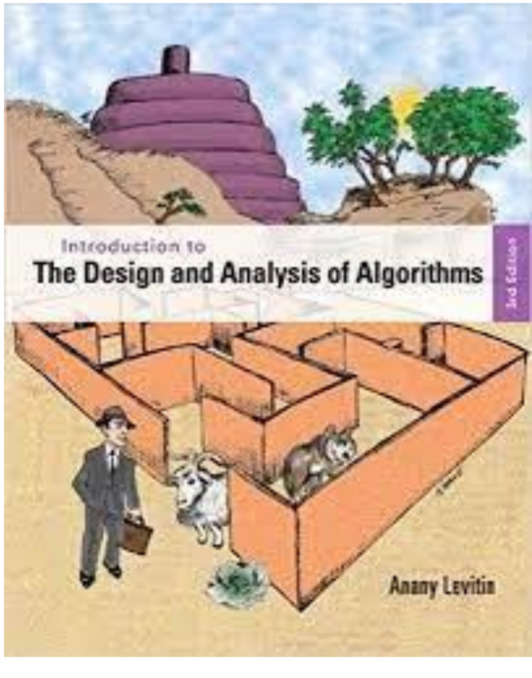
7-Space and Time Trade-Offs

A. Levitin “Introduction to the Design & Analysis of Algorithms,” 3rd ed., Ch. 1 ©2012 Pearson Education, Inc. Upper Saddle River, NJ.



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
  - 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

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

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

- 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 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
```

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

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

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

- The rightmost characters do match

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

How far to shift?

Four possibilities may occur:

Case 1 If there are no c's in the pattern—e.g., c is letter S in our example—we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text's character c that is known not to be in the pattern):

```
s0  ...        S              ... s_{n-1}
               ≠
     B A R B E R
                 B A R B E R
```

Case 2 If there are occurrences of character c in the pattern but it is not the last one there—e.g., c is letter B in our example—the shift should align the rightmost occurrence of c in the pattern with the c in the text:

```
s0  ...        B              ... s_{n-1}
               ≠
     B A R B E R
           B A R B E R
```

Case 3 If c happens to be the last character in the pattern but there are no c's among its other $m - 1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 1 and the pattern should be shifted by the entire pattern's length m:

```
s0  ...        M E R          ... s_{n-1}
               ≠ ‖ ‖
     L E A D E R
                 L E A D E R
```

Case 4 Finally, if c happens to be the last character in the pattern and there are other c's among its first $m - 1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 2 and the rightmost occurrence of c among the first $m - 1$ characters in the pattern should be aligned with the text's c:

```
s0  ...        A R            ... s_{n-1}
               ≠ ‖
 R E O R D E R
       R E O R D E R
```

Shift table

Shift sizes can be precomputed by the formula

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases} \qquad \textbf{(7.1)}$$

by scanning pattern before search begins and stored in a table called *shift table*

ALGORITHM *ShiftTable*($P[0..m-1]$)

//Fills the shift table used by Horspool's and Boyer-Moore algorithms
//Input: Pattern $P[0..m-1]$ and an alphabet of possible characters
//Output: *Table*$[0..size-1]$ indexed by the alphabet's characters and
//   filled with shift sizes computed by formula (7.1)
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*

Shift table

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

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

Example of Horspool's alg. application

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

```
BARD LOVED BANANAS
BAOBAB
      BAOBAB
        BAOBAB
              BAOBAB
```
(unsuccessful search)

Pseudocode of 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 ← m − 1                    //position of the pattern’s right end
    while i ≤ n − 1 do
        k ← 0                    //number of matched characters
        while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
            k ← k + 1
        if k = m
            return i − m + 1
        else  i ← i + Table[T[i]]
    return −1
```

Example

EXAMPLE As an example of a complete application of Horspool's algorithm, consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

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        ■
```

Boyer-Moore algorithm

Based on same two ideas:

- comparing pattern characters to text from right to left
- precomputing shift sizes in two tables
  - *bad-symbol table* indicates how much to shift based on text's character causing a mismatch
  - *good-suffix table* indicates how much to shift based on matched part (suffix) of the pattern

Bad-symbol shift in Boyer-Moore algorithm

- If the rightmost character of the pattern doesn't match, BM algorithm acts as Horspool's
- If the rightmost character of the pattern does match, BM compares preceding characters right to left until either all pattern's characters match or a mismatch on text's character *c* is encountered after $k > 0$ matches

$$
\begin{array}{cccccccccc}
s_0 & \dots & & c & s_{i-k+1} & \dots & s_i & \dots & s_{n-1} & \text{text} \\
 & & & \not\parallel & \parallel & & \parallel & & & \\
 & p_0 & \dots & p_{m-k-1} & p_{m-k} & \dots & p_{m-1} & & & \text{pattern}
\end{array}
$$

bad-symbol shift $d_1 = \max\{t_1(c) - k, 1\}$

Bad-symbol shift in Boyer-Moore algorithm

bad-symbol shift $d_1 = \max\{t_1(c\,) - k, 1\}$

$t_1(c)$ is the entry in the precomputed table used by Horspool's algorithm and k is the number of matched characters

$$
\begin{array}{cccccccccl}
s_0 & \dots & & c & s_{i-k+1} & \dots & s_i & \dots & s_{n-1} & \text{text} \\
 & & & \not\parallel & \parallel & & \parallel & & & \\
 & p_0 & \dots & p_{m-k-1} & p_{m-k} & \dots & p_{m-1} & & & \text{pattern} \\
 & & & & p_0 & \dots & & p_{m-1} & &
\end{array}
$$

Good-suffix shift in Boyer-Moore algorithm

- Good-suffix shift d_2 is applied after $0 < k < m$ last characters were matched
- $d_2(k)$ = the distance between matched suffix of size k and its rightmost occurrence in the pattern that is not preceded by the same character as the suffix

Example: `CABABA` $d_2(1) = 4$

Good-suffix shift in Boyer-Moore algorithm

- If there is no such occurrence, match the longest part of the k-character suffix with corresponding prefix; if there are no such suffix-prefix matches, $d_2\,(k) = m$

  Example: `WOWWOW` $d_2(2) = 5$, $d_2(3) = 3$, $d_2(4) = 3$, $d_2(5) = 3$

Good-suffix shift in the Boyer-Moore alg. (cont.)

After matching successfully $0 < k < m$ characters, the algorithm shifts the pattern right by

$$d = \max \{d_1, d_2\}$$

where $d_1 = \max\{t_1(c) - k, 1\}$ is bad-symbol shift

$d_2(k)$ is good-suffix shift

Boyer-Moore Algorithm (cont.)

Step 1 Fill in the bad-symbol shift table

Step 2 Fill in the good-suffix shift table

Step 3 Align the pattern against the beginning of the text

Step 4 Repeat until a matching substring is found or text ends:

Compare the corresponding characters right to left.

If no characters match, retrieve entry $t_1(c)$ from the bad-symbol table for the text's character c causing the mismatch and shift the pattern to the right by $t_1(c)$.
If $0 < k < m$ characters are matched, retrieve entry $t_1(c)$ from the bad-symbol table for the text's character c causing the mismatch and entry $d_2(k)$ from the good-suffix table and shift the pattern to the right by

$$d = \max \{d_1, d_2\}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

Example of Boyer-Moore alg. application

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

```
B E S S _ K N E W _ A B O U T _ B A O B A B S
B A O B A B
```

$d_1 = t_1(\mathrm{K}) = 6$ B A O B A B

$d_1 = t_1(\_)-2 = 4$

$\underline{d_2(2) = 5}$

B A O B A B

$\underline{d_1 = t_1(\_)-1 = 5}$

$d_2(1) = 2$

B A O B A B (success)

k	pattern	d_2
1	BAO**B**AB	2
2	BAO**BAB**	5
3	BAO**BAB**	5
4	BA**OBAB**	5
5	B**AOBAB**	5

Boyer-Moore example from their paper

Find pattern `AT_THAT` in

`WHICH_FINALLY_HALTS.__AT_THAT`

Hashing

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

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

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

Hash tables and hash functions

The idea of *hashing* is to map keys of a given file of size n into a table of size m, called the *hash table*, by using a predefined function, called the *hash function*,

$$h: K \rightarrow \text{ location (cell) in the hash table}$$

Example: student records, key = SSN. Hash function:
$h(K) = K \bmod m$ where m is some integer (typically, prime)
If $m = 1000$, where is record with SSN= 314159265 stored?

Generally, a hash function should:

- be easy to compute
- distribute keys about evenly throughout the hash table

Collisions

- If $h(K1) = h(K2)$, there is a collision
- Good hash functions result in fewer collisions, but some collisions should be expected (birthday paradox)
- Two principal hashing schemes handle collisions differently:
  - Open hashing
    - each cell is a header of linked list of all keys hashed to it
  - Closed hashing
    - one key per cell
    - in case of collision, finds another cell by
      - linear probing: use next free bucket
      - double hashing: use second hash function to compute increment

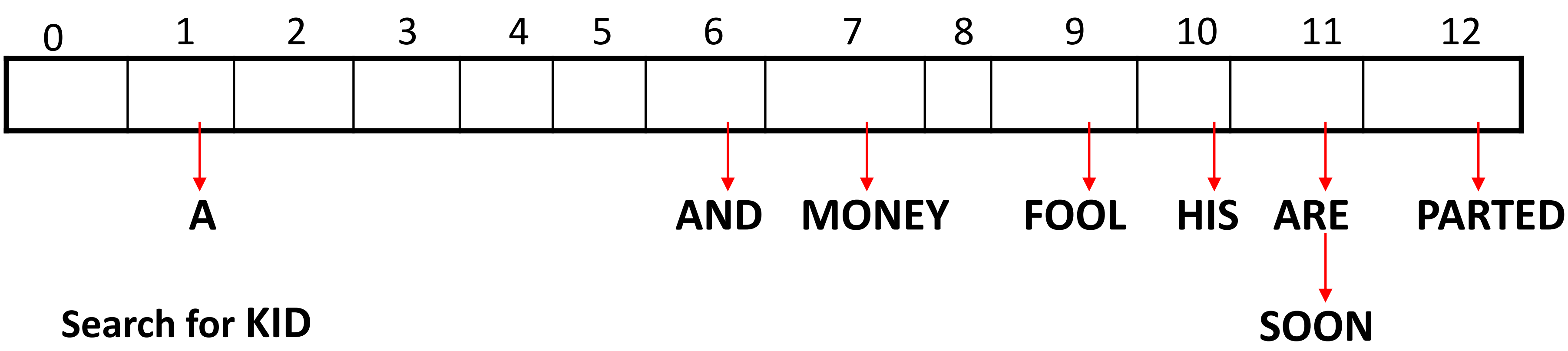
Open hashing (Separate chaining)

Keys are stored in linked lists outside 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 (cont.)

- If the hash function distributes *n* keys among *m* cells of the hash table about evenly, each list will be about *n/m* keys long. This ratio is called load factor.

- Average number of probes in successful, S, and unsuccessful searches, U:

$$S \approx 1 + \alpha/2, \quad U = \alpha$$

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

- Open hashing still works if n > m

Closed hashing (Open addressing)

- Keys are stored inside a hash table.

keys	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
hash addresses	1	9	6	10	7	11	11	12

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

Closed hashing (cont.)

- Does not work if n > m
- Avoids pointers
- Deletions are not straightforward
- 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	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5