CSCE 310J: Data Structures & Algorithms

Space-Time Tradeoffs

Dr. Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE310J

CSCE 310J: Data Structures & Algorithms

- **Giving credit where credit is due:**
  - **Most of the lecture notes are based on the slides from the Textbook's companion website**
    - **http://www.aw.com/cssuport/**
  - **Some examples and slides are based on lecture notes created by Dr. Ben Choi, Louisiana Technical University and Dr. Chuck Cusack, UNL**
  - **I have modified many of their slides and added new slides.**

Space-time tradeoffs

For many problems some extra space really pays off:

- **extra space in tables (breathing room?)**
  - **hashing**
  - **non comparison-based sorting**
- **input enhancement**
  - **indexing schemes (eg, B-trees)**
  - **auxiliary tables (shift tables for pattern matching)**
- **tables of information that do all the work**
  - **dynamic programming**

String matching

- ***pattern*: a string of *m* characters to search for**
- ***text*: a (long) string of *n* characters to search in**

- ***Brute force algorithm:***
  1. **Align pattern at beginning of text**
  2. **moving from left to right, compare each character of pattern to the corresponding character in text until**
     - **all characters are found to match (successful search); or**
     - **a mismatch is detected**
  3. **while pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat step 2.**

String searching - History

- **1970: Cook shows (using finite-state machines) that problem can be solved in time proportional to *n+m***
- **1976 Knuth and Pratt find algorithm based on Cook's idea; Morris independently discovers same algorithm in attempt to avoid "backing up" over text**
- **At about the same time Boyer and Moore find an algorithm that examines only a fraction of the text in most cases (by comparing characters in pattern and text from right to left, instead of left to right)**
- **1980 Another algorithm proposed by Rabin and Karp virtually always runs in time proportional to *n+m* and has the advantage of extending easily to two-dimensional pattern matching and being almost as simple as the brute-force method.**

Horspool's Algorithm

- **A simplified version of Boyer-Moore algorithm that retains key insights:**
  - **compare pattern characters to text from right to left**
  - **given a pattern, create a shift table that determines how much to shift the pattern when a mismatch occurs (*input enhancement*)**

How far to shift?

Look at first (rightmost) character in text that was compared. Three cases:

- The character is not in the pattern
  `.....c......................` (c not in pattern)
  `BAOBAB`
- The character is in the pattern (but not at rightmost position)
  `.....O......................` (O occurs once in pattern)
  `BAOBAB`

  `.....A......................` (A occurs twice in pattern)
  `BAOBAB`
- The rightmost characters produced a match
  `.....B......................`
  `BAOBAB`

- Shift Table: Stores number of characters to shift by depending on first character compared

Shift table

- Constructed by scanning pattern before search begins
- Indexed by text and pattern alphabet
- All entries are initialized to length of pattern. Eg, 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
6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6

- For c occurring in pattern, update table entry to distance of rightmost occurrence of c from end of pattern
- We can do this by processing pattern from L→R:

Example

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

`BARD LOVED BANANAS`
`BAOBAB`

Boyer-Moore algorithm

- Based on same two ideas:
  - compare pattern characters to text from right to left
  - given a pattern, create a shift table that determines how much to shift the pattern when a mismatch occurs (*input enhancement*)
- Uses additional shift table with same idea applied to the number of matched characters

Efficient Searching of Dynamic Sets

- Dynamic set (e.g., dictionary) operations are required by many applications:
  1. Insert
  2. Search
  3. Delete
- A Hash Table is an effective data structure that can provide
  - Average time complexity of O(1) for the basic operations
  - Worst case time complexity can be as bad as a linked list, O(*n*).

Hash Table

- Imagine that we could assign a unique array index to every possible key that could occur in an application.
  - Locating, inserting, deleting elements could be done very easily and quickly.
  - However, the key space may be much too large to use an array in a real system.
- A Hash Table is a generalization of an ordinary array that does not require one position for every possible key.
  - Advantageous when # of keys actually stored << # keys possible
  - Uses an array whose size is proportional to the # of keys stored
- The key is not used as the index!
- Instead, the array index is computed, by a hashing function, using the key.

Hashing to aid searching

- The purpose of hashing is to translate (via the *hash function*) an extremely large key space into a reasonable small range of integers (called the *hash code* or the *hash value*).
- Hash Table
  - An array H of indexes (*hash code*) 0, …, h-1
  - Hash function *hashCode*(k) maps a key k into an integer in the range 0, …, h-1
  - Each entry may contain one or more keys!
    - That is, the hash function is a many-to-one function

Hash Table Example

- Data k: 1055, 1492, 1776, 1812, 1918, and 1945
- Hash function
  - hashCode(k) = 5k mod 8
- hashCode: key
  - 0: 1776
  - 1:
  - 2:
  - 3: 1055
  - 4: 1492, 1812 // Collision!
  - 5: 1945
  - 6: 1918
  - 7:

Hash Table Problems

- The problem is that two keys can have the same hash code: collision
- What should we do?
  1. Pick the hash function hashCode(k) to minimize the number of collisions
  2. Implement the hash table in a way that allows keys with the same hash value to also be stored
- The first solution should be common sense, but often difficult to do. Why?
- The second method is almost always needed. Why?
- Two common collision resolution techniques:
  1. Chaining or Closed Address Hashing
  2. Open Addressing

Uniform Distribution of the Hash Code

- We want the hash code for each key in our set to be equally likely to be any integer in the range 0, …, h-1
- If n/h is a constant then
  - O(1) key comparisons can be achieved, on average, for successful search and unsuccessful search.
- Uniform distribution of the hash code depends on the choice of the Hash Function

Choosing a Hash Function

- If the key type is integer
  - hashCode(k) = (a k) mod h
  - Choose h as a power of 2, and h >= 8
  - Choose a = 8 Floor[h/23] + 5
- Or, let hashCode(k) = (k) mod p where p is a prime number close to the table size we want.
  - Avoid powers of 2 and 10 for values of p
  - This is sometimes called the division method
- If the key type is string of l characters, treat them as sequence of integers, k_1, k_2, k_3, …, k_l
  - hashCode(K) = $(a^l k_1 + a^{l-1} k_2 + \ldots + a\, k_l)$ mod h
- Use array doubling whenever the load factor $\alpha = n/h$ gets high, say 0.5 (where n is the number of elements in the table)

Closed Address Hashing (Open Hashing)

- H[i] is a linked list; hashCode(k) = 5k mod 8
- hashCode : key
  - 0: → 1776
  - 1: →
  - 2: →
  - 3: → 1055
  - 4: → 1492 → 1812
  - 5: → 1945
  - 6: → 1918
  - 7: →
- To search a given key k, first compute its hash code, say i, then search through the linked list at H[i], comparing k with the keys of the elements in the list.

Analysis of Searching with Closed Address Hashing

- **Basic Operation: comparisons**
  - **Assume computing a hash code equals a unit of comparison**
  - **there are total of *n* elements stored in the table,**
  - **each elements is equally likely to be search**
- **Average number of comparison for an unsuccessful search (including hashing) is $A_u(n) = 1 + n/h = \Theta(1+\alpha)$**
- **Average cost of a successful search**
  - **When key $i = 1, \ldots, n$, was inserted at the end of a linked list, each linked list had average length given by $(i-1)/h$**
  - **The expected number of key comparisons = 1 + comparisons made for inserting an element at the end of a linked list**

Analysis of Searching with Closed Address Hashing

- **How do we implement Insert?**
- **How do we implement Delete?**

Open Address Hashing (Closed Hashing)

- **All elements are stored in the array of the hash table, rather than using linked lists to accommodate collisions**
  - **If the hash cell corresponding to the hash code is occupied by a different element,**
  - **then a sequence of alternative locations for the current element is defined (by *rehashing*)**
- **Rehashing by linear probing**
  - **rehash(*j*) = (*j*+1) mod *h***
  - **where *j* is the location most recently probed,**
  - **initially *j* = *i*, the hash code for *k***
- **Rehashing by double hashing**
  - **rehash(*j*, *d*) = (*j* + *d*) mod h**
  - **e.g., d = hashIncr(*k*) = (2*k* + 1) mod h**
  - **computing an odd increment ensures that whole hash table is accessed in the search (provided *h* is a power of 2)**

Open Address Hashing with Linear probing

- **hashCode: key**
  - **0: 1776**
  - **1:**
  - **2:**
  - **3: 1055**
  - **4: 1492**
  - **5: 1945**
  - **6: 1918**
  - **7:**
- Now insert 1812, hashCode(1812) = 4, i.e., *i* = 4
  - » *h* = 8, initially *j* = *i* = 4
  - » rehash(*j*) = (*j*+1) mod *h*
  - » rehash(4) = (4+1) mod 8 = 5 // collision again
  - » rehash(5) = (5+1) mod 8 = 6 // collision again
  - » … put in 7

Retrieval and Deletion under Open Addressing Hashing

- **Retrieval procedure imitates the insertion procedure, stop search as soon as an empty cell is encountered.**
- **Deletion of a key**
  - **Cannot simply delete the the key and label the cell empty. Why?**
  - **Need to label the cell “obsolete”**
- **How do we implement Insert?**

Analysis of Searching with Open Address Hashing

- **Basic Operation: comparisons**
  - **Assume computing a hash code equals a unit of comparison**
  - **there are total of *n* elements stored in the table,**
  - **each elements is equally likely to be search**
  - **Again, $\alpha = n/h$ is the average number of keys per array index.**
    - **Note: $\alpha \le 1$ for open addressing**
- **The average number of comparison for an insertion or an unsuccessful search (including hashing) is at most $1/(1-\alpha)$**
- **The average number of comparison for a successful search is at most**