Lecture 5: Hash Tables

Mircea Marin
mmarin@info.uvt.ro

October 30, 2012

Preliminary notions

- A dynamic set is a collection of items that can shrink (by removing elements from it), grow (by inserting into it), and on which we can test membership of an element.
- Hash tables are a kind of dynamic set whose elements are objects of an abstract data type (ADT) with an identifying **key** field. The object may contain **satellite data** and other fields that are manipulated by the set operations (e.g., pointers to other objects in the set).

SEARCH(S, k): returns a pointer x to an element in S such that $x.key = k$ or NIL if there is no such x.
INSERT(S, x): augments S with the element pointed to by x. We usually assume that all fields of x have been initialized before using this operation.
DELETE(S, x): given a pointer x to an element in the set S, removed x from S. (Note that this operation uses a pointer to an element x, not a key value.)
MINIMUM(S): a query on a totally ordered set that returns the element of S with the smallest key.
MAXIMUM(S): a query on a totally ordered set that returns the element of S with the largest key.
SUCCESSOR(S, x): a query that, given an element x whose key is from S, return the next larger element, or NIL is x is the maximum element.

PREDECESSOR(S, x): a query that, given an element x whose key is from S, return the next smaller element, or NIL is x is the minimum element.

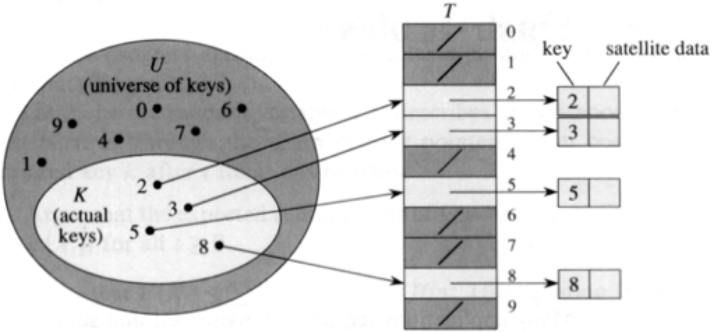
- SUCCESSOR and PREDECESSOR are often extended to sets with nondistinct keys. For a set with n keys, it is assumed that a call to MINIMUM followed by $n - 1$ calls to SUCCESSOR enumerates the elements in the set in sorted order.
- The time taken to execute an operation is usually measured in terms of the size of the given collection.

Dictionaries

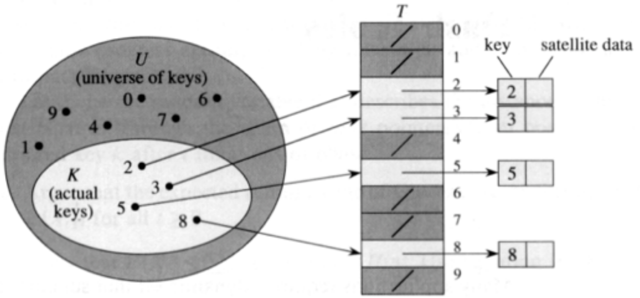
- Dictionaries are dynamic sets that support only the dictionary operations INSERT , SEARCH, and DELETE.
- Typical implementations of dictionaries:
  - Direct-access tables
  - Hash tables
- Requirement: average run-time complexity of dictionary operations should be $O(1)$.

Direct-access tables

- Simple technique to implement a dictionary when the set U of keys is reasonably small.
- **Scenario:** An application needs a collection of elements with keys from $U = \{0, 1, \ldots, m - 1\}$, where m is not too large. We assume that no 2 elements have the same key.



Direct-access tables



- m is the maximum number of keys.
- $T[0..m-1]$ is the direct-access table, implemented as an array, in which each position (or **slot**), corresponds to a key from U:
  - If $x = T[k]$, then $x.key = k$.
  - If there is no x with key k then $T[k] = \text{NIL}$.

DIRECTADDRESSACCESS(T, k)
return $T[k]$

DIRECTADDRESSINSERT(T, x)
$T[x.key] := x$

DIRECTACCESSDELETE(T, x)
$T[x.key] :=$NIL

$\text{DIRECTADDRESSACCESS}(T, k)$
  return $T[k]$

$\text{DIRECTADDRESSINSERT}(T, x)$
  $T[x.key] := x$

$\text{DIRECTACCESSDELETE}(T, x)$
  $T[x.key] :=$NIL

- Time complexity of these operations: $O(1)$.

Hash tables

Problems with direct addressing:

- Impractical when the set of keys U is very large: allocating a table of T size $|U|$ may be even impossible.
- The set K of keys actually stored may be very small relative to U
  $\Rightarrow$ most of the space allocated for T would be wasted.
- In such situations, it is better to use
  - A hash table $T[0..m-1]$, together with
  - A hash function $h : U \to \{0, 1, \ldots, m-1\}$ that hashes keys from U to slots in $\{0, \ldots, m-1\}$.
- Since there are more keys than slots, different keys get hashed to the same slot (they collide).
  - Collisions must be resolved.
  - The choice of a good hash function can avoid many collisions.

Hash tables

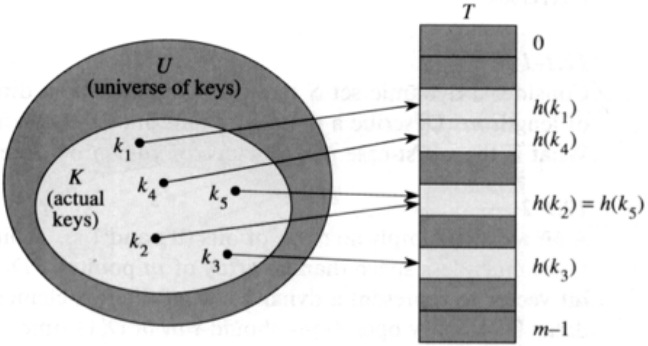
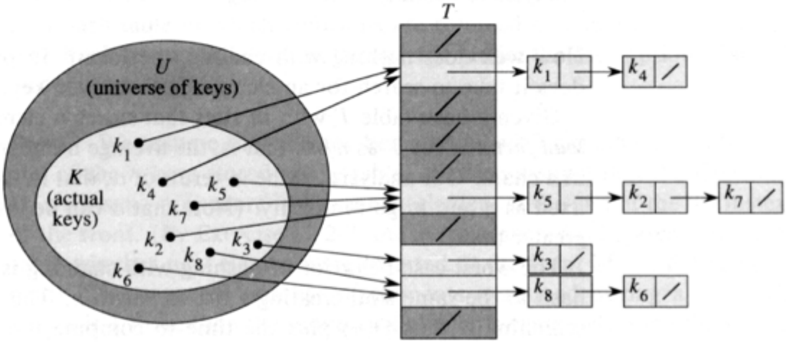


Figure 12.2 Using a hash function h to map keys to hash-table slots. Keys k_2 and k_5 map to the same slot, so they collide.

MAIN IDEA:

- Elements that hash to the same slot are put in a linked list.



- Each hash-table slot $T[j]$ contains a linked-list of all the keys whose hash value is j.
E.g., $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

CHAINEDHASHINSERT(T, x)
insert x at the head of list $T[h(x.key)]$

CHAINEDHASHSEARCH(T, k)
search for an element with key k in list $T[h(k)]$

CHAINEDHASHDELETE(T, x)
delete x from the list $T[h(x.key)]$

CHAINEDHASHINSERT(T, x)
insert x at the head of list $T[h(x.key)]$

CHAINEDHASHSEARCH(T, k)
search for an element with key k in list $T[h(k)]$

CHAINEDHASHDELETE(T, x)
delete x from the list $T[h(x.key)]$

Worst-case running time complexity analysis:

- CHAINEDHASHINSERT(T, x): $O(1)$
- CHAINEDHASHSEARCH(T, k): running time proportional to the length of $T[h(k)]$
- CHAINEDHASHDELETE(T, x):
  - $O(1)$ if $T[k(x.key)]$ is doubly linked list.
  - Time to find x in $T[k(x.key)]$ if $T[k(x.key)]$ is singly linked.

- The load factor α of a hash table T with m slots that stores n elements is the average number of elements stored in a chain, that is, $\alpha = n/m$.
- Worst case: all n elements have the same hash value $\Rightarrow$ all elements are stored in one linked list
  $\Rightarrow$ worst case time = worst case time for searching (which is $\Theta(n)$)+time to compute the hash function
  = worse than if we used on linked list to store all elements.
- The performance of hashing depends on how well the hashing function h distributes the set U of keys among the m slots of T.

Simple uniform hashing assumes that h distributes keys with equal likelihood into any of the m slots of the hash table.

- Assumptions:
  - simple uniform hashing with chaining.
  - The value of $h(k)$ can be computed in $O(1)$ time.

$\Rightarrow$ searching an element with key k depends linearly on the length of the list $T[h(k)]$.

Theorem

On average, an unsuccessful search takes time $\Theta(1+\alpha)$, under the assumption of simple hashing.

Theorem (Cormen *et al.*, 2000)

On average, a successful search takes time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing with load factor α.

Corollary

If the number of hash-table slots is at least proportional to the number of elements in the table, we have $n = O(m)$ and, consequently, $\alpha = n/m = O(m)/m = O(1)$. In this case, searching takes constant time on the average.
The other dictionary operations (DELETE and INSERT)can be performed in constant time $O(1)$ too.

Good hash functions

- A good hash function must satisfy (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots.
- Suppose we know the probability distribution P, that is, for every k we know the probability $P(k)$ to draw k from the set U of all keys. Then the assumption of simple uniform hashing is

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \quad \text{for } j = 0, 1, \ldots, m-1.$$

- This condition is generally impossible to check, because we usually don't know P.
- Special case when P is known: the keys are random real numbers independently and uniformly distributed in the range $0 \leq k < 1$. In this case $h(k) := \lfloor k\, m \rfloor$ satisfies the assumption of simple uniform hashing.

Interpreting keys as natural numbers

- Most hash functions assume that the set U of keys is (a subset of) the set $\mathbb{N}$ of natural numbers.
- If the keys are not from $\mathbb{N}$, we shall find a way to interpret them as natural numbers.

Example

The codes of ASCII characters are numbers between 0 and $2^7 - 1 = 127$. Therefore, any string "$c_1 c_2 \ldots c_n$" of ASCII characters can be interpreted as the number

$$128^{n-1} k_1 + 128^{n-2} k_2 + \ldots + k_n$$

where k_i is the ASCII code of character c_i for all $1 \leq i \leq n$. Note that this number can be quite large.

Hash functions with the division method

- $h(k) = k \mod m$.

Example

$m = 12, k = 100$. Then $h(k) = 4$.

- Very simple and fast to compute.
- Good values for m are prime numbers not too close to exact powers of 2.

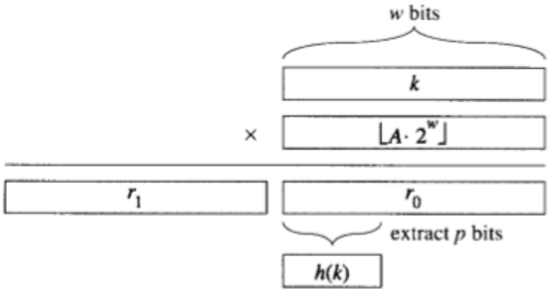
Hash functions with the multiplication method

Computation of $h(k)$ proceeds in 2 steps:

1. Multiply k by a constant $0 < A < 1$ and take the fractional part of $k \cdot A$,
2. Multiply this value by m and take the floor of the result.

Formally, $h(k) = \lfloor m\,(k\,A \mod 1)\rfloor$, where "$k\,A \mod 1$" means the fractional part of $k\,A$, that is, $k\,A - \lfloor k\,A\rfloor$.

- Typical choice for m: $m = 2^p$ for some integer p.



Universal hashing

- Malitious users who know the hash function may choose n keys that all hash to the same slot $\Rightarrow$ average retrieval time $\Theta(n)$.

Universal hashing

- Malitious users who know the hash function may choose n keys that all hash to the same slot $\Rightarrow$ average retrieval time $\Theta(n)$.
- This can be avoided by universal hashing:

Definition

A finite set $\mathcal{H}$ of hash functions that map keys from U into the range $\{0, 1, \ldots, m-1\}$ is universal if, for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is precisely $|\mathcal{H}|/m$.

- Malitious users who know the hash function may choose n keys that all hash to the same slot $\Rightarrow$ average retrieval time $\Theta(n)$.
- This can be avoided by universal hashing:

Definition

A finite set $\mathcal{H}$ of hash functions that map keys from U into the range $\{0, 1, \ldots, m-1\}$ is universal if, for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is precisely $|\mathcal{H}|/m$.

- The hash function is chosen randomly, at run time, from a carefully designed collection of hash functions.

Universal hashing

- Malitious users who know the hash function may choose n keys that all hash to the same slot $\Rightarrow$ average retrieval time $\Theta(n)$.
- This can be avoided by universal hashing:

Definition

A finite set $\mathcal{H}$ of hash functions that map keys from U into the range $\{0, 1, \ldots, m-1\}$ is universal if, for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is precisely $|\mathcal{H}|/m$.

- The hash function is chosen randomly, at run time, from a carefully designed collection of hash functions.
- The algorithm can behave differently on each execution, even for the same input.

Universal hashing

- Malitious users who know the hash function may choose n keys that all hash to the same slot $\Rightarrow$ average retrieval time $\Theta(n)$.
- This can be avoided by universal hashing:

Definition

A finite set $\mathcal{H}$ of hash functions that map keys from U into the range $\{0, 1, \ldots, m-1\}$ is universal if, for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is precisely $|\mathcal{H}|/m$.

- The hash function is chosen randomly, at run time, from a carefully designed collection of hash functions.
- The algorithm can behave differently on each execution, even for the same input.
  $\Rightarrow$ good average-case performance, no matter what keys are provided as input.

Universal hashing (continued)

Theorem

If h is chosen from a universal collection of hash functions and is used to hash n keys into a table of size m where $n \leq m$, the expected number of collisions involving a particular key x is less than 1.

Universal hashing (continued)

Theorem

If h is chosen from a universal collection of hash functions and is used to hash n keys into a table of size m where $n \leq m$, the expected number of collisions involving a particular key x is less than 1.

How easy is to design a universal class of hash functions?

Universal hashing (continued)

Theorem

If h is chosen from a universal collection of hash functions and is used to hash n keys into a table of size m where $n \leq m$, the expected number of collisions involving a particular key x is less than 1.

How easy is to design a universal class of hash functions?

- Choose table size m to be a prime number.

Universal hashing (continued)

Theorem

If h is chosen from a universal collection of hash functions and is used to hash n keys into a table of size m where $n \leq m$, the expected number of collisions involving a particular key x is less than 1.

How easy is to design a universal class of hash functions?

- Choose table size m to be a prime number.
- Decompose key x into $r + 1$ chunks $(x_0, x_1, \ldots, x_r)$ such that the maximum value of every chunk is $< m$.

Universal hashing (continued)

Theorem

If h is chosen from a universal collection of hash functions and is used to hash n keys into a table of size m where $n \leq m$, the expected number of collisions involving a particular key x is less than 1.

How easy is to design a universal class of hash functions?

- Choose table size m to be a prime number.
- Decompose key x into $r+1$ chunks $(x_0, x_1, \ldots, x_r)$ such that the maximum value of every chunk is $< m$.
- For any $a = (a_0, a_1, \ldots, a_r)$ randomly chosen from $\{0, 1 \ldots, m-1\}$, define the hash function $h_a(x) := \sum_{i=0}^{r} a_i\, x_i \mod m$, and let $\mathcal{H} := \left\{h_a \mid a = (a_0, \ldots, a_r) \in \{0, 1, \ldots, m-1\}^{r+1}\right\}$.

Universal hashing (continued)

Theorem

If h is chosen from a universal collection of hash functions and is used to hash n keys into a table of size m where $n \leq m$, the expected number of collisions involving a particular key x is less than 1.

How easy is to design a universal class of hash functions?

- Choose table size m to be a prime number.
- Decompose key x into $r + 1$ chunks $(x_0, x_1, \ldots, x_r)$ such that the maximum value of every chunk is $< m$.
- For any $a = (a_0, a_1, \ldots, a_r)$ randomly chosen from $\{0, 1 \ldots, m-1\}$, define the hash function $h_a(x) := \sum_{i=0}^{r} a_i\, x_i \mod m$, and let $\mathcal{H} := \left\{h_a \mid a = (a_0, \ldots, a_r) \in \{0, 1, \ldots, m-1\}^{r+1}\right\}$.
- Then $\mathcal{H}$ is a universal class of hash functions.

Open addressing

- All elements are stored in the hash table:
  - The insertion of an element must examine (or probe) the hash table until it finds an empty slot in which to put the key.
  - The sequence of positions probed depends on the key being inserted:
    - Instead of $h : U \to \{0, 1, \ldots, m-1\}$, we have $h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$, where the second argument is the probe number (starting from 0) of the hash function.
    - Requirement: for every key $k \in U$, the probe sequence

$$(h(k, 0), h(k, 1), \ldots, h(k, m-1))$$

should be a permutation of $\{0, 1, \ldots, m-1\}$

```
HASHINSERT(T, x)
i := 0 k := x.key
repeat
    j = h(k, i)
    if T[j] = NIL
       T[j] := x
       return j
    else i := i + 1
until i = m
error "hash overflow"
```

HASHSEARCH(T, x)
$i := 0$
$k := x.key$
repeat
    $j = h(k, i)$
    if $T[j] = x$
        return j
    $i := i + 1$
until $T[j] =$ NIL or $i = m$
return NIL

Poses some difficulties:

- "Deleting an element from slot i" $\neq$ "store NIL in slot i," because if we do so, we can not retrieve any element x with key k inserted after slot i was occupied.
- Possible solution:
  - Mark slot i with a special value DELETED instead of NIL.
  - Modify HASHSEARCH so that it keeps on looking when it encounters DELETED
  - Modify HASHINSERT to treat a DELETED slot as if it were empty so that a new key can be inserted.

Probe sequences must fulfill the requirement that $(h(k,1),\ldots,h(k,m))$ is a permutation of $\{1,\ldots,m\}$ for each key k.

- There are 3 commonly used techniques to compute probe sequences for open addressing:

  Linear probing: $h(k,i) = (h'(k) + i) \mod m$
  for $0 \leq i < m$
  where h' is an ordinary hash function.

  Quadratic probing: $h(k,i) = (h'(k) + c_1\, i + c_2\, i^2) \mod m$
  for $0 \leq i < m$, c_1, c_2 constants with $c_2 \neq 0$,
  and h' is an ordinary hash function.

  Double hashing: $h(k,i) = (h_1(k) + i\, h_2(k)) \mod m$
  where h_1, h_2 are auxiliary ordinary hash functions.

Chapter 12: Hash Tables from

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 2000.

Homeworks and exercises

(1) Indicate the insertion of keys 5, 28, 19, 15, 20, 33,12, 17, 10 into a hash table with collisions resolved by chaining. Assume that the table bas 9 slots and that the hash function is $h(k) = k \mod 9$.

(2) A **bit vector** is simply an array of bits (0 and 1). A bit vector of length m occupies much less space then an array of m pointers. Describe a way to use a bit vector to represents a collection of distinct elements with no satellite data. Dictionary operations should run in $O(1)$ time.

(3) Consider a collection of strings, where no 2 strings start with the same character.

- Implement a direct-access table together with the dictionary operations, where the hashing of a string is the ASCII code of the first character of the string. Example: $h(''alligator'') = 97$ because the ASCII code of $'a'$ is 97.

NOTE: The ASCII codes of string characters are integer numbers between 0 and 127.

(4) Implement a hash table for a collection of strings, together with the dictionary operations, where the hashing of a string is the ASCII code of the first character of the string.