WIA2005: Algorithm Design and Analysis

Semester 2, Session 2016/17

Lecture 8: Hash Table

Learning objectives

- Know what is
  - Direct access table
  - Hash table
  - Collision and chaining
  - Hash function
  - Open addressing

Introduction

- A hash table is an effective data structure for implementing dictionaries.
- Although searching for an element in a hash table can take as long as searching for an element in a linked list—O(n) time in the worst case—in practice, hashing performs extremely well.
- Under reasonable assumptions, the average time to search for an element in a hash table is O(1).

Direct Addressing

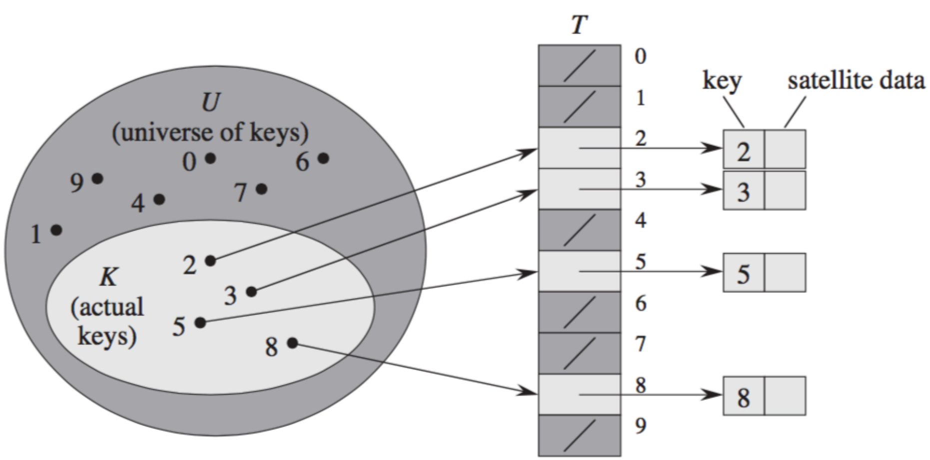
- A hash table generalizes the simpler notion of an ordinary array.
- Directly addressing into an ordinary array makes effective use of our ability to examine an arbitrary position in an array in O(1) time.

Direct-address tables

- Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.
- Suppose that an application needs a dynamic set in which each element has a key drawn from the universe U = {0, 1, 2,.., m-1}, where m is not too large.
- We shall assume that no two elements have the same key.

Direct-address table - Illustrated

- To represent the dynamic set, we use an array, or ***direct-address table***, denoted by T[0..m-1], in which each position, or ***slot***, corresponds to a key in the universe U .



Each key in the universe U = {0,1,..,9} corresponds to an index in the table. The set K = {2,3,5,8} of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

DIRECT-ADDRESS-SEARCH(T, k)

return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

$T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)

$T[x.key] = \text{NIL}$

Each of these operation takes O(1) time

Hash tables

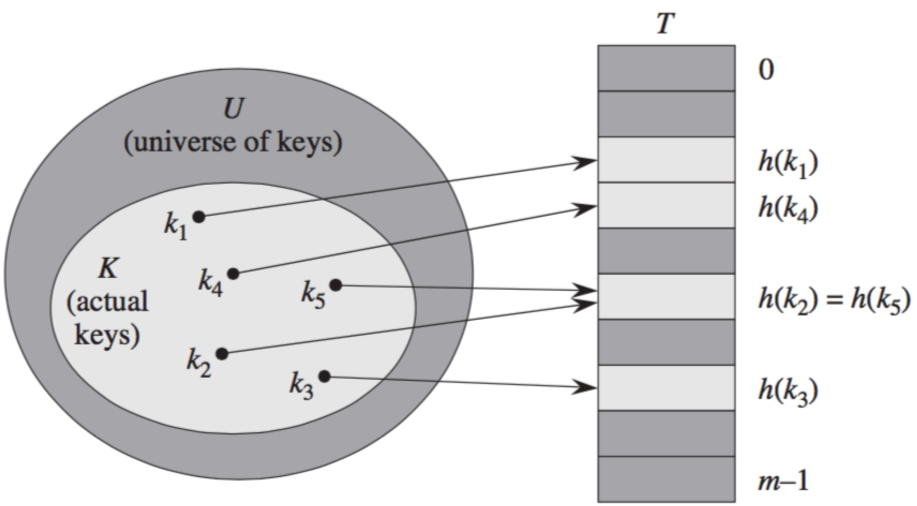
- The downside of direct addressing is obvious: if the universe U is large, storing a table T of size |U| may be impractical, or even impossible, given the memory available on a typical computer.
- Furthermore, the set K of keys *actually stored* may be so small relative to U that most of the space allocated for T would be wasted.
- With direct addressing, an element with key k is stored in slot k.
  - With hashing, this element is stored in slot h(k); that is, we use a ***hash function*** h to compute the slot from the key k.

Hash table - Illustrated

- The hash function reduces the range of array indices and hence the size of the array.
- Instead of a size of |U|, the array can have size m.
- Hash function h maps the universe U of keys into the slots of a ***hash table*** T[0..m-1] :

$$h : U \to \{0,1,..,m-1\}$$

  - where the size m of the hash table is typically much less than |U|. We say that an element with key k ***hashes*** to slot h(k), we also say that h(k) is the ***hash value*** of key k.
- There is one hitch: two keys may hash to the same slot. We call this situation a ***collision***.
- Fortunately, we have effective techniques for resolving the conflict created by collisions.



Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

Resolving collision by chaining

- In ***chaining***, we place all the elements that hash to the same slot into the same linked list.
- Each hash-table slot T[j] contains a linked list of all the keys whose hash value is j.
  - For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_7) = h(k_2)$.
- The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

O(1)

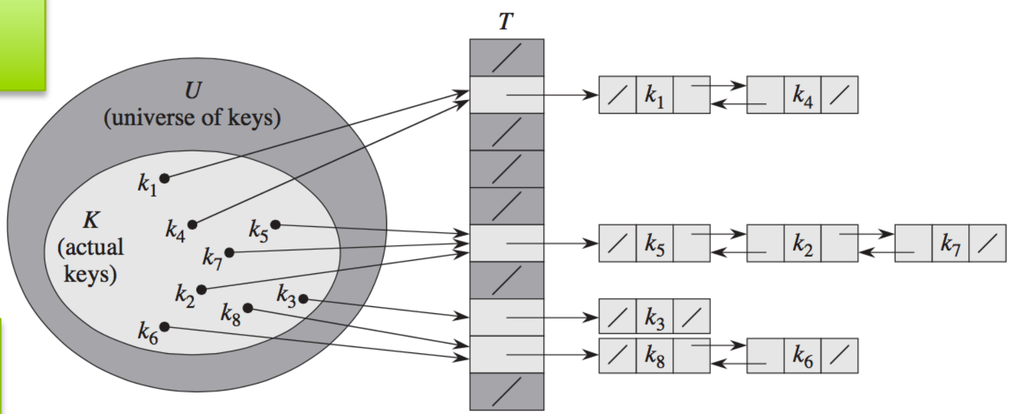
```
CHAINED-HASH-INSERT(T, x)
    insert x at the head of list T[h(x.key)]

CHAINED-HASH-SEARCH(T, k)
    search for an element with key k in list T[h(k)]

CHAINED-HASH-DELETE(T, x)
    delete x from the list T[h(x.key)]
```

?

O(1)



Analysis of hashing with chaining

- Given a hash table T with m slots that stores n elements, we define the ***load factor α*** *,* for T as n/m, that is, the average number of elements stored in a chain.
- Our analysis will be in terms of α, which can be less than, equal to, or greater than 1.
- The worst-case behavior of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n.
  - The worst-case time for searching is thus O(n) plus the time to compute the hash function—no better than if we used one linked list for all the elements.

Analysis of hashing with chaining - cont

- The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.
- But, for now we shall assume that any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to. We call this the assumption of ***simple uniform hashing***.

Unsuccessful search analysis

Theorem 11.1
In a hash table in which collisions are resolved by chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

Proof Under the assumption of simple uniform hashing, any key k not already stored in the table is equally likely to hash to any of the m slots. The expected time to search unsuccessfully for a key k is the expected time to search to the end of list $T[h(k)]$, which has expected length $\mathrm{E}[n_{h(k)}] = \alpha$. Thus, the expected number of elements examined in an unsuccessful search is α, and the total time required (including the time for computing $h(k)$) is $\Theta(1 + \alpha)$. ■

Successful search analysis

- What about successful search? What is the average running time for hash table with chaining?

Hash functions

- A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to.
- The methods:
    - Division method
    - Multiplication method

Interpreting keys as natural numbers

- Most hash functions assume that the universe of keys is the set of natural numbers.
  - Thus, if the keys are not natural numbers, we find a way to interpret them as natural numbers.
  - For example, we can interpret a character string as an integer expressed in suitable radix notation.

- In what follows, we assume that the keys are natural numbers.

The division method

- In the ***division method*** for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m.

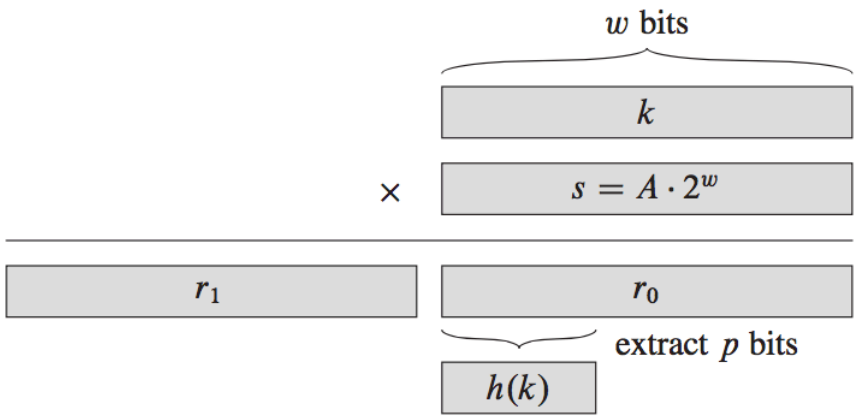
$$h(k) = k \bmod m$$

- For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$.
- Avoid:
  - m should not be a power of 2, since if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k.
  - A prime not too close to an exact power of 2 is often a good choice for m.

The multiplication method

- The ***multiplication method*** for creating hash functions operates in two steps.

$$h(k) = (A.k \bmod 2^w) >> (w - r)$$



- An advantage of the multiplication method is that the value of m is not critical.

Open addressing

- In ***open addressing***, all elements occupy the hash table itself.
  - That is, each table entry contains either an element of the dynamic set or NIL
- When searching for an element, we systematically examine table slots until either we find the desired element or we have ascertained that the element is not in the table.
- No lists and no elements are stored outside the table, unlike in chaining.
  - Thus, in open addressing, the hash table can “fill up” so that no further insertions can be made; one consequence is that the load factor α can never exceed 1.

Probing

- To perform insertion using open addressing, we successively examine, or ***probe***, the hash table until we find an empty slot in which to put the key.
- Instead of being fixed in the order 0, 1,.., m-1 (which requires O(n) search time), the sequence of positions probed *depends upon the key being inserted*.
- To determine which slots to probe, we extend the hash function to include the probe number (starting from 0) as a second input.

$$h: U \times \{0, 1,.., m-1\} \to \{0, 1,.., m-1\}$$

With open addressing, we require that for every key k, the ***probe sequence***

$$\langle h(k, 0), h(k, 1),.., h(k,m-1)\rangle$$

be a permutation of $\langle 0, 1,.., m-1\rangle$, so that every hash-table position is eventually considered as a slot for a new key as the table fills up.

Hash Insert and Search

- Each slot contains either a key or NIL (if the slot is empty).
- The HASH-INSERT procedure takes as input a hash table T and a key k.
  - It either returns the slot number where it stores key k or flags an error because the hash table is already full.
- The procedure HASH-SEARCH takes as input a hash table T and a key k, returning j if it finds that slot j contains key k, or NIL if key k is not present in table T .

```
HASH-INSERT(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL
        T[j] = k
        return j
    else i = i + 1
until i == m
error "hash table overflow"
```

```
HASH-SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL or i == m
return NIL
```

Hash Delete

- Deletion from an open-address hash table is difficult.
- When we delete a key from slot i, we cannot simply mark that slot as empty by storing NIL in it.
- If we did, we might be unable to retrieve any key k during whose insertion we had probed slot i and found it occupied.
- We can solve this problem by marking the slot, storing in it the special value DELETED instead of NIL.
- We would then modify the procedure HASH-INSERT to treat such a slot as if it were empty so that we can insert a new key there.

Probing techniques

- We will examine three commonly used techniques to compute the probe sequences required for open addressing:
  - Linear probing.
  - Quadratic probing.
  - Double hashing.

Linear probing

- Given and ordinary hash function h':U->{0, 1,..,m-1} which we refer to as an ***auxiliary hash function***, the method of ***linear probing*** uses the hash function

$$h(k, i) = (h'(k) + i) \bmod m$$

for i = 0, 1,..,m-1

- Linear probing is easy to implement, but it suffers from a problem known as ***primary clustering***.
- Long runs of occupied slots build up, increasing the average search time.

Quadratic probing

- ***Quadratic probing*** uses a hash function of the form

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

where h' is an auxiliary hash function, c_1 and c_2 are positive auxiliary constants, and $i = 0, 1, .., m-1$

- This property leads to a milder form of clustering, called ***secondary clustering***.

Double hashing

- Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations.
- ***Double hashing*** uses a hash function of the form

$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$$

where both h_1 and h_2 are auxiliary hash functions.

Analysis of open hashing

- With open addressing, at most one element occupies each slot, and thus $n \leq m$, which implies $\alpha \leq 1$.
- We assume that we are using uniform hashing.
- In this idealized scheme, the probe sequence <h(k, 0), h(k, 1),.., h(k,m-1)> used to insert or search for each key k is equally likely to be any permutation of <0, 1,.., m-1>.

Theorem 11.6

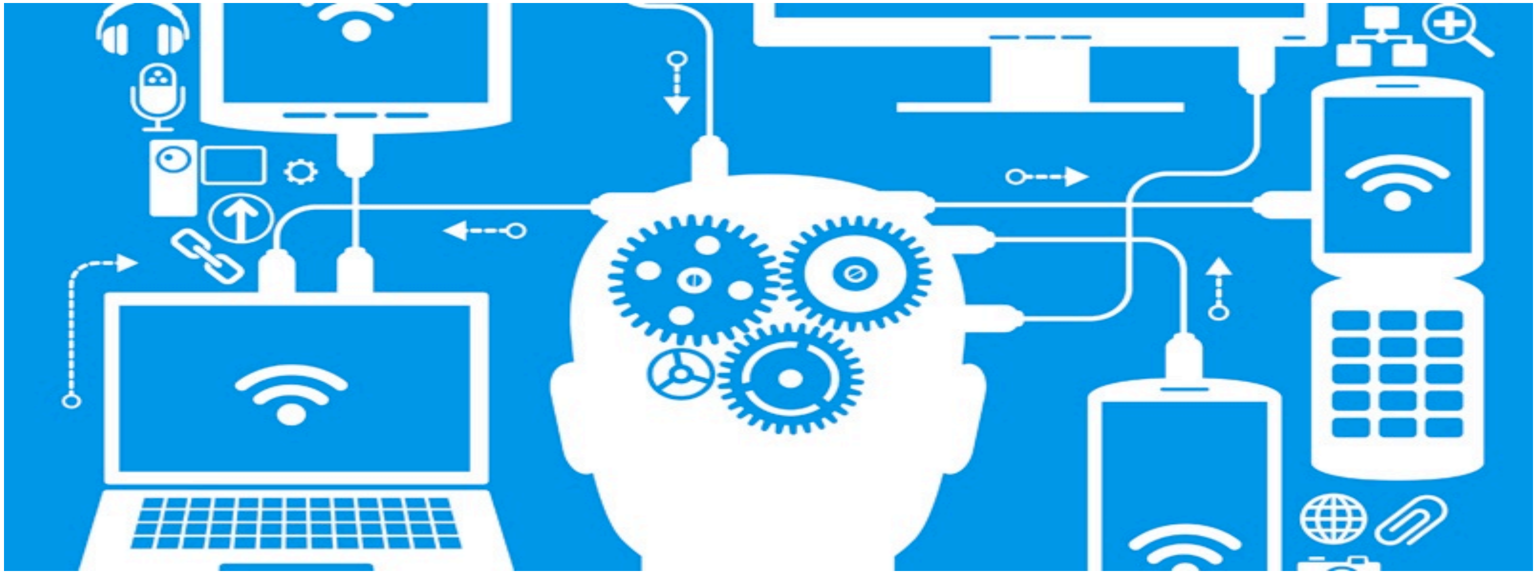
Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Proof

$$
\begin{aligned}
\Pr\{X \geq i\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \\
&\leq \left(\frac{n}{m}\right)^{i-1} \\
&= \alpha^{i-1} .
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\
&\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\
&= \sum_{i=0}^{\infty} \alpha^{i} \\
&= \frac{1}{1-\alpha} .
\end{aligned}
$$

In the next lecture..



Lecture 9: Dynamic Programmimg