Dictionaries

- Container of elements from a totally ordered universe that supports the basic operations of
- inserting/deleting elements and searching for a given element.

Sets

- Collection of well defined elements.
- Members of a set are all distinct.
- Set as ADT: operations:

1. Union (A, B, C)
2. Intersection (A, B, C)
3. Difference (A, B, C)
4. Merge (A, B, C)
5. Find (x)
6. Member (x, A) or Search (x, A)
7. Makenull (A)
8. Equal (A, B)
9. Assign (A, B)
10. Insert (x, A)
11. Delete (x, A)
12. Min (A) (if A is an ordered set)

Set implementation:

- Bit Vector
- Array
- Linked List
  - Unsorted
  - Sorted

Dictionaries

- is a dynamic set
- ADT
  1. Makenull (D)
  2. Insert (x, D)
  3. Delete (x, D)
  4. Search (x, D)
- Useful in implementing symbol tables, text retrieval systems, database systems, page mapping tables, etc.

Dictionary Implementation

1. Fixed Length arrays
2. Linked lists : sorted, unsorted, skip-lists
3. Hash Tables : open, closed
4. Trees
   - Binary Search Trees (BSTs)
   - Balanced BSTs
   - AVL Trees
   - Red-Black Trees
   - Splay Trees
   - Multiway Search Trees
   - 2-3 Trees
   - B Trees
   - Tries

Example

- Collection of student records in this class.
- (key, element) = (student name, linear list of assignment and exam scores)
- All keys are distinct.
- Get the element whose key is Amit Goyal.
- Update the element whose key is Mayukh rath.
- put() implemented as update when there is already a pair with the given key.
  - remove() followed by put().

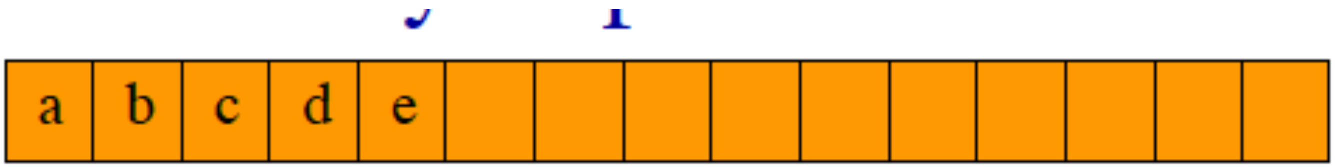
Dictionary With Duplicates

- Keys are not required to be distinct.
- Word dictionary.
  - Pairs are of the form (word, meaning).
  - May have two or more entries for the same word
  - (bolt, a threaded pin)
  - bolt, a crash of thunder)
  - (bolt, to shoot forth suddenly)
  - (bolt, a gulp)
  - (bolt, a standard roll of cloth)

Implementation via Linear List

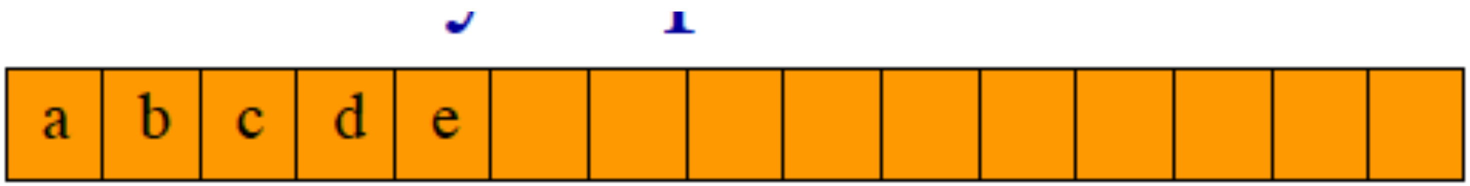
- L = (e0, e1, e2, e3, …, en-1)
- Each ei is a pair (key, element).
- 5-pair dictionary D = (a, b, c, d, e).
- a = (aKey, aElement), b = (bKey, bElement) …
- Array or linked representation.

Through Array



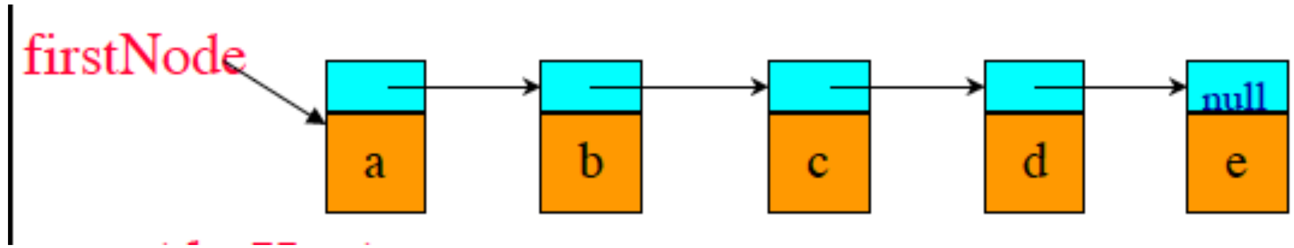
- get(theKey)
  - O(size) time
- put(theKey, theElement)
  - O(size) time to verify duplicate, O(1) to add at right end.
- remove(theKey)
  - O(size) time.

Through Sorted Array



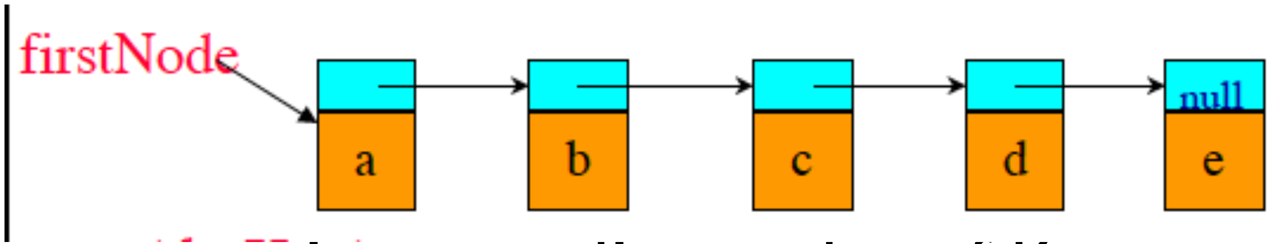
- elements are in ascending order of key.
- get(theKey)
  - O(log size) time
- put(theKey, theElement)
  - O(log size) time to verify duplicate, O(size) to add.
- remove(theKey)
  - O(size) time.

Through unsorted Chain



- get(theKey)
  - O(size) time
- put(theKey, theElement)
  - O(size) time to verify duplicate, O(1) to add at left end.
- remove(theKey)
  - O(size) time.

Through Sorted Chain



- Elements are in ascending order of Key.
- get(theKey)
  - O(size) time (why??)
- put(theKey, theElement)
  - O(size) time to verify duplicate, O(1) to put at proper place. (Why??)
- remove(theKey)
  - O(size) time. (Why??)

Summarizing Performance

- N: number of elements in dictionary D

	Worst case complexity of			
	Search	Delete	Insert	min
Array	O(n)	O(n)	O(n)	O(n)
Sorted List	O(n)	O(n)	O(n)	O(1)
Unsorted List	O(n)	O(n)	O(n)	O(n)

- Sorted list has the best average case performance.

Hash Tables

- Implementations of Dictionary
  - Hash Tables
    - Worst-case time for get, put, and remove is O(size). Expected time is O(1).
  - Skip Lists

Hashing

- Open Hashing:
  - Keys are stored in linked lists attached to cells of a hash table
- Closed Hashing:
  - All keys are stored in the hash table itself without the use of linked lists.
- Hash Functions:
  - Hash Code Maps
  - Compression Maps

Good Hash Functions

- Fast to Compute
- Uniform distribution
- Good Hash Functions – Rare
  - Look at Birthday Paradox

Hashing Code Maps on non-integers

- Turn key to Integers: 9835-467→ 9835467
- Strings:
  - Value of Ascii values
  - Polynomial code maps
  - Add up (is it good?)
- Numerics types
  - Find functions to map to integers
  - Long, double ---
  - Is adding words good?

- Polynomial Code Maps
- a0+ X a1+ … + X^2 a2 … + X^ (n-1) a(n-1)
- Use Hornets Rule
- A1 + X (a1 + X(a2+ … + X(a(n-2) + X a (n-1))…)
- Choice X = 33, 37, 39, 41
  - At most 6 collisions on vocabulary of 500000

Compression Maps

- H (k) = k mod m --- use remainder
  - m = b^e -- bad
    - M is power of 2. h(k) gives e least significant bits of k
    - All keys with the same ending go to the same value
    - M prime good (not too close to power of 20
      - Helps uniform distribution
- Floor function :
  - H(k) = Floor (m (k A mod 1)); 0 < A < 1
  - Value of m is not critical (could be 2^p)
  - Optimal choice depends on data
  - Knuth – A → square_root(5) -1/2 – conjugate of Golden ration; Fibonacci Hashing
- Multiply, divide – see later

Ideal Hashing

- Uses a 1D array (or table) table[0:b-1].
- Each position of this array is a bucket.
  - A bucket can normally hold only one dictionary pair.
- Uses a hash function f that converts each key k into an index in the range [0, b-1].
  - f(k) is the home bucket for key k.
- Every dictionary pair (key, element) is stored in its home bucket table[f[key]].

Example

- Pairs are: (22,a), (33,c), (3,d), (73,e), (85,f).
- Hash table is table[0:7], b = 8.
- Hash function is key/11.
- Pairs are stored in table as below:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- get, put, and remove take O(1) time.

Problem

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- get, put, and remove take O(1) time.

- Insert (26,g): Where does (26,g) go?
- Keys that have the same home bucket are synonyms.
  - 22 and 26 are synonyms with respect to the hash function that is in use.
  - The home bucket for (26,g) is already occupied.

Collision

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)

- A collision occurs when the home bucket for a new pair is occupied by a pair with a different key.
- An overflow occurs when there is no space in the home bucket for the new pair.
- When a bucket can hold only one pair, collisions and overflows occur together.
- Need a method to handle overflows.

Issues with Hash

- Choice of hash function.
- Overflow handling method.
- Size (number of buckets) of hash table.

Open Hashes

U be the universe of keys:

- Integers
- character strings
- – complex bit patterns

B the set of hash values (or buckets or bins).

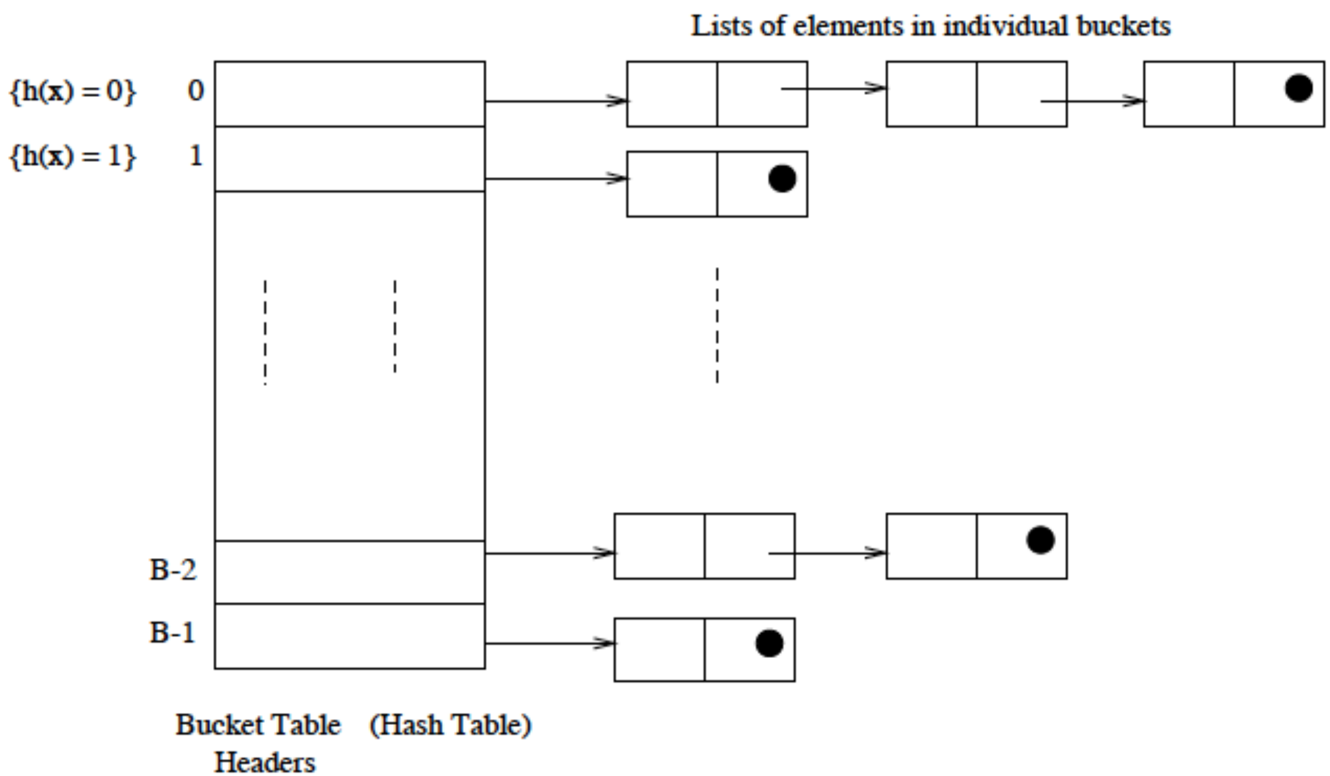
- Let B = {0, 1, . . . ,m − 1} where m > 0 is a positive integer.

Open Hashing

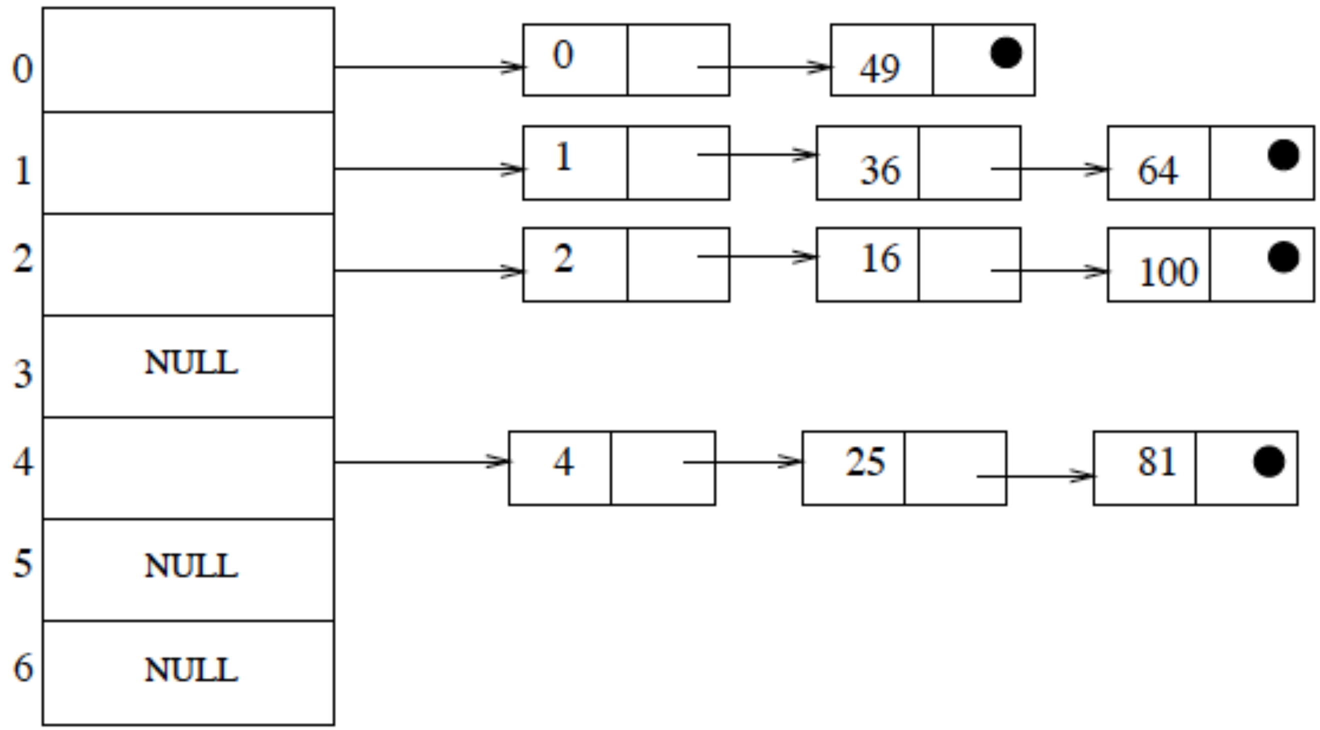
A hash function $h : U \rightarrow B$ associates buckets (hash values) to keys.

- Two issues:
- Collisions
  - If x1 and x2 are two different keys, if $h(x1) = h(x2)$, it is called **collision**. Collision resolution is the most important issue in hash table implementations.
- Hash Functions
  - Choosing a hash function that minimizes the number of collisions and also hashes uniformly is another critical issue.
- Collision Resolution : By Chaining

Collision Resolution : Chaining

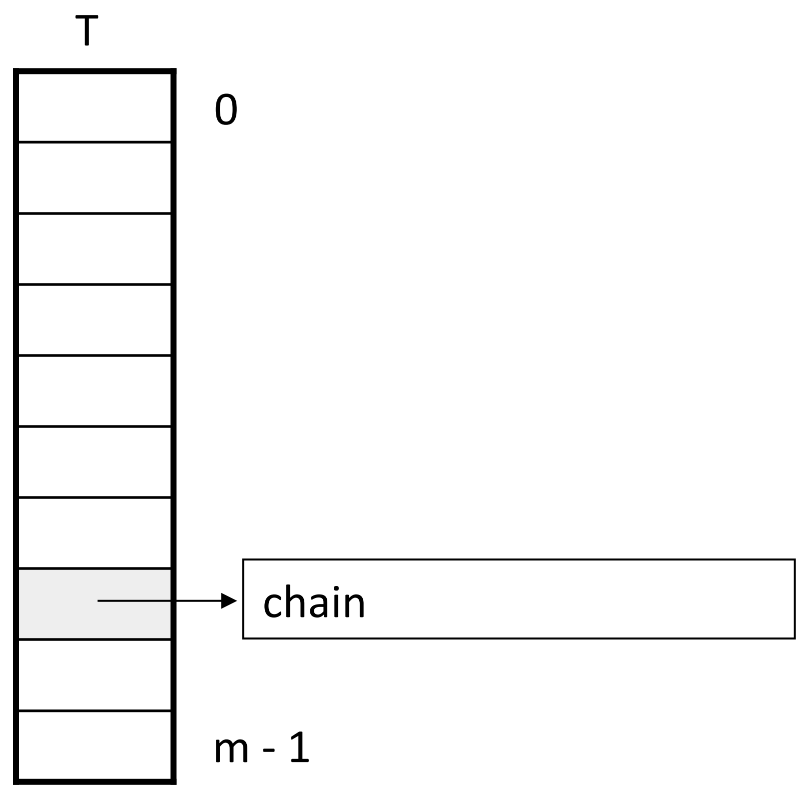


Open hashing: An example



Analysis of Hashing with Chaining: Worst Case

- How long does it take to search for an element with a given key?
- Worst case:
  - All n keys hash to the same slot
  - Worst-case time to search is $\Theta(n)$, plus time to compute the hash function



Analysis of Hashing with Chaining: Average Case

- Average case
  - depends on how well the hash function distributes the n keys among the m slots
- **Simple uniform hashing** assumption:
  - Any given element is equally likely to hash into any of the m slots (i.e., probability of collision Pr(h(x)=h(y)), is 1/m)
- Length of a list:

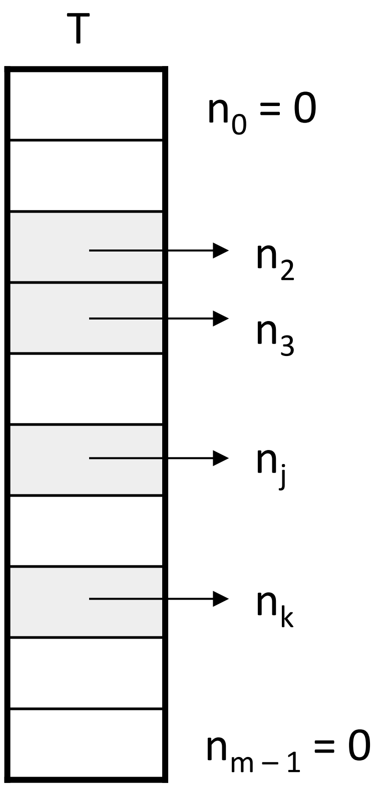
$$T[j] = n_j, \qquad j = 0, 1, \ldots, m - 1$$

- Number of keys in the table:

$$n = n_0 + n_1 + \cdots + n_{m-1}$$

- Average value of n_j:

$$E[n_j] = \alpha = n/m$$



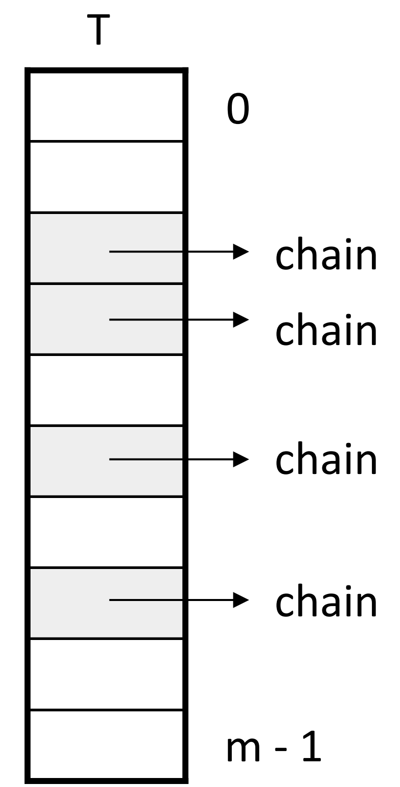
Load Factor of a Hash Table

- Load factor of a hash table T:

$$\alpha = n/m$$

  - n = # of elements stored in the table
  - m = # of slots in the table = # of linked lists

- α encodes the average number of elements stored in a chain
- α can be <, =, > 1



Chains

Successful search (Insert beginning vs end)

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\underbrace{\frac{i-1}{m}}\right) \leftarrow$$ Expected No. of Elements examined

expected length of the list on inserting the $(i-1)$th element

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

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

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

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

Note: In case of a successful search, E(# elements) examined is 1 more than the no. of elements examined when the sought for element was inserted.

Considering the time for the hash function

$$\Theta\left(2+\frac{\alpha}{2}-\frac{1}{2m}\right) = \Theta(1+\alpha)$$

Average Case Analysis

- Let the no. of slots be proportional to the no. of elements in the Table
- $n = O(m)$
- $\alpha = n/m = O(m)/n = O(1)$
- Searching takes a constant time on the average
- Insertion takes $O(1)$ worst case
- Deletion takes $O(1)$ worst-case time when the lists are doubly linked list

Case 1: Unsuccessful Search (i.e., item not stored in the table)

Theorem

An unsuccessful search in a hash table takes expected time $\Theta(1+\alpha)$

Under the assumption of simple uniform hashing

(i.e., probability of collision Pr(h(x)=h(y)), is 1/m)

Proof

- Searching unsuccessfully for any key k
  - need to search to the end of the list T[h(k)]
- Expected length of the list:
  - $E[n_{h(k)}] = \alpha = n/m$
- Expected number of elements examined in an unsuccessful search is α
- Total time required is:
  - O(1) (for computing the hash function) + α → $\Theta(1+\alpha)$

Case 2: Successful Search

Successful search: $\Theta(1 + \frac{a}{2}) = \Theta(1 + a)$ time on the average

(search half of a list of length a plus $O(1)$ time to compute $h(k)$)

Analysis of Search in Hash Tables

- If m (# of slots) is proportional to n (# of elements in the table):
- $n = O(m)$
- $\alpha = n/m = O(m)/m = O(1)$

$\Rightarrow$ Searching takes constant time on average

Example

- Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100.
- Hash Function: x % 7

Complexity

- Bucket lists
  - unsorted lists
  - sorted lists (these are better)
- Insert (x, T)
  - Insert x at the head of list T[h(key (x))]
- Search (x, T)
  - Search for an element x in the list T[h(key (x))]
- Delete (x, T)
  - Delete x from the list T[h(key (x))]
- Worst case complexity of all these operations is O(n)
  - It is assumed that the h(k) can be computed in O(1) time.
  - In the average case, the running time is $O(1 + \alpha)$
- n = number of elements stored
- m = number of hash values or buckets
- Load factor = α = n/m
- If n is O(m), average case complexity of these operations becomes O(1) !

Closed Hashing

- All elements are stored in the hash table itself
- Avoids pointers; only computes the sequence of slots to be examined.
- Collisions are handled by generating a sequence of rehash values.

$$h : \underbrace{U}_{\text{universe of primary keys}} \times \underbrace{\{0,1,2,...\}}_{\text{probe number}} \rightarrow \{0,1,2,...,m-1\}$$

Closed Hashing

- Given a key x, it has a hash value h(x,0) and
  - a set of rehash values h(x, 1), h(x,2), . . . , h(x, m-1)
- We require that for every key x, the probe sequence
  - < h(x,0), h(x, 1), h(x,2), . . . , h(x, m-1)> be a permutation of <0, 1, ..., m-1>.
  - This ensures that every hash table position is eventually considered as a slot for storing a record with a key value x.

Closed Hashing

- Search (x, T) : Search will continue until you find the element x (successful search) or an empty slot (unsuccessful search).
- Delete (x, T): No delete if the search is unsuccessful; If the search is successful, then put the label DELETED (different from an empty slot).
- Insert (x, T): No need to insert if the search is successful. If the search is unsuccessful, insert at the first position with a DELETED tag.

Handling Overflows

- An overflow occurs when the home bucket for a new pair (key, element) is full.
- Handle overflows by:
  - Search the hash table in some systematic fashion for a bucket that is not full.
  - Linear probing (linear open addressing).
  - Quadratic probing.
  - Random probing.
- Eliminate overflows by permitting each bucket to keep a list of all pairs for which it is the home bucket.
  - Array linear list.
  - Chain

Rehashing Methods

Denote h(x, 0) by simply h(x).

1. Linear probing:
   – $h(x, i) = (h(x) + i) \bmod m$
2. Quadratic Probing
   – $h(x, i) = (h(x) + C_1 i + C_2 i^2) \bmod m$ where C_1 and C_2 are constants.
3. Double Hashing $h(x,i) = (h(x) + i \underbrace{h'(x))}_{\text{another hash function}} \bmod m$

Linear Probing – Get And Put

- divisor = b (number of buckets) = 17.
- Home bucket = key % 17.

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- Put in pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

Linear Probing – Remove

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- remove(0)

0				4				8				12				16
34		45				6	23	7			28	12	29	11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
34	45					6	23	7			28	12	29	11	30	33

Linear Probing – remove(34)

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
	0	45				6	23	7			28	12	29	11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
0		45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
0	45					6	23	7			28	12	29	11	30	33

Linear Probing – remove(29)

34	0	45				6	23	7			28	12	29	11	30	33

0				4				8				12				16
34	0	45				6	23	7			28	12		11	30	33

- Search cluster for pair (if any) to fill vacated bucket.

0				4				8				12				16
34	0	45				6	23	7			28	12	11		30	33

0				4				8				12				16
34	0	45				6	23	7			28	12	11	30		33

0				4				8				12				16

Performance – Linear Probing

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- Worst-case get/put/remove time is Theta(n), where n is the number of pairs in the table.
- This happens when all pairs are in the same cluster.

Expected Performance

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

- alpha = loading density = (number of pairs)/b.
  - alpha = 12/17.
- S_n = expected number of buckets examined in a successful search when n is large
- U_n = expected number of buckets examined in a unsuccessful search when n is large
- Time to put and remove governed by U_n.

Expected Performance

- $S_n \sim ½(1 + 1/(1 – alpha))$
- $U_n \sim ½(1 + 1/(1 – alpha)^2)$
- Note that 0 <= alpha <= 1.

alpha	S_n	U_n
0.50	1.5	2.5
0.75	2.5	8.5
0.90	5.5	50.5

Alpha <= 0.75 is recommended.

Hash Table Design

•

- Performance requirements are given, determine maximum permissible loading density.
- We want a successful search to make no more than 10 compares (expected).
- Sn ~ 1/2(1 + 1/(1 – alpha))
- alpha <= 18/19
- We want an unsuccessful search to make no more than 13 compares (expected).
- Un ~ 1/2(1 + 1/(1 – alpha)2)
- .alpha <= 4/5
- So alpha <= min{18/19, 4/5} = 4/5.

Hash Table Design

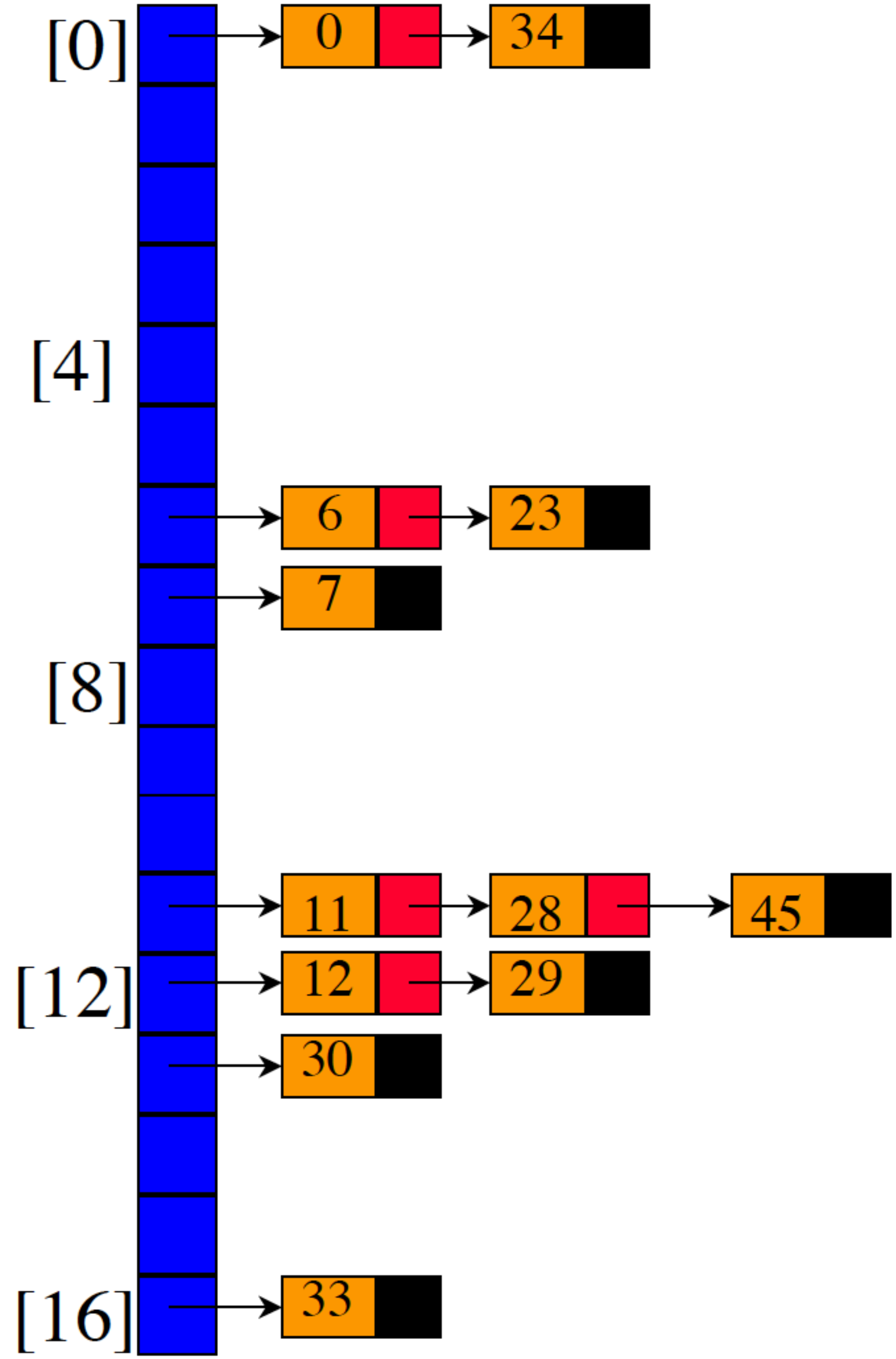
- Dynamic resizing of table.
  - Whenever loading density exceeds threshold (4/5 in our example), rehash into a table of approximately twice the current size.
- Fixed table size.
- Know maximum number of pairs.
- No more than 1000 pairs.
- Loading density <= 4/5 => b >= 5/4*1000 = 1250.
- Pick b (equal to divisor) to be a prime number or an odd number with no prime divisors smaller than 20.

Linear List Of Synonyms

- Each bucket keeps a linear list of all pairs for which it is the home bucket.
- The linear list may or may not be sorted by key.
- The linear list may be an array linear list or a chain.

Sorted Chains

- Put in pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45
- Home bucket = key % 17.



Example 1

- Linear probing
- $h(x, 0) = x\%7$
- $h(x, i) = (h(x, 0) + i)\%7$
- Start with an empty table

Insert (20, T)
Insert (30, T)
Insert (9, T)
Insert (45, T)
Insert (14, T)

0	14
1	empty
2	30
3	9
4	45
5	empty
6	20

Search (35, T)
Delete (9, T)

0	14
1	empty
2	30
3	deleted
4	45
5	empty
6	20

Search (45, T)
Search (52, T)
Search (9, T)
Insert (45, T)
Insert (10, T)

0	14
1	empty
2	30
3	10
4	45
5	empty
6	20

Delete (45, T)
Insert (16, T)

0	14
1	empty
2	30
3	10
4	16
5	empty
6	20

Example 2

Let m be the number of slots.

Assume :
- every even numbered slot occupied and every odd numbered slot empty
- any hash value between 0 . . . m-1 is equally likely to be generated.
- linear probing

empty
occupied
empty
occupied
empty
occupied
empty
occupied

Expected number of probes for a successful search = 1

Expected number of probes for an unsuccessful search

$$= \left(\frac{1}{2}\right)(1) + \left(\frac{1}{2}\right)(2)$$

$$= 1.5$$

Comparison of Rehashing

$\boxed{\text{Linear Probing}}$	m distinct probe sequences	Primary clustering
$\boxed{\text{Quadratic Probing}}$	m distinct probe sequences	No primary clustering; but secondary clustering
$\boxed{\text{Double Hashing}}$	m^2 distinct probe sequences	No primary clustering No secondary clustering

A Uniform Hash Function

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Let keySpace be the set of all possible keys.
- A uniform hash function maps the keys in keySpace into buckets such that approximately the same number of keys get mapped into each bucket

Uniform Hash Function

(3,d)		(22,a)	(33,c)			(73,e)	(85,f)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Equivalently, the probability that a randomly selected key has bucket i as its home bucket is

1/b,0 <= i < b.

- A uniform hash function minimizes the likelihood of an overflow when keys are selected at random.

What is a good function

- Should satisfy the simple uniform hashing property.
- Let U = universe of keys
- Let the hash values be 0, 1, . . . , m-1

Let us assume that each key is drawn independently from U according to a probability distribution P. i.e., for k $\in$ U

$$P(k) = \text{ Probability that } k \text{ is drawn}$$

Then simple uniform hashing requires that

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \text{ for each } \; j = 0, 1, \ldots, m-1$$

that is, each bucket is equally likely to be occupied.

Simple Hash Function with Uniform hashing Property

- Suppose the keys are known to be random real numbers k independently and uniformly distributed in the range [0,1).

$$h(k) = \lfloor km \rfloor$$

  - satisfies the simple uniform hashing property

Use of Qualitative information about P

- For example, consider a compiler's symbol table in which the keys are arbitrary character strings representing identifiers in a program. It is common for closely related symbols,
  - say pt, pts, ptt, to appear in the same program.
- A good hash function would minimize the chance that such variants hash to the same slot

A Common Approach

- Derive a hash value in a way that is expected to be independent of any patterns that might exist in the data.
- The division method computes the hash value as the remainder when the key is divided by a prime number.
  - Unless that prime is somehow related to patterns in the distribution P, this method gives good results.

Division Method

A key is mapped into one of m slots using the function

$$h(k) = k \bmod m$$

- Requires only a single division, hence fast
- m should not be :
  - a power of 2, since if $m = 2^p$, then $h(k)$ is just the p lowest order bits of k
  - a power of 10, since then the hash function does not depend on all the decimal digits of k
  - $2^p - 1$. If k is a character string interpreted in radix 2^p, two strings that are identical except for a transposition of two adjacent characters will hash to the same value.
- Good values for m

primes not too close to exact powers of 2.

Selecting The Divisor

- Because of this correlation, applications tend to have a bias towards keys that map into odd integers (or into even ones).
- When the divisor is an even number, odd integers hash into odd home buckets and even integers into even home buckets.
  - 20%14 = 6, 30%14 = 2, 8%14 = 8
  - 15%14 = 1, 3%14 = 3, 23%14 = 9
- The bias in the keys results in a bias toward either the odd or even home buckets.

Selecting the Divisor

- When the divisor is an odd number, odd (even) integers may hash into any home.
  - 20%15 = 5, 30%15 = 0, 8%15 = 8
  - 15%15 = 0, 3%15 = 3, 23%15 = 8
- The bias in the keys does not result in a bias toward either the odd or even home buckets.
- Better chance of uniformly distributed home buckets.
- So do not use an even divisor.

Selecting The Divisor

- Similar biased distribution of home buckets is seen, in practice, when the divisor is a multiple of prime numbers such as 3, 5, 7,…
- The effect of each prime divisor p of b decreases as p gets larger.
- Ideally, choose b so that it is a prime number.
- Alternatively, choose b so that it has no prime factor smaller than 20.

Multiplication Method

Two Steps

1. Multiply the key k by a constant A in the range $0 < A < 1$ the fractional part of kA

2. Multiply this fractional part by m and take the floor.

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

where

$$kA \bmod 1 = kA - \lfloor kA \rfloor$$
$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

- Advantage of the method is that the value of m is not critical. We typically choose it to be a power of 2:

$$m = 2^p$$

for some integer p so that we can then easily implement the function on most computers as follows:

Suppose the word size = w. Assume that k fits into a single word. First multiply k by the w-bit integer $\lfloor A.2^w \rfloor$. The result is a $2w$ - bit value

$$r_1 2^w + r_0$$

where r_1 is the high order word of the product and r_0 is the low order word of the product. The desired p-bit hash value consists of the p most significant bits of r_0.

- Works practically with any value of A, but works better with some values than the others. The optimal choice depends on the characteristics of the data being hashed. Knuth recommends

$$A \simeq \frac{\sqrt{5}-1}{2} = 0.6180339887\ldots \quad \text{(Golden Ratio)}$$

Universal Hashing

- Involves choosing a hash function randomly in a way that is independent of the keys that are actually going to be stored.
- Select the hash function at random from a carefully designed class of functions.

  - Let Φ be a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, 2, \ldots, m-1\}$.
  - Φ is called **universal** if for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in \Phi$ for which $h(x) = h(y)$ is precisely equal to

$$\frac{|\Phi|}{m}$$

- With a function randomly chosen from Φ, the chance of a collision between x and y where $x \neq y$ is exactly $\frac{1}{m}$.

Example: Universal class of hash functions

Let table size m be prime. Decompose a key x into r +1 bytes. (i.e., characters or fixed-width binary strings). Thus

$$x = (x_0, x_1, \ldots, x_r)$$

Assume that the maximum value of a byte to be less than m.

Let $a = (a_0, a_1, \ldots, a_r)$ denote a sequence of $r + 1$ elements chosen randomly from the set $\{0, 1, \ldots, m-1\}$. Define a hash function $h_a \in \Phi$ by

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m$$

With this definition, $\Phi = \sqcup_a \{h_a\}$ can be shown to be universal. Note that it has m^{r+1} members.

Hash Table Restructuring

- When a hash table has a large number of entries (say $n \geq 2m$ in open hash table or $n \geq 0.9m$ in closed hash table), the average time for operations can become quite substantial.
  - one idea is to simply create a new hash table with more number of buckets (say twice or any appropriate large number); the currently existing elements will have to be inserted into the new table.
  - May call for rehashing of all these key values transferring all the records
  - Effort will be less than it took to insert them into the original table.
- Subsequent dictionary operations will be more efficient and can more than make up for the overhead in creating the larger table.