Topic 5

Space and Time Tradeoffs

المقايضة بالوقت والمساحة

Space-for-time tradeoffs

Two varieties of space-for-time algorithms:

- ***input enhancement*** — preprocess the input (or its part) to store some info to be used later in solving the problem
  - counting sorts (Ch. 7.1)
  - string searching algorithms
- ***prestructuring*** — preprocess the input to make accessing its elements easier
  - hashing
  - indexing schemes (e.g., B-trees)

Review: String searching by brute force

pattern: a string of *m* characters to search for

text: a (long) string of *n* characters to search in

Brute force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until either all characters are found to match (successful search) or a mismatch is detected

Step 3 While a mismatch is detected and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

Time complexity (worst-case): O(mn)

String searching by preprocessing

Several string searching algorithms are based on the input enhancement idea of preprocessing the pattern

- Knuth-Morris-Pratt (KMP) algorithm preprocesses pattern left to right to get useful information for later searching

  O(m+n) time in the worst case

- Boyer -Moore algorithm preprocesses pattern right to left and store information into two tables

  O(m+n) time in the worst case

- Horspool's algorithm simplifies the Boyer-Moore algorithm by using just one table

Horspool's Algorithm

A simplified version of Boyer-Moore algorithm:

- **preprocesses pattern to generate a shift table that determines how much to shift the pattern when a mismatch occurs**

- **always makes a shift based on the text's character *c* aligned with the last compared (mismatched) character in the pattern according to the shift table's entry for *c***

How far to shift?

Look at first (rightmost) character in text that was compared:

- **The character is not in the pattern**

```
.....c......................   (c not in pattern)
BAOBAB
```

6

- **The character is in the pattern (but not the rightmost)**

```
.....O......................   (O occurs once in pattern)
BAOBAB
```

3

```
.....A......................   (A occurs twice in pattern)
BAOBAB
```

1

لو متكرر تأخذ اقرب واحد

- **The rightmost characters do match**

```
.....B......................
BAOBAB
```

2

2

Shift table

- Shift sizes can be precomputed by the formula

$$t(c) = \begin{cases} \text{distance from } c\text{'s rightmost occurrence in pattern} \\ \text{among its first } m\text{-1 characters to its right end} \\ \\ \text{pattern's length } m\text{, otherwise} \end{cases}$$

by scanning pattern before search begins and stored in a table called *shift table.* After the shift, the right end of pattern is *t(c)* positions to the right of the last compared character in text.

- Shift table is indexed by text and pattern alphabet
Eg, for BAOBAB :

ALGORITHM *ShiftTable*$(P[0..m-1])$
  for $i \leftarrow 0$ **to** $size-1$ **do** $Table[i] \leftarrow m$
  for $j \leftarrow 0$ **to** $m-2$ **do** $Table[P[j]] \leftarrow m-1-j$
  return *Table*

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6

Horspool's algorithm

ALGORITHM *HorspoolMatching*($P[0..m-1]$, $T[0..n-1]$)
//Implements Horspool's algorithm for string matching
//Input: Pattern $P[0..m-1]$ and text $T[0..n-1]$
//Output: The index of the left end of the first matching substring
// or -1 if there are no matches

ShiftTable($P[0..m-1]$) //generate *Table* of shifts
$i \leftarrow m-1$ //position of the pattern's right end
while $i \le n-1$ **do**
  $k \leftarrow 0$ //number of matched characters
  while $k \le m-1$ **and** $P[m-1-k] = T[i-k]$ **do**
    $k \leftarrow k+1$
  if $k = m$
    return $i - m + 1$
  else $i \leftarrow i + Table[T[i]]$ Shift
return -1 x

Example of Horspool's algorithm

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

```
      6         2  6
BARD LOVED BANANAS
BAOBAB
   1  BAOBAB
         2  BAOBAB
               3      BAOBAB (unsuccessful search)
```

_ 1

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

The actual search in a particular text proceeds as follows:

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R             B A R B E R
        B A R B E R         B A R B E R
          B A R B E R             B A R B E R
```

BARBER

A B C D E F G H I J K L M N O P Q R S T V U W X Y

4 2 6 6 1 6 6 6 6 6 6 6 6 6 6 6 6 6 3 1 6 6 6 6 6 6 6

Hashing

- A very efficient method for implementing a *dictionary*, i.e., a set with the operations:
  1 – find
  2 – insert
  3 – delete

A dictionary returns data (called element) when a key is passed. Example: Given a phone number, return the caller's name.
Key: phone number; Element: caller's name

- Based on representation-change and space-for-time tradeoff ideas

- Important applications:
  1 – symbol tables
  2 – databases (*extendible hashing*)

Caller ID Problem

Given a phone number, return the caller's name

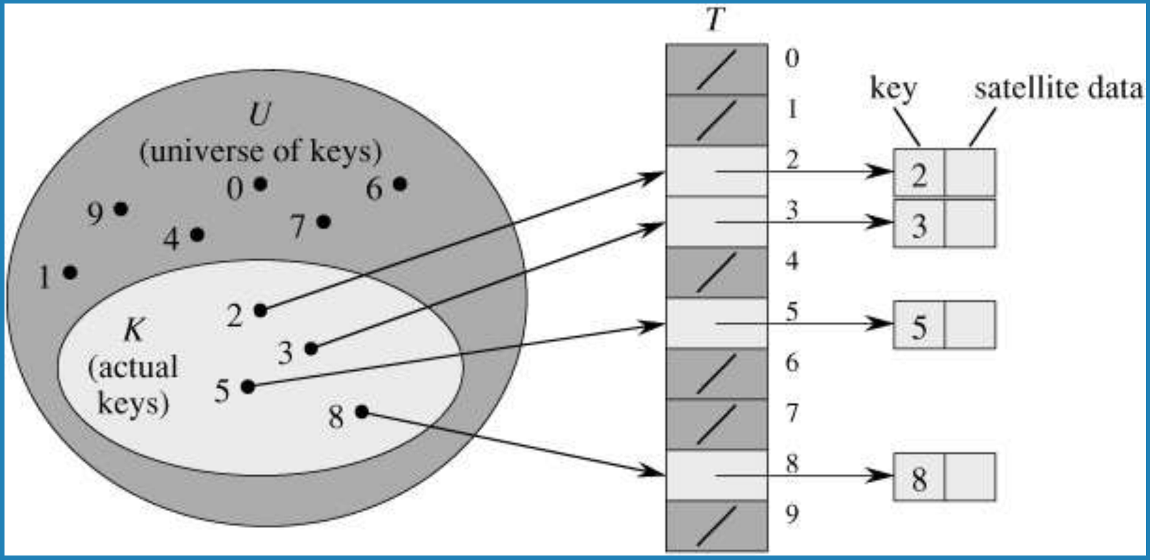
<u>Key</u>	<u>Element</u>
phone number	caller's name

- **Solutions**
  - Use a linked list
  - Use a balanced binary search tree
  - Direct Access Table

DAT

Direct Access Table

1 - Key becomes address of the record.
2 - Extremely Efficient
3 - Size of array becomes extremely large if a key value is large



DAT vs Hash Table

Direct Access Table

Address	Record
…	
5336663	"Sara"
…	
…	
5661116	" Ross"
…	

Hash Table

Hash Function = mod(key, 10)

Address		Record
0		
1		
2		
3	5336663	"Sara"
4		
5		
6	5661116	" Ross"
7		
8		
9		

Non-numeric keys can be converted to numeric values, for example by adding the ASCII values of all characters of the key.

Hash tables and hash functions

- **Hashing**: The idea is to map keys of size n into a *hash table* of size m.
- **Hash Function**: A predefined function for mapping

  h: $K \rightarrow$ location (cell) in the hash table
- **Generally, a hash function should:**
  1. be easy to compute سهلة
  2. distribute keys about evenly throughout the hash table

  موزعة كل جدول
- **Some hash functions**
  1. mod
  2. Truncation: e.g. keep 3 right most digits
  3. Folding: e.g. 123|456|789: add them and take mod.
  4. Squaring: e.g. square and truncate
  5. Radix conversion: e.g. 1234 treat it to be base 11, truncate if necessary.

Collisions

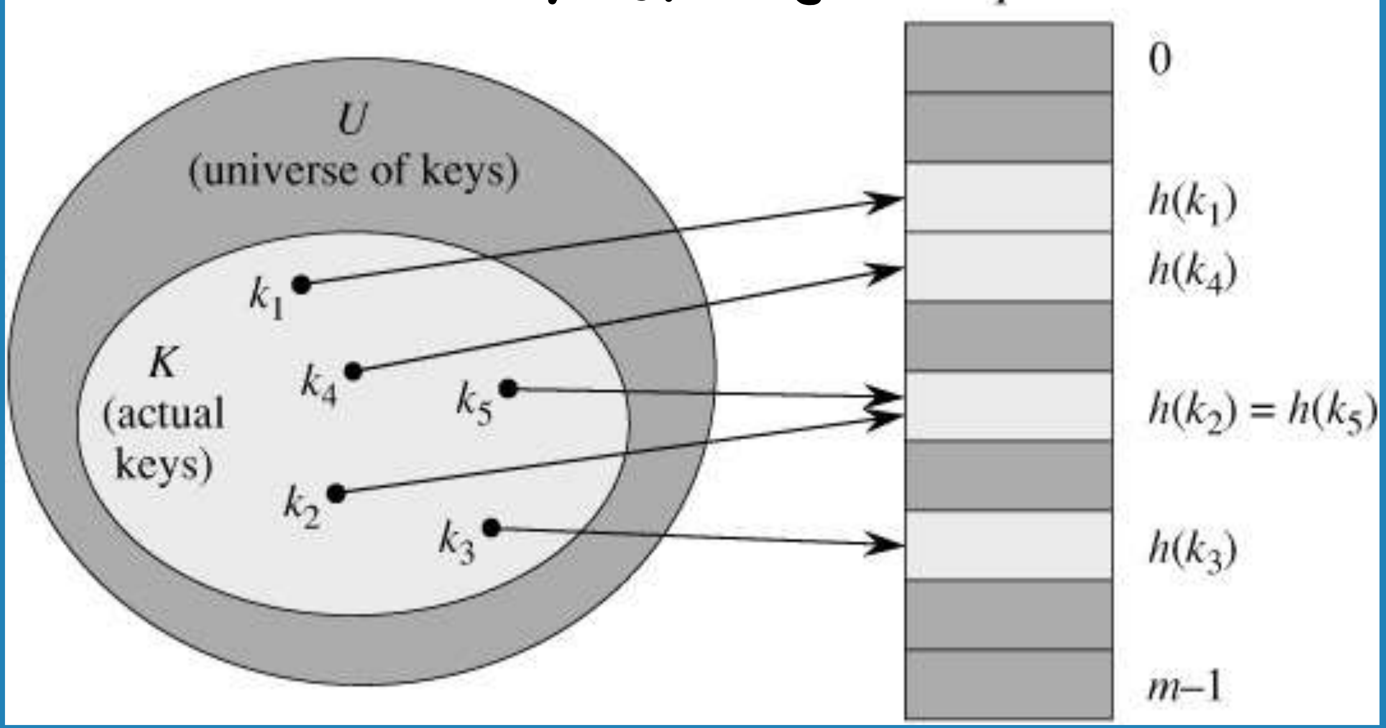
تعارض

- If $h(K_1) = h(K_2)$, there is a *collision*.

إذا صار عندي 2 داتا بتتخزن بنفس المكان بصير

- Good hash functions result in fewer collisions, but some collisions should be expected

birthday paradox



Handling Collisions

Two principal hashing schemes handle collisions differently:

1. *Open hashing*
   - each cell is a header of linked list of all keys hashed to it

البحث والإضافة والحذف O(1)

2. *Closed hashing*
   - one key per cell
   - in case of collision, finds another cell by
     - *linear probing:* use next free bucket
     - *Quadratic Probing:* use next free bucket distant by 1, 4, 9, 16, … positions
     - *double hashing:* use second hash function to compute increment

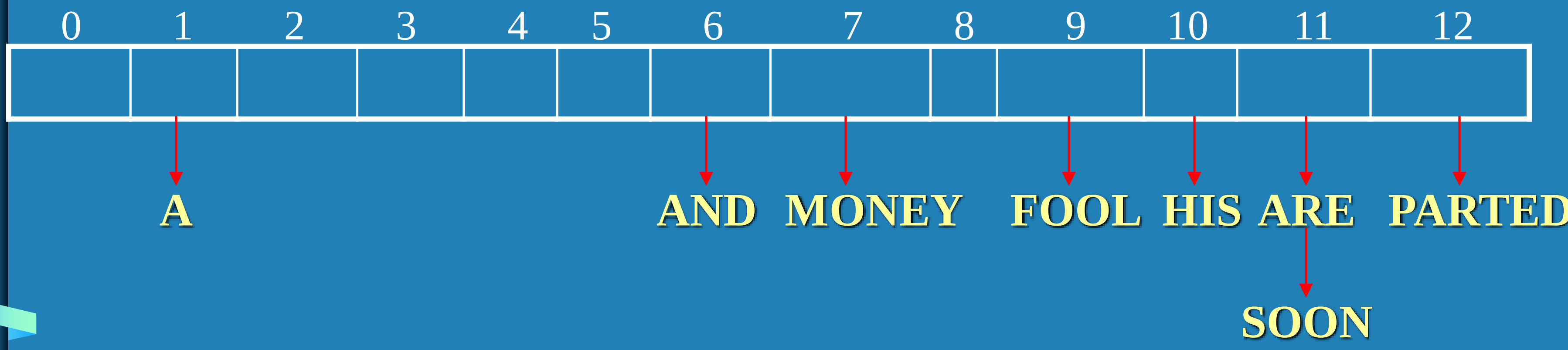
Open hashing (Separate chaining)

Keys are stored in linked lists <u>outside</u> a hash table whose elements serve as the lists' headers.

Example: A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

$h(K)$ = sum of K's letters' positions in the alphabet MOD 13

Key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
$h(K)$	1	9	6	10	7	11	11	12



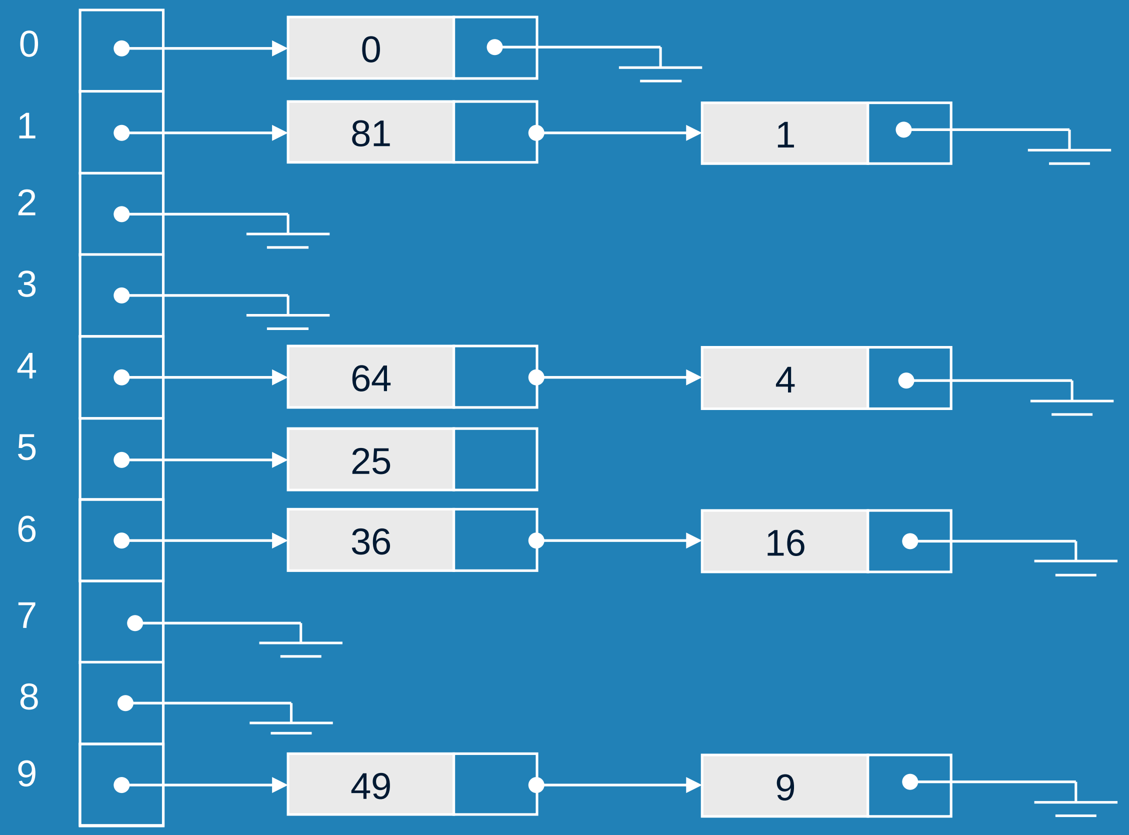
Search for KID

Open hashing (Separate chaining)

Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

hash(key) = key % 10.

Ex: ?



Open hashing (cont.)

- **If hash function distributes keys uniformly, average length of linked list will be $\alpha = n/m$. This ratio is called *load factor*.**

ألفا

- **For ideal hash functions, the average numbers of probes in successful, *S*, and unsuccessful searches, *U*:**

  $S \approx 1+\alpha/2$, $U = \alpha$ (CLRS, Ch. 11)

- **Load α is typically kept small (ideally, about 1)**

الـKey أكبر من المساحة لكن كفاءة سيئة

- **Open hashing still works if $n > m$**

Disadvantage: Requires the implementation of a second data structure, a linked list using pointers.

Closed hashing (Open addressing)

Keys are stored inside a hash table.

Key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
$h(K)$	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

- Try slot $hash(x) \% S$ — جرب 12
- If full then try $(hash(x) + 1) \% S$ — إذا فل جرب إلى بعده
- If full, then try $(hash(x) + 2) \% S$
- If full, then try $(hash(x) + 3) \% S$
- ….

- If last slot is full then the process starts again from slot 1 (wrap around)

وصلت آخر مكان ارجع لبداية ودور

مشكلة إذا فلة خلاص مايقدر يضيف

Closed hashing (Open addressing)

Insert items with keys:
89, 18, 49, 58, 9
into an empty hash table using linear probing

Ex:

```
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

Closed hashing (cont.)

- Does not work if $n > m$
- Avoids pointers
- Deletions are *not* straightforward الحذف مو مباشر
- Number of probes to find/insert/delete a key depends on load factor $\alpha = n/m$ (hash table density) and collision resolution strategy. For linear probing:

  $S = (½)(1 + 1/(1-\alpha))$ and $U = (½)(1 + 1/(1-\alpha)^2)$
- As the table gets filled (α approaches 1), number of probes in linear probing increases dramatically:

α	S $\frac{1}{2}(1+\frac{1}{1-\alpha})$	U $\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5