

Chapter 7: Space and Time Tradeoffs

The biggest difference between time and space is that you can't reuse time. (Merrick Furst)

Introduction	2
Distribution Counting	3
Sorting Integers by Counting Them	3
Distribution Counting Algorithm	4
Distribution Counting Example	5
Input Enhancement in String Matching	6
String Matching Speedup Idea	6
Horspool's Algorithm	7
Examples of Horspool's Algorithm	8
Shift Table for Horspool's Algorithm	9
Horspool's Algorithm	10
Comments on Horspool's Algorithm	11
Hashing	12
Introduction to Hashing	12
Hash Functions	13
Collision Resolution by Chaining	14
Collision Resolution by Probing	15
Comments on Hashing	16
B-Trees	17
Introduction to B-Trees	17
Example of B-Tree	18

Introduction

Often, we can solve a problem faster by using additional space.

- *Input enhancement* is based on preprocessing the instance to obtain additional information that can be used to solve the instance in less time.
- *Prestructuring* is based on using extra space to facilitate faster and/or more flexible access to the data.
- *Dynamic programming* (Chapter 8) is based on recording solutions to smaller instances so that each smaller instance is solved only once.

Distribution Counting

Sorting Integers by Counting Them

- The idea to count how many times each integer appears in an array.
- This information can be used to sort the array.
- For example, if we knew there were three 0s (and 0 is the minimum), then the sorted array will have 0s in positions 0, 1, and 2.
- This requires an additional array to store the counts.
- This only works well if the range of integers is $O(n)$, order n or less.

Distribution Counting Algorithm

```
algorithm DistributionCounting( $A[0..n-1], m$ )
// Sort by distribution counting
// Input: An array  $A$  of integers  $\geq 0$  and  $< m$ 
// Output: Sorted array  $S$ 
for  $j \leftarrow 0$  to  $m-1$  do  $D[j] \leftarrow 0$ 
for  $i \leftarrow 0$  to  $n-1$  do  $D[A[i]] \leftarrow D[A[i]] + 1$ 
for  $j \leftarrow 1$  to  $m-1$  do  $D[j] \leftarrow D[j-1] + D[j]$ 
for  $i \leftarrow n-1$  downto  $0$  do
     $D[A[i]] \leftarrow D[A[i]] - 1$ 
     $S[D[A[i]]] \leftarrow A[i]$ 
return  $S$ 
```

CS 3343 Analysis of Algorithms

Chapter 7: Slide – 4

Distribution Counting Example

- Suppose $A = 3, 0, 2, 3, 2, 2$.
- Second loop. $D = 1, 0, 3, 2$.
- Third loop. $D = 1, 1, 4, 6$.
- Fourth loop.
 - $i = 5$. $D = 1, 1, 3, 6$. $S = \cdot, \cdot, \cdot, 2, \cdot, \cdot$
 - $i = 4$. $D = 1, 1, 2, 6$. $S = \cdot, \cdot, 2, 2, \cdot, \cdot$
 - $i = 3$. $D = 1, 1, 2, 5$. $S = \cdot, \cdot, 2, 2, \cdot, 3$
 - $i = 2$. $D = 1, 1, 1, 5$. $S = \cdot, 2, 2, 2, \cdot, 3$
 - $i = 1$. $D = 0, 0, 1, 5$. $S = 0, 2, 2, 2, \cdot, 3$
 - $i = 0$. $D = 0, 0, 1, 4$. $S = 0, 2, 2, 2, 3, 3$

CS 3343 Analysis of Algorithms

Chapter 7: Slide – 5

Input Enhancement in String Matching

String Matching Speedup Idea

We can speed up string matching by matching from the right side of the string.

- Consider a search for “analysis” in some text.
- A possible match is from positions 0 to 7 of the text.
- If we compare “s” to position 7, and
- this letter is not “s” or any letter in “analysis”,
- then the next possible match for “analysis” is positions 8 to 15,
- so we next compare “s” to position 15.
- Other misses shift the next possible match different amounts.
- The shift amounts can be stored in a table.

CS 3343 Analysis of Algorithms

Chapter 7: Slide – 6

Horspool’s Algorithm

- Horspool’s algorithm performs a shift based on the text character c compared to the last character in the pattern.
- If c does not appear in the rest of the pattern, shift the length of the pattern.
- If c appears in the rest of the pattern, shift the number of characters between the last c in the rest of the pattern to the end of the pattern.

CS 3343 Analysis of Algorithms

Chapter 7: Slide – 7

Examples of Horspool's Algorithm

```

T ...          S          ...
P   B A R B E R
shift          B A R B E R
-----
T ...          B          ...
P   B A R B E R
shift          B A R B E R
-----
T ...          T E R          ...
P   L E A D E R
shift          L E A D E R
-----
T ...          T E R          ...
P   E R A S E R
shift          E R A S E R

```

CS 3343 Analysis of Algorithms

Chapter 7: Slide – 8

Shift Table for Horspool's Algorithm

```

algorithm HorspoolTable( $P[0..m-1]$ )
// Input: Pattern  $P$ 
// Output: Shift table for Horspool's Algorithm
for  $k \leftarrow 0$  to character set size – 1 do
     $S[k] \leftarrow m$ 
for  $j \leftarrow 0$  to  $m - 2$  do
     $S[P[j]] \leftarrow m - 1 - j$ 
return  $S$ 

```

CS 3343 Analysis of Algorithms

Chapter 7: Slide – 9

Horspool's Algorithm

```

algorithm Horspool( $P[0..m-1], T[0..n-1]$ )
// Implements Horspool's Algorithm
// Input: Pattern  $P$  and text  $T$ 
// Output: Index of match in text or –1
 $S \leftarrow$  HorspoolTable( $P$ )
 $i \leftarrow 0$  //  $i$  is index into  $T$ 
while  $i \leq n - m$  do
     $k \leftarrow m - 1$  //  $k$  is index into  $P$ 
    while  $k \geq 0$  and  $P[k] = T[i + k]$  do
         $k \leftarrow k - 1$ 
    if  $k < 0$  then return  $i$ 
    else  $i \leftarrow i + S[T[i + m - 1]]$ 
return –1

```

CS 3343 Analysis of Algorithms

Chapter 7: Slide – 10

Comments on Horspool's Algorithm

- Horspool's Algorithm uses $\Theta(m)$ additional space, the space for the shift table.
- Horspool's Algorithm can run anywhere from $\Omega(m + n/m)$ (best case) to $O(mn)$ (worst case).
- The $O(n)$ Boyer-Moore Algorithm is based on the all the characters that are compared rather than just the one compared to the last character of the pattern.

CS 3343 Analysis of Algorithms

Chapter 7: Slide – 11

Hashing

12

Introduction to Hashing

- Hashing is an approach to implement a *dictionary*, to insert, search, and delete elements of a set (records in a file).
- Typically, records have several fields, one of which is the *key*, which identifies the record.
- Hashing distributes the keys in *cells* of a *hash table* (an array).
- A *hash function* h computes the *hash address* from a key.
- A *collision* is when two different keys have the same hash address, i.e., $h(K_1) = h(K_2)$.
- *Chaining* and *probing* are two approaches for collision resolution.

CS 3343 Analysis of Algorithms

Chapter 7: Slide – 12

Hash Functions

- A hash function should distribute keys evenly/randomly in the hash table and should be easy to compute.
- [Cryptographic hash functions should also be hard to reverse, i.e., for an arbitrary a , hard to find a key such that $h(K) = a$.]
- A simple hash function is $h(K) = K \bmod m$. m is the hash table size. m should be prime.
- For a character string $T[0..l-1]$, where C is the size of the character set, use:
 - $a \leftarrow 0$
 - for** $i \leftarrow 0$ **to** $l-1$ **do**
 - $a \leftarrow (a * C + T[i]) \bmod m$

CS 3343 Analysis of Algorithms

Chapter 7: Slide – 13

Collision Resolution by Chaining

In chaining, each cell is a linked list that contains all the keys hashed to that cell.

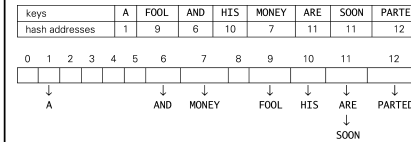


FIGURE 7.5 Example of a hash table construction with separate chaining

The *load factor* is $\alpha = n/m$ (number of keys/ number of cells). The average number of cell/list accesses is $\Theta(\alpha)$.

CS 3343 Analysis of Algorithms

Chapter 7: Slide – 14

Collision Resolution by Probing

In linear probing, a key is inserted in the first empty cell, starting with the hash address.

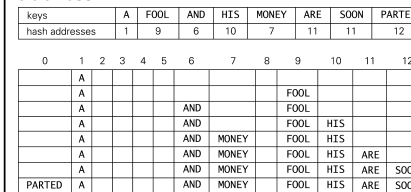


FIGURE 7.6 Example of a hash table construction with linear probing

On average, this is $\Theta(1/(1-\alpha))$ for successful search and $\Theta(1/(1-\alpha)^2)$ for unsuccessful search.

CS 3343 Analysis of Algorithms

Chapter 7: Slide – 15

Comments on Hashing

- The efficiency of hashing depends on well-distributed keys and low load factors.
- Chaining can tolerate load factors > 1 , but for probing, the load factor must be < 1 , and should be < 0.75 .
- If a hash table becomes full or nearly full, a common solution is *rehashing*, moving all keys to a larger hash table.
- *Double hashing* is a more efficient alternative to linear probing.

CS 3343 Analysis of Algorithms

Chapter 7: Slide – 16

Introduction to B-Trees

- B-trees are search trees where each node can have many keys and children.
- A B-tree of order $m \geq 3$ satisfies:
 - A node with k children has $k - 1$ keys.
 - Each nonroot node has $\geq \lceil m/2 - 1 \rceil$ keys.
 - The root is a leaf or has between 2 and m children (between 1 and $m - 1$ keys).
 - Internal nodes have between $\lceil m/2 \rceil$ and m children (between $\lceil m/2 \rceil - 1$ and $m - 1$ keys).
 - All leaves are at the same level.
- A B-tree of height h has $\geq 2 \lceil m/2 \rceil^h - 1$ keys.

Example of B-Tree

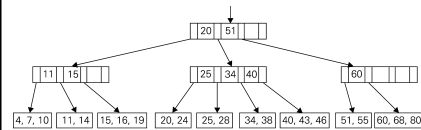


FIGURE 7.8 Example of a B-tree of order 4

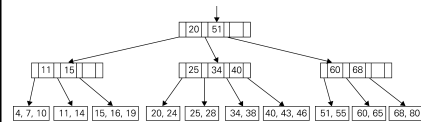


FIGURE 7.9 B-tree obtained after inserting 65 into the B-tree in Figure 7.8