
Data Structures and Algorithms
(ES221)

# HASHING

Dr. Zubair Ahmad

# Hashing

- A hash function is any algorithm or subroutine that maps large data sets, called keys, to smaller data sets.
  - Example: a single integer can serve as an index to an array.
- The values returned by a hash function are called
  - hash values
  - hash codes
  - hash sums
  - Checksums, or
  - simply hashes.
- Hash functions are mostly used to accelerate table lookup or data comparison tasks such as
  - finding items in a database,
  - detecting duplicated or similar records in a large file,
  - finding similar stretches in DNA sequences,
  - and so on.


2

# Hash Table

- Ideal hash table data structure
  - → an array of some fixed size, containing the keys.
- key →
  - a string with an associated value
    - for instance, salary information
- *H_SIZE* → table size
  - part of a hash data structure
- The table runs from 0 to *H_SIZE-1*
- Each key is mapped into some number in the range 0 to *H_SIZE - 1* and placed in the appropriate cell.
- Each hash table entry is a unique ordered pair *(Key, Information)*
- The mapping is called a *hash function*


3

# Hashing Function

- A hash function should be referentially transparent,
  - i.e. if called twice on input that is "equal" it should give the same result.
  - → if two objects are equal their hash codes must be the same.


4

# Hash Table

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | |
| 3 | John 25000 |
| 4 | Phil 31250 |
| 5 | |
| 6 | Dave 27500 |
| 7 | Marry 28200 |
| 8 | |
| 9 | |


5

# Hashing – Collision

- Map two or more keys to the same hash value, causing a collision.
- Although the idea was conceived in the 1950s,[1] the design of good hash functions is still a topic of active research.


6

# Hashing Function

- The simplest hash function
  - *key mod H_SIZE*
- Example 1
  - *H_SIZE* = 9, Key = 100
    - 100 *mod 9* = 1
  - *H_SIZE* = 9, Key = 80
    - 80 *mod* 9 = 8
- Example 2
  - *H_SIZE* = 10, Key = 100
    - 100 *mod* 10 = 0
  - *H_SIZE* = 10, Key = 80
    - 80 *mod 10* = 0
- What was wrong?


7

# Hashing Function

- A better idea →
  - Ensure that the table size is prime.
  - If the input keys are random integers
    - function is very simple to compute, and
    - distributes the keys evenly

- Usually, the keys are strings
  - → the hash function needs to be chosen carefully


8

# Another Hash Function

- Usually, the keys are strings;
  - → the hash function needs to be chosen carefully

- Add up the ASCII values of the characters in the string.

```
typedef unsigned int INDEX;
INDEX hash( char *key, unsigned int H_SIZE )
{
unsigned int hash_val = 0;
/*1*/    while( *key != '\0' )
/*2*/                    hash_val += *key++;
/*3*/     return( hash_val % H_SIZE );
}
```


9

| Dec | Hx | Oct | Char | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 000 | NUL (null) | 32 | 20 | 040 |   | Space | 64 | 40 | 100 | @ | @ | 96 | 60 | 140 | ` | ` |
| 1 | 1 | 001 | SOH (start of heading) | 33 | 21 | 041 | ! | ! | 65 | 41 | 101 | A | A | 97 | 61 | 141 | a | a |
| 2 | 2 | 002 | STX (start of text) | 34 | 22 | 042 | " | " | 66 | 42 | 102 | B | B | 98 | 62 | 142 | b | b |
| 3 | 3 | 003 | ETX (end of text) | 35 | 23 | 043 | # | # | 67 | 43 | 103 | C | C | 99 | 63 | 143 | c | c |
| 4 | 4 | 004 | EOT (end of transmission) | 36 | 24 | 044 | $ | $ | 68 | 44 | 104 | D | D | 100 | 64 | 144 | d | d |
| 5 | 5 | 005 | ENQ (enquiry) | 37 | 25 | 045 | % | % | 69 | 45 | 105 | E | E | 101 | 65 | 145 | e | e |
| 6 | 6 | 006 | ACK (acknowledge) | 38 | 26 | 046 | & | & | 70 | 46 | 106 | F | F | 102 | 66 | 146 | f | f |
| 7 | 7 | 007 | BEL (bell) | 39 | 27 | 047 | ' | ' | 71 | 47 | 107 | G | G | 103 | 67 | 147 | g | g |
| 8 | 8 | 010 | BS (backspace) | 40 | 28 | 050 | ( | ( | 72 | 48 | 110 | H | H | 104 | 68 | 150 | h | h |
| 9 | 9 | 011 | TAB (horizontal tab) | 41 | 29 | 051 | ) | ) | 73 | 49 | 111 | I | I | 105 | 69 | 151 | i | i |
| 10 | A | 012 | LF (NL line feed, new line) | 42 | 2A | 052 | * | * | 74 | 4A | 112 | J | J | 106 | 6A | 152 | j | j |
| 11 | B | 013 | VT (vertical tab) | 43 | 2B | 053 | + | + | 75 | 4B | 113 | K | K | 107 | 6B | 153 | k | k |
| 12 | C | 014 | FF (NP form feed, new page) | 44 | 2C | 054 | , | , | 76 | 4C | 114 | L | L | 108 | 6C | 154 | l | l |
| 13 | D | 015 | CR (carriage return) | 45 | 2D | 055 | - | - | 77 | 4D | 115 | M | M | 109 | 6D | 155 | m | m |
| 14 | E | 016 | SO (shift out) | 46 | 2E | 056 | . | . | 78 | 4E | 116 | N | N | 110 | 6E | 156 | n | n |
| 15 | F | 017 | SI (shift in) | 47 | 2F | 057 | / | / | 79 | 4F | 117 | O | O | 111 | 6F | 157 | o | o |
| 16 | 10 | 020 | DLE (data link escape) | 48 | 30 | 060 | 0 | 0 | 80 | 50 | 120 | P | P | 112 | 70 | 160 | p | p |
| 17 | 11 | 021 | DC1 (device control 1) | 49 | 31 | 061 | 1 | 1 | 81 | 51 | 121 | Q | Q | 113 | 71 | 161 | q | q |
| 18 | 12 | 022 | DC2 (device control 2) | 50 | 32 | 062 | 2 | 2 | 82 | 52 | 122 | R | R | 114 | 72 | 162 | r | r |
| 19 | 13 | 023 | DC3 (device control 3) | 51 | 33 | 063 | 3 | 3 | 83 | 53 | 123 | S | S | 115 | 73 | 163 | s | s |
| 20 | 14 | 024 | DC4 (device control 4) | 52 | 34 | 064 | 4 | 4 | 84 | 54 | 124 | T | T | 116 | 74 | 164 | t | t |
| 21 | 15 | 025 | NAK (negative acknowledge) | 53 | 35 | 065 | 5 | 5 | 85 | 55 | 125 | U | U | 117 | 75 | 165 | u | u |
| 22 | 16 | 026 | SYN (synchronous idle) | 54 | 36 | 066 | 6 | 6 | 86 | 56 | 126 | V | V | 118 | 76 | 166 | v | v |
| 23 | 17 | 027 | ETB (end of trans. block) | 55 | 37 | 067 | 7 | 7 | 87 | 57 | 127 | W | W | 119 | 77 | 167 | w | w |
| 24 | 18 | 030 | CAN (cancel) | 56 | 38 | 070 | 8 | 8 | 88 | 58 | 130 | X | X | 120 | 78 | 170 | x | x |
| 25 | 19 | 031 | EM (end of medium) | 57 | 39 | 071 | 9 | 9 | 89 | 59 | 131 | Y | Y | 121 | 79 | 171 | y | y |
| 26 | 1A | 032 | SUB (substitute) | 58 | 3A | 072 | : | : | 90 | 5A | 132 | Z | Z | 122 | 7A | 172 | z | z |
| 27 | 1B | 033 | ESC (escape) | 59 | 3B | 073 | ; | ; | 91 | 5B | 133 | [ | [ | 123 | 7B | 173 | { | { |
| 28 | 1C | 034 | FS (file separator) | 60 | 3C | 074 | < | < | 92 | 5C | 134 | \ | \ | 124 | 7C | 174 | | | \| |
| 29 | 1D | 035 | GS (group separator) | 61 | 3D | 075 | = | = | 93 | 5D | 135 | ] | ] | 125 | 7D | 175 | } | } |
| 30 | 1E | 036 | RS (record separator) | 62 | 3E | 076 | > | > | 94 | 5E | 136 | ^ | ^ | 126 | 7E | 176 | ~ | ~ |
| 31 | 1F | 037 | US (unit separator) | 63 | 3F | 077 | ? | ? | 95 | 5F | 137 | _ | _ | 127 | 7F | 177 | &#127; | DEL |

**Source: www.LookupTables.com**

# Another Hash Function

- Usually, the keys are strings;
  - → the hash function needs to be chosen carefully

- Add up the ASCII values of the characters in the string.

```
typedef unsigned int INDEX;
INDEX hash( char *key, unsigned int H_SIZE )
{
unsigned int hash_val = 0;
/*1*/    while( *key != '\0' )
/*2*/                    hash_val += *key++;
/*3*/     return( hash_val % H_SIZE );
}
```

**Example:**
***H_SIZE* = 100**
**Key = "abcdefgh"**
**hash_val= 97+ 98 +.. 104 =804**
**Index = 804%100 =4**

**Example:**
***H_SIZE* = 100**
**Key = "bcdefghi"**
**hash_val= 98+ 98 +.. 105 =812**
**Index = 812%100 =12**


11

# Another Hash Function

- Usually, the keys are strings;
  - → the hash function needs to be chosen carefully
  - Add up the ASCII values of the characters in the string.

```
typedef unsigned int INDEX;
INDEX hash( char *key, unsigned int H_SIZE )
{
unsigned int hash_val = 0;
/*1*/    while( *key != '\0' )
/*2*/                    hash_val += *key++;
/*3*/     return( hash_val % H_SIZE );
}
```

**Example:**
***H_SIZE*** **= 10,007**
**Key = 8 NULs**
**hash_val= 0 x 8 = 0**
**Index = 0% 10,007 = 0**

**Example:**
***H_SIZE*** **= 10,007**
**Key = 8 DELs**
**hash_val= 127 x 8 = 1016**
**Index = 1016 % 10,007 = 1016**

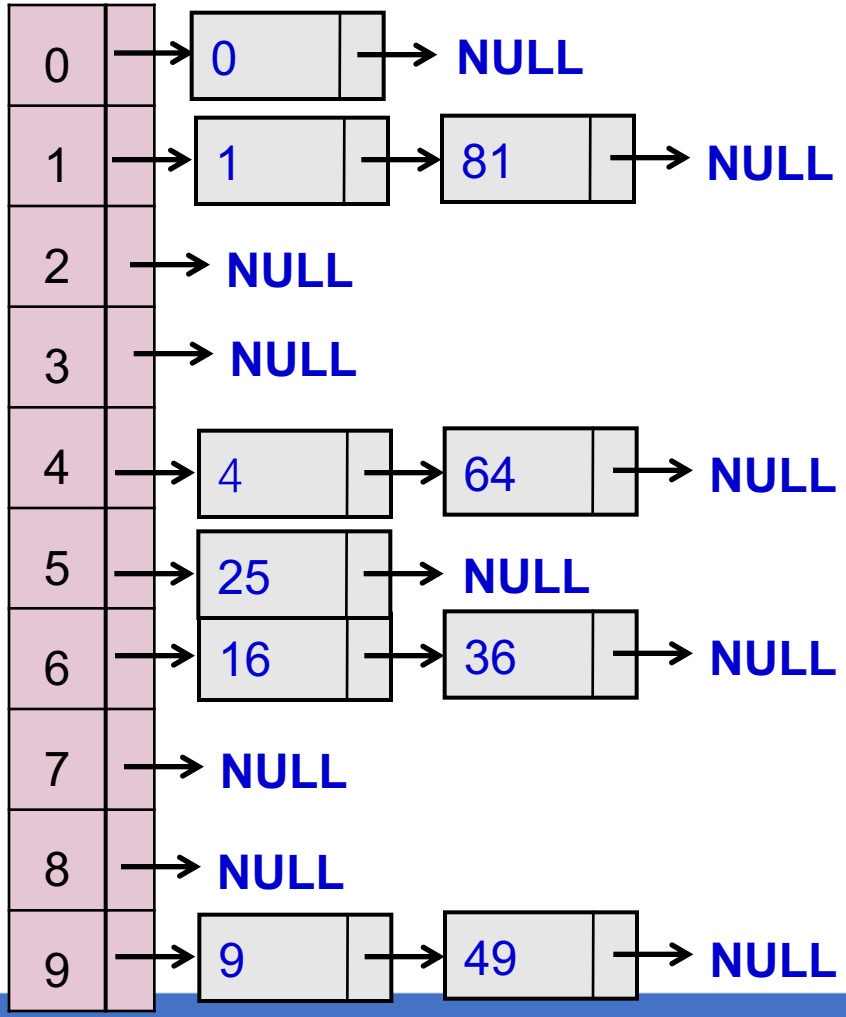
➔ the hash function can only assume values between 0 and 1016


12

# Open Hashing (Separate Chaining)

- The first strategy, commonly known as either *open hashing,* or *separate chaining*, is to keep a list of all elements that hash to the same value.
- For convenience, our lists have headers
- For simplicity, only for this section, we assume that
  - the keys are the first 10 perfect squares, and
  - the hashing function is simply *hash*(*x*) = *x mod* 10
    - The table size is not prime, but is used here for simplicity

# Open Hashing (Separate Chaining)




## Questions?

zahmaad.github.io