Parma University, Math. Deptartment

Algorithms and Data Structures
Closed Hashing

Marco Pellegrini

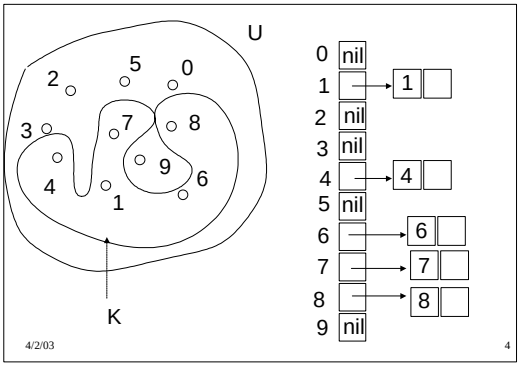
Summary of lecture 10

- (CLR 12.1-3)
  DS x Dictionary ADS:
  Direct address tables,
  Hash tables:
  Hashing with chaining
  hash functions

Direct address-tables.

- We see now a DS implementing a Dynamic Dictionary with operations: Insert, Delete, Search.
- We suppose input objects of the form (key, data).
- Insert/delete/search is done on the key.
- The key is taken from a Universe U, and the size of the Universe is M.
  To make things simple set U={0,1,2,..M-1}
- For Direct Address tables we assume also that:
  1) M is not too large.
  2) no two elements of the set K we want to store have the same key.

Example of direct address tables



Observations

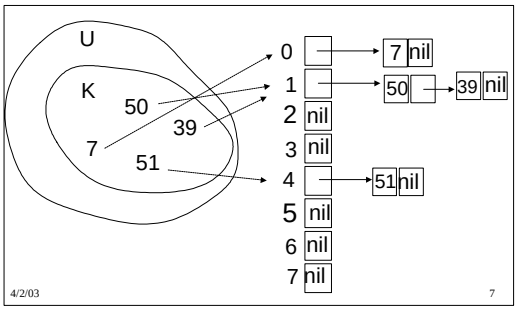
- the code is very simple:
- DIR-ACC-SEARCH(T,k)
  RETURN T[k]
- DIR-ACC-INSERT(T,x)
  T[key(x)] := ^x
- DIRR-ACC-DELETE(T,x)
  T[key(x)] := nil
- Each operation is O(1). The problem is when M is not small but large. In this case we want to use storage O(size(K)) not O(size(U)).

hash tables.

- Hash tables support insert/delete/search on a set K using storage O(size(K)). Each operation takes O(1) on average!
- The main idea is not to use the key to index an array but to compute the index from the key using an hash function.
- hash: U --> {0,1,....,m-1}
- So an element with key k is inserted at a position hash(k) in an array T[0,..m-1].
- Problem since M > m, then there are elements that hash in the same position (a collision).
- One technique to solve collision is called: chaining.

hashing with chaining

- We keep all elements hashing in the same slot in a linked list.



Operations on hash tables with chaining

- CH-HASH-INSERT(T,x)
  insert x at head of list T[hash(key(x))]
- CH-HASH-SEARCH(T,k)
  search for k in list T[hask(k)]
- CH-HASH-DELETE(T,^x)
  Delete x from list T[hash(key(x))]
- Insert and delete take time O(1), search takes time proportional to the length of the list T[hash(key(x))].
- Calling n the size of K, the Load factor is the ratio A= n/m.
- The performance of search depends on how well the hash function spreads the elements on the array T.

- Assume that the hashing is simple and uniform, meaning that taking any element X in K uniformly at random, the probability of hash(X)=i is 1/m for each slot i.
- Under this hypothesis, the expected length of a list is equal to the load factor A so searching takes time O(1+A) for unsuccessful search and O(1+A/2) for successful searches.
- Exercise: If we keep the lists in sorted order what is the effect on the running time of the operations insert/delete/retrieve(successfully) search (unsuccessfully)?

How to find a good hash function.

- The performance of the hash-table DS depends on the choice of the hash function.
- We should try to get as close as possible to the simple uniform hashing behavior.
- If the keys k are drawn uniformly in the interval [0..1] then the function $h(k) = \lfloor km \rfloor$ can be easily proven to satisfy the uniform hashing hypothesis.
- If we do not know how the keys are drown from U we use heristic methods: we shall see: the *division method*, the *multiplication method* and the *universal hashing method*.

Keys as numbers

- We assume keys to be natural numbers. If they are not, for example they are strings out of the alphabet of 26 characters, we transform them into numbers first.
- Usually we make sure that no two strings are represented by the same number.
- Example: MAY = $(13,1,24)_{26}$ = $13x26^2 + 1x26^1 + 24$= = 8838
- Sometimes we can just access the strings of bits representing the key and express it in octal notation.

Division method

- We take the reminder of the key after division by m.
- h(k) = k mod m.
- Tips. Certain values of m should be avoided, for example m should not be a power of 2, nor of 10.
- It can be shown that if $m = 2^p - 1$, then strings that are identical except for transposition of two adjacent character map onto the same address.
- Usually we choose m to be a prime number which is far from a power of 2 or 10.

Multiplication method

- Taka a constant c between 0 and 1.
- $h(k) = \lfloor (kc \bmod 1)\, m \rfloor$.
- Multiply k by c, take the fractional part, multiply by m, take the integer part.
- Here we have more freedom in choosing m, the size of the hash-table. It can be a power of two.
- A value of c suggested by Knuth is the *golden ratio*: $(\sqrt{5}-1)/2 = 0.618033...$

Universal hashing

If a malicious adversary chooses the keys to store in your table and knows your hashing function it can give you keys so that they all hash in the same address.

The idea of universal hashing is to choose your hash function at random among a set of candidates.

Given a collection of hash functions H, this collection is called universal if, for each pair of distinct keys x, y in U, the number of h in H for which h(x)=h(y) is exactly size(H)/m.

Theorem[12.3]
If h is chosen from an universal collection H and is used to hash n keys into an m-table, then the expected number of collisions involving a fixed key x is (n-1)/m.

Note that here the expectation is with respect to your random choice of h in H, we do not suppose anything on how the n keys are chosen.

Proof: Fix two keys x,y and choose h randomly in H. Set c(x,y,h)=1 if h(x)=h(y), set c(x,y,h)=0 if $h(x)\neq h(y)$. By definition of universal family:

$E[c(x,y,h)] = 1/m$. The collisions with x are $C(x,h)=\sum_{y\in T} c(x,y,h)$. So

$E[C(x,h)] = E[\sum_{y\in T} c(x,y,h)] = \sum_{y\in T} E[c(x,y,h)] \leq (n-1)/m$.

Building a universal family H.

- Choose m to be a prime number.
- Choose r+1 numbers in {0,...,m-1}, call them $S = a_0, a_1, a_2, .. a_r$.
- Take the key x in binary and split it into r+1 bytes, $x = (x_0, x_1, ..., x_r)$.
- We choose r and m so that $x_i < m$, for i=0,..,r.
- Build $h_S(x) = (\sum_{i=0,r} a_i x_i) \bmod m$
- H is the collection of all functions for all m^{r+1} choices of S.

Proof of universality

Theorem[CLR 12.4] The class H is universal.

Note that to generate at random h in H one does not have to create the whole H, which might be huge, it is enough to generate uniformly at random a sequence S of r+1 numbers from {0,1,..m-1}.

Lemma [CLR 33.25]
If gcd(a,m)=1, the equation $ax=b \pmod m$ has a unique solution.

Take two keys $x=(x_0,x_1,..x_r)$ and $y=(y_0,y_1,..y_r)$, such that $x_0 \neq y_0$.

Fix $(a_1,\ldots,a_r)$. Now h(x)=h(y) is equivalent to:

$a_0(x_0-y_0) = \sum_{i=1,r} a_i(y_i-x_i) \bmod m$ which by the lemma has a uniqe solution.

A random value a_0 has probability 1/m of being that solution.

Exercises and HW.

Exercise: hash the values 61,62,63,64 and 65 in a table of size m=1000 using the multiplication method with $c = (\sqrt{5}-1)/2 = 0.6180339...$

Exercise: Use h(x)= x mod 9 to hash

5,28, 19,15, 20, 33, 12, 17 ,10 in a table with 9 slots.

HW2.5 (CLR 12.3-1) Suppose you have to search a linked list where each element is a very long string. How could you use hash functions to speed up the search?

Conclusions

- Direct-address tables are useful for sets out of a small universe of keys.
- For large universes of keys we use hashing. Collision can be solved by chaining.
- Good hash functions can be obtained by the division method, the multiplication method and by choosing a function at random in a universal family of hash functions.
- Search take O(1 + n/m) when the hash function has the simple uniform property.
- After searching Insert/delete take O(1) additional time.