Hashtables

Presentation by Antonio-Gabriel Sturzu

Agenda

- Direct address tables
- Hashtables
  - Chaining
  - Good hash functions
  - Open addressing

Direct address tables

- The main goal is to design a datastructure that supports the basic dictionary operations (Insert, Search, Delete) in O(1) time
- We have a universe of keys U and we want to map those keys in a table
- With direct address tables an array is used and the value of the key is the actual index in the address table

Direct address tables

- This data structure works well when the size of the universe is small enough and the values of the keys is also small
- Let U={0,1,..... m-1} the universe of the keys
- Then we can store these keys in the array T[0..m-1]
- The array will also contain additional info associated with the keys

Direct address tables

- Operations

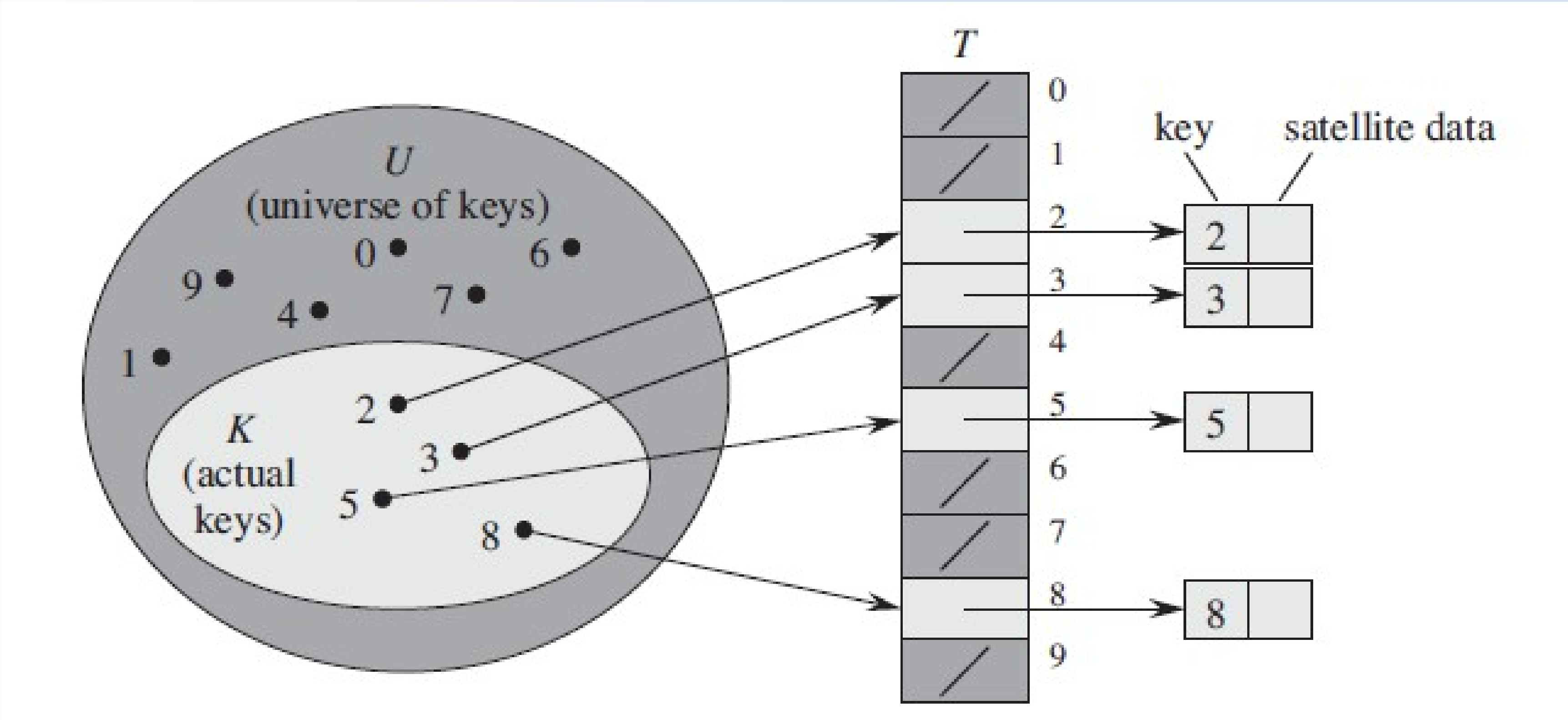
$\text{DIRECT-ADDRESS-SEARCH}(T, k)$
1 **return** $T[k]$

$\text{DIRECT-ADDRESS-INSERT}(T, x)$
1 $T[x.key] = x$

$\text{DIRECT-ADDRESS-DELETE}(T, x)$
1 $T[x.key] = \text{NIL}$

Each of these operations takes only $O(1)$ time.

Direct address tables



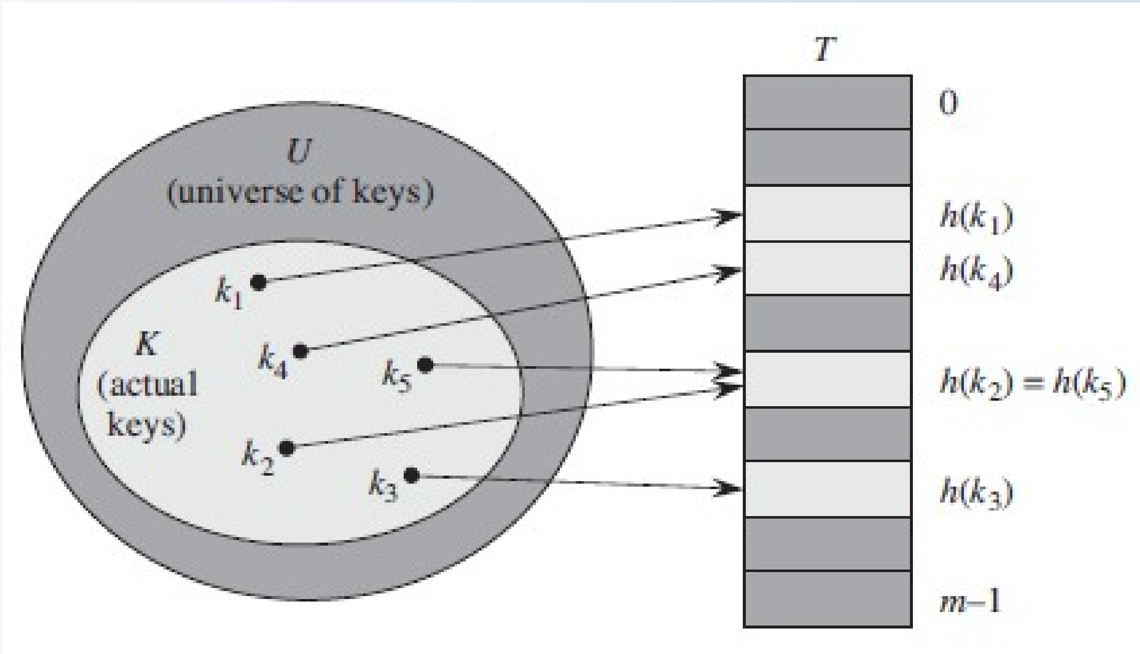
Hashtables

- Hashtables
- They optimize the use of memory when the size of the universe of keys is very big and the number of actual stored keys is relatively small
- Also they optimize the use of memory when the values of the keys are large
- The main disadvantage is that they do not preserve the O(1) worst case time for the dictionary operations

Hashtables

- The good news is that the O(1) time is kept on average for all the dictionary operations
- A hashtable uses a hash function that maps a key to a slot in the array
- It can be defined like $h:U \to \{0,1,\ldots\ldots m-1\}$ where h is the hash function, U is the universe of the keys and m is the size of the actual hash array

Hashtables



T
0
U
(universe of keys)
$h(k_1)$
$h(k_4)$
k_1
K
(actual
keys)
k_4
k_5
$h(k_2) = h(k_5)$
k_2
k_3
$h(k_3)$
$m-1$

Hashtables

- From the picture we observe that some keys get hashed in the same slot
- These are called collisions and they can‘t be avoided when $|U|>m$
- So, in practice it is good to find a hash function that generates few collisions but we still need to treat them

Hashtables

- The first solution for collisions is to use chaining
- In this solution in every slot of the array we store a pointer to the head of a linked list
- When a collision occurs we just add the new element to the head of the list

Hashtables

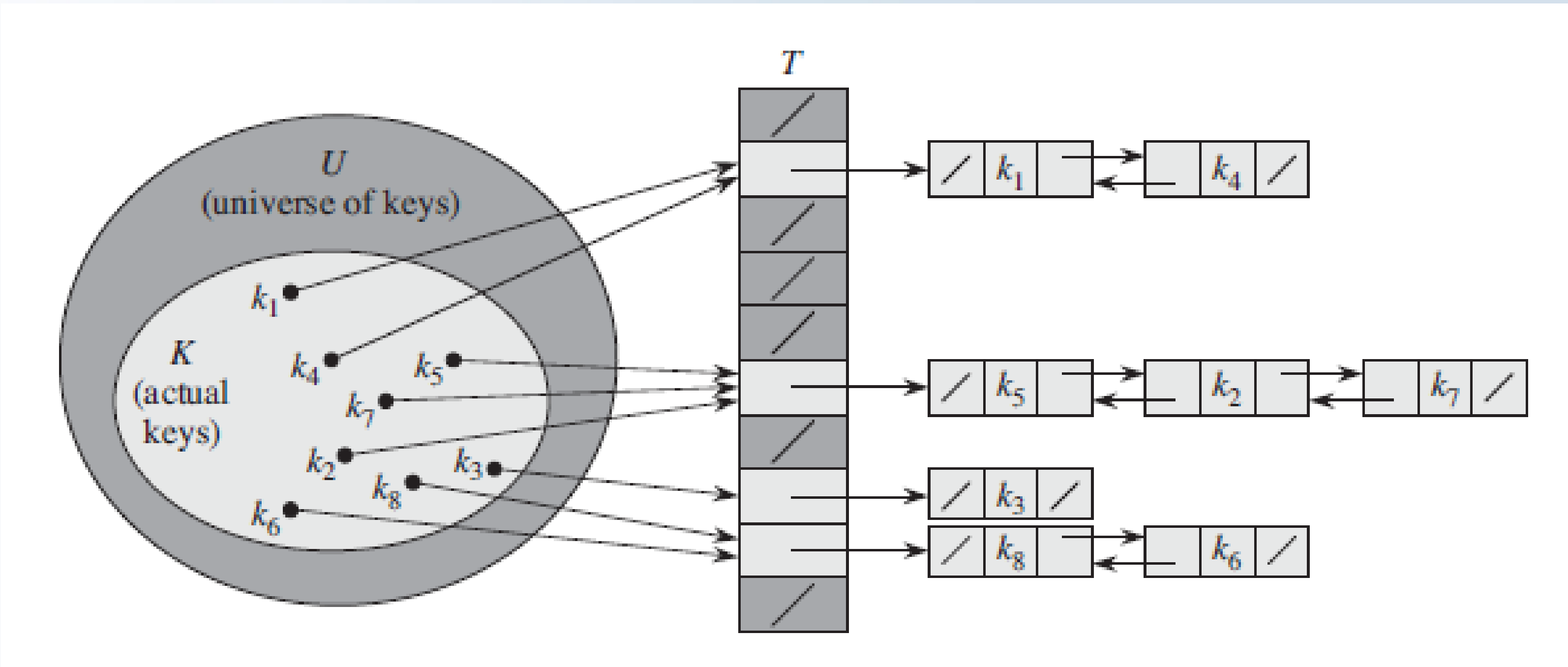
- The operations can be implemented like this

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

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

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

Hashtables



Hashtables

- The insertion and deletion can be implemented in O(1) worst case time
- The problem is the search operation because if we receive a key that hashes to a slot with collisions in the worst case we must scan through all the linked list
- In the worst case when we use chaining all the elements could end up chained in the same slot
- In this situation the worst case behavior will be $\Theta(n)$ where n is the number of hashed keys

Hashtables

- If n is the number of stored elements and m is the number of slots in then array we define L=n/m as the load factor of the hashtable
- It approximates the average number of chained elements in a slot
- L can be less than, equal or greater than 1
- The average case performance depends on how well the hash function h distributes the elements among the m slots

Hashtables

- The analysis of the search operation assumes that each element is equally likely to be hashed among all the m slots
- This assumption is called simple uniform hashing
- It has been proved that the average case time for the search operation for chaining is $\Theta(1+L)$
- So, if $n=O(m)$ the average runtime for the search operation will be $O(1)$

Hashtables

- But this time complexity holds under the assumption of simple uniform hashing
- An approximation of simple uniform hashing can be obtained by using only an efficient hash function !

Hash functions

- In general it is difficult for a hash function to assure a simple uniform hashing because we don't know in advance the probability distribution of the keys
- In some situation we can know the probability distribution in advance
- For example say we know that the keys are real random numbers independently and uniformly distributed in the range $0 \le k < 1$

Hash functions

- A hash function in this case that will respect the simple uniform hashing condition could be h(k)= [km]
- A good hash function should give hash values in a way that is independent of the possible patterns in the data
- For example if we have to hash strings that are differ by few letters the hash function should be able to disperse them as much as possible

Hash functions

- Also, before applying the hash function we should transform the keys into natural numbers in general
- If we receive strings with ASCII characters we could transform the characters into their ASCII code and after that consider each code as a digit in some base
- For example we could use base 128

Hash functions

- The first good method for hashing is the division method
- The hash function is h(k)= k mod m where m is the number of slots in the hash array
- It is also important how the value of m is chosen
- Values that are powers of 2 should be avoided because if p is the exponent then hash value will be just the p low order bits of the key

Hash functions

- By doing this we restrict our hash function to depend only on the p low order bits of the key
- In most cases the hash value should depend on all the bits of the key
- In practice m is chosen a prime number not too close to the power of 2

Hash functions

- The second method for hashing is the multiplication method
- The hash function has the following form
- h(k) = [m{kA}] where the {} mean fractional part and the [ ] mean the floor and A is a constant with values in the (0,1) open interval and m is the number of slots in the array
- An advantage of this hash function is that it doesn’t depend on the values of m unlike the previous case

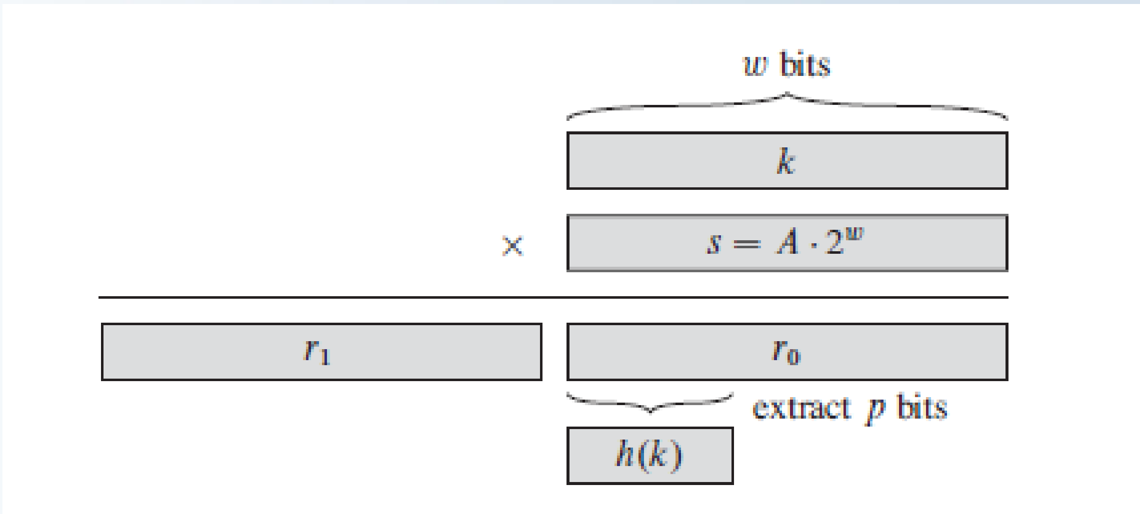
Hash functions

- Another cool thing for this hash function is that it can be implemented on most machines without using operations on floating numbers which are rather slow
- We first assume that the word size of the machine is w bits and that the key k fits in this word size
- Then we define A=s/2^w where 0<s<2^w and s in an integer
- Also m is chosen as a power of 2, say 2^p

Hash functions

- We then multiply k with s and obtain a two word result that can be expressed as r1*2^w+r0 where r1 and r0 are of w bits each
- The hash value will be the p most significant bits of r0 !

Hash functions



Hash functions

- The studies say that a good value for A is

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887\ldots$$

Open addressing

- Another method for treating collisions is open addressing
- In open addressing all elements ocupy the hashtable itself
- We don’t store any additional memory like linked lists which are used in chaining
- An advantage of open addressing is that it spares memory thus offering more empty slots for the same amount of memory as chaining

Open addressing

- By doing this we can obtain fewer collisions because we have more spare slots
- In open addressing when we search for an element we probe table slots until we find the desired element or we are certain that it doesn't exist
- A disadvantage of open addressing is that it can get filled up so the load factor can be at most 1

Open addressing

- The basic idea is that when we insert or search an element we don’t do it in the fixed order 0,1,……m-1 because this would lead to linear time complexity
- The sequence of slots that should be probed is computed using an extended hash function
- The hash function is extended to include the slot number as a second argument

Open addressing

- The hash function is now h:Ux{0,1,…..m-1} - > {0,1, ….m-1}
- An additional constraint is that for every key k the sequence (h(k,0), h(k,1),…….. h(k,m-1)) must be a permutation of (0,1,…… m-1)
- This condition assures that every slot in the hashtable will be eventually visited when the table begins to get filled

Open addressing

Insertion

```
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"
```

Open addressing

- ## Search

```
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
```

Open addressing

- The search function has a problem if we want to also support deletion in this hashtable
- If when we delete an entry from the table we mark it with NIL then search procedure will sometimes fail
- The solution for this would be to mark the node with a different flag and modify the insertion procedure to treat that flag the same as it treats the NIL
- The problem with this solution is that search times will no longer depend on the load factor L

Open addressing

- After a lot of nodes will get deleted the search performance will degrade severely
- This is why open addressing in general is not used when deletions must be performed
- When we have a lot of deletions we should use chaining instead

Open addressing

- The final challenge with open addressing is what kind of hash functions should we use
- For open addressing the ideal hashing is uniform hashing
- In uniform hashing it is assumed that the probe sequence for any key k is equally likely among all the m! permutations of the {0,1……. m-1} sequence
- Uniform hashing is a generalization of simple uniform hashing

Open addressing

- In practice true uniform hashing is very difficult to achieve but some good approximations exist
- Three common techniques for computing the probe sequences will be presented:
  - Linear probing
  - Quadratic probing
  - Double hashing
- None of them achieve more than m^2 different probe sequences !

Open addressing

- Linear probing
- All the methods use auxiliary hash functions
- Let h’:U -> {0,1,……m-1} be the auxiliary hash function
- We define h(k,i)=(h’(k)+i) mod m for i=0,1,….. m-1
- It is easy to see that for each key the probe sequence is a permutation of {0,1,….. m-1}
- The maximum number of different probe sequences is m because the initial hash value of h’ determines the entire probe sequence

Open addressing

- Linear probing is easy to implement but suffers from primary clustering
- Long continuous chains of occupied slots build up and increase the average search time
- The clusters arise because if we have i continuous full slots then slot i+1 will get filled next with (i+1)/m probability

Open addressing

- Quadratic probing
- We define h(k,i)=(h'(k)+c1*i+c2*i^2) mod m where h' an auxiliary one variable hash function like in the previous case
- The maximum number of different probe sequences is still m because the initial slot determines the entire sequence

Open addressing

- In practice it works much better than linear probing but in order to use all the slots the c1, c2 and m variables must be constrained

Open addressing

- Double hashing
- It is the best method of the three
- We define $h(k,i) = (h1(k) + i * h2(k)) \bmod m$
- The value $h2(k)$ must be relatively prime to m in order to be able to search the entire hashtable
- From the formula we can observer that the probe sequence now depends on two parameters that can vary both

Open addressing

- Because of this the number of different probe sequences will be Ө(m^2)
- In order to ensure that h2(k) and m are always relatively prime we can choose m as a power of 2 and ensure that h2(k) is always odd
- A better way to do this is to just choose m as a prime number and design h2 to always output a number that is less than m

Open addressing

- A solution would be:
  - $h1(k) = k \bmod m$
  - $h2(k) = 1 + (k \bmod (m-1))$
- In practice it has been observed that the performance of the double hashing is comparable to that of the ideal uniform hashing

Open addressing

- Time complexity analysis of open addressing
- The analysis is made in terms of the load factor L<1
- It assumes uniform hashing
- The first theorem states that the expected number of probes in an unsuccessful search is 1/(1-L)
- The second theorem states that inserting an element into a hashtable requires 1/(1-L) probes on average

Open addressing

- The third theorem states that the expected number of probes in a successful search is 1/L*ln(1/1-L) with the additional condition that each key in the table is equally likely to be searched for

Problems

- Problems:
  - Given an array of natural numbers find three elements that sum up to the number S
  - Given a string A determine if B is a subsequence of A
  - Given an array of natural numbers and size n find a subset whose sum is divisible by n