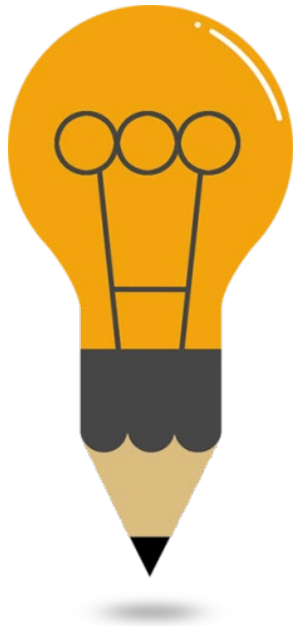
STRUKTUR DATA

Pertemuan 10

(Ratih Ngestrini)

Agenda Pertemuan



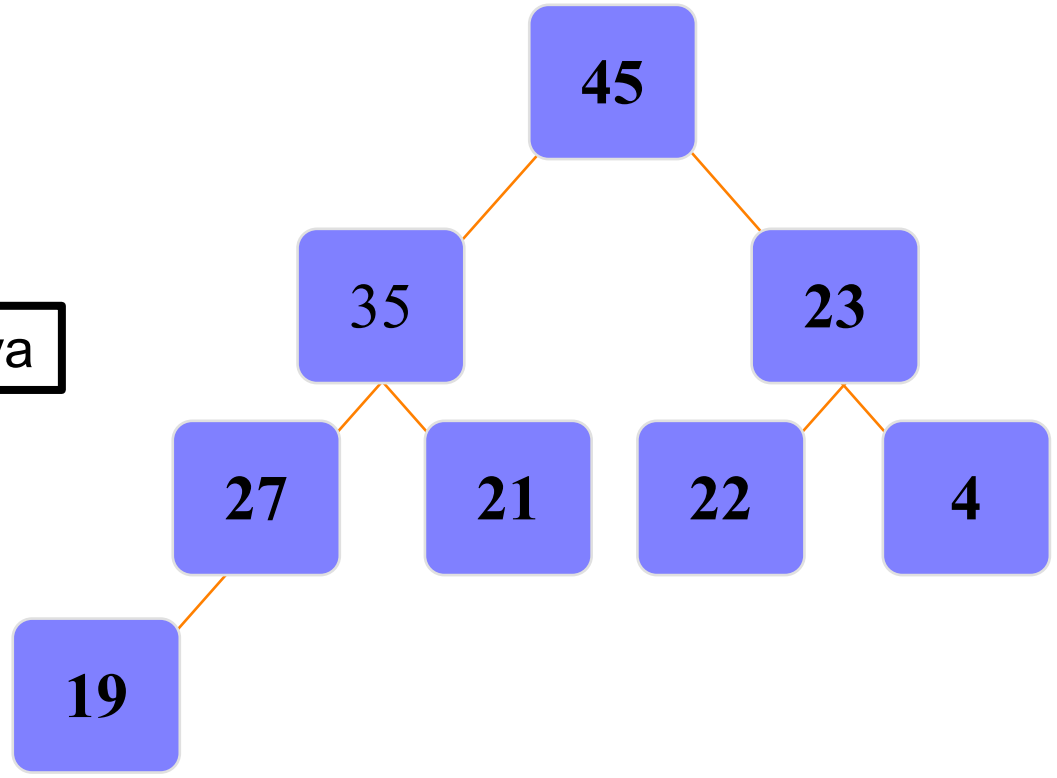
1. Heap
2. Hash Table

Heap

- **Heap** adalah suatu **Complete Binary Tree** (semua level pada tree, kecuali level terakhir, sepenuhnya diisi, dan, jika tingkat terakhir tree itu tidak lengkap, maka node pada level itu diisi kiri dulu)
- Bukan Binary Search Tree! Apa perbedaannya?

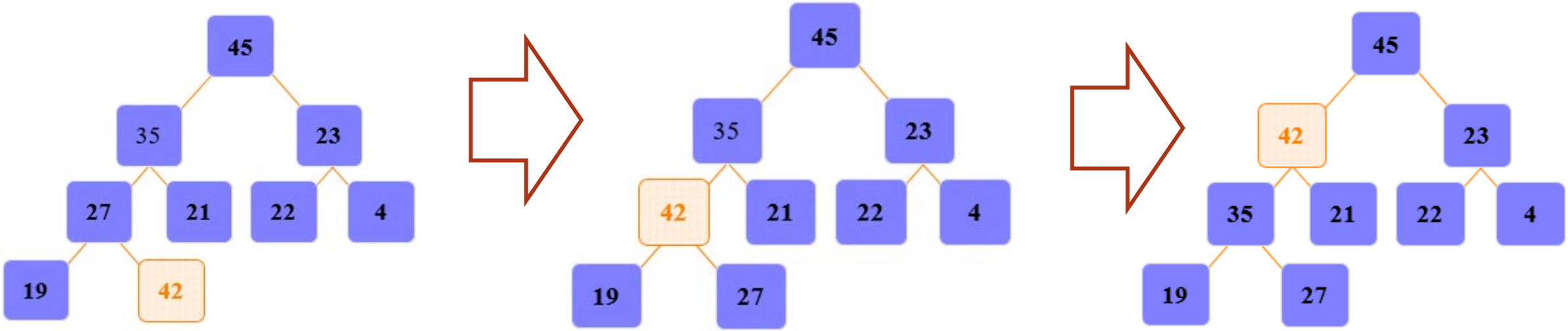
setiap node nilainya >= nilai anak - anaknya

Atau biasa disebut
Max Heap



Heap: Insert (Menambahkan Node Baru ke Heap)

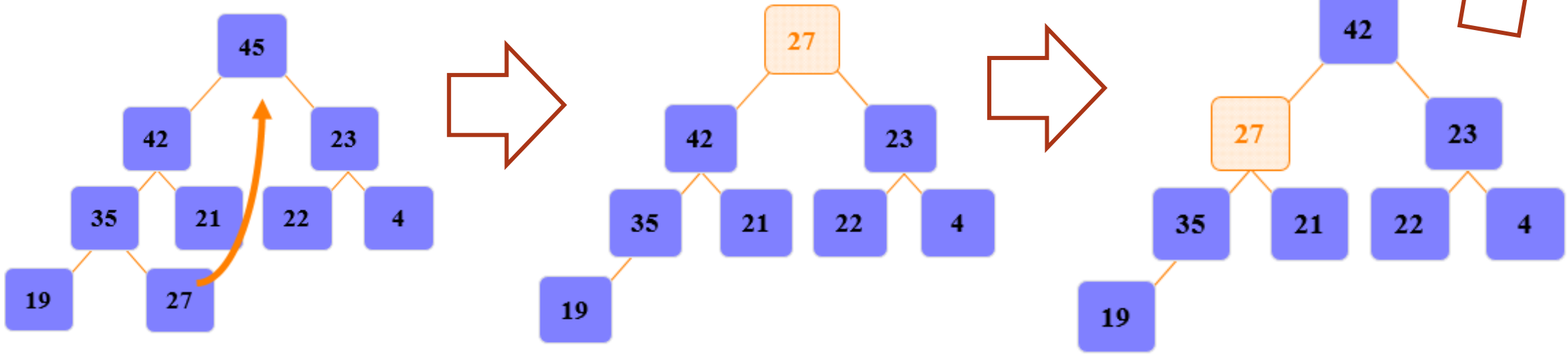
1. Tambahkan node baru pada posisi setelahnya (urutan: root – kiri – kanan)
2. Naikan node tersebut ke atas menggantikan parent-nya (**reheapification upward**) sampai dengan:
   - Node parent >= node tersebut
   - Node tersebut mencapai root



Heap: Delete Root Paling Atas (Nilai Paling Besar)

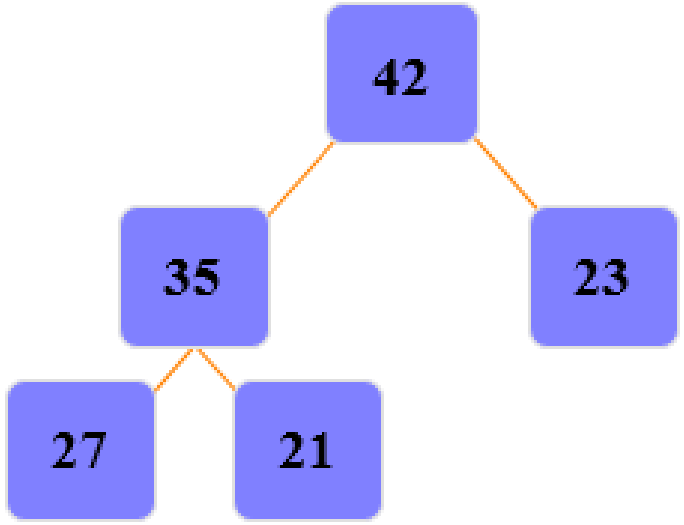
Keluarkan nilai paling besar → **antrian berprioritas**

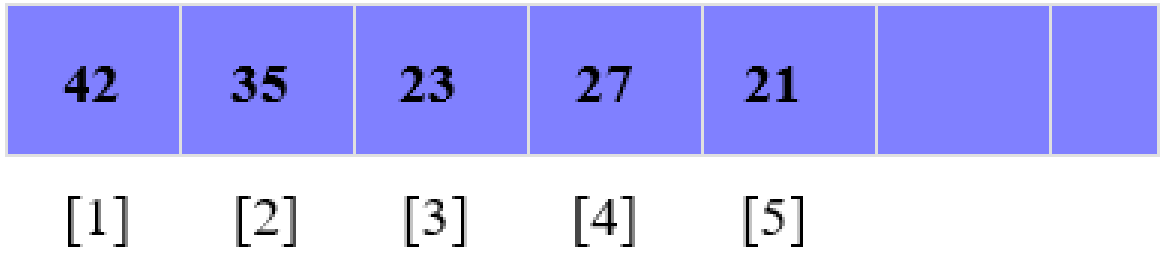
1. Pindahkan node terakhir ke node paling atas
2. Turunkan node paling atas bertukar dengan node yang lebih besar (**reheapification downward**), sampai dengan:
   - Node anak – anaknya <= node tersebut
   - Node tersebut mencapai leaf



Implementasi Heap

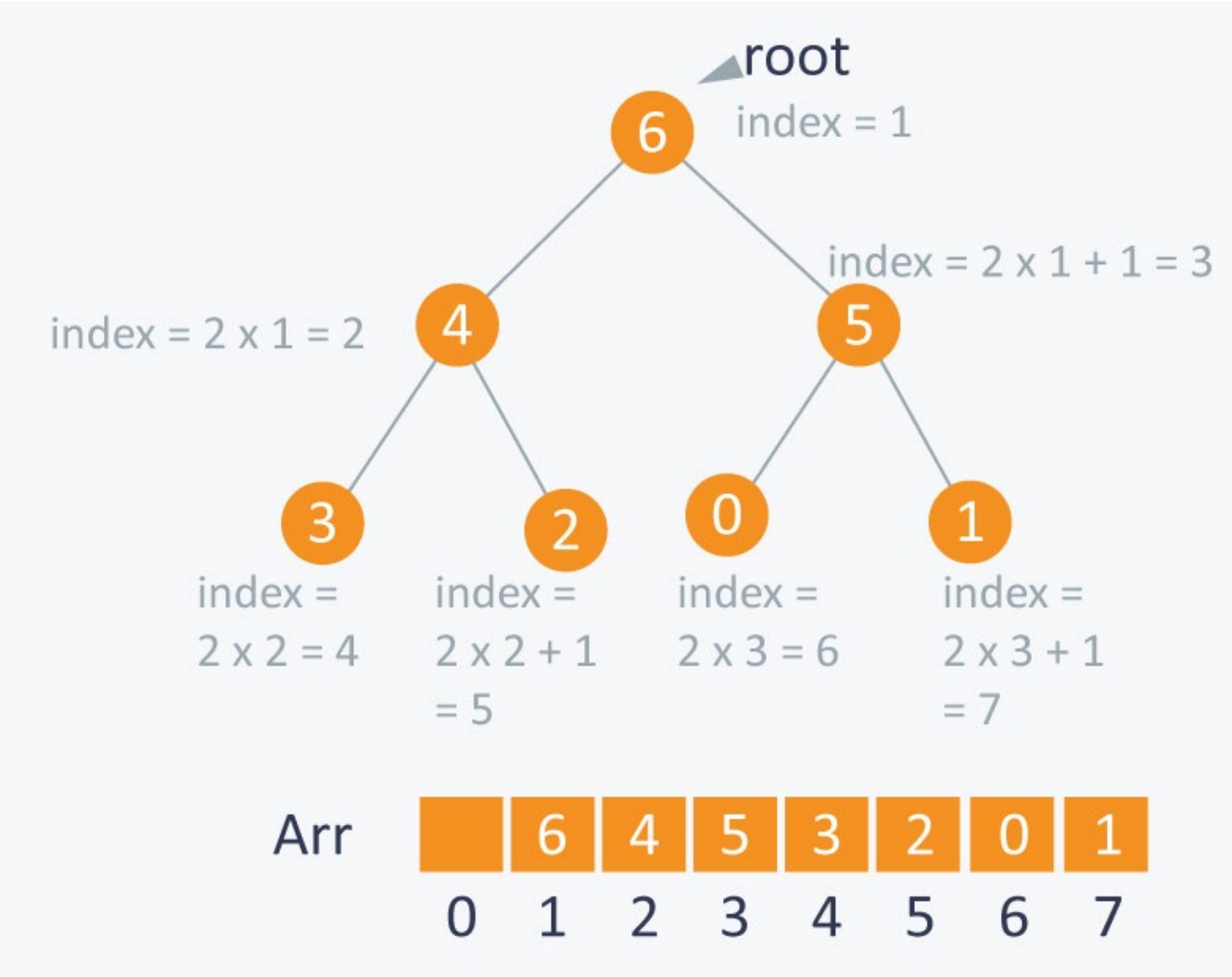
- Kita dapat menggunakan array untuk menyimpan node – node dari heap
- Masukan node – node di heap dengan urutan dari level paling atas turun ke level di bawahnya: anak kiri – anak kanan





Implementasi Heap

- Index root selalu 1
- Elemen di index ***i*** dalam array **Arr** berarti:
  - Parent-nya ada di index **i/2** (kecuali root karena tidak punya parent) dan dapat diakses di **Arr[i/2]**
  - Anak kirinya ada di index **2*i** dan dapat diakses di **Arr[2*i]**
  - Anak kanannya ada di index **2*i+1** dan dapat diakses di **Arr[2*i+1]**



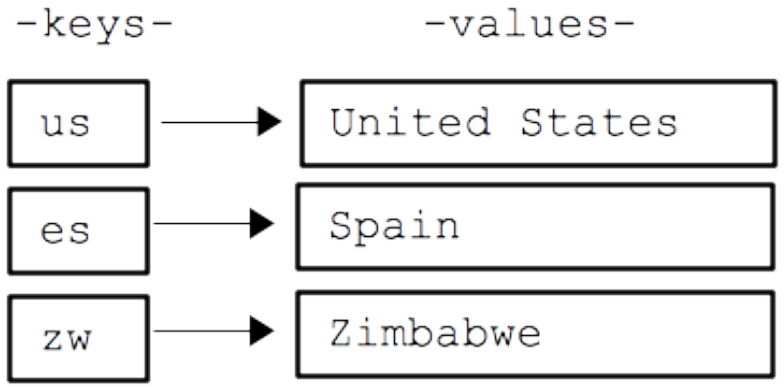
Implementasi Heap (Program)

Lihat di file heap.c

- `int parent (int i)` //returns index of parent
- `int left (int i)` //returns index of left child
- `int right (int i)` //return index of right child
- `void swap(int *p, int *q)` //swap value of two variables: parent dan child
- `void insert(int a[], int heapsize, int data)`
- `void delete(int a[],int heapsize)`
- `void display(int a[],int heapsize)`
- Main function

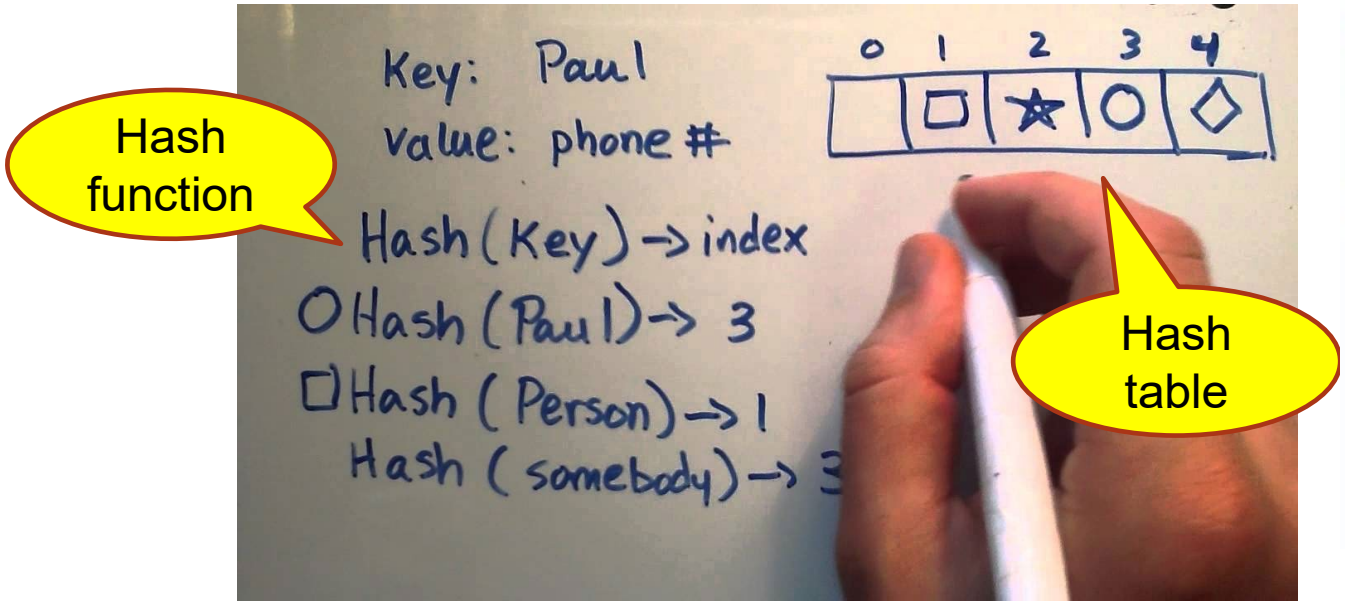
Hashing

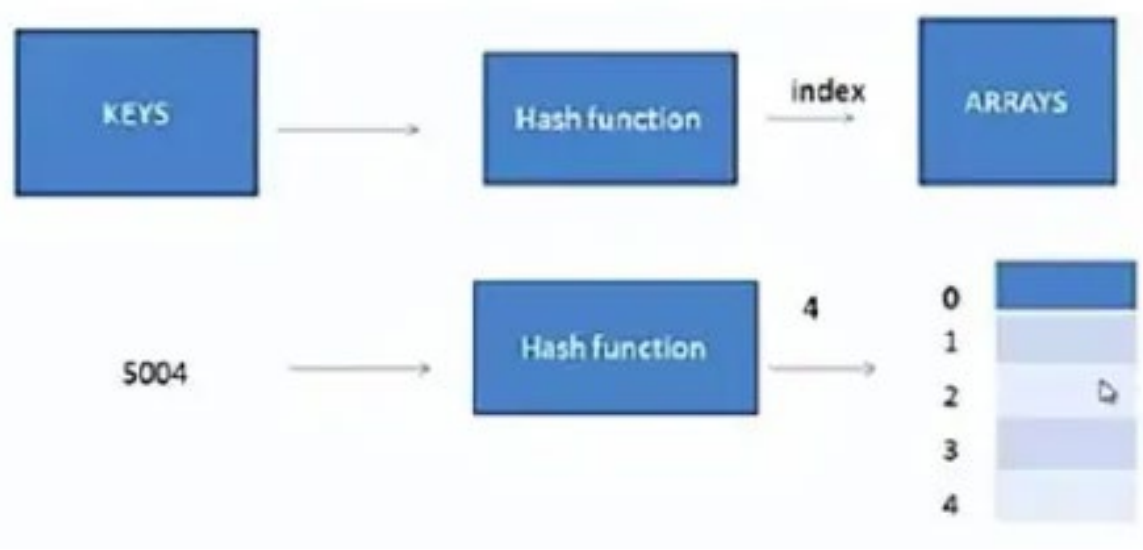
- Metode untuk menyimpan data dalam sebuah array agar penyimpanan, pencarian, penambahan, dan penghapusan data dapat dilakukan dengan cepat
- Dengan cara mengakses lokasi/index penyimpanan data secara langsung
- Ilustrasi:
  - Misal menyimpan barang di suatu loker yang ada nomor lokernya. Jika kita lupa nomor lokernya, kita akan cek satu – satu loker yang ada sampai barangnya ketemu (binary search). Tetapi jika kita tahu nomor lokernya, kita langsung bisa menemukan barang di loker tersebut dalam waktu yang cepat.
- Data akan dicari berdasarkan informasi uniknya → data disimpan dalam format **dictionary** atau **key-value pairs**



Hashing

- Dalam hashing, untuk penambahan atau pencarian, **key** tersebut akan dipetakan ke dalam suatu **index/lokasi** dari suatu data menggunakan **hash function**, dan terdapat **hash table** yang merupakan array tempat penyimpanan index/lokasi data yang berdasarkan output hash function.

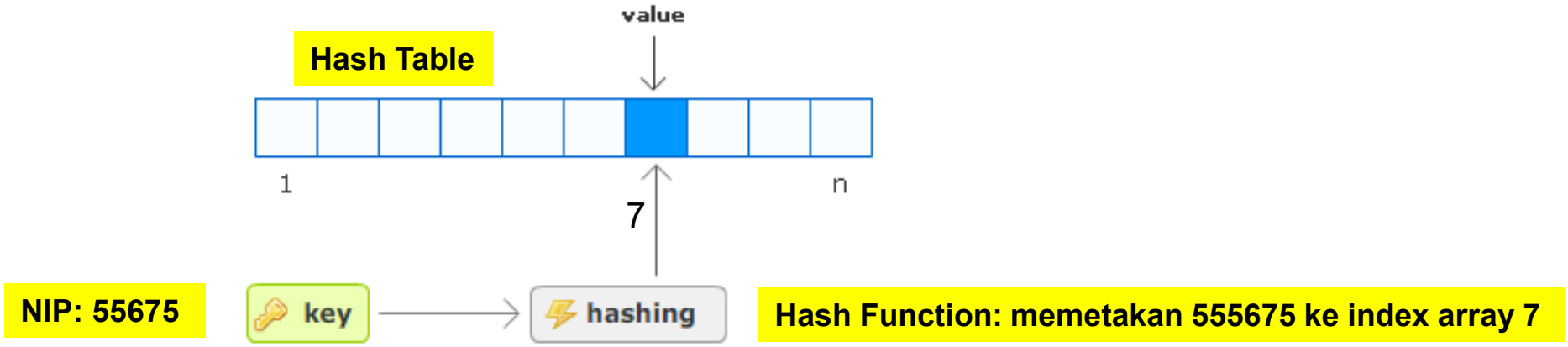




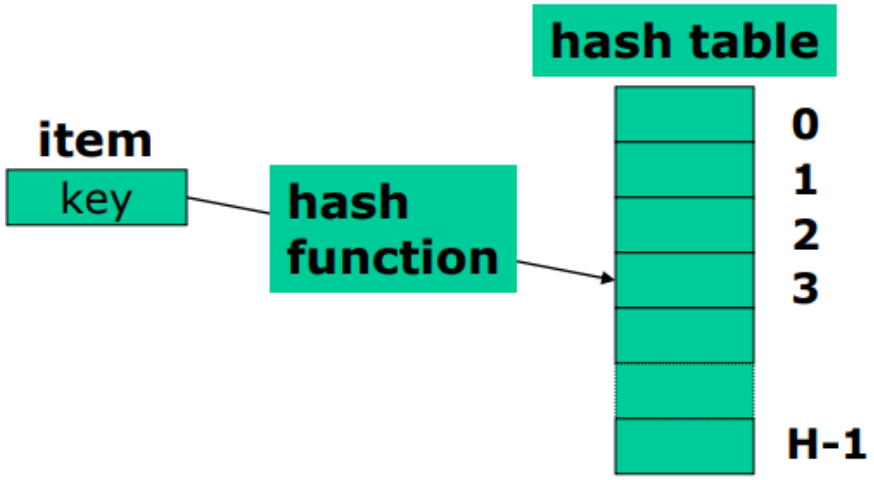
Mengapa Hash Table?

Contoh:

- Nomor Induk Pegawai (NIP) suatu perusahaan terdiri dari 5 digit antara 00000 – 99999. Bila menggunakan array, diperlukan array yang dapat menampung 100.000 elemen (karena array diakses berdasarkan index-nya, maka index = NIP). Kenyataannya, hanya ada 100 pegawai di perusahaan tersebut, sehingga akan terjadi pemborosan memory.
  - Diperlukan array yang berukuran kecil tetapi bisa menampung semua data → **Hash Table**
  - Bagaimana memetakan NIP dan index array hash table? → **Hash Function**



Hash Function



- Fungsi Hash memetakan elemen pada indeks array dari hash table
- Harus mempunyai sifat berikut:
  - Mudah dihitung
  - Dua key yang berbeda akan dipetakan pada dua sel yang berbeda pada array

  → Meminimalkan **Collision** (kondisi di mana nilai data yang berbeda mendapatkan nilai hashing atau posisi pada hash table yang sama)
  - Membagi key secara rata pada seluruh sel

Hash Function: Truncation

- Sebagian dari key dapat dibuang/diabaikan, bagian key sisanya digabungkan untuk membentuk index
- Contoh:

Phone no:	*index*
731-3018	338
539-2309	329
428-1397	217

- Cepat, tetapi sering gagal untuk membagi key secara merata pada seluruh index array

Hash Function: Folding

- Data dipecah menjadi beberapa bagian, kemudian tiap bagian tersebut digabungkan lagi dalam bentuk lain
- Contoh:

Phone no:	*3-group*	*index*
7313018	73+13+018	104
5392309	53+92+309	454
4281397	42+81+397	520

- Penyebaran index hash table lebih baik daripada truncation

Hash Function: Modular Arithmetic

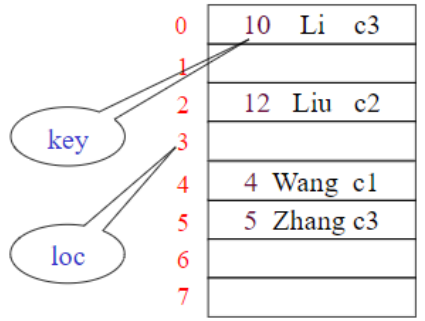
- Melakukan konversi data ke bentuk bilangan bulat, dibagi dengan ukuran hash table, dan mengambil hasil sisa baginya sebagai indeks
- Contoh: Ukuran hash table = 100

Phone no:	*2-group*	*index*
7313018	731+3018	3749 % 100 = 49
5392309	539+2309	2848 % 100 = 48
4281397	428+1397	1825 % 100 = 25

Hash function is H(key) = key mod 10,
The record set is:

No.	Name	Class	…
5	Zhang	c1	
12	Liu	c2	
4	Wang	c1	
10	Li	c3	
…			

Then the hash table will be:



- Metode paling baik, karena memberikan penyebaran index yang baik dan dapat memastikan berada di interval tertentu

Bagaimana Hash Function untuk String?

- Konversi non integer key ke nilai integer, kemudian gunakan hash function untuk memetakannya ke index hash table

Contoh:

- Misal string = HAI
- Ambil nilai ASCII dari masing – masing huruf (H,A,I)
- Petakan menggunakan hash function, misal:

H = 72 % 6 = 0

A = 65 % 6 = 5

I = 73 % 6 = 1

Index array

Integer key

000 (nul)	016 ► (dle)	032 sp	048 0	064 @	080 P	096 `	112 p
001 ☺ (soh)	017 ◄ (dc1)	033 !	049 1	065 A	081 Q	097 a	113 q
002 ☻ (stx)	018 ↕ (dc2)	034 "	050 2	066 B	082 R	098 b	114 r
003 ♥ (etx)	019 ‼ (dc3)	035 #	051 3	067 C	083 S	099 c	115 s
004 ♦ (eot)	020 ¶ (dc4)	036 $	052 4	068 D	084 T	100 d	116 t
005 ♣ (enq)	021 § (nak)	037 %	053 5	069 E	085 U	101 e	117 u
006 ♠ (ack)	022 – (syn)	038 &	054 6	070 F	086 V	102 f	118 v
007 • (bel)	023 ↨ (etb)	039 '	055 7	071 G	087 W	103 g	119 w
008 ◘ (bs)	024 ↑ (can)	040 (	056 8	072 H	088 X	104 h	120 x
009 (tab)	025 ↓ (em)	041)	057 9	073 I	089 Y	105 i	121 y
010 (lf)	026 (eof)	042 *	058 :	074 J	090 Z	106 j	122 z
011 ♂ (vt)	027 ← (esc)	043 +	059 ;	075 K	091 [	107 k	123 {
012 ♀ (np)	028 ∟ (fs)	044 ,	060 <	076 L	092 \	108 l	124 |
013 (cr)	029 ↔ (gs)	045 -	061 =	077 M	093]	109 m	125 }
014 ♫ (so)	030 ▲ (rs)	046 .	062 >	078 N	094 ^	110 n	126 ~
015 ☼ (si)	031 ▼ (us)	047 /	063 ?	079 O	095 _	111 o	127 ⌂

Latihan

Bagaimana pemetaan key berikut ke index array hash table?

1. Hash (x) = x mod 10, Key : 45, 72, 39, 48, 56, 77, 91, 63, 84, 90
2. Hash (x) = x mod 6, Key : 45, 72, 40, 49, 14

1.

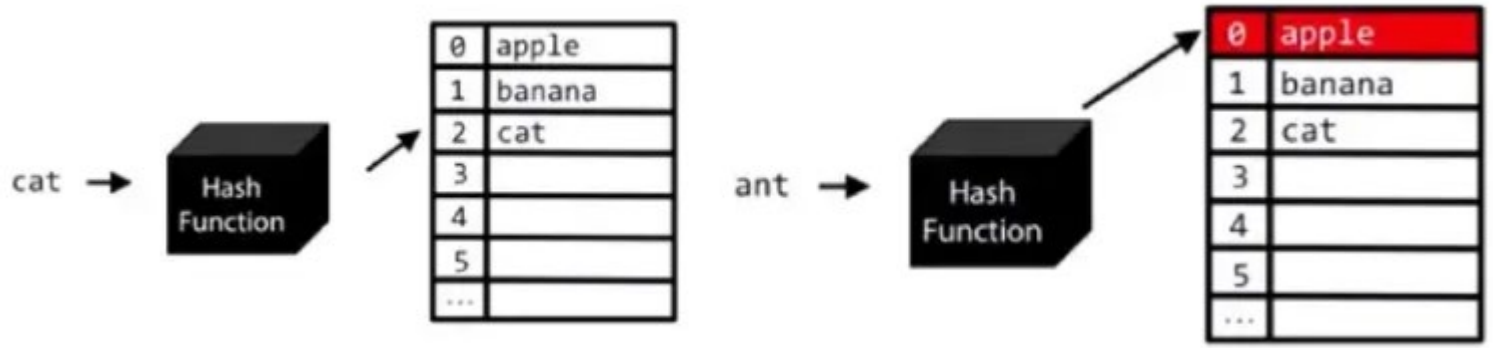
0	90
1	91
2	72
3	63
4	84
5	45
6	56
7	77
8	48
9	39

2.

0	72
1	49
2	14
3	45
4	40
5	

Collision Resolution

- **Collision Resolution**: Penyelesaian bila terjadi *collision* (tabrakan)
- Dikatakan terjadi *collision* jika dua buah keys dipetakan pada sebuah sel (index array yang sama)
- *Collision* bisa terjadi saat melakukan insertion dan dibutuhkan prosedur tambahan untuk mengatasi terjadinya *collision*
- Dua strategi umum:
  - **Closed Hashing (Open Addressing)**
  - **Open Hashing (Chaining)**



Collision Resolution: Closed Hashing (Open Addressing)

- Ide: mencari alternatif sel lain pada hash table
- Pada proses insertion, coba sel lain sesuai urutan dengan menggunakan fungsi pencari urutan seperti berikut:

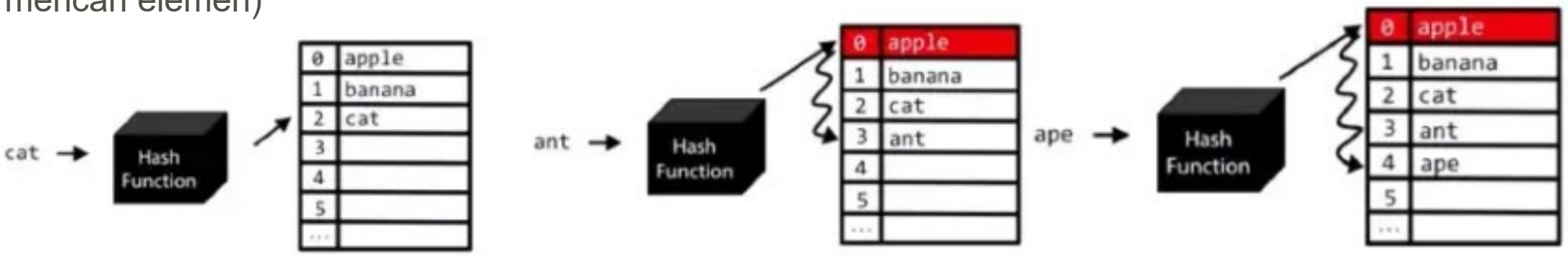
$$h_i(x) = (\text{hash}(x) + \mathbf{f(i)}) \bmod \text{H-size}$$

- Fungsi **f** digunakan sebagai pengatur strategi collision resolution. Bagaimana bentuk fungsi **f** ?

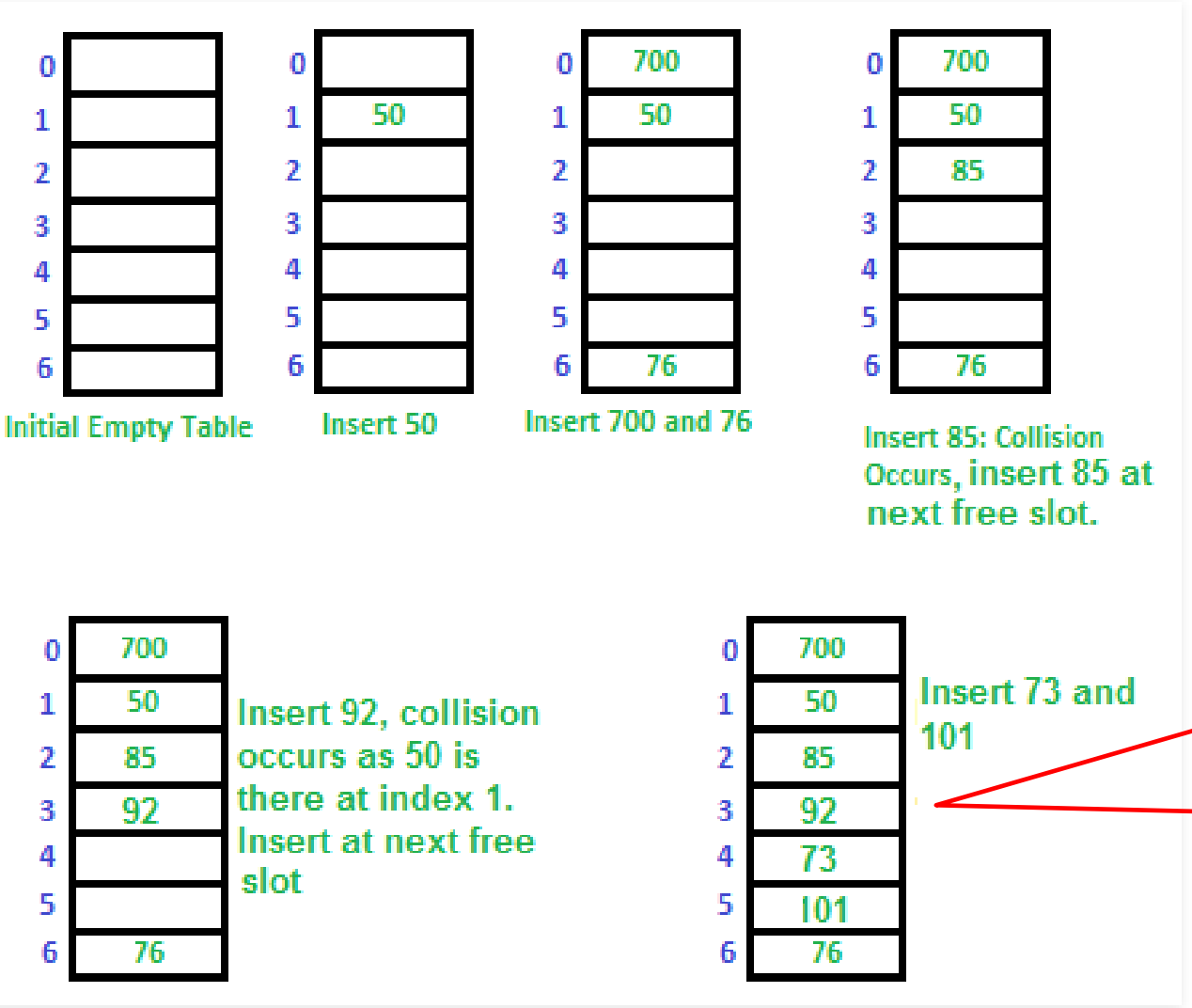
  Ada 3 strategi: **Linear probing, Quadratic probing, Double hashing**

Linear Probing

- Gunakan fungsi linear: $f(i) = i$
- Bila terjadi collision, cari posisi pertama pada tabel yang terdekat dengan posisi yang seharusnya. Telusuri table di sampingnya sampai ada tempat yang kosong. Dimulai dari nilai yang didapat dari hash function (array hash table dianggap circular, sehingga jika sampai akhir array dia akan balik lagi ke awal index 0)
- Fungsi linear relatif paling sederhana, mudah diimplementasikan
- Dapat menimbulkan masalah → **primary clustering** (elemen-elemen yang menurut perhitungan hash diletakkan pada lokasi sel berbeda, diarahkan pada sel pengganti yang sama, sehingga membentuk cluster, sehingga membutuhkan waktu yang lama untuk mencari sel kosong untuk pengganti atau ketika mencari elemen)



Linear Probing (contoh: H(key) = key mod 7 untuk 50, 700, 76, 85, 92, 73, 101)



Linear Probing hanya disarankan untuk ukuran hash table yang ukurannya lebih besar dua kali dari jumlah data

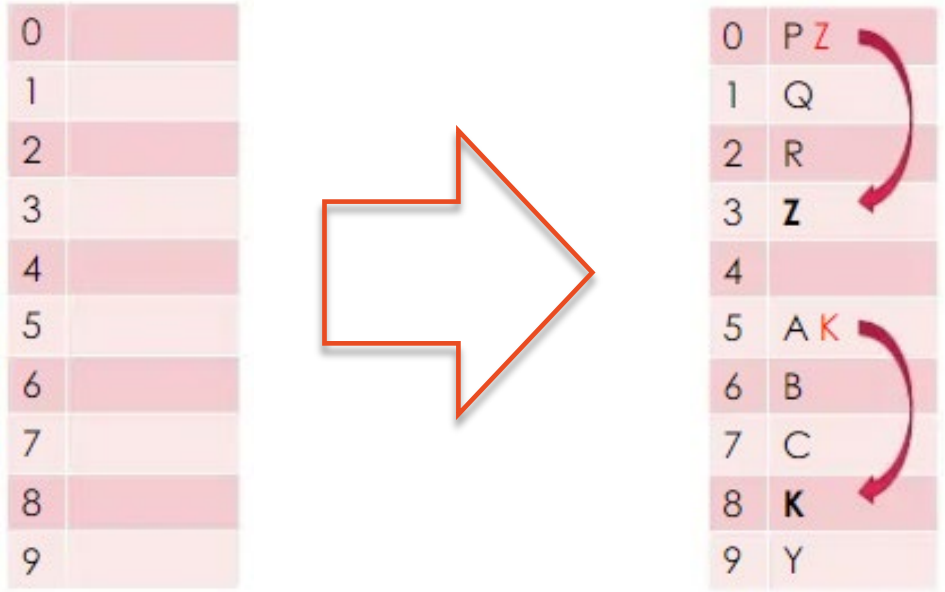
Membentuk suatu group (cluster), sehingga membutuhkan waktu lama untuk mencari sel kosong karena harus mengecek satu – satu sel terdekat (**Primary Clustering**)

Latihan

- Misal terdapat data dan hash function sbb:

Rekaman	A	B	C	K	P	Q	R	Y	Z
H(k)	5	6	7	5	0	1	2	9	0

Data diatas dimasukkan ke dalam urutan yang sama. Bagaimana pemetaan ke hash table di bawah!



Latihan

- Fungsi hash yang dipakai adalah f(key) = key mod 10
- Ruang alamat yang tersedia adalah 10 alamat
- Key: 20, 31, 33, 40, 10, 12, 30, dan 15
- Metode collision resolution yang dipakai adalah Open Addressing **dengan jarak probe 3**. Bagaimana pemetaannya?

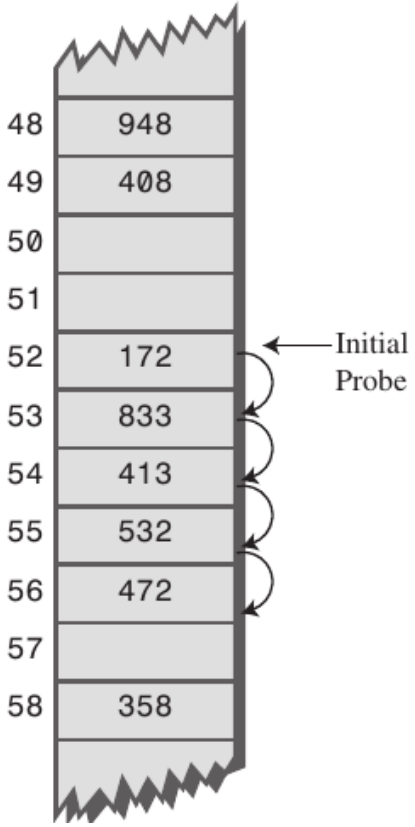
Akan mengecek f(key),
f(key) + 3, f(key) + 6, dst

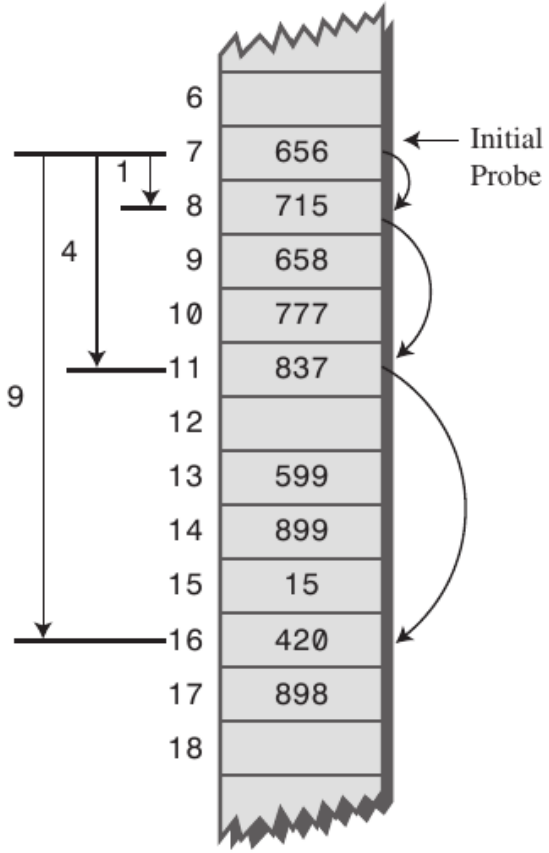
0	20
1	31
2	12
3	33
4	
5	30
6	40
7	
8	15
9	10

Quadratic Probing

- Menghindari primary clustering dengan menggunakan fungsi: $f(i) = i^2$
- Telusuri table dengan men-skip index sesuai nilai kuadratik (1, 4, 9, 16, …)

Linear Probing





Quadratic Probing

Bisa terjadi **secondary clustering**: cluster – cluster terbentuk di tempat – tempat dilaluinya quadratic probing yaitu f(x) + 1, f(x) + 4, f(x) + 9, dst

Quadratic Probing

Table Size is 11 (0..10)

Hash Function: **h(x) = x mod 11**

Insert keys: 20, 30, 2, 13, 25, 24, 10, 9

- 20 mod 11 = 9
- 30 mod 11 = 8
- 2 mod 11 = 2
- 13 mod 11 = 2 → $2+1^2=3$
- 25 mod 11 = 3 → $3+1^2=4$
- 24 mod 11 = 2 → $2+1^2$, $2+2^2=6$
- 10 mod 11 = 10
- 9 mod 11 = 9 → $9+1^2$, $9+2^2$ mod 11, $9+3^2$ mod 11 =7

0	
1	
2	2
3	13
4	25
5	
6	**24**
7	**9**
8	30
9	20
10	10

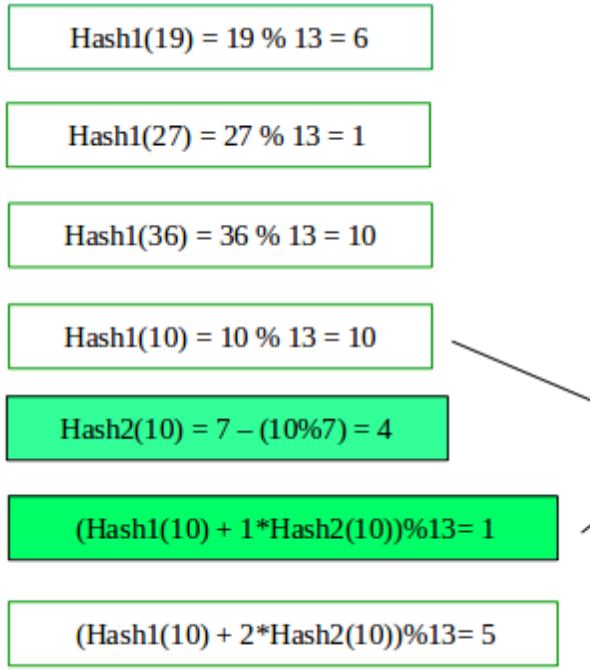
Double Hashing

- Double hashing dapat menemukan slot kosong lebih cepat daripada **linear probing**
- Menggunakan 2 fungsi hash, sehingga menjadi sebagai berikut:

  (hash1(key) + i * hash2(key)) % TABLE_SIZE

Hash1 (key) = key % 13

Hash2 (key) = 7 – (key % 7)



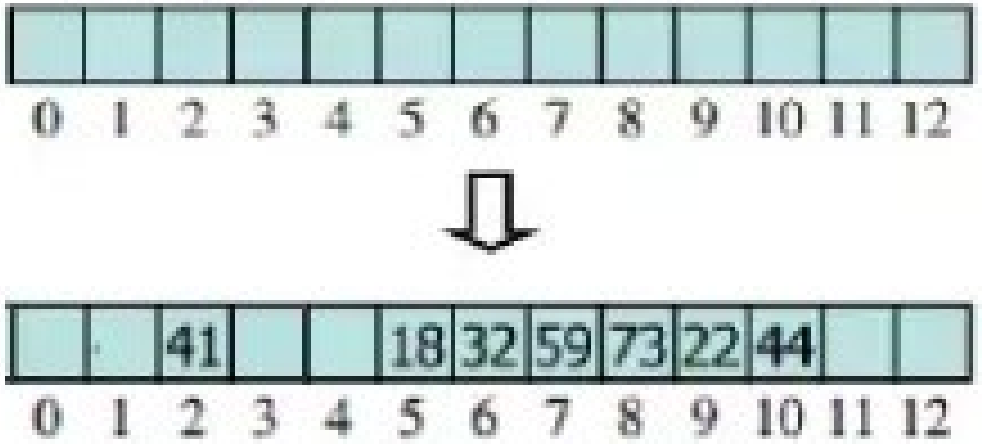
- Cek lokasi **hash1(key)** apakah kosong. Jika kosong, masukan.
- Jika tidak kosong, hitung **hash2(key).** Cek apakah **(hash1(key) + hash2(key))%size** kosong
- Jika tidak kosong, cek berulang untuk:
  - **(hash1(key) + 2*hash2(key))%size,**
  - **(hash1(key) + 3*hash2(key))%size,**
  - **(hash1(key) + 4*hash2(key))%size,** dst sampai menemukan index yang kosong

Latihan

- $N = 13$
- $h(k) = k \bmod 13$
- $d(k) = 7 - k \bmod 7$

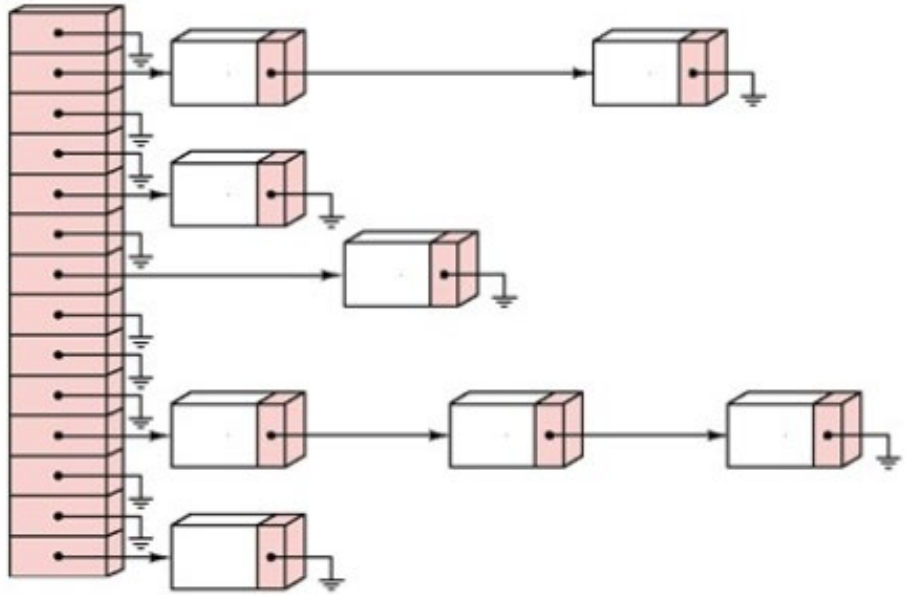
Untuk key 44:
- h(k) = 5 → sudah ada isinya yaitu 18
- d(k) = 5 → (h(k) + d(k)) mod 13 = 10 → kosong, masukkan 44 ke index 10

Insert keys 18, 41, 22, 44, 59, 32, 73, in this order



Collision Resolution: Open Hashing (Chaining)

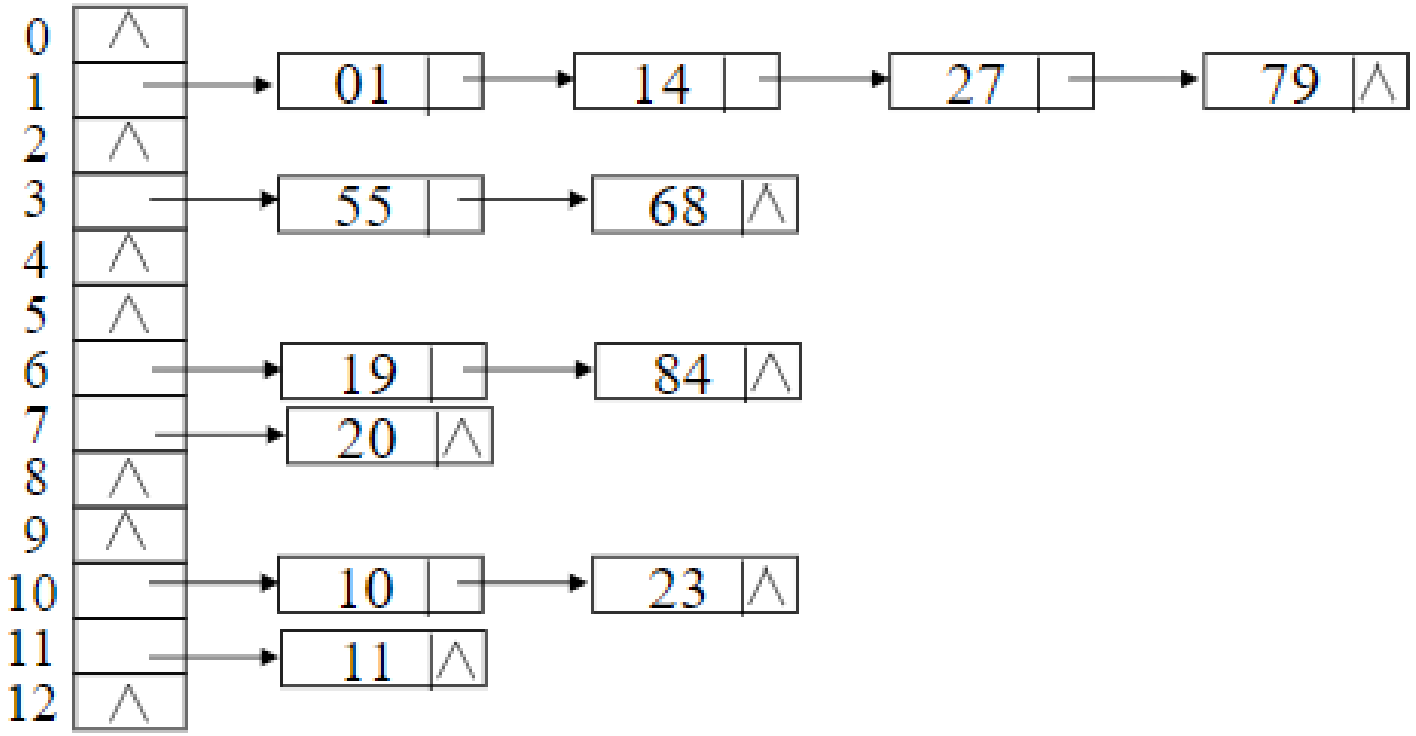
- Permasalahan Collision diselesaikan dengan menambahkan seluruh elemen yang memilih nilai hash sama pada sebuah set
- Open Hashing:
  - Menyediakan sebuah linked list untuk setiap elemen yang memiliki nilai hash sama
  - Tiap sel pada hash table berisi pointer ke sebuah linked list yang berisikan data/elemen



Collision Resolution: Open Hashing (Chaining)

Contoh:

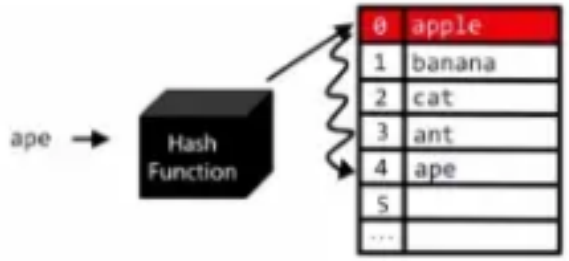
- Key: 19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79
- H(key) = key mod 13
- Hash table yang terbentuk menjadi:



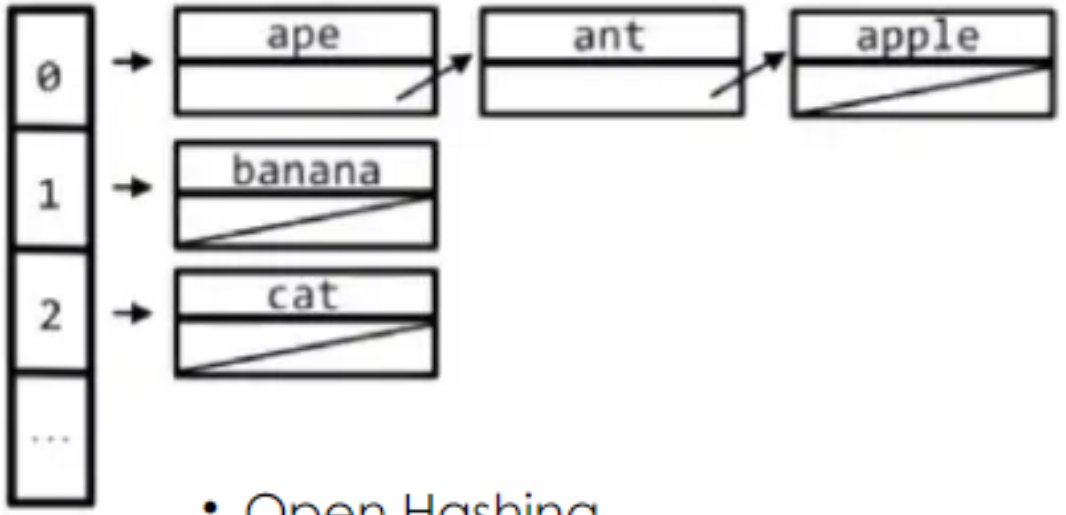
Collision Resolution: Open Hashing (Chaining)

- Untuk pencarian, gunakan fungsi hash untuk menentukan linked list mana yang memiliki elemen yang dicari, kemudian lakukan pembacaan terhadap linked list tersebut
- Dapat juga digunakan struktur data lain selain linked list untuk menyimpan elemen yang memiliki fungsi hash yang sama tersebut
- Kelebihan utama dari metode ini adalah dapat menyimpan data yang tak terbatas (*dynamic expansion*)
- Kekurangan utama adalah penggunaan memori pada setiap sel lebih banyak

Open vs Closed Hashing



- Closed Hashing



- Open Hashing

Hash Table VS Array dan Linked List

- Array
  - Ukuran tetap (fix)
  - Banyak memori tidak terpakai
  - Boros ruang memori
- Linked List
  - Ukuran dinamis
  - Hemat memori
  - Boros waktu (waktu yang dipakai untuk search lama)

- **Hash Table**
  - \+ Waktu aksesnya yang cukup cepat, jika record yang dicari langsung berada pada angka hash lokasi penyimpanannya
  - \+ Hashing relatif lebih cepat
  - \+ Kecepatan dalam insert, delete, maupun search relatif sama
  - \- Sering sekali ditemukan hash table yang record – recordnya mempunyai angka hash yang sama
  - \- Tidak efisien untuk mencetak seluruh elemen pada hash table
  - \- Tidak efisien untuk mencari elemen minimum dan maksimum

Implementasi Hash Table: Insert

- Untuk menambahkan data baru, key perlu dikonversi menjadi index array hash table menggunakan salah satu fungsi hash

  Contoh:

  - Fungsi untuk memetakan NIP Pegawai/key adalah **hash(key) = key mod 701**

    NIP Pegawai 580625685 → maka **hash(580625685) = 580625685 mod 701 = 3**

    (Jika terjadi collision, maka bisa diimplementasikan collision resolution)

  - Tambahkan data baru (nama, tanggal lahir, alamat, dll) pada array hash table index ke- 3

Implementasi Hash Table: Search

- Hitung nilai hash value/index array menggunakan fugsi hash yang digunakan untuk operasi insert (termasuk juga implementasi collision resolution-nya)
- Baca nilai array pada index tersebut

Implementasi Hash Table: Delete

- Data dapat dihapus dari hash table
- Tetapi index array/lokasi yang dihapus datanya tidak boleh dibiarkan sebagai lokasi yang kosong karena akan mempengaruhi hasil proses search
- Index array/lokasi tersebut harus ditandai bahwa lokasi tersebut sebelumnya ada isinya. Karena jika kita biarkan dia kosong setelah penghapusan, jika kita mau search suatu elemen A, akan mengubah proses atau alur awal ketika kita insert elemen A.

Latihan

1. Bagaimana implementasi insert dan delete antrian berprioritas dari **Min Heap** dengan root index = 0? (customize fungsi yang diberikan)

*__Max Heap__ = setiap node nilainya >= nilai anak – anaknya, root paling besar nilainya → antrian berprioritas

*__Min Heap__ = setiap node nilainya <= nilai anak – anaknya, root paling kecil nilainya

2. Key 12, 18, 13, 2, 3, 23, 5 dan 15 dimasukan ke dalam Hash Table kosong ukuran 10 menggunakan hash function h(k) = k mod 10 dan linear probing. Hasil dari pemetaan key ke index array Hash Table adalah

TERIMA KASIH