Algoritme dan Struktur Data

Map, Hashing, & Hashmap/Hashtable

Putra Pandu Adikara

Universitas Brawijaya

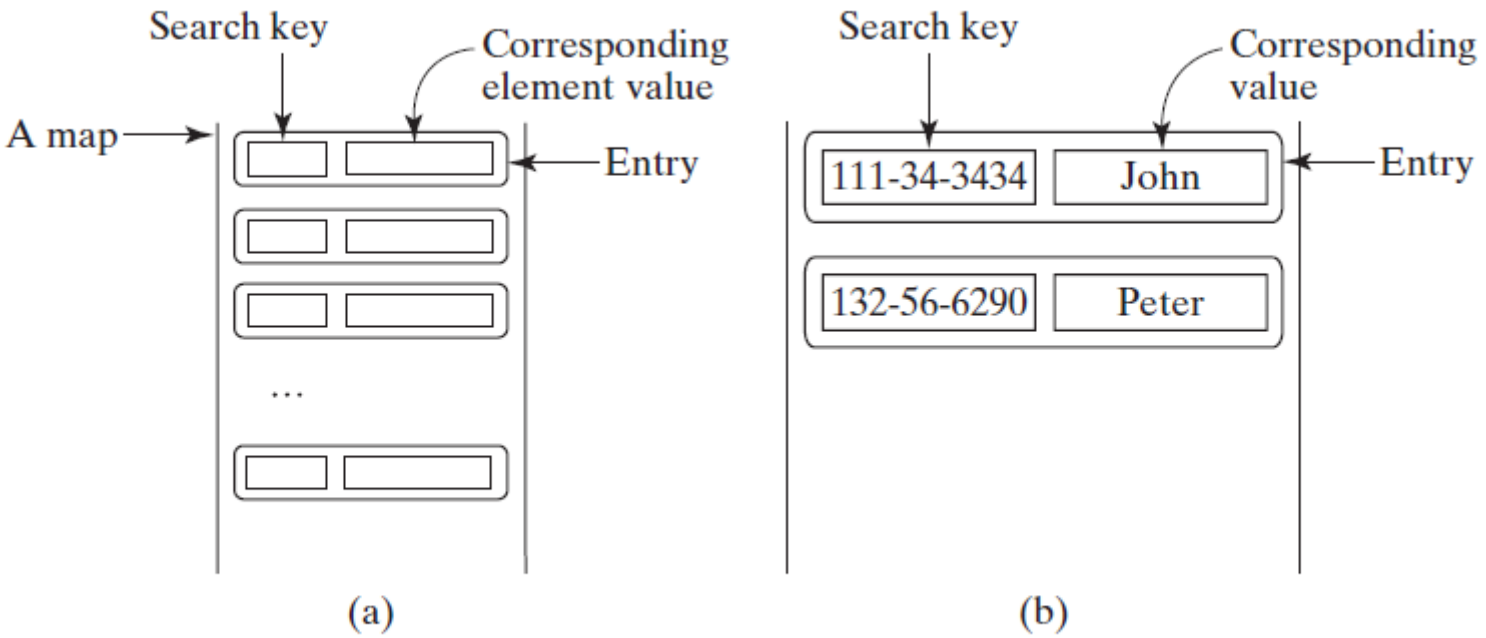
Outline

- Hashing
  - Definition
  - Hash function
  - Collition resolution
    - Open hashing
      - Separate chaining
    - Closed hashing (Open addressing)
      - Linear probing
      - Quadratic probing
      - Double hashing
    - Primary Clustering, Secondary Clustering
  - Access: insert, find, delete

Map/Dictionary/Hashtable

- **Map** adalah container yang menyimpan sekumpulan (*collection*) pasangan kunci *K* dan nilai *V* (key/value pairs)
  - Nama lain: Key-Value Pair, Associative Array, Dictionary, Hashtable, symbol table
- Map memungkinkan pencarian/pengaksesan, penghapusan, dan pembaruan nilai data dengan sangat cepat melalui key-nya → O(1)
- Map menyimpan nilai yang sesuai dengan key-nya
- Key ini seperti indeks pada array
  - Indeks pada array berupa angka 0..n-1
  - Key pada map bisa sembarang objek
- Map tidak memperbolehkan key yang sama → harus unik, sehingga satu kunci untuk satu nilai

Isi map berupa key dan value



21.2 The entries consisting of key/value pairs are stored in a map.

- Sumber: Liang, Introduction to Java Programming

Jenis-Jenis Map

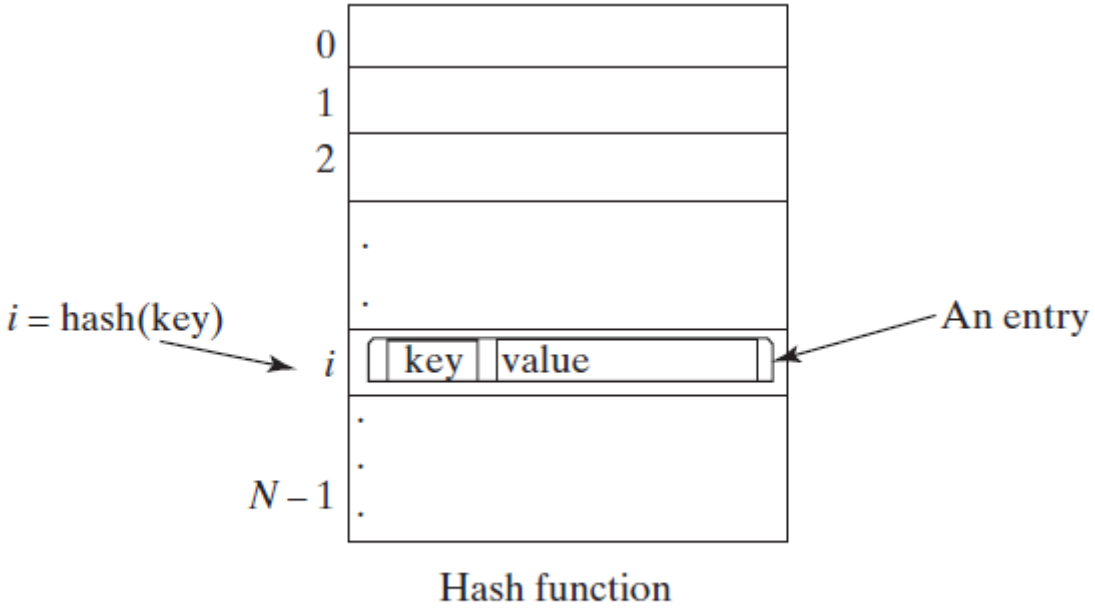
- Di Java API, ada beberapa jenis map (*concrete class*):
  - **`HashMap`** → efisien untuk mencari lokasi, menambah, menghapus entry (tidak terurut)
  - **`LinkedHashMap`** → menggunakan linked-list dan mendukung urutan entry dalam map (terurut)
  - **`TreeMap`** → efisien untuk penelusuran key yang terurut (dapat diurutkan dengan `Comparable` atau `Comparator`)

- Map ini diturunkan dari `AbstractMap` (*abstract class*)
  - Di Java, ada class Hashtable dan Map, yang sedikit perbedaan dalam implementasinya, namun kegunaannya sama.

Hashing

- Hashtable merupakan array yang menyimpan nilai data dan key yang bersesuaian dengan data dan ukurannya telah ditentukan
  - Ukuran Hash table (H_{size}), biasanya lebih besar dari jumlah data yang hendak disimpan.
- Hash function/fungsi Hash adalah fungsi yang memetakan kunci ke indeks pada hash table.
- Hashing adalah teknik mendapatkan nilai menggunakan indeks yang didapat dari key tanpa melakukan penelusuran dari awal s.d. akhir
- Teknik Hashing memungkinkan operasi pencarian, penambahan, pemutakhiran, penghapusan dengan **constant average time**
- Load factor (λ) adalah perbandingan antara data yang disimpan dengan ukuran hash table.

Hashing dan Fungsi Hash



A hash function maps a key to an index in the hash table.

- Bagaimana kita mendesain fungsi yang menghasilkan indeks dari sebuah key?
- Pemetaan yang ideal satu key ke satu indeks disebut perfect hash function

Fungsi Hash

- Untuk setiap key, fungsi hash memetakan key pada bilangan dalam rentang 0 – hingga H_{size}-1
- Fungsi hash harus memiliki sifat:
  - Mudah dihitung
  - Dua key yang berbeda, dipetakan pada dua sel berbeda pada array
    - Kondisi ideal ini sulit tercapai, kecuali semesta dari key relatif kecil sehingga dapat dicapai dengan menggunakan direct-address table
  - Membagi key secara rata pada seluruh sel (*uniform*)
- Fungsi hash sederhana memanfaatkan fungsi mod dengan bilangan prima

Fungsi Hash: Truncation

- Sebagian dari key *K* dibuang/diabaikan, bagian sisanya digabung untuk membentuk index *h(K)*
- Contoh:

	Phone no.	**Index**
Ambil *n* tertentu	577-9112	792
	212-8790	180
Ambil *n* terakhir	577-9112	112
	212-8790	790

- Bukan metode yang sangat baik: tidak bisa mendistribusikan key secara merata

Fungsi Hash: Folding

- Membagi data menjadi beberapa bagian
- Tiap bagian digabung ke bentuk lain (mis. dengan penambahan, pengurangan, dan/atau perkalian)
- Contoh:

Phone no.	**3-group**	**Index**
5779112	57+79+112	248
2128790	21+28+790	839

Fungsi Hash: Modular Arithmetic

- Konversi data ke bilangan bulat, dibagi dengan ukuran hash table, dan ambil hasil sisa bagi sebagai indeks
- Contoh: ukuran hash table 100

Phone no.	**2-group**	**Index**
5779112	577+9112	9689 % 100 = 89
2128790	212+8790	9002 % 100 = 2

Pemilihan Fungsi Hash

- Masih ada beberapa fungsi hash lain
  - Division
  - Middle Squaring
  - dll.
- Fungsi hash yang baik memiliki dua kriteria:
  - Cepat dihitung
  - Meminimalkan *collision* yang terjadi
- Collision: jika dua kunci berbeda $K_1 \neq K_2$, dipetakan ke alamat tabel yang sama $h(K_1) = h(K_2)$
  - 577-9112 → 792
  - 172-9452 → 792

Bagaimana mengatasinya?

Contoh Fungsi Hash

- Fungsi Hash untuk string
  - X = 128
  - $A_3 X^3 + A_2 X^2 + A_1 X^1 + A_0 X^0$
  - $(((A_3 X) + A_2) X + A_1) X + A_0$
- Hasil dari fungsi hash jauh lebih besar dari ukuran table, sehingga perlu di-modulo dengan ukuran hash table.

Contoh Fungsi Hash

```
int hash(String key, int tableSize) {
    int hashVal = 0;
    for (int i=0; i < key.length(); i++) {
        hashVal = (hashVal * 128
                   + key.charAt(i)) % tableSize;
    }
    return hashVal % tableSize;
}
```

- Modulo
  - (A + B) % C = (A % C + B % C) % C
  - (A * B) % C = (A % C * B % C) % C

Contoh Fungsi Hash

```
int hash(String key, int tableSize) {
    int hashVal = 0;
    for (int i=0; i < key.length(); i++) {
        hashVal = (hashVal * 37
                + key.charAt(i));
    }
    hashVal %= tableSize;
    if (hashVal < 0) {
        hashVal += tableSize;
    }
    return hashVal;
}
```

Contoh Fungsi Hash

```
int hash(String key, int tableSize) {
    int hashVal = 0;
    for (int i=0; i < key.length(); i++) {
        hashVal += key.charAt(i)
    }
    return hashVal % tableSize;
}
```

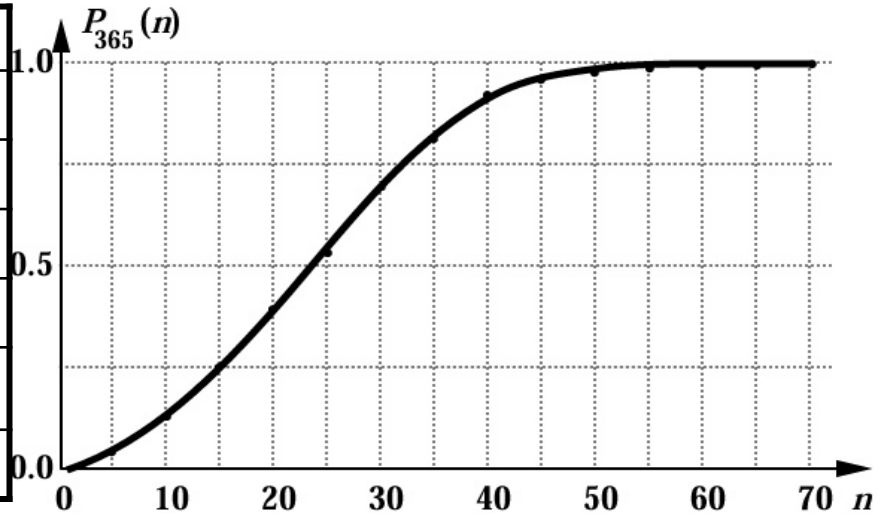
Collision Resolution Policy

- Collision resolution: penyelesaian bila terjadi collision (tabrakan)
- Collision terjadi bila dua key menempati/dipetakan pada sel tabel (indeks) yang sama
- Collision terjadi saat *insertion*
- Untuk mengatasi collision, diperlukan prosedur tambahan
- Strategi yang umum:
  - Closed Hashing (Open Addressing)
  - Open Hashing (Chaining)

Seberapa sering collision terjadi?

- Von Mises Birthday Paradox:
  - Apabila ada lebih dari 23 orang di suatu ruangan, kemungkinan dua orang atau lebih yang memiliki tanggal lahir yang sama lebih besar dari 50% (threshold)
  - Peluang mencapai 100% bila ada 367 orang (ada 366 kemungkinkan tanggal lahir termasuk 29 Feb)
  - 99,9% tercapai hanya dengan ~60-70 orang

n	%	$P_{365}(n)$
10	2.7	0.1169
20	5.5	0.4114
30	8.2	0.7063
40	11.0	0.8912
50	13.7	0.9704
60	16.4	0.9941



Closed Hashing (Open Addressing)

- Ide: mencari alternatif indeks lain pada tabel.
- Pada proses insertion, coba indeks lain sesuai urutan dengan menggunakan fungsi pencari urutan seperti berikut:
  - $h_i(x) = (\text{hash}(x) + f(i)) \bmod H_{size}$ $\qquad f(0) = 0$
- Fungsi *f* digunakan sebagai pengatur strategi collision resolution.
- Bagaimana bentuk fungsi *f*?

Closed Hashing (Open Addressing)

- Ada beberapa strategi untuk menentukan fungsi f:
  - Linear probing
  - Quadratic probing
  - Double hashing

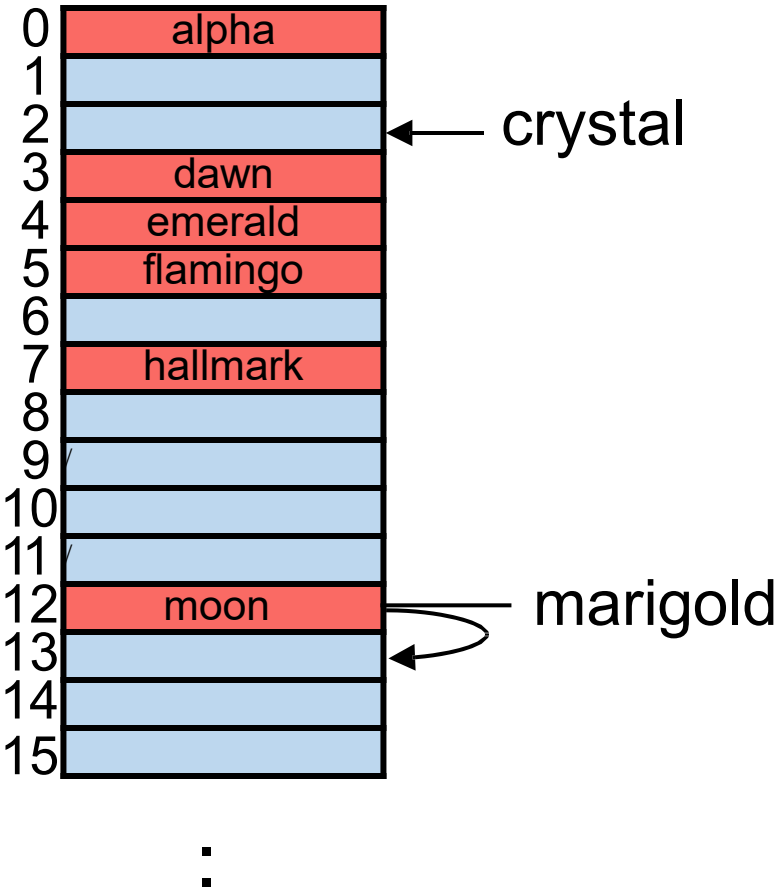
Linear Probing

- Bila terjadi collision, cari posisi lokasi terdekat berikutnya yang tersedia secara sekuensial:
  - $f(i) = i$
- Misal: jika ada collision pada hashTable[k % N], cek apakah hashTable[k + 1 % N] tersedia, jika tidak cek hashTable[k + 2 % N], dst. hingga ditemukan.
- Fungsi linear sederhana, mudah diimplementasikan
- **Masalah:**
  - **primary clustering:** Banyak elemen berurutan membentukan grup/klaster karena perhitungan hash diarahkan pada sel pengganti yang sama → mulai butuh waktu menemukan slot kosong/cari elemen

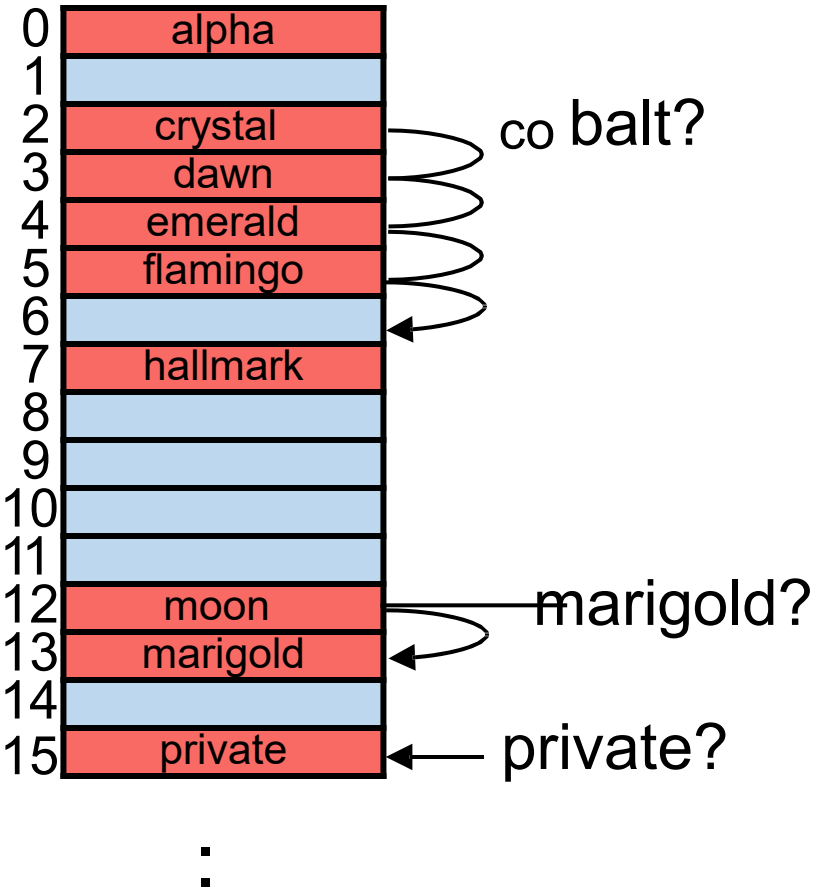
Linear Probing

- Kompleksitas teknik ini bergantung pada nilai Load factor (λ)
- Definisi ***Load factor (λ)***
  - Untuk hash table ***T*** dengan ukuran ***m***, yang menyimpan ***n*** data
  - **Load factor (λ)** dari **T** adalah **n/m**
- **Linear Probing** tidak disarankan bila: $\lambda > 0{,}5$
- **Linear Probing** hanya disarankan bila ukuran hash table ukurannya lebih besar dua kali dari jumlah data

Hashing - insert

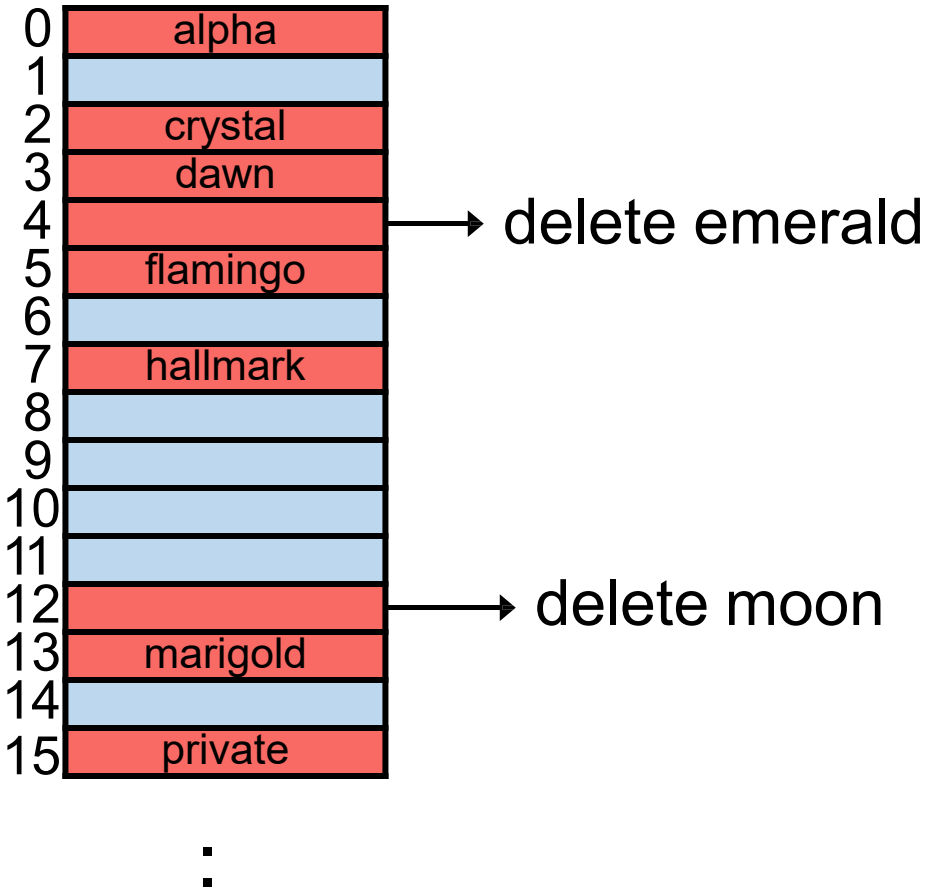


Hashing - lookup

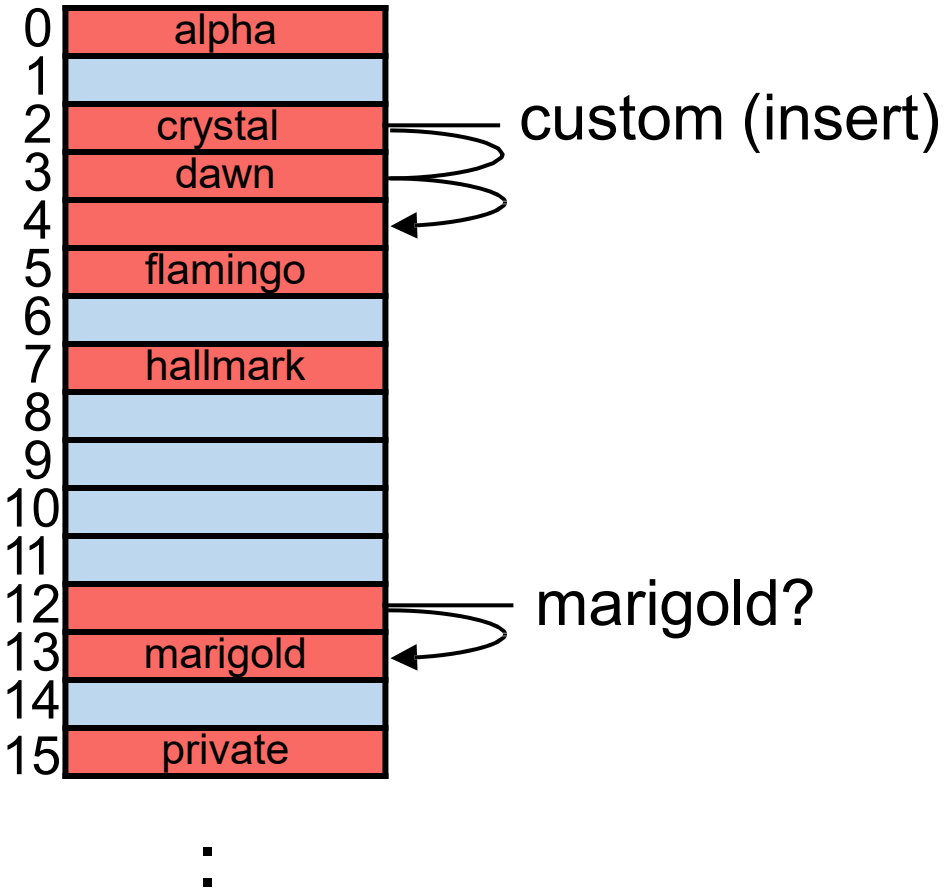


Hashing - delete

- lazy deletion
- mengapa?

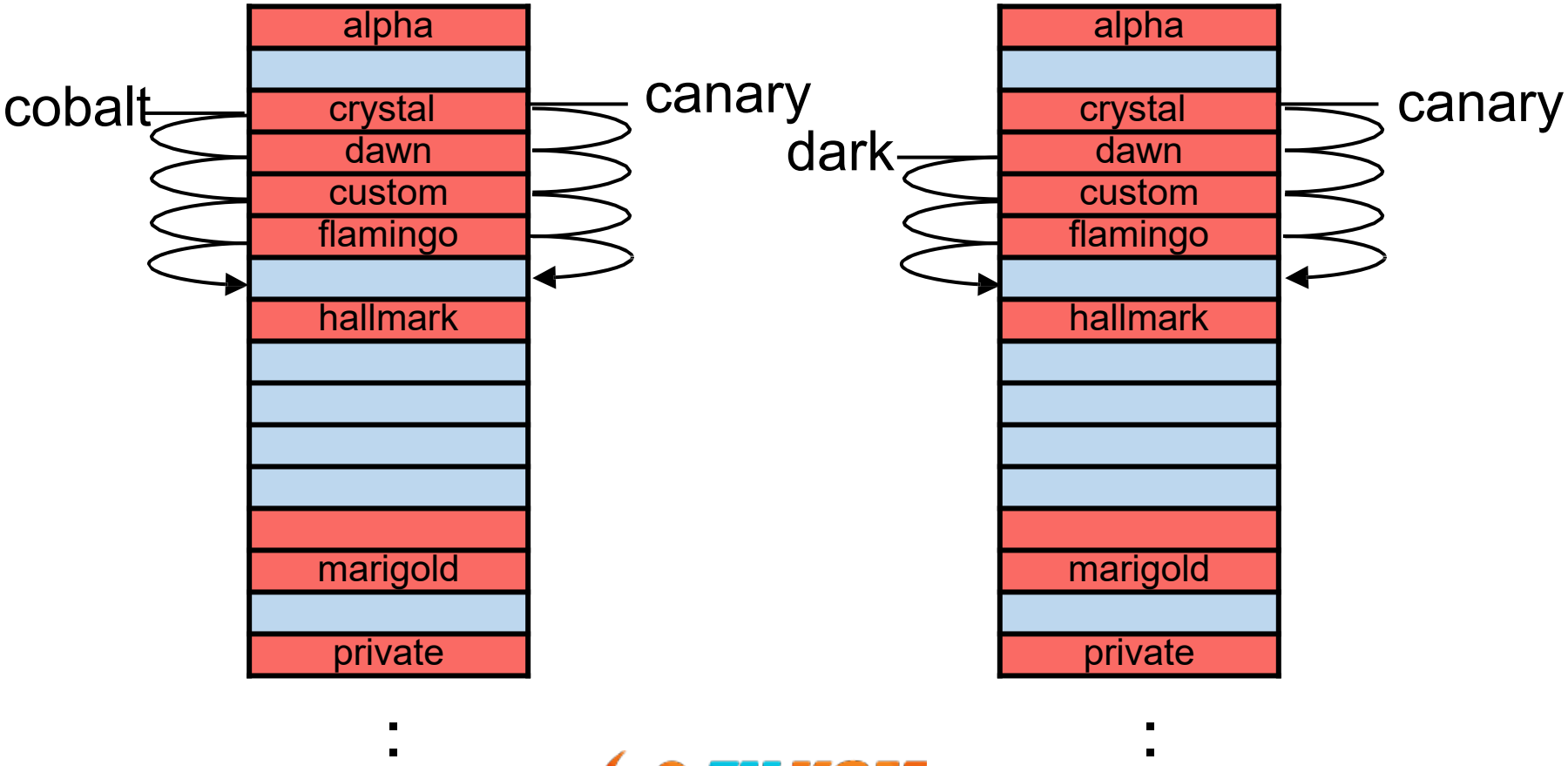


Hashing - operation after delete



Primary Clustering

- Elemen-elemen yang menurut perhitungan hash diletakkan pada lokasi sel berbeda, diarahkan (probe) pada sel pengganti yang sama.



Quadratic Probing

- Untuk menghindari primary clustering menggunakan fungsi:
  - $f(i) = i^2$
- Menimbulkan banyak permasalahan bila hash table telah terisi lebih dari setengah.
- Perlu dipilih ukuran hash table yang bukan bilangan kuadrat.
- Dengan ukuran hash table yang merupakan bilangan prima dan hash table yang terisi kurang dari setengah, strategy quadratic probe dapat selalu menemukan lokasi untuk setiap elemen baru.

Quadratic Probing

- Dapat melakukan *increment* bila terjadi *collision*
- Perhatikan bahwa fungsi *quadratic* dapat dijabarkan sebagai berikut:
  - $f(i) = i^2 = f(i\text{-}1) + 2\,i - 1.$
- Menimbulkan second clustering:
  - Elemen-elemen yang menurut perhitungan hash diletakkan pada lokasi sel sama, diarahkan pada sel pengganti yang sama.

Double hashing

- fungsi untuk *collision resolution* disusun dengan fungsi hash seperti:
  - $\mathbf{f(i) = i * hash2\ (x)}$
- Setiap saat faktor *hash2(x)* ditambahkan pada *probe*.
- Harus hati-hati dalam memilih fungsi hash kedua untuk menjamin agar tidak menghasilkan nilai 0 dan mem-probe ke seluruh sel.
- Salah satu syaratnya ukuran hash table haruslah bilangan prima.

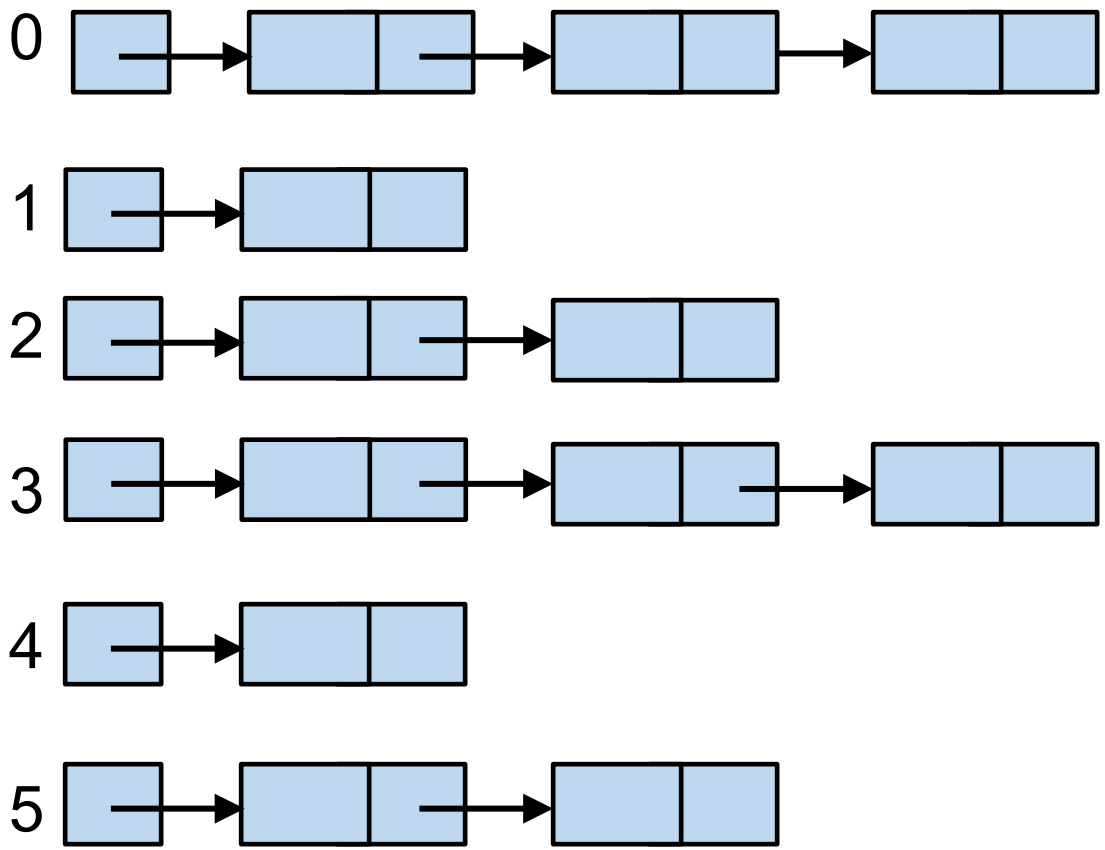
Open Hashing (Chaining)

- 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.

Open Hashing

- Fungsi dan analisis Open Hashing:
  - Menambahkan sebuah elemen ke dalam tabel:
    - Dilakukan dengan menambahkan elemen pada akhir atau awal linked-list yang sesuai dengan nilai hash.
  - Bergantung apakah perlu ada pengujian nilai duplikasi atau tidak.
  - Dipengaruhi berapa sering elemen terakhir akan diakses.

Open Hashing



Open Hashing

- Untuk pencarian, gunakan fungsi hash untuk menentukan linked list mana yang memiliki elemen yang dicari, kemudian lakukan pembacaan terhadap *linked list* tersebut.
- Penghapusan dilakukan pada *linked list* setelah pencarian elemen dilakukan.
- Dapat saja 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 tiap sel.

Analisis Open Hash

- Secara umum panjang dari linked list yang dihasilkan sejalan dengan nilai λ.
- Kompleksitas insertion bergantung pada fungsi hash dan insertion pada linked-list.
- Untuk pencarian, kompleksitasnya adalah waktu konstan dalam mengevaluasi fungsi hash + pembacaan list.
- Worst case O(n) untuk pencarian.
- Average case bergantung pada λ.
- Aturan umum untuk open hashing adalah untuk menjaga agar:
- $\lambda \approx 1$.
- Digunakan untuk data yang ukurannya dinamis.

Isu-isu lain

- Hal-hal lain yang umum dan perlu diperhatikan pada metode *closed hashing resolutions*:
  - Proses menghapus agak membingungkan karena tidak benar-benar dihapus.
  - Secara umum lebih sederhana dari pada open hashing.
  - Bagus bila diperkirakan tidak akan terjadi banyak collision.
  - Jika pencarian berdasarkan fungsi hash gagal, kemungkinan harus mencari/membaca seluruh tabel.
  - Menggunakan ukuran table yang lebih besar dari data yang diharapkan.

Perbandingan Kinerja ADT

Operation	Representation		
	Sorted array	AVL tree	Hash table
Initialize	$O(N)$	$O(1)$	$O(N)$
Is full?	$O(1)$	$O(1)$	$O(1)$
Search*)	$O(\log N)$	$O(\log N)$	$O(1)$
Insert	$O(N)$	$O(\log N)$	$O(1)$
Delete	$O(N)$	$O(\log N)$	$O(1)$
Enumerate	$O(N)$	$O(N)$	$O(N \log N)$**)

*) also: **Retrieve, Update** **)To enumerate a hash table, entries must first be sorted in ascending order of keys that takes $O(N \log N)$ time

Hashset

```
public class HashSet<AnyType> extends AbstractCollection<AnyType>
implements Set<AnyType> {
    /**
     * Construct an empty HashSet.
     */
    public HashSet() {
        allocateArray(DEFAULT_TABLE_SIZE);
        clear();
    }
    /**
     * Construct a HashSet from any collection.
     */
    public HashSet(Collection<? extends AnyType> other) {
        allocateArray(nextPrime(other.size() * 2));
        clear();
        for (AnyType val: other)
            add(val);
    }
```

```
/**
 *
 * This method is not part of standard Java.
 * Like contains, it checks if x is in the set.
 * If it is, it returns the reference to the matching
 * object; otherwise it returns null.
 * @param x the object to search for.
 * @return if contains(x)  is false, the return value is null
 *         otherwise, the return value is the object that causes
 *         contains(x) to return true.
 */
public AnyType getMatch(AnyType x) {
    int currentPos = findPos(x);

    if (isActive(array, currentPos))
        return (AnyType) array[currentPos].element;
    return null;
}
```

```
/**
 * Tests if some item is in this collection.
 * @param x any object.
 * @return true if this collection contains an item equal to x.
 */
public boolean contains(Object x) {
    return isActive(array, findPos(x));
}

/**
 * Tests if item in pos is active.
 * @param pos a position in the hash table.
 * @param arr the HashEntry array (can be oldArray during rehash).
 * @return true if this position is active.
 */
private static boolean isActive(HashEntry[] arr, int pos) {
    return arr[pos] != null && arr[pos].isActive;
}
```

```
/**
 * Adds an item to this collection.
 * @param x any object.
 * @return true if this item was added to the collection.
 */
public boolean add(AnyType x) {
    int currentPos = findPos(x);
    if (isActive(array, currentPos))
        return false;
    if (array[currentPos] == null)
        occupied++;
    array[currentPos] = new HashEntry(x, true);
    currentSize++;
    modCount++;
    if (occupied > array.length / 2)
        rehash();
    return true;
}
```

```
/**
 * this inner class is needed to encapsulate the element
 * and provide the flag field required by the Hash Table
 */
private static class HashEntry {
    public Object element; // the element
    public boolean isActive; // false if marked deleted
    public HashEntry(Object e) {
        this(e, true);
    }
    public HashEntry(Object e, boolean i) {
        element = e;
        isActive = i;
    }
}
```

```
/**
 * Private routine to perform rehashing.
 * Can be called by both add and remove.
 */
private void rehash() {
    HashEntry[] oldArray = array;
    // Create a new, empty table
    allocateArray(nextPrime(4 * size()));
    currentSize = 0;
    occupied = 0;
    // Copy table over
    for (int i = 0; i < oldArray.length; i++)
        if (isActive(oldArray, i)) add((AnyType) oldArray[i].element);
}
/**
 * Internal method to allocate array.
 * @param arraySize the size of the array.
 */
private void allocateArray(int arraySize) {
    array = new HashEntry[nextPrime(arraySize)];
}
```

```
/**
 * Removes an item from this collection.
 * @param x any object.
 * @return true if this item was removed from the collection.
 */
public boolean remove(Object x) {
    int currentPos = findPos(x);
    if (!isActive(array, currentPos))
        return false;
    array[currentPos].isActive = false;
    currentSize--;
    modCount++;
    if (currentSize < array.length / 8)
        rehash();
    return true;
}
```

```
/** * Method that performs quadratic probing resolution.
 * @param x the item to search for.
 * @return the position where the search terminates.
 */
private int findPos(Object x) {
    int offset = 1;
    int currentPos = (x == null) ? 0 : Math.abs(x.hashCode() %
array.length);
    while (array[currentPos] != null) {
        if (x == null) {
            if (array[currentPos].element == null) break;
        } else if (x.equals(array[currentPos].element)) break;
        currentPos += offset;           // Compute ith probe
        offset += 2;
        if (currentPos >= array.length) // Implement the mod
            currentPos -= array.length;
    }
    return currentPos;
}
```

OPEN HASHING (CHAINING)

```
/**
 * this inner class is needed to encapsulate the element
 * and provide the next field to implement the linked-list chaining
 */
private static class HashEntry {
    public Object element; // the element
    public HashEntry next; // linked list chaining.
    public HashEntry(Object e) {
        this(e, null);
    }
    public HashEntry(Object e, HashEntry n) {
        element = e;
        next = n;
    }
}
```

```
/**
 * Adds an item to this collection.
 * @param x any object.
 * @return true if this item was added to the collection.
 */
public boolean add(AnyType x) {
    if (getMatch(x))
        return false;
    int currentPos = x.hashCode();
    array[currentPos] = new HashEntry(x, array[currentPost]);
    currentSize++;
    return true;
}
```

Hashmap

```
/**
 * Hash table implementation of the Map.
 */
public class HashMap<KeyType, ValueType>
    extends MapImpl<KeyType, ValueType> {
        /**
         * Construct an empty HashMap.
         */
        public HashMap() {
            super(new HashSet <Map.Entry<KeyType, ValueType>> ());
        }
        /**
         * Construct a HashMap with same key/value pairs as another map.
         * @param other the other map.
         */
        public HashMap(Map <KeyType, ValueType> other) {
            super(other);
        }
```

```
public int hashCode() {
    KeyType k = getKey();
    return k == null ? 0 : k.hashCode();
}
/**
 * Computes the hashcode for this String.
 * A String is represented by an array of Character.
 * This is done with int arithmetic,
 * where ** represents exponentiation, by this formula:<br>
 * <code>s[0]*31**(n-1) + s[1]*31**(n-2) + ... + s[n-1]</code>.
 *
 * @return hashcode value of this String
 */
public int hashCode() {
    int hashCode = 0;
    int limit = count + offset;
    for (int i = offset; i < limit; i++)
        hashCode = hashCode * 31 + value[i];
    return hashCode;
}
```

Rangkuman

- Hash tables: array
- Hash function: Fungsi yang memetakan keys menjadi bilangan [0 ⇒ ukuran dari hash table)
- Collision resolution
  - Open hashing
    - Separate chaining
  - Closed hashing (Open addressing)
    - Linear probing
    - Quadratic probing
    - Double hashing
  - Primary Clustering, Secondary Clustering

Rangkuman

- **Advantage**
  - running time
    - O(1) + O(*collition resolution*)
  - Cocok untuk merepresentasikan data dengan frekuensi *insert, delete* dan *search* yang tinggi.
- **Disadvantage**
  - Sulit (tidak efisien) untuk mencetak seluruh elemen pada hash table
  - tidak efisien untuk mencari elemen minimum or maximum
  - tidak bisa di-*expand* (untuk closed hash/open addressing)
  - ada pemborosan memory/space

Referensi

- http://www.cs.auckland.ac.nz/software/AlgAnim/hash_tables.html
- http://www.engin.umd.umich.edu/CIS/course.des/cis350/hashing/WEB/HashApplet.htm
- Slide materi UI