CS 483 - Data Structures and Algorithm Analysis

Lecture VII: Chapter 7

R. Paul Wiegand

George Mason University, Department of Computer Science

March 29, 2006

Outline

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($A[0 \ldots n-1]$)

```
for $i \leftarrow 0$ to $n-1$ do $Count[i] \leftarrow 0$
for $i \leftarrow 0$ to $n-2$ do
  for $j \leftarrow i+1$ to $n-1$ do
    if $A[i] < A[j]$ then
      $Count[j] \leftarrow Count[j] + 1$
    else $Count[i] \leftarrow Count[i] + 1$
for $i \leftarrow 0$ to $n-1$ do $S[Count[i]] \leftarrow A[i]$
return $S$
```

$A =$	64	31	84	96	19	47

init	0	0	0	0	0	0
$i = 0$	3	0	1	1	0	0
$i = 1$	3	1	2	2	0	1
$i = 2$	3	1	4	3	0	1
$i = 3$	3	1	4	5	0	1
$i = 4$	3	1	4	5	0	2
$i = 5$	3	1	4	5	0	2

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.

Distribution Counting

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

```
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
    $j \longleftarrow A[i] - \ell$
    $S[D[j] - 1] \longleftarrow A[i]$
    $D[j] \longleftarrow D[j] - 1$
```

After accumulation...

$D[0 \ldots 2]$

$A[i=5] = 12$	1	**4**	6
$A[i=4] = 12$	1	**3**	6
$A[i=3] = 13$	1	2	**6**
$A[i=2] = 12$	1	**2**	5
$A[i=1] = 11$	**1**	2	5
$A[i=0] = 13$	0	1	**5**

$S[0 \ldots 5]$

			12		
		12	12		
		12	12		**13**
	12	12	12		13
11	12	12	12		13
11	12	12	12	**13**	13

- Sometimes the input is constrained
    - Fixed array of values
    - Each in $[\ell, u]$
- Sometimes we want additional information information

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:
text = “FOUR SCORE ...”
pattern = “FATHER”

F	O	U	R		S	C
F	A	T	H	E	R	
	F	A	T	H	E	R
		F	A	T	H	E
	. . .					

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

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

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

ShiftTable($P[0 \ldots m-1]$)

```
for $j \leftarrow 0$ to $m-2$ do $T[P[j]] \leftarrow m-1-j$
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

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

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

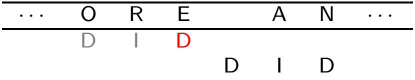
May repeatedly overwrite shift value for a given character

ShiftTable($P[0 \ldots m-1]$)

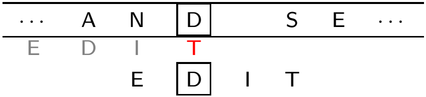
```
for $j \leftarrow 0$ to $m-2$ do $T[P[j]] \leftarrow m-1-j$
return $T$
```

Horspool's Algorithm (2)

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

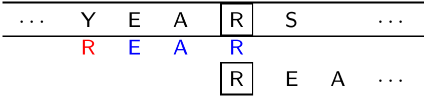


2.) c is in pattern but this is not the last one, shift to align right-most c in pattern



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

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



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: $h(k) = \left(\sum_{i=0}^{\ell-1} ord(c_i)\right) \mod m$
  - Or, where C is a larger constant than any $ord(c_i)$: $h \leftarrow 0$; `for` $i \leftarrow 0$ `to` $\ell - 1$ `do` $h \leftarrow (h \cdot C + ord(c_i))$

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:

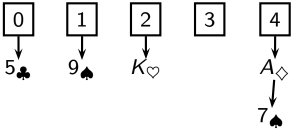
$m = 5$

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

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

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

Data	$A_{\diamondsuit}$	$5_{\clubsuit}$	$9_{\spadesuit}$	$7_{\spadesuit}$	$K_{\heartsuit}$
$h(k)$	4	0	1	4	2



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:

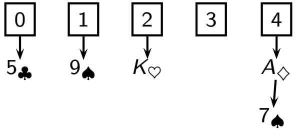
$m = 5$

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

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

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

Data	$A_\diamondsuit$	$5_\clubsuit$	$9_\spadesuit$	$7_\spadesuit$	$K_\heartsuit$
$h(k)$	4	0	1	4	2



When load factor is near 1 & keys are well distributed, access is $\Theta(1)$ on average

Closed Hashing with Linear Probing

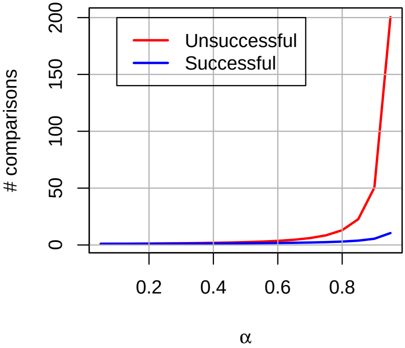
- All keys are stored in table
- On collisions, we shift right until we find an open position
- At the end, we wrap back to the start
- DELETE is problematic (mark & skip)
- Avg. # comparisons when successful: $S \approx \frac{1}{2}\left(1 - \frac{1}{1-\alpha}\right)$
- Avg. # comparisons when unsuccessful: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

Example:

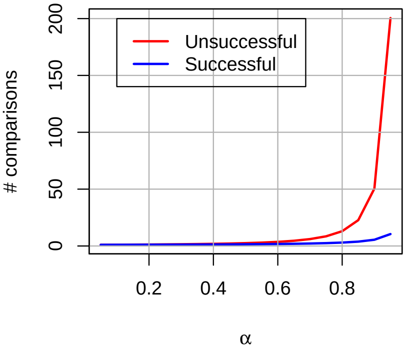
Data	$A_\diamondsuit$	$5_\clubsuit$	$9_\spadesuit$	$7_\spadesuit$	$K_\heartsuit$
$h(k)$	4	0	1	4	2

0	1	2	3	4
				$A_\diamondsuit$
$5_\clubsuit$				$A_\diamondsuit$
$5_\clubsuit$	$9_\spadesuit$			$A_\diamondsuit$
$5_\clubsuit$	$9_\spadesuit$	$7_\spadesuit$		$A_\diamondsuit$
$5_\clubsuit$	$9_\spadesuit$	$7_\spadesuit$	$K_\heartsuit$	$A_\diamondsuit$

Clustering & Double Hashing

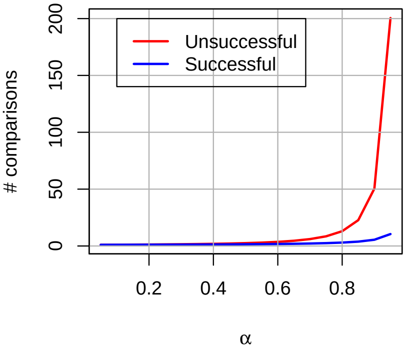


Clustering & Double Hashing



- 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

Clustering & Double Hashing



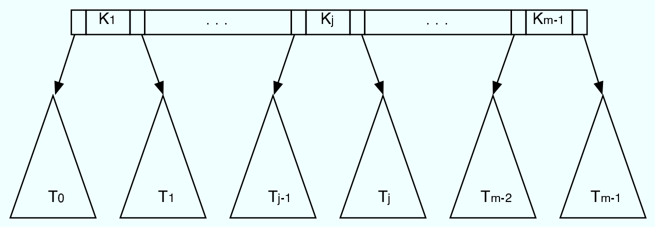
- The main problem is *clustering*
- A *cluster* is a sequence of consecutive filled positions in the table
- One possible solution: *double hash*
  - Use a second hash function to compute the probe interval
  - $h_2(k) = m - 2 - k \mod (m-2)$
  - We need $h_2(k)$ and m to be "relatively prime" (only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

- We can *still* have problems as α approaches 1
- Only solution: *rehash* (scan table & relocate into a bigger table)

Storing Data on Disk

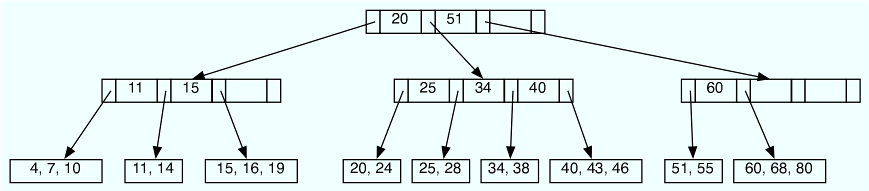
- Often we need access to data stored on disk
- There can be a large number of data records
- And the records are typically *indexed* — indexes provide key values and information about the record's location
- In such cases, we typically are less interested in counting key comparisons and more interested in counting disk accesses
- B-Trees extend the idea of 2-3 Trees to make such considerations easier

B-Trees



- 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 $\vee$ 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 `s.t.` the node size corresponds with disk pages
- How many nodes do we have to consider? Height plus 1 ...

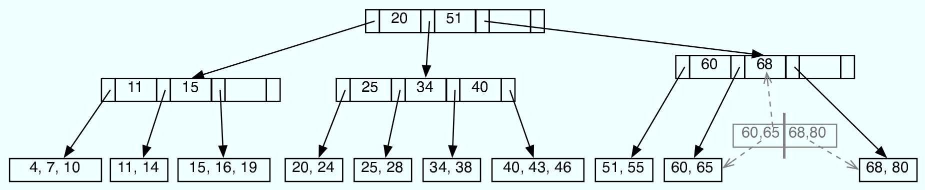
Analyzing Search

- 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: $h \leq \left\lfloor log_{\lceil m/2 \rceil} \frac{n+1}{4} \right\rfloor + 1$

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: $h \leq \left\lfloor log_{\lceil m/2 \rceil} \frac{n+1}{4} \right\rfloor + 1$
    - Since m is a constant (even if very large), this is $O(\log n)$

Inserting in a B-Tree



- There are a variety of 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