BIRZEIT UNIVERSITY

COMP242

Data Structure

Lectures Note: Hash Tables

Prepared by: **Dr. Mamoun Nawahdah**

2016/2017

Hash Tables

- **Hashing**: is a technique that determines element **index** using only element's distinct **search key**.
- **Hash function**:
  - Takes a **search key** and produces the integer **index** of an element in the **hash table.**
  - Search key-maps, or hashes, to the index.

Example 1: Phone numbers (xxx-xxxx).
- **Bad**: first three digits. // identical for same area
- **Better**: last four digits. // distinct

Example 2: University ID numbers (114-xxxx).
- **Bad**: first three digits. // identical for same period
- **Better**: last four digits. // distinct

Practical challenge: Need different approaches for each key type.

Simple algorithms for the hash operations that **add** and **retrieve**:

```
Algorithm add(key, value)
index = h(key)
hashTable[index] = value

Algorithm getValue(key)
index = h(key)
return hashTable[index]
```

Typical Hashing

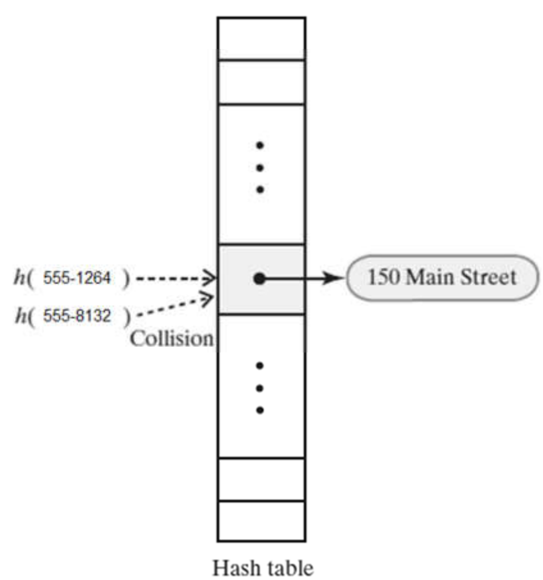
Typical hash functions perform two steps:
1. **Convert search key** to an integer called the **hash code**.
2. **Compress hash code** into the range of indices for hash table.

```
Algorithm getHashIndex(phoneNumber)
// Returns an index to an array of tableSize loc

i = last four digits of phoneNumber
return i % tableSize
```

- Typical hash functions are **not perfect**:
  - Can allow more than one **search key** to map into a **single index.**
  - Causes a **collision** in the hash table.

Example: Consider table (array) size = **101**
- **getHashIndex(555-1264)** = 52
- **getHashIndex(555-8132**) = 52 also!!!



Hash Functions

- A good hash function should:
  - **Minimize collisions**
  - **Be fast to compute**
- To reduce the chance of a collision
  - Choose a hash function that distributes entries **uniformly** throughout hash table.

Java's hash code conventions

All Java classes inherit a method ***hashCode()*** from **Object** class, which returns a **32-bit** int.

Default implementation: **Memory address**.

Customized implementations: Integer, Double, String, File, URL, Date, …

User-defined types: Users are on their own.

Java library implementations:

Integer

```java
public final class Integer
{
   private final int value;
   ...

   public int hashCode()
   {  return value;  }
}
```

Boolean

```java
public final class Boolean
{
   private final boolean value;
   ...

   public int hashCode()
   {
      if (value) return 1231;
      else       return 1237;
   }
}
```

Double

```java
public final class Double
{
   private final double value;
   ...

   public int hashCode()
   {
      long bits = doubleToLongBits(value);
      return (int) (bits ^ (bits >>> 32));
   }
}
```

convert to IEEE 64-bit representation;
xor most significant 32-bits
with least significant 32-bits

String

```
public final class String
{
   private final char[] s;
   ...

   public int hashCode()
   {
      int hash = 0;
      for (int i = 0; i < length(); i++)
         hash = s[i] + (31 * hash);
      return hash;
   }
}
```

i^{th} character of s

char	Unicode
...	...
'a'	97
'b'	98
'c'	99
...	...

Horner's method to hash a String of length ***L***:

$$h = s[0] \cdot 31^{L-1} + \ldots + s[L-3] \cdot 31^2 + s[L-2] \cdot 31^1 + s[L-1] \cdot 31^0.$$

Example:

```
String s = "call";
int code = s.hashCode();
```

$$\begin{aligned} 3045982 &= \overset{c}{99} \cdot 31^3 + \overset{a}{97} \cdot 31^2 + \overset{l}{108} \cdot 31^1 + \overset{l}{108} \cdot 31^0 \\ &= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99))) \end{aligned}$$

Implementing hash code: user-defined types

Hash code design

"**Standard**" recipe for user-defined types:

- Combine each significant field using the ***31x + y*** rule.
- If field is a primitive type, use **wrapper** type **hashCode()**.
- If field is **null**, return **0**.
- If field is a reference type, use **hashCode()**.
- If field is an array, apply to each entry. ← or use **Arrays.deepHashCode()**

Example:

```
public final class Transaction {
   private final String  who;
   private final Date    when;
   private final double  amount;

   public int hashCode()
   {
      int hash = 17;
      hash = 31*hash + who.hashCode();
      hash = 31*hash + when.hashCode();
      hash = 31*hash + ((Double) amount).hashCode();
      return hash;
   }
}
```

nonzero constant

for reference types, use hashCode()

for primitive types, use hashCode() of wrapper type

typically a small prime

Compressing a Hash Code

Hash code: An **int** between **-2^{31}** and **2^{31} - 1**.

Hash function: returns an **int** between **0** and **M-1** (for use as array index).

- Common way to scale an integer
  - Use Java **%** operator ➔ **hash code % m**
- Avoid **m** as power of **2** or **10**
- Best to use an **odd** number for **m**
- **Prime numbers** often give good distribution of hash values

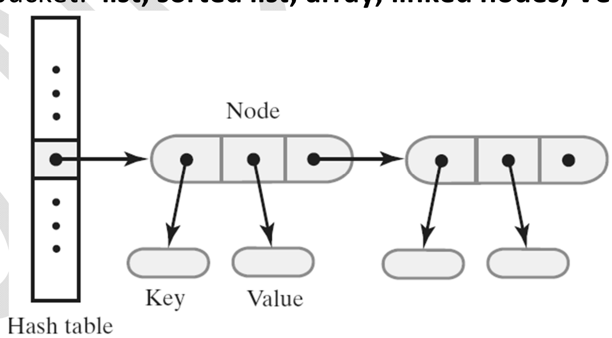
```
private int hash(Key key)
{  return (key.hashCode() & 0x7fffffff) % M;  }
```

Resolving Collisions

- **Collisions**: Two distinct **keys** hashing to the same **index**.
- Two choices:
  - Change the structure of the hash table so that each array location can represent more than one value. (**Separate Chaining**)
  - Use another empty location in the hash table. (**Open Addressing**)

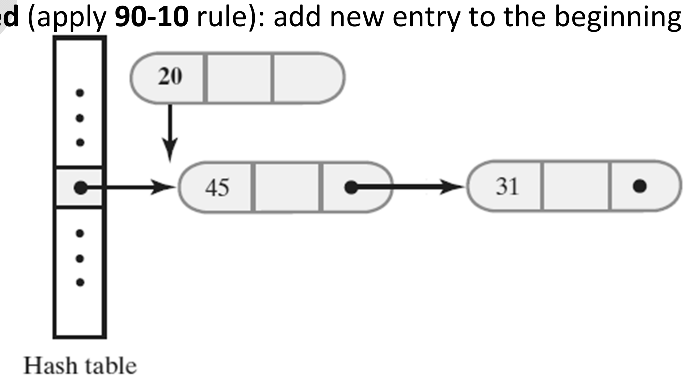
Separate Chaining

- Alter the structure of the hash table:
  - Each location can represent more than one value.
  - Such a location is called a ***bucket***
- Decide how to represent a bucket: **list, sorted list; array; linked nodes; Vector; etc.**

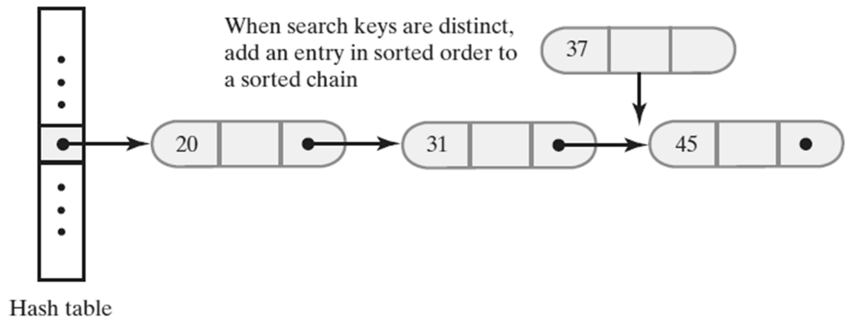


Where to insert a new entry into a linked bucket?

(a) If **unsorted** (apply **90-10** rule): add new entry to the beginning of chain



(b) If **sorted**:



Time Complexity

Worst case: all keys mapped to the same location ➔ one long list of size **N**

Find(key) ➔ **O(n)** ☹

Best case: hashing uniformly distribute records over the hash table ➔ each list long = **N/M** = **α** **(α** is load factor**)**

Find(key) ➔ **O(1 + α)** ☺

Design Consequences

- **M** too large ➔ too many empty chains.
- **M** too small ➔ chains too long.
- Typical choice: **M ≈ N / 5** ➔ constant-time ops.

Open Addressing

Linear Probing

- When a new key collides, find **next** empty slot, and put it there.
- **Hash**: Map key to integer **k** between **0** and **M-1**.
- **Insert**: Put at table index **k** if free; **if not** try **k+1**, **k+2**, etc.
  - If reaches end of table, go to beginning of table (**Circular hash table**)
- Hash function: ***h(k , i)* = (*h(k , 0)*+i) % m**
- Array size **M** must be greater than number of key-value pairs **N**.

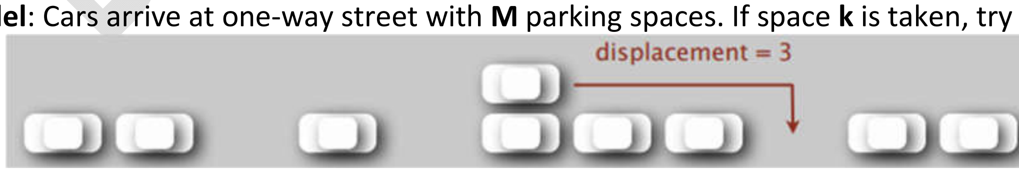
Example: Linear hash table demo: **take last 2 digits of student's ID and run a demo**

	0	1	2	3	4	5	6	7	8	9
st[]										

Clustering problem: A contiguous block of items will be easily formed which in turn will affect performance.

Knuth's Parking Problem

- **Model**: Cars arrive at one-way street with **M** parking spaces. If space **k** is taken, try **k+1**, **k+2**, etc.

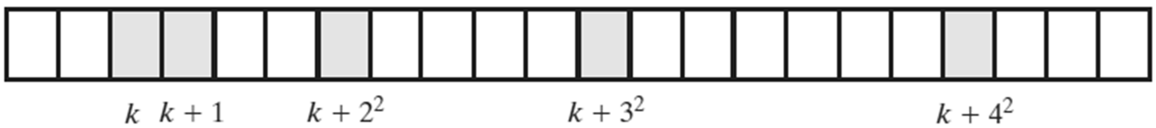


Parameters.

- M too large ⇒ too many empty array entries.
- M too small ⇒ search time blows up.
- Typical choice: $\alpha = N / M \sim ½$. ⟵ # probes for search hit is about 3/2 # probes for search miss is about 5/2

➢ Quadratic Probing

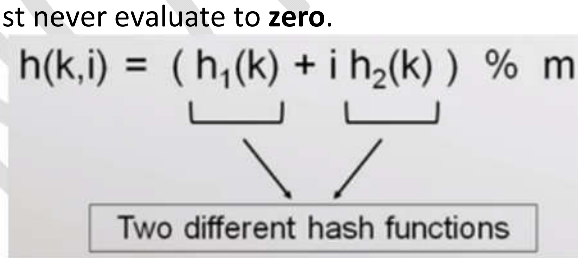
- Linear probing looks at **consecutive** locations beginning at index ***k***
- Quadratic probing, considers the locations at indices **$k + j^2$**
  - Uses the indices ***k, k*+1, *k* + 4, *k* + 9**, …

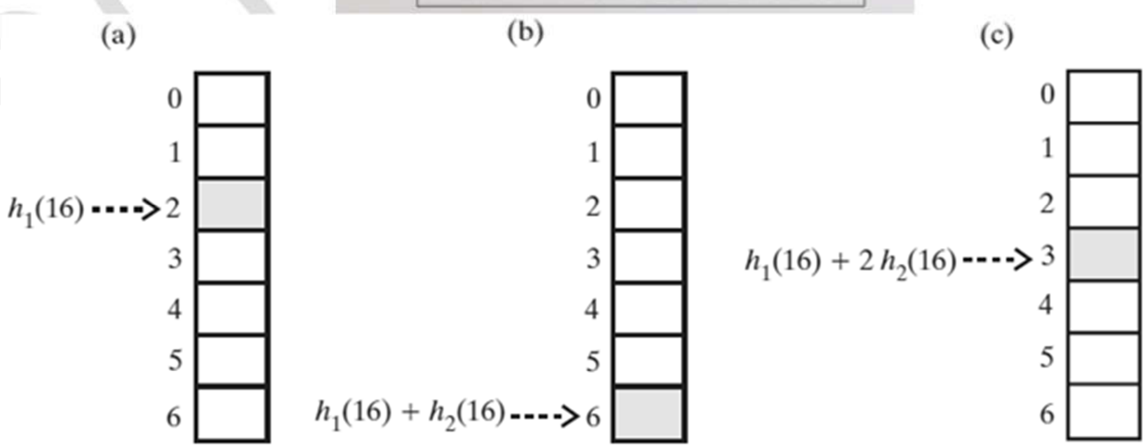


- Hash function: **$h(k, i) = (h(k, 0)+i^2)$ % m**
- For linear probing it is a bad idea to let the hash table get nearly full, because performance degrades.
- For quadratic probing, the situation is even worse: There is no guarantee of finding an empty cell once the table gets more than half full, or even before the table gets half full if the table size is not **prime**.
- Standard **deletion** cannot be performed in a probing hash table, because the cell might have caused a collision to go past it. (instead **soft deletion** is used)

Double Hashing

- **Linear probing** and **quadratic probing** add increments to **k** to define a probe sequence
  - Both are ***independent*** of the search key
- **Double hashing** uses a **second hash function** to compute these increments
  - This is a key-***dependent*** method.
  - The 2nd hash function must never evaluate to **zero**.





The 1st three locations in a probe sequence generated by double hashing for the search key 16

Potential Problem with Open Addressing

- Note that each location is either **occupied**, **empty (null)**, or **available (removed)**
    - Frequent additions and removals can result in *no* locations that are **null**
- Thus searching a probe sequence will not work
- Consider separate chaining as a solution

Time Complexity

Worst case: O(n)

Average case:

$$\text{Number of probes} \le \frac{1}{1-\alpha} \qquad \alpha = n/m$$

if, $\alpha < 1$ (i.e. $n < m$)

If the table is 50% full, $\alpha = 0.5$

Number of probes ≤ 2

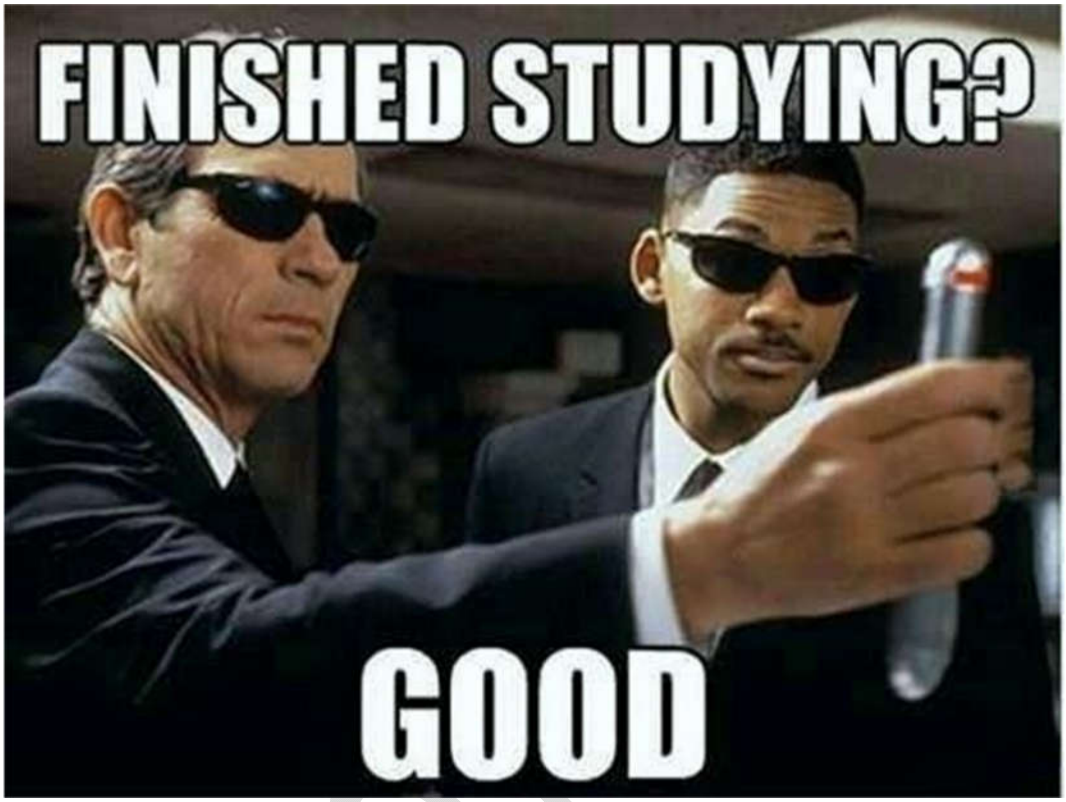
If the table is 80% full, $\alpha = 0.8$

Number of probes ≤ 5

$\alpha \to 1$ (near full space utilization), Performance $\downarrow$

Rehashing

- If the table gets **too full**, the running time for the operations will start taking too long and insertions might fail for open addressing hashing with quadratic resolution.
- A solution, then, is to build another table that is about **twice as big** (with an associated new hash function) and scan down the entire original hash table, computing the new hash value for each (non-deleted) element and inserting it in the new table.
- This entire operation is called **rehashing**.
    - This is obviously a very expensive operation; the running time is **O(N)**, since there are **N** elements to rehash and the table size is roughly **2N**, but it is actually not all that bad, because it happens very infrequently.


FINISHED STUDYING?
GOOD