Data Structures & Algorithms

Lecture 5: Hash Tables

Chapter 11

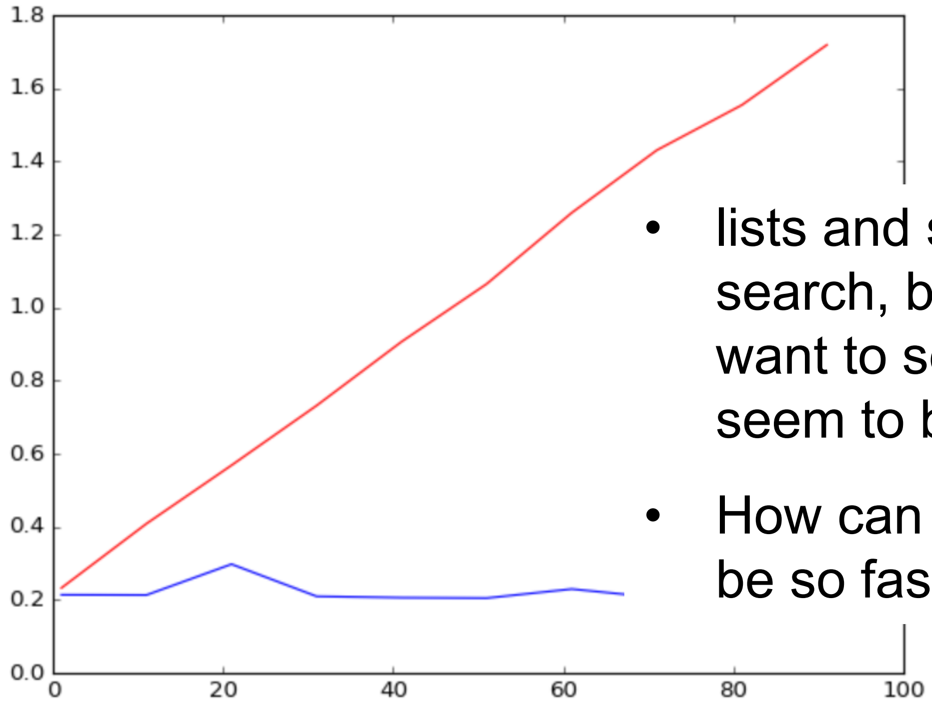
Many slides by Kevin Buchin

Abstract Data Types

From Lecture 1: searching an element

```
In [5]: timeit.timeit(stmt='1 in A', setup='A = list(range(2, 300))')
Out[5]: 4.774117301917343

In [6]: timeit.timeit(stmt='1 in A', setup='A = set(range(2, 300))')
Out[6]: 0.0499541983165841
```



- lists and sets both allow to search, but if we primarily want to search then sets seem to be the better option
- How can searching on sets be so fast?

Abstract data types and Data Structures

Data Structure

a way to store and organize data to facilitate access and modifications.

Ex. array, hash table, … later in the course: linked list, heap, …

Abstract Data Type (ADT)

a set of data values and associated operations that are precisely specified independent of any particular implementation.

Ex. dictionary, … later in the course: stack, queue, priority queue,…

- ADT describe the functionality of data structures
- Data structures implement ADT
  - how is the data stored?
  - which algorithms implement the operations?

Abstract data types and Data Structures

Abstract Data Types

are defined independent of their implementation.

- We can focus on solving the problem instead of the implementation details
- Reduce logical errors by preventing direct access to the implementation
- Implementation can be changed
- We can have multiple, different implementations for the same data type
- Easier to manage and divide larger programs into smaller modules

Dictionary

Dictionary

Stores a set S of elements, each with an associated key (integer value).

Operations

Search(S, k): return a pointer to an element x in S with key[x] = k, or NIL if such an element does not exist.

Insert(S, x): inserts element x into S, that is, $S \leftarrow S \cup \{x\}$

Delete(S, x): remove element x from S

S: personal data

- key: burger service number
- name, date of birth, address, … (satellite data)

Dictionaries in Python

- Dictionaries are available in Python as dict
- keys don’t need to be integers but for instance can also be strings
- dict is implemented using hash tables, which we will look at in detail today

- a set in Python is like a dictionary but the elements consist of the keys only
- keys do not need to be integers

- set is also implemented as hash table

Implementing a dictionary

	Search	Insert	Delete
array*	Θ(n)	Θ(1)	Θ(1)
sorted array	Θ(log n)	Θ(n)	Θ(n)

Today hash tables

* Θ(1) Insert and delete for arrays assumes that we have allocated enough (but not more than O(n)) memory or dynamically allocate memory

Hash Tables

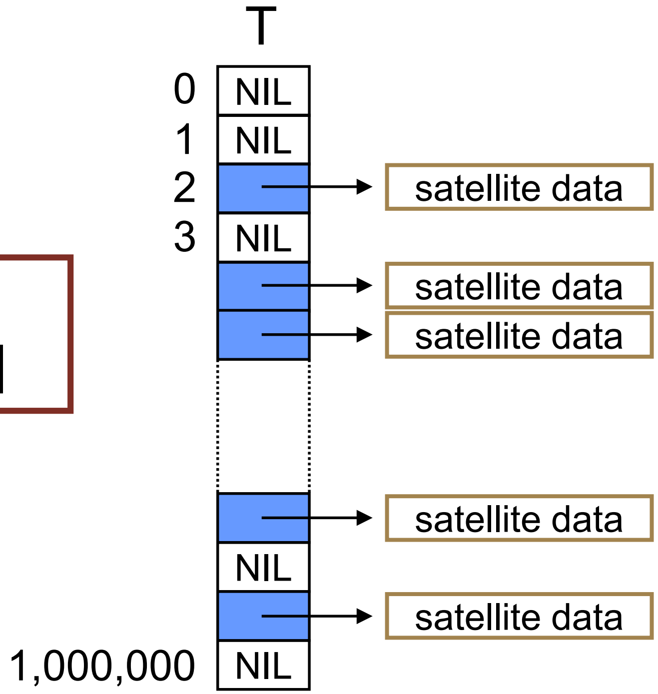
Hash tables

- Hash tables generalize ordinary arrays

Hash tables

- S: personal data in population register
  - key: bsn (burgerservicenummer)
  - name, date of birth, address, … (satellite data)

Assume: bsn-numbers are integers in the range [0 … 1,000,000]



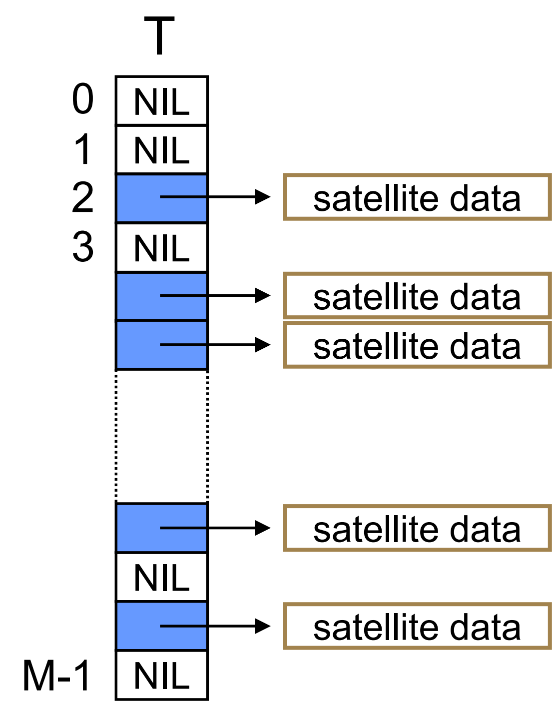
Direct addressing
use table T[0 .. 1,000,000]

Direct-address tables

- S: set of elements
  - key: unique integer from the universe U= {0,…, M-1}
  - satellite data

- use table (array) T[0..M-1]

T[i] = { NIL if there is no element with key i in S; pointer to the satellite data if there is an element with key i in S }



Analysis:

- Search, Insert, Delete: O(1)
- Space requirements: O(M)

Direct-address tables

- S: personal data
  - key: bsn
  - name, date of birth, address, … (satellite data)

Assume: bsn are integers with 9 digits

➡ use table T[0 .. 999,999,999] ?!?

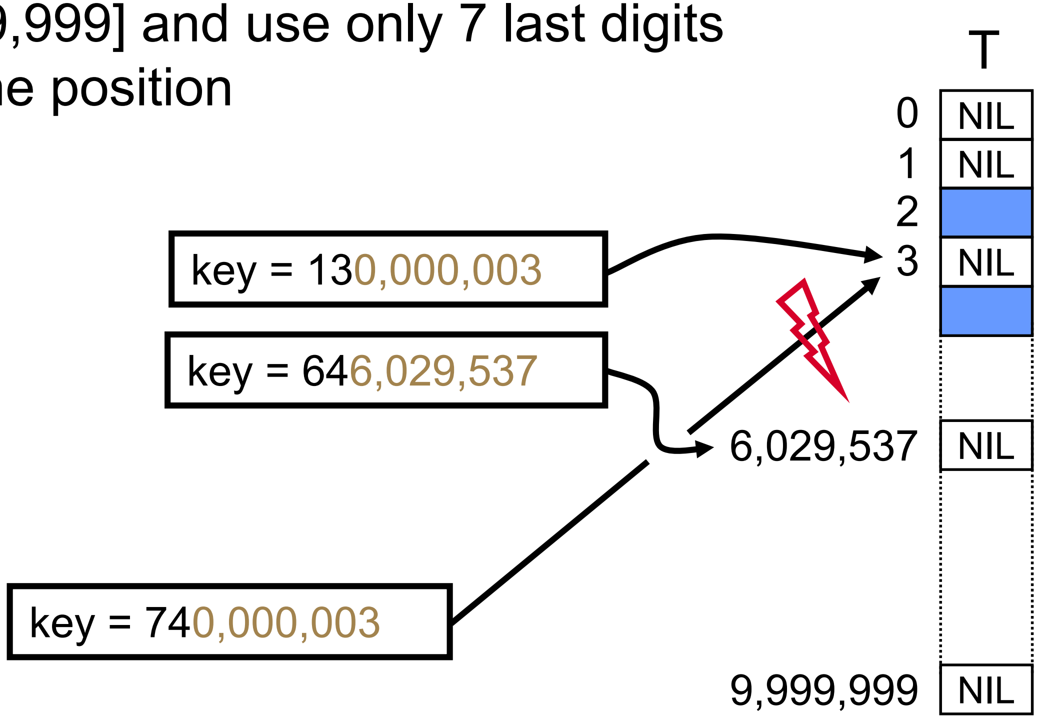
- uses too much memory, most entries will be NIL …

- if the universe U is large, storing a table of size |U| may be impractical or impossible
- often the set K of keys actually stored is small, compared to U
  ➡ most of the space allocated for T is wasted.

Hash tables

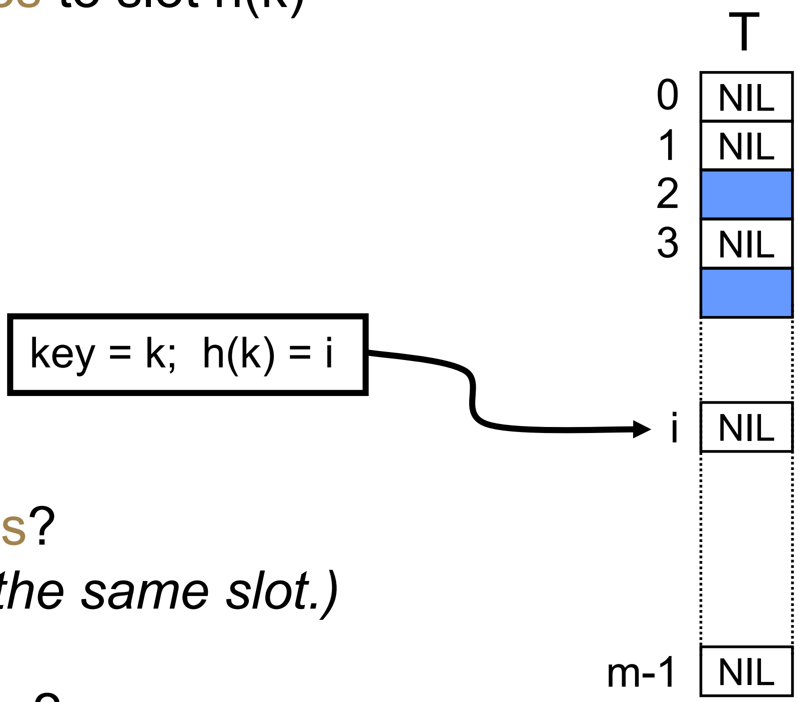
- S: personal data
  - key = bsn = integer from U = {0 .. 999,999,999}

Idea: use a smaller table, for example,
T[0 .. 9,999,999] and use only 7 last digits
to determine position



Hash tables

- S set of keys from the universe U = {0 .. M-1}
  - use a hash table T [0..m-1]   (with m ≤ M)
  - use a hash function h : U → {0 … m-1} to determine the position of each key: key k hashes to slot h(k)



- How do we resolve collisions?
  (Two or more keys hash to the same slot.)

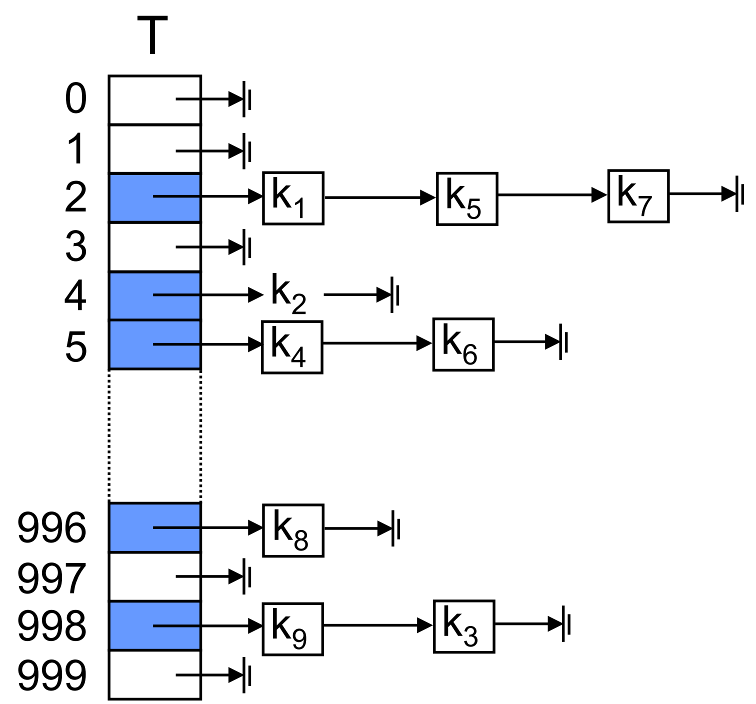
- What is a good hash function?

Resolving collisions: chaining

Chaining: put all elements that hash to the same slot into a linked list

Example (m=1000):

$h(k_1) = h(k_5) = h(k_7) = 2$
$h(k_2) = 4$
$h(k_4) = h(k_6) = 5$
$h(k_8) = 996$
$h(k_9) = h(k_3) = 998$

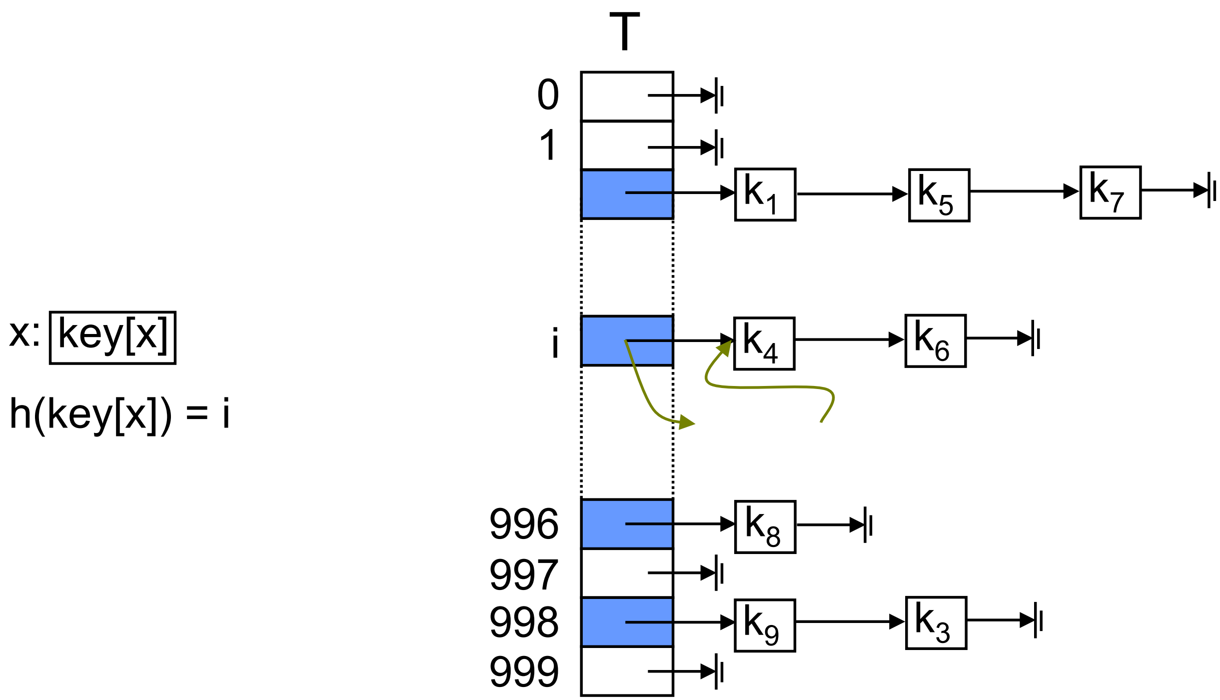


Pointers to the satellite data also need to be included ...

Hashing with chaining: dictionary operations

Chained-Hash-Insert(T,x)
    insert x at the head of the list T[h(key[x])]

Time: O(1)

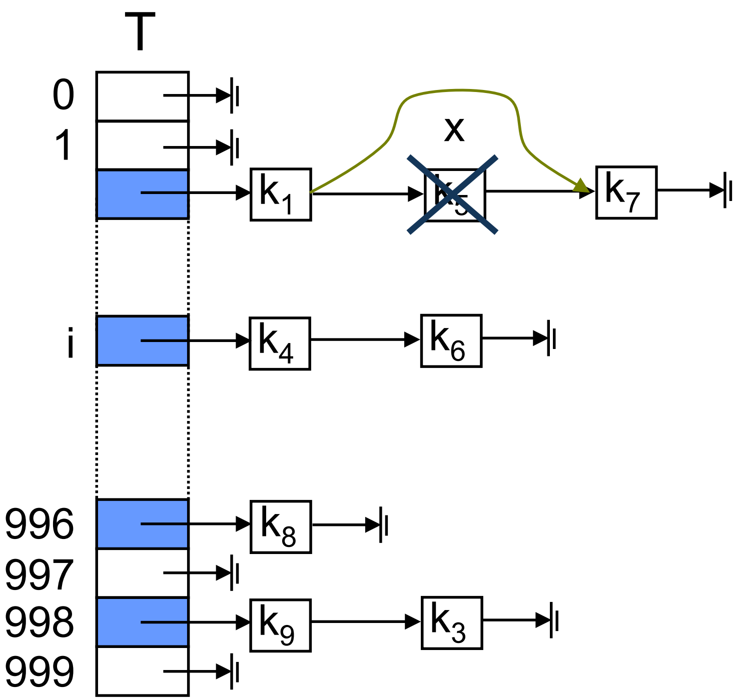


Hashing with chaining: dictionary operations

Chained-Hash-Delete(T,x)
    delete x from the list T[h(key[x])]

- x is a pointer to an element

Time: O(1)
(store pointers to previous and next element, update these pointers for previous and next element)

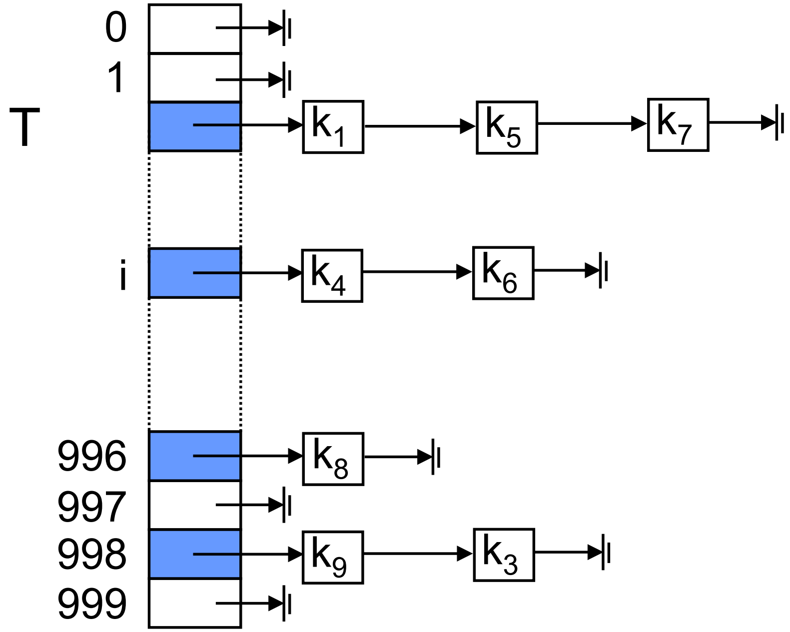


Hashing with chaining: dictionary operations

Chained-Hash-Search(T, k)
search for an element with key k in list T[h(k)]

Time:

- unsuccessful: O(1 + length of T[h(k)])
- successful: O(1 + # elements in T[h(k)] ahead of k)



Hashing with chaining: analysis

Time:

- unsuccessful*:* O(1 + length of T[h(k)])
- successful: O(1 + # elements in T[h(k)] ahead of k)

➡ worst case O(n)

Can we say something about the average case?

Simple uniform hashing

any given element is equally likely to hash into any of the m slots

Hashing with chaining: analysis

Simple uniform hashing

any given element is equally likely to hash into any of the m slots

in other words …

- the hash function distributes the keys from the universe U uniformly over the m slots
- the keys in S, and the keys with whom we are searching, behave as if they were randomly chosen from U

➡ we can analyze the average time it takes to search as a function of the load factor $\alpha = n/m$

(m: size of table, n: total number of elements stored)

Hashing with chaining: analysis

Theorem

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

Proof (for an arbitrary key)

- the key we are looking for hashes to each of the m slots with equal probability
- the average search time corresponds to the average list length
- average list length = total number of keys / # lists = α

■

- The $\Theta(1+\alpha)$ bound also holds for a successful search (although there is a greater chance that the key is part of a long list).
- If $m = \Omega(n)$, then a search takes $\Theta(1)$ time on average.

What is a good hash function?

What is a good hash function?

1. as random as possible
   get as close as possible to simple uniform hashing …

   - the hash function distributes the keys from the universe U uniformly over the m slots
   - the hash function has to be as independent as possible from patterns that might occur in the input

2. fast to compute

What is a good hash function?

Example: hashing performed by a compiler for the symbol table

- keys: variable names which consist of (capital and small) letters and numbers: i, i2, i3, Temp1, Temp2, …

Idea:

- use table of size $(26+26+10)^2$
- hash variable name according to the first two letters: Temp1 → Te

Bad idea: too many "clusters"
(names that start with the same two letters)

What is a good hash function?

Assume: keys are natural numbers

if necessary first map the keys to natural numbers

"aap" → $\boxed{1000001}\ \boxed{1000001}\ \boxed{1011010}$ → map bit string to natural number

$\boxed{\textit{ascii representation}}$

➡ the hash function is h: **N** → {0, …, m-1}

- the hash function always has to depend on all digits of the input

Common hash functions

Division method: $h(k) = k \bmod m$

Example: $m=1024$, $k = 2058$ ➡ $h(k) = 10$

- don't use a power of 2
  $m = 2^p$ ➡ $h(k)$ depends only on the p least significant bits
- use m = prime number, not near any power of two

Multiplication method: $h(k) = \lfloor m \boxed{(kA \bmod 1)} \rfloor$

1. $0 < A < 1$ is a constant
2. compute kA and extract the fractional part
3. multiply this value with m and then take the floor of the result

- Advantage: choice of m is not so important,
  can choose m = power of 2

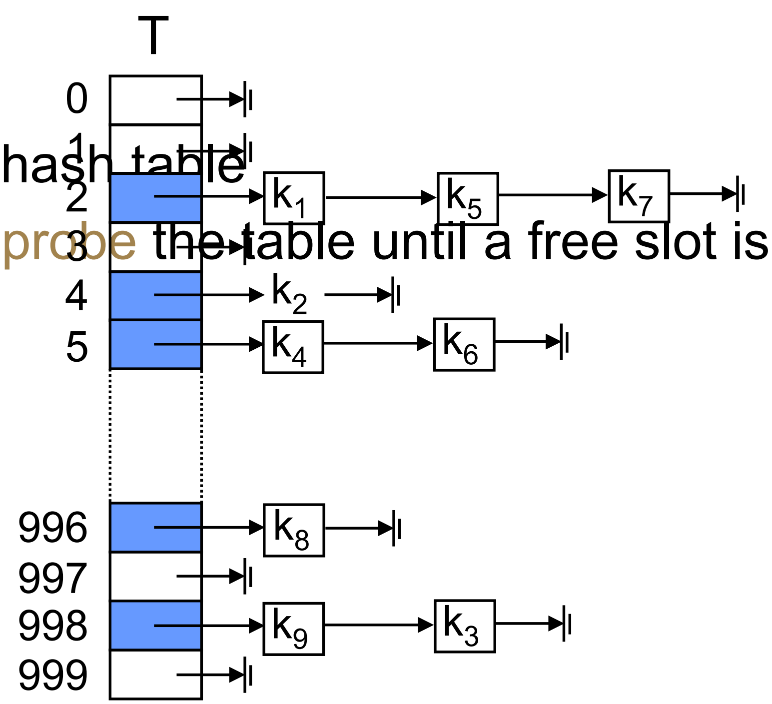
Resolving collisions

more options …

Resolving collisions

Resolving collisions

1. Chaining: put all elements that hash to the same slot into a linked list
2. Open addressing:
   - store all elements in the hash table
   - when a collision occurs, probe the table until a free slot is found



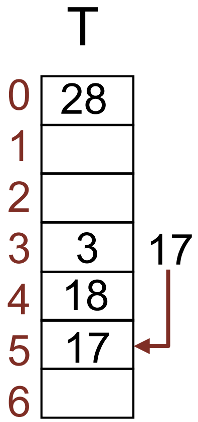
Hashing with open addressing

Open addressing:

- store all elements in the hash table
- when a collision occurs, probe the table until a free slot is found

Example: T[0..6] and h(k) = k mod 7

1. insert 3
2. insert 18
3. insert 28
4. insert 17



- no extra storage for pointers necessary
- the hash table can “fill up”
- the load factor α is always ≤ 1

Hashing with open addressing

- there are several variations on open addressing depending on how we search for an open slot

- the hash function has two arguments:
  the key and the number of the current probe

  ➡ probe sequence ‹h(k,0), h(k, 1), … h(k, m-1)›

The probe sequence has to be a permutation of ‹0, 1, … ,m-1› for every key k.

Open addressing: dictionary operations

```
Hash-Insert(T, k)
i = 0
while (i < m) and (T[ h(k,i) ] ≠ NIL )
      do i = i +1
if i < m
      then T [h(k,i)] = k
      else "hash table overflow"
```

we're actually inserting element x with key[x] = k

Example: Linear Probing

- T[0..m-1]
- h'(k) ordinary hash function
- h(k,i) = (h'(k) + i) mod m
- Hash-Insert(T,17)

T

0	28	
1		
2		
3	3	17
4	18	17
5	17	17
6		

Open addressing: dictionary operations

Hash-Search(T,k)

```
i = 0
while  (i < m)  and  (T [ h(k,i) ] ≠ NIL)
    do  if  T[ h(k,i) ] = k
            then return "k is stored in slot h(k,i)"
            else  i = i +1
return  "k is not stored in the table"
```

Example: Linear Probing

- h'(k) = k mod 7
  h(k,i) = (h'(k) + i) mod m
- Hash-Search(T,17)

	T	
0	28	
1		
2		
3	3	17
4	18	17
5	17	17
6		

Open addressing: dictionary operations

Hash-Search(T,k)

```
i = 0
while  (i < m)  and  (T [ h(k,i) ] ≠ NIL)
     do  if  T[ h(k,i) ] = k
              then return "k is stored in slot h(k,i)"
              else  i = i +1
return  "k is not stored in the table"
```

Example: Linear Probing

- $h'(k) = k \bmod 7$
  $h(k,i) = (h'(k) + i) \bmod m$
- Hash-Search(T,17)
- Hash-Search(T,25)

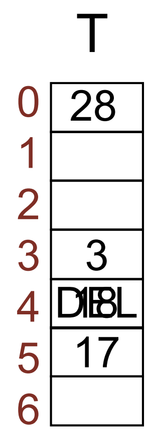
	T	
0	28	
1		
2		
3	3	
4	18	25
5	17	25
6		25

Open addressing: dictionary operations

Hash-Delete(T,k)

1. remove k from its slot
2. mark the slot with the special value DEL

Example: delete 18



- Hash-Search passes over DEL values when searching
- Hash-Insert treats a slot marked DEL as empty
  - ➡ search times no longer depend on load factor
  - ➡ use chaining when keys must be deleted

Open addressing: probe sequences

- $h'(k)$ = ordinary hash function

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

- $h'(k_1) = h'(k_2)$ ➡ k_1 and k_2 have the same probe sequence
- the initial probe determines the entire sequence
  ➡ there are only m distinct probe sequences
- all keys that test the same slot follow the same sequence afterwards

- Linear probing suffers from primary clustering: long runs of occupied slots build up and tend to get longer
  ➡ the average search time increases

Open addressing: probe sequences

- $h'(k)$ = ordinary hash function

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

- $h'(k_1) = h'(k_2)$ ➡ k_1 and k_2 have the same probe sequence
- the initial probe determines the entire sequence
  ➡ there are only m distinct probe sequences
- but keys that test the same slot do not necessarily follow the same sequence afterwards

- quadratic probing suffers from secondary clustering: if two distinct keys have the same h' value, then they have the same probe sequence

Note: c_1, c_2, and m have to be chosen carefully, to ensure that the whole table is tested.

Open addressing: probe sequences

- $h'(k)$ = ordinary hash function

Double hashing: $h(k,i) = (h'(k) + i\, h''(k)) \bmod m$,
$h''(k)$ is a second hash function

- keys that test the same slot do not necessarily follow the same sequence afterwards
- h'' must be relatively prime to m to ensure that the whole table is tested.
- $O(m^2)$ different probe sequences

Open addressing: analysis

Uniform hashing

each key is equally likely to have any of the m! permutations of ‹0, 1, …, m-1› as its probe sequence

Assume: load factor $\alpha = n/m < 1$, no deletions

Theorem

The average number of probes is

- $\Theta(1/(1-\alpha))$ for an unsuccessful search
- $\Theta((1/\alpha) \log (1/(1-\alpha)))$ for a successful search

Open addressing: analysis

Theorem

The average number of probes is

- $\Theta(1/(1-\alpha))$ for an unsuccessful search
- $\Theta((1/\alpha)\log(1/(1-\alpha)))$ for a successful search

Proof: E [#probes] $= \sum_{1\le i\le n} i \cdot \Pr[\#\text{ probes} = i]$

$= \sum_{1\le i\le n} \Pr[\#\text{ probes} \ge i]$

Pr [#probes ≥ i] $= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdot \cdots \cdot \frac{n-i+2}{m-i+2}$

$\le \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$

E [#probes] $\le \sum_{1\le i\le n} \alpha^{i-1} \le \sum_{0\le i\le\infty} \alpha^{i} = \frac{1}{1-\alpha}$ ■

Check the CLRS book for details!

Hash tables

- Hash tables generalize ordinary arrays
  - map a large universe to a small table

- How do we resolve collisions?
  - Chaining
  - Open addressing: linear and quadratic probing, double hashing

- What is a good hash function?
  - Division method
  - Multiplication method

Implementing a dictionary

	Search	Insert	Delete
array	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sorted array	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
hash table	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

- Running times are average times and assume (simple) uniform hashing and a large enough table (for example, of size 2n). Also inserting/deleting in the array will require resizing.

Drawbacks of hash tables: operations such as finding the min or the successor of an element are inefficient.