Unit I Hashing

Hash Table- Concepts-hash table, hash function, basic operations, bucket, collision, probe, synonym, overflow, open hashing, closed hashing, perfect hash function, load density, full table, load factor, rehashing, issues in hashing, hash functions- properties of good hash function, division, multiplication, extraction, mid-square, folding and universal, Collision resolution strategies- open addressing and chaining, Hash table overflow- open addressing and chaining, extendible hashing, closed addressing and separate chaining.

Skip List- representation, searching and operations- insertion, removal

Dictionary Implementations So Far

	Unsorted linked list	Sorted Array	BST	AVL	Splay (amortized)
Insert					
Find					
Delete					

Hashing

- If we have a table organization and a search technique which tries to retrieve the key in a single access, it would be very efficient. i.e now our need is to search the element in constant time and less key comparisons should be involved. To do so, the position of the key in the table should not depend upon the other keys but the location should be calculated on the basis of the key itself. Such an organization and search technique is called hashing.

In hashing the address or location of an identifier X is obtained by using some function f(X) which gives the address of X in a table.

For storing record	**For accessing record**
Key ↓ Generate array index ↓ Store the record on that array index	Key ↓ Generate array index ↓ Get the record from that array index

Hashing Terminology

- ***Hash Function :*** A function that transforms a key X into a table index is called a hash function.
- ***Hash Address :*** The address of X computed by the hash function is called the hash address.
- ***Synonyms :*** Two identifiers I1 and I2 are synonyms if f(I1) = f(I2)

Hashing Terminology

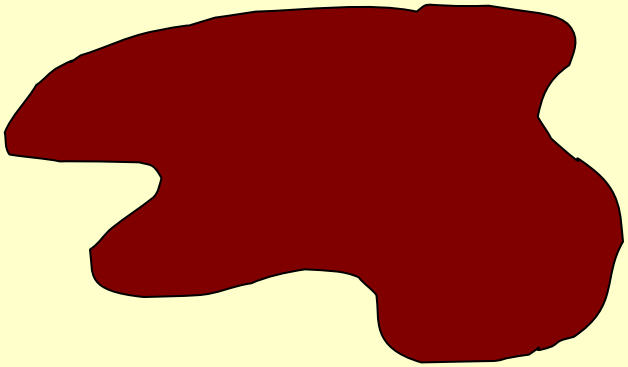
- ***Collision :*** When two non identical identifiers are mapped into the same location in the hash table, a collision is said to occur, i.e. f(I1) = f(I2)
- ***Overflow :*** An overflow is said to occur when an identifier gets mapped onto a full bucket. When s = 1 i.e, a bucket contains only one record, collision and overflow occur simultaneously. Such situation is called a Hash clash.
- ***Bucket :*** Each hash table is partitioned into b buckets ht[0] …… ht[b-1].
- Each bucket is capable of holding ‘s’ records. Thus, a bucket consists of ‘s’ slots. When s = 1, each bucket can hold 1 record.
- The function f(X) maps an identifier X into one of the ‘b’ buckets i.e. from 0 to b-1.

- The capacity is the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created.

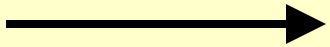
- The load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased.

Hash Tables

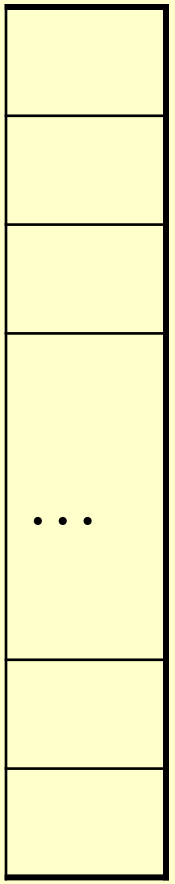
- Constant time accesses!
- A **hash table** is an array of some fixed size, usually a prime number.
- General idea:



hash function:
h(K)



hash table



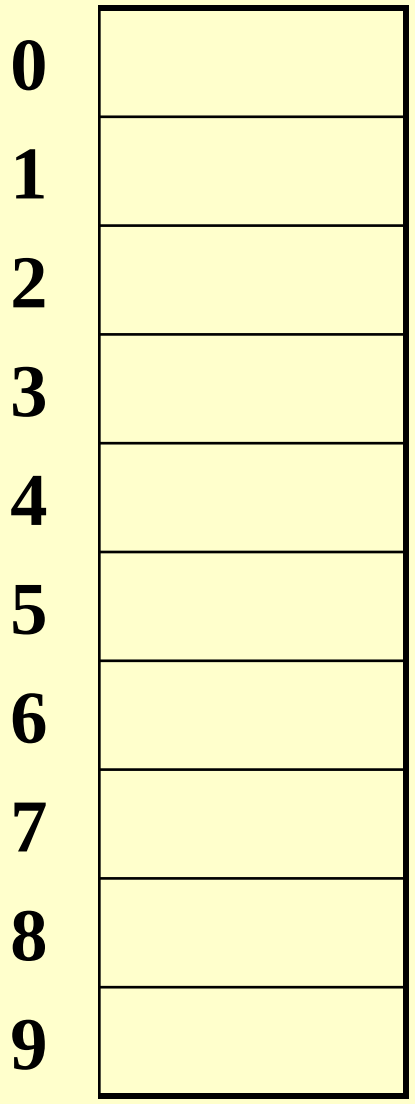
key space (e.g., integers, strings)

Example

- key space = integers
- TableSize = 10

- **h**(K) = K mod 10

- **Insert**: 7, 18, 41, 94

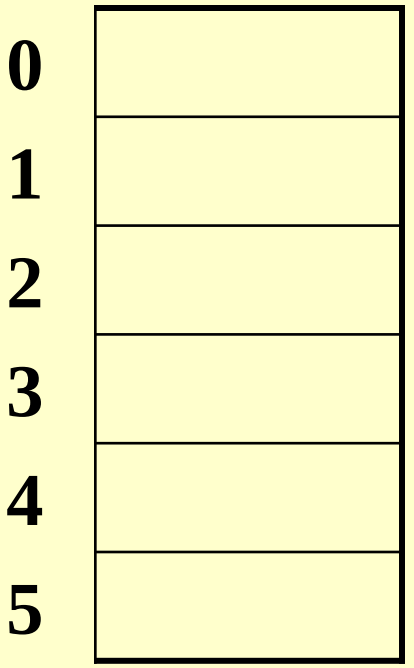


Another Example

- key space = integers
- TableSize = 6

- **h**(K) = K mod 6

- **Insert**: 7, 18, 41, 34



Hash Functions

A hashing function f transforms an identifier X into a bucket address in the hash table.

Characteristics:

1. **simple/fast** to compute,
2. Avoid **collisions**
3. have keys distributed **evenly** among cells.

Hash Functions

- Truncation Method
- Mid Square Method
- Folding Method
- Modular Method
- Hash function for floating point numbers
- Hash function for strings.

Truncation Method

- Easiest Method
- Take only part of the key as address, it can be some rightmost digit or leftmost digit
- Example : Let us take some 8 digit keys
- 82394561, 87139465, 83567271, 85943228
- Hash Addresses : 61, 65, 71 and 28 (Rightmost two digits for Hash table size 100)
- Easy to compute but chances of collision is more because last two digits can be same in different keys.

Mid Square Method

- We square the key, and we take some digits from the middle of that number as an address.
- This is a very widely used function in symbol table applications. Since middle bits of the square will depend on all the bits in the identifier/ key, different identifiers will result in different hash addresses thus minimizing collision.
- **Example :** We will choose the middle two digits (the size of hash table = 100)
- If X = 225 , X^2 = 050625 , Hash address = 06
- If X = 3205 , X^2 = 01027205 , Hash address = 27

Folding Method

- In this method, the identifier X is partitioned into several parts all of the same length except the last. These parts are added to obtain the hash address.
- Addition is done in two ways.
  - Shift Folding : All parts except the last are shifted so that their least significant bits correspond to each other.
  - Folding at the boundaries : The identifier is folded at the part boundaries and the bits falling together are added.

Example

X = 12320324211220

P1	P2	P3	P4	P5
123	203	241	112	20

P1	123	P1	123
P2	203	P2	302
P3	241	P3	241
P4	112	P4	211
P5	20	P5	20
	------		-------
	699		897
	Shift Folding		Folding at boundaries

Modular Method

- Take the key, do the modulus operation and get the remainder as address for hash table.
- It ensures that the address will be in the range of hash table.
- Here only one thing we should keep in mind that table size should not be in power of two, otherwise it will give more collision.
- The best way to minimize the collision is to take the table size a prime number.
- Example : table size : 31

  X = 134

  f(X) = X % 31 = 134 % 31 = 10

tableSize: Why Prime?

- Suppose
  - data stored in hash table: 7160, 493, 60, 55, 321, 900, 810
  - tableSize = 10

    data hashes to 0, 3, 0, 5, 1, 0, 0
  - tableSize = 11

    data hashes to 10, 9, 5, 0, 2, 9, 7

Real-life data tends to have a pattern

Being a multiple of 11 is usually *not* the pattern ☺

Hash function for floating point numbers

- Getting the hash address for floating point numbers has some what little bit different approach but it also requires modulus operation at the end for getting hash address in range of hash table. The whole operation can be defined as :
  - Take the fractional part of key.
  - Multiply the fractional part with the size of the hash table.
  - Take the integer part of the multiplication result as a hash address of key.
- Example : X = 19.463 Hash table size = 97
- 0.463 x 97 = 44.911
- f(X) = 44

Hash function for strings

- Every character has some ASCII value that can be used for calculation in generating hash key value and that value can be used with modulus operation for mapping with hash table.
- Example : X = suresh Hash table size = 97
- Suresh = s + u + r + e + s + h
- = 115 + 117 + 114 + 101 + 115 + 104
- ` = 666
- After modulus operation
- f(X) = 666 % 97 = 84

Sample Hash Functions:

- key space = strings
- $s = s_0\, s_1\, s_2 \ldots s_{k-1}$

1. $h(s) = s_0 \bmod \text{TableSize}$
2. $h(s) = \left(\sum_{i=0}^{k-1} s_i \right) \bmod \text{TableSize}$
3. $h(s) = \left(\sum_{i=0}^{k-1} s_i \cdot 37^i \right) \bmod \text{TableSize}$

Perfect Hash Function

- In computer science, a **perfect hash function** for a set S is a **hash function** that maps distinct elements in S to a set of integers, with no collisions.
- A **perfect hash function** has many of the same applications as other **hash functions**, but with the advantage that no collision resolution has to be implemented.

Collision Resolution

Collision: when two keys map to the same location in the hash table.

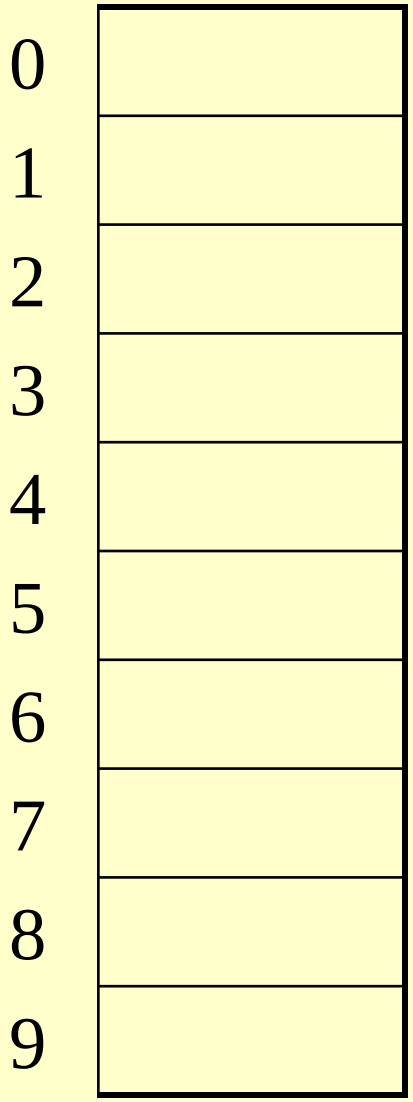
Two ways to resolve collisions:

1. Separate Chaining
2. Open Addressing (linear probing, quadratic probing, double hashing)

Collision Resolution Policies

- Two classes:
  - (1) “Open hashing” equals “separate chaining”
  - (2) “Closed hashing” equals “open addressing”
- Difference has to do with whether collisions are stored *outside the table* (open hashing) or whether collisions result in storing one of the records at *another slot in the table* (closed hashing)

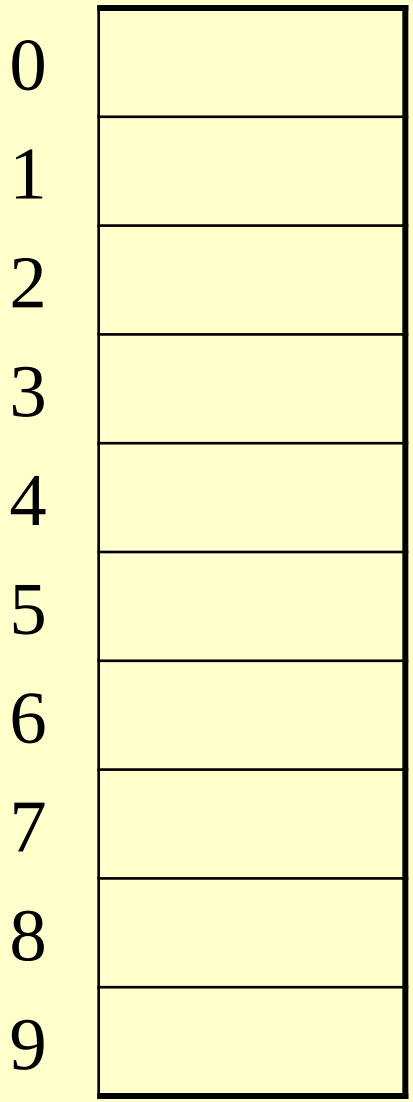
Separate Chaining



Insert:
10
22
107
12
42

- **Separate chaining**: All keys that map to the same hash value are kept in a list (or “bucket”).

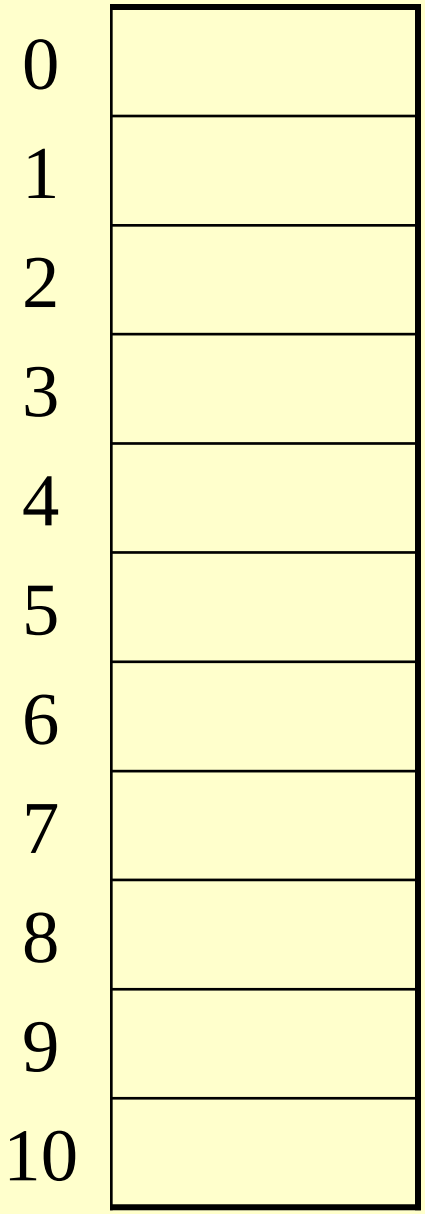
Open Addressing



Insert:
38
19
8
109
10

- **Linear Probing**: after checking spot h(k), try spot h(k)+1, if that is full, try h(k)+2, then h(k)+3, etc.

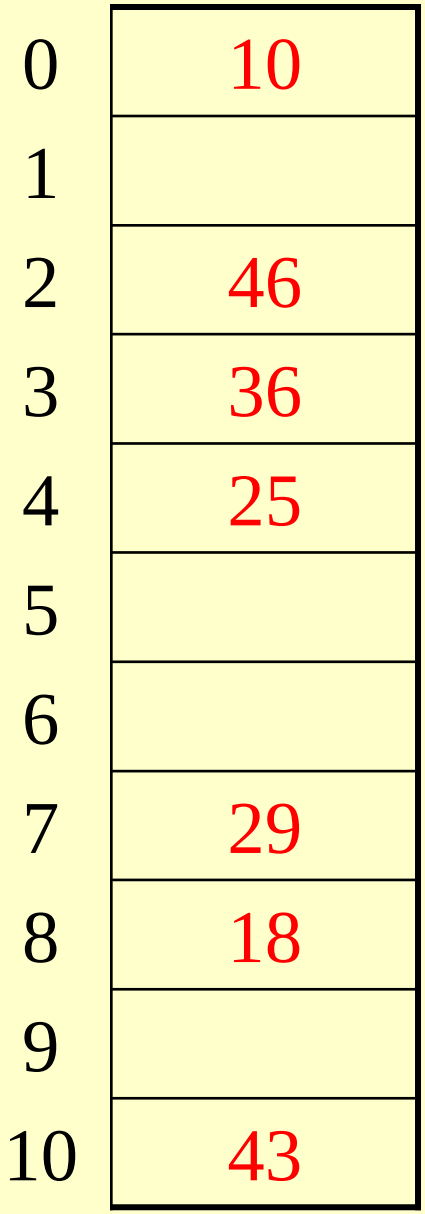
Linear Probing



- Hash table size : 11
- Hashing Function
- F(X) = X mod HTsize

Insert:
29
18
43
10
36
25
46

Linear Probing



- Hash table size : 11
- Hashing Function
- F(X) = X mod HTsize

Insert:
29
18
43
10
36
25
46

Linear Probing

f(i) = i

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

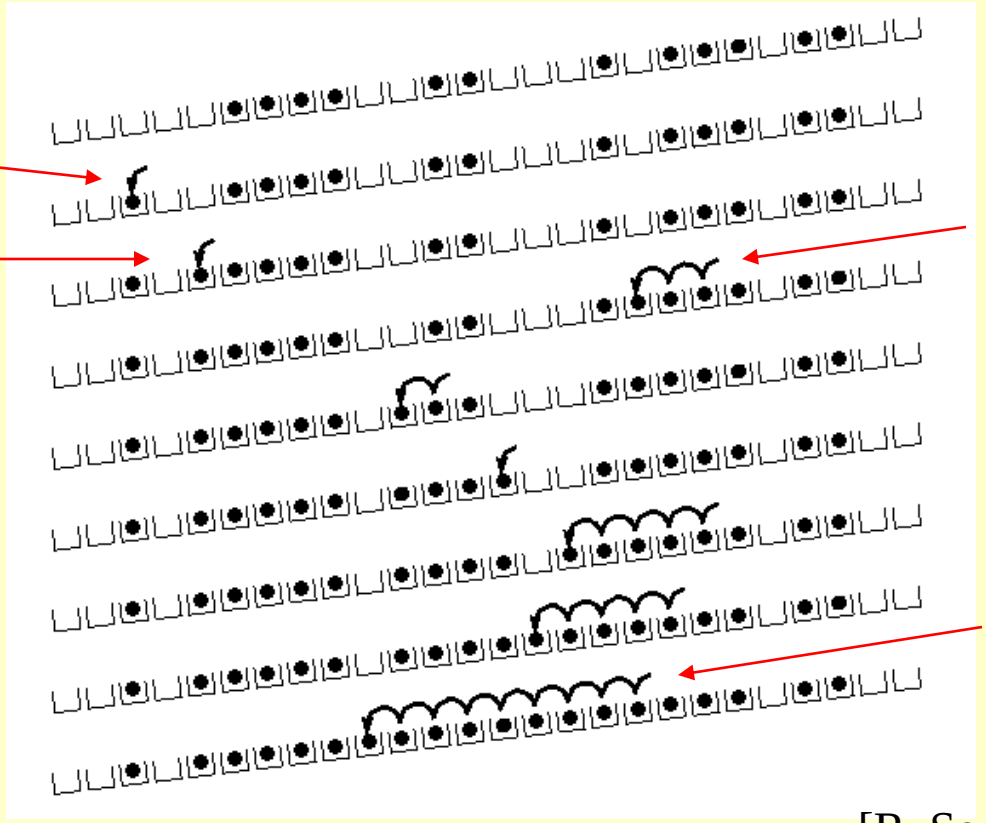
  1^{th} probe = (h(k) + 1) mod TableSize

  2^{th} probe = (h(k) + 2) mod TableSize

  . . .

  i^{th} probe = (h(k) + i) mod TableSize

Linear Probing – Clustering



[R. Sedgewick]

Quadratic Probing

- This method is used to avoid the clustering problem in the above method.
- Suppose hash address is h then in the case of collision, linear probing search the location h, h+1, h+2 …..(%size).
- In Quadratic probing a quadratic function of i is used as the increment i.e. instead of checking (i+ 1)th index, this method checks the index computed from a quadratic equation. This ensures that the identifiers are fairly spread out in the table.

Quadratic Probing

Less likely to encounter Primary Clustering

$f(i) = i^2$

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + 1) mod TableSize

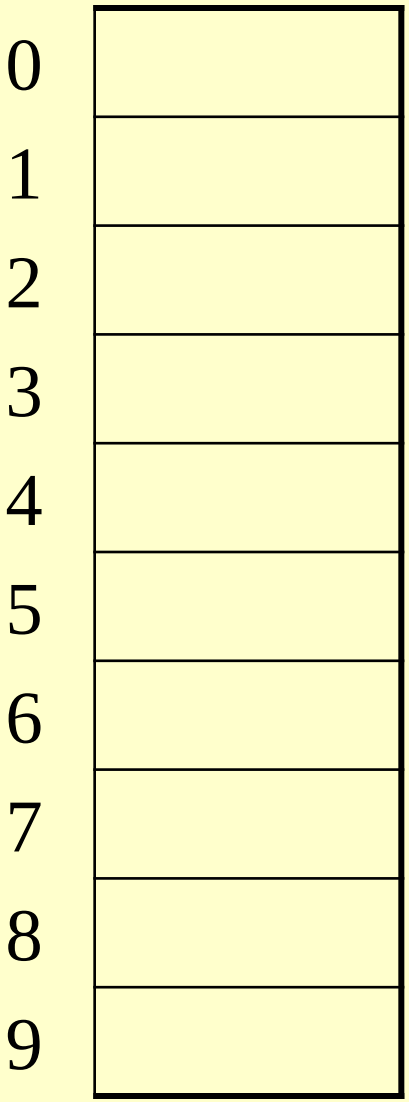
  2^{th} probe = (h(k) + 4) mod TableSize

  3^{th} probe = (h(k) + 9) mod TableSize

  . . .

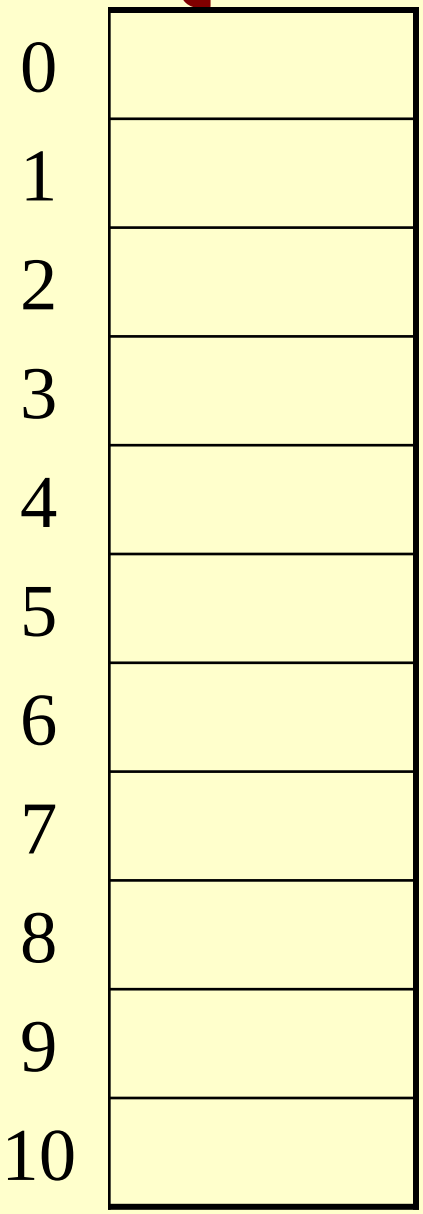
  i^{th} probe = (h(k) + i^2) mod TableSize

Quadratic Probing



Insert:
89
18
49
58
79

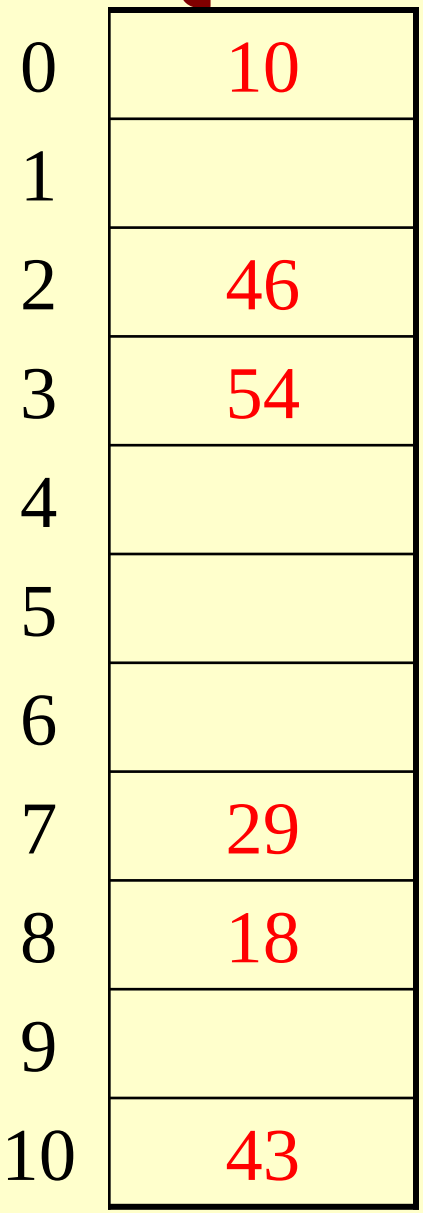
Quadratic Probing



Insert:
29
18
43
10
46
54

Hash table size 11
Quadratic equation = i^2

Quadratic Probing



Insert:
29
18
43
10
46
54

Hash table size 11
Quadratic equation = i^2

Quadratic Probing Example

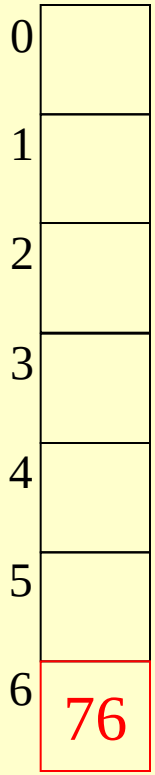
insert(76)
76%7 = 6

insert(40)
40%7 = 5

insert(48)
48%7 = 6

insert(5)
5%7 = 5

insert(55)
55%7 = 6



But…

insert(47)
47%7 = 5

Double hashing

- In this method, if an overflow occurs, a new address is computed by using another hash function. A series of hash functions f1,f2, …..,fn are used. Hashed values f1(X), f2(X), …. fn(X) are examined in order till an empty slot is found.
- Example : H = key % 13 H’ = 11 – (key %11)
- So at the time of collision hash address for the next prob will be as
- (H+H’) % 13 = ((key % 13) + (11 – (key % 11))) % 13

Double Hashing

$f(i) = i * g(k)$

where g is a second hash function

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + g(k)) mod TableSize

  2^{th} probe = (h(k) + 2*g(k)) mod TableSize

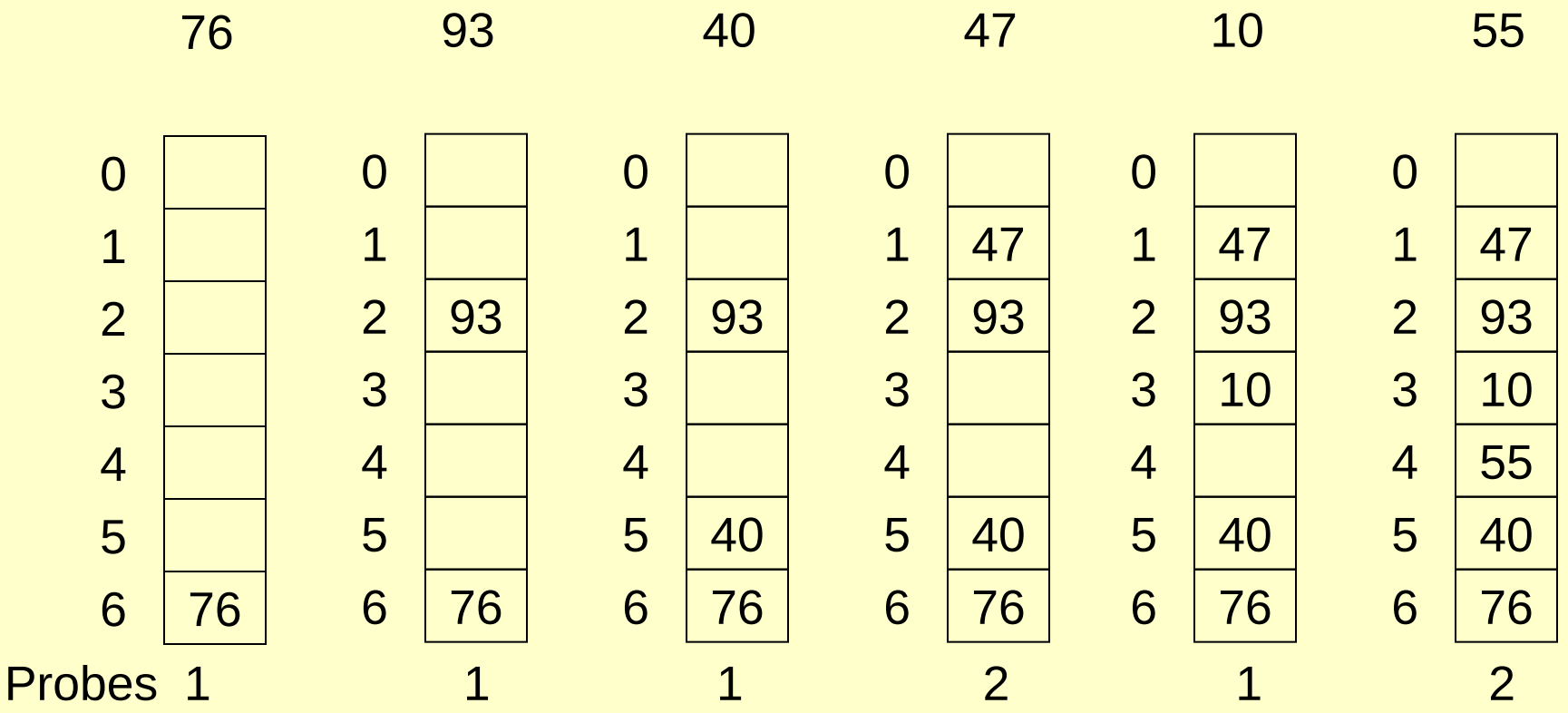
  3^{th} probe = (h(k) + 3*g(k)) mod TableSize

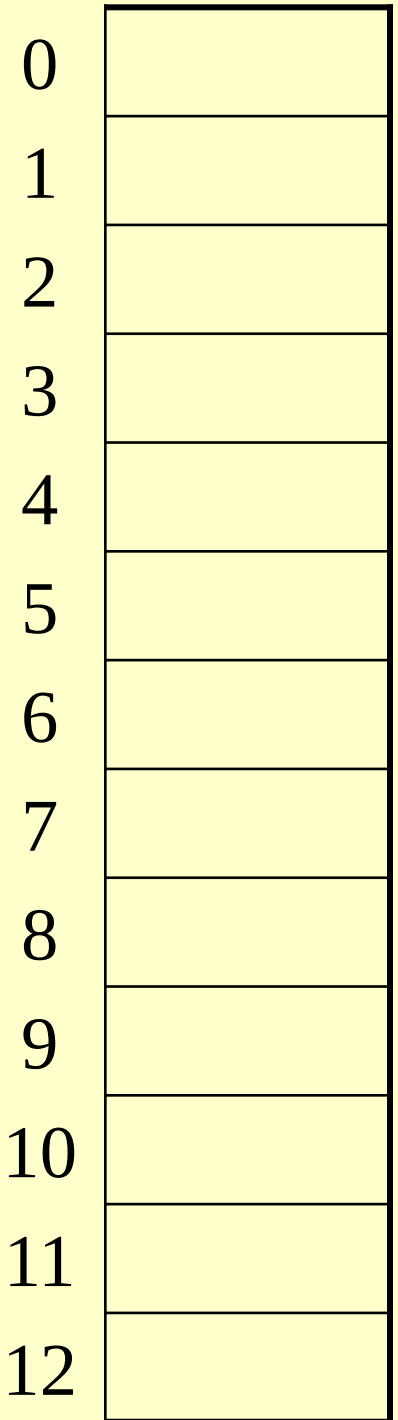
  . . .

  i^{th} probe = (h(k) + i*g(k)) mod TableSize

Double Hashing Example

h(k) = k mod 7 and g(k) = 5 – (k mod 5)





Double Hashing

Insert:
8
55
48
68

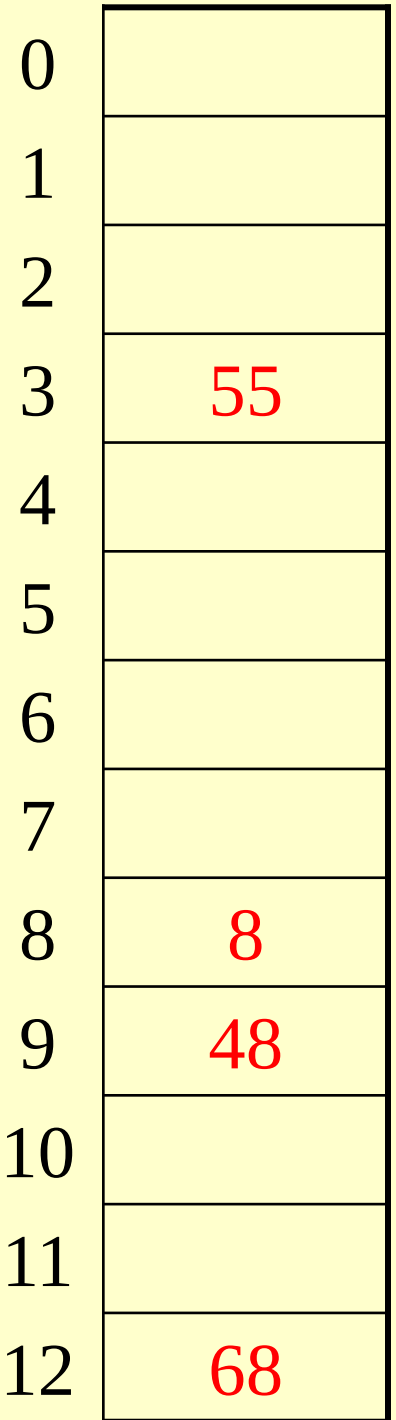
Hash table size 13

H = key % 13

H' = 11 – (key %11)

(H+H') % 13 =

= ((key % 13) + (11 – (key % 11))) % 13



Double Hashing

Insert:
8
55
48
68

Hash table size 13

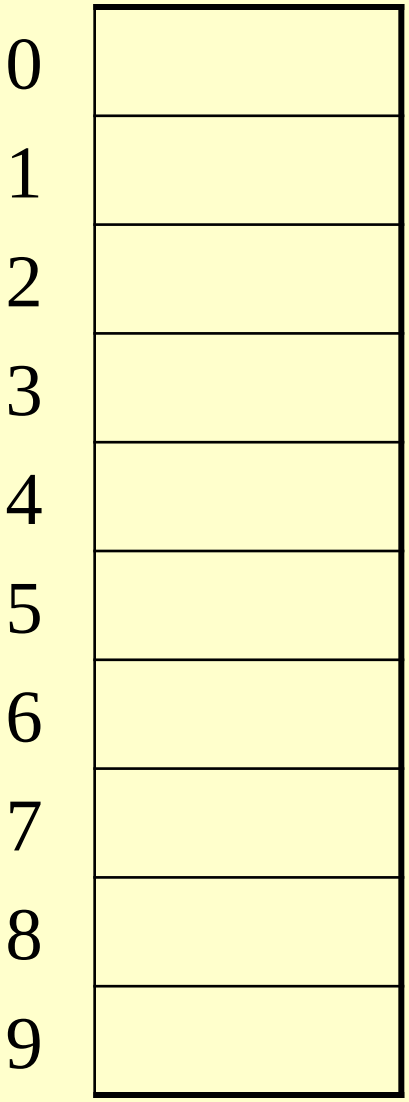
H = key % 13

H' = 11 – (key %11)

(H+H') % 13 =

= ((key % 13) + (11 – (key % 11))) % 13

Resolving Collisions with Double Hashing



Hash Functions:

$H(K) = K \bmod M$

$H_2(K) = 1 + ((K/M) \bmod (M-1))$

M =

Insert these values into the hash table in this order. Resolve any collisions with double hashing:

13

28

33

147

43

Rehashing

Idea: When the table gets too full, create a bigger table (usually 2x as large) and hash all the items from the original table into the new table.

- When to rehash?
  - half full ($\lambda = 0.5$)
  - when an insertion fails
  - some other threshold
- Cost of rehashing?

Rehashing

- There are chances of insertion failure when hash table is full. So the solution for this particular case is to create a new hash table with the double size of previous hash table.
- Here we will use new hash function and we will insert all the elements of the previous hash table.
- So we will scan the elements of previous hash table one by one and calculate the hash key with new hash function and insert them into new hash table.
- This technique is called rehashing.
- It ensures the insertion of element in hash table.

Chaining

- The problem of the complexity of the search algorithm will still come into picture as we are using the linear search for some portion.
- To avoid this the concept of the chain comes into picture.
- We will now provide an extra field with the record which will point to the record number which contains the record having same hash value.
- Chaining means remembering the record number , where the record which has same hash value is stored

Chaining without Replacement

- The hash address of an identifier is computed
- If this position is vacant, it is placed there.
- It its position is occupied, the identifier is put in the next vacant position and a chain is formed to the new position.

Resolving Collisions with Chaining without replacement

	Value	Chain
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Hash function : f(X) = X % 10
Table size = 10:

Insert these values into the hash table in this order.

11
32
41
54
33

Resolving Collisions with Chaining without replacement

	Value	Chain
0		-1
1	11	3
2	32	-1
3	41	5
4	54	-1
5	33	-1
6		-1
7		-1
8		-1
9		-1

Hash function : f(X) = X % 10
Table size = 10:

Insert these values into the hash table in this order.
11
32
41
54
33

Disadvantage

- The main idea is to chain all identifiers having same hash address (synonyms).
- However, since an identifier occupies the position of another identifier, even non-synonyms get chained together thereby increasing complexity.

Chaining with Replacement

- In this method, if another identifier Y is occupying the position of an identifier X, X replaces it and then Y is relocated to a new position.

Resolving Collisions with Chaining without replacement

	Value	Chain
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Hash function : f(X) = X % 10
Table size = 10:

Insert these values into the hash table in this order.

11
32
41
54
33

Resolving Collisions with Chaining without replacement

	Value	Chain
0		-1
1	11	5
2	32	-1
3	33	-1
4	54	-1
5	41	-1
6		-1
7		-1
8		-1
9		-1

Hash function : f(X) = X % 10
Table size = 10:

Insert these values into the hash table in this order.
11
32
41
54
33

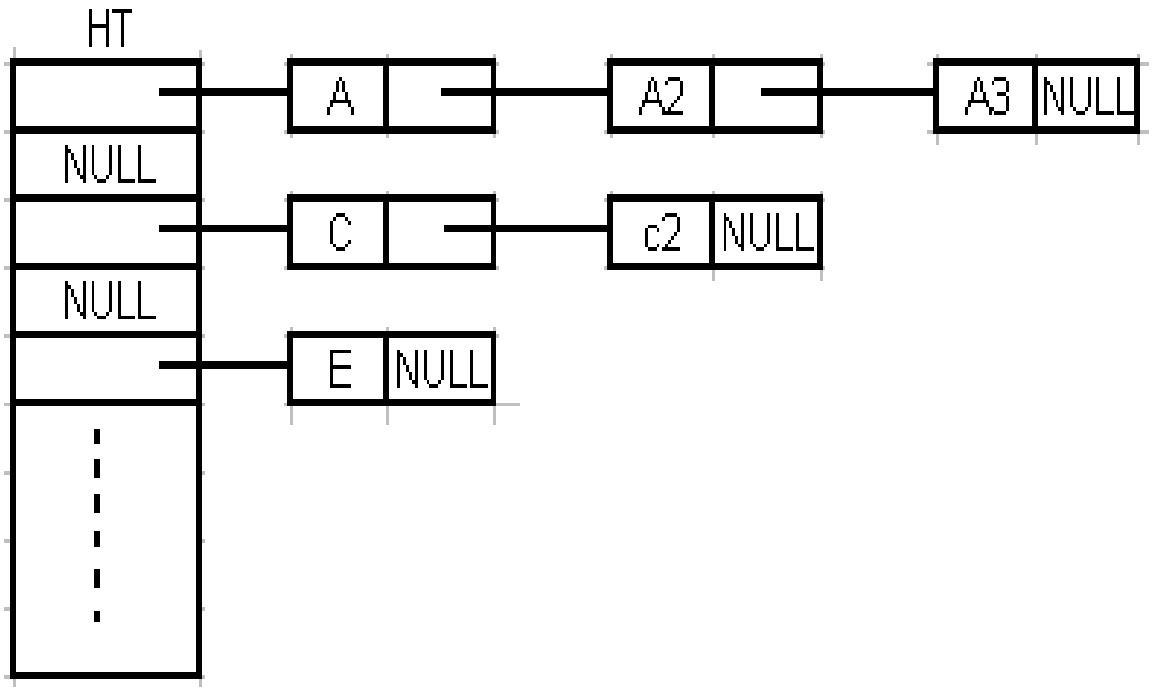
Pros & Cons

- ***Advantage :*** Most of the identifiers occupy their valid position. Searching becomes easier since only the synonyms are chained.
- ***Disadvantages :*** Insertions and deletions take more time.

Chaining using linked lists

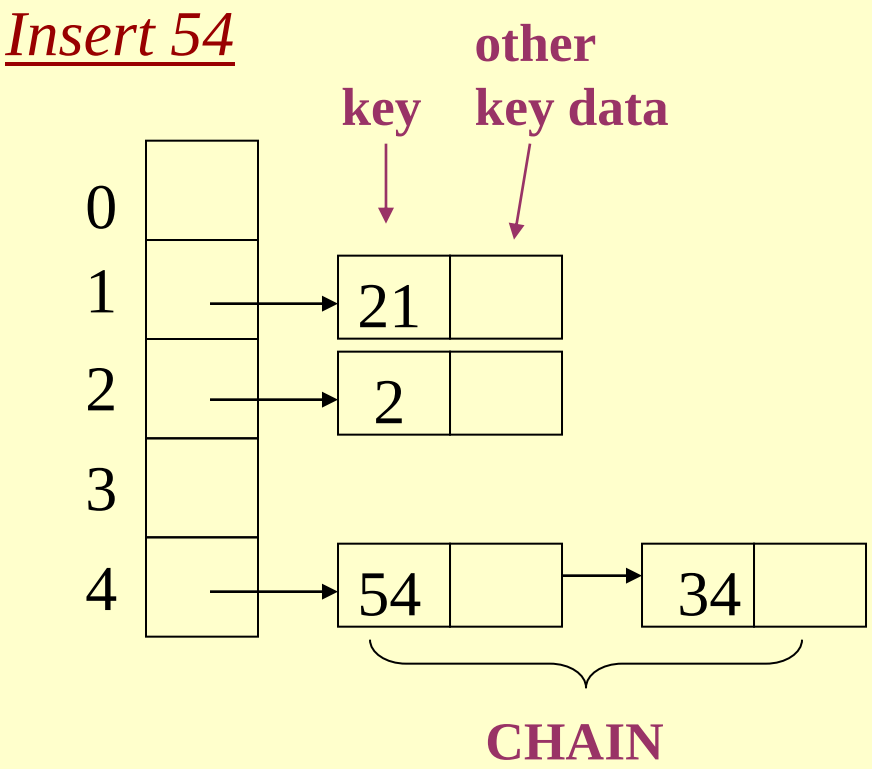
- Dynamic representation of hash table
- Used when the number of identifiers varies dynamically.
- No limit on the number of identifiers in a bucket.
- possible using linked lists.
- Memory is allocated and de-allocated for an identifier dynamically.
- The Hash table is maintained as an array of pointers each pointing to a linked list.
- Each list contains all identifiers having the same bucket address (synonyms).
- When the address of a new identifier is computed, this identifier gets added to the list corresponding to its bucket.
- Searching involves computing the hash address of the identifier and traversing only the list stored at that bucket

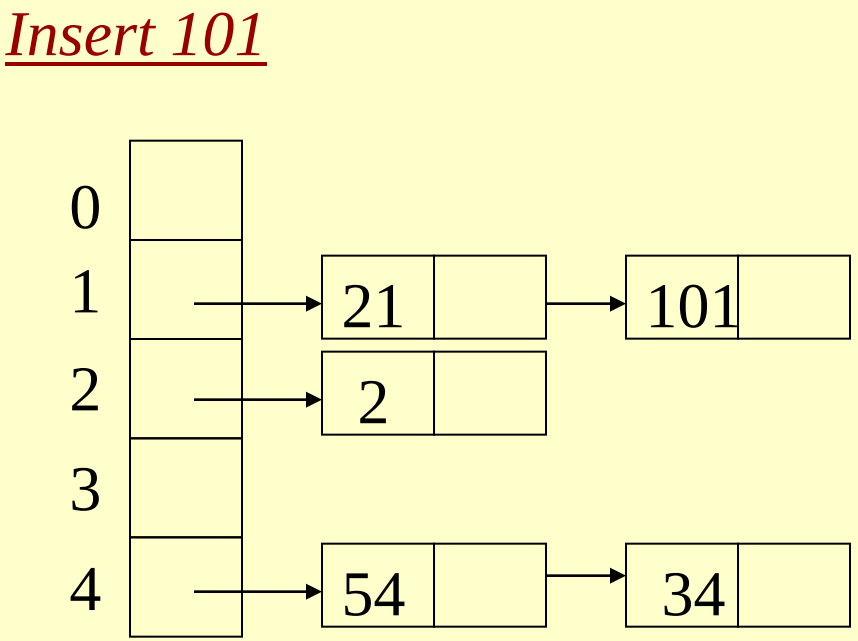
Example



Hashing with Chaining using linklists

The problem is that keys 34 and 54 hash in the same entry (4). We solve this *collision* by placing all keys that hash in the same hash table entry in a LIFO linked list (**chain**) pointed by this entry:





Pros & Cons

Advantages :

- Most efficient method to resolve collision
- There is no limit on the number of identifiers
- Searching, Insertion, and Deletions are done efficiently.
- The hash table will never be full.

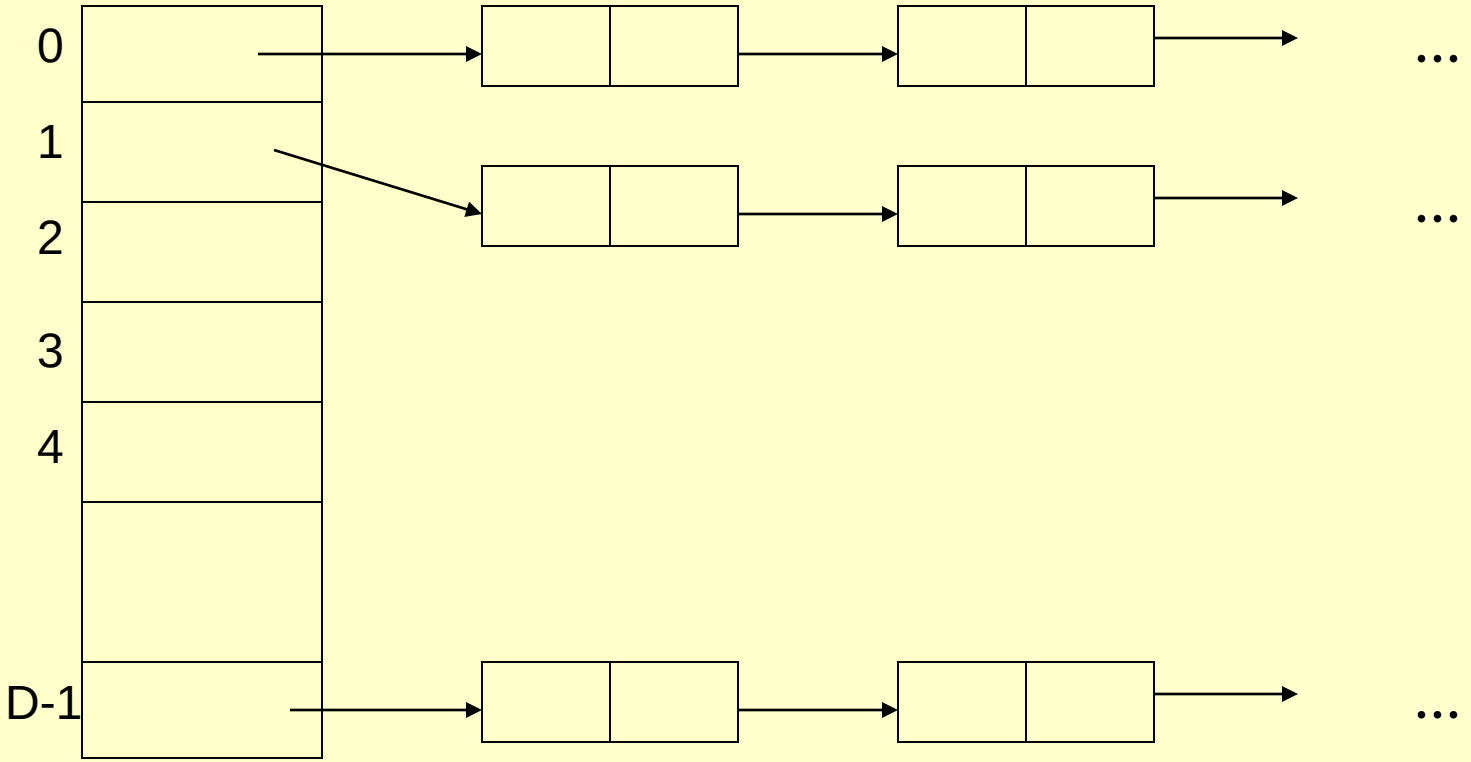
Disadvantages:

- If many identifiers are put into a single list, searching time within a bucket will increase.
- Handling of the pointer array and linked lists is more complex as compared to simple arrays

Open Hashing

- Each bucket in the hash table is the head of a linked list
- All elements that hash to a particular bucket are placed on that bucket's linked list
- Records within a bucket can be ordered in several ways
  - by order of insertion, by key value order, or by frequency of access order

Open Hashing Data Organization



Use

- Open hashing is most appropriate when the hash table is kept in main memory, implemented with a standard in-memory linked list

Hashing Summary

- Hashing is one of the most important data structures.
- Hashing has many applications where operations are limited to find, insert, and delete.
- Dynamic hash tables have good amortized complexity.

Skip Lists

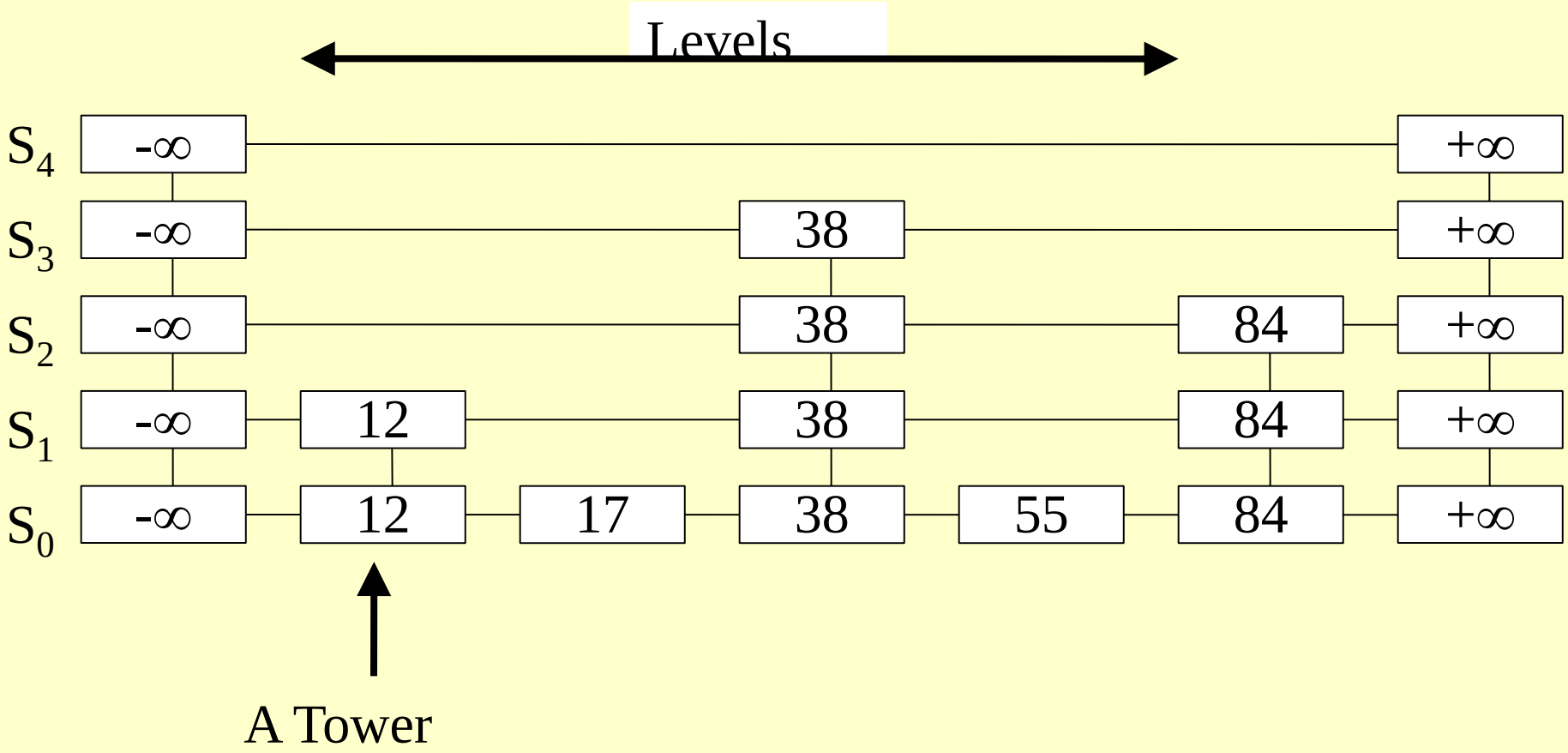
- Binary Search Trees: O(log n)
  - If, and only if, the tree is “balanced”
- Insertion of partially sorted data is optimally bad for binary search trees.
- Skip Lists make use of some randomization to maintain O(log n) search and update times
  - These are *on average* figures
- Analysis of skip lists is interesting
  - Worst case is potentially very bad
  - We will not cover the analysis here.

Randomness

- Skip lists need random numbers
- Systems provide pseudo-random number generators
- Usually a linear congruential feedback algorithm:
  - $X_i = (A \times X_{i-1} + B) \bmod C$
  - X_0 is called the “seed” value
- Yields numbers in the range [0,1]
- Note: **NOT REALLY RANDOM**!
- Computers DO NOT produce random numbers – these are deterministic
  - Some applications need truly random numbers – this is hard!

Skip Lists

- A Skip List for a Dictionary D is a series of lists: $\{S_0, S_1, \ldots, S_h\}$
  - S_0 contains all the elements in D
  - S_i contains a randomly chosen subset of elements from S_{i-1}
  - Each list S_i is sorted
- We introduce two special elements: $-\infty$ & $+\infty$ which have their obvious meaning.
  - Each list S_i contains these, S_h contains only these.



H = height of a skip list

Skip List - Informal

- We set the lists up so that:
  - An item that is in S_i is in S_{i+1} with probability ½
- So S_1 has about n/2 items in it
  - S_2 has about n/4 items
  - S_i has about $n/(2^i)$ items
- The height of the skip list is about $\log_2(n)$

NOTE:

- Unlike a binary search tree, this “halving” property is not enforced
  - It is approximate

Skip List Operations

- We use a positional abstraction

Operations

- After(p) – position after p on the same level
- Before(p) – position before p on the same level
- Below(p) – position below p in the same tower
- Above(p) – position above p in the same tower

Searching

Simple: binary search (modified)

- The upper level lists provide something like a "thumb index"

- Algorithm for searching for the key k in a skip list
  - Returns p = reference to the key in the list which is equal or larger than k

p $\leftarrow$ first element of S_k
while below(p) $\neq$ **null do**
    p $\leftarrow$ below(p) {drop down}
    while key(after(p)) $\leq$ k **do**
        p $\leftarrow$ after(p) {scan forward}

Inserting

- Uses insertAfterAbove(p,q,(k,e)) which inserts (k,e) after p and above q. Also uses the search we just did.

```
p ← SkipSearch(k)
q ← insertAfterAbove(p,null,(k,e))
while random() < ½ do
        while above(p) = null do
                p ← before(p) {scan backwards}
        p ← above(p)          {jump up}
        q ← insertAfterAbove(p,q,(k,e))
```

- NOTE: the outer while loop might run for a very long time!
  - Normal to limit the height to something reasonable….

Deleting

- Removal of an item is easier than insertion
- We simply find the item (SkipSearch(k))
  - Then we remove this from the list S_0
  - We recursively remove it from the lists S_i above this
    - This is simply deletion from a linked list, which is simple...

Skip Lists - Summary

- Towers *can* be more compactly stored
  - No need to store the (k,e) tuple – only need the key
- Height is unbounded, need to limit:
  - $H = 3\lceil \log_2 n \rceil$ is viewed as safe
  - Note that this isn't really an issue as the random number generator will not continue to generate values $\leq ½$
- Search time = Insert time = Delete time = $\log_2(n)$
  - No problems with ordered data being inserted..

Skip Lists - Summary

- The *above* and *before* methods are not needed
  - We can always insert, search and delete using a simple scan-forward-and-down regime
  - For deletion, we will encounter the item at the top of a tower, for example...
- Storage overhead is therefore just a (key,next,down) triple for each tower element...
- Total space requirement is expected to be $<<2n$ entities (all S_i).

Dictionary

- The ADT Dictionary
- Possible Implementations
- Selecting an Implementation
- Hashing

The ADT Dictionary

- Recall concept of a **sort key** in Chapter 11
- Often must search a collection of data for specific information
  - Use a **search key**
- Applications that require value-oriented operations are frequent

The ADT Dictionary

City	Country	Population
Buenos Aires	Argentina	13,170,000
Cairo	Egypt	14,450,000
Cape Town	South Africa	3,092,000
London	England	12,875,000
Madrid	Spain	4,072,000
Mexico City	Mexico	20,450,000
Mumbai	India	19,200,000
New York City	U.S.A.	19,750,000
Paris	France	9,638,000
Sydney	Australia	3,665,000
Tokyo	Japan	32,450,000
Toronto	Canada	4,657,000

Consider the need for searches through this data based on other than the name of the city

- A collection of data about certain cities

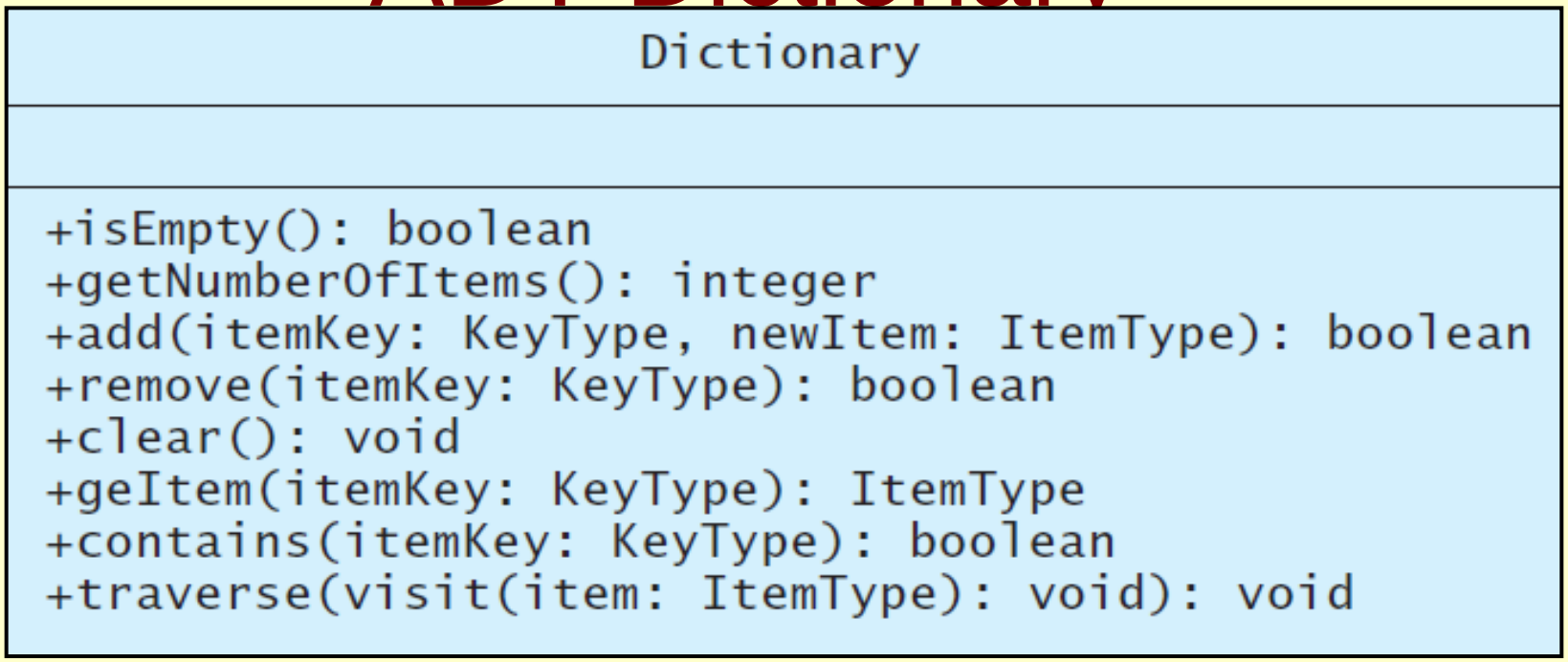
ADT Dictionary Operations

- Test whether dictionary is empty.
- Get number of items in dictionary.
- Insert new item into dictionary.
- Remove item with given search key from dictionary.
- Remove all items from dictionary.

ADT Dictionary Operations

- Get item with a given search key from dictionary.
- Test whether dictionary contains an item with given search key.
- Traverse items in dictionary in sorted search-key order.

ADT Dictionary



- View interface

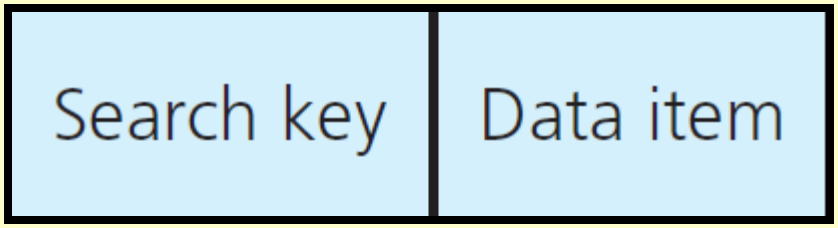
- UML diagram for a class of dictionaries

Possible Implementations

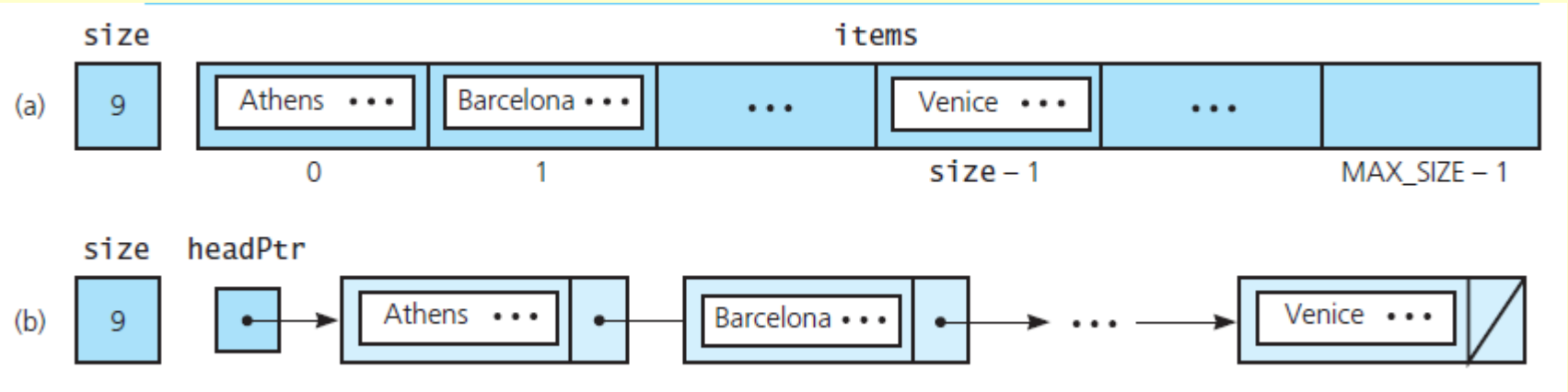
- Sorted (by search key), array-based
- Sorted (by search key), link-based
- Unsorted, array-based
- Unsorted, link-based

Possible Implementations

- A dictionary entry



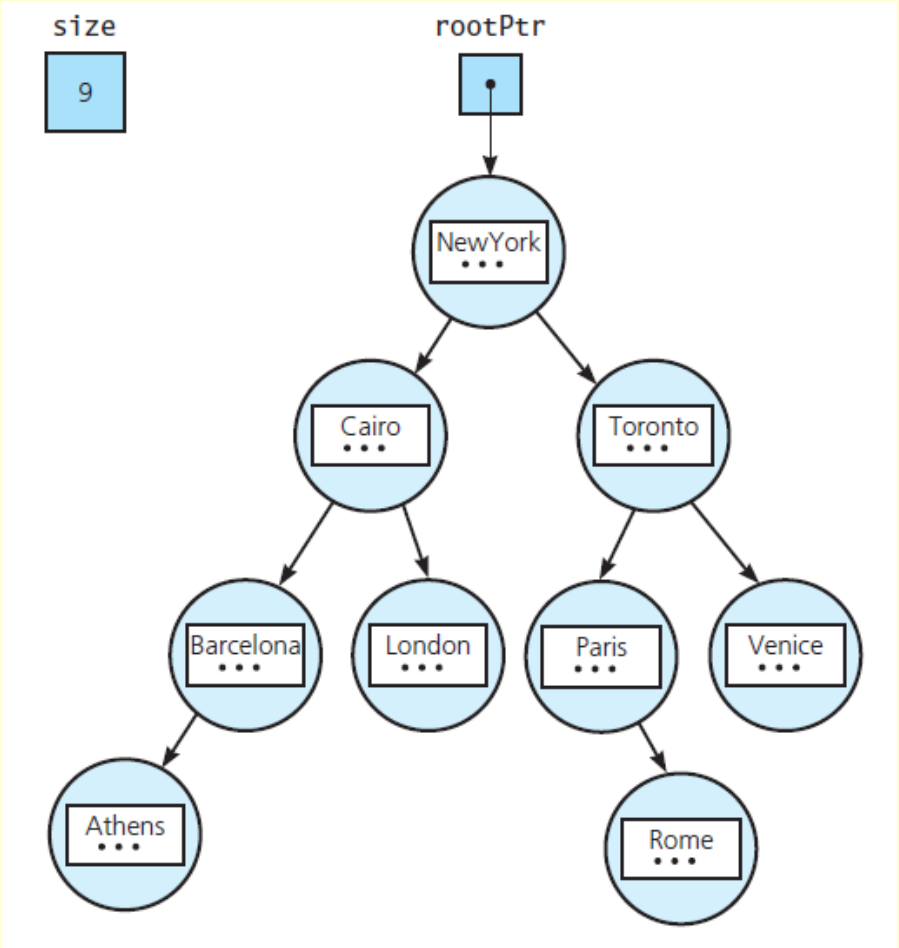
Possible Implementations



- The data members for two sorted linear implementations of the ADT dictionary for the data: (a) array based; (b) link based
- View header file for class of dictionary entries,

Possible Implementations

- The data members for a binary search tree implementation of the ADT dictionary for the data



Sorted Array-Based Implementation of ADT Dictionary

- Consider header file for the class **`ArrayDictionary`**, Listing 18-3

- Note definition of method add, Listing 18-A
  - Bears responsibility for keeping the array **`items`** sorted

Binary Search Tree Implementation of ADT Dictionary

- Dictionary class will use composition
  - Will have a binary search tree as one of its data members
  - Reuses the class **`BinarySearchTree`** from Chapter 16

- View header file, Listing 18-4

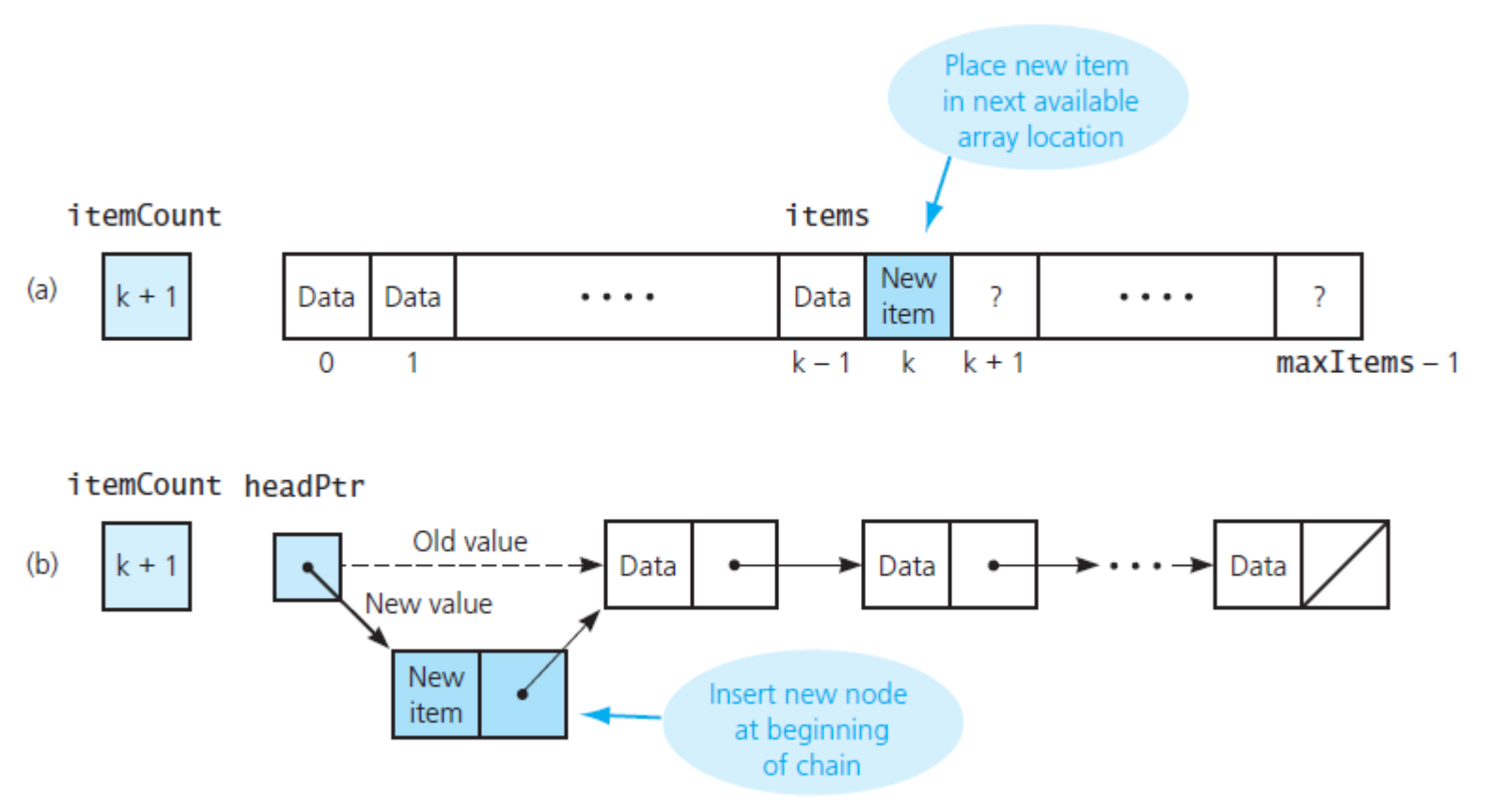
Selecting an Implementation

- Reasons for considering linear implementations
  - Perspective,
  - Efficiency
  - Motivation
- Questions to ask
  - What operations are needed?
  - How often is each operation required?

Selecting an Implementation

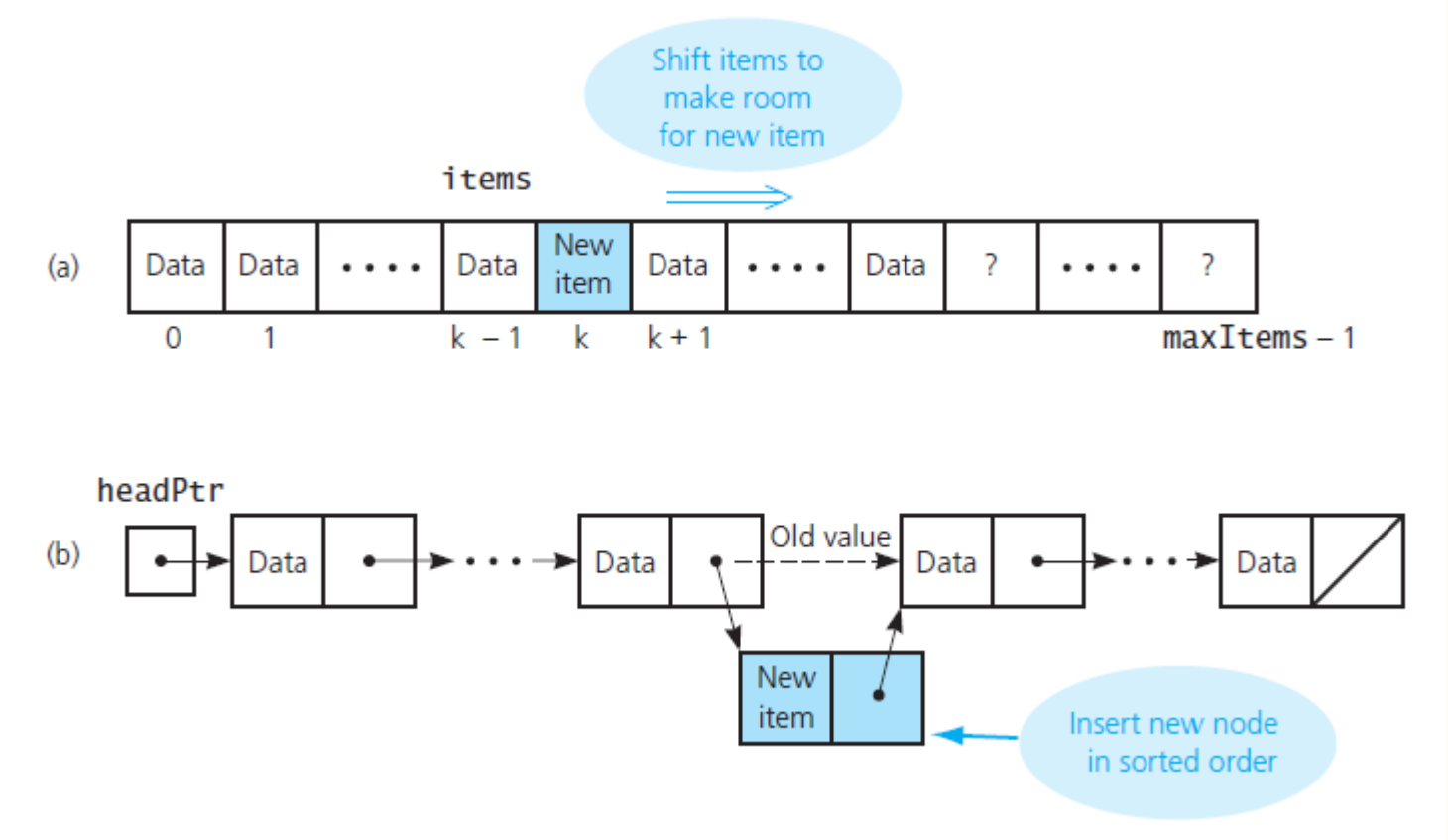
- Three Scenarios
  - Insertion and traversal in no particular order
  - Retrieval – consider:
    - Is a binary search of a linked chain possible?
    - How much more efficient is a binary search of an array than a sequential search of a linked chain?
  - Insertion, removal, retrieval, and traversal in sorted order – add and remove must:
    - Find the appropriate position in the dictionary.
    - Insert into (or remove from) this position.

Selecting an Implementation



- Insertion for unsorted linear implementations: (a) array based; (b) link based

Selecting an Implementation



- Insertion for sorted linear implementations: (a) array based; (b) pointer based

Selecting an Implementation

	Insertion	Removal	Retrieval	Traversal
Unsorted array-based	O(1)	O(n)	O(n)	O(n)
Unsorted link-based	O(1)	O(n)	O(n)	O(n)
Sorted array-based	O(n)	O(n)	O(log n)	O(n)
Sorted link-based	O(n)	O(n)	O(n)	O(n)
Binary search tree	O(log n)	O(log n)	O(log n)	O(n)

- The average-case order of the ADT dictionary operations for various implementations