

G205

Fundamentals of Computer Engineering

CLASS 9, Wed. Oct. 6 2004

Stefano Basagni

Fall 2004

M-W, 1:30pm-3:10pm

Hash Tables, 1

- ◆ Dictionary operations: Insert, delete and Search
- ◆ Implementation of Dictionaries:

HASH TABLES

- ◆ Expected time for search: $O(1)$
- ◆ Worst case time for search: $\Theta(n)$

Hash Tables, 2

◆ Generalize ordinary arrays

- Ordinary array stores the element whose key is k in position k of the array
- Direct addressing: Given a key k , the element whose key is k is in the array k th position
- Direct addressing requires to allocate an array with one position for every possible key

Use of Hash Tables

- ◆ Number of keys actually stored is « of the number of possible keys
- ◆ Size is proportional to the number of keys to be stored (rather than the number of possible keys)
- ◆ Given a key k , the index into the array is not necessarily k
- ◆ Compute a function of k , and use that value to index into the array
- ◆ This function is a *hash function*

Direct-Address Tables, 1

◆ Scenario

- Maintain a dynamic set
- Each element has a key drawn from a universe $U = \{0, 1, \dots, m-1\}$
- m isn't too large
- No two elements have the same key

Direct-Address Tables, 2

- ◆ Represented by an array $T [0 \dots m-1]$:
 - Each **slot**, or position, corresponds to a key in U
 - If there's an element x with key k , then $T[k]$ contains a pointer to x
 - Otherwise, $T [k]$ is empty, represented by **NIL**

Dictionary Operations

◆ Very simple, all $O(1)$:

- **DIRECT-ADDRESS-SEARCH**(T, k)
return $T[k]$
- **DIRECT-ADDRESS-INSERT**(T, x)
 $T[key[x]] = x$
- **DIRECT-ADDRESS-DELETE**(T, x)
 $T[key[x]] = \text{NIL}$

Using Hash Tables

- ◆ If U is large direct-address is unpractical
- ◆ Often the set K of keys actually stored is small compared to U
 - When K is $\ll U$ less space is required than a direct-address table
 - Storage requirements are down to $\Theta(|K|)$.
 - Can still get **average** case $O(1)$ search (not worst case)

The Idea Behind Hashing

- ◆ Instead of storing an element with key k in slot k , use a function h and store the element in slot $h(k)$
- ◆ We call h a **hash function**
- ◆ $h : U \rightarrow \{0, 1, \dots, m-1\}$, so that $h(k)$ is a legal slot number
- ◆ We say that k **hashes** to slot $h(k)$

Collisions

- ◆ **Collisions:** Two or more keys hash to the same slot
 - Can happen when there are more possible keys than slots ($|U| > m$)
 - For a given set K of keys with $|K| \leq m$, may or may not happen. Definitely happens if $|K| > m$
 - Must be prepared to handle collisions in all cases
 - Two methods: chaining and open addressing
 - Chaining is usually better than open addressing

Collisions Resolution by Chaining

- ◆ All elements that hash to the same slot are organized into a list
- ◆ Slot j contains a pointer to the head of the list of all stored elements that hash to j
- ◆ If there are no such elements, slot j contains NIL

Dictionary Operations with Chaining: Insertion

CHAINED-HASH-INSERT(T, x)

insert x at the head of list T [$h(\text{key}[x])$]

- ◆ Worst-case running time is $O(1)$
- ◆ Element being inserted is not already in the list

(It would take an additional search to check if it was already inserted)

Dictionary Operations with Chaining: Search

CHAINED-HASH-SEARCH(T, k)

search for a key k element in list T
 $[h(k)]$

- ◆ Running time is proportional to the length of the list of elements in slot $h(k)$

Dictionary Operations with Chaining: Deletion

CHAINED-HASH-DELETE(T, x)

delete x from the list $T[h(\text{key}[x])]$

- ◆ Pointer x to the element to delete is given (so no search is needed)
- ◆ Worst-case running time is $O(1)$ time if the lists are doubly linked
- ◆ If the lists are singly linked, then deletion takes as long as searching (we must find x 's predecessor in its list to correctly update next pointers)

Analysis of Hashing with Chaining

- ◆ Given a key, how long does it take to find an element with that key, if any?
- ◆ Analysis is in terms of the **load factor**
$$\alpha = n/m$$
 - n = # of elements in the table
 - m = # of slots in the table = # of (possibly empty) linked lists
- ◆ α = number of elements per linked list
- ◆ Can have $\alpha < 1$, $\alpha = 1$, or $\alpha > 1$

Worst case analysis

- ◆ Worst case is when all n keys hash to the same slot: Single list of length n
- ◆ Worst-case time to search is $\Theta(n)$, plus time to compute hash function
- ◆ Average case depends on how well the hash function distributes the keys among the slots

Average Case Analysis, 1

- ◆ Assume **simple uniform hashing**: Any given element is equally likely to hash into any of the m slots
- ◆ For $j = 0, 1, \dots, m-1$, denote the length of list $T[j]$ by n_j . Then $n = n_0 + n_1 + \dots + n_{m-1}$
- ◆ Average value of n_j is $E[n_j] = a = n/m$
- ◆ Assume the hash function takes $O(1)$ time \rightarrow Search time depends on the length $n_{h(k)}$ of the list $T[h(k)]$

Average Case Analysis, 2

- ◆ We consider two cases:
 - Unsuccessful search: The hash table contains no element with key k
 - Successful Search: The hash table does contain an element with key k

Unsuccessful Search

- ◆ **Theorem:** An unsuccessful search takes expected time $\Theta(1 + \alpha)$
- ◆ **Proof** Simple uniform hashing \rightarrow any key not in the table is equally likely to hash to any of the m slots. Search unsuccessfully for any key $k =$ search to the end of the list $T[h(k)] \rightarrow$ Expected length $E[n_{h(k)}] = \alpha \rightarrow$ The expected number of elements examined in an unsuccessful search is α . Adding in the time to compute the hash function, the total time required is $\Theta(1 + \alpha)$.

Successful Search

- ◆ The expected time for a successful search is also $\Theta(1+a)$
- ◆ The circumstances are slightly different from an unsuccessful search
- ◆ The probability that each list is searched is proportional to the number of elements it contains
- ◆ **Theorem:** A successful search takes expected time $\Theta(1 + a)$

Analysis Interpretation

- ◆ If $n = O(m)$ then $a = n/m = O(m)/m = O(1)$
- ◆ Search takes constant time on average
- ◆ Insertion takes $O(1)$, worst case
- ◆ Deletion takes $O(1)$, worst case (doubly-linked list)
- ◆ **ON AVERAGE** dictionary operations take constant time

Hash Functions

- ◆ What is a good hash function

$$h:U \rightarrow \{0, 1, \dots, m-1\}?$$

- ◆ Simple uniform hashing (ideally)

- ◆ Unpractical: Key distribution is not known and possibly not independent

- ◆ Use heuristics based on the domain of keys to create a hash function that performs well

Keys as Natural Numbers

- ◆ Hash functions assume that keys are natural numbers (or interpreted as such)
- ◆ Example. Interpret strings in some radix notation: CLRS
 - ASCII values: C = 67, L = 76, R = 82, S = 83
 - There are 128 basic ASCII values
 - So interpret CLRS as $(67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0) = 141,764,947$

Division Method

- ◆ $h(k) = k \bmod m \rightarrow m=20, k=91 \rightarrow h(k)=11$
- ◆ **Advantage:** Fast, requires just one division
- ◆ **Disadvantage:** There are bad values of m :
 - Powers of 2 are bad. If $m = 2^p$ for integer p , then $h(k)$ is just the least significant p bits of k
 - If k is a character string interpreted in radix 2^p then $m = 2^p - 1$ is bad: permuting characters in a string does not change its hash value
- ◆ **Good choice for m :** A prime not too close to an exact power of 2

Multiplication Method

1. Choose constant A in the range $0 < A < 1$
 2. Multiply key k by A
 3. Extract the fractional part of kA
 4. Multiply the fractional part by m
 5. Take the floor of the result
- ◆ **Advantage:** Value of m is not critical
 - ◆ **Disadvantage:** Slower than division method

How to Choose A

- ◆ The multiplication method works with any legal value of A
- ◆ It works better with some values than with others, depending on the keys being hashed
- ◆ D. E. Knuth suggests using $A \approx (\sqrt{5}-1)/2$

Assignments

- ◆ Textbook, Chapter 11
- ◆ Updated information on the class web page:

www.ece.neu.edu/courses/eceg205/2004fa