentu, dialokasikan tabel baru yang lebih besar, kemudian setiap elemen Map di-*hash* ulang.

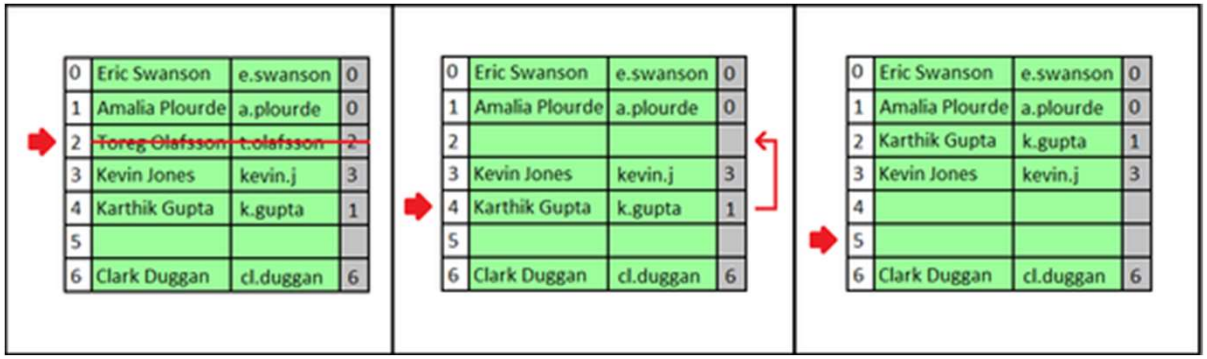
   Konsekuensi: tabel harus dinamis.

Penghapusan elemen?

Karena algoritma pencarian berhenti jika menemukan slot kosong, penghapusan elemen (*unset*) bisa mengakibatkan pencarian berikutnya salah.

> Mengira *not found*, padahal disimpan di slot berikutnya akibat *collision* ketika penambahan elemen.

Untuk itu pada penghapusan elemen di indeks i diperlukan pencarian & pemindahan elemen yang mungkin bisa menempati slot i, yaitu elemen yang nilai *hash*-nya ≤ i.



Latihan

Jika `MapEntry` pada Map dengan Hash diganti dengan `ElType`,

> dan operasi/ekspresi yang melibatkan `MapEntry`/`KeyType`/`ElType` diganti dengan `ElType`,
>
> → didapatkan implementasi ADT Set menggunakan Hash.

Buatlah kembali algoritma **`add`**, **`remove`**, **`isIn`**, dan **`union`** untuk Set dengan Hash, dan analisis lagi kinerjanya!