# Chapter 7: 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 methods for sorting
  - string searching algorithms

- ***prestructuring*** — preprocess the input to make accessing its elements easier
  - hashing
  - indexing schemes (e.g., B-trees)

# 7.1 Sorting by Counting

- **Comparison-counting Sort**
  - **for each element of a list to be sorted, count the total number of elements smaller than this element and record the results in a table**
- **Example of sorting by comparison counting**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Array *A*[0..5] | | 62 | 31 | 84 | 96 | 19 | 47 |
| Initially | *Count* [] | 0 | 0 | 0 | 0 | 0 | 0 |
| After pass *i* = 0 | *Count* [] | 3 | 0 | 1 | 1 | 0 | 0 |
| After pass *i* = 1 | *Count* [] | | 1 | 2 | 2 | 0 | 1 |
| After pass *i* = 2 | *Count* [] | | | 4 | 3 | 0 | 1 |
| After pass *i* = 3 | *Count* [] | | | | 5 | 0 | 1 |
| After pass *i* = 4 | *Count* [] | | | | | 0 | 2 |
| Final state | *Count* [] | 3 | 1 | 4 | 5 | 0 | 2 |
| Array *S*[0..5] | | 19 | 31 | 47 | 62 | 84 | 96 |

# Seudocode of Comparison-counting Sort

**ALGORITHM** *ComparisonCountingSort*($A[0..n-1]$)

```
//Sorts an array by comparison counting
//Input: An array A[0..n − 1] of orderable elements
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for i ← 0 to n − 1 do Count[i] ← 0
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] < A[j]
            Count[j] ← Count[j] + 1
        else Count[i] ← Count[i] + 1
for i ← 0 to n − 1 do S[Count[i]] ← A[i]
return S
```

- **time efficiency $\Theta(n^2)$: is the same as the selection sort**

# Sorting by distribution counting

- **EXAMPLE:**
  - **Consider sorting the array: 13, 11, 12, 13, 12, 12**
  - **Compute frequencies and distribution:**

    **Distribution value indicates position of last occurrence of the array value in the sorted array.**

| Array values | 11 | 12 | 13 |
|---|---|---|---|
| Frequencies | 1 | 3 | 2 |
| Distribution values | 1 | 4 | 6 |

  - **Process the array from right to left**

    **put each array value in the position indicated by distribution value and reduce the distribution value by 1**

| | *D*[0..2] | | | *S*[0..5] | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| *A* [5] = 12 | 1 | **4** | 6 | | | | 12 | | |
| *A* [4] = 12 | 1 | **3** | 6 | | | 12 | | | |
| *A* [3] = 13 | 1 | 2 | **6** | | | | | | 13 |
| *A* [2] = 12 | 1 | **2** | 5 | | 12 | | | | |
| *A* [1] = 11 | **1** | 1 | 5 | 11 | | | | | |
| *A* [0] = 13 | 0 | 1 | **5** | | | | | 13 | |

# Pseudocode of distribution counting

**ALGORITHM** *DistributionCountingSort*$(A[0..n-1], l, u)$

```
//Sorts an array of integers from a limited range by distribution counting
//Input: An array A[0..n − 1] of integers between l and u (l ≤ u)
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for j ← 0 to u − l do D[j] ← 0                          //initialize frequencies
for i ← 0 to n − 1 do D[A[i] − l] ← D[A[i] − l] + 1 //compute frequencies
for j ← 1 to u − l do D[j] ← D[j − 1] + D[j]           //reuse for distribution
for i ← n − 1 downto 0 do
    j ← A[i] − l
    S[D[j] − 1] ← A[i]
    D[j] ← D[j] − 1
return S
```

- **Time efficiency: $\Theta(n)$**

# 7.2 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**

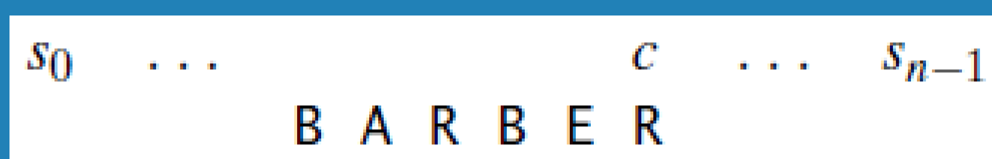
- **Horspool's algorithm simplifies the Boyer-Moore algorithm by using just one table**

# 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`

# Shift table

- Shift sizes can be precomputed by the formula

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

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

- 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 | 6 |

BARD LOVED BANANAS

BAOBAB

BAOBAB

BAOBAB

BAOBAB (unsuccessful search)

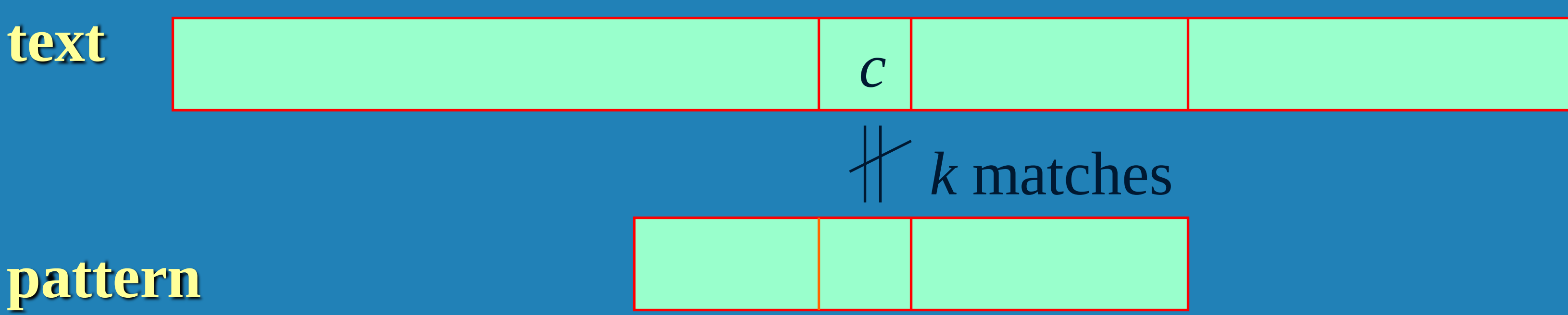
# Boyer-Moore algorithm

**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




- bad-symbol shift $d_1 = \max\{t_1(c) - k, 1\}$, where $t_1(c)$ is pre-computed by Horspool's algorithm

# 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` $d_2(1) = 2$

- 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` $d_2(2) = 3$, $d_2(3) = 3$, $d_2(4) = 5$, $d_2(5) = 5$

# 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

$d_1 = t_1(\text{K}) = 6$

B A O B A B

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

$d_2(2) = 5$

B A O B A B

$d_1 = t_1(\_)-1 = 5$

$d_2(1) = 2$

B A O B A B (success)

| *k* | pattern | $d_2$ |
|---|---|---|
| 1 | BAOBAB | 2 |
| 2 | BAOBAB | 5 |
| 3 | BAOBAB | 5 |
| 4 | BAOBAB | 5 |
| 5 | BAOBAB | 5 |

# Boyer-Moore example from their paper

**Find pattern `AT_THAT` in**

**`WHICH_FINALLY_HALTS. _ _ AT_THAT`**

# 7.3 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 $h(K_1) = h(K_2)$, 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

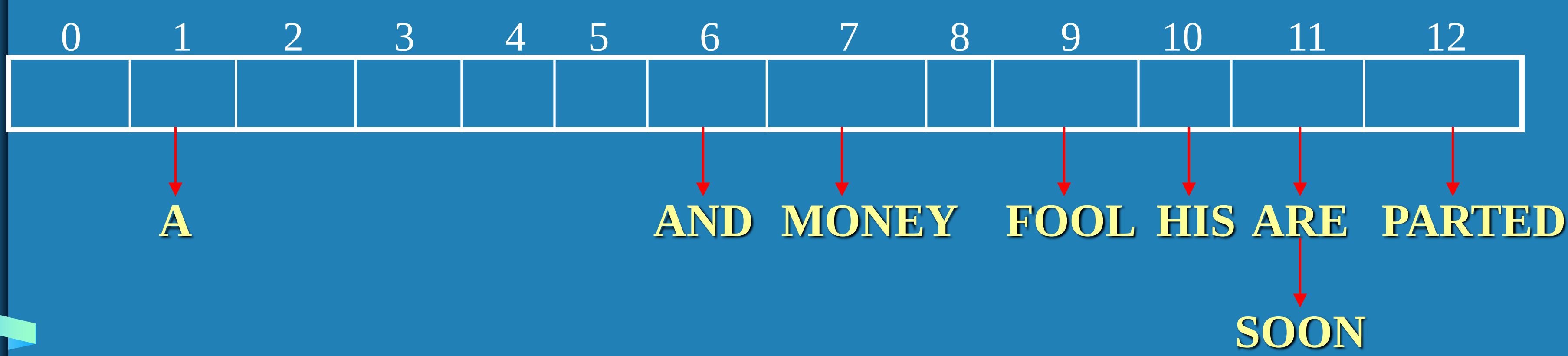
# Open hashing (Separate chaining)

**Keys are stored in linked lists <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 (cont.)

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

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

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

- Load $\alpha$ 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.**

| 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 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 ($\alpha$ approaches 1), number of probes in linear probing increases dramatically:

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

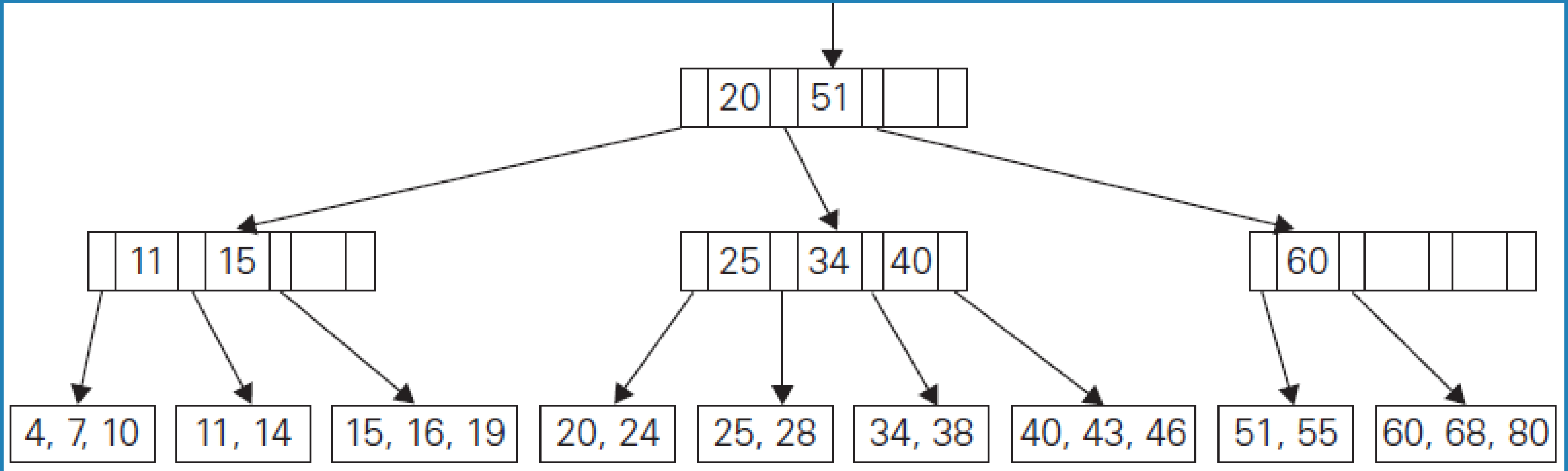
# 7.4 B-Trees

- All data records (or record keys) are stored at the leaves, in increasing order of the keys
- The parental nodes are used for indexing
  - Keys are interposed with pointers to children.
  - Key left to a pointer ≤ all keys in child pointed by the pointer
    < key right to the pointer
- In addition, a B-tree of order m ≥ 2 must satisfy the following structural properties:
  - The root is either a leaf or has between 2 and *m children*.
  - Each node, except for the root and the leaves, has between *m/2 and m* children
  - The tree is balanced, i.e., all its leaves are at the same level.

# B-Trees (Cont.)

- Example of a B-tree of order 4




- Search operation in B-tree
- B-tree often used for indexing large data file
  - Nodes represent disk pages
  - Minimizing the node accesses (minimizing the height) will minimizes disk accesses.

# B-Trees (Cont.)

- For any B-tree of order m with n nodes and height h>0, we have the following inequality

$$n \geq 1 + \sum_{i=1}^{h-1} 2\lceil m/2 \rceil^{i-1}(\lceil m/2 \rceil - 1) + 2\lceil m/2 \rceil^{h-1}.$$

- This gives an upper bound of h

$$h \leq \lfloor \log_{\lceil m/2 \rceil} \frac{n+1}{4} \rfloor + 1.$$

- Example: for a file of 100 million records, we have

| | | | |
|---|---|---|---|
| order $m$ | 50 | 100 | 250 |
| $h$'s upper bound | 6 | 5 | 4 |