

IKI10400 • Struktur Data & Algoritma: Hashtables

Fakultas Ilmu Komputer • Universitas Indonesia

Slide acknowledgments:

Suryana Setiawan, Ade Azurat, Denny, Ruli Manurung



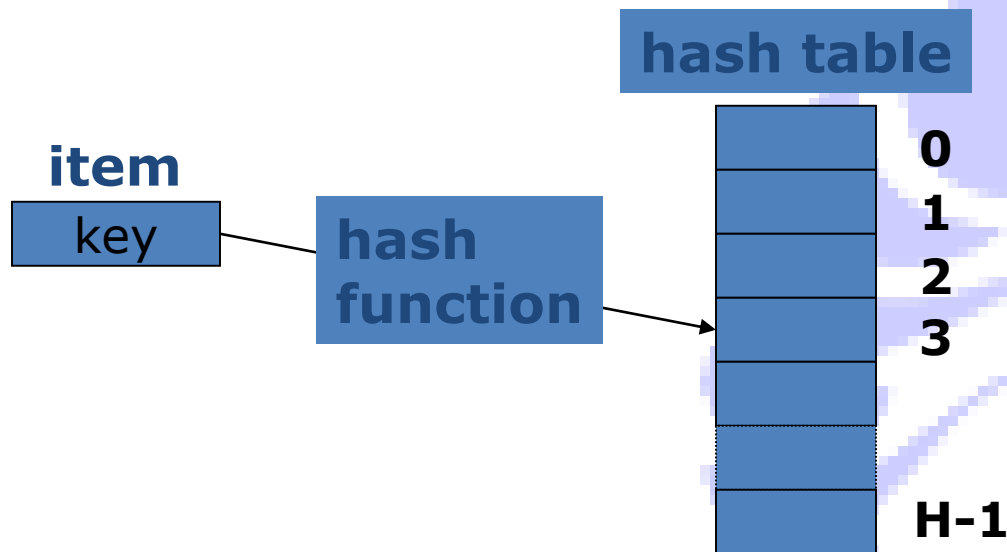
Outline

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



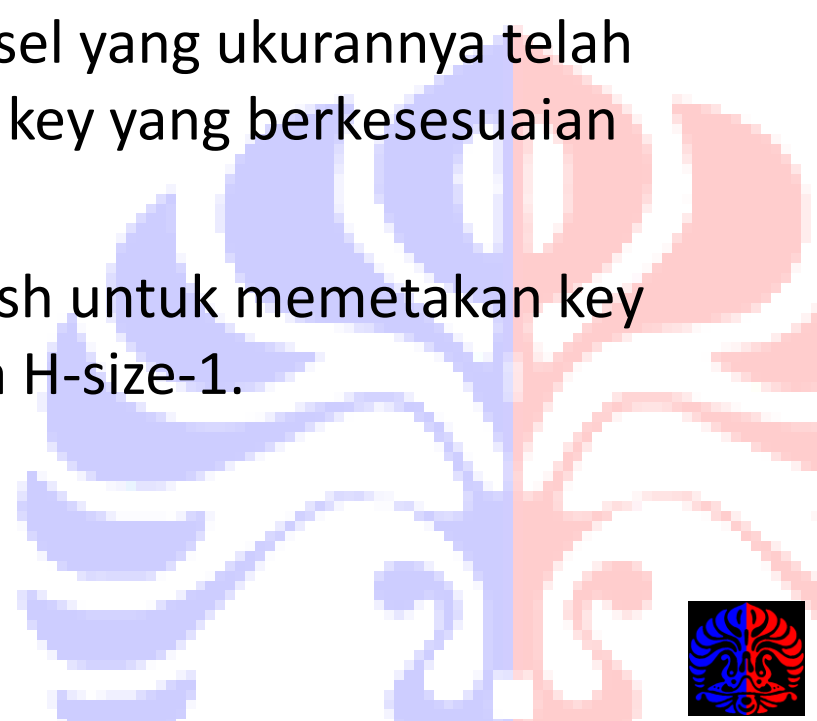
Hash Tables

- Hashing digunakan untuk menyimpan data yang cukup besar pada ADT yang disebut hash table.
- Ukuran Hash table (H-size), biasanya lebih besar dari jumlah data yang hendak disimpan.
- **load factor** (λ) adalah perbandingan antara data yang disimpan dengan ukuran hash table.
- **Fungsi Hash** memetakan elemen pada indeks dari hash table.



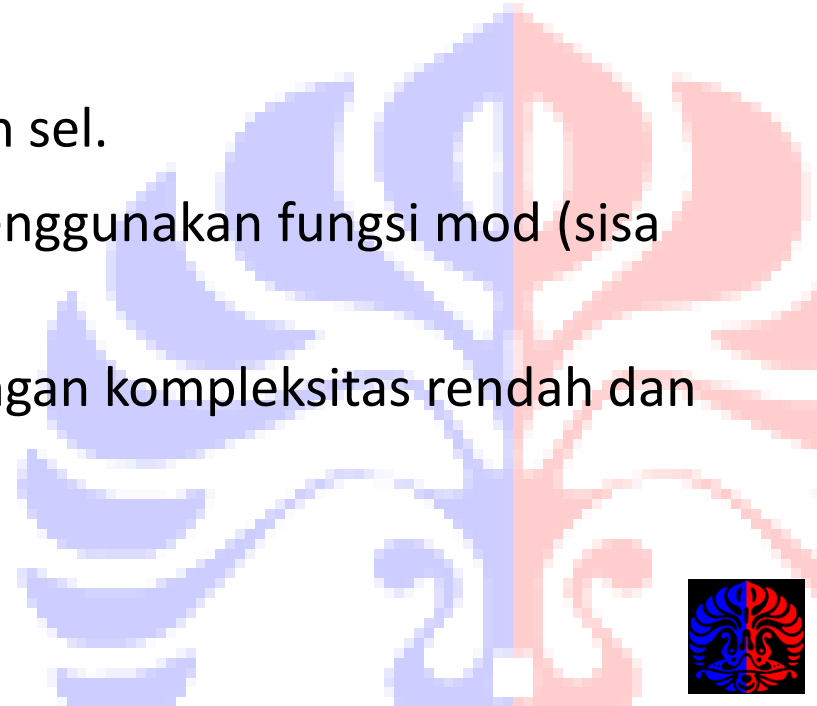
Hash Tables (2)

- Hashing adalah teknik untuk melakukan penambahan, penghapusan dan pencarian dengan **constant average time**.
- Untuk menambahkan data atau pencarian, ditentukan key dari data tersebut dan digunakan sebuah fungsi hash untuk menetapkan lokasi untuk key tersebut.
- Hash tables adalah arrays dengan sel-sel yang ukurannya telah ditentukan dan dapat berisi data atau key yang berkesesuaian dengan data.
- Untuk setiap key, digunakan fungsi hash untuk memetakan key pada bilangan dalam rentang 0 hingga $H\text{-size}-1$.



Fungsi Hash

- Fungsi hash harus memiliki sifat berikut:
 - mudah dihitung.
 - dua key yang berbeda akan dipetakan pada dua sel yang berbeda pada array. (secara umum tidak bisa berlaku, mengapa?).
 - dapat dicapai dengan menggunakan direct-address table dimana semesta dari key relatif kecil.
 - membagi key secara rata pada seluruh sel.
- Sebuah fungsi hash sederhana adalah menggunakan fungsi mod (sisa bagi) dengan bilangan prima.
- Dapat menggunakan manipulasi digit dengan kompleksitas rendah dan distribusi key yang rata.



Fungsi Hash: Truncation

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

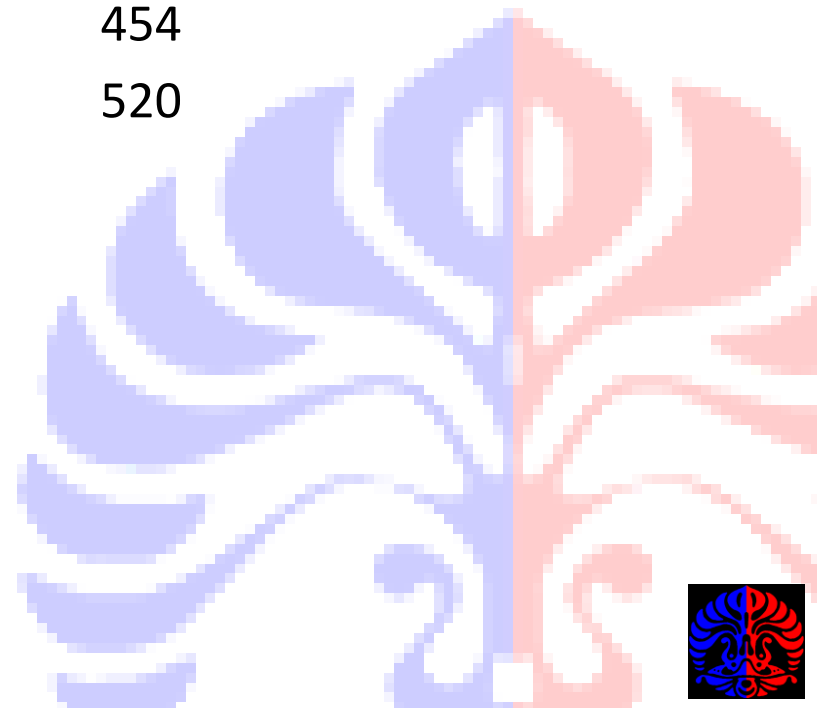
<i>Phone no:</i>	<i>index</i>
731-3018	338
539-2309	329
428-1397	217



Fungsi Hash: Folding

- Data dipecah menjadi beberapa bagian, kemudian tiap bagian tersebut digabungkan lagi dalam bentuk lain.
- contoh.

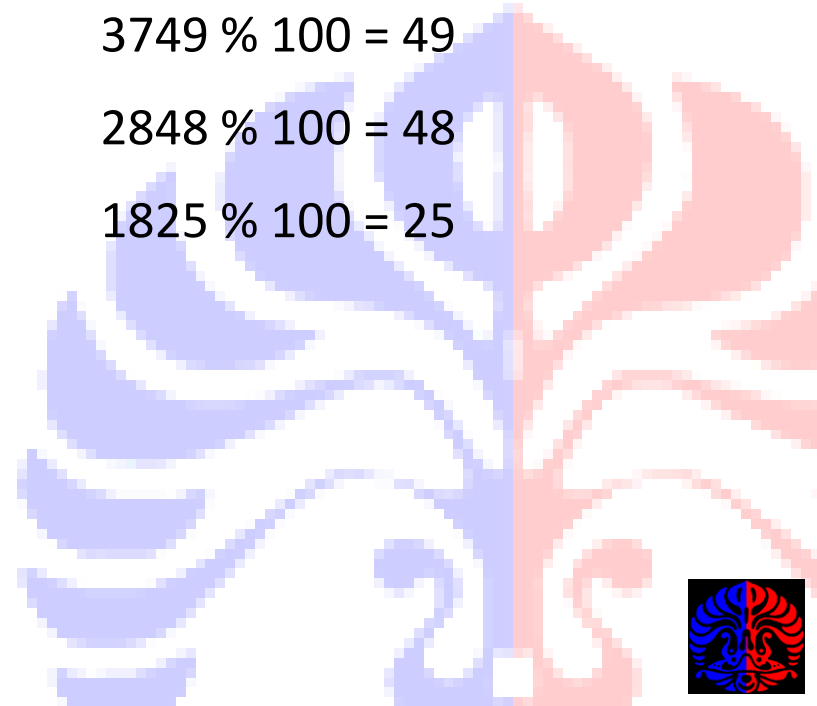
<i>Phone no:</i>	<i>3-group</i>	<i>index</i>
7313018	73+13+018	104
5392309	53+92+309	454
4281397	42+81+397	520



Fungsi Hash: Modular arithmetic

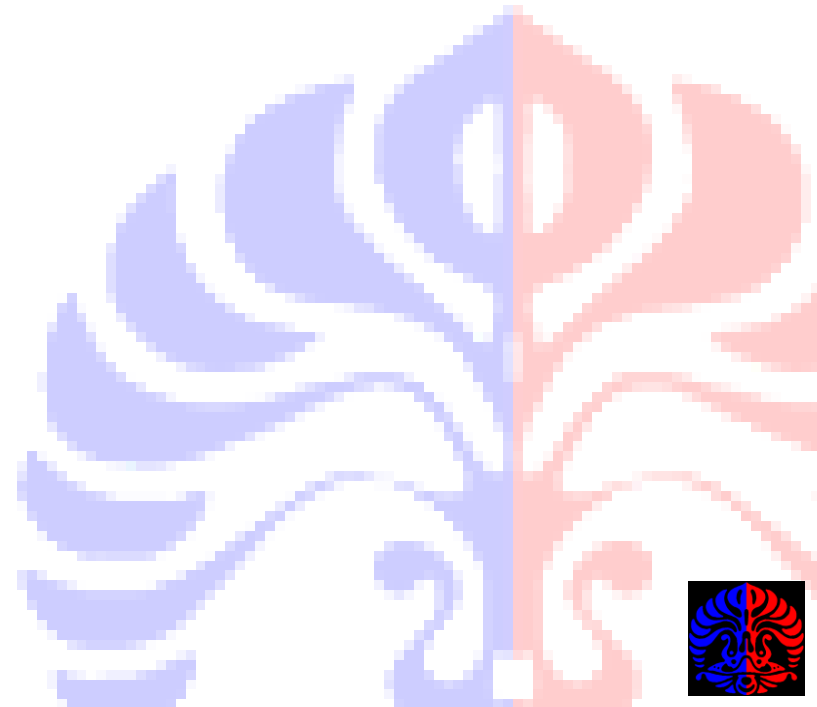
- Melakukan konversi data ke bentuk bilangan bulat, dibagi dengan ukuran hash table, dan mengambil hasil sisa baginya sebagai indeks.
- contoh:

<i>Phone no:</i>	<i>2-group</i>	<i>index</i>
7313018	731+3018	$3749 \% 100 = 49$
5392309	539+2309	$2848 \% 100 = 48$
4281397	428+1397	$1825 \% 100 = 25$



Memilih Fungsi Hash

- Sebuah fungsi hash yang bagus memiliki dua kriteria:
 1. Harus dapat cepat dihitung.
 2. Harus meminimalkan juga collisions yang terjadi.



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

- $(A + B) \% C = (A \% C + B \% C) \% C$
- $(A * B) \% C = (A \% C * B \% C) \% C$



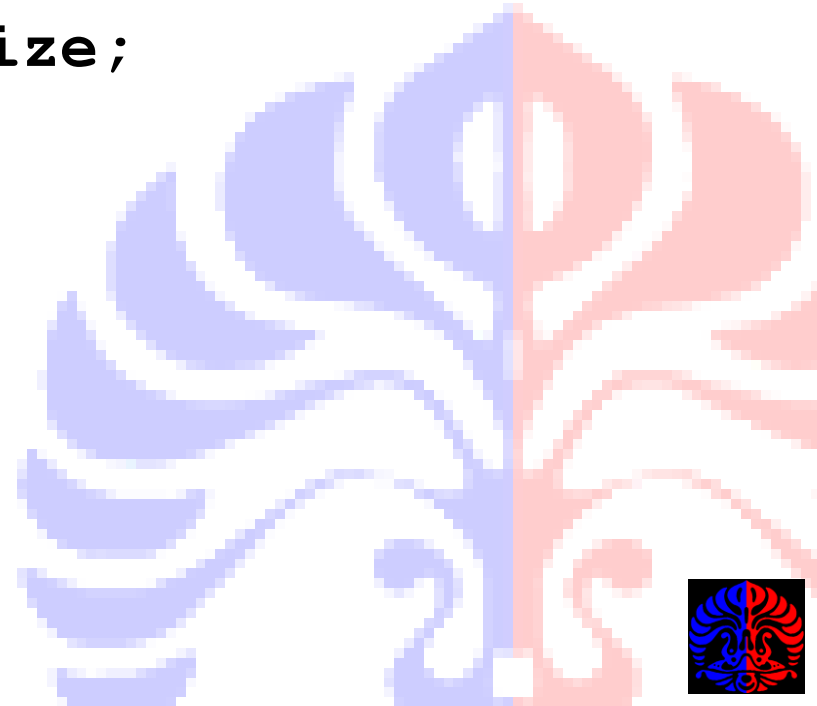
Contoh Fungsi Hash

```
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

- Collision Resolution: Penyelesaian bila terjadi **collision** (*tabrakan*).
- Dikatakan terjadi **collision** jika dua buah *keys* dipetakan pada sebuah sel.
- **Collision** bisa terjadi saat melakukan **insertion**.
- Dibutuhkan prosedur tambahan untuk mengatasi terjadinya **collision**.
- Ada dua strategi umum:
 - Closed Hashing (Open Addressing)
 - Open Hashing (Chaining)



Closed Hashing

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



Closed Hashing

- Beberapa strategy/alternatif untuk menentukan bentuk fungsi *f*, yaitu:
 - *Linear probing*
 - *Quadratic probing*
 - *Double hashing*

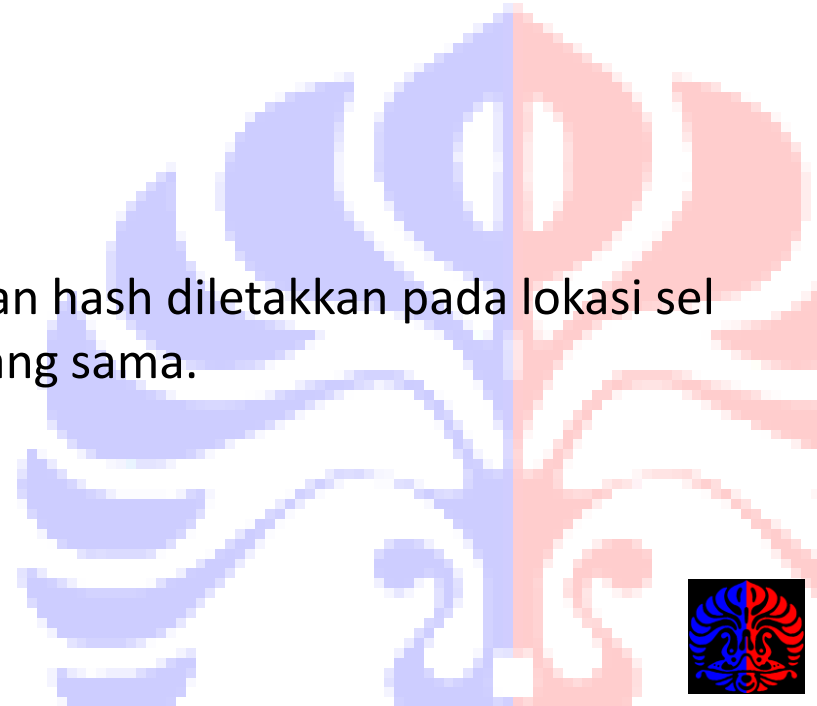


Linear Probing

- Gunakan fungsi linear

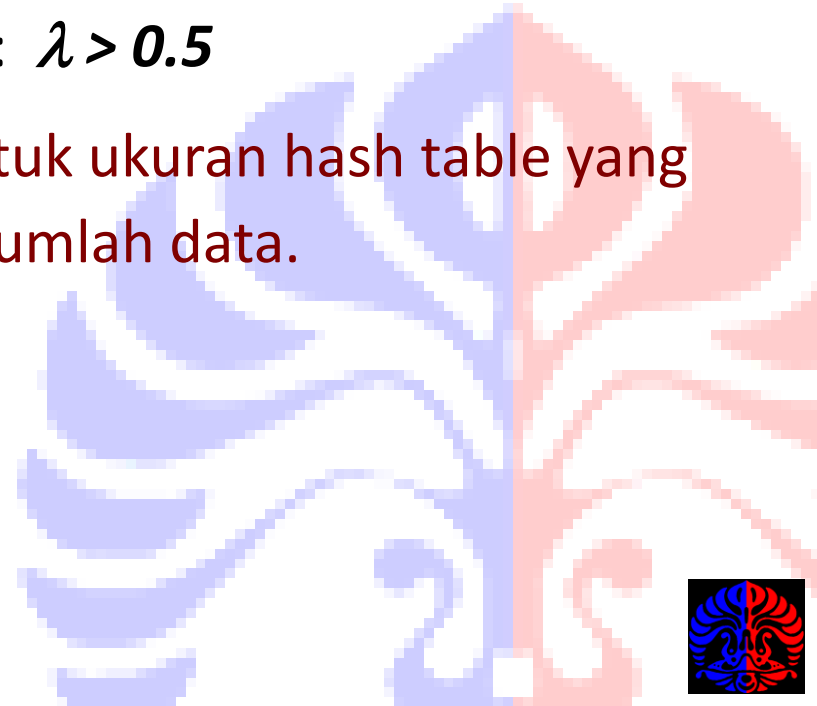
$$f(i) = i$$

- Bila terjadi collision, cari posisi pertama pada tabel yang terdekat dengan posisi yang seharusnya.
- 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.

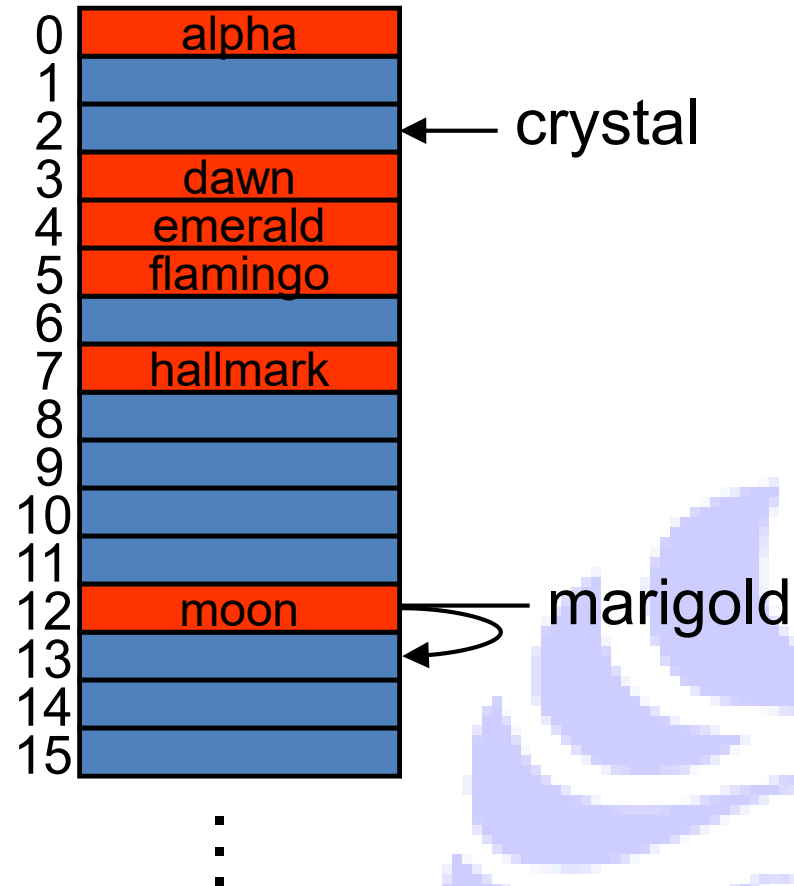


Linear Probing: Load factor

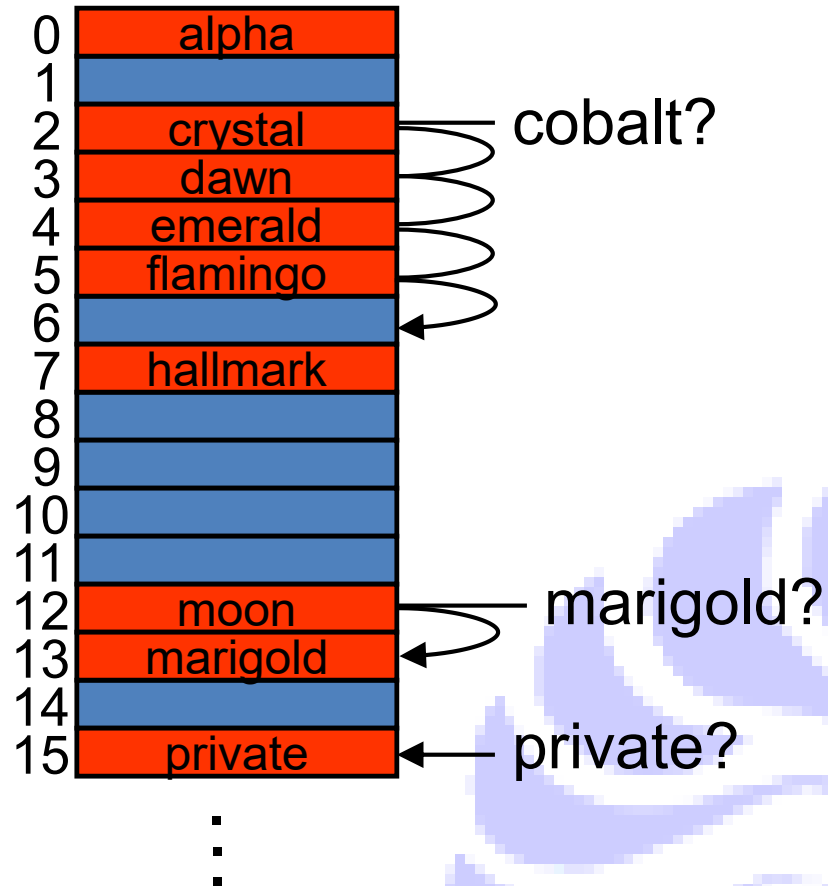
- Kompleksitas dari teknik ini bergantung pada nilai λ (*load factor*).
- **Definisi λ (load factor):**
 - Untuk hash table T dengan ukuran m , yang menyimpan n data.
 - λ (*Load factor*) dari T adalah n/m
- **Linear Probing** tidak disarankan bila: $\lambda > 0.5$
- **Linear Probing** hanya disarankan untuk ukuran hash table yang ukurannya lebih besar dua kali dari jumlah data.



Hashing - insert

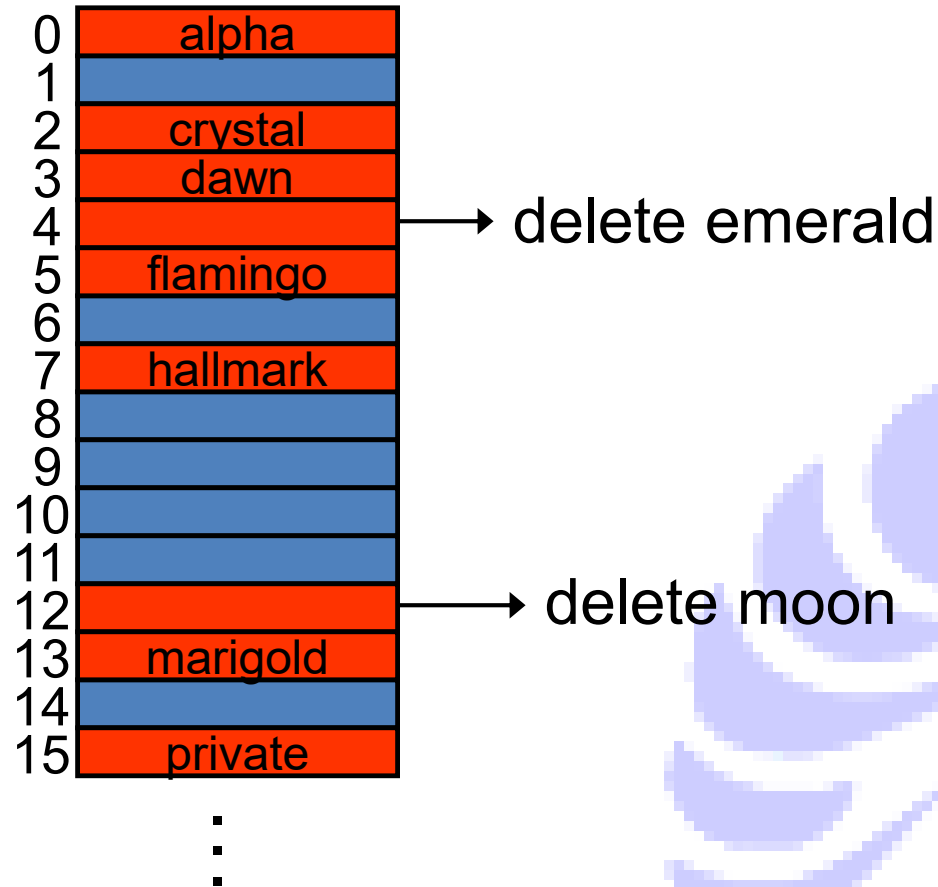


Hashing - lookup

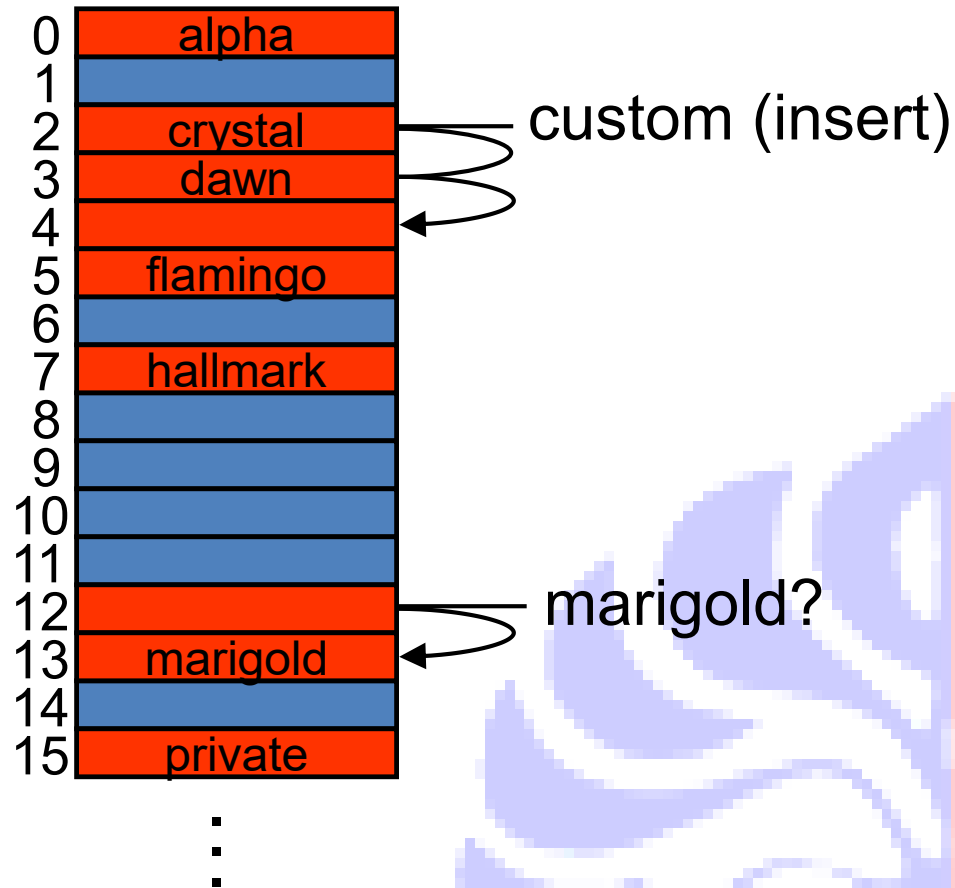


Hashing - delete

- lazy deletion - mengapa?

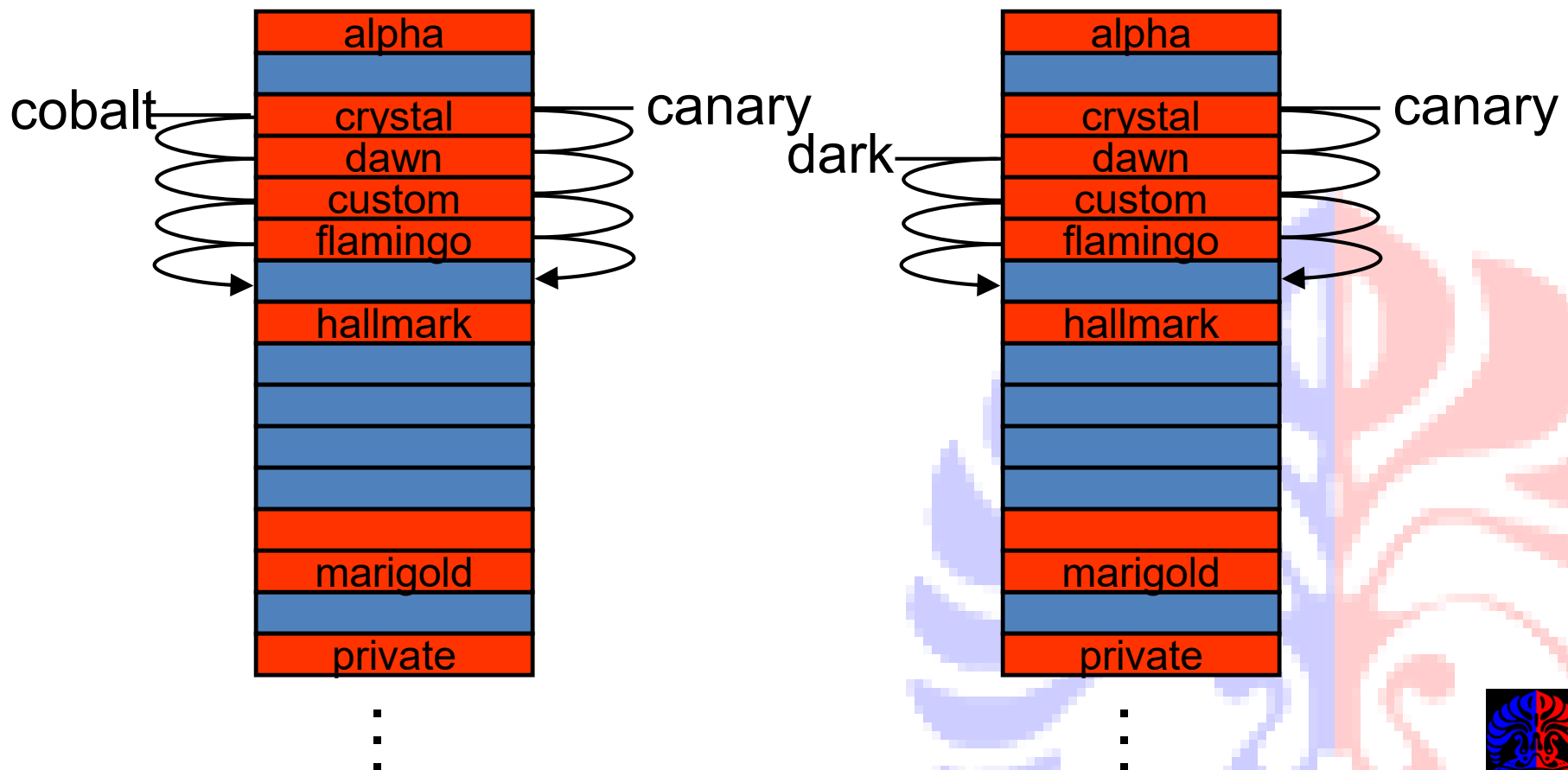


Hashing - operation after delete



Primary Clustering

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



Quadratic Probing

- Menghindari *primary clustering* dengan 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.

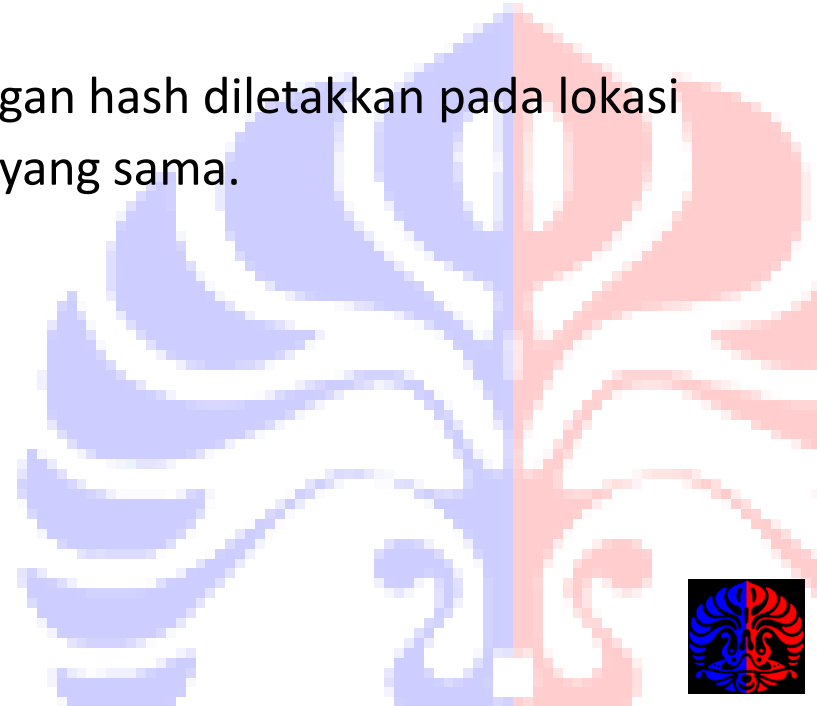


Quadratic Probing

- Dapat melakukan *increment* bila terjadi *collision*
- Perhatikan bahwa fungsi *quadratic* dapat dijabarkan sebagai berikut:

$$f(i) = i^2 = f(i-1) + 2i - 1.$$

- Menimbulkan ***second clustering***:
 - Elemen-elemen yang menurut perhitungan hash diletakkan pada lokasi sel sama, diarahkan pada sel pengganti yang sama.

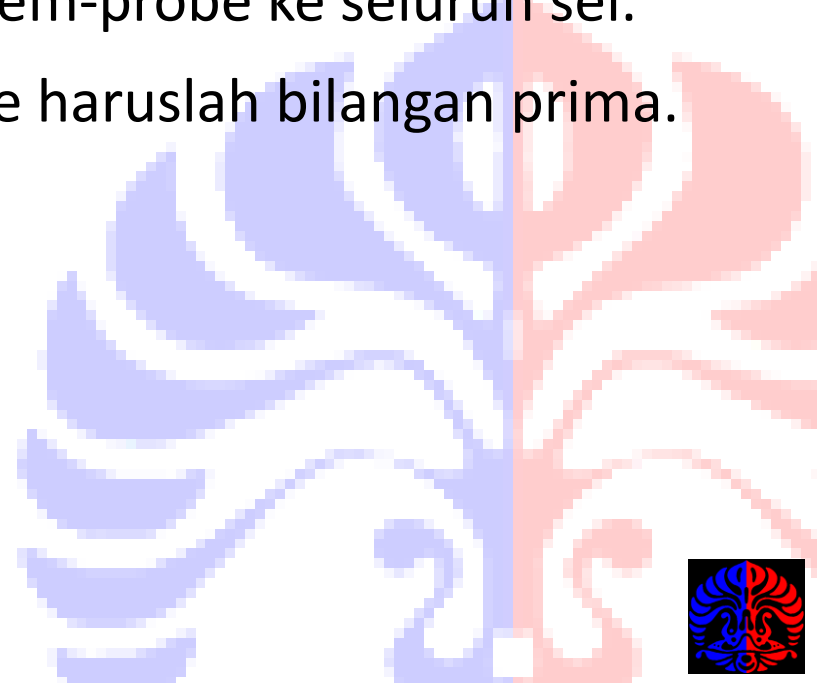


Double hashing

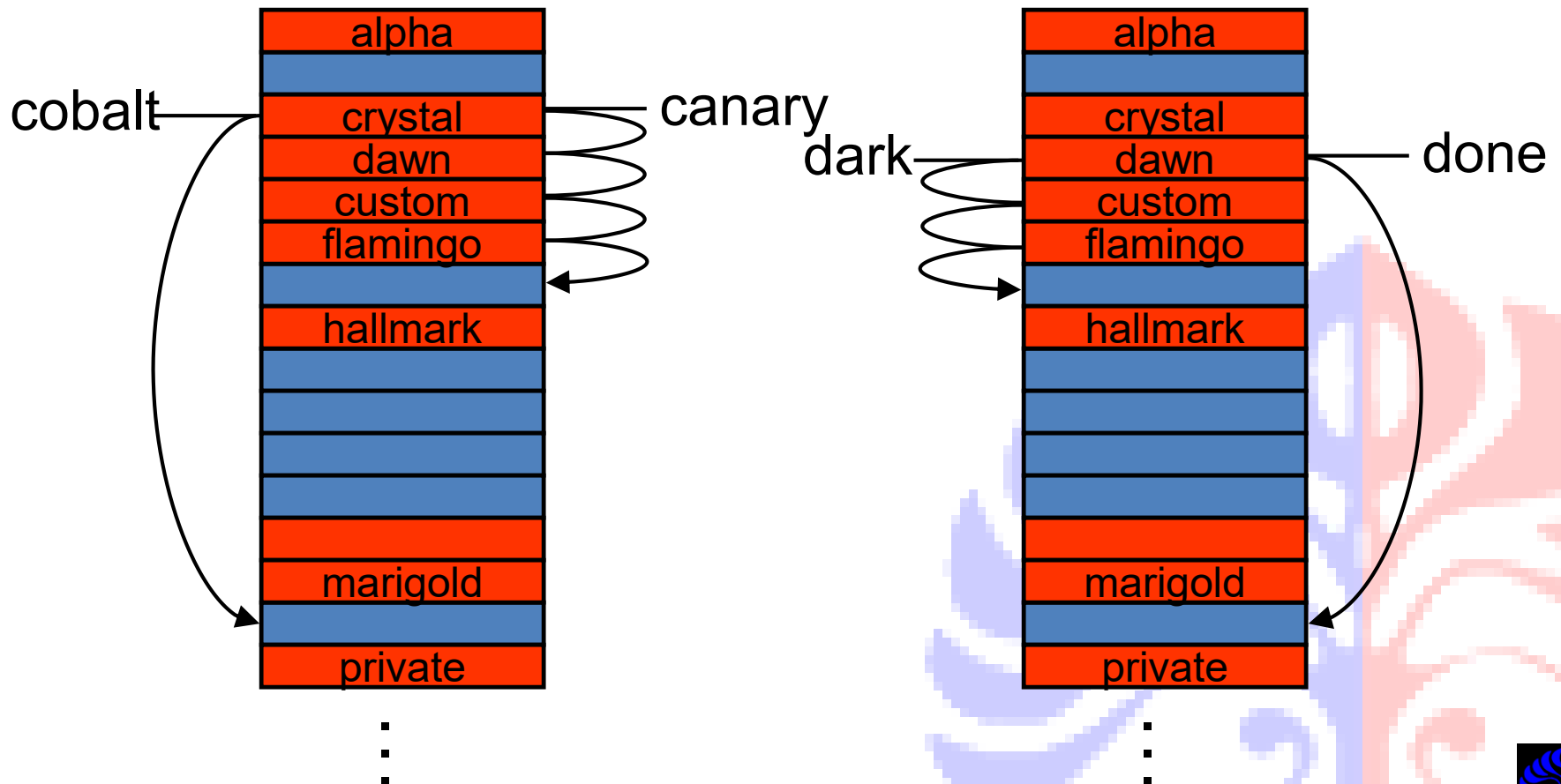
- fungsi untuk *collision resolution* disusun dengan fungsi hash seperti :

$$f(i) = i * \text{hash2}(x)$$

- Setiap saat faktor $\text{hash2}(x)$ 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.



Double Hashing



Open Hashing

- 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.

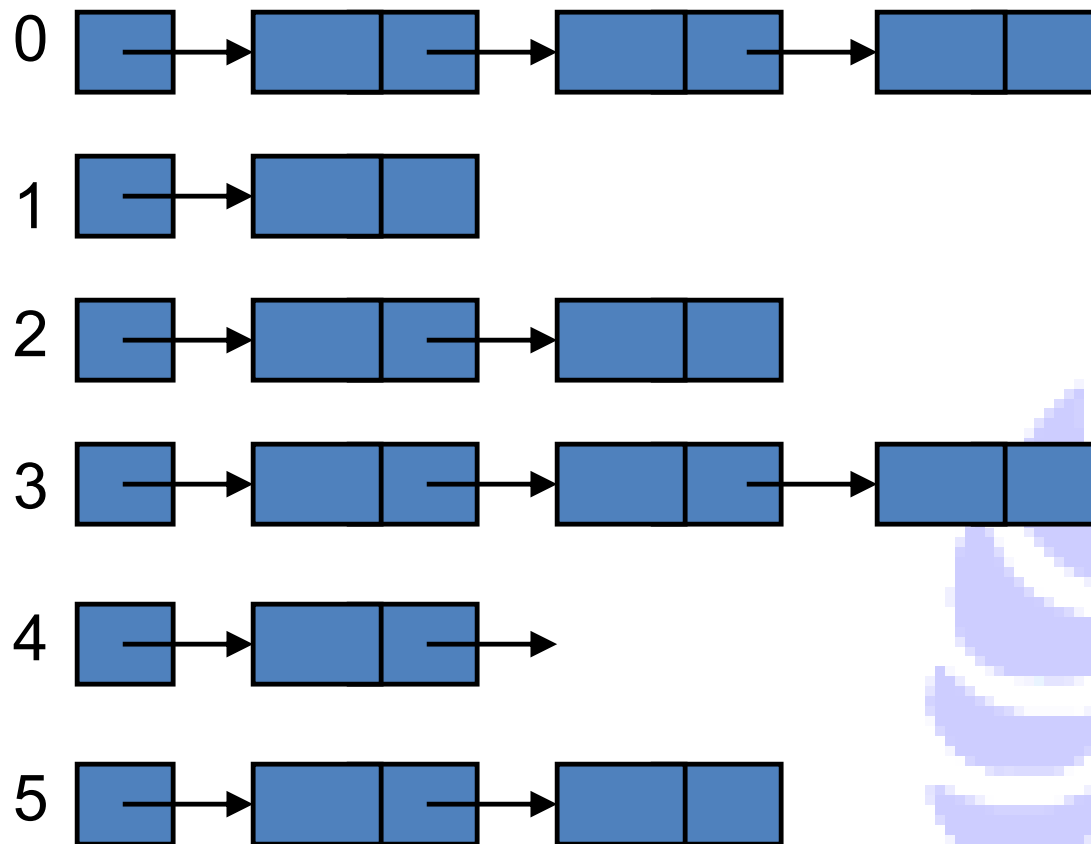


Open Hashing

- Fungsi dan analisa *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 memory pada tiap sel.



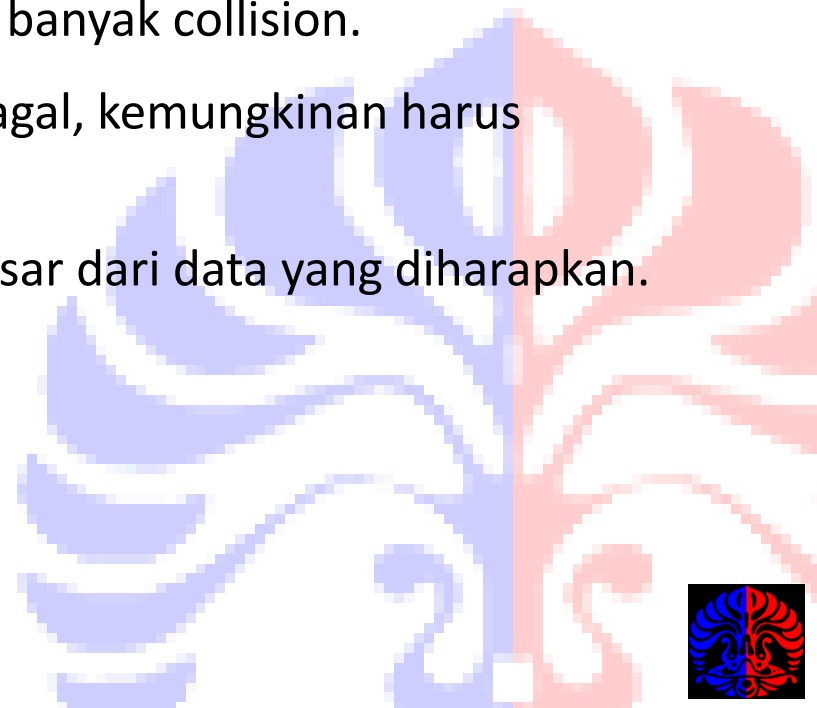
Analisa *Open Hash*

- 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 ukuran-nya dinamic.



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.



HASHSET

Contoh Implementasi Hash Table
Quadratic Probing



```

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;
}

```



- Source Code lengkap bisa dilihat di:
- <http://telaga.cs.ui.ac.id/WebKuliah/IKI20100/resources/weiss.code/weiss/util/HashSet.java>
- <http://telaga.cs.ui.ac.id/WebKuliah/IKI20100/resources/weiss.code/weiss/util/HashMap.java>



Rangkuman

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



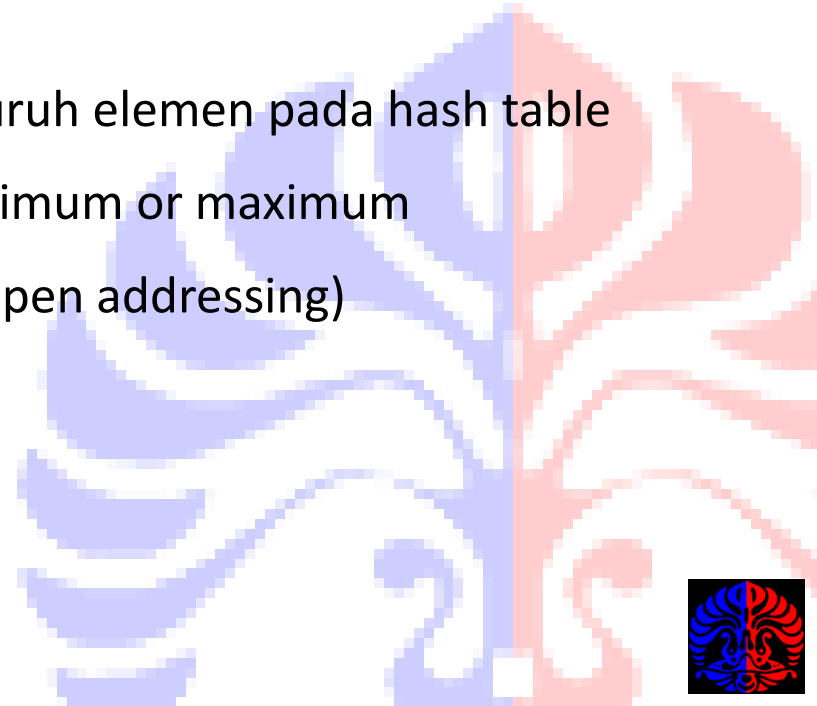
Rangkuman

- Advantage

- running time
 - $O(1) + O(\text{collision resolution})$
- Cocok untuk merepresentasikan data dengan frekuensi insert, delete dan search yang tinggi.

- Disadvantage

- Sulit (tidak efficient) untuk mencetak seluruh elemen pada hash table
- tidak efficient untuk mencari elemen minimum or maximum
- tidak bisa di expand (untuk closed hash/open addressing)
- ada pemborosan memory/space



Referensi

- Bab 19 pada buku teks
- 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>

