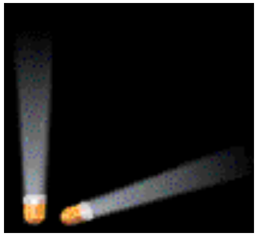
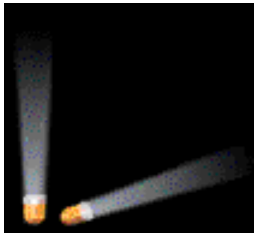
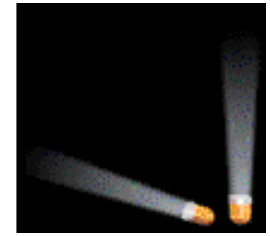
Hashing



- **The Dictionary**
  - a dictionary (table) is an abstract model of a database
  - a dictionary stores key-element pairs
  - the main operation supported by a dictionary is searching by key

Hashing





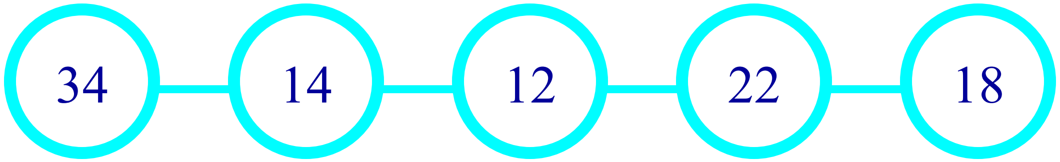
- Applications
  - Telephone directory
  - Library catalogue
  - Books in print: key ISBN
  - FAT (File Allocation Table)

ADT

```
template <class K, class E>
class Dictionary {
public:
        virtual bool IsEmpty () const = 0;
        virtual pair<K,E>* Get(const K&) const = 0;
        virtual void Insert(const pair<K,E>&) = 0;
        virtual void Delete(const K&) = 0;
};
```

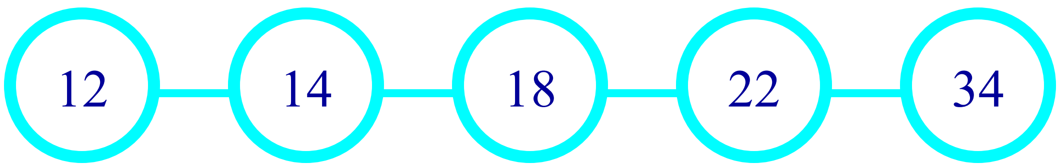
Implementing a Dictionary with a Sequence

- ***unordered sequence***
  - searching and removing takes O(n) time
  - inserting takes O(1) time
  - applications to log files (frequent insertions, rare searches and removals) 34 14 12 22 18



Implementing a Dictionary with a Sequence

- ***array-based ordered sequence***
  (assumes keys can be ordered)
- - searching takes O(log n) time ***(binary search*h)**
  - inserting and removing takes O(n) time
  - application to look-up tables
  (frequent searches, rare insertions and removals)



Other Implementations?

- Binary search tree
  - O(h) → O(n)
- Balanced search trees
  - O(logN)
- Key comparison based
- Can we do better?

Application

- China Telecom is a large phone company, and they want to provide enhanced caller ID capability:
  - given a phone number, return the caller's name
  - phone numbers are in the range 0 to $R = 10^{10}-1$
  - **n** is the number of phone numbers used
  - want to do this as efficiently as possible

Bucket Array

- Each cell is thought of as a bucket or a container
  - Holds key element pairs
  - In array A of size N, an element *e* with key *k* is inserted in *A[k]*.

(null)	(null)	…	Roberto	…	(null)

0000000000 4109321568 9999999999

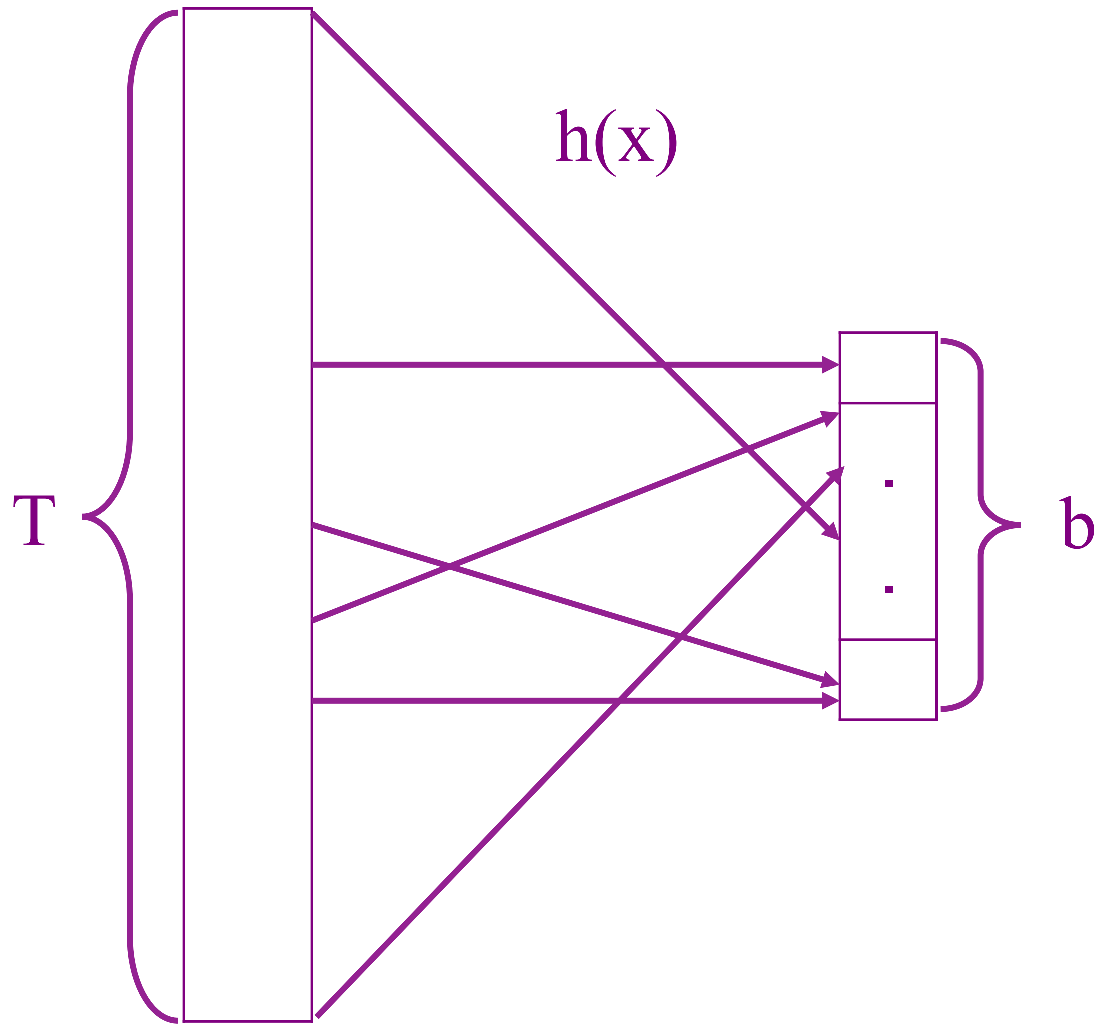
  - ***A bucket array*** indexed by the phone number has optimal O(1) query time
  - There is a huge amount of wasted space

Bucket Array

- A data structure
- The location of an item is determined by:
  - directly as a function of the item itself: f(key)=key
  - Not by a sequence of trial and error comparisons
- Commonly used to provide faster searching
  - $O(n)$ for linear searches
  - $O(\log n)$ for binary search
  - $O(1)$ for hash table

Space Solution

- ***A Hash Table*** is an alternative solution with O(1) expected query time and O(n + N) space, where N is the size of the table
- Like an array, but with a function to map the large range of keys into a smaller one
  - e.g., take the original key, ***mod*** the size of the table, and use that as an index


h(x)
T
b

Example

- Insert item (401-863-7639, Roberto) into a table of size 5
- 4018637639 mod 5 = 4, so item (401-863-7639, Roberto) is stored in slot 4 of the table
- A lookup uses the same process: map the key to an index, then check the array cell at that index

				401- 863-7639 Roberto
0	1	2	3	4

Static hashing

- dictionary pairs are stored in a table, **ht**, called hash table
- **ht** is partitioned into **b** buckets: ht[0:b-1]
- **ht** is maintained in sequential memory
- each bucket holds **s** slots, each slot holds one pair, usually, $s = 1$
- the address of a pair with key k is determined by a hash function h, h(k) is the hash or home address of k, $h(k) \in \{0, 1, \ldots, b-1\}$

Notations

T --- the total number of possible keys.

n --- the number of pairs in the hash table.

Definition:

The **key density** of a hash table is the ratio **n/T**.

The **loading density (or factor)** of a hash table is $\alpha = n/(s \times b)$.

Usually, $n \ll T$, and $b \ll T$.

Notations

- 2 keys k_1 and k_2 are said to be **synonyms** with respect to h if **h(k1) = h(k2).**
- a **collision** occurs when the home bucket for the new pair is not empty.
- an **overflow** occurs when a new pair is hashed into a full bucket.
- when s=1, collisions and overflows occur simultaneously.

An Example

- b=26, s=2, n=10, hence α = ?
  - 10/52 = 0.19
- Keys: GA, D, A, G, L, A2, A1, A3, A4, E
- h(k) = the first character of k
  - A to Z corresponds to 0 to 25

- GA, D, A, G, L entered
- A2 entered
  - Collision
- A1 entered
  - Collision
  - Overflow

ht	Slot 1	Slot 2
0	A	A2
1		
2		
3	D	
4		
5		
6	GA	G
.	.	.
.	.	.
25		

Analysis

- No overflow
- Performance of insert, delete, search
  - Hash function
  - Searching within a bucket
  - Independent of **n**
- However,Overflow is happening
  - T >> b

From Keys to Indices

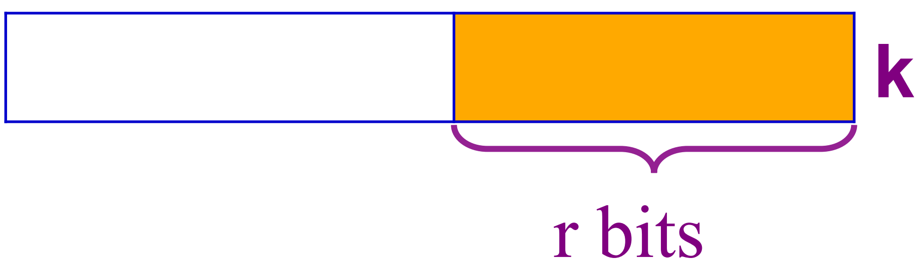
- The mapping of keys to indices of a hash table is called a **hash function**
- A hash function is usually the composition of two maps:
  - **hash code map**: key ◊ integer
  - **compression map**: integer ◊ [0, N - 1]

Hash function

- Essential requirement of the hash function
  - map **equal keys to equal indices**
- A “good” hash function
  - minimizes the **probability of collisions**
  - **Easy to compute**
- uniform hash function
  - If k is a key chosen at random from the key space, then the probability that h(k)=i to be 1/b for all buckets i

Hash function
compression map

- Division
  - $h(k) = |k| \ \mathbf{mod}\ N$
  - ***Selection of N is critical***
  - $N = 2^r$ is bad because not all the bits are taken into account



  - the table size N is usually chosen as a **prime number**

Hash function

hash code **map**

- Mid-square
  - h(k) is computed by using an appropriate number of bits from the middle of k^2 to obtain the bucket address.

If r bits used, $b= 2^r$

Hash function

hash code **map**

- Folding
  - k is partitioned into several parts, all but the last being of the same length
  - These partitions are then added together to obtain the hash address for k.

k=12320324111220 is partitioned into parts that are 3 decimal digits long.

$P_1=123$, $P_2=203$, $P_3=241$, $P_4=112$, $P_5=20$.

- shift folding

  $h(k)=123+203+241+112+20=699$

- folding at the boundaries

  $h(k)=123+302+241+211+20=897$

Hash function
hash code map

- Digit Analysis
  - each k is interpreted as a number using some radix r
  - the digits of each k are examined
    - digits having the most skewed distribution are deleted
    - until the number of digits left is small enough to give an address

Hash function

hash code map

- Converting Keys to integers
  - for strings of a natural language, combine the character values (ASCII or Unicode) $a_0 a_1 \ldots a_{n-1}$ by viewing them as the coefficients of a polynomial: $a_0 + a_1 x + \ldots + x_{n-1} a_{n-1}$

Overflow handling

- A key is mapped to an already occupied table location
  - what to do?!?
- Use a collision handling technique
  - ***Open Addressing***
    - ***Linear Probing***
    - ***Quadratic probing***
    - ***Double Hashing***
  - ***Chaining***

Linear Probing

- **$h_i(K)$ = (hash(K) + i) mod m**
- Insertion:
  - Let K be the new key to be inserted, compute hash(K)
  - For i = 0 to m-1
    - compute L = (hash(K) + I) mod m
    - T[L] is empty, then we put K there and stop.
  - If we cannot find an empty entry to put K, it means that the table is full and we should report an error.

Quadratic Probing

- **$h_i(K) = (hash(K) + i^2) \bmod m$**
- Insertion:
  - Let K be the new key to be inserted, compute hash(K)
  - For i = 0 to m-1
    - compute $L = (hash(K) + i^2) \bmod m$
    - T[L] is empty, then we put K there and stop.
  - If we cannot find an empty entry to put K, it means that the table is full and we should report an error.

Double Hashing

- Hash1(), Hash2(), ……,HashN()

An Open Hash Table

Hash (key) produces an index in the range 0 to 6. That index is the “home address”

Some insertions:
K1 --> 3
K2 --> 5
K3 --> 2

	key	value
0		
1		
2	K3	K3info
3	K1	K1info
4		
5	K2	K2info
6		

Handling Collisions

Some more insertions:
K4 --> 3
K5 --> 2
K6 --> 4

0	K6	K6info
1		
2	K3	K3info
3	K1	K1info
4	K4	K4info
5	K2	K2info
6	K5	K5info

Linear probing collision resolution strategy

Search Performance

0	K6	K6info
1		
2	K3	K3info
3	K1	K1info
4	K4	K4info
5	K2	K2info
6	K5	K5info

Average number of probes needed to retrieve the value with key K?

K	hash(K)	#probes
K1	3	1
K2	5	1
K3	2	1
K4	3	2
K5	2	5
K6	4	4

14/6 = 2.33 (successful)

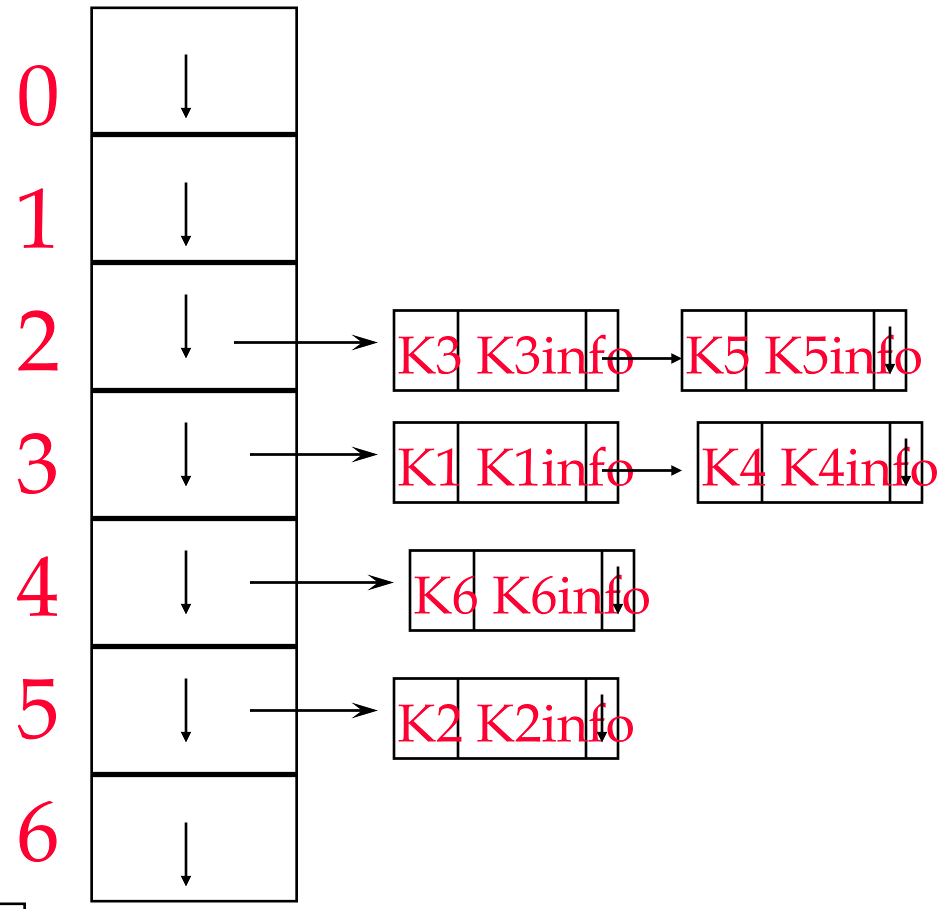
unsuccessful search?

Chaining

- Linear probing performs poorly
  - the search for a key involves comparison with keys having different hash values
  - making a local collision a global one

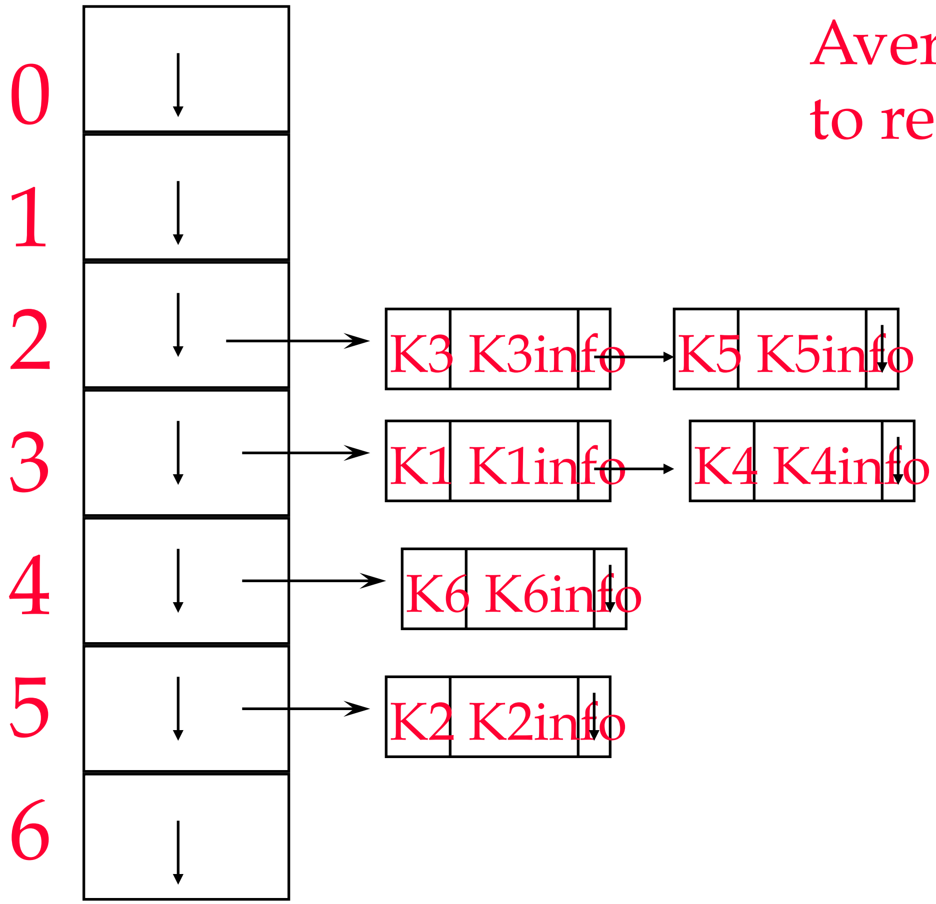
A Chained Hash Table

insert keys:
K1 --> 3
K2 --> 5
K3 --> 2
K4 --> 3
K5 --> 2
K6 --> 4



linked lists of synonyms

Search Performance



Average number of probes needed to retrieve the value with key K?

K	hash(K)	#probes
K1	3	1
K2	5	1
K3	2	1
K4	3	2
K5	2	2
K6	4	1

8/6 = 1.33 (successful)

unsuccessful search?

successful search performance

load factor	open addressing (linear probing)	open addressing (double hashing)	chaining
0.5	1.50	1.39	1.25
0.7	2.17	1.72	1.35
0.9	5.50	2.56	1.45
1.0	----	----	1.50
2.0	----	----	2.00

Exercises: P475-3, 6