2IL50 Data Structures

2023-24 Q3

Lecture 6: Hash Tables

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Abstract Data Types

Abstract data type

Abstract Data Type (ADT)

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

Dictionary, stack, queue, priority queue, set, bag ...

Priority queue

Max-priority queue

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

Operations

Insert(S, x):	inserts element x into S, that is, $S \leftarrow S \cup \{x\}$
Maximum(S):	returns the element of S with the largest key
Extract-Max(S):	removes and returns the element of S with the largest key
Increase-Key(S, x, k):	gives $x.\text{key}$ the value k condition: k is larger than the current value of $x.\text{key}$

Implementing a priority queue

	Insert	Maximum	Extract-Max	Increase-Key
sorted list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
sorted array	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$?	$\Theta(n)$
heap	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

Dictionary

Dictionary
: stores a set S of elements, each with an associated key (integer value)

Operations

Search(S, k): returns a pointer to an element x in S with $x.\mathrm{key} = k$,
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): removes element x from S

S: personal data

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

Implementing a dictionary

	Search	Insert	Delete
linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sorted array	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$

Today	hash tables
Next lecture	*(balanced) binary search trees*

Hash Tables

Hash tables

Hash tables generalize ordinary arrays

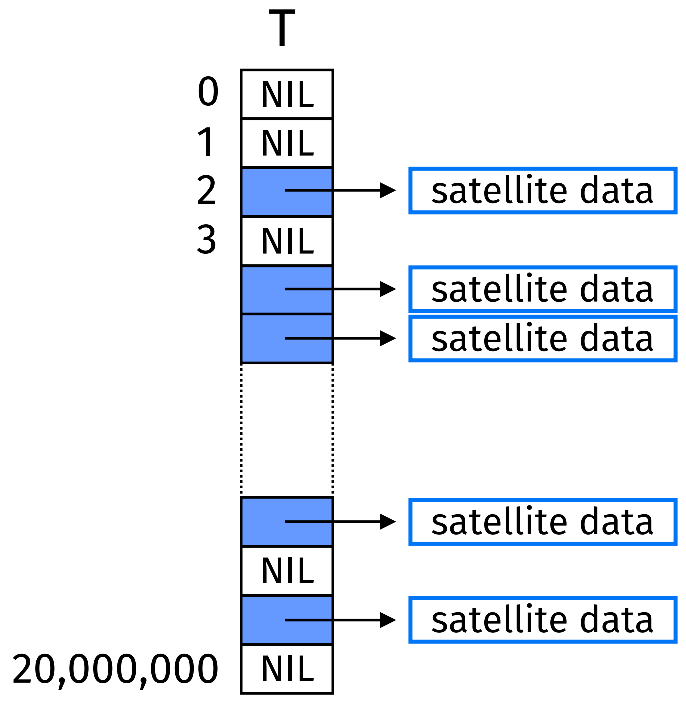
Hash tables

S: personal data

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

Assume: bsn are integers in the range [0 … 20,000,000]

Direct addressing
use table $T[0: 20{,}000{,}000]$



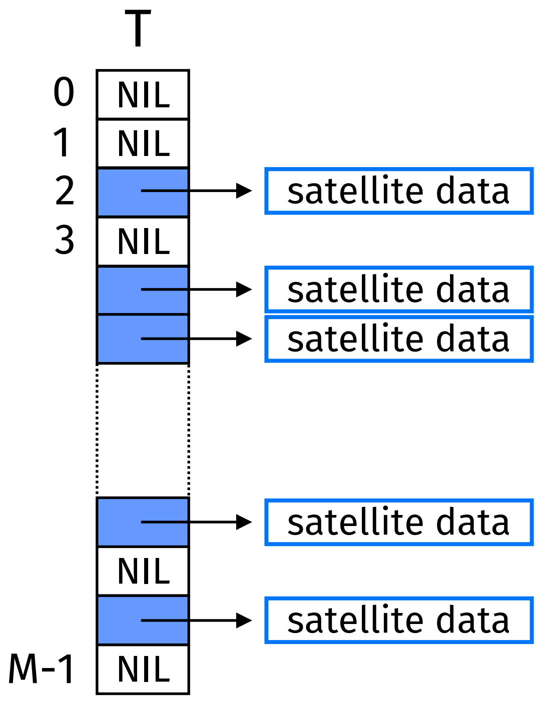
Direct-address tables

S: set of elements

- key: unique integer from the universe $U = \{0, \ldots, M-1\}$
- satellite data

use table (array) $T[0\colon M-1]$

$$T[i] = \begin{cases} NIL \text{ if there is no element with key } i \text{ in } S \\ \text{pointer to the satellite data if there is an element with key } i \text{ in } S \end{cases}$$



Analysis:

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

Direct-address tables

S: personal data

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

Assume: bsn are integers with 10 digits

use table $T[0: 9{,}999{,}999{,}999]$?!?

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

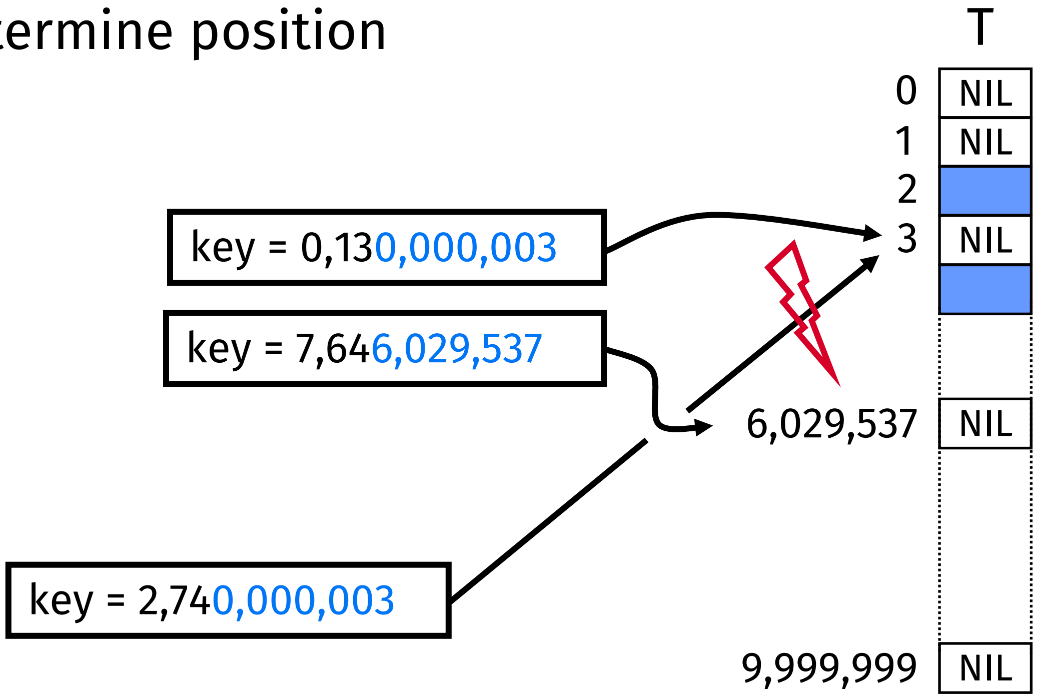
often the set K of keys actually stored is small, compared to U
➡ most of the space allocated for T is wasted.

Hash tables

S: personal data

- key $=$ bsn $=$ integer from $U = \{0, \ldots, 9{,}999{,}999{,}999\}$

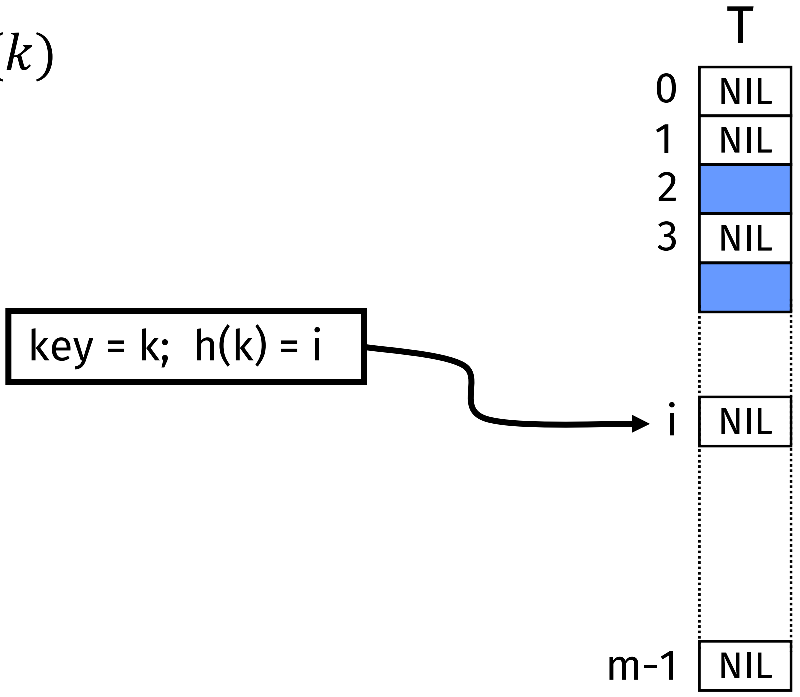
Idea: use a smaller table, for example, $T[0{:}\,9{,}999{,}999]$
and use only 7 last digits to determine position



Hash tables

S set of keys from the universe $U = \{0, \ldots, M-1\}$

- use a hash table $T\,[0 : m-1]$ (with $m \leq M$)
- use a hash function $h : U \to \{0, \ldots, m-1\}$ 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.)

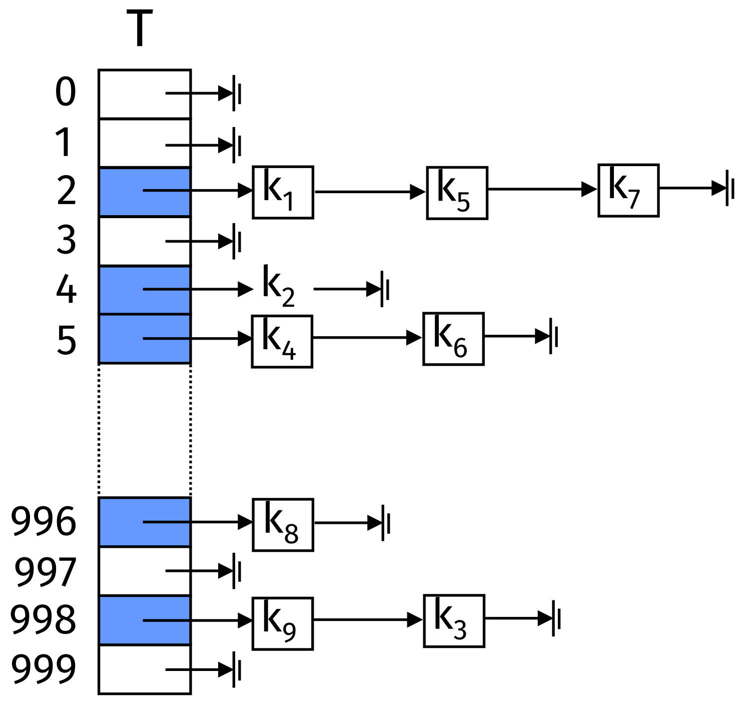
What is a good hash function?

Resolving collisions: chaining

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

Example ($m = 1000$):

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

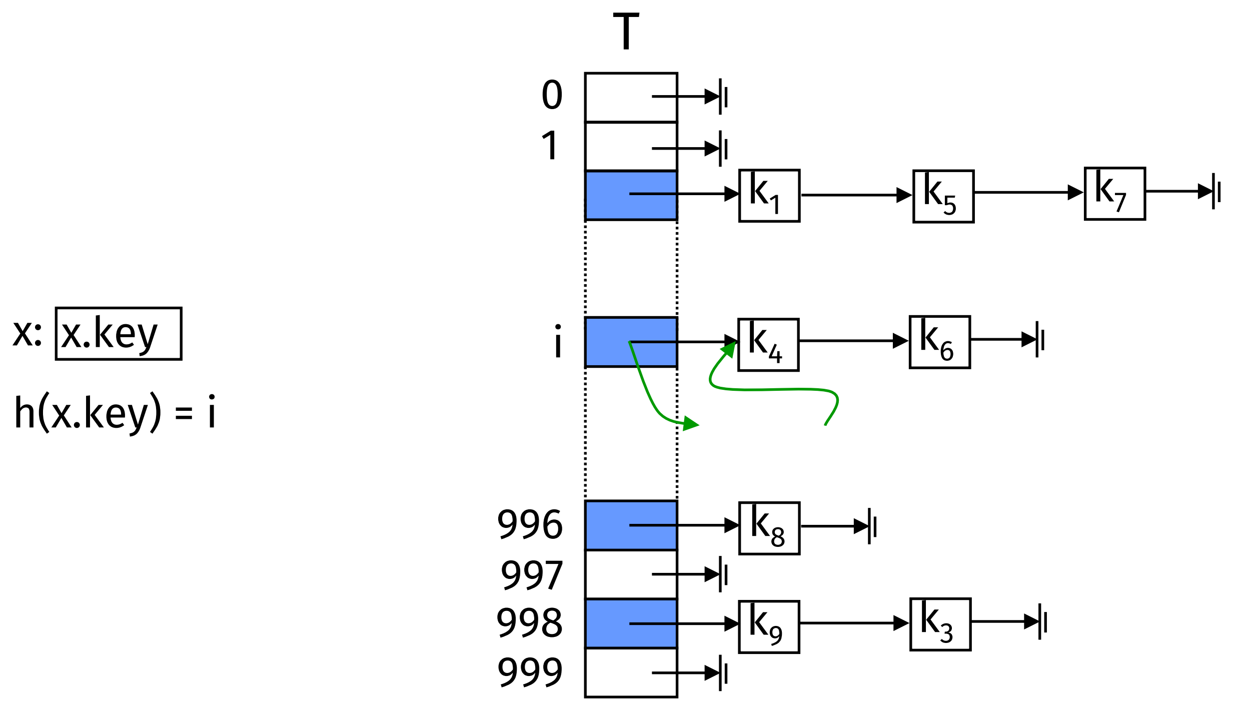


Pointers to the satellite data also need to be included ...

Hashing with chaining: dictionary operations

Chained-Hash-Insert(T, x)
    insert x at the head of the list $T[h(x.\text{key})]$

Time: $O(1)$



Hashing with chaining: dictionary operations

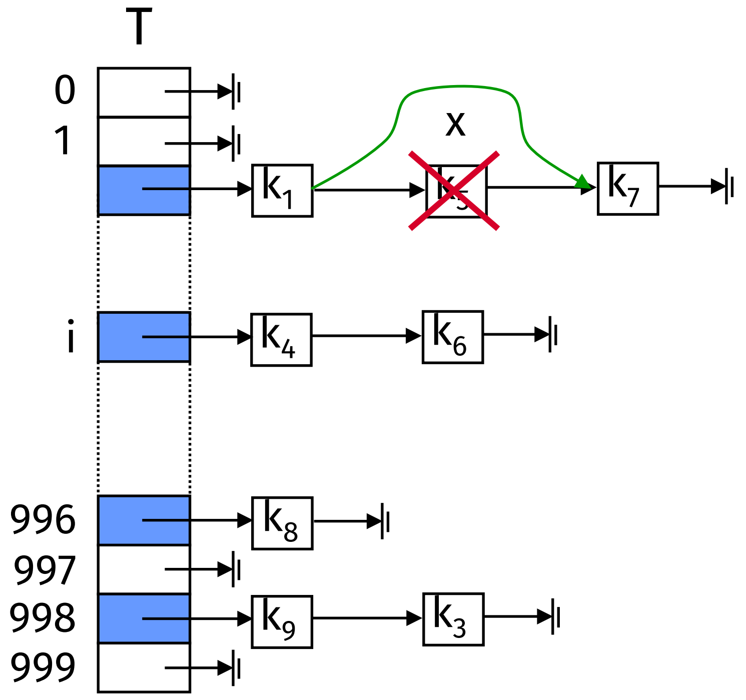
```
Chained-Hash-Delete($T$, $x$)
    delete $x$ from the list $T[h(x.\text{key})]$
```

---

x is a pointer to an element

Time: $O(1)$

(with doubly-linked lists)

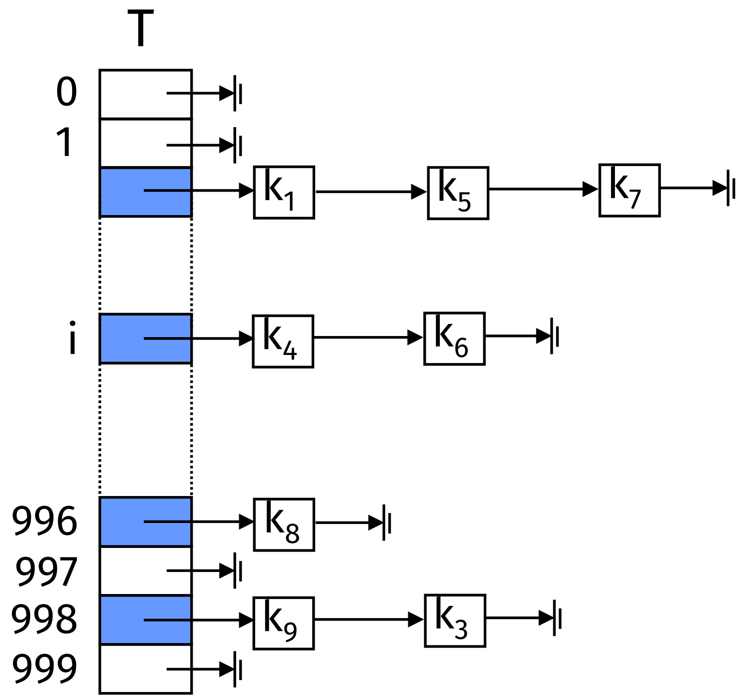


Hashing with chaining: dictionary operations

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

Time:

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



Hashing with chaining: analysis

Time:

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

worst case $O(n)$

Can we say something about the average case?

Hashing with chaining: analysis

Independent uniform hashing

- any given element is equally likely to hash into any of the m slots (uniform)
- the slot to which an element hashes is independent of other keys

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 collisions are resolved by chaining,
an unsuccessful search takes time $\Theta(1 + \alpha)$, on average,
under the assumption of independent 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.

Hash functions

What is a good hash function?

1. as random as possible
   get as close as possible to independent 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 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 $h\colon \boldsymbol{N} \rightarrow \{0, \dots, m-1\}$

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

Common hash functions

Division method: $h(k) = k \; mod \; m$

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

- don’t use a power of 2
  $m = 2^p \qquad h(k)$ 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

Analysis assumptions

Independent uniform hashing

- any given element is equally likely to hash into any of the m slots (uniform)
- the slot to which an element hashes is independent of other keys

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 $\boxed{\text{randomly chosen from } U}$

Depends on assumptions on the input!

Static versus random

Static hashing Use a single pre-defined hash function

Uniformity and independence rely on randomness of the input

Has “bad” inputs

Recall Quicksort

Using last element as pivot requires randomness in input for efficiency (average case)

Solution: Quicksort with random pivot moves randomness into algorithm
➡ no more “bad” inputs.

Random hashing

Random hashing Select a hash function at random from a family of hash functions $H = \{h_1, h_2, h_3, \dots, h_k\}$

Hash function is selected when initializing the table.

Why at initialization? *See practice set*

Independent uniform hashing is (theoretically) possible with random hashing

- H is the family of all possible hash functions
- But there are m^M possible hash functions (universe of size M) ➡ not practical

Universal hashing

Random hashing	Select a hash function at random from a family of hash functions $H = \{h_1, h_2, h_3, \dots, h_k\}$
Universal hashing	A finite family of hash functions H is universal if for each pair of distinct keys $k_1, \mathrm{k}_2 \in U$ the number of hash function $h \in H$ for which $h(k_1) = h(k_2)$ is most $|H|/m$.

Universal hashing has provably good performance on average

Theorem

Using universal hashing and collision resolution by chaining in an initially empty table with m slots, it takes $\Theta(s)$ expected time to handle any sequence of s insert, search and delete operations containing $n = O(m)$ insert operations.

Universal hashing

Input: Universe U of integers 0 up to $M-1$ and table size m

Choose prime p with $p > M-1$

Define H as a family of hash functions h_{ab} for all $a \in \{1 \ldots p-1\}$ and $b \in \{0 \ldots p-1\}$ with

$h_{ab}(k) = ((ak + b) \bmod p) \bmod m$

Theorem: This family H is universal

Proof:

… *see book*

BCS 1st year: Hashing will be covered in Probability & Statistics in March

Resolving collisions

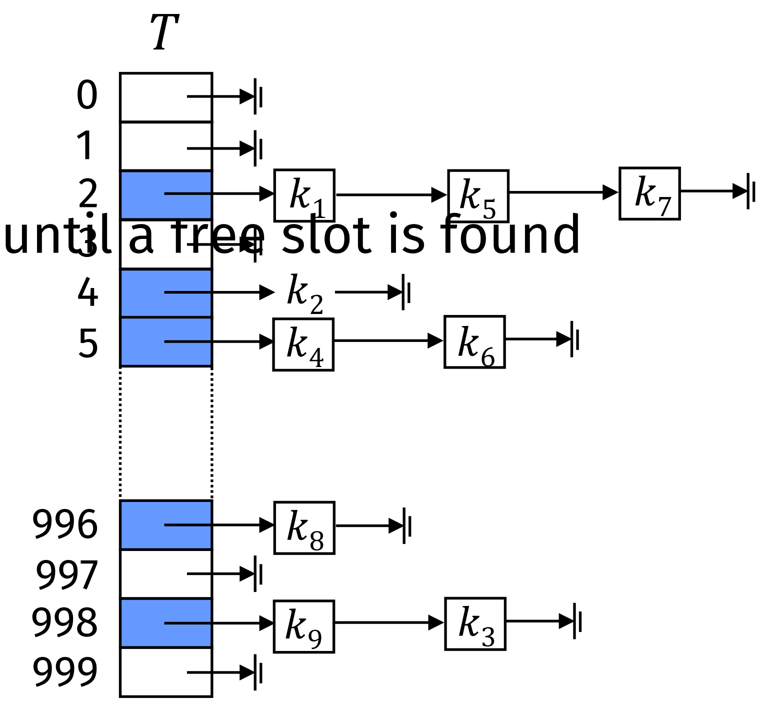
more options …

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



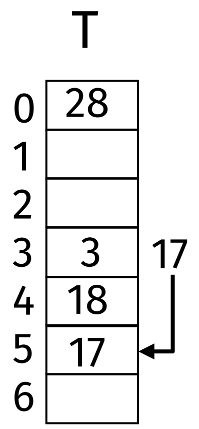
Hashing with open addressing

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 $h(k) = k \bmod 7$

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

The probe sequence has to be a permutation of $\langle 0, 1, \dots\ , m-1 \rangle$ for every key k.

Open addressing: dictionary operations

```
Hash-Insert($T$, $k$)
1  $i = 0$
2  while $i < m$ and $T[h(k,i)] \neq NIL$
3          $i = i + 1$
4  if $i < m$
5          $T[h(k,i)] = k$
6  else error “hash table overflow”
```

we’re actually inserting element x with $x.\mathrm{key} = k$

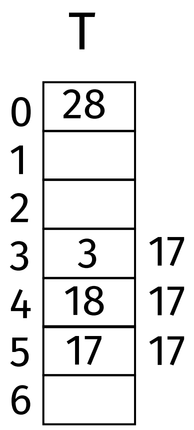
Example: Linear Probing

$T[0{:}\,m-1]$

$h'(k)$ ordinary hash function

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

Hash-Insert(T, 17)



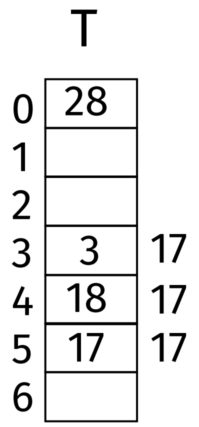
Open addressing: dictionary operations

```
Hash-Search($T$, $k$)
$i = 0$
while $i < m$ and $T[h(k,i)] \neq NIL$
    if $T[h(k,i)] == k$
        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)



Open addressing: dictionary operations

Hash-Search(T, k)

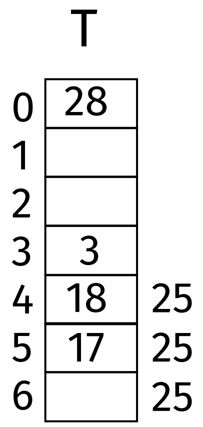
```
1  $i = 0$
2  while $i < m$ and $T[h(k,i)] \neq NIL$
3          if $T[h(k,i)] == k$
4                  return "$k$ is stored in slot $h(k,i)$"
5          else
6                  $i = i + 1$
7  return "$k$ is not stored in the table"
```

Example: Linear Probing

$$h'(k) = k \bmod 7$$
$$h(k,i) = (h'(k) + i) \bmod m$$

Hash-Search(T, 17)

Hash-Search(T, 25)



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

T

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

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

T

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

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
    - Note: there are $m!$ probe sequences…

Open addressing: analysis

Uniform hashing

each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1 \rangle$ 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: $\mathrm{E}[\#probes] = \sum_{i=1}^{m} i \cdot \Pr[\#\text{probes} = i]$

$= \sum_{i=1}^{m} \Pr[\#\text{probes} \geq i]$

$$\Pr[\#probes \geq i] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdot \ldots \cdot \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$$

$$E[\#probes] \leq \sum_{i=1}^{m} \alpha^{i-1} \leq \sum_{i=1}^{\infty} \alpha^{i} = \frac{1}{1-\alpha}$$

Check the book for details!

BCS 1st year: Hashing will be covered in Probability & Statistics in March

Implementing a dictionary

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

Running times are average times and assume independent uniform hashing and a large enough table (for example, of size $2n$)

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