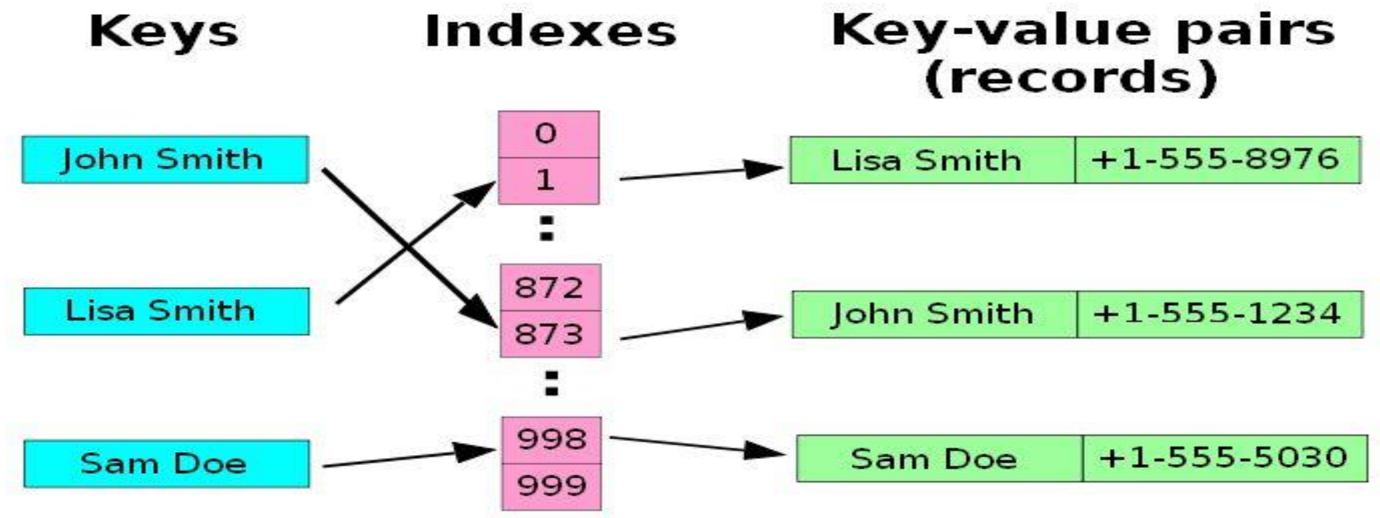
Unit 6-Part 2

Hashing

Dictionary

- A general purpose data structure that supports operations insert, delete and search
- Consists of key-value pairs
- Key must be unique (set)
- Values may not be unique (list)

Example 1



A small phone book as a Dictionary

Example 2

Example: Symbol Table

Name	Type	Offset	Scope
inGlobal	int	0	global
inLocal	int	0	main
outLocalA	int	-1	main
outLocalB	int	-2	main

Dictionary ADT operations

- **create()** – creates empty dictionary
- **put(Dictionary d, Key k, value v)** – Inserts a key value pair. If a key is already present then?? Old value is replaced by new value
- **value get (d,k)** – returns a value associated with the key k or null, if dictionary contains no such key
- **remove()**- removes key k & associated value
- **destroy()**- destroys dictionary d

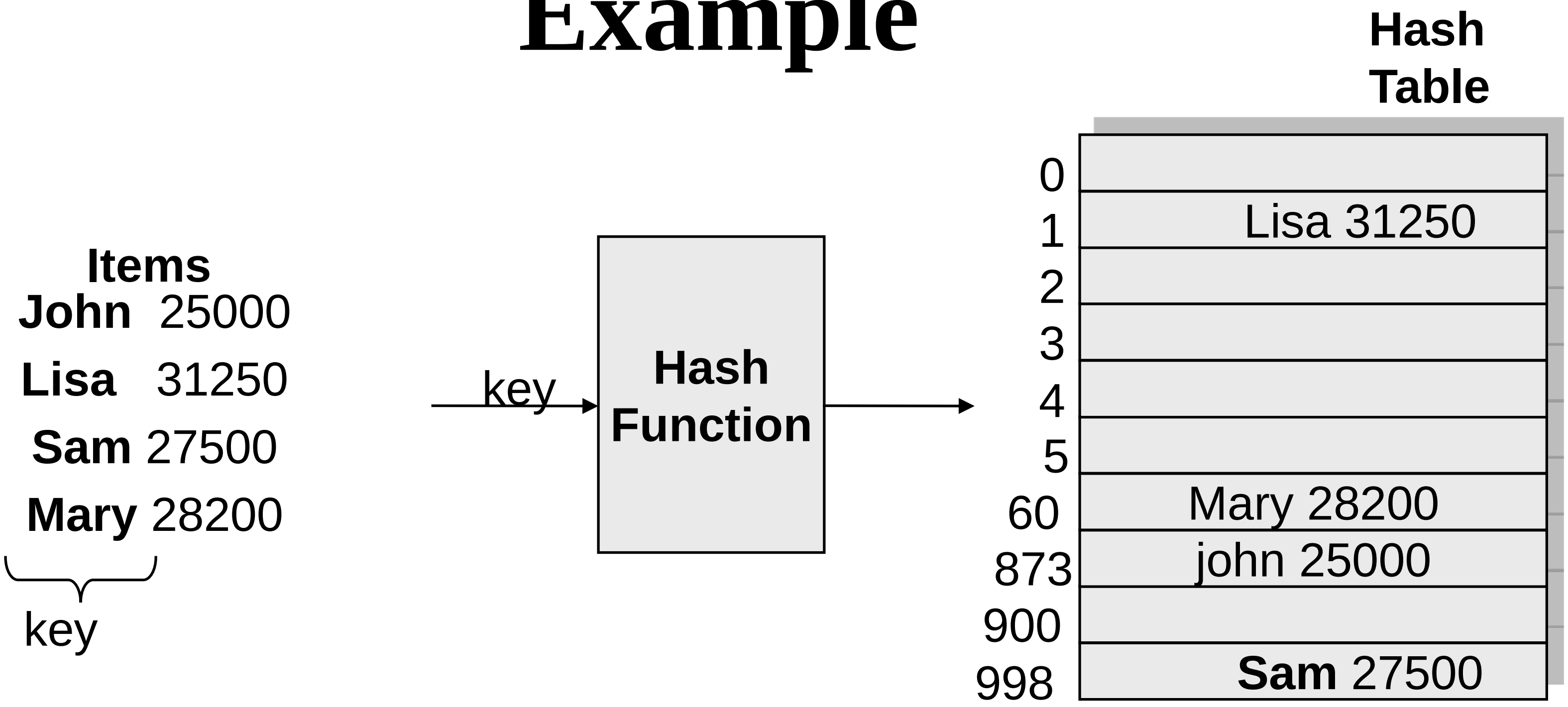
Implementation of Dictionary

Data Structure	Search	Insert	Delete
BST	O(n)	O(n)	O(n)
AVL	O(log n)	O(log n)	O(log n)
2-3 Tree	O(log n)	O(log n)	O(log n)
Red Black Tree	O(log n)	O(log n)	O(log n)
B-Tree	O(log n)	O(log n)	O(log n)
Hash Table	**O(1)**	**O(1)**	**O(1)**
Array	O(log n)	O(n)	O(n)
Linked List	O(n)	O(1)	O(n)

Concept of Hashing

- Hashing means mapping a data into a fixed-size value applying a function (Hash Function)
- A **hash table**, is a data structure that associates **keys** (names) with **values** (attributes)
- Hashing technique is used to perform insert, delete and search in constant average time

Example


Hash Table
Items
John 25000
Lisa 31250
Sam 27500
Mary 28200
key
key
Hash Function
0
1
2
3
4
5
60
873
900
998
Lisa 31250
Mary 28200
john 25000
Sam 27500

Hash Table

- Hash table :
  - Collection of key-value pairs
  - Hash function

Hash Tables

Idea:

- Use a **function h** to compute the **slot for each key**
- **Store** the element in slot h(k)

- A **hash function** h transforms a key into an index in a hash table T[0…m-1]:
- We say that k **hashes** to slot h(k)
- A hash table maps a larger key space to a smaller table space (Ex. A few of 5-character length keys generated from 26 alphabets; to be stored in a table of size 50; Now compare the size of keyset with size of hash table)

Hashing cannot be used for

- Not efficient in operations that require
- Any ordering information among the elements such as
- **FindMin**
- **FindMax**
- **Printing the entire table in sorted order**

Choice of Hash Function

- Requirements
  - easy to compute
  - minimal number of collisions
- If a hashing function groups key values together, this is called clustering of the keys
- A good hashing function distributes the key values uniformly throughout the range

Some Hash functions

- Division modulo /remainder method
- Middle of square (Mid Square)
- Folding
- Digit Analysis Method

The Division Modulo/Remainder Method

- **Idea:**
  - Map a key k into one of the m slots by taking the remainder of k divided by m

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

Ex. K=3205, m=97

H(3205)=3205 mod 97 = 4

Here, **m is a prime number close to the size of hash table (less collision)**

Avoid taking m as a power of 2 (if m=2^p then, then h(k) is p lower order bits of k)

Example

- Which of the following is the correct hash function to have a range 1 to 23?
  – X mod 23
  – (x+1) mod 23
  – X mod (23+1)
  – (X mod 23)+1

Example

- Keys : 14,23,11,15,19,65,73,12,9
- Use division modulo method
- For
  - m=5
  - m=7
  - m=13

Observation for no. of collisions??

m=5
m=7
m=13

The Mid Square Method

- **Idea:**

Map a key k into one of the m slots by taking the middle digits of square of a key

h(k):= return middle digits of k^2

Example : k=3205 k^2=10272025

H(3205)=72 (Middle 2 digits)

Or

H(3205)=22 (2nd and 4th digit from right)

- The method is used to build symbol table in compiler

The Folding Method

- **Idea:**

Map a key k into one of the m slots by **partitioning it into several parts**, and **add the parts together** to obtain the hash address

$$h(k) = k1+k2+\ldots kn$$

Here, each part has same no. of digits as the required address, except the last part

e.g. x=12320324111220; partition x into 123,203,241,112,20; then return the address 123+203+241+112+20=699

Example

- x=12320324111220; partition x into 123,203,241,112,20; then return the address
- **Pure folding**
  - 123+203+241+112+20=699
- **Fold shifting**
  - 321+203+142+112+02=780
- **Fold boundary**
  - 321+203+241+112+02=879

Digit Analysis Method

- The hash value is formed by extracting and manipulating specific digits from the key
- Ex, key = 1234567 , select digits in position 2 to 4 i.e. 234 followed by reversing, swapping, circular shifting, etc.
- Selection of positions depends on the uniform distribution of all digits in a particular position

Example

	Key Position									
Digit	1	2	3	4	5	6	7	8	9	10
0	5000	531	594	499	590	721	1565	1133	562	2540
1	0	582	568	536	467	905	874	759	612	1581
2	0	571	620	531	563	553	657	606	542	557
3	0	546	565	511	512	277	555	482	522	332
4	0	518	529	495	461	0	284	521	546	0
5	0	503	503	500	463	673	276	469	472	0
6	0	488	456	469	510	629	263	296	426	0
7	0	449	411	500	459	0	212	365	425	0
8	0	422	431	470	457	501	159	310	455	0
9	0	390	323	489	518	741	155	59	438	0

- If key=1234567890
- Then, select **9542**
- (i.e. select digits at position 2,4,5 and 9 and reverse them)

- Red coloured circles are peaks and valleys which distort uniform distribution and hence the positions 1,3,6,7,8,10 are not a good choice)

Collisions

Two or more keys hash to the same slot!!

- For a given set K of keys
  - If $|K| \leq m$, collisions may or may not happen, depending on the **hash function**
- Avoiding collisions completely is hard, even with a good hash function (see birthday paradox)

Handling Collisions

- Collision handling techniques
  - Chaining (Open hashing)
  - Open addressing (Closed hashing)
    - Linear probing
    - Quadratic probing
    - Double hashing

Open Addressing

- **If we have enough contiguous memory to store all the keys (m > N) $\Rightarrow$ store the keys in the table itself**
- No need to use linked lists
- Basic idea:
  - Insertion: if a slot is full, try another one,
    until you find an empty one
  - Search: follow the same sequence of probes
- Search time depends on the length of the probe sequence

Linear probing (linear open addressing)

- **Linear Probing** resolves collisions by placing the data into the next open slot in the table

Linear probing: Inserting a key

- Idea: when there is a collision, check the next available position in the table (i.e., probing)

$$h(k,i) = (h(k) + i) \ [\mathrm{mod}\ m]$$

$$i = 0, 1, 2, \ldots (m-1)$$

- First slot probed: h(k)
- Second slot probed: h(k) + 1
- Third slot probed: h(k)+2, and so on

  probe sequence: < h(k), h(k)+1 , h(k)+2 , ...>

Linear Probing – Example

- 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

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

Problem of Linear Probing

- Identifiers tend to cluster together (**primary clustering**:: Long chunks of occupied slots are created)

- Worst case complexity for insert, delete and search is O(n)

Quadratic Probing

Quadratic probing searches buckets **$(h(k)+i^2)$, $i=0,1,2,\ldots(m-1)/2$**

- Instead of storing addresses in h,h+1,h+2, etc. it stores in h, h+1, h+4, h+9, etc.
  - ***h(k)=(h(k)+i^2)%table size***

Problem of Quadratic Probing

- O(n) complexity - worst case searching time
- When table gets more than half full, there is no guarantee of finding an empty cell
- Suffers from secondary clustering – Keys with the same initial hash value generate the same probe sequence
- Homework
  - If table size is 16 (power of 2), then calculate the possible locations of probes

Double Hashing

$$h(i,k) = (h_1(k) + i \cdot h_2(k)) \bmod |T|.$$

- Use a **secondary hash of the key as an offset** when a collision occurs
- The 2nd hash function $h_2(k)$ is dependent on the key, unlike linear and quadratic probing
  - it should never yield an index of zero
  - it should cycle through the whole table
  - it should be very fast to compute

Double Hashing

- ***h(i,k) = (h1(k) + (i* (h-k mod h))) mod T***
- **h is a prime number** less than T and greater than 2
- **i** is a variable ranging from 0-h (increments by 1 each time)
- [Example
- Insert the keys 45,98,12,55,46,89,65,88,36,21 in an initially empty hash table of size 11
- Use double hashing for collision resolution

Advantages of Double Hashing

- More efficient as compared to linear and quadratic probing
- Alleviates the problem of clustering
- Guarantee to find an empty slot (if h2(k) is co-prime with Table size *T*)

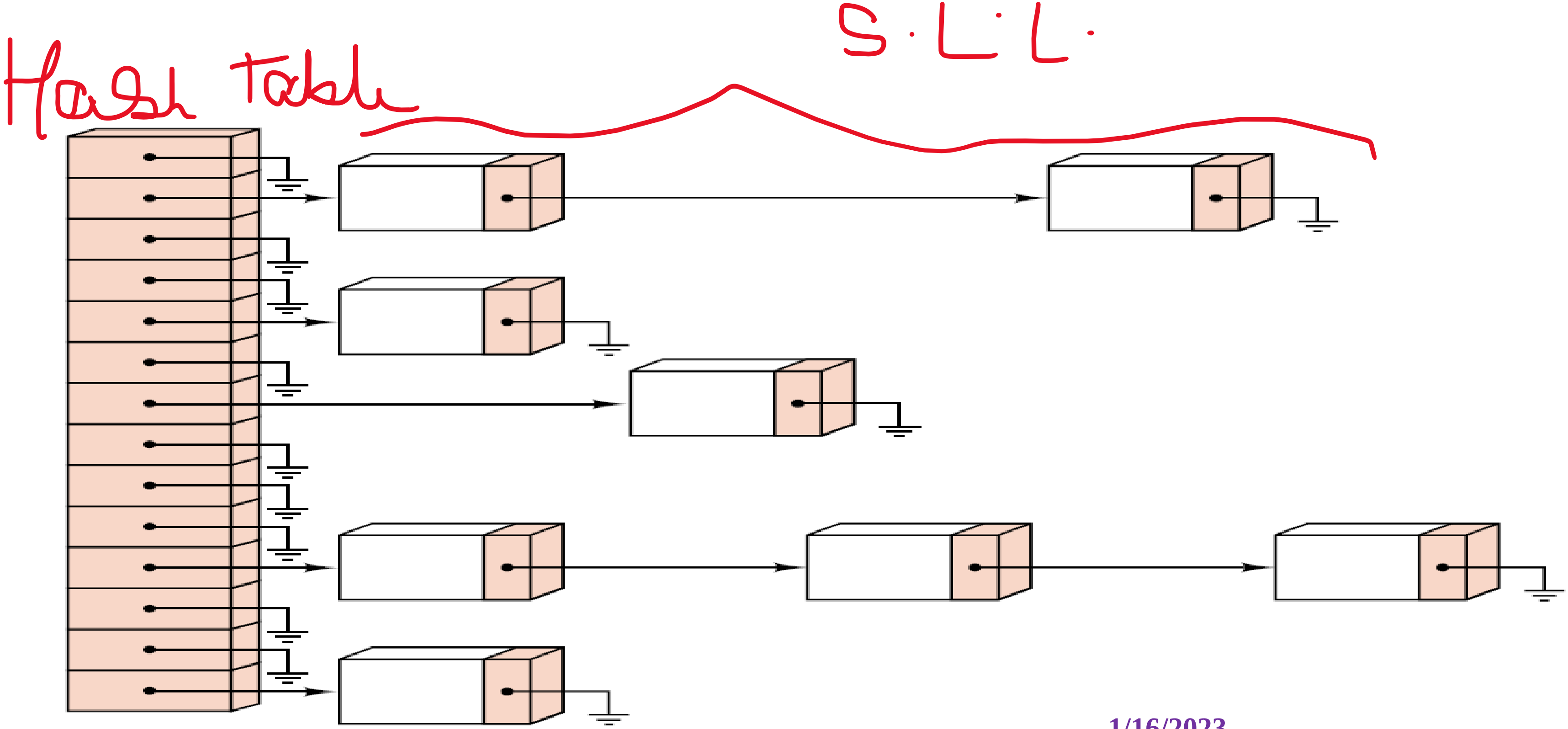
Drawbacks of closed hashing

- As half of the table is filled, there is tendency towards clustering
- As a result, sequential search becomes slower
- Cannot increase data size
- Suitable for static data;Not suitable for real time data

Chaining (Open Hashing)

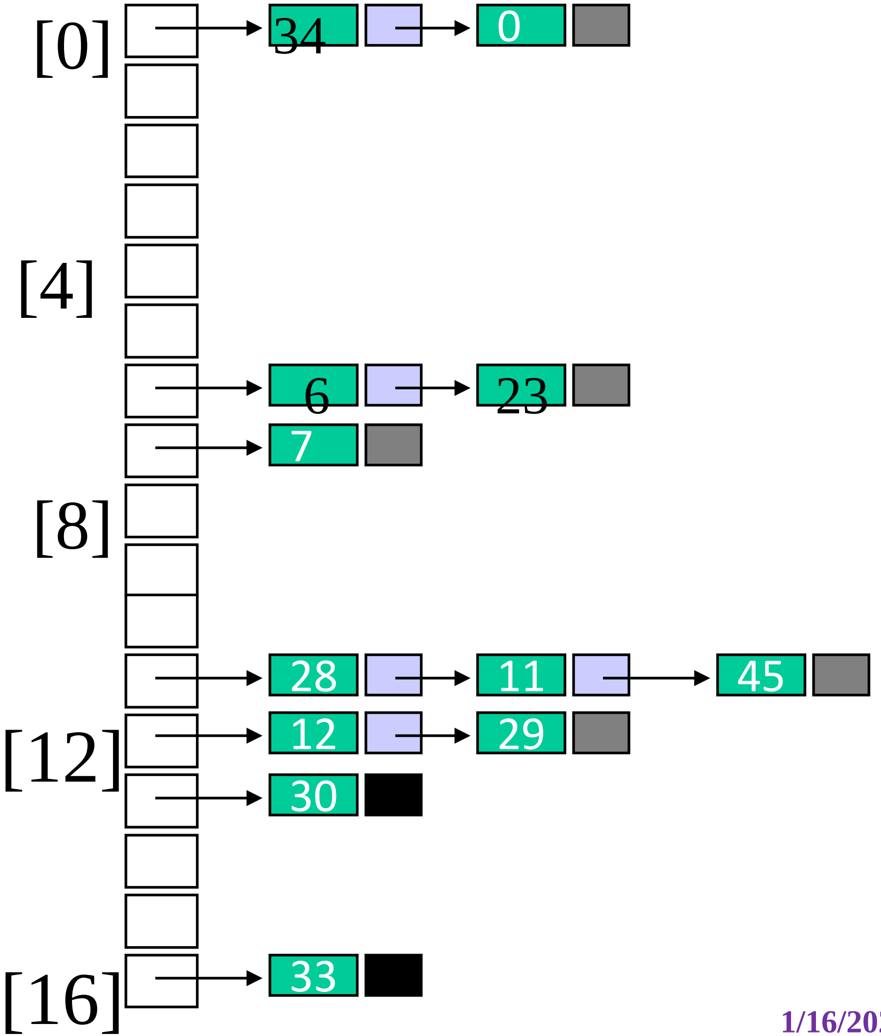
- The idea of **Chaining** is to combine the linked list and hash table to solve the collision problem
- Hash table stores pointers, each pointer pointing to a linked list
- A linked list contains all the keys that hash to a particular address

Example representation of Chaining



Example Chains

- keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45
- Bucket = key % 17



Advantages and Disadvantages

- Advantages
  - Overflow situation never arises
  - Easy insertion and deletion
  - Suitable for applications where number of key values varies drastically as it uses dynamic storage management policy
- Disadvantages
  - Maintenance of linked list
  - Extra storage space for link fields
  - If chain becomes long, then search time

  can become O(n) in worst case

  - Suitable for evenly distributed data

Analysis:
Worst case: O(n)
Best : O(1)
Average : **O (1+ α/2)**

Analysis of open and closed hashing

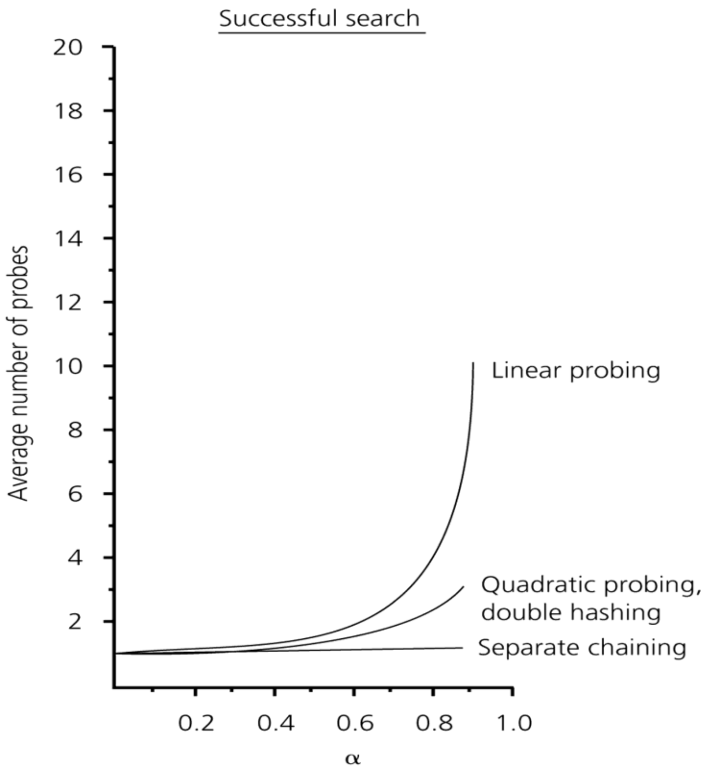
Open Hashing (Chaining)

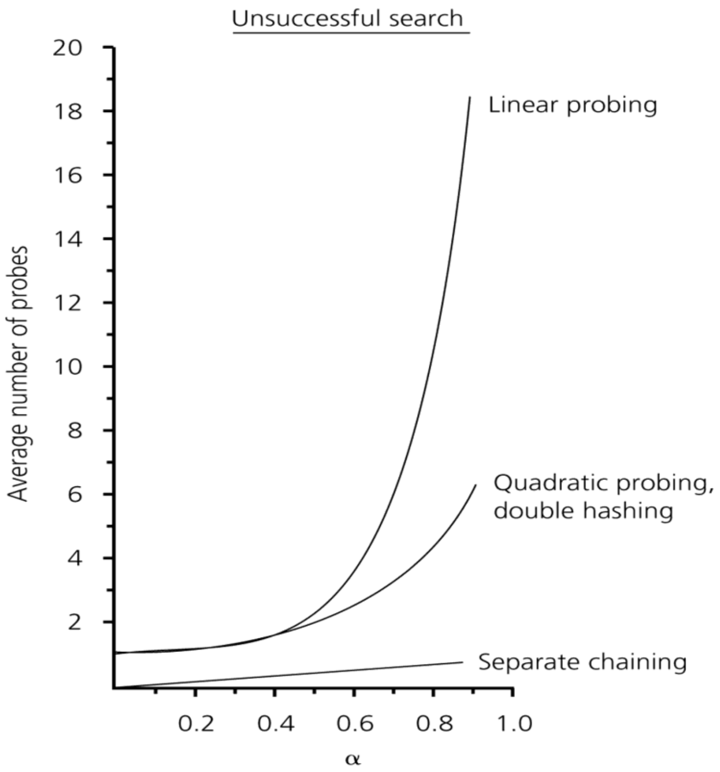
- Deletion is easy
- Overhead of extra space
- No. of keys can exceed the table size
- Performance is better for higher load factor
- Cache performance is poor

Closed Hashing (Open Addressing)

- Deletion is less easier
- No overhead of extra space
- No. of keys is limited to table size (requires rehashing)
- Performance degrades for higher loas factor
- Cache performance is poor

The relative efficiency of four collision-resolution methods





Linear Probing : Representative values for some setting

Load factor	Number of Probes	
α	Successful	Unsuccessful
.10	1.056	1.118
.20	1.125	1.281
.30	1.214	1.520
.40	1.333	1.889
.50	1.500	2.500
.60	1.750	3.625
.70	2.167	6.060
.80	3.000	13.000
.90	5.500	50.500
.95	10.500	200.500

Double Hashing : Representative values for some setting

Load factor	*Number of Probes*	
α	*Successful*	*Unsuccessful*
.10	1.050	1.005
.20	1.100	1.019
.30	1.150	1.041
.40	1.200	1.070
.50	1.250	1.107
.60	1.300	1.149
.70	1.350	1.197
.80	1.400	1.249
.90	1.450	1.307
.95	1.475	1.337

Rehashing

- If table gets too full , the running time for the operations will start taking too long and insertions may also fail
- Solved by building another table that is about twice as big and a new hash function (new table size should be a prime number)
- The original hash table is scanned & new hash value for each element is computed & inserted into new table
- This operation is called "**Rehashing**"
- Rehashing is a costly operation with O(n) time

When is rehashing done??

- **Load factor (α) = n / k**

  where

- *n* is the number of entries occupied in the hash table

- *k* is the number of location (table size)

- Rehashing is done when load factor reaches a particular threshold

Load Factor

- For Open addressing load factor should not be more than 0.5
- For chaining load factor should be close to 1

Hashing Applications

- **Compilers** use hash tables to implement the ***symbol table*** (a data structure to keep track of declared variables)
- **(Hashing application) Cryptography** : Apply hash function on message of a user to generate hash code. If the original message is manipulated then same hash code cannot be generated
- **Object representation :** the keys are the names of the members and methods of the object, and the values are pointers to the corresponding member or method
- **String pools/literal pools**

Hashing Applications

- **Gaming applications** – to store a position and possible moves from that position to avoid expensive re-computation
- **Online Spelling checker**

More efficient hashing techniques

- Cuckoo Hashing
- Hopscotch Hashing
- Extendible Hashing
- Perfect Hashing (Useful for static data like word dictionary)

[NOTE: This slide is optional. Not included in syllabus]