CS 483 - Data Structures and Algorithm Analysis
Lecture VII: Chapter 7

R. Paul Wiegand

George Mason University, Department of Computer Science

March 29, 2006
Space vs. Time Tradeoff Introduction

**input enhancement** — Preprocess the problem’s input and store additional information to accelerate problem solving
- Counting methods for sorting
- Improvements to string matching algorithm

**prestructuring** — Use extra space to facilitate faster and/or flexible access to data
- Hashing
- Indexing with B-trees
- Sometimes we gain time efficiency at the expense of space (or vice-versa)
- Sometimes we gain time efficiency *while* gaining space efficiency (e.g., adjacency list representation & graph traversal algorithms)
Comparison Count Sort

**ComparisonCountingSort** \(A[0 \ldots n-1]\)

\[
\text{for } i \leftarrow 0 \text{ to } n-1 \text{ do } \text{Count}[i] \leftarrow 0
\]

\[
\text{for } i \leftarrow 0 \text{ to } n-2 \text{ do }
\]

\[
\quad \text{for } j \leftarrow i+1 \text{ to } n-1 \text{ do }
\]

\[
\quad \quad \text{if } A[i] < A[j] \text{ then }
\]

\[
\quad \quad \quad \text{Count}[j] \leftarrow \text{Count}[j] + 1
\]

\[
\quad \quad \text{else } \text{Count}[j] \leftarrow \text{Count}[j] + 1
\]

\[
\text{for } i \leftarrow 0 \text{ to } n-1 \text{ do } S[\text{Count}[i]] \leftarrow A[i]
\]

return \(S\)

\[
C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \frac{n(n-1)}{2} \in \Theta(n^2)
\]

For each element to be sorted, count the total number of elements smaller than this element.
**Comparison Counting Sort**

$$\text{ComparisonCountingSort}(A[0 \ldots n-1])$$

```plaintext
for \( j \leftarrow 0 \) to \( u-\ell \) do \( D[j] \leftarrow 0 \)
for \( i \leftarrow 0 \) to \( n-1 \) do \( D[A[i]-\ell] \leftarrow D[A[i]-\ell] + 1 \)
for \( j \leftarrow 1 \) to \( u-\ell \) do \( D[j] \leftarrow D[j-1] + D[j] \)
for \( i \leftarrow n-1 \) downto 0 do
    \( j \leftarrow A[i]-\ell \)
    \( S[D[j]-1] \leftarrow A[i] \)
    \( D[j] \leftarrow D[j]-1 \)
```

**Sometimes the input is constrained**
- Fixed array of values
- Each in \([\ell, u]\)

**Sometimes we want additional information information**

<table>
<thead>
<tr>
<th>( A[i] )</th>
<th>( D[0 \ldots 2] )</th>
<th>( S[0 \ldots 5] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1 4 6</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>1 3 6</td>
<td>12 12</td>
</tr>
<tr>
<td>3</td>
<td>1 2 6</td>
<td>12 12</td>
</tr>
<tr>
<td>2</td>
<td>1 2 5</td>
<td>12 12 13</td>
</tr>
<tr>
<td>1</td>
<td>1 2 5</td>
<td>12 12</td>
</tr>
<tr>
<td>0</td>
<td>0 1 5</td>
<td>11 12 12</td>
</tr>
</tbody>
</table>

---

R. Paul Wiegand  
George Mason University, Department of Computer Science  
CS483 Lecture II
String Matching Basics

- We have a *pattern* string and a *text* string
- We want to find the position of the first occurrence of the pattern in the text
- Recall brute force:
  - Align the pattern at the start of the text
  - Compare each character of the pattern to each of the text
  - If there’s a mismatch, shift the pattern one to the right and repeat
  - If the pattern matches, you are done
  - If the end of the pattern is reached, shift the pattern one to the right and repeat
  - $\Theta(nm)$ in the worst case
- But why shift *only one* each time?

**Example:**

<table>
<thead>
<tr>
<th>text</th>
<th>“FOUR SCORE ...”</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern</td>
<td>“FATHER”</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F</th>
<th>O</th>
<th>U</th>
<th>R</th>
<th>S</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>A</td>
<td>T</td>
<td>H</td>
<td>E</td>
<td>R</td>
</tr>
<tr>
<td>F</td>
<td>A</td>
<td>T</td>
<td>H</td>
<td>E</td>
<td>R</td>
</tr>
<tr>
<td>F</td>
<td>A</td>
<td>T</td>
<td>H</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Horspool’s Algorithm

- Idea: When we shift, make as large a shift as possible
- Match pattern from right to left
- Consider character $c$ of the text that was aligned against the last character of the pattern

$$t(c) = \begin{cases} m, & \text{if } c \text{ is not in the first } m-1 \text{ characters} \\ \text{dist from rightmost } c \text{ in first } m-1 \text{ characters}, & \text{otherwise} \end{cases}$$

- Still $\Theta(n)$ in Avg case, $\Theta(nm)$ in worst case
- But on average, must faster than brute force

```
SHIFT_TABLE(P[0...m-1])
for j ← 0 to m-2 do T[P[j]] ← m-1-j
return T
```
Horspool’s Algorithm

- Idea: When we shift, make as large a shift as possible
- Match pattern from right to left
- Consider character $c$ of the text that was aligned against the last character of the pattern

$$t(c) = \begin{cases} m, & \text{if } c \text{ is not in the first } m - 1 \text{ characters} \\ \text{dist from rightmost } c \text{ in first } m - 1 \text{ characters}, & \text{otherwise} \end{cases}$$

- Still $\Theta(n)$ in Avg case, $\Theta(nm)$ in worst case
- But on average, must faster than brute force

May repeatedly over-write shift value for a given character

```
SHIFT_TABLE(P[0...m-1])

for j ← 0 to m - 2 do  T[P[j]] ← m - 1 - j
return T
```
**Horspool’s Algorithm (2)**

1.) No $c$ in the pattern, shift entire pattern length

```
... O R E A N ...
D I D
```

2.) $c$ is in pattern but this is not the last one, shift to align rightmost $c$ in pattern

```
... A N D S E ...
E D I T
```

3.) $c$ is last character in pattern & no others in remaining $m - 1$, shift entire pattern length

```
... S E V E N ...
G I V E
```

4.) $c$ is last character in pattern & $\exists$ others in remaining $m - 1$, shift to align rightmost $c$ in pattern

```
... Y E A R S ...
R E A R
```
The Basics of Hashing

- Hashes are often useful for implementing *dictionaries* (basic operations: **Insert**, **Search**, & **Delete**)
- Construct a data type to store records by key value (**Hash Table**), generally an array $H[0 \ldots m - 1]$
- Use the key to access the table by computing its address with a predefined **Hash Function**, $h(k)$
  - If keys are nonnegative integers, a simple hash function is $h(k) = k \mod m$
  - For strings of a fixed length, we might use:
    $$h(k) = \left( \sum_{i=0}^{\ell-1} \text{ord}(c_i) \right) \mod m$$
  - Or, where $C$ is a larger constant than any $\text{ord}(c_i)$:
    $$h \leftarrow 0; \text{ for } i \leftarrow 0 \text{ to } \ell - 1 \text{ do } h \leftarrow (h \cdot C + \text{ord}(c_i))$$
Collisions

- Hash functions should try to:
  1. Distribute keys in the table as evenly as possible
  2. Be easy to compute

- When the hash functions computes the same value for different keys, a *collision* occurs
  - When \( m < n \) (\( n \) is the number of keys inserted into the table), this *will* occur
  - Even when \( m \geq n \) it is still possible (depending on the data and the hash function)

- Hash implementations need to have a *collision resolution method*, such as:
  - Open hashing (separate chaining)
  - Closed hashing (open addressing)
Open Hashing

- Each cell in the hash table is a linked list.
- Values are stored in list, collisions are handled by chaining values.
- If $n$ keys are distributed evenly, each list is about the same size: $\frac{n}{m}$
- Load factor — $\alpha = \frac{n}{m}$
- Average number of nodes visited during a successful search:
  $S \approx 1 + \frac{\alpha}{2}$
- Average number of nodes visited during an unsuccessful search:
  $U \approx \alpha$

**Example:**

$m = 5$

$h(k) = (suitvalue + cardvalue) \mod m$

$\{\spadesuit, \diamondsuit, \heartsuit, \clubsuit\} = \{42, 28, 14, 0\}$

$\{K, Q, J, A\} = \{13, 12, 11, 1\}$

Data

<table>
<thead>
<tr>
<th>$h(k)$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>A♦</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5♠</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9♠</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7♠</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K♥</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example hash table:

- $h(5) = 0$
- $h(9) = 1$
- $h(7) = 4$
- $h(K) = 2$

Example hash function:

- $h(5) = 5 \mod 5 = 0$
- $h(9) = 9 \mod 5 = 4$
- $h(7) = 7 \mod 5 = 2$
- $h(K) = 13 \mod 5 = 3$

Average number of nodes visited during a successful search:

$S = 1 + \frac{\alpha}{2}$

Average number of nodes visited during an unsuccessful search:

$U = \alpha$
Open Hashing

- Each cell in the hash table is a linked list.
- Values are stored in list, collisions are handled by chaining values.
- If $n$ keys are distributed evenly, each list is about the same size: $\frac{n}{m}$.
- **load factor** — $\alpha = \frac{n}{m}$.
- Average number of nodes visited during a successful search: $S \approx 1 + \frac{\alpha}{2}$.
- Average number of nodes visited during an unsuccessful search: $U \approx \alpha$.

**Example:**

$m = 5$

$h(k) = (suitvalue + cardvalue) \mod m$

- $\{\spadesuit, \heartsuit, \clubsuit, \diamondsuit\} = \{42, 28, 14, 0\}$
- $\{K, Q, J, A\} = \{13, 12, 11, 1\}$

<table>
<thead>
<tr>
<th>Data</th>
<th>$h(k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A♦</td>
<td>4</td>
</tr>
<tr>
<td>5♠</td>
<td>0</td>
</tr>
<tr>
<td>9♠</td>
<td>1</td>
</tr>
<tr>
<td>7♠</td>
<td>4</td>
</tr>
<tr>
<td>K♥</td>
<td>2</td>
</tr>
</tbody>
</table>

When load factor is near 1 & keys are well distributed, access is $\Theta(1)$ on average.
Closed Hashing with Linear Probing

- All keys are stored in table
- On collisions, we shift right until we find an open position
- At the end, we wrap back to the start
- **DELETE** is problematic (mark & skip)
- Avg. # comparisons when successful:
  \[ S \approx \frac{1}{2} \left( 1 - \frac{1}{1-\alpha} \right) \]
- Avg. # comparisons when unsuccessful:
  \[ U \approx \frac{1}{2} \left( 1 - \frac{1}{(1-\alpha)^2} \right) \]
Clustering & Double Hashing

# comparisons

\[ \alpha \]

Unsuccessful

Successful

R. Paul Wiegand
George Mason University, Department of Computer Science
CS483 Lecture II
Clustering & Double Hashing

- The main problem is *clustering*
- A *cluster* is a sequence of consecutive filled positions in the table
- One possible solution: *double hash*
  - Use a second hash function to compute the probe interval
    - \( h_2(k) = m - 2 - k \mod (m - 2) \)
    - We need \( h_2(k) \) and \( m \) to be “relatively prime” (only common divisor is 1)
  - Choosing a prime \( m \) ensures this

---

R. Paul Wiegand  
George Mason University, Department of Computer Science  
CS483 Lecture II
The main problem is clustering

A cluster is a sequence of consecutive filled positions in the table

One possible solution: double hash

- Use a second hash function to compute the probe interval
- $h_2(k) = m - 2 - k \mod (m - 2)$
- We need $h_2(k)$ and $m$ to be “relatively prime” (only common divisor is 1)
- Choosing a prime $m$ ensures this

We can still have problems as $\alpha$ approaches 1

Only solution: rehash (scan table & relocate into a bigger table)
Often we need access to data stored on disk

There can be a large number of data records

And the records are typically *indexed* — indexes provide key values and information about the record’s location

In such cases, we typically are less interested in counting key comparisons and more interested in counting disk accesses

B-Trees extend the idea of 2-3 Trees to make such considerations easier
Data records stored in *leaves* in increasing order of the keys

Each parental node contains \( m - 1 \) (distinct) ordered keys

All keys in \( T_0 \) are smaller than \( K_1 \), all keys in \( T_1 \) are in \([K_1, K_2)\), etc.

Every B-Tree of order \( m > 2 \) must satisfy:

- Root is leaf or has between 2 and \( m \) children
- Internal nodes (\( \sim \) root \( \lor \) leaf) have b/w \( \lceil m/2 \rceil \) and \( m \) children
- The tree is (perfectly) balanced; all leaves at same level
Searching in a B-Tree

- Keys are ordered in the node, so we can use binary search to find the pointer to follow.
- But we don’t care about key comparisons, we care about disk access.
- We usually choose the order of a B-Tree so that the node size corresponds with disk pages.
- How many nodes do we have to consider? Height plus 1 ...
What is the height of a B-Tree?

Find: smallest # of keys a B-Tree of order $m$ and height $h$ can have:

- Root has at least one key
- Level 1 has at least two nodes with at least $\lceil m/2 \rceil - 1$ keys
- Level 2 has at least $2 \lceil m/2 \rceil$ nodes with at least $\lceil m/2 \rceil - 1$ keys
- For a B-Tree of order $m$ with $n$ nodes and height $h$:
  \[
  n \geq 1 + \sum_{i=1}^{h-1} 2 \lceil m/2 \rceil^{i-1} (\lceil m/2 \rceil - 1) + 2 \lceil m/2 \rceil^{h-1}
  \]
- Which reduces to:
  \[
  n \leq 4 \lceil m/2 \rceil^{h-1} - 1
  \]
- So height is:
  \[
  h \leq \lceil \log_{\lceil m/2 \rceil} \frac{n+1}{4} \rceil + 1
  \]
Analyzing Search

- What is the height of a B-Tree?
- Find: smallest # of keys a B-Tree of order $m$ and height $h$ can have:
  - Root has at least one key
  - Level 1 has at least two nodes with at least $\lceil m/2 \rceil - 1$ keys
  - Level 2 has at least $2 \lceil m/2 \rceil$ nodes with at least $\lceil m/2 \rceil - 1$ keys
  - For a B-Tree of order $m$ with $n$ nodes and height $h$:
    \[ n \geq 1 + \sum_{i=1}^{h-1} 2 \lceil m/2 \rceil^{i-1} (\lceil m/2 \rceil - 1) + 2 \lceil m/2 \rceil^{h-1} \]
    - Which reduces to:
      \[ n \leq 4 \lceil m/2 \rceil^{h-1} - 1 \]
    - So height is:
      \[ h \leq \lceil \log_{\lceil m/2 \rceil} \frac{n+1}{4} \rceil + 1 \]
  - Since $m$ is a constant (even if very large), this is $O(\log n)$
There are a variety of \texttt{INSERT} functions for B-Trees.

Here’s a simple one:

- Find the appropriate leaf & insert key
- If there are too many keys:
  - Split node in half
  - Promote smallest key of new node to parent
  - This may percolate up the tree

Analysis is difficult, but this is also $O(\log n)$.
Book Topics Skipped in Lecture

- In section 7.2:
  - Boyer-Moore Algorithm (pp. 255–259)
Assignments

- This week’s assignments:
  - Section 7.1: Problems 3 & 7
  - Section 7.2: Problems 2, 5, & 7
  - Section 7.3: Problems 1, 2, & 8
  - Section 7.4: Problems 3 & 4