Dictionaries

- the dictionary ADT
- binary search
- Hashing

Dictionaries

- Dictionaries store elements so that they can be located quickly using **keys**
- A dictionary may hold bank accounts
  - each account is an object that is identified by an account number
  - each account stores a wealth of additional information
    - including the current balance,
    - the name and address of the account holder, and
    - the history of deposits and withdrawals performed
  - an application wishing to operate on an account would have to provide the account number as a search **key**

The Dictionary ADT

- A dictionary is an abstract model of a database
  - A dictionary stores key-element pairs
  - The main operation supported by a dictionary is searching by key
- simple container methods: size(), isEmpty(), elements()
- query methods: findElem(k), findAllElem(k)
- update methods: insertItem(k,e), removeElem(k), removeAllElem(k)
- special element: NIL, returned by an unsuccessful search

The Dictionary ADT

- Supporting order (methods *min, max, successor, predecessor*) is not required, thus it is enough that **keys are comparable for equality**

The Dictionary ADT

- Different data structures to realize dictionaries
  - arrays, linked lists (inefficient)
  - **Hash table** (used in Java...)
  - Binary trees
  - Red/Black trees
  - AVL trees
  - B-trees
- In Java:
  - java.util.Dictionary – abstract class
  - java.util.Map – interface

Searching

INPUT

- sequence of numbers (database)
- a single number (query)

OUTPUT

- index of the found number or *NIL*

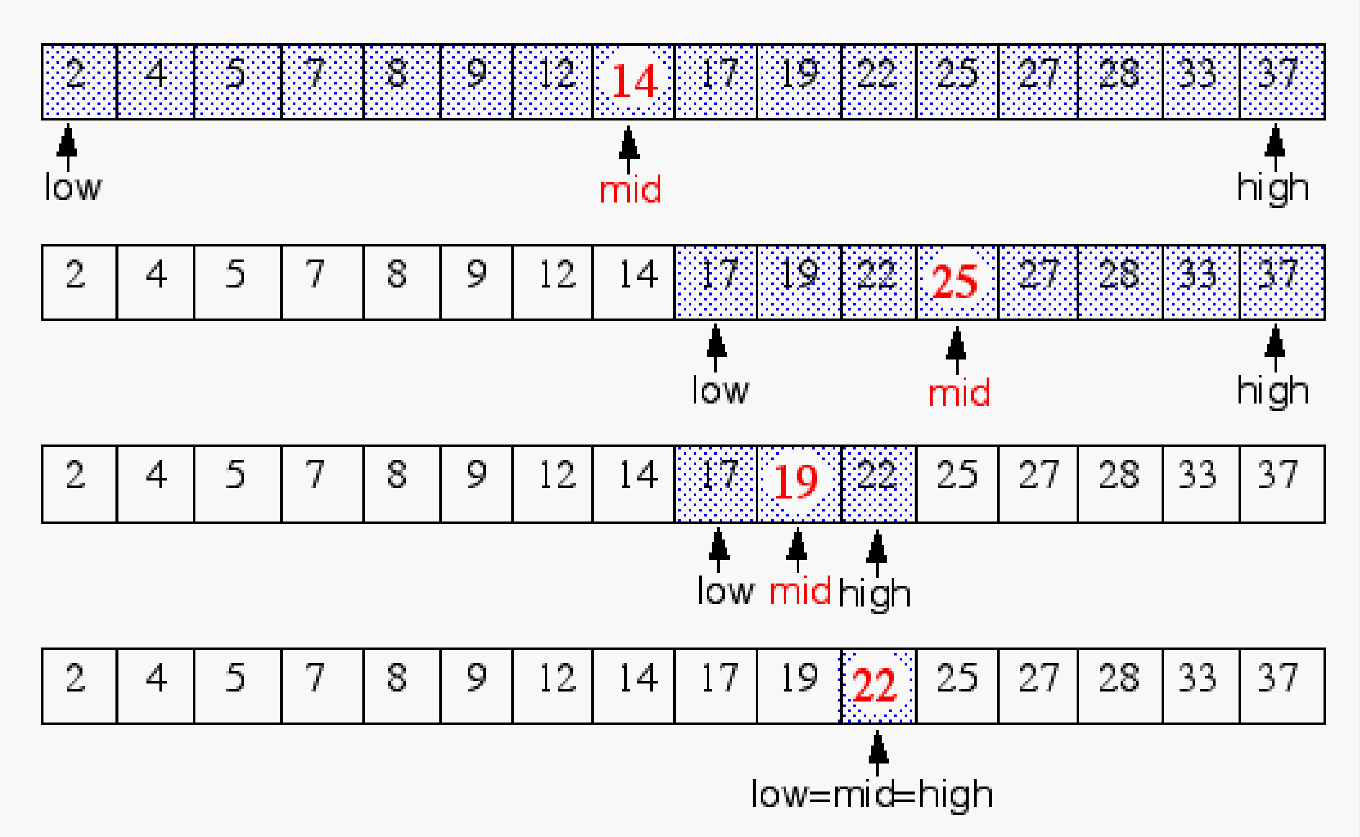
$a_1, a_2, a_3, \ldots, a_n;\ q$ → j

2 5 4 10 7; 5 → 2

2 5 4 10 7; 9 → *NIL*

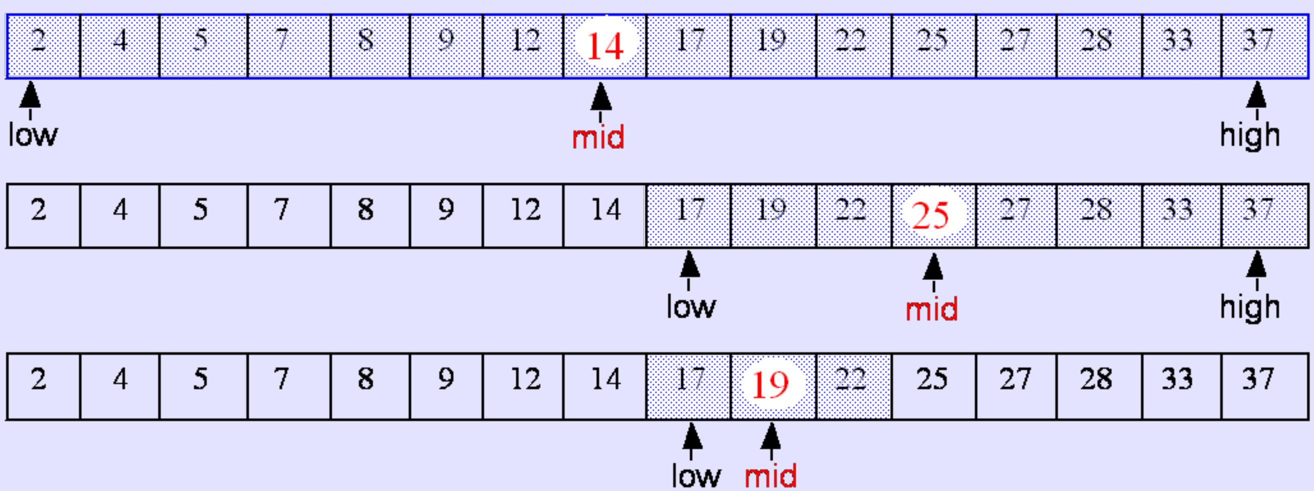
Binary Search

- Idea: *Divide and conquer*, a key design technique
- narrow down the search range in stages
- findElement(22)



A recursive procedure

```
Algorithm BinarySearch(A, k, low, high)
if low > high then return Nil
else   mid ← (low+high) / 2
       if k = A[mid] then return mid
       elseif k < A[mid] then
              return BinarySearch(A, k, low, mid-1)
       else return BinarySearch(A, k, mid+1, high)
```

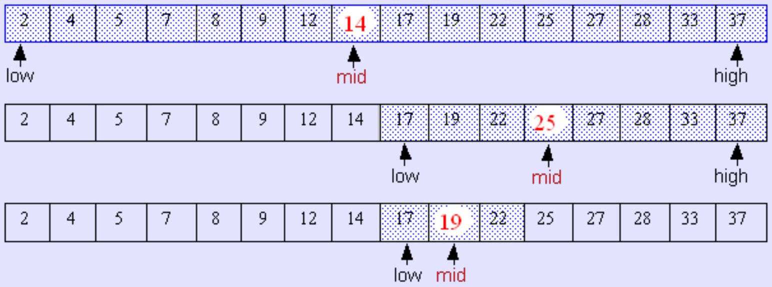


An iterative procedure

INPUT: A[1..n] – a sorted (non-decreasing) array of integers, key – an integer.

OUTPUT: an index *j* such that A[*j*] = k.

NIL, if $\forall j$ $(1 \le j \le n)$: A[*j*] $\neq$ k



low ← 1
high ← *n*
do
    mid ← (low+high)/2
    if A[mid] = k **then return** mid
    else if A[mid] > k **then** high ← mid-1
                        else low ← mid+1
while low <= high
return *NIL*

Running Time of Binary Search

- The range of candidate items to be searched is ***halved after each comparison***

comparison	search range
0	n
1	$n/2$
2	$n/4$
...	...
2^i	$n/2^i$
$\log_2 n$	1

- In the array-based implementation, access by rank takes O(1) time, thus binary search runs in O(log n) time

Searching in an unsorted array

INPUT: A[1..n] – an array of integers, q – an integer.
OUTPUT: an index j such that A[j] = q. NIL, if $\forall$j ($1 \le j \le n$): A[j] $\neq$ q

j $\leftarrow$ 1
while j $\leq$ n **and** A[j] $\neq$ q
do j++
if j $\leq$ n **then return** j
else return *NIL*

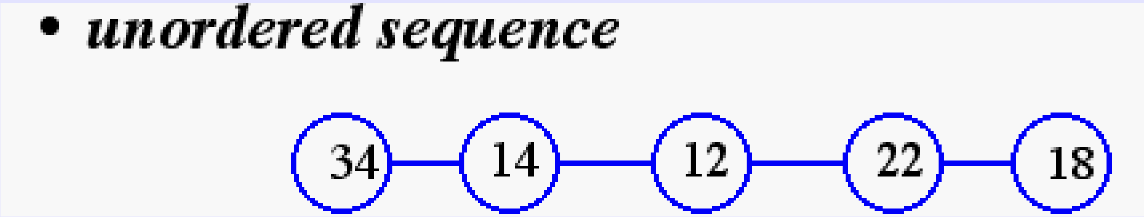
- Worst-case running time: O(n), average-case: O(n)
- We can't do better. This is a *lower bound* for the problem of searching in an arbitrary sequence.

The Problem

T&T is a large phone company, and they want to provide caller ID capability:

- given a phone number, return the caller's name
- phone numbers range from 0 to $r = 10^8 - 1$
- There are n phone numbers, $n << r$.
- want to do this as efficiently as possible

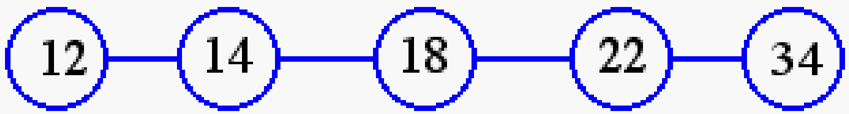
Using an unordered sequence



- searching and removing takes O(n) time
- inserting takes O(1) time
- applications to log files (frequent insertions, rare searches and removals)

Using an ordered sequence

- ***array-based ordered sequence*** (assumes keys can be ordered)



- searching takes O(log n) time (binary search)
- inserting and removing takes O(n) time
- application to look-up tables (frequent searches, rare insertions and removals)

Other Suboptimal ways

direct addressing: an array indexed by key:

- takes O(1) time,
- O(*r*) space where r is the range of numbers (10^8)
- huge amount of wasted space

(null)	(null)	Ankur	(null)	(null)
0000-0000	0000-0000	9635-8904	0000-0000	0000-0000

Another Solution

- Can do better, with a **Hash table** -- O(1) expected time, O(n+m) space, where m is table size
- Like an array, but come up with a function to map the large range into one which we can manage
  - e.g., take the original key, modulo the (relatively small) size of the array, and use that as an index
  - Insert (9635-8904, Ankur) into a hashed array with, say, five slots. 96358904 mod 5 = 4

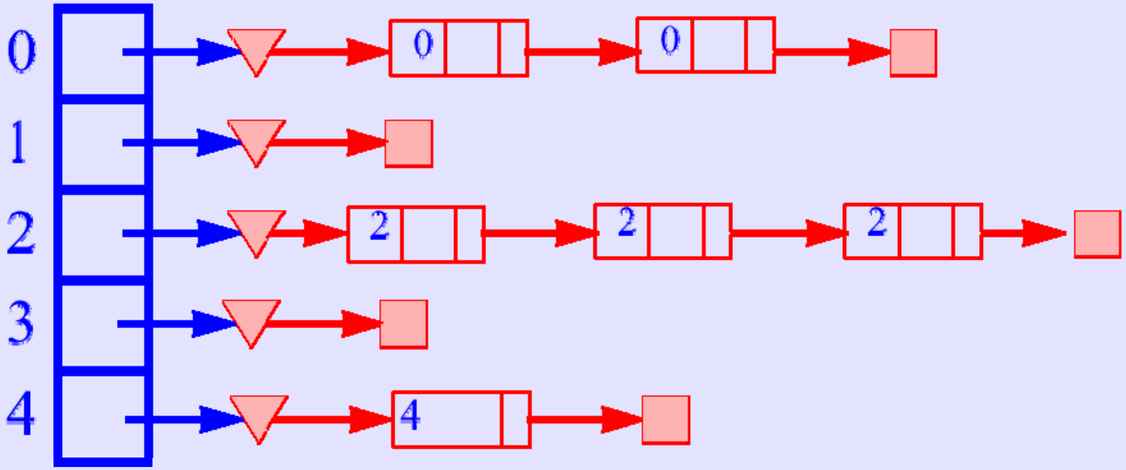
(null)	(null)	(null)	(null)	Ankur
0	1	2	3	4

An Example

- Let keys be entry no's of students in CSL201. eg. 2004CS10110.
- There are 100 students in the class. We create a hash table of size, say 100.
- Hash function is, say, last two digits.
- Then 2004CS10110 goes to location 10.
- Where does 2004CS50310 go?
- Also to location 10. We have a collision!!

Collision Resolution

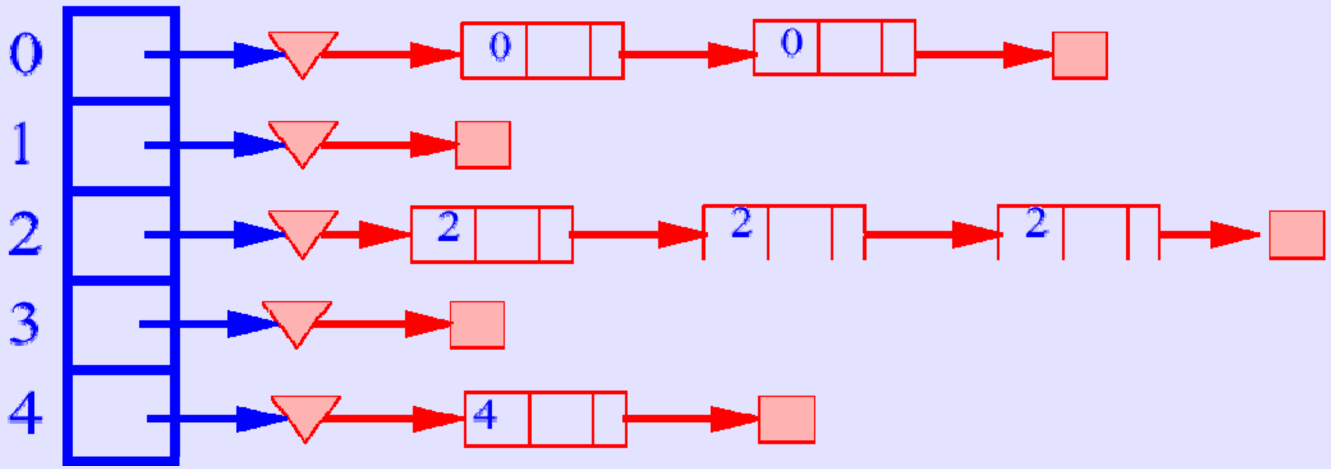
- How to deal with two keys which hash to the same spot in the array?
- Use **chaining**
  - Set up an array of links (a **table**), indexed by the keys, to **lists** of items with the same key



  - Most efficient (time-wise) collision resolution scheme

Collision resolution (2)

- To find/insert/delete an element
  - using *h*, look up its position in table *T*
  - Search/insert/delete the element in the linked list of the hashed slot



Analysis of Hashing

- An element with key *k* is stored in slot *h*(*k*) (instead of slot *k* without hashing)
- The hash function *h* maps the universe *U* of keys into the slots of hash table T[0...*m*-1]

$$h : U \rightarrow \{0, 1, ..., m-1\}$$

- Assume time to compute *h*(*k*) is Θ(1)

Analysis of Hashing(2)

- An good hash function is one which distributes keys evenly amongst the slots.
- An ideal hash function would pick a slot, uniformly at random and hash the key to it.
- However, this is not a hash function since we would not know which slot to look into when searching for a key.
- For our analysis we will use this simple uniform hash function
- Given hash table T with m slots holding n elements, the **load factor** is defined as $\alpha = n/m$

Analysis of Hashing(3)

Unsuccessful search

- element is not in the linked list
- *Simple uniform* hashing yields an average list length $\alpha = n/m$
- expected number of elements to be examined α
- search time $O(1+\alpha)$ (includes computing the hash value)

Analysis of Hashing (4)

Successful search

- assume that a new element is inserted at the end of the linked list
- upon insertion of the i-th element, the expected length of the list is $(i-1)/m$
- in case of a successful search, the expected number of elements examined is 1 more that the number of elements examined when the sought-for element was inserted!

Analysis of Hashing (5)

- The expected number of elements examined is thus

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{m}\right) = 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1)$$

$$= 1+\frac{1}{nm}\cdot\frac{(n-1)n}{2}$$

$$= 1+\frac{n-1}{2m}$$

$$= 1+\frac{n}{2m}-\frac{1}{2m}$$

$$1+\frac{\alpha}{2}-\frac{1}{2m}$$

- Considering the time for computing the hash function, we obtain

$$\Theta(2+\alpha/2-1/2m)=\Theta(1+\alpha)$$

Analysis of Hashing (6)

Assuming the number of hash table slots is proportional to the number of elements in the table

- n=O(m)
- α = n/m = O(m)/m = O(1)
- searching takes constant time on average
- insertion takes *O*(1) worst-case time
- deletion takes *O*(1) worst-case time when the lists are doubly-linked