Pengantar hash tables

Beberapa diambil dari slide kuliah SDA

Set / Himpunan

{A,B,D,E,F}

Operasi:

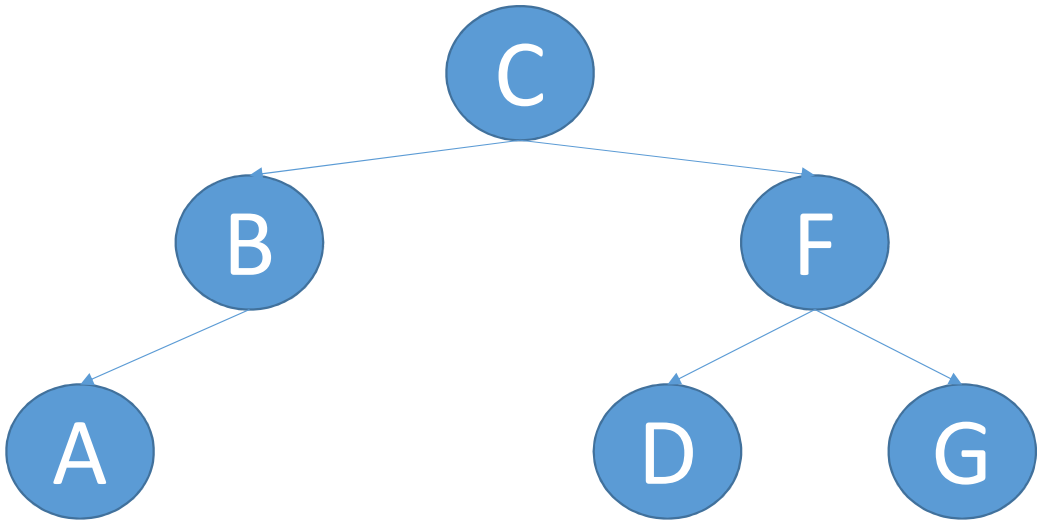
Insert (X)

Find (X)

Delete (X)

Bagaimana cara mengimplementasikan **Set** agar operasi tersebut efisien?

Solusi 1 : BST



Berapa kompleksitas kasus rata-rata operasi **Insert, find, delete** ?

Insert, find, delete O(1) ??

- Apakah kita bisa membuat representasi **Set** sehingga operasi tersebut konstan O(1) ?

- Ketika ukuran **Set** semakin besar, rasanya sulit mencapai O(1) untuk operasi **find** !

Hash tables

- Prinsip **Hashing** !

- Sebuah teknik untuk melakukan insert, find, dan delete elemen dengan kompleksitas kasus rata-rata konstan O(1)

- Sebuah usaha untuk merepresentasikan **Set** agar operasi insert, find, dan delete menjadi konstan O(1)

Hash tables

Misal, data kita adalah integer dari berkisar antara 0 – 65,535

Untuk menyimpan data tersebut, kita perlu membuat array berukuran 65,536 !

Hash table

0	
1	
2	

...

65534	
65535	

Hash tables

Misal, data kita adalah integer dari berkisar antara 0 – 65,535

Kita isi 0 jika elemen tersebut belum ada pada struktur data

Hash table

0	0
1	0
2	0
…	
65534	0
65535	0

Hash tables

Misal, data kita adalah integer dari berkisar antara 0 – 65,535

Insert (2) -> array[2]++

array[i] merepresentasikan berapa kali i di-insert !

Hash table

0	0
1	0
2	1
…	
65534	0
65535	0

Insert O(1) !

Hash tables

Misal, data kita adalah integer dari berkisar antara 0 – 65,535

Insert (2) -> array[2]++

array[i] merepresentasikan berapa kali i di-insert !

Hash table

0	0
1	0
2	2
…	
65534	0
65535	0

Insert O(1) !

Hash tables

Misal, data kita adalah integer dari berkisar antara 0 – 65,535

Insert (65534) -> array[65534]++ **array[i] merepresentasikan berapa kali i di-insert !**

Hash table

0	0
1	0
2	2
…	
65534	1
65535	0

Insert O(1) !

Hash tables

Misal, data kita adalah integer dari berkisar antara 0 – 65,535

Find (i) -> cukup pastikan bahwa array[i] bukan berisi 0

Hash table

0	0
1	0
2	1
…	
65534	0
65535	0

Find O(1) !

Hash tables

- Ada masalah dengan representasi sebelumnya ??
- Kalau integer kita bukan 16 bit, tetapi 32 bit, array kita harus sanggup menampung **4 juta item !**
- Kalau item **bukan integer**, tetapi **String** ! atau yang lain
  - Bagaimana cara akses indeks array yang bukan integer ???
  - BankAccount, Person, Mahasiswa, dsb

Hash tables

- Kalau item bukan integer, tetapi **String** !

- **Solusi** : ubah string ke integer dengan sebuah rumus ☺

Misal: string "**junk**" dapat diindeks dengan

= **'j'** * 128^3 + **'u'** * 128^2 + **'n'** * 128^1 + **'k'** * 128^0

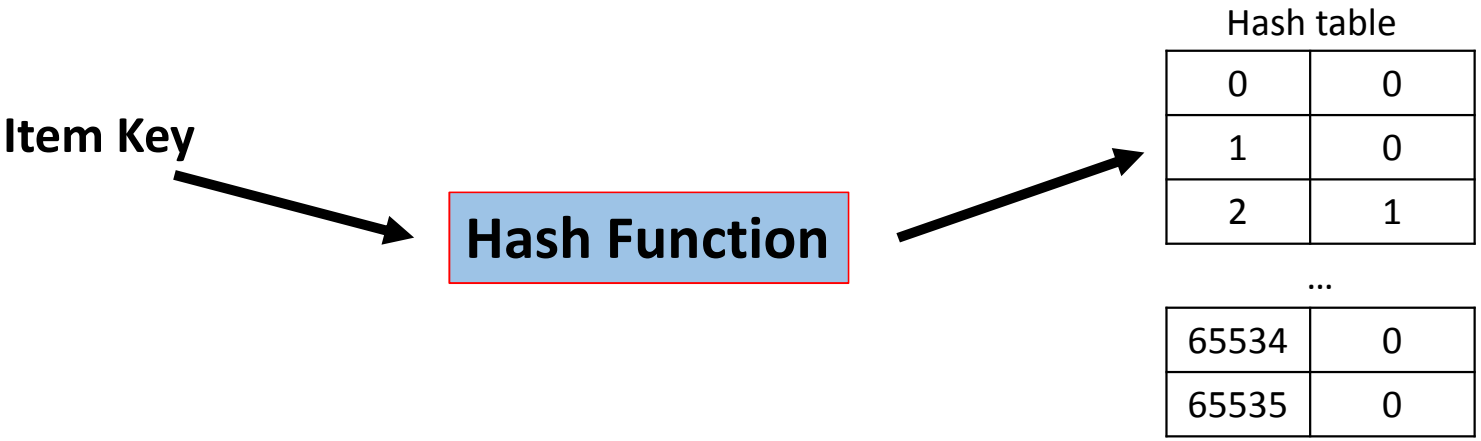
Hash tables

- **‘j’** * 128^3 + **‘u’** * 128^2 + **‘n’** * 128^1 + **‘k’** * 128^0 = **224,229,227** !

- Balik lagi ke permasalahan pertama ! **Bagaimana menghindari penggunaan array yang ukuran sangat besar ??**

Hash tables & Hash function

- Perlu ada **fungsi** yang memetakan **nilai yang besar** ke nilai indeks yang lebih kecil yang lebih mudah untuk diatur dan sesuai dengan ukuran **hash table** !

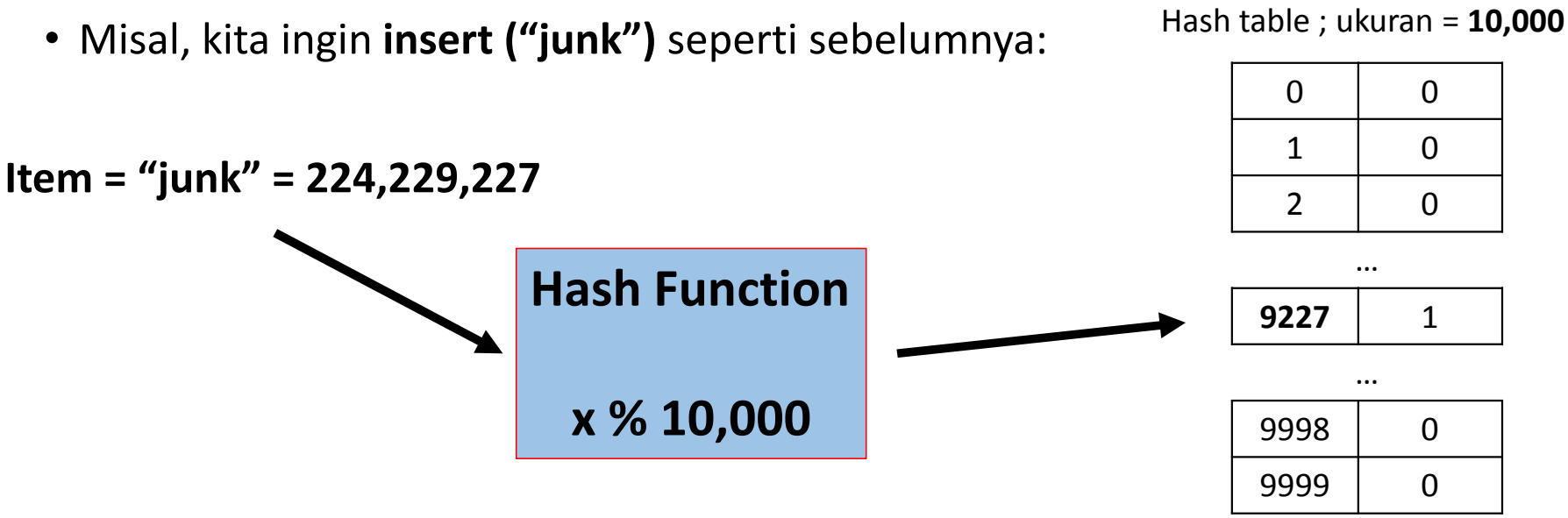


Hash tables & Hash function

- Hash Function yang paling sederhana:

  H(item_key) = item_key % ukuran hash table

- Misal, kita ingin **insert (“junk”)** seperti sebelumnya:



Hashing : summary

- Data record disimpan pada **Hash Table**
- Posisi data record di hash table ditentukan oleh **key**-nya
- **Hash Function** adalah fungsi yang memetakan key ke posisi pada hash table
- Jika hash function memetakan dua key ke posisi yang sama, maka terjadi **collision**.

HashTable vs Linked List and Trees

- Hash tables mendukung proses insert/find/delete yang efisien (**almost O(1)**)
  - Cocok untuk program yang banyak melakukan insert/find/delete (frekuensi tinggi)

- Tetapi, Hash Table kehilangan **locality of data**
  - **no sorting**, or **min/max (sulit/tidak efisien)**
  - Jika disuruh untuk mencetak semua elemen pada hash table, sulit, tidak efisien
  - Boros memori

hashCode() pada kelas java.lang.String

```
public final class String
{
    public int hashCode( )
    {
        if( hash != 0 )
            return hash;

        for( int i = 0; i < length( ); i++ )
            hash = hash * 31 + (int) charAt( i );
        return hash;
    }

    private int hash = 0;
}
```

Linear Probing Hash Table

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

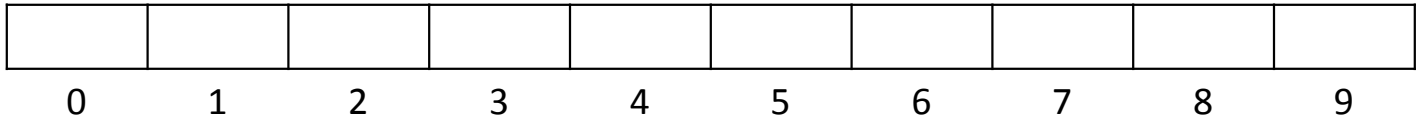
	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Linear Probing Hash Table

Tablesize = 10

H(x) = x % tablesize

Insert 45 13 34 67 23 74

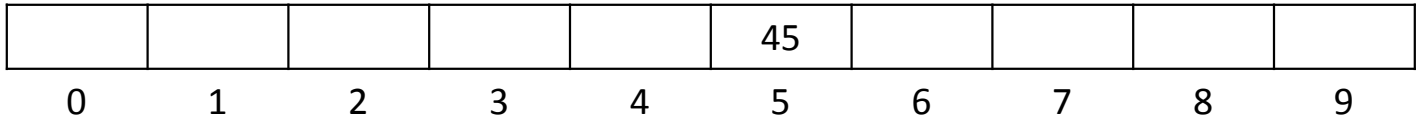


Linear Probing Hash Table

Tablesize = 10

H(x) = x % tablesize

Insert 45 13 34 67 23 74

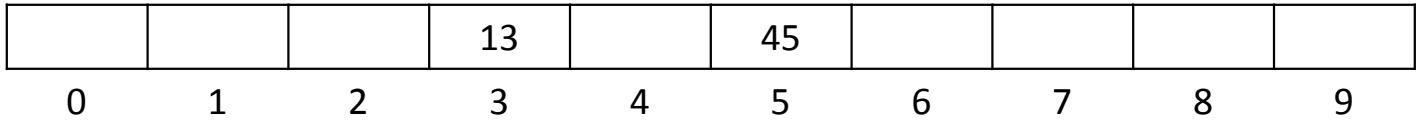


Linear Probing Hash Table

Tablesize = 10

H(x) = x % tablesize

Insert 45 13 34 67 23 74

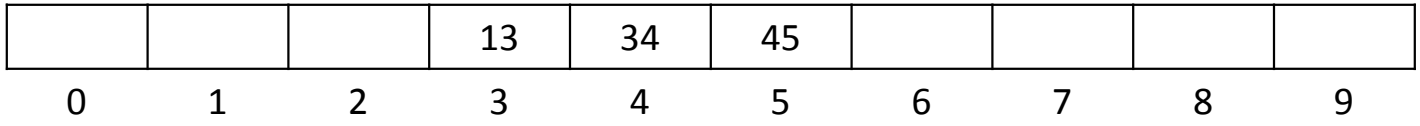


Linear Probing Hash Table

Tablesize = 10

H(x) = x % tablesize

Insert 45 13 34 67 23 74



Linear Probing Hash Table

Tablesize = 10

H(x) = x % tablesize

Insert 45 13 34 67 23 74

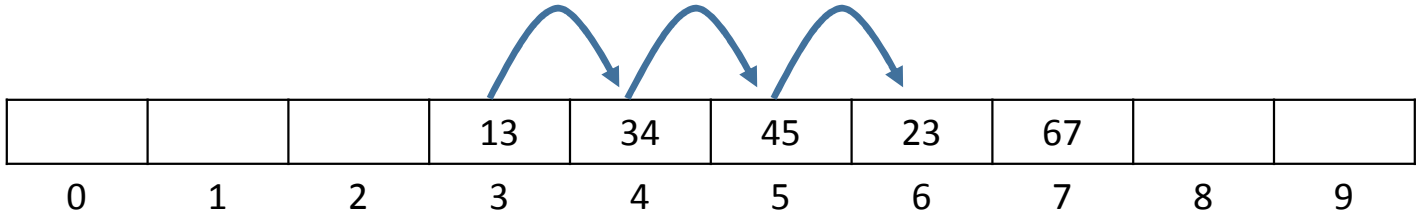
			13	34	45		67		
0	1	2	3	4	5	6	7	8	9

Linear Probing Hash Table

Tablesize = 10

H(x) = x % tablesize

Insert 45 13 34 67 23 74

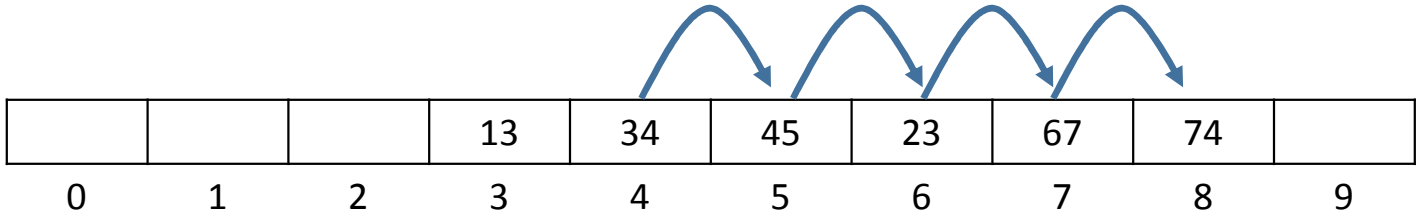


Linear Probing Hash Table

Tablesize = 10

H(x) = x % tablesize

Insert 45 13 34 67 23 74



Linear Probing Hash Table

Yang perlu diperhatikan jika menggunakan linear probing:

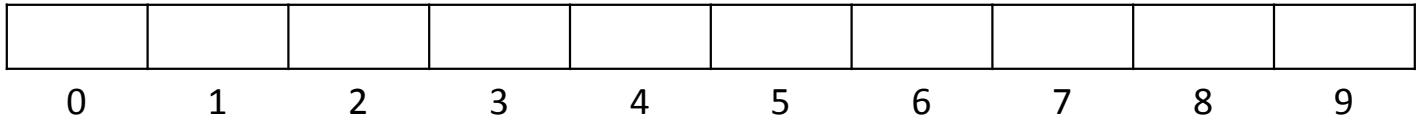
- Hash function harus bisa mendistribusikan item dengan baik di hash table
- Hash table harus cukup besar untuk menghindari primary clustering
- Biasanya, table harus 2 kali lebih besar dari banyaknya item yang dikandung
  - **$\lambda <= 0.5$**

Quadratic Probing Hash Table

Tablesize = 10

H(x) = x % tablesize

Insert 45 13 34 67 23 74

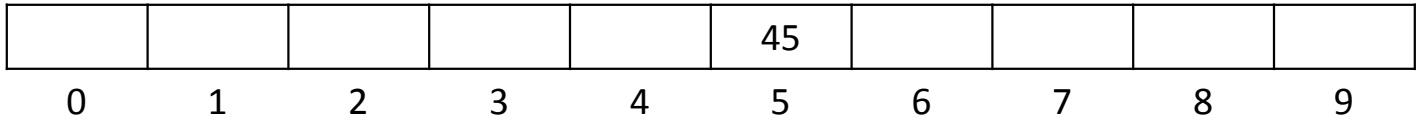


Quadratic Probing Hash Table

Tablesize = 10

H(x) = x % tablesize

Insert 45 13 34 67 23 74

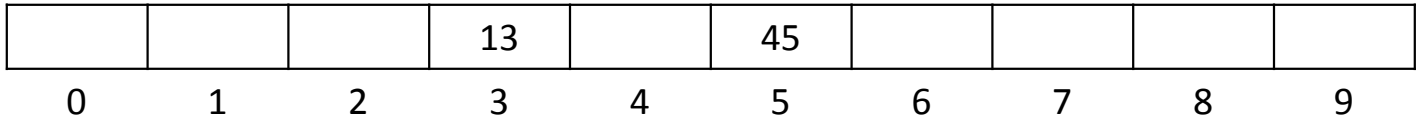


Quadratic Probing Hash Table

Tablesize = 10

H(x) = x % tablesize

Insert 45 13 34 67 23 74



Quadratic Probing Hash Table

Tablesize = 10

H(x) = x % tablesize

Insert 45 13 34 67 23 74

			13	34	45				
0	1	2	3	4	5	6	7	8	9

Quadratic Probing Hash Table

Tablesize = 10

H(x) = x % tablesize

Insert 45 13 34 67 23 74

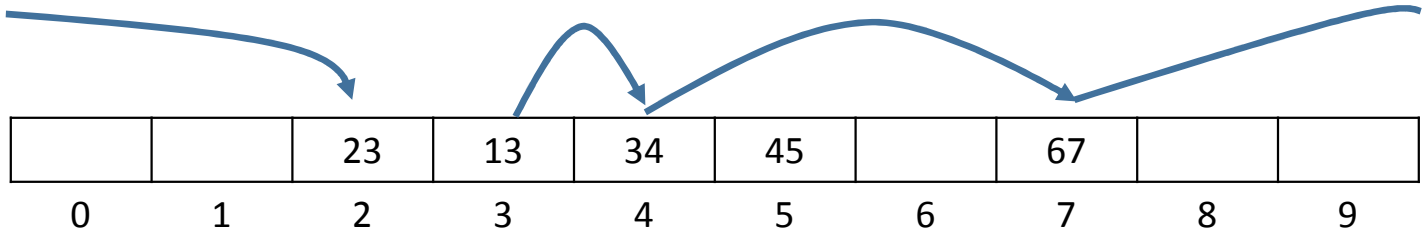
			13	34	45		67		
0	1	2	3	4	5	6	7	8	9

Quadratic Probing Hash Table

Tablesize = 10

H(x) = x % tablesize

Insert 45 13 34 67 23 74

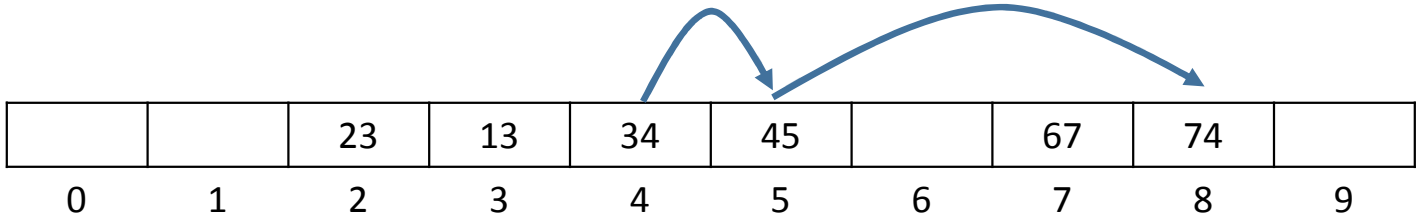


Quadratic Probing Hash Table

Tablesize = 10

H(x) = x % tablesize

Insert 45 13 34 67 23 74



Quadratic Probing Hash Table

- Biasanya collision lebih sedikit, walaupun mungkin terjadi secondary clustering

- Ukuran table harus **bilangan prima**
  - **Kalau linear probing, tidak perlu bilangan prima**

- Harus memastikan ukuran Hash table 2x lebih besar dari banyaknya item yang dikandung

Quadratic Probing Hash Table

Mengapa bilangan prima ?? **Sebuah cara untuk mengurangi collisions**

Kalau tidak prima, urutan slot yang diperiksa menggunakan quadratic probing akan berulang sebelum semua slot diperiksa

Contoh kasus sebelumnya, jika pertama kali terjadi collisions di indeks 1, maka urutan pengecekan berikutnya adalah:

1 4 9 6 5 6 9 4 1 0 1 4 9 6 5 6 9 4 1 0

2,3,7,9 tidak pernah diperiksa !

Double Hashing

- Ada 2 hash function: h1(x) dan h2(x)
- Setelah terjadi i collisions : (h1(x) + i * h2(x)) mod tablesize
- Biasanya, h2(x) = r – (x mod r)
  - r adalah **bilangan prima < tablesize**
  - h2(x) tidak boleh nol
  - Pilihan populer adalah r = 7

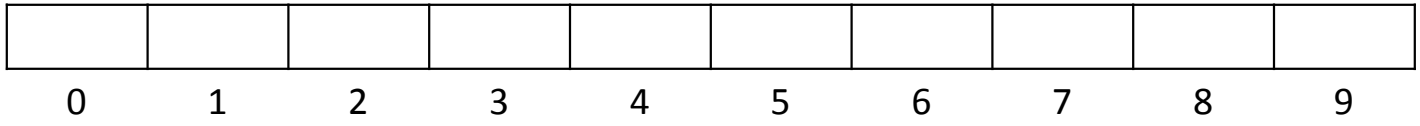
Double Hashing

Tablesize = 10

H1(x) = x % tablesize

H2(x) = 7 – (x mod 7)

Insert 45 13 34 67 23 74



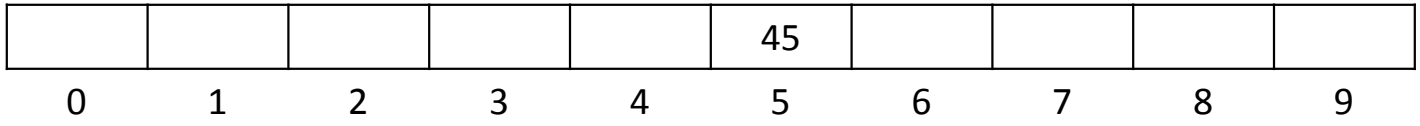
Double Hashing

Tablesize = 10

H1(x) = x % tablesize

H2(x) = 7 – (x mod 7)

Insert 45 13 34 67 23 74



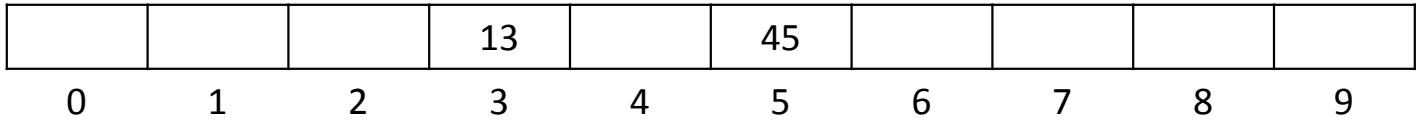
Double Hashing

Tablesize = 10

H1(x) = x % tablesize

H2(x) = 7 – (x mod 7)

Insert 45 13 34 67 23 74



Double Hashing

Tablesize = 10

H1(x) = x % tablesize

H2(x) = 7 – (x mod 7)

Insert 45 13 34 67 23 74

			13	34	45				
0	1	2	3	4	5	6	7	8	9

Double Hashing

Tablesize = 10

H1(x) = x % tablesize

H2(x) = 7 – (x mod 7)

Insert 45 13 34 67 23 74

			13	34	45		67		
0	1	2	3	4	5	6	7	8	9

Double Hashing

Tablesize = 10

H1(x) = x % tablesize

H2(x) = 7 – (x mod 7)

Insert 45 13 34 67 23 74

			13	34	45		67	23	
0	1	2	3	4	5	6	7	8	9

1st collision : (h1(23) + 1 * h2(23)) mod 10 = (3 + 1* 5) mod 10 = **8**

Double Hashing

Tablesize = 10

H1(x) = x % tablesize

H2(x) = 7 – (x mod 7)

Insert 45 13 34 67 23 74

74			13	34	45		67	23	
0	1	2	3	4	5	6	7	8	9

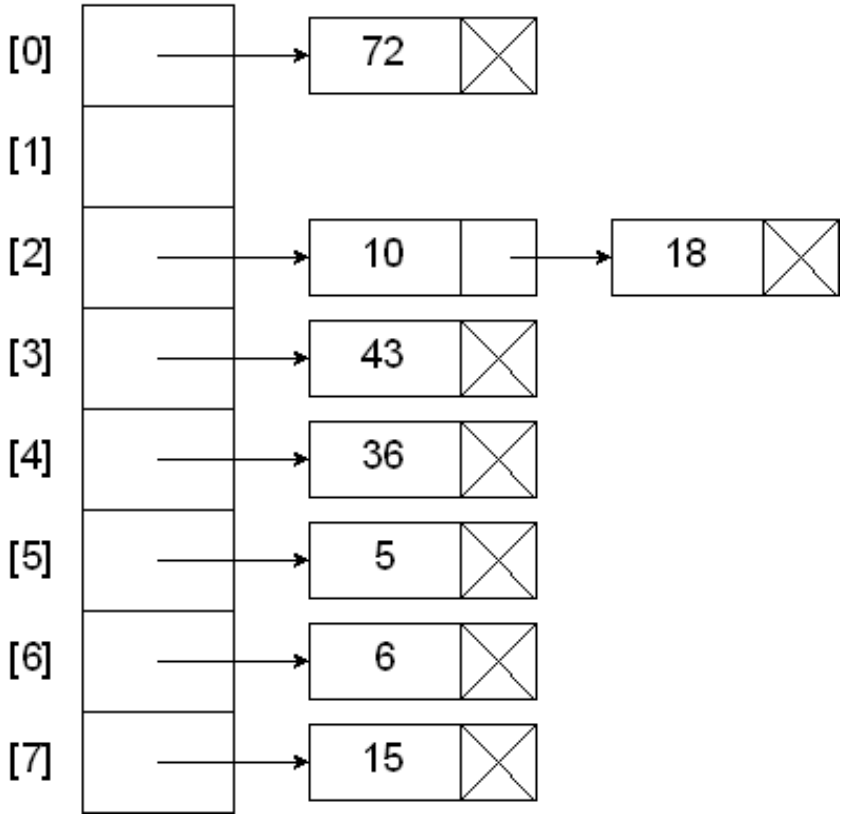
1st collision : (h1(74) + 1 * h2(74)) mod 10 = (4 + 1* 3) mod 10 = **7**

2nd collision : (h1(74) + 2 * h2(74)) mod 10 = (4 + 2* 3) mod 10 = **0**

Open Hashing (chaining) / Closed Addressing

Hash key = key % table size

```
4      = 36 % 8
2      = 18 % 8
0      = 72 % 8
3      = 43 % 8
6      =  6 % 8
2      = 10 % 8
5      =  5 % 8
7      = 15 % 8
```



http://faculty.cs.niu.edu/~freedman/340/340notes/340hash.htm