Hashing

BITE1523: Computer Game Programming

Hashing ini dari segi analoginya kita nak label barangan kita yang disimpan dalam kepelbagaian kotak dan kita label menggunakan nombor. Patutnya boleh by kategori tapi kita guna nombor.

Introduction - Searching

- Searching (internal or external) is a process of finding an element within the list of elements in order or randomly.
- A table of records in which a key is used for retrieval is often called a **SEARCH TABLE** or **DICTIONARY**.
- Methods :
  - Sequential or Linear Searching.
  - Binary Search.
  - Hashing

Searching Methods

O(N)

O(log N)

Hashing can make searching happen in **O(1)** and is quite fast in practice.

O (N) vs O (Log N)

Input Size	O(N) Steps	O(log N) Steps
10	10	4
1000	1000	10
1,000,000	1,000,000	20
1,000,000,000	1,000,000,000	30

Introduction - Hashing

- **Hashing** is a technique that support very fast retrieval via a key
- It is about finding an **address** to store/locate data using a **key (hash function)**
- A hash function maps a key into the range [0 to Max − 1], which is used as an **index** / **address** to **hash table** for storing and retrieving record

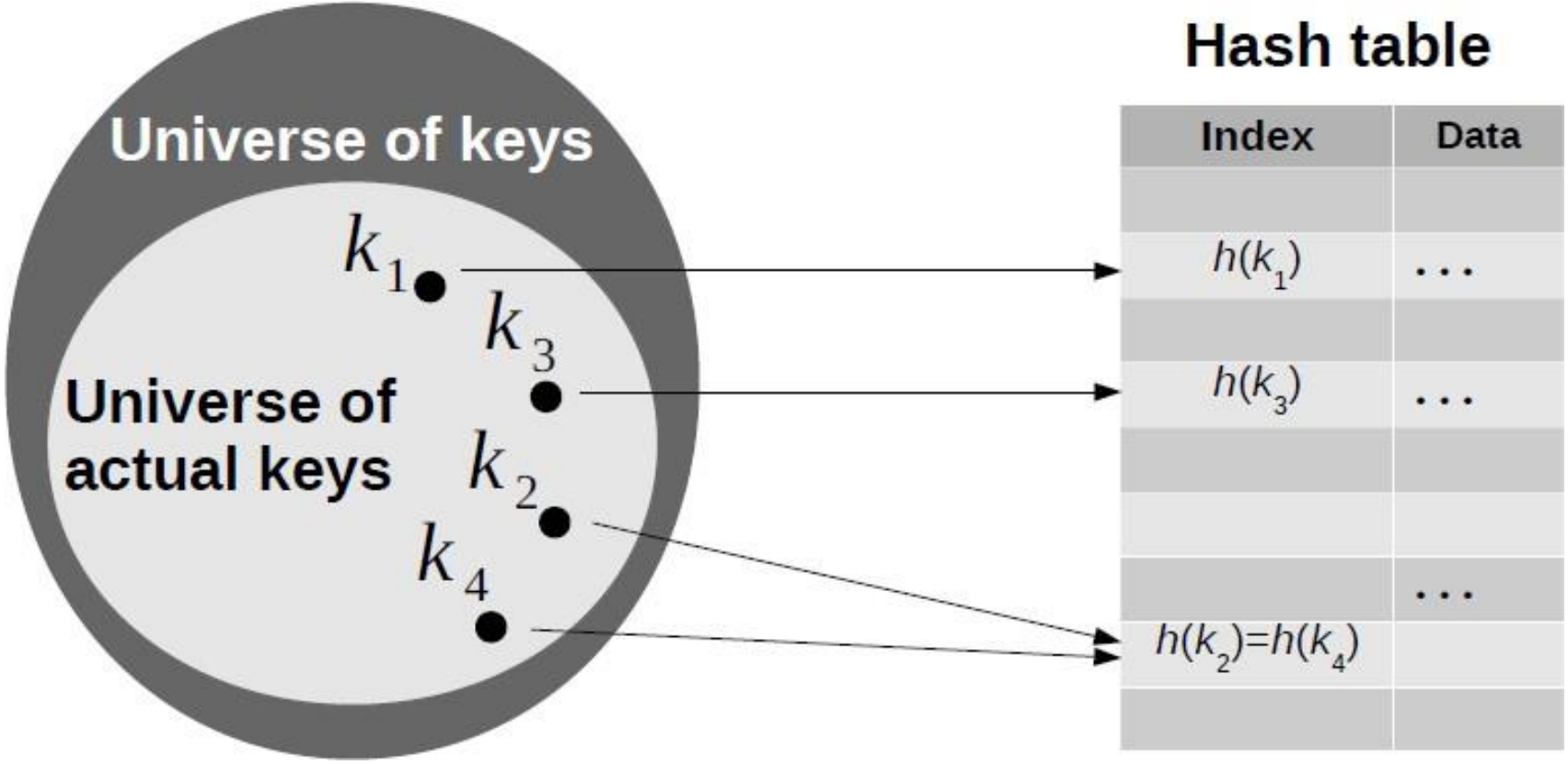
So what is Hashing?

Hashing ialah teknik simpan & cari data dengan sangat pantas.

Guna "key" untuk tentukan alamat dalam table.

Contoh: Key 123 → Hash Function → Index 3

Hash Table


Universe of keys
Universe of actual keys
k_1
k_3
k_2
k_4
Hash table
Index
Data
$h(k_1)$
$h(k_3)$
$h(k_2)=h(k_4)$
. . .
. . .
. . .

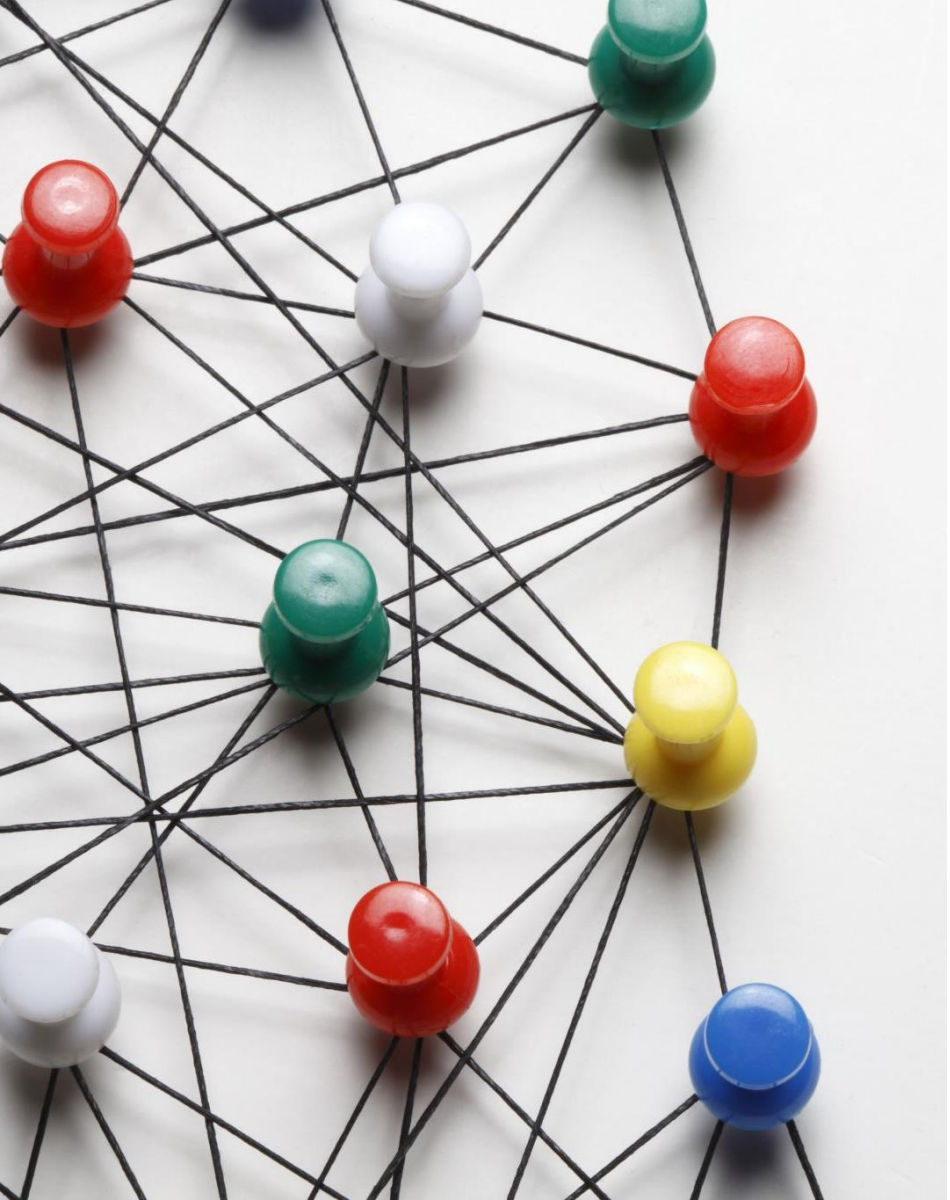
Hash Table

- A hash table is a data structure that stores **<key, information> pairs** and allows insertions, lookups, and deletions to be performed in O(1) time.
- A hash table is also called maps or dictionaries
- Hash Table Operations

  Search: compute f(k) and see if a pair exists

  Insert: compute f(k) and place it in that position

  Delete: compute f(k) and delete the pair in that position



Issues

- A problem arises when the hash function returns the same value when applied to two different keys
- To handle the situation, where two records need to be hashed to the same address we can implement a table structure, so as to have a room for two or more members at the same index positions

Key Terms

Bucket

- An index position in hash table that can store more than one record. When the same index is mapped with two keys, then both the records are stored in the same bucket.

Collision

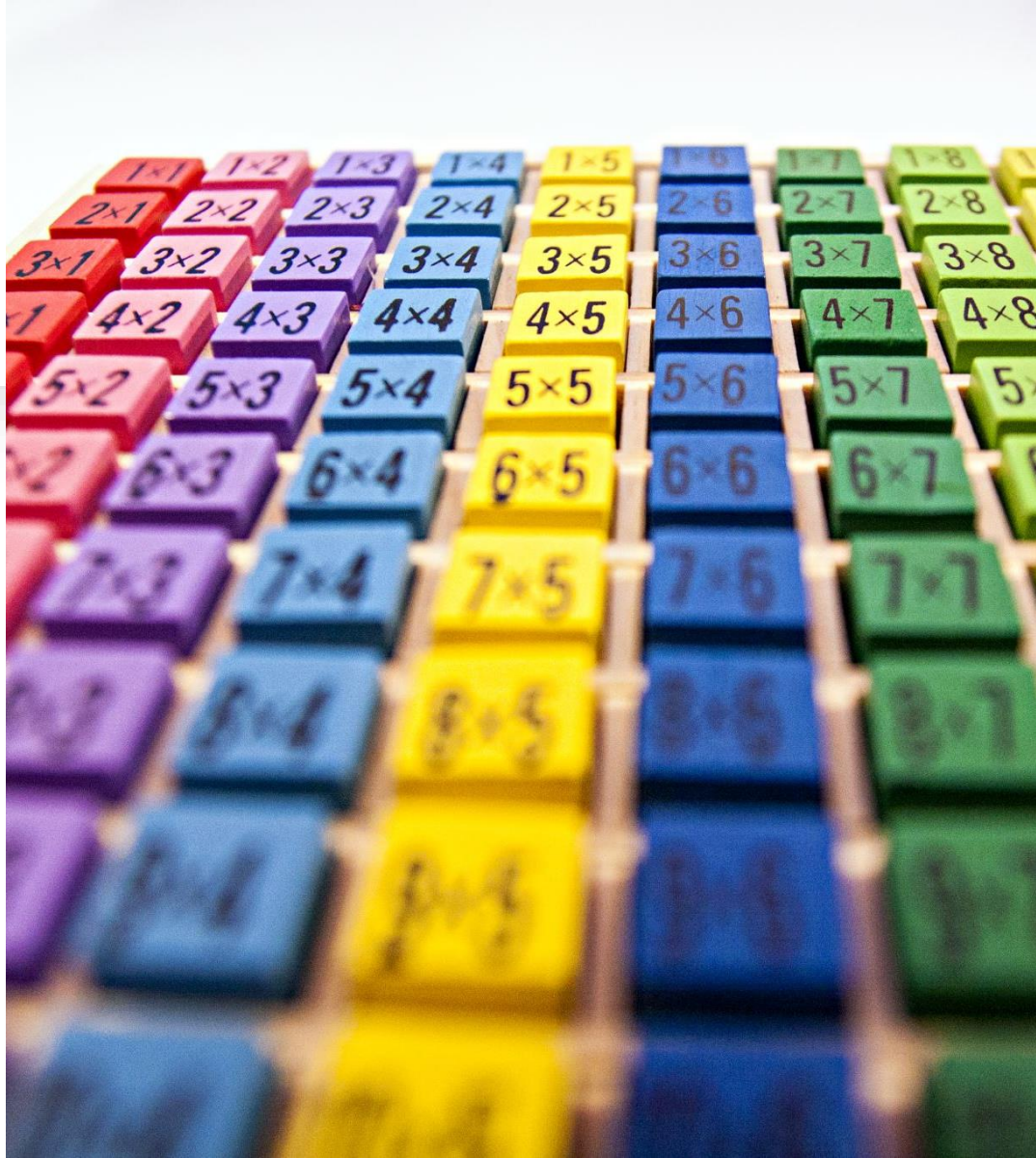
- The result of two keys hashing into the same address is called collision.

Overflow

- If there is no room in the bucket, then overflow has occurred. Collision and overflow are synonymous when the bucket is of size 1

Hashing Function Methods

- Division Method
- Mid-Square Method
- Folding Method
- Etc.
    - Multiplication Method
    - Extraction Method
    - Rotation
    - Universal Hashing



1 2 3 4 Kaedah Hashing Function Yang Popular

Kaedah	Penerangan Ringkas
1. Division Method	Guna formula: h(k) = k % m, paling mudah & biasa digunakan.
2. Mid-Square Method	Gandakan nilai key, ambil digit tengah sebagai index.
3. Folding Method	Pecah key kepada beberapa bahagian, jumlahkan, buang carry.
4. Multiplication Method	Guna formula: h(k) = floor(m * (k*A % 1)), A adalah nombor perpuluhan antara 0 dan 1.
5. Extraction Method	Ambil digit tertentu dari key sebagai index. Contoh: ambil digit ke-2 dan ke-4.
6. Rotation Method	Putar digit dalam key (rotate bits/digits), kemudian apply fungsi lain.
7. Universal Hashing	Guna satu set fungsi hash rawak untuk elak collision — biasanya untuk aplikasi teori atau kriptografi.

Division Method

Common Hashing Strategy

 Apa Itu Division Method?

- Division Method ni cara paling senang untuk cari slot dalam hash table.
- Caranya macam ni:
  - *Ambil nombor key, bahagikan dengan saiz table, dan ambil baki*.

Formula dia:

$$h(k) = k \% m$$

k = key (nombor yang nak dimasukkan)

m = saiz hash table

% = ambil baki (modulus)

 Apa Itu Division Method?

Bayangkan kau ada table saiz 8 (index 0 hingga 7)

Key = 22h

(22) = 22 % 8 = 6

Maksudnya, data akan disimpan di slot ke-6 dalam table tu.

Apa Itu Division Method?

A key k is mapped into one of the *N* buckets by taking the remainder of *k* mod *N*:

$$h(k) = k \bmod N$$

k	k mod 2	k mod 3	k mod 4	k mod 5
1	1	1	1	1
2	0	2	2	2
3	1	0	3	3
4	0	1	0	4
5	1	2	1	0

📊 Apa Maksud Jadual tu

Jadual tu tunjuk macam mana fungsi hash division berfungsi bila kita ubah-ubah nilai N (size hash table).

🔢 Formula:

$$h(k) = k \bmod N$$

Di mana:

- k = key
- N = saiz hash table
- mod = baki selepas bahagi

✅ Apa Yang Jadual ni Buat?

Dia tunjuk:

- Bila k = 1 hingga 5, apa hasil k mod N untuk pelbagai nilai N.

- Contoh:
- Bila k = 3:
- 3 mod 2 = 1 → index 1
- 3 mod 3 = 0 → index 0
- 3 mod 4 = 3 → index 3
- 3 mod 5 = 3 → index 3

Tujuan dan Kepentingan

🎯 Tujuan Jadual Ini

- Untuk tunjuk corak atau pattern macam mana nilai key k dipetakan ke index berbeza bergantung pada:

  Saiz hash table (N)

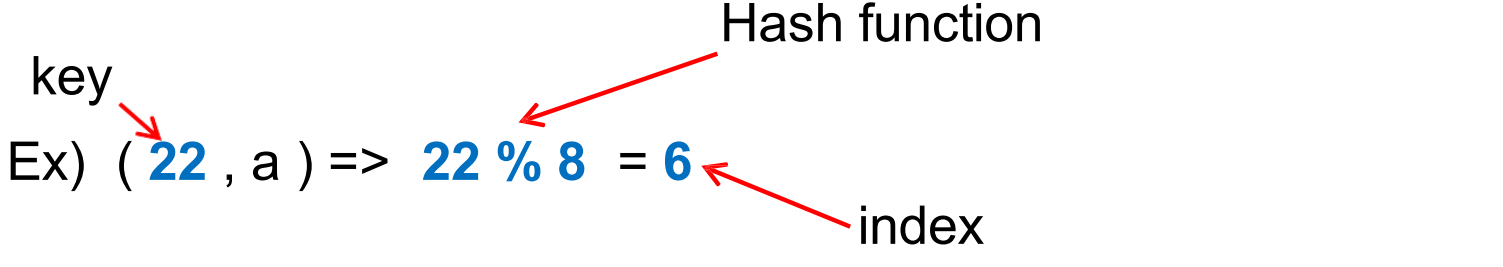
🔑 Kenapa Ini Penting dalam Division Method?

Sebab bila kita guna Division Method (k mod N), pemilihan N sangat penting — kalau N tak sesuai, collision boleh jadi kerap (contoh: N = 2 → hasil cuma 0 atau 1 aje!)

Hashing Example - Division

- Pairs are: (22,a), (33,c), (3,d), (72,e), (85,f) - (key, value) pairs
- Hash table is ht [0:7], m = 8 (m is the number of positions in the hash table)
- Hash function **h is k % m** = k % 8

Where are the pairs stored?



data	(72,e)	(33,c)		(3,d)		(85,f)	(22,a)	
index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Mid-Square Method

Common Hashing Strategy

🟩 Apa Itu Mid-Square Method?

- Bayangkan kita ada satu nombor (key), pastu kita **darabkan dengan diri sendiri** (kuasa dua), dan dari nombor panjang tu kita **ambil nombor tengah-tengah** untuk jadi index dalam hash table.

🔢 Langkah-Langkah Mid-Square Method

- Ambil key → contoh k = 123
- Darabkan dengan dirinya sendiri
  - 123 × 123 = 15129
- Ambil digit tengah dari hasil tu
  - Contoh: ambil 2 digit tengah → 51
- Jadikan 51 sebagai **alamat/index dalam hash table**

Apa Itu Mid-Square Method? (detailed)

- The key **k** is multiplied by itself and the address is obtained by selecting an appropriate number of digits from the middle of the square.

- The number of digits selected depends on the size of the table.

Example: If key=123456 is to be transformed.
$(123456)^2 = 1524\mathbf{138}3936$

- If a three-digit address is required, positions 5 to 7 could be chosen giving address **138**.

Example Mid-Square Method? (detailed)

Table size [0..99]

A..Z ---> 1,2,...26
0..9 ---> 27,...36

Key: CS1 --->3+19+28 = 31,928
$(31,928)^2$ = 1,019,397,184 (10 digits)

Extract middle 2 digits (5th and 6th) as table size is 0..99. =>
1019**39**7184 = 39

So : H(CS1) = 39.

Kenapa Guna Cara Ni?

- Kadang-kadang Division Method bagi result yang selalu sama (bila key serupa corak)
- Mid-Square kasi kita lebih variasi sebab nombor tengah tu macam “random” sikit
- Boleh kurangkan collision kalau hash table kecil

🧠 Contoh Lain (Visual)

- Key = 321
- 321 × 321 = 103041
- Ambil 3 digit tengah: 304 → Index = 304

Kalau table size kecil, kita buat:

- 304 % 100 = 4 → Ambil index 4

🛠️ Tips Untuk Guna Mid-Square

- Pilih berapa digit nak ambil ikut saiz table
  - Kalau table ada 100 slot → ambil 2 digit Tengah
  - Kalau table ada 1000 slot → ambil 3 digit tengah

Folding Method

Common Hashing Strategy

🧩 Apa Itu Folding Method?

- Bayangkan kita ada nombor yang **panjang**, susah nak masukkan terus dalam hash table. Jadi kita **potong-potong nombor tu jadi beberapa bahagian kecil**, pastu **tambah balik semua bahagian tu**, dan hasilnya jadi **alamat** dalam table.

🔢 Langkah-Langkah Folding Method

1. **Ambil nombor key**
   - Contoh: k = 356942781
2. **Bahagikan nombor tu kepada beberapa bahagian (block)**
   - Contoh: 3 digit setiap satu →P1 = 356, P2 = 942, P3 = 781
3. **Tambah semua bahagian tu**
   - 356 + 942 + 781 = 2079
4. **Kalau table kecik (cth size 100), ambil:**
   - 2079 % 100 = 79 → Index = 79

🧠 Kenapa Nama Dia Folding?

Sebab macam kita **lipat** (fold) nombor panjang tu jadi beberapa bahagian kecil, then **campurkan** balik jadi satu nombor yang sesuai untuk disimpan dalam hash table.

Folding Method

- The key **k** is partitioned into a few parts ,each of which has the same length as the required address with the possible exception of the last part .

- The parts are then added together, ignoring the final carry, to form an address.

Example:
- If key = 356942781 is to be transformed into a three digitaddress.
- P1=356, P2=942, P3=781 are added to yield 2**079**.

💡 Tips Guna Folding Method

- Biasanya guna bila key tu **panjang sangat** (contoh nombor IC, no telefon, kad matrik)
- **Pilih saiz bahagian** ikut target table (contoh nak 2-digit index, potong ikut 2-3 digit)
- Boleh juga guna teknik "fold and reverse" – potong separuh, satu bahagi kekal, satu bahagi diterbalikkan

📦 Contoh Mudah

Key = 123456

Potong: 123 + 456 = 579

Index = 579 % table_size

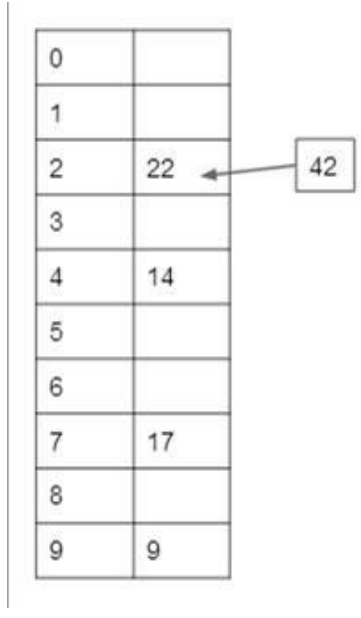
Issues

COLLISION

• A problem of collision arises when the hash function returns the same value when applied to two different keys

HOW TO HANDLE

• To handle the situation, we can implement a table structure, so as to have a room for two or more members at the same index positions



1. Chaining (Link List Method)

Idea:Setiap slot dalam hash table **bukan simpan satu item** saja, tapi **simpan satu list** (biasanya linked list).

Bila ada collision, append data baru tu dalam list kat slot tersebut.

Contoh mudah:

Hash Table:

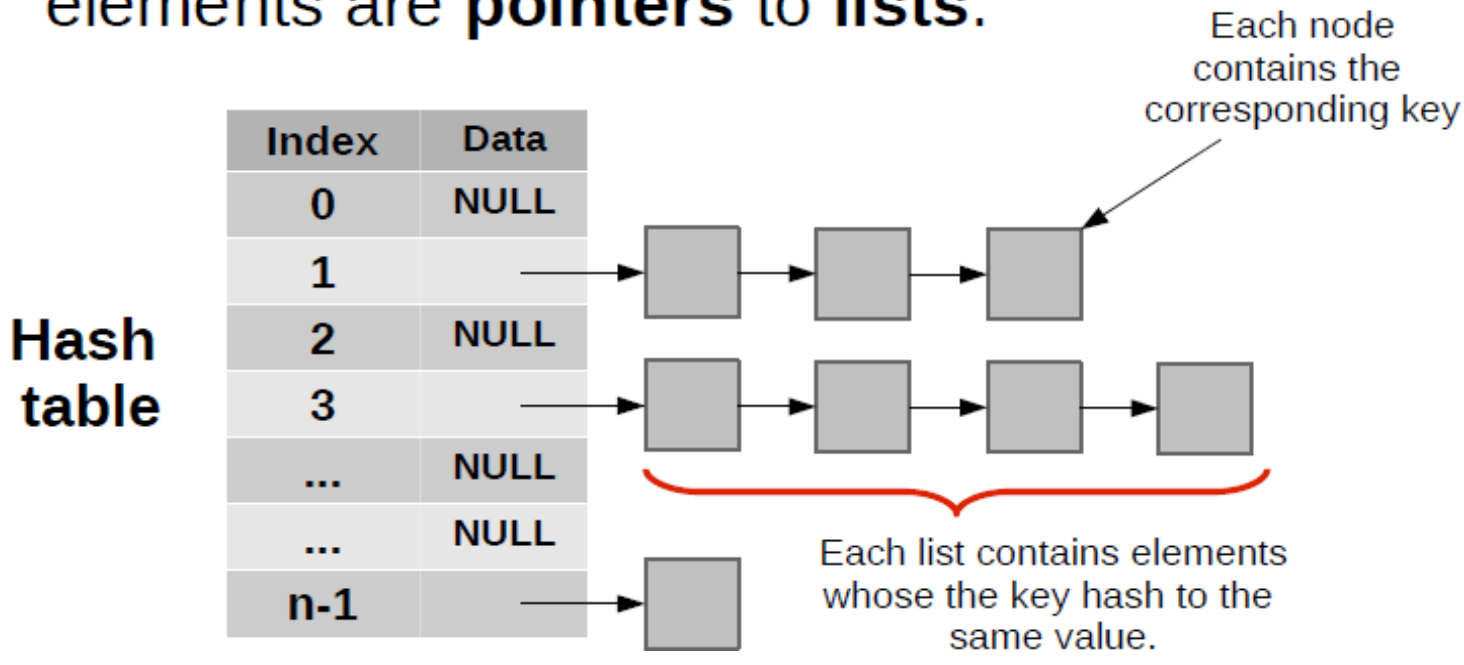
- [0] → 15 → 25 → 35
- [1] → 12
- [2] → (kosong)

Pro: Senang nak tambah.

Con: Kalau banyak collision, list jadi panjang ➔ perlahan sikit.

Chaining

- The **hash table** consists of an **array** whose elements are **pointers** to **lists**:



2. **Open Addressing (Cari tempat kosong)**

- Idea:
  - Kalau slot tu dah penuh, cari **slot lain** yang kosong ikut certain rule.
- Jenis-jenis Open Addressing:
  - Linear Probing:
    - Pergi ke **next slot** satu-satu sampai jumpa kosong.
      - index = (hash(key) + i) % table_size
    - Quadratic Probing:
      - Cari slot berdasarkan **kuasa dua**.
      - index = (hash(key) + i^2) % table_size
    - Double Hashing:
      - Guna **hash function kedua** untuk cari langkah berapa nak lompat.
      - index = (hash1(key) + i * hash2(key)) % table_size
- Pro: Semua data dalam array (senang cari).
- Con: Kalau penuh banyak sangat ➔ susah nak cari tempat kosong (dipanggil clustering).

Linear Probing

When collision is detected, instead of inserting the node in a list, the algorithm searches for a **free slot** in the **array** to **insert** the **new element**.

3. **Rehashing (Expand Table)**

- **Idea**:
  - Kalau collision terlampau banyak atau table dah penuh ➔ **buat table baru**, besar sikit, rehash semua data ke table baru.
- **Biasanya** bila load factor (jumlah data / size table) > 0.7 baru buat.

4. **Separate Hash Tables for Buckets (Advanced sikit)**

- **Idea:**
  Macam chaining, tapi setiap bucket tu guna **struktur data lain** (contohnya AVL Tree, Red-Black Tree instead of linked list) untuk kurangkan waktu carian.

Summary

Cara	Macam Mana	Kelebihan	Kekurangan
Chaining	Linked List per slot	Senang tambah	Kalau list panjang, lambat
Linear Probing	Cari slot seterusnya	Data dalam array	Boleh jadi clustering
Quadratic Probing	Lompat ikut kuasa dua	Kurang clustering	Susah sikit cari slot
Double Hashing	Lompat ikut 2nd hash	Baik untuk collision berat	Perlukan 2 hash function
Rehashing	Resize table	Kurangkan collision	Costly masa rehash

Example : Hash Function

Let h(k)= k%15. Then,

if	k	=	25	129	35	2501	47	36
	h(k)	=	10	9	5	11	2	6

Storing the keys in the array is straightforward:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36			129	25	2501			

Thus, delete and find can be done in O(1), and also insert, except…

Example : Hash Function

What happens when you try to insert: k = 65 ?

k = 65
h(k) = 5

h(k)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36			129	25	2501			

k

k = 35 already store at the location h(k) = 5

Therefore, **collision**.

Handling Collision : Chaining

Let each array element be the head of a chain

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	16	47			65	36	127		99	25	2501			14
					35				129					29

Where would you store: 29, 16, 14, 99, 127 ? New keys go at the front of the relevant chain

Chaining: Disadvantages

- Parts of the array might never be used.
- As chains get longer, search time increases to O(n) in the worst case.
- Constructing new chain nodes is relatively expensive.

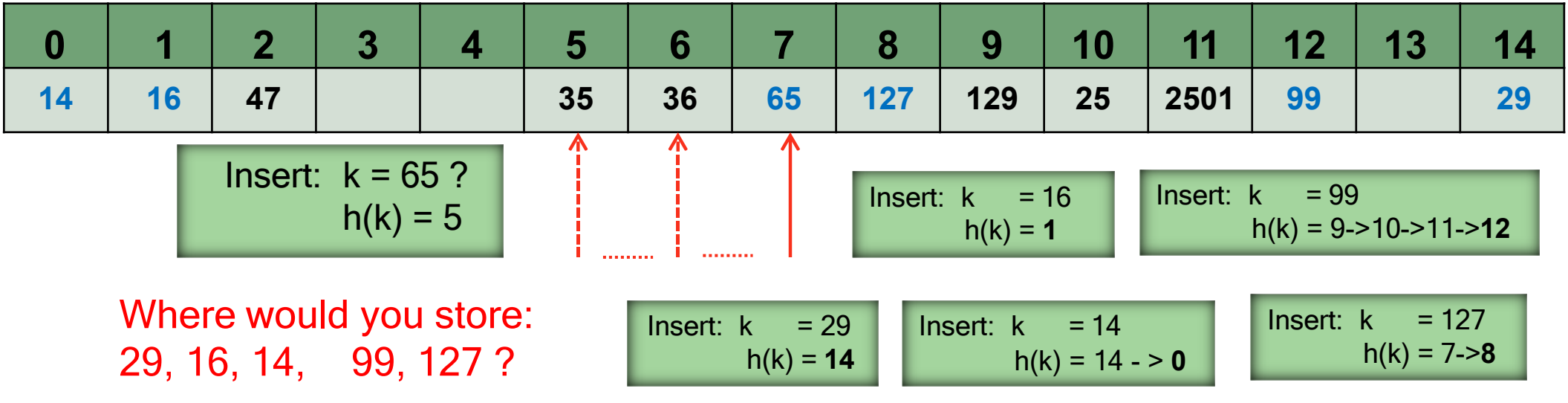
Is there a way to use the “unused” space in the array instead of using chains to make more space ?

Linear Probing

What do you do in case of a collision?

If the hash table is not full, attempt to store key in the next
Array element
(in this case (k+1)%N, (k+2)%N, (k+3)%N…) until you find an empty slot.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	16	47			35	36	65	127	129	25	2501	99		29



Hang paham?

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36	65		129	25	2501			

✅ Insert k = 65

65 % 15 = 5 → slot 5 → dah ada 35 → collision

Cuba 6 → ada 36 → penuh

Cuba 7 → kosong → simpan di index 7

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		47			35	36	65		129	25	2501			29

✅ Insert k = 29

29 % 15 = 14 → index 14 → → kosong -> terus simpan di situ 29

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	16	47			35	36	65		129	25	2501			29

✅ Insert k = 16

16 % 15 = 1 → index 1 → kosong → terus simpan di situ 16

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	16	47			35	36	65		129	25	2501			29

✅ Insert k = 14

14 % 15 = 14 → index 14 → ada 29

Cuba 0 -> kosong -> simpan di Index 0

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	16	47			35	36	65		129	25	2501	99		29

✅ Insert k = 99

99 % 15 = 9 → 129 → penuh

Cuba 10 → 25 → penuh

Cuba 11 → 2501 → penuh

Cuba 12 → kosong → simpan di index 12 ✅

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
14	16	47			35	36	65	127	129	25	2501	99		29

✅ Insert k = 127

127 % 15 = 7 → index 7 → ada 65 → collision

Cuba 8 → kosong → jadi simpan kat 8

Linear Probing : Disadvantages

- Leads to problem of clustering. Elements tend to cluster in dense intervals in the array
- Slower than chaining as one might have to walk along the table for a long time

Sample Code Hashing

```cpp
#include <iostream>
#include <vector>
using namespace std;

const int TABLE_SIZE = 10; // saiz hash table

// Kita guna vector untuk chaining (linked list pun boleh tapi vector lagi simple)
vector<int> hashTable[TABLE_SIZE];

// Hash function
int hashFunction(int key) {
    return key % TABLE_SIZE;
}

// Insert data
void insert(int key) {
    int index = hashFunction(key);
    hashTable[index].push_back(key); // tambah ke list di index tu
}

// Print hash table
void printHashTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        cout << "Index " << i << ": ";
        for (int j = 0; j < hashTable[i].size(); j++) {
            cout << hashTable[i][j] << " -> ";
        }
        cout << "NULL" << endl;
    }
}

int main() {
    int n, value;

    cout << "How many numbers you want to insert? ";
    cin >> n;

    for (int i = 0; i < n; i++) {
        cout << "Enter value " << i + 1 << ": ";
        cin >> value;
        insert(value);
    }

    cout << "\nCurrent Hash Table:\n";
    printHashTable();

    return 0;
}
```

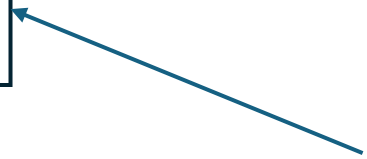
Sample Code Hashing

```cpp
#include <iostream>
#include <vector>
using namespace std;

const int TABLE_SIZE = 10; // saiz hash table

// Kita guna vector untuk chaining (linked list pun boleh tapi vector lagi simple)
vector<int> hashTable[TABLE_SIZE];

// Hash function
int hashFunction(int key) {
    return key % TABLE_SIZE;
}

// Insert data
void insert(int key) {
    int index = hashFunction(key);
    hashTable[index].push_back(key); // tambah ke list di index tu
}

// Print hash table
void printHashTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        cout << "Index " << i << ": ";
        for (int j = 0; j < hashTable[i].size(); j++) {
            cout << hashTable[i][j] << " -> ";
        }
        cout << "NULL" << endl;
    }
}

int main() {
    int n, value;

    cout << "How many numbers you want to insert? ";
    cin >> n;

    for (int i = 0; i < n; i++) {
        cout << "Enter value " << i + 1 << ": ";
        cin >> value;
        insert(value);
    }

    cout << "\nCurrent Hash Table:\n";
    printHashTable();

    return 0;
}
```

- Ini declaration hash table kita.
- hashTable ada 10 slot (index 0 hingga 9).
- Setiap slot ialah vector — untuk simpan banyak data kalau ada collision (chaining style).

Sample Code Hashing

```cpp
#include <iostream>
#include <vector>
using namespace std;

const int TABLE_SIZE = 10; // saiz hash table

// Kita guna vector untuk chaining (linked list pun boleh tapi vector lagi simple)
vector<int> hashTable[TABLE_SIZE];

// Hash function
int hashFunction(int key) {
    return key % TABLE_SIZE;
}

// Insert data
void insert(int key) {
    int index = hashFunction(key);
    hashTable[index].push_back(key); // tambah ke list di index tu
}

// Print hash table
void printHashTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        cout << "Index " << i << ": ";
        for (int j = 0; j < hashTable[i].size(); j++) {
            cout << hashTable[i][j] << " -> ";
        }
        cout << "NULL" << endl;
    }
}

int main() {
    int n, value;

    cout << "How many numbers you want to insert? ";
    cin >> n;

    for (int i = 0; i < n; i++) {
        cout << "Enter value " << i + 1 << ": ";
        cin >> value;
        insert(value);
    }

    cout << "\nCurrent Hash Table:\n";
    printHashTable();

    return 0;
}
```



- Ini fungsi untuk kira index.
- Dia ambil key dari user, buat mod TABLE_SIZE (dalam kes ni % 10).
- So key 23 ➔ 23 % 10 = 3 ➔ Masuk index 3.

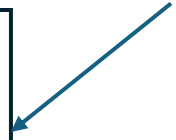
Sample Code Hashing

```cpp
#include <iostream>
#include <vector>
using namespace std;

const int TABLE_SIZE = 10; // saiz hash table

// Kita guna vector untuk chaining (linked list pun boleh tapi vector lagi simple)
vector<int> hashTable[TABLE_SIZE];

// Hash function
int hashFunction(int key) {
    return key % TABLE_SIZE;
}

// Insert data
void insert(int key) {
    int index = hashFunction(key);
    hashTable[index].push_back(key); // tambah ke list di index tu
}

// Print hash table
void printHashTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        cout << "Index " << i << ": ";
        for (int j = 0; j < hashTable[i].size(); j++) {
            cout << hashTable[i][j] << " -> ";
        }
        cout << "NULL" << endl;
    }
}

int main() {
    int n, value;

    cout << "How many numbers you want to insert? ";
    cin >> n;

    for (int i = 0; i < n; i++) {
        cout << "Enter value " << i + 1 << ": ";
        cin >> value;
        insert(value);
    }

    cout << "\nCurrent Hash Table:\n";
    printHashTable();

    return 0;
}
```

- Ini fungsi untuk masukkan data ke hash table.
- Mula-mula kira index pakai hashFunction.
- Lepas dapat index, kita push_back masuk dalam vector kat slot tu.
- Kalau dah ada data lain kat situ (collision), dia tetap tambah di belakang.

Sample Code Hashing

```cpp
#include <iostream>
#include <vector>
using namespace std;

const int TABLE_SIZE = 10; // saiz hash table

// Kita guna vector untuk chaining (linked list pun boleh tapi vector lagi simple)
vector<int> hashTable[TABLE_SIZE];

// Hash function
int hashFunction(int key) {
    return key % TABLE_SIZE;
}

// Insert data
void insert(int key) {
    int index = hashFunction(key);
    hashTable[index].push_back(key); // tambah ke list di index tu
}

// Print hash table
void printHashTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        cout << "Index " << i << ": ";
        for (int j = 0; j < hashTable[i].size(); j++) {
            cout << hashTable[i][j] << " -> ";
        }
        cout << "NULL" << endl;
    }
}

int main() {
    int n, value;

    cout << "How many numbers you want to insert? ";
    cin >> n;

    for (int i = 0; i < n; i++) {
        cout << "Enter value " << i + 1 << ": ";
        cin >> value;
        insert(value);
    }

    cout << "\nCurrent Hash Table:\n";
    printHashTable();

    return 0;
}
```



- Ini fungsi untuk print semua isi dalam hash table.
- Loop luar ➔ jalan semua index dari 0 sampai 9.
- Loop dalam ➔ jalan semua item dalam vector kat index tu (kalau ada banyak).
- Setiap data sambung ->, lepas habis data tulis NULL (nampak lebih cantik dan mudah baca).

Sample Code Hashing

```cpp
#include <iostream>
#include <vector>
using namespace std;

const int TABLE_SIZE = 10; // saiz hash table

// Kita guna vector untuk chaining (linked list pun boleh tapi vector lagi simple)
vector<int> hashTable[TABLE_SIZE];

// Hash function
int hashFunction(int key) {
    return key % TABLE_SIZE;
}

// Insert data
void insert(int key) {
    int index = hashFunction(key);
    hashTable[index].push_back(key); // tambah ke list di index tu
}

// Print hash table
void printHashTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        cout << "Index " << i << ": ";
        for (int j = 0; j < hashTable[i].size(); j++) {
            cout << hashTable[i][j] << " -> ";
        }
        cout << "NULL" << endl;
    }
}
```

```cpp
int main() {
    int n, value;

    cout << "How many numbers you want to insert? ";
    cin >> n;

    for (int i = 0; i < n; i++) {
        cout << "Enter value " << i + 1 << ": ";
        cin >> value;
        insert(value);
    }

    cout << "\nCurrent Hash Table:\n";
    printHashTable();

    return 0;
}
```

- Ini bahagian utama program start.
- Tanya user: "Berapa banyak nombor nak masuk?“
- Loop untuk ambil input satu-satu ➔ panggil insert(value) ➔ masuk ke table.
- Lepas semua data masuk ➔ print keseluruhan hash table.

The End

Thank you