Hashing

Hash Table, Hashing Algorithm

Dr. Bibhudatta Sahoo
Communication & Computing Group
Department of CSE, NIT Rourkela
Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

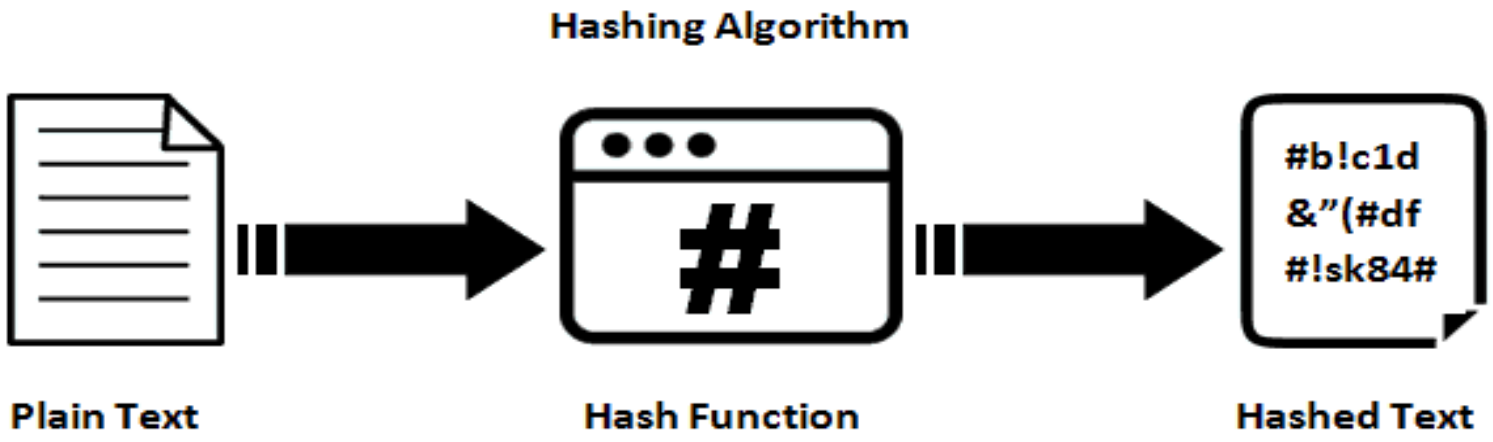
Linked list : COMPUTER VIRUS

102	START
105	AVAIL

ADDRESS	INFO	LINK
101	I	112
102	C	110
103	U	111
104	P	103
105		116
106	M	104
107	U	113
108	E	109
109	R	114
110	O	106
111	T	108
112	R	107
113	S	000
114		115
115	V	101
116		117
117		118
118		119
119		120
120		000

What is Hashing?

- Hashing is an algorithm that calculates a fixed-size bit string value from a file. A file basically contains blocks of data.
- Hashing transforms this data into a far shorter fixed-length value or key which represents the original string.
- The hash value can be considered the distilled summary of everything within that file.

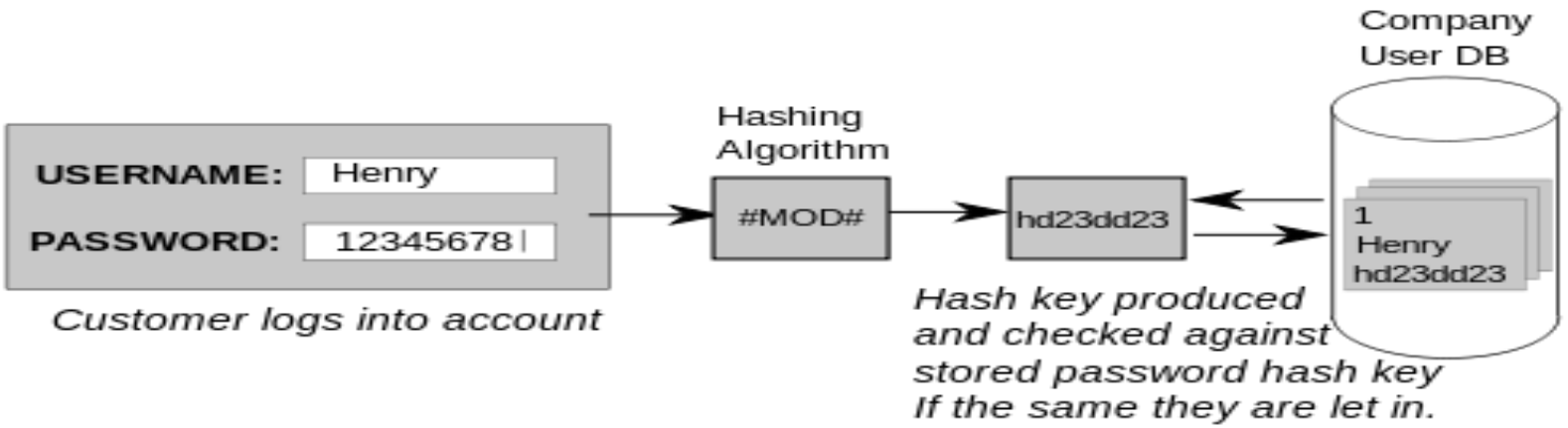


What is Hashing?

- A good hashing algorithm would exhibit a property called the avalanche effect, where the resulting hash output would change significantly or entirely even when a single bit or byte of data within a file is changed.
- A hash function that does not do this is considered to have poor randomization, which would be easy to break by hackers.

What is Hashing?

- A hash is usually a hexadecimal string of several characters.
- Hashing is also a unidirectional process so you can never work backwards to get back the original data.
- A good hash algorithm should be complex enough such that it does not produce the same hash value from two different inputs.
- If it does, this is known as a hash collision. A hash algorithm can only be considered good and acceptable if it can offer a very low chance of collision.

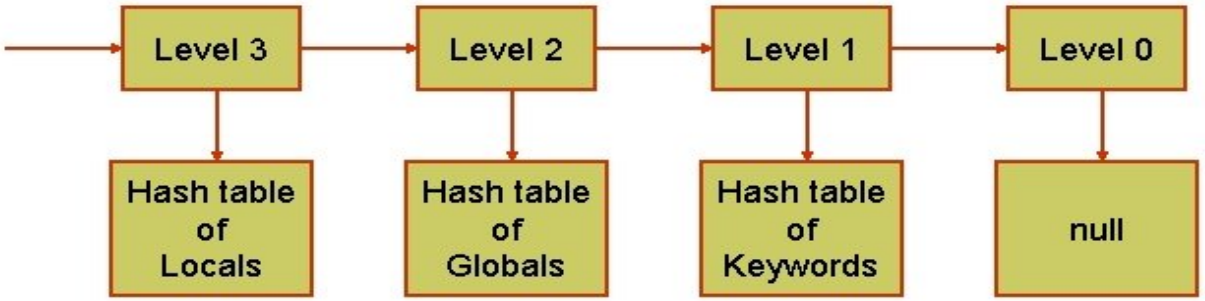


Symbol Table

- Definition

  A set of name-attribute pairs

- Operations
  - Determine if a particular name is in the table
  - Retrieve the attributes of the name
  - Modify the attributes of that name
  - Insert a new name and its attributes
  - Delete a name and its attributes



The ADT of Symbol Table

search, insertion, deletion

Structure **SymbolTable**(SymTab) is
objects: a set of name-attribute pairs, where the names are unique
functions:
for all *name* belongs to *Name*, *attr* belongs to *Attribute*, *symtab* belongs to *SymbolTable*, *max_size* belongs to integer

SymTab Create(max_size) ::= create the empty symbol table whose maximum capacity is *max_size*

Boolean IsIn(symtab, name) ::= if (name is in symtab) return TRUE
else return FALSE

Attribute Find(symtab, name) ::= if (name is in symtab) return the corresponding attribute else return null attribute

SymTab Insert(symtal, name, attr) ::= if (name is in symtab)
replace its existing attribute with attr
else insert the pair (name, attr) into symtab

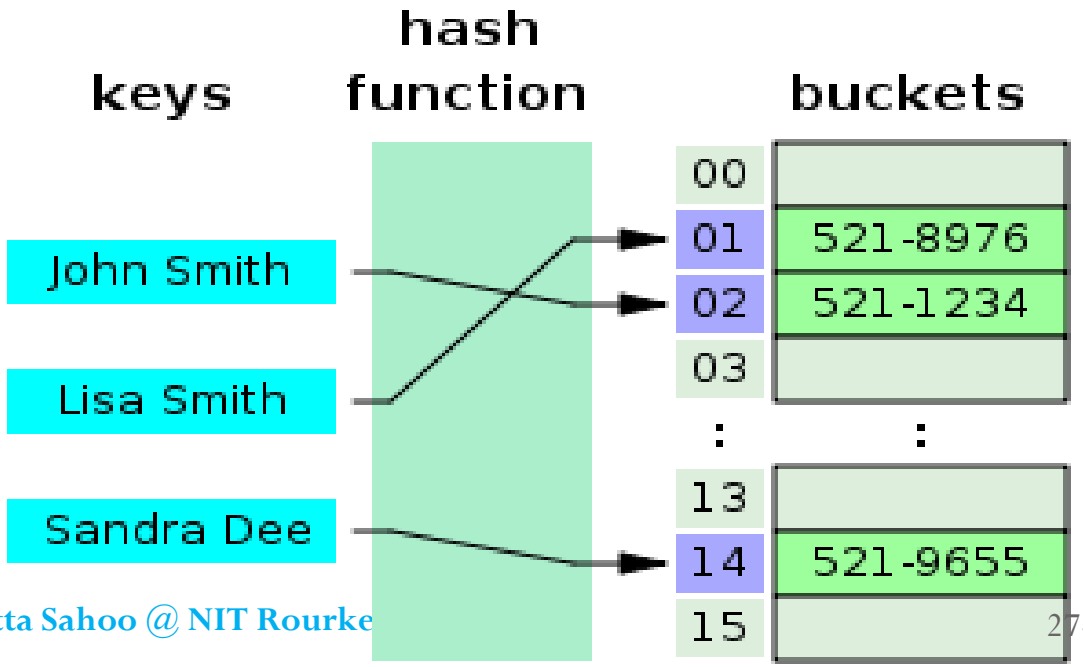
SymTab Delete(symtab, name) ::= if (name is not in symtab) return
else delete (name, attr) from symtab

Implementing a basic hash table (also known as a hash map) in Java

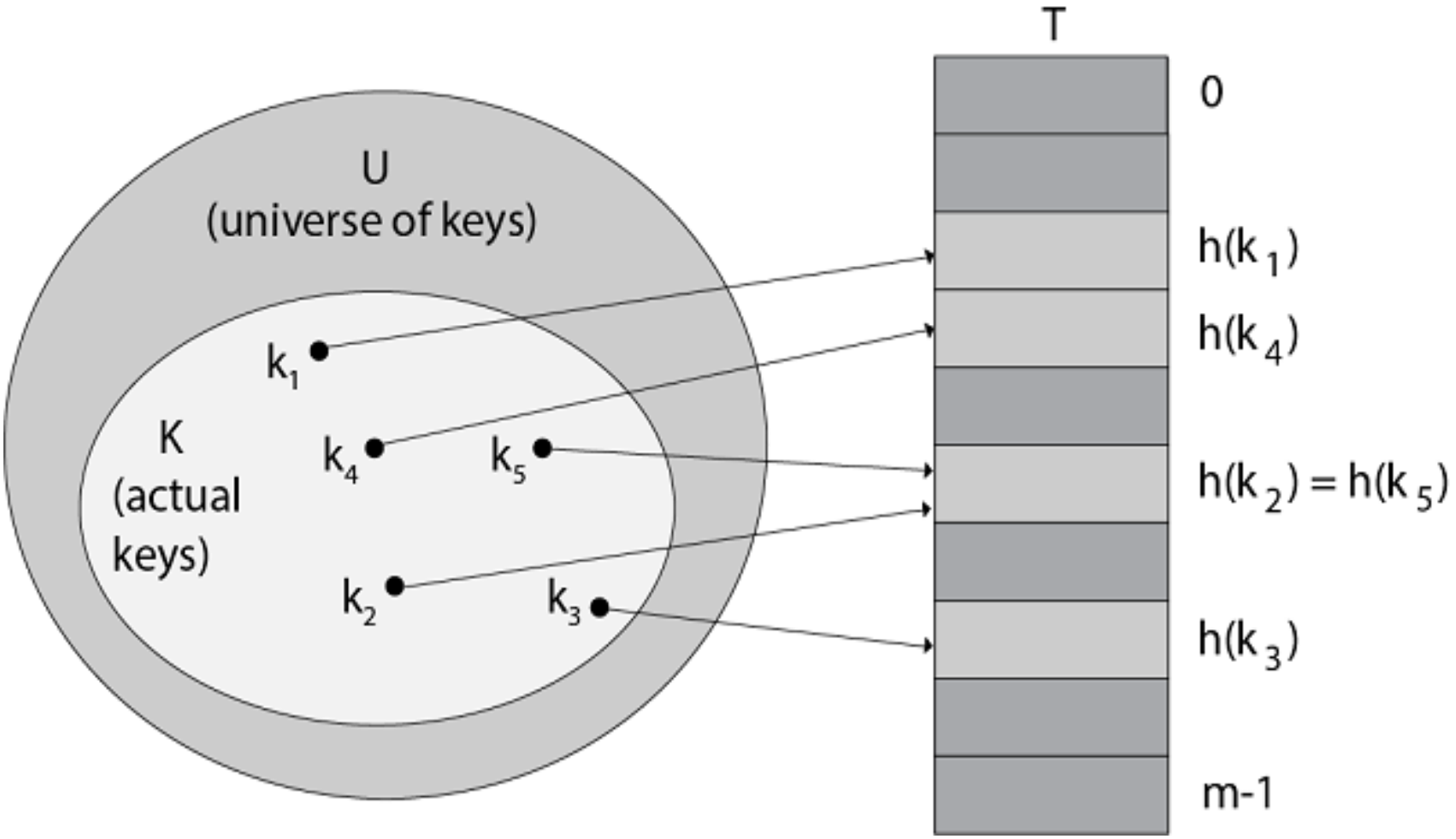
- Java's in-built **HashMap** class is a comprehensive implementation of a hash table
- **Key-Value Pair**: Represents an entry in the hash table.
- **Hash Function**: A function to calculate an index where an element will be stored.
- **Bucket Array**: An array where the data gets stored. If two keys get the same index (collision), then a linked list is used to store the new key-value pair. This is called chaining.
- **Basic Operations**: The primary operations include put, get, and remove.

Hashing

- Another important and widely useful technique for implementing dictionaries
- Constant time per operation (on the average)
- Worst case time proportional to the size of the set for each operation (just like array and chain implementation)

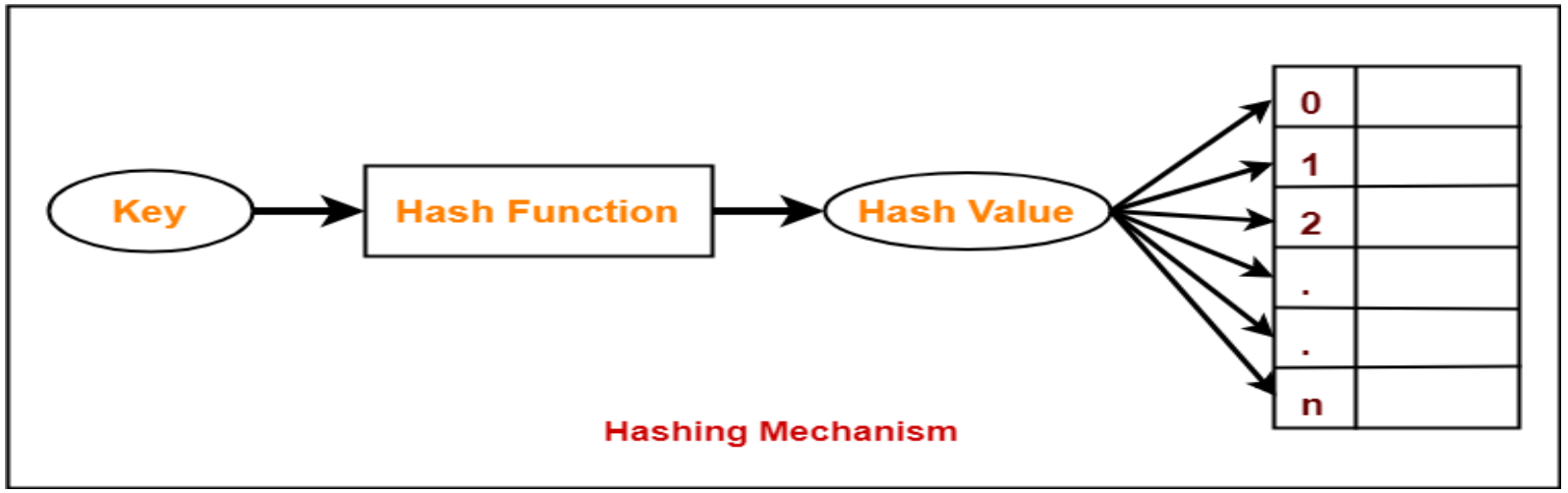


Hashing


T
0
U
(universe of keys)
h(k 1)
k1
h(k 4)
K
(actual
keys)
k4
k5
h(k 2) = h(k 5)
k2
k3
h(k 3)
m-1

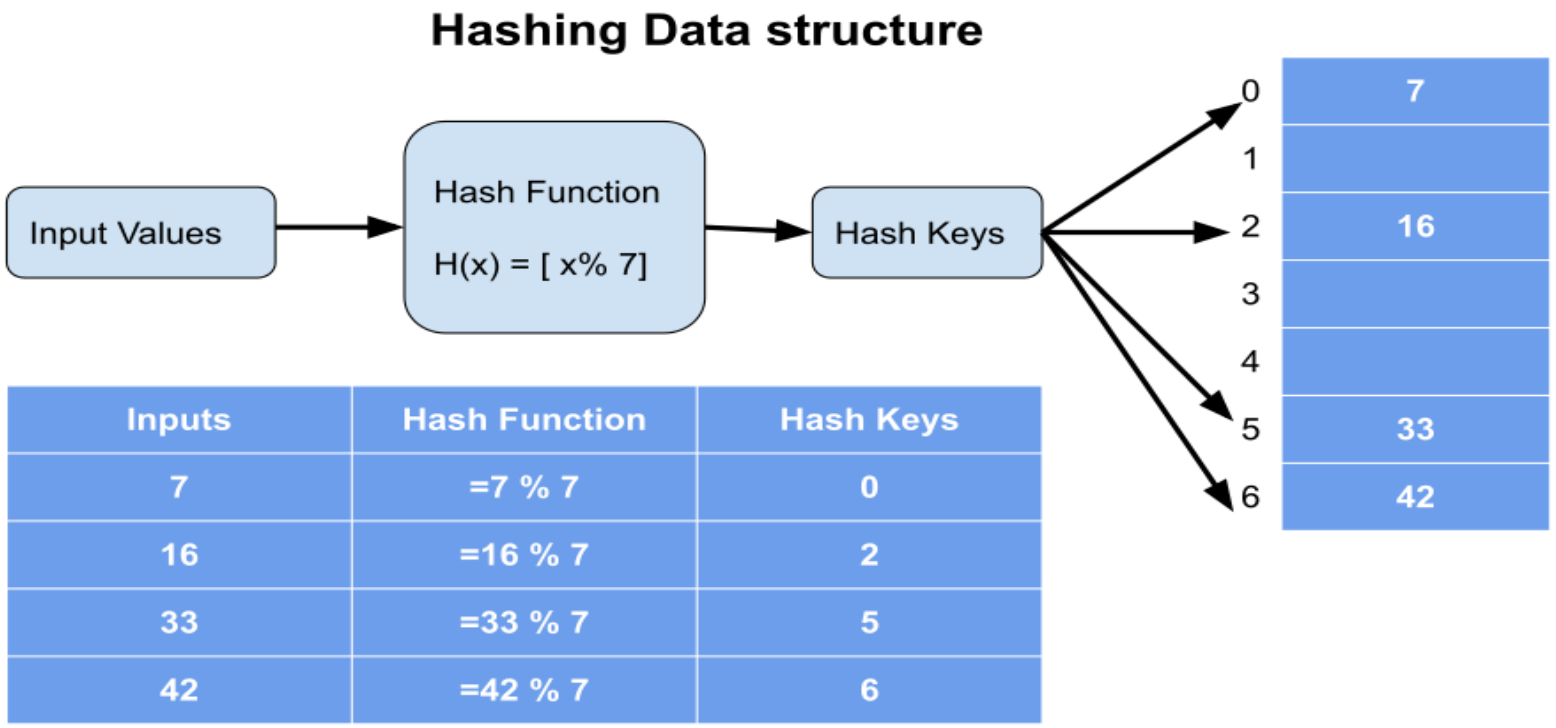
Hashing

- **Hashing in the data structure** is a technique of mapping a large chunk of data into small tables using a hashing function.
- It is also known as the message digest function.
- It is a technique that uniquely identifies a specific item from a collection of similar items.
- Hashing is designed to solve the problem of needing to efficiently find or store an item in a collection.



Hashing Mechanism

Hashing

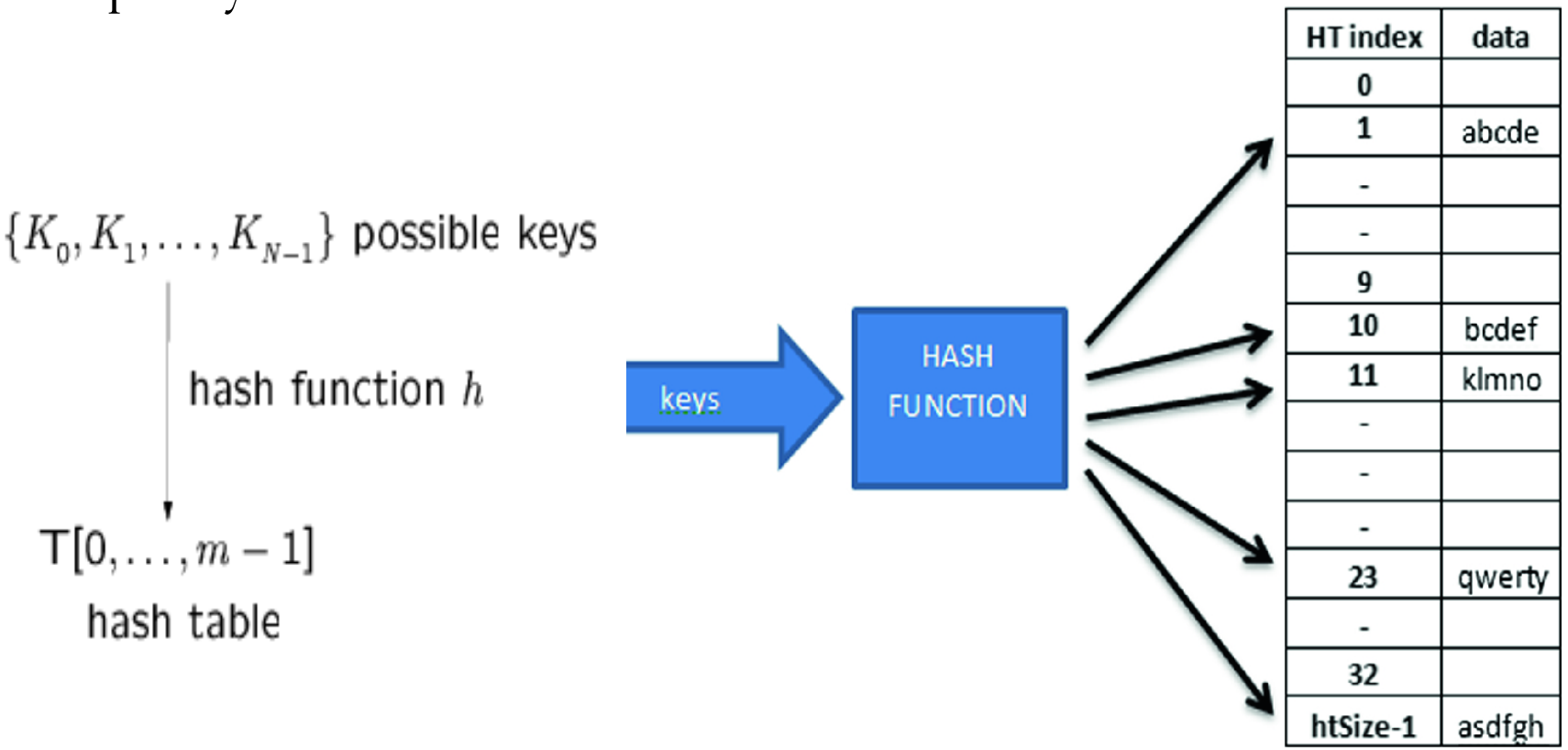


Inputs	Hash Function	Hash Keys
7	=7 % 7	0
16	=16 % 7	2
33	=33 % 7	5
42	=42 % 7	6

A hashing algorithm is a mathematical algorithm that converts an input data array of a certain type and arbitrary length to an output bit string of a fixed length.

Hashing in a data structure is a two-step process.

- The hash function converts the item into a small integer or hash value. This integer is used as an index to store the original data.
- It stores the data in a hash table. You can use a hash key to locate data quickly.



Hashing

- Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
- Hashing is also known as **Hashing Algorithm** or **Message Digest Function**.
- It is a technique to convert a range of key values into a range of indexes of an array.
- It is used to facilitate the next level searching method when compared with the linear or binary search.
- Hashing allows to update and retrieve any data entry in a constant time O(1).
- Constant time O(1) means the operation does not depend on the size of the data.
- Hashing is used with a database to enable items to be retrieved more quickly.
- It is used in the encryption and decryption of digital signatures.

What are the benefits of Hashing?

- One main use of hashing is to compare two files for equality. Without opening two document files to compare them word-for-word, the calculated hash values of these files will allow the owner to know immediately if they are different.
- Hashing is also used to verify the integrity of a file after it has been transferred from one place to another, typically in a **file backup** program like **SyncBack**. To ensure the transferred file is not corrupted, a user can compare the hash value of both files. If they are the same, then the transferred file is an identical copy.
- In some situations, an encrypted file may be designed to never change the file size nor the last modification date and time (for example, virtual drive container files). In such cases, it would be impossible to tell at a glance if two similar files are different or not, but the hash values would easily tell these files apart if they are different.

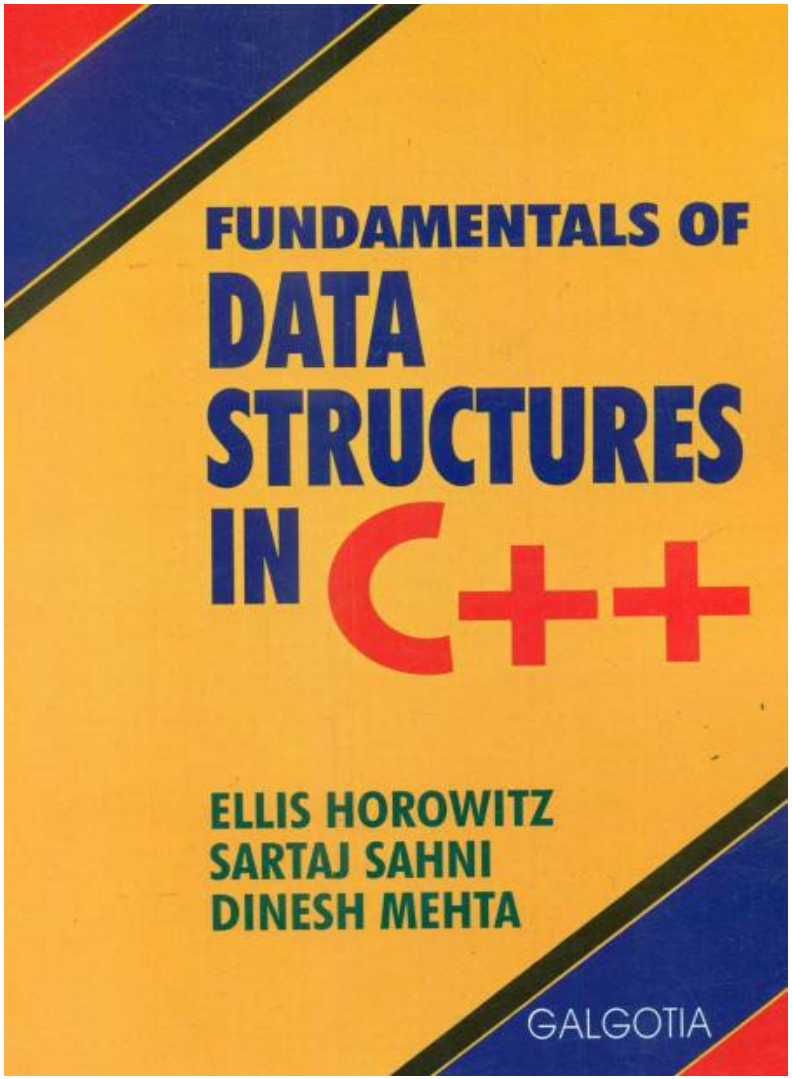
Types of Hashing [file integrity checks]

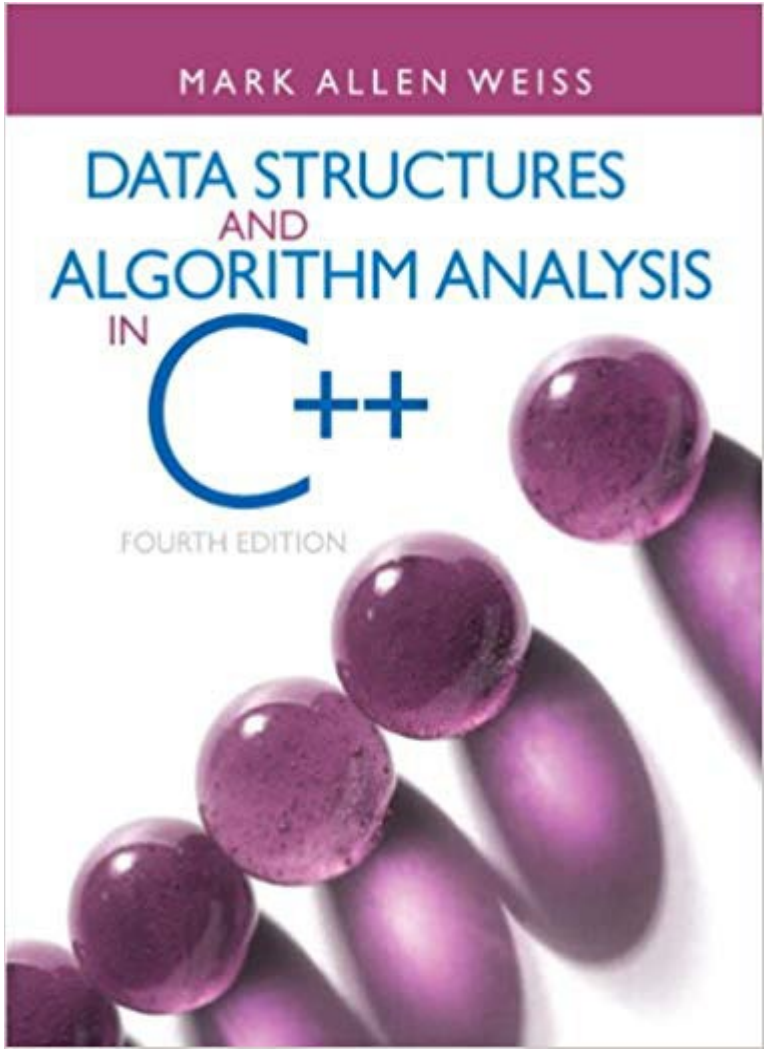
- **MD5** - An MD5 hash function encodes a string of information and encodes it into a 128-bit fingerprint. MD5 is often used as a checksum to verify data integrity. However, due to its age, MD5 is also known to suffer from extensive hash collision vulnerabilities, but it's still one of the most widely used algorithms in the world.
- **SHA-2** – SHA-2, developed by the National Security Agency (NSA), is a cryptographic hash function. SHA-2 includes significant changes from its predecessor, SHA-1.
- The SHA-2 family consists of six hash functions with digests (hash values) that are 224, 256, 384 or 512 bits: SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256.

Types of Hashing [file integrity checks]

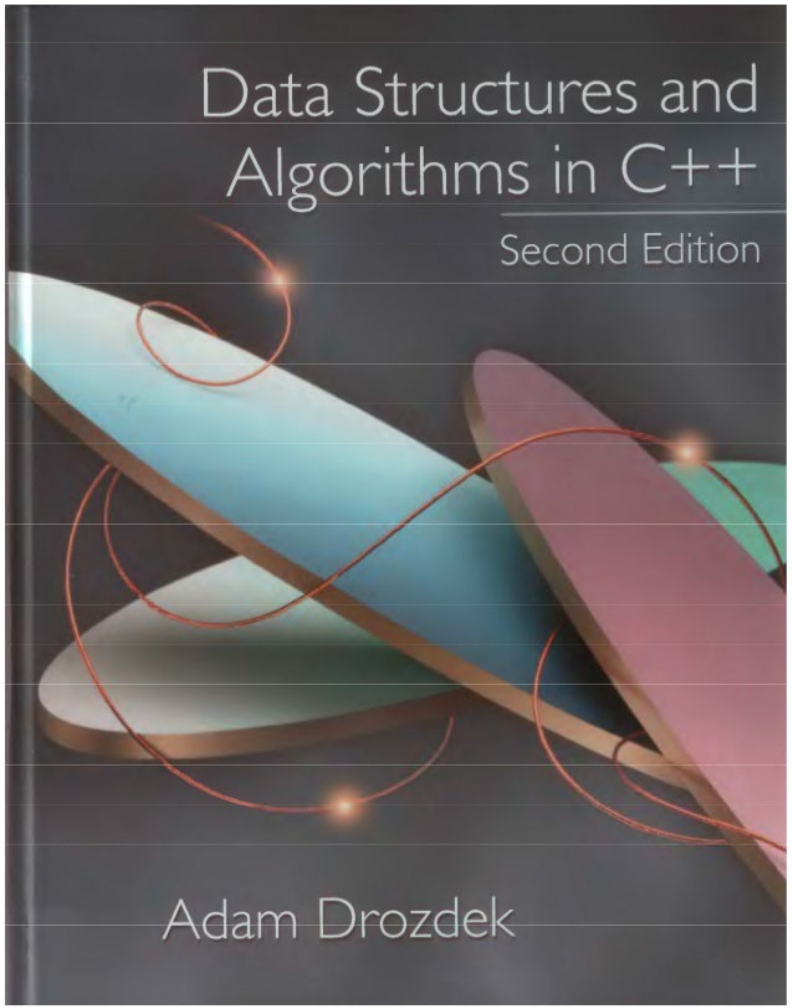
- **CRC32** – A cyclic redundancy check (CRC) is an error-detecting code often used for detection of accidental changes to data.
- Encoding the same data string using CRC32 will always result in the same hash output, thus CRC32 is sometimes used as a hash algorithm for file integrity checks. These days, CRC32 is rarely used outside of **Zip** files and **FTP** servers.

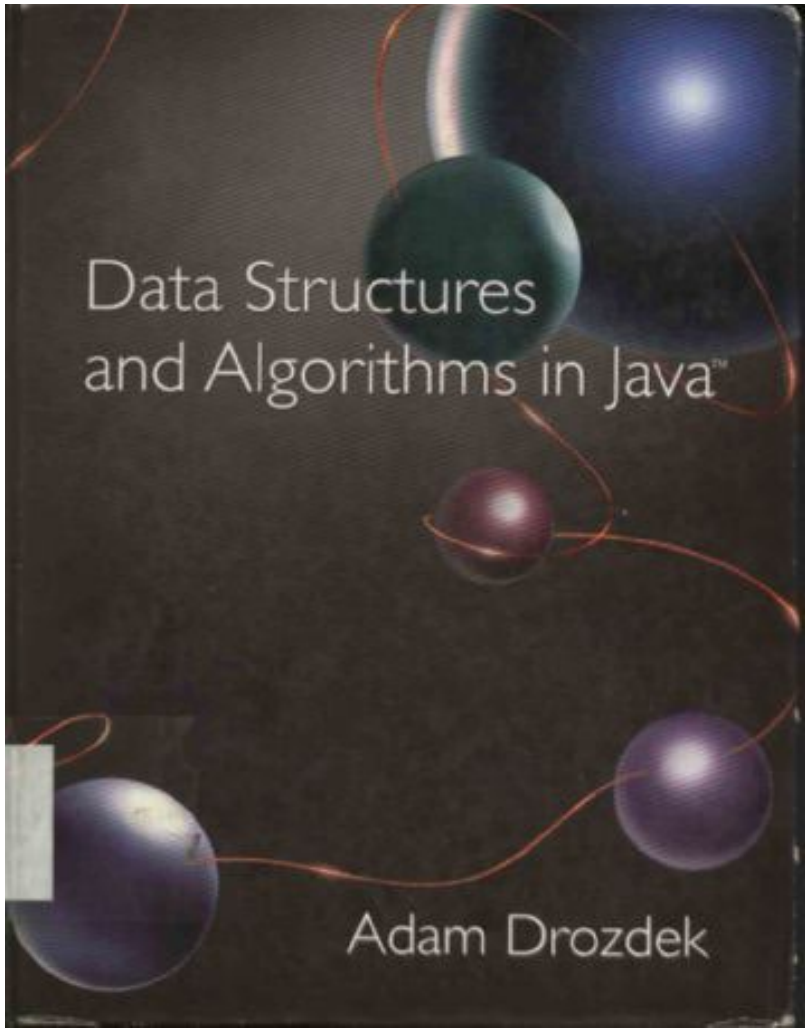
Suggested Reading





Suggested Reading





Why do we need hashing

- Many applications deal with lots of data

  Search engines and web pages
- There are innumerable look ups.
- The look ups are time critical.
- Typical data structures like arrays and lists, may not be sufficient to handle efficient lookups
- In general: When look-ups need to occur in near constant time O(1)

What is Hashing???

1. The search concept either it is **linear or binary search**, it always depends upon the number of elements in the list.
2. As linear search complexity **O(n)=n** and binary search having time complexity **O(n)=log n**, which always depends on the number of elements ‘n’.
3. ***Hashing*** is a searching technique which is **independent** of the number of elements(say ‘n’).

Hence, using hashing searching will be finished within **constant** period of time.

Hash table

- A hash table is a fundamental data structure that allows for the efficient storage, retrieval, and deletion of data by using a mechanism called hashing.
- The primary advantage of hash tables is their ability to access stored data in constant time, O(1), under ideal conditions.

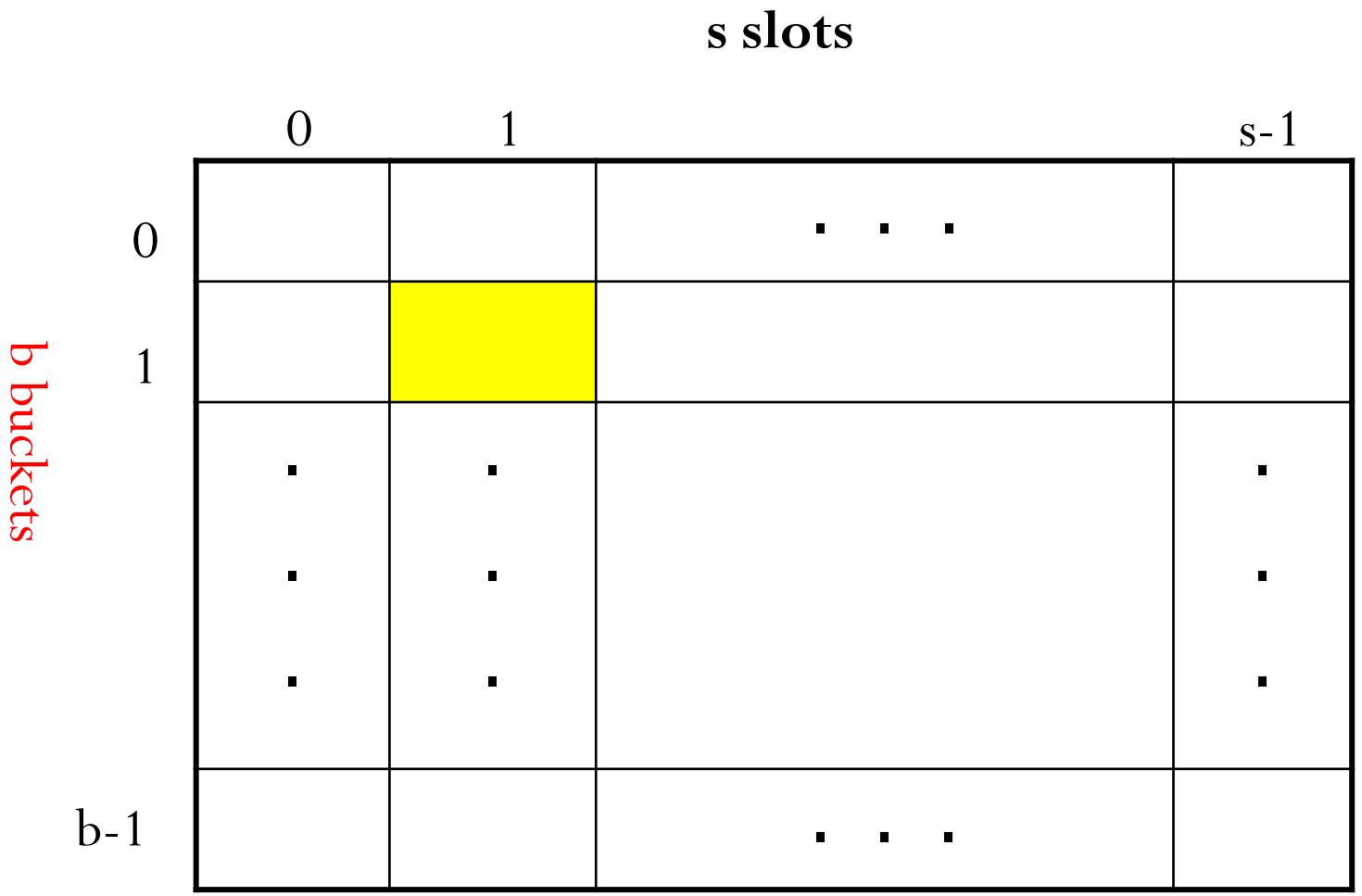
Basics of Hash Tables:

- **Key-Value Pairs**: Hash tables store data as key-value pairs. Given a key, you can quickly retrieve its associated value.
- **Hash Function**: To determine where to store a key-value pair, the key is passed through a hash function. This function converts the key into an index in the array where the value will be stored. The goal is to distribute keys uniformly across the array.
- **Buckets**: Sometimes, two keys might hash to the same index. This is called a collision. One way to handle collisions is to store a list (or another kind of collection) at each index. Each list is called a bucket.

Hash table

- The ideal hash table structure is merely an array of some fixed size, containing the items.
- A stored item needs to have a data member, called ***key***, that will be used in computing the index value for the item.
  - Key could be an *integer*, a *string*, etc.
  - e.g. a name or Id that is a part of a large employee structure
- The size of the array is *TableSize*.
- The items that are stored in the hash table are indexed by values from *0* to *TableSize – 1*.
- Each key is mapped into some number in the range 0 to *TableSize – 1*.
- The mapping is called a *hash function*.

Hash table



Hash table

- Hash table or hash map is a data structure used to store key-value pairs.
- It is a collection of items stored to make it easy to find them later.
- It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.
- A hash table is a collection of items which are stored in such a way as to make it easy to find them later.
- Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0.
  - A hash table is partitioned into many *buckets*.
  - Each bucket has many *slots*.
  - Each slot holds one record.
  - A hash function h(x) transforms the identifier (key) into an address in the hash table

Hash table

- **The implementation of hash tables is called hashing.**
- Hashing is a technique used for performing insertions, deletions and finds in constant average time (i.e. O(1))
- This data structure, however, is not efficient in operations that require any ordering information among the elements, such as **findMin**, **findMax** and printing the entire table in sorted order.
- The mapping between an item and the slot where that item belongs in the hash table is called the **hash function**.
- The hash function will take any item in the collection and return an integer in the range of slot names, between **0** and ***b*-1**.

Hashtable

- The **Hashtable** class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value.
- To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.
- The java.util.Hashtable class is a class in Java that provides a key-value data structure, similar to the Map interface.
- It was part of the original Java Collections framework and was introduced in Java 1.0.

Hash table :Operations

- **Insertion –** When a new record is inserted into the table, The hash function h generate a bucket address for the new record based on its hash key K. Bucket address = h(K)
- **Searching –** When a record needs to be searched, The same hash function is used to retrieve the bucket address for the record.
- For Example, if we want to retrieve whole record for ID 76, and if the hash function is **mod (5) on that ID**, the bucket address generated would be 4. Then we will directly got to address 4 and retrieve the whole record for ID 104. Here ID acts as a hash key.
- **Deletion –** If we want to delete a record, Using the hash function we will first fetch the record which is supposed to be deleted. Then we will remove the records for that address in memory.
- **Updation –** The data record that needs to be updated is first searched using hash function, and then the data record is updated.

Adding element to Hash Table

```
// Java program to demonstrate
// adding elements to Hashtable

import java.io.*;
import java.util.*;

class AddElementsToHashtable {
    public static void main(String args[])
    {
        // No need to mention the
        // Generic type twice
        Hashtable<Integer, String> ht1 = new Hashtable<>();

        // Initialization of a Hashtable
        // using Generics
        Hashtable<Integer, String> ht2
            = new Hashtable<Integer, String>();

        // Inserting the Elements
        // using put() method
        ht1.put(1, "one");
        ht1.put(2, "two");
        ht1.put(3, "three");

        ht2.put(4, "four");
        ht2.put(5, "five");
        ht2.put(6, "six");

        // Print mappings to the console
        System.out.println("Mappings of ht1 : " + ht1);
        System.out.println("Mappings of ht2 : " + ht2);
    }
}
```

Hashing

Static hashing

- **perfect hashing without collision**
- **hashing with collision**
  - **separate chaining**
  - **open addressing**
    - linear probing
    - quadratic probing
    - double hashing
  - **Rehashing**

Dynamic hashing

- extensible hashing
- linear hashing

Static hashing

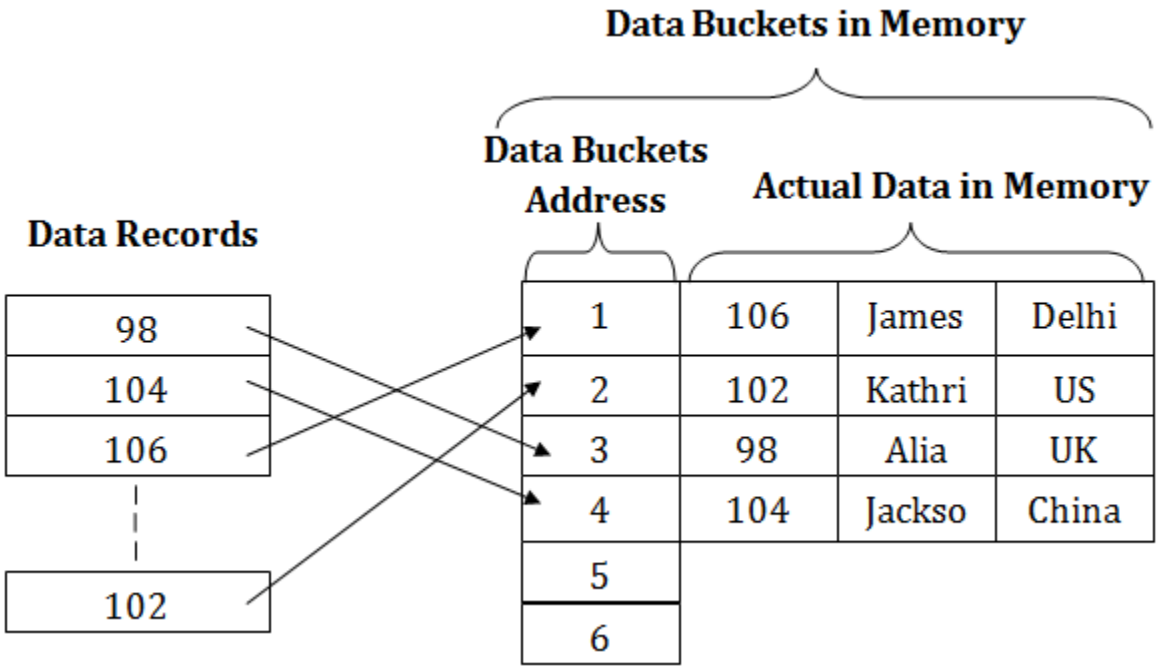
- Identifiers are stored in a fixed-size table called hash table.
- The address of location of an identifier, **x**, is obtained by computing some arithmetic function **h(x)**.
- The memory available to maintain the symbol table (hash table) is assumed to be sequential.
- The hash table consists of **b** buckets and each bucket contains **s** records.
- **h(x)** maps the set of possible identifiers onto the integers 0 through **b-1**.
- The **identifier density** of a hash table is the ratio ***n/T***, where **n** is the number of identifiers in the table and ***T*** is the total number of possible identifiers.
- The **loading density** or loading factor of a hash table is **$\alpha = n/(sb)$**.

Static hashing

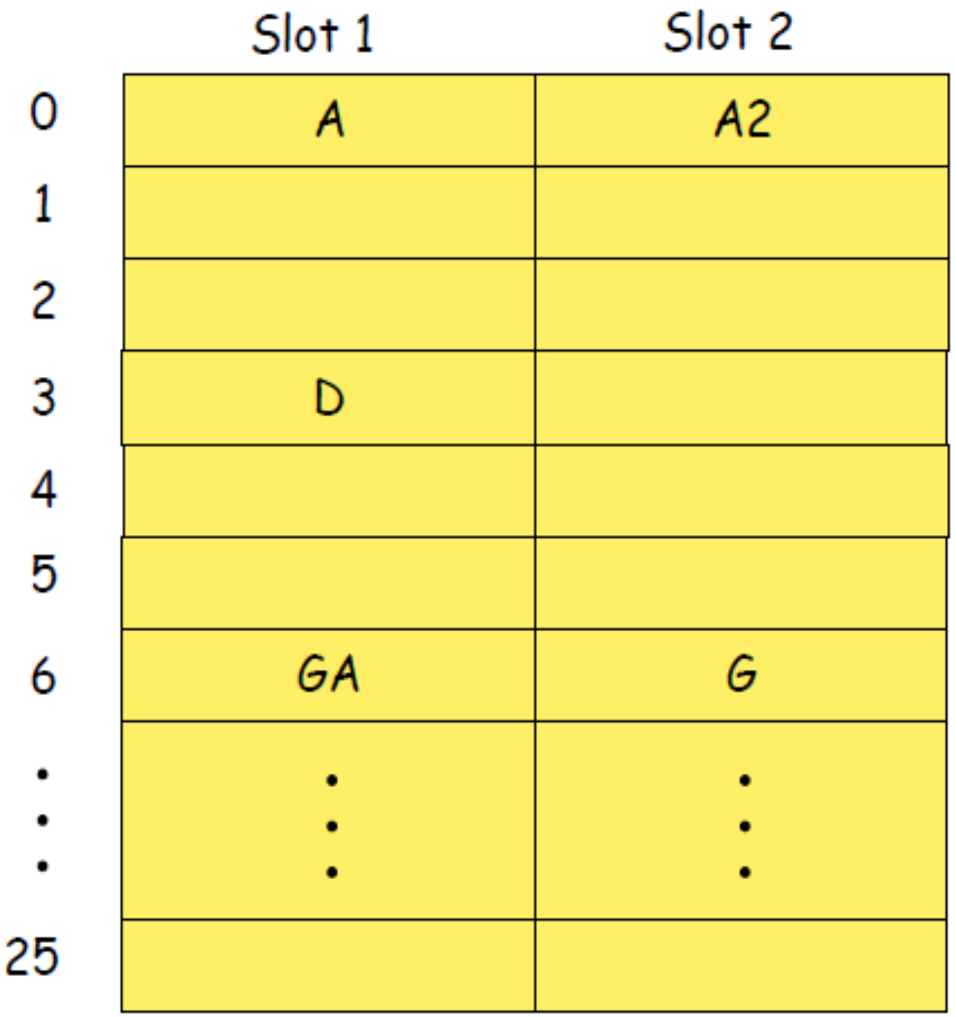
- Two identifiers, I_1, and I_2, are said to be synonyms with respect to HASH FUNCTION *h* if $\boldsymbol{h(I_1) = h(I_2)}$.
- An **overflow** occurs when a new identifier *i* is mapped or hashed by *h* into a full bucket .

Static hashing

- A **collision** occurs when two non-identical identifiers are hashed into the same bucket.
- If the bucket size is 1, **collisions** and **overflows** occur at the same time.



Static hashing



Assume there are 10 distinct identifiers

GA, D,A,G,L,A2,A1,A3,A4 and E

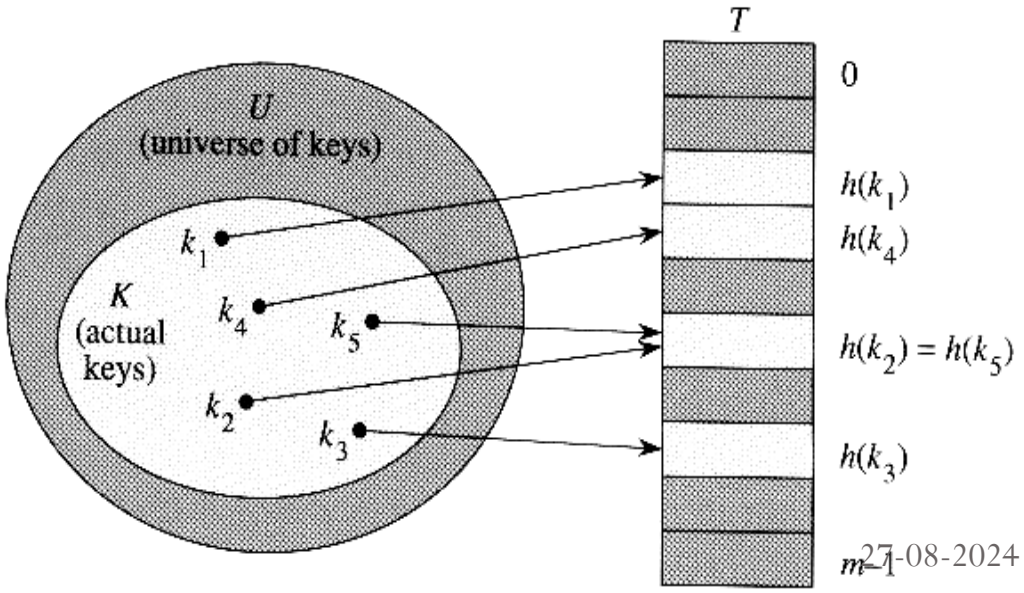
Large number of collisions and overflows!

If no overflow occur, the time required for hashing depends only on the time required to compute the hash function h.

Hash Function

- Simple to compute
- Minimize the number of collisions
- Dependent upon all the characters in the identifiers

  ⇒ **uniform hash function**
- If there are **b** buckets, we hope to have **h(x)=i** with the probability being **(1/b)**



Different type of uniform Hash Function/methods

1. Direct Method
2. Subtraction method
3. Modulo division
4. Digit Extraction
5. Mid square
6. Folding
7. Rotation
8. Pseudorandom generation

#1;Hashing methods: Direct Method

- Key is the address without algorithmic manipulation
- Data structure contain element for possible key

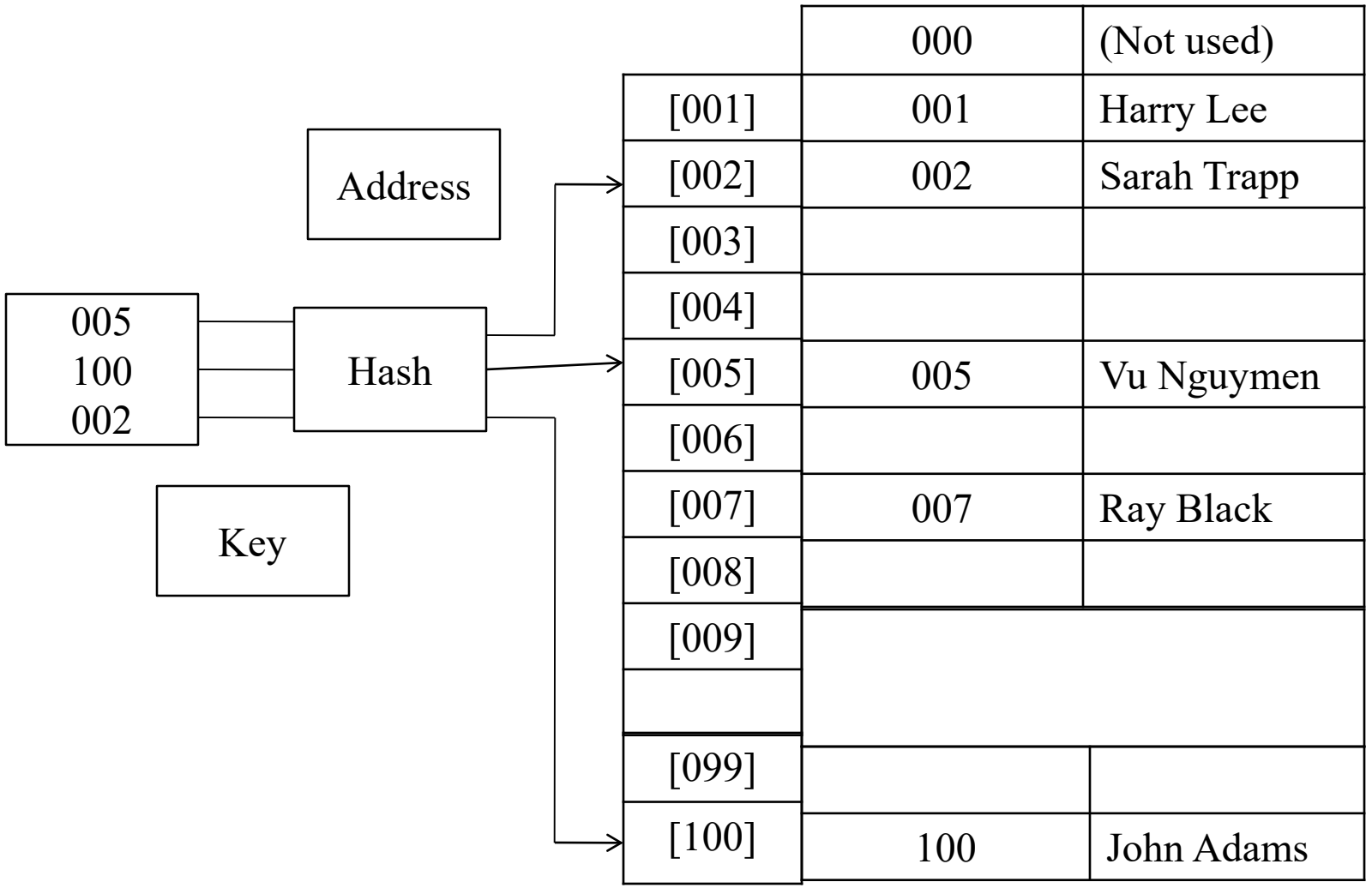
Example problem

- To analyze total monthly sales by days of month
- For each sale we need date and amount of sale
- To calculate sales record for month, we need day of month

as key for array and add sales amount to accumulator

```
daily Sales[ sale. day ] = daily Sales[ sale . day ] + sale . amount;
```

Direct Hashing of Employee numbers



	000	(Not used)
[001]	001	Harry Lee
[002]	002	Sarah Trapp
[003]		
[004]		
[005]	005	Vu Nguymen
[006]		
[007]	007	Ray Black
[008]		
[009]		
[099]		
[100]	100	John Adams

#2; Hashing Methods: Subtraction method

- Direct and subtraction hash functions guarantee search effort of one with no collisions
- In one to one hashing method only one key hashes to each address

Example

- Company have 100 employees, employee number starts from 1001 to 1100
- Hashing function subtracts 1000 from key to determine address

#3; Hashing Methods: Modulo division

- Divides the key by array size and use remainder for address
- Algorithm works with any list size, but list size is a prime number produces fewer collisions

address = key MODULO list Size

$$h(x) = x \bmod M$$

#4; Hashing Methods: Digit Extraction

- Digits are extracted from key and used as address

Example

- Six digit employee number is used to hash three digit address (000-999)
- Select first, third and fourth digits use them as address

  379452-**394**

  121267-**112**

  378845-**388**

  160252-**102**

  045128-**051**

#5; Hashing Methods: Mid square

- Key is squared and the address is selected from the middle of the squared number

Example

- Given a key of 9452, midsquare address calculation is shown using four digit address (0000-9999)

  9452^2=89340304: address is 3403

- Select first three digits and then use midsquare method

  379452 : 379^2 = 143641 - 364

  121267 : 121^2 = 014641 - 464

  378845 : 378^2 = 142884 - 288

  160252 : 160^2 = 025600 - 560

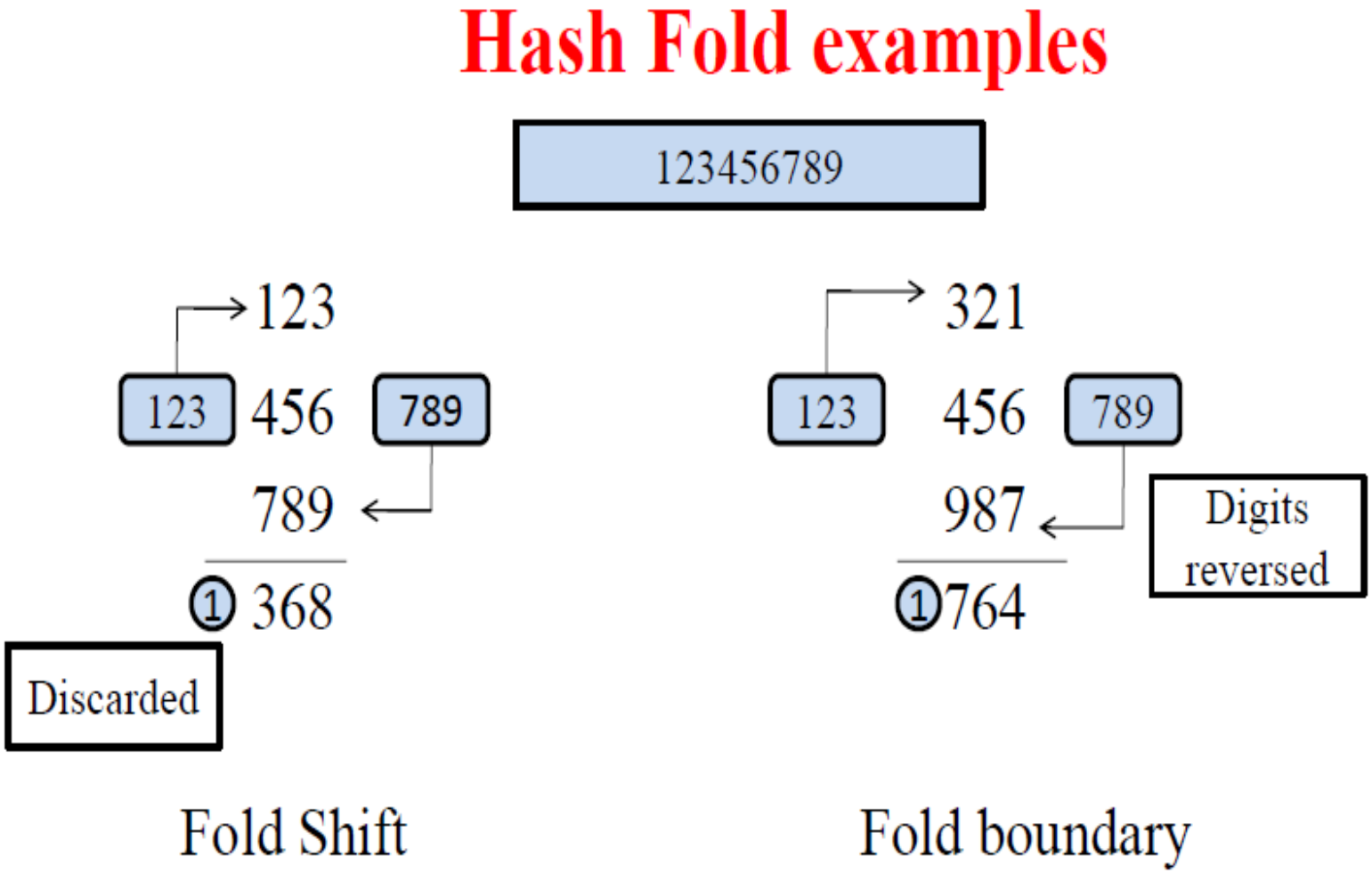
  045128 : 045^2 = 002025 - 202

#6; Hashing Methods: Folding

Two folding methods

- Fold shift
- Fold boundary
- Fold shift key value is divided into parts whose size matches size of required address
- Left and right parts are shifted and added with middle part
- In fold boundary, left and right numbers are folded on a fixed boundary between them and center number

Hash Folding



#6; Hashing Methods: Folding

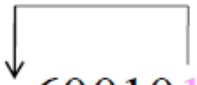
- Partition the identifier x into several parts, and add the parts together to obtain the hash address
- e.g. x=12320324111220; partition x into 123,203,241,112,20; then return the address 123+203+241+112+20=699
- Shift folding vs. folding at the boundaries

#7; Hashing Methods: Rotation

- Rotation is used in combination with folding and pseudorandom hashing
- Hashing keys are identical except for last character
- Rotating last character to front of key and minimize the effect

Example

- Consider case of six digit employee number that is used in large company



Original Key	Rotation	Rotated Key
600101	600101	160010
600102	600102	260010
600103	600103	360010
600104	600104	460010
600105	600105	560010

#8; Hashing Methods: Pseudorandom generation

- Key is used as seed in pseudorandom number generator and the random number is scaled into possible address range using modulo-division
- Pseudorandom number generator generate same number of series
- A common random number generator is y = ax + c

Example

Assume a=17, c=7, x=121267,Prime number=307

y= ((17*121267) + 7) modulo 307

y= (2061539 + 7) modulo 307

y= 2061546 modulo 307

y= 41

Hash Functions

- Theoretically, the performance of a hash table depends only on the method used to handle overflows and is independent of the hash function as long as an uniform hash function is used.
- In reality, there is a tendency to make a biased use of identifiers.
- Many identifiers in use have a common suffix or prefix or are simple permutations of other identifiers.
  - Therefore, different hash functions would give different performance.

Average Number of Bucket Accesses Per Identifier Retrieved

a = n/b	0.50		0.75		0.90		0.95	
Hash Function	Chain	Open	Chain	Open	Chain	Open	Chain	Open
mid square	1.26	1.73	1.40	9.75	1.45	37.14	1.47	37.53
division	1.19	4.52	1.31	7.20	1.38	22.42	1.41	25.79
shift fold	1.33	21.75	1.48	65.10	1.40	77.01	1.51	118.57
bound fold	1.39	22.97	1.57	48.70	1.55	69.63	1.51	97.56
digit analysis	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
theoretical	1.25	1.50	1.37	2.50	1.45	5.50	1.48	10.50

Hashing Methods: Summary

- In hashed search, key through algorithmic function determines location of data
- Hashing functions including direct, subtraction, modulo division, digit extraction, midsquare, folding, rotation, pseudorandom generation
- In direct hashing, key is address without algorithmic manipulation
- In subtraction hashing key is transformed to address by subtracting a fixed number from it
- In modulo division hashing key is divided by list size and remainder plus 1 is used as address
- In digit extraction hashing select digits are extracted from key and used as address

Hashing Methods: Summary

- In **midsquare hashing** key is squared and address is selected from middle of result
- In **fold shift hashing** key is divided into parts whose size match size of required address. Then parts are added to obtain address
- In **fold boundary hashing**, key is divided into parts whose size match size of required address. Then left and right parts are reserved and added to middle part to obtain address
- In **rotation hashing** far right digit of key is rotated to left to determine address
- In **pseudorandom generation hashing**, key is used as seed to generate pseudorandom number. Result is scaled to obtain address

Collision Resolution

Collision Resolution Strategies

- If two keys map on the same hash table index then we have a collision.
- As the number of elements in the table increases, the likelihood of a *collision* increases - so make the table **as large as practical**
- Collisions may still happen, so we need a *collision resolution strategy*

- Two approaches are used to resolve collisions.

- **Open addressing**: store the key/entry in a different position

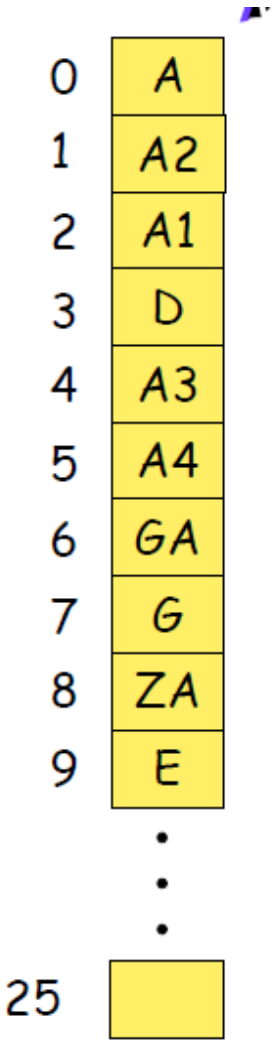
- **Separate chaining**: chain together several keys/entries in each position.

Open Addressing

- Assumes the hash table is an array
- The hash table is initialized so that each slot contains the null identifier.
- When a new identifier is hashed into a full bucket, find the closest unfilled bucket. This is called **linear probing** or **linear open addressing**
- **Open addressing**: Search the hash table in some systematic fashion for a bucket that is not full.
  - Linear probing (linear open addressing)
  - Quadratic probing
  - Rehashing

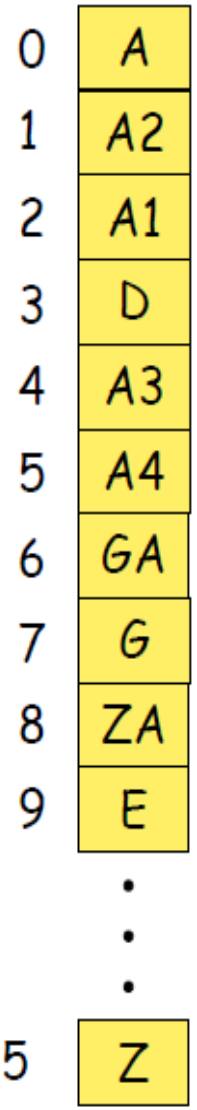
Example : 1

- Assume 26-bucket table with one slot per bucket and the following identifiers: **GA**, D, A, **G**, L, A2, A1, A3, A4, Z, ZA, E. Let the hash function **h(x) = first character of x.**
- When entering G, G collides with GA and is entered at ht[7] instead of ht[6].
- When linear open address is used to handle overflows, a hash table search for identifier x proceeds as follows:
  - compute h(x)
  - examine identifiers at positions ht[h(x)], ht[h(x)+1], …, ht[h(x)+j], in this order until one of the following condition happens:
    - ht[h(x)+j]=x; in this case x is found
    - ht[h(x)+j] is null; x is not in the table
    - We return to the starting position h(x); the table is full and x is not in the table



#1:Linear Probing

- In **Example-1**, we found that when linear probing is used to resolve overflows, identifiers tend to cluster together. Also adjacent clusters tend to coalesce. This increases search time.
  - e.g., to find ZA, you need to examine ht[25], ht[0], …, ht[8] (total of 10 comparisons).
  - To retrieve each identifier once, a total of 39 buckets are examined **(average 3.25 bucket** per identifier).
- The expected average number of identifier **comparisons**, P, to look up an identifier is approximately $(2 - \alpha)/(2-2\alpha)$, where α is the loading density.
- **Example-1** $\alpha = 12/26 = .47$ and $P = 1.5$.
- Even though the average number of probes is small, the worst cas can be quite large.



1:Linear Probing

- hi(x) = {**Hash**(x) + f(i) }mod **Hsize**, with f(0)=0, Let **f(i) = i**
- Inserting the keys{ 89, 18,49,58,69} in order

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

1:Linear Probing :2/2

- Linear probing is easy to implement, but it suffers from a problem known as ***primary clustering***.
- Long runs of occupied slots build up, increasing the average search time. For example, if we have $n = m/2$ keys in the table, where every even-indexed slot is occupied and every odd-indexed slot is empty, then the average unsuccessful search takes 1.5 probes.
- If the first $n = m/2$ locations are the ones occupied, however, the average number of probes increases to about $n/4 = m/8$.
- Clusters are likely to arise, since if an empty slot is preceded by i full slots, then the probability that the empty slot is the next one filled is $(i + 1)/m$, compared with a probability of $1/m$ if the preceding slot was empty. Thus, runs of occupied slots tend to get longer, and linear probing is **not a very good approximation to uniform hashing**.

Theoretical Evaluation of Overflow Techniques (Cont.)

Theorem 8.1: Let a=n/b be the loading density of a hash table using a uniform hashing function h. Then

(1) for linear open addressing

$$U_n \approx \frac{1}{2}[1+\frac{1}{(1-\alpha)^2}] \qquad S_n \approx \frac{1}{2}[1+\frac{1}{1-\alpha}]$$

(2) for rehashing, random probing, and quadratic probing

$$U_n \approx 1/(1-\alpha) \qquad S_n \approx -[\frac{1}{\alpha}]\log_e(1-\alpha)$$

(3) for chaining

$$U_n \approx \alpha \qquad S_n \approx 1+\alpha/2$$

Performance of linear probing (dashed curves) with what would be expected from more random collision resolution.

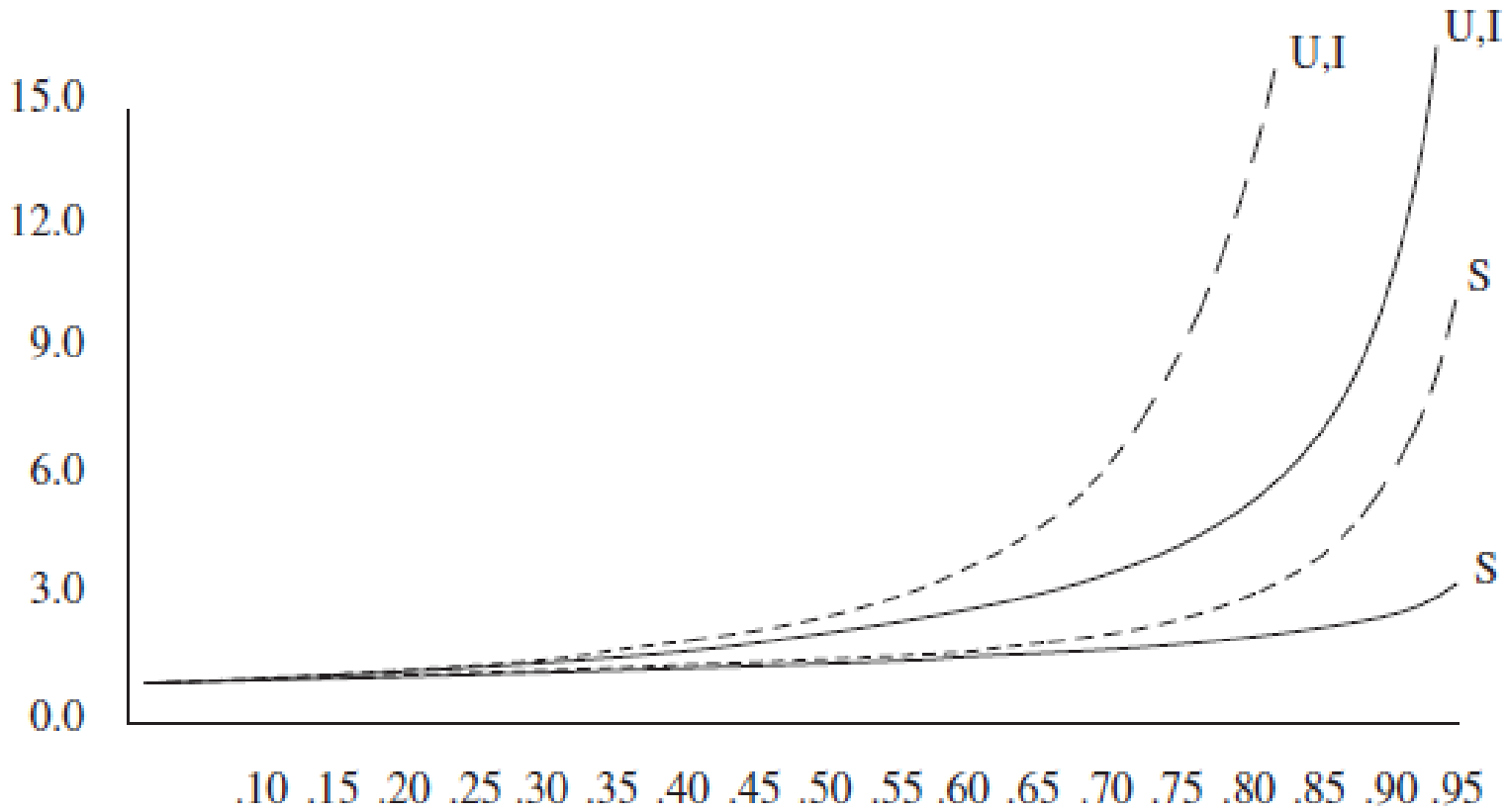


Figure 5.12 Number of probes plotted against load factor for linear probing (dashed) and random strategy (*S* is successful search, *U* is unsuccessful search, and *I* is insertion)

#2: Quadratic Probing

- One of the problems of linear open addressing is that it tends to create clusters of identifiers.
- These clusters tend to merge as more identifiers are entered, leading to bigger clusters.
- A quadratic probing scheme improves the growth of clusters. A quadratic function of i is used as the increment when searching through buckets.
- Perform search by examining bucket h(x), $(h(x)+i^2)\%b$, $(h(x)-i^2)\%b$ for $1 \leq i \leq (b-1)/2$.
- When b is a prime number of the form 4j+3, for j an integer, the quadratic search examine every bucket in the table.

Quadratic Probing

$$f(i) = i^2$$

- 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

Less likely to encounter Primary Clustering

#2: Quadratic Probing

- hi(x) = {Hash(x) + f(i) }mod Hsize, with f(0)=0, Let **f(i) = i^2**
- Inserting the keys{ 89, 18,49,58,69} in order

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Quadratic Probing: Success guarantee for $\alpha < ½$

- If **size** is prime and $\alpha < ½$, then quadratic probing will find an empty slot in size/2 probes or fewer.
- – show for all $\mathbf{0 \le i, j \le size/2}$ and $\mathbf{i \ne j}$

$$\mathbf{(h(x) + i2) \bmod size \ne (h(x) + j2) \bmod size}$$

- – by contradiction: suppose that for some i ≠ j:

$$\mathbf{(h(x) + i2) \bmod size = (h(x) + j2) \bmod size}$$

- $\mathbf{\Rightarrow i2 \bmod size = j2 \bmod size}$

Quadratic Probing: Properties

- For *any* $\alpha < ½$, quadratic probing will find an empty slot; for bigger α, quadratic probing *may* find a slot
- Quadratic probing does not suffer from *primary* clustering: keys hashing to the same *area* are not bad
- But what about keys that hash to the same *spot*? – ***Secondary Clustering!***

Quadratic Probing:

Theorem 5.1

If quadratic probing is used, and the table size is **prime**, then a new element can always be inserted if the table is at least half empty.

#3: Rehashing

- Another way to control the growth of clusters is to use a series of hash functions $h_1, h_2, \ldots, h_m$. This is called rehashing.
- Buckets $h_i(x)$, $1 \leq i \leq m$ are examined in that order.
- Table size : $m > n$; where n is no of item to be inserted

- For small load factor the performance is much better,
- than for n/m close to one.

Best choice: $n/m = 0.5$

- When $n/m > 0.75$ - **rehashing**

#3: Rehashing

- Build a second table twice as large as the original and rehash there all the keys of the original table.
- Expensive operation,
- running time O(N)
- However, once done, the new hash table will have good performance.

#3: Rehashing

- Hash table with linear probing with input 13, 15, 6, 24 are inserted into a linear probing hash table of size 7.
- The hash function is $h(x) = x \bmod 7$.

0	6
1	15
2	
3	24
4	
5	
6	13

0	6
1	15
2	23
3	24
4	
5	
6	13

- If 23 is inserted into the table, the resulting table will be over 70 percent full. Because the table is so full, a new table is created

#3: Rehashing

- The size of this table is **17**, because this is the first prime that is twice as large as the old table size.
- The new hash function is then $h(x) = x \bmod 17$. The old table is scanned, and elements 6, 15, 23, 24, and 13 are inserted into the new table.

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

Rehashing

- Rehashing can be implemented in several ways with quadratic probing.

1. One alternative is to rehash as soon as the table is half full.
2. The other extreme is to rehash only when an insertion fails.
3. A third, middle-of-the-road strategy is to rehash when the table reaches a certain load factor.

- Since performance does degrade as the **load factor increases**, the third strategy, implemented with a good cutoff, could be best.

#4: Double Hashing

- Linear probing collision resolution leads to clusters in the table, because if two keys collide, the next position probed will be the same for both of them.
- The idea of double hashing: Make the offset to the next position probed depend on the key value, so it can be different for different keys
  - Need to introduce a second hash function $h_2(K)$, which is used as the offset in the probe sequence (think of linear probing as double hashing with $h_2(K) == 1$)
  - For a hash table of size M, $h_2(K)$ should have values in the range 1 through M-1; if M is prime, one common choice is h2(K) = 1 + ((K/M) mod (M-1))

#4: Double Hashing

- The insert algorithm for double hashing is then:

  1. Set indx = h(K); offset = h_2(K)
  2. If table location indx already contains the key, no need to insert it. Done!
  3. Else if table location indx is empty, insert key there. Done!
  4. Else collision. Set indx = (indx + offset) mod M.
  5. If indx == h(K), table is full! (Throw an exception, or enlarge table.) Else go to 2.

- With prime table size, double hashing works very well in practice

#3: Double Hashing

hi(x) = {Hash(x) + f(i) }mod Hsize, with f(0)=0, and

f(i)= i * $h_2(x)$; Let $h_2(x)$ x mod 9

Inserting the keys{ 89, 18,49,58,69} in order, What if 99 to be inserted

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

#3: Double Hashing

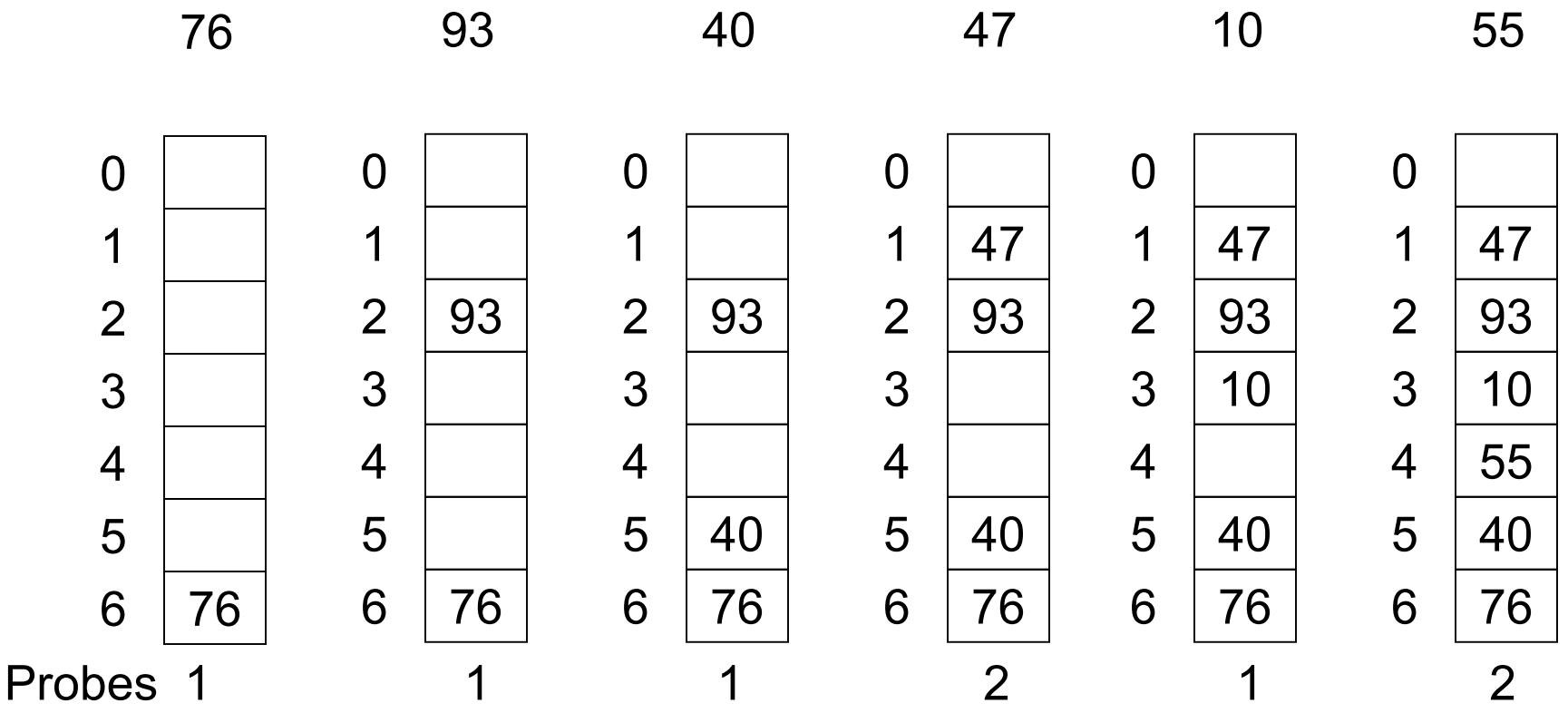
Inserting the keys{ 89, 18,49,58,69} in order, What if 99 to be inserted

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Double Hashing Example

- $f(i) = i * g(k)$, where g is a second hash function

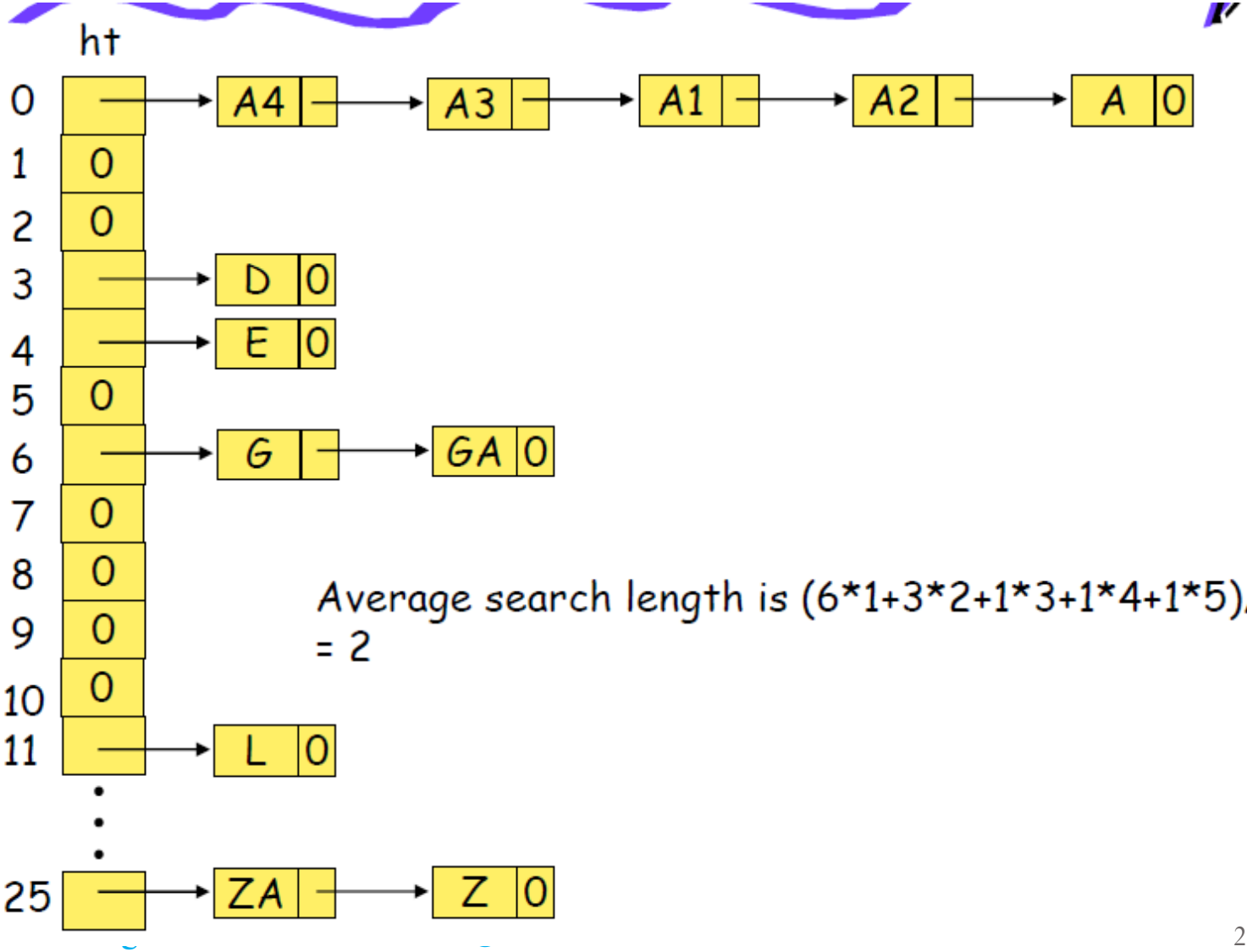
h(k) = k mod 7 and **g(k) = 5 – (k mod 5)**



Separate Chaining (Open Hashing)

- Linear probing perform poorly because the search for an identifier involves comparisons with identifiers that have different hash values.
  - e.g., search of ZA involves comparisons with the buckets ht[0] – ht[7] which are not possible of colliding with ZA.
- Unnecessary comparisons can be avoided if all the synonyms are put in the same list, where one list per bucket.
- As the size of the list is unknown before hand, it is best to use linked chain.
- Each chain has a head node. Head nodes are stored sequentially.

Hash Chain Example



Average search length is (6*1+3*2+1*3+1*4+1*5)/12
= 2

Separate Chaining (Cont.)

- The expected number of identifier comparisons can be shown to be ~ $1 + \alpha/2$, where α is the loading density n/b (b=number of head nodes). For α=0.5, it's 1.25. And if α=1, then it's about 1.5.
- Another advantage of this scheme is that only the b head nodes must be sequential and reserved at the beginning.
- The scheme only allocates other nodes when they are needed. This could reduce overall space requirement for some load densities, despite of links.

Analysis of Separate Chaining (Open Hashing)

- Open hashing is most appropriate when the hash table is kept in main memory, implemented with a standard in-memory linked list
- We hope that number of elements per bucket roughly equal in size, so that the lists will be short
- If there are n elements in set, then each bucket will have roughly n/b
- If we can estimate n and choose b to be roughly as large, then the average bucket will have only one or two members.
- b buckets, n elements in dictionary $\Rightarrow$ average n/b elements per bucket
- *insert, search, remove* operation take O(1+n/b) time each
- If we can choose b to be about n, constant time
- Assuming each element is likely to be hashed to any bucket, running time constant, independent of n
- Worst case performance is O(n) for both open and close hashing

Sample Problems:

1. Basic Hash Function

Problem: Given the hash function $h(x) = x \mod 10$ and a list of keys [12, 26, 33, 14, 25], determine the hash values. **Solution:**

- $h(12) = 2$
- $h(26) = 6$
- $h(33) = 3$
- $h(14) = 4$
- $h(25) = 5$

2. Collision Resolution Using Linear Probing

Problem: Using the same keys and hash function as above, resolve any collision by linear probing.

Solution:

- Initial hash values: [2, 6, 3, 4, 5]
- No collisions to resolve as all hash values are unique.

Sample Problems:

3. Collision Resolution Using Quadratic Probing

Problem: Use quadratic probing to resolve collisions for keys [12, 22, 32] with hash function $h(x) = x \mod 10$. **Solution:**

- $h(12) = 2$
- $h(22) = 2$ (Collision! Apply quadratic probing: $h(x) + (i^2 \mod 10), i = 1, h(22) = 3$)
- $h(32) = 2$ (Collision! $i = 2, h(32) = 6$)

4. Double Hashing

Problem: Use double hashing to resolve collisions for keys [34, 54, 24] with hash functions $h_1(x) = x \mod 10$ and $h_2(x) = 7 - (x \mod 7)$. **Solution:**

- $h_1(34) = 4$
- $h_1(54) = 4$ (Collision! $h(x) = (h_1(x) + i \cdot h_2(x)) \mod 10, i = 1, h(54) = (4 + 1 \cdot 3) \mod 10 = 7$)
- $h_1(24) = 4$ (Collision! $i = 2, h(24) = (4 + 2 \cdot 3) \mod 10 = 0$)

Sample Problems:

5. Load Factor Calculation

Problem: Calculate the load factor of a hash table with 10 slots and 6 entries. **Solution:**

- Load factor $\lambda = \frac{\text{number of entries}}{\text{number of slots}} = \frac{6}{10} = 0.6$

6. Chain Hashing

Problem: For keys [15, 25, 35, 45] using a hash function $h(x) = x \mod 10$, create a chaining hash table. **Solution:**

- $h(15) = 5, h(25) = 5, h(35) = 5, h(45) = 5$
- All keys hash to the same value, creating a chain at index 5: [15 -> 25 -> 35 -> 45]

7. Rehashing Threshold

Problem: A hash table has a maximum load factor of 0.75. If it currently has 12 slots and 9 entries, determine if rehashing is necessary after adding one more entry. **Solution:**

- New load factor after adding an entry: $\frac{10}{12} = 0.83$
- Since 0.83 > 0.75, rehashing is necessary.

Sample Problems:

8. Hash Function Design

Problem: Design a hash function for a set of strings where the hash value is determined by adding ASCII values of characters modulo 20. **Solution:**

- Example for "ace": $h(\text{"ace"}) = (97 + 99 + 101) \mod 20 = 297 \mod 20 = 17$

9. Perfect Hashing Scenario

Problem: Given keys [3, 5, 9], create a perfect hash function without collisions. **Solution:**

- $h(x) = x \mod 6$ yields unique hash values for all keys without collisions.

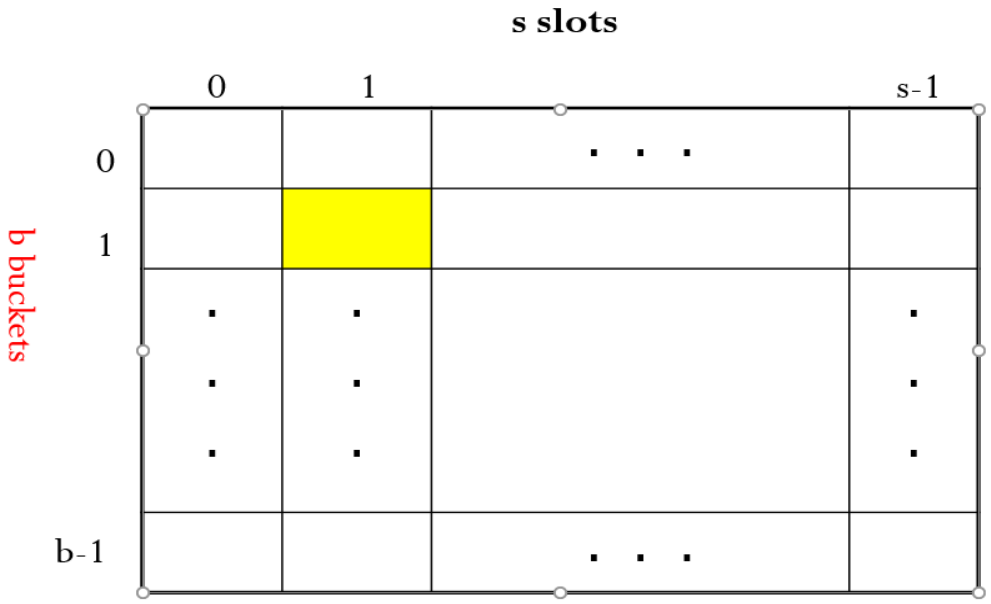
10. Universal Hashing

Problem: Implement a simple universal hash function $h(x) = ((ax + b) \mod p) \mod m$ where p is a prime larger than the largest key, a and b are random numbers, and m is the table size. Use $a = 3$, $b = 2$, $p = 11$, and $m = 5$ for the key $x = 7$. **Solution:**

- $h(7) = ((3 \cdot 7 + 2) \mod 11) \mod 5 = (23 \mod 11) \mod 5 = 1 \mod 5 = 1$

Theoretical Evaluation of Overflow Techniques

- In general, hashing provides pretty good performance over conventional techniques such as balanced tree, however, the worst-case performance of hashing can be O(n).
- Let ht[0..b-1] be a hash table with b buckets. If n identifiers x_1, x_2, …, x_n are entered into the hash table, then there are b^n distinct hash sequences $h(x_1), h(x_2), \ldots, h(x_n)$.



Theoretical Evaluation of Overflow Techniques (Cont.)

- S_n is the average number of comparisons needed to find the j^{th} key x_j, averaged over $1 \le j \le n$, with each j equally likely, and averaged over all b^n hash sequences, assuming each of these also to be equally likely.
- U_n is the expected number of identifier comparisons when a search is made for an identifier not in the hash table.

Theorem 8.1: Let a=n/b be the loading density of a hash table using a uniform hashing function h. Then

(1) for linear open addressing

$$U_n \approx \frac{1}{2}[1+\frac{1}{(1-\alpha)^2}] \qquad S_n \approx \frac{1}{2}[1+\frac{1}{1-\alpha}]$$

(2) for rehashing, random probing, and quadratic probing

$$U_n \approx 1/(1-\alpha) \qquad S_n \approx -[\frac{1}{\alpha}]\log_e(1-\alpha)$$

(3) for chaining

$$U_n \approx \alpha \qquad S_n \approx 1+\alpha/2$$

Theoretical Evaluation of Overflow Techniques (Cont.)

Theoretical Evaluation of Overflow Techniques (Cont.)

Theorem 8.1: Let a=n/b be the loading density of a hash table using a uniform hashing function h. Then

(1) for linear open addressing

$$U_n \approx \frac{1}{2}[1+\frac{1}{(1-\alpha)^2}] \qquad S_n \approx \frac{1}{2}[1+\frac{1}{1-\alpha}]$$

(2) for rehashing, random probing, and quadratic probing

$$U_n \approx 1/(1-\alpha) \qquad S_n \approx -[\frac{1}{\alpha}]\log_e(1-\alpha)$$

(3) for chaining

$$U_n \approx \alpha \qquad S_n \approx 1+\alpha/2$$

- Proof:

Comparison: Average number of bucket accesses per identifier Retrieved

- If **open addressing** is used, then each table slot holds at
- most one element, therefore, the loading factor α can **never** be greater than 1.
- If **external chaining** is used, then each table slot can hold many elements, therefore, the loading factor **may be** greater than 1.

$\alpha = \frac{n}{b}$	.50		.75		.90		.95	
Hash Function	Chain	Open	Chain	Open	Chain	Open	Chain	Open
mid square	1.26	1.73	1.40	9.75	1.45	37.14	1.47	37.53
division	1.19	4.52	1.31	7.20	1.38	22.42	1.41	25.79
shift fold	1.33	21.75	1.48	65.10	1.40	77.01	1.51	118.57
bound fold	1.39	22.97	1.57	48.70	1.55	69.63	1.51	97.56
digit analysis	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
theoretical	1.25	1.50	1.37	2.50	1.45	5.50	1.48	10.50

(Adapted from V. Lum, P. Yuen, and M. Dodd, *CACM*, 1971, Vol. 14, No. 4)

Applications of Hashing

- **Compilers** use hash tables to keep track of declared variables
- A hash table can be used for **on-line spelling checkers** — if misspelling detection (rather than correction) is important, an entire dictionary can be hashed and words checked in constant time
- **Game playing** programs use hash tables to store seen positions, thereby saving computation time if the position is encountered again
- Hash functions can be used to quickly check for **inequality** — if two elements hash to different values they must be different

An analysis of separate chaining:

Advantages:

1. **Simple Implementation**: Implementing a hash table using separate chaining is relatively straightforward. Each slot in the table simply holds a reference to the head of a linked list.
2. **Handles High Load Factors Well**: Separate chaining can still perform effectively even when the load factor (ratio of number of elements to table size) exceeds 1.
3. **Efficient Deletions**: Deleting an entry from a hash table implemented using separate chaining can be efficient, especially when compared to open addressing schemes where deleted slots need special handling.
4. **Dynamically Sized**: Since each slot is a linked list, the table can handle a variable number of collisions without needing to be resized.

An analysis of separate chaining:

Disadvantages:

- **Memory Overhead**: Each entry in a hash table using separate chaining typically requires additional memory overhead due to the linked list pointers.
- **Cache Performance**: Since linked lists are not cache-friendly data structures (due to their non-contiguous memory allocation), hash tables implemented using separate chaining might exhibit poor cache performance, especially when compared to linearly probed open addressing hash tables.
- **Potential for Worst-case Scenarios**: If the hash function is poorly designed, many keys could hash to the same slot, resulting in a linked list that is very long, which would undermine the average-case performance of $O(1)$ and bring it closer to $O(n)$.

Performance Analysis of separate chaining:

- **Best-case & Average-case Scenario**: If the hash function is uniform and distributes keys evenly across slots, and the load factor α (number of elements/table size) is kept under control, then the average length of a linked list will be α. Thus, operations like insertion, deletion, and lookup will take $O(\alpha)$ time on average. Ideally, for good average-case performance, α should be kept close to 1 or less.
- **Worst-case Scenario**: In the worst case, all keys could hash to the same slot, making the length of a single linked list equal to the number of keys, n. This would make operations like insertion, deletion, and lookup take $O(n)$ time. However, such a worst-case scenario is indicative of a poor hash function.

Optimizing the performance of separate chaining:

- **Good Hash Function**: Always use a good hash function that distributes keys uniformly across the hash table.
- **Resizing**: Implement dynamic resizing of the hash table when the load factor exceeds a certain threshold to ensure that the linked lists' average length remains reasonable.
- **Alternative Structures**: Instead of linked lists, one could also use other data structures like balanced trees. This would guarantee a worst-case time complexity of $O(\log n)$ for operations, but this is typically more complex to implement than a simple linked list.

A sample problem with separate chaining:

Consider a dynamic hashing scheme that starts with a directory of depth $D = 1$ and uses the first D bits of a hash function's output to index the directory. Each slot in the directory points to a bucket. Instead of each bucket having a fixed size, we utilize separate chaining, so each bucket can have an unlimited number of elements but uses a linked list.

The hash function takes a key and outputs a 4-bit binary value.

Given the following keys and their respective 4-bit hash values, insert them into the hash table:

Keys: A, B, C, D, E, F

Hash Values: $1100, 1001, 1100, 0110, 1101, 1001$

Insert these keys sequentially and show the state of the directory and buckets after each insertion.

Summary

- Hash tables can be used to implement the insert and find operations in constant average time. it depends on the load factor not on the number of items in the table.
- It is important to have a prime **TableSize** and a correct choice of **load factor** and **hash function**.
- For separate chaining the load factor should be close to 1.
- For open addressing load factor should not exceed 0.5 unless this is completely unavoidable.
- Rehashing can be implemented to grow (or shrink) the table.
- Hashing is one of the most important data structures.
- Hashing has many applications where operations are limited to find, insert, and delete.

Summary

- The main **tradeoffs** between these methods are that **linear probing** has the best cache performance but is most sensitive to clustering, while **double hashing** has poorer cache performance but exhibits virtually no clustering; **quadratic probing** falls in between the previous two methods.

Deficiencies of Static Hashing

- In static hashing, function ***h*** maps search-key values to a fixed set of ***b*** of bucket addresses.
  - Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be wasted initially.
  - If database shrinks, again space will be wasted.
  - One option is periodic re-organization of the file with a new hash function, but it is very expensive.
- These problems can be avoided by using techniques that allow the **number of buckets** to be modified dynamically.

Deficiencies of Static Hashing

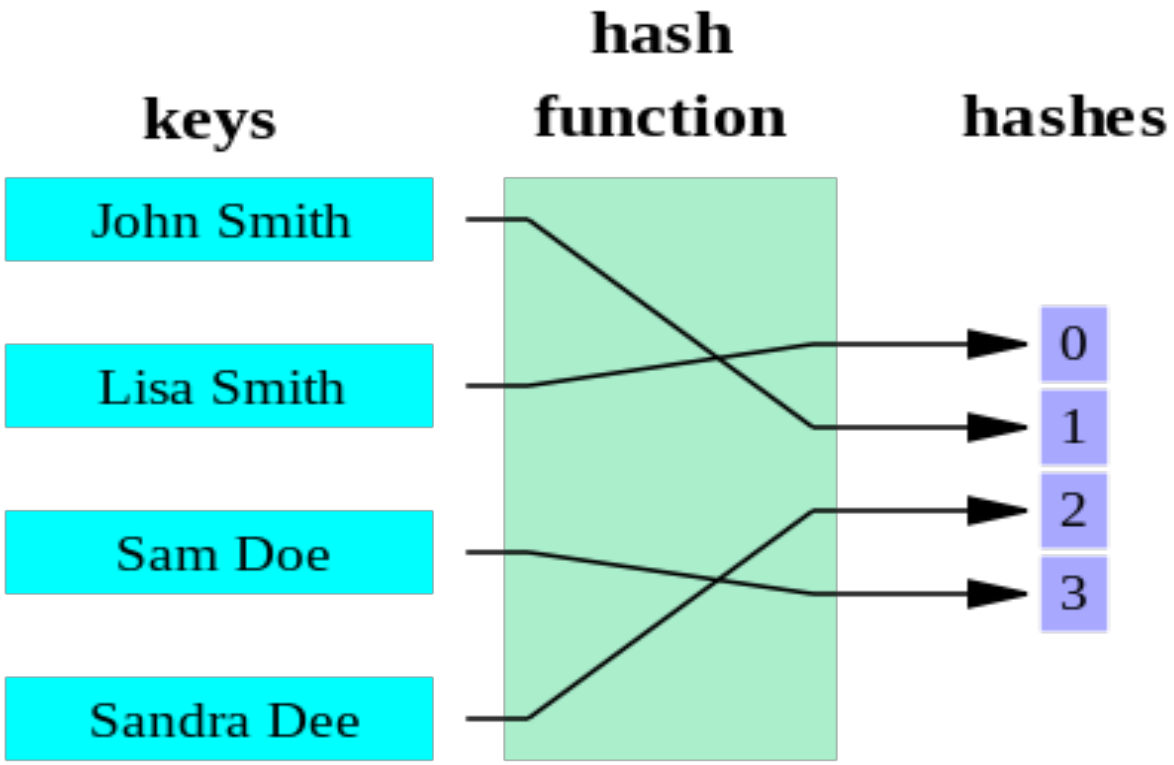
…These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically.

- This is called **dynamic hashing**.
- There are several types of dynamic hashing, we will learn about **extendible** hashing, and **linear** hashing.

Perfect Hashing

- 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. In mathematical terms, it is a total *injective function*.
- A perfect hash function for a specific set S that can be evaluated in constant time, and with values in a small range, can be found by a **randomized algorithm** in a number of operations that is proportional to the size of S.

Perfect Hashing



Perfect hash function

- A *perfect hash function* is one that maps the set of actual key values to the table **without any collisions**.
- A ***minimal perfect hash function*** does so using a table that has only as many slots as there are key values to be hashed.
- If the set of keys IS known in advance, it is possible to construct a specialized hash function that is perfect, perhaps even minimal perfect.
- Algorithms for constructing perfect hash functions tend to be tedious, but a number are known.

Perfect hash function: Cichelli's Method

This is used primarily when it is necessary to hash a relatively small collection of keys, such as the set of reserved words for a programming language.

The basic formula is:

```
h(S) = S.length() + g(S[0]) + g(S[S.length()-1])
```

where `g()` is constructed using Cichelli's algorithm so that h() will return a different hash value for each word in the set.

The algorithm has three phases:

- computation of the letter frequencies in the words
- ordering the words
- searching

Perfect hash function: Cichelli's Method

Suppose we need to hash the words in the list below:

```
calliope
clio
erato
euterpe
melpomene
polyhymnia
terpsichore
thalia
urania
```

Determine the frequency with which each first and last letter occurs:

letter:	e	a	c	o	t	m	p	u
freq:	6	3	2	2	2	1	1	1

Score the words by summing the frequencies of their first and last letters, and then sort them in that order:

calliope	8
clio	4
erato	8
euterpe	12
melpomene	7
polyhymnia	4
terpsichore	8
thalia	5
urania	4

```
euterpe
calliope
erato
terpsichore
melpomene
thalia
clio
polyhymnia
urania
```

Perfect hash function: Cichelli's Method

Finally, consider the words in order and define `g(x)` for each possible first and last letter in such a way that each of the words will have a distinct hash value:

word	g_value assigned	h(word)	table slot
euterpe	e-->0	7	7 ok
calliope	c-->0	8	8 ok
erato	o-->0	5	5 ok
terpsichore	t-->0	11	2 ok
melpomene	m-->0	9	0 ok
thalia	a-->0	6	6 ok
clio	none	4	4 ok
polyhymnia	p-->0	10	1 ok
urania	u-->0	6	6 reject
	u-->1	7	7 reject
	u-->2	8	8 reject
	u-->3	9	0 reject
	u-->4	10	1 reject

Perfect hash function: Cichelli's Method

Cichelli's method imposes a limit on the search at this point (we're assuming it's 5 steps), and so we back up to the previous word and redefine the mapping there:

word	g_value assigned	h(word)	table slot	
polyhymnia	p-->0	10	1	reject
	p-->1	11	2	reject
	p-->2	12	3	
urania	u-->0	6	6	reject
	u-->1	7	7	reject
	u-->2	8	8	reject
	u-->3	9	0	reject
	u-->4	10	1	ok

So, if we define `g()` as determined above, then `h()` will be a minimal perfect hash function on the given set of words.

The primary difficulty is the cost, because the search phase can degenerate to exponential performance, and so it is only practical for small sets of words.

Perfect Hashing

- The minimal size of the description of a perfect hash function depends on the range of its function values: The smaller the range, the more space is required. Any perfect hash functions suitable for use with a hash table require at least a number of bits that is proportional to the size of S.
- A perfect hash function with values in a limited range can be used for efficient lookup operations, by placing keys from S (or other associated values) in a table indexed by the output of the function.
- Using a perfect hash function is best in situations where there is a frequently queried large set, S, which is seldom updated.
- Efficient solutions to performing updates are known as dynamic perfect hashing, but these methods are relatively complicated to implement.
- A simple alternative to perfect hashing, which also allows **dynamic updates**, is **cuckoo hashing**.

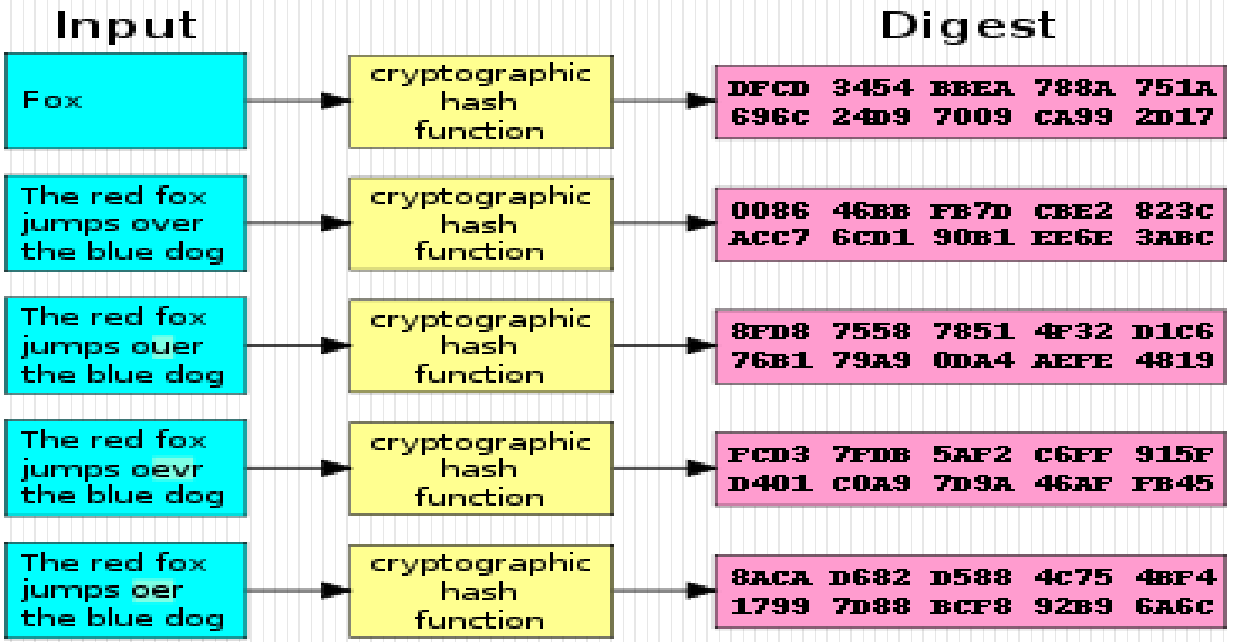
Minimal perfect hash function

- A **minimal perfect hash function** is a perfect hash function that maps n keys to n consecutive integers—usually [0…n−1] or [1…n]. A more formal way of expressing this is: Let j and k be elements of some finite set **K**. F is a minimal perfect hash function iff F(j) =F(k) implies j=k and there exists an integer a such that the range of F is a… a+|**K**|−1. It has been proved that a general purpose minimal perfect hash scheme requires at least 1.44 bits/key. However the smallest currently use around 2.5 bits/key.
- A minimal perfect hash function F is **order preserving** if keys are given in some order a_1, a_2, …, and for any keys a_j and a_k, j<k implies F(a_j)<F(a_k). Order-preserving minimal perfect hash functions require necessarily $\Omega(n \log n)$ bits to be represented.
- A minimal perfect hash function F is **monotone** if it preserves the lexicographical order of the keys.
- Monotone minimal perfect hash functions can be represented in very little space.

Perfect Hashing

Compute $x=h(k,s)$
Let $<p,i,r>$ be $H[x]$
If $r=0$ then
  Let y = the index of any free slot in D
  Store $<y,0,1>$ back into $H[x]$
  Store $<k,d>$ into $D[y]$
Otherwise:
  Let $<k_j,d_j>$ be $D[p+j]$ $0 \le j < r$
  Search to find an m such that the $r+1$ values
    $h_m(k_j,r+1)$ $0 \le j < r$,
    and $h_m(k,r+1)$
  are all distinct
  Let y= the index of the first of $r+1$ consecutive free slots in D
  Store $<y,m,r+1>$ in $H[x]$
  Store $<k,d>$ into $D[y+h_m(k,r+1)]$
  For $0 \le j < r$
    Move $<k_j,d_j>$ to $D[y+h_m(k_j,r+1)]$
  Mark the r slots $D[p]$ .. $D[p+r-1]$ as free

Cryptographic Hash Function

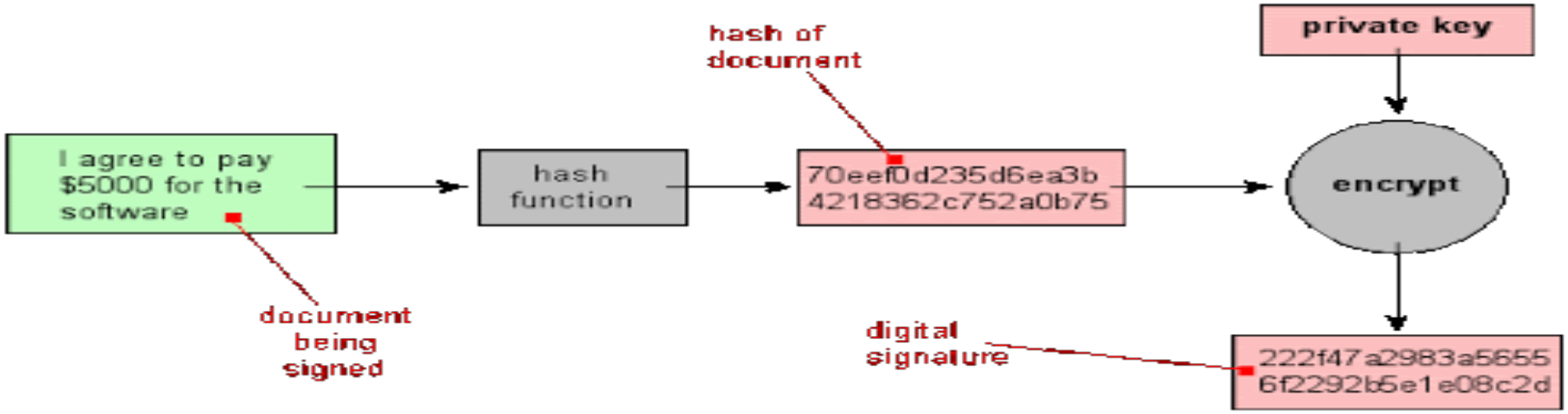


Introduction

- The origin of the term "hash" has non-technical meaning.
- Hans Peter Luhn (1896 – 1964), a computer scientist for IBM, was the first that used the term "hash" to explain that hash functions "chop" the input domain into many sub-domains that get "mixed" into the output range to improve the uniformity of the key distribution.
- Hash functions are used for password protecting because the main purpose for this issue is to encrypt passwords in a form that is unfeasible to decrypt and also confirm if the given password is correct.

Cryptographic Hash Function: Introduction

- Hash functions are commonly used data structures in computing systems for tasks, such as checking the integrity of messages and authenticating information. While they are considered cryptographically "weak" because they can be solved in polynomial time, they are not easily decipherable.
- Cryptographic hash functions add security features to typical hash functions, making it more difficult to detect the contents of a message or information about recipients and senders.

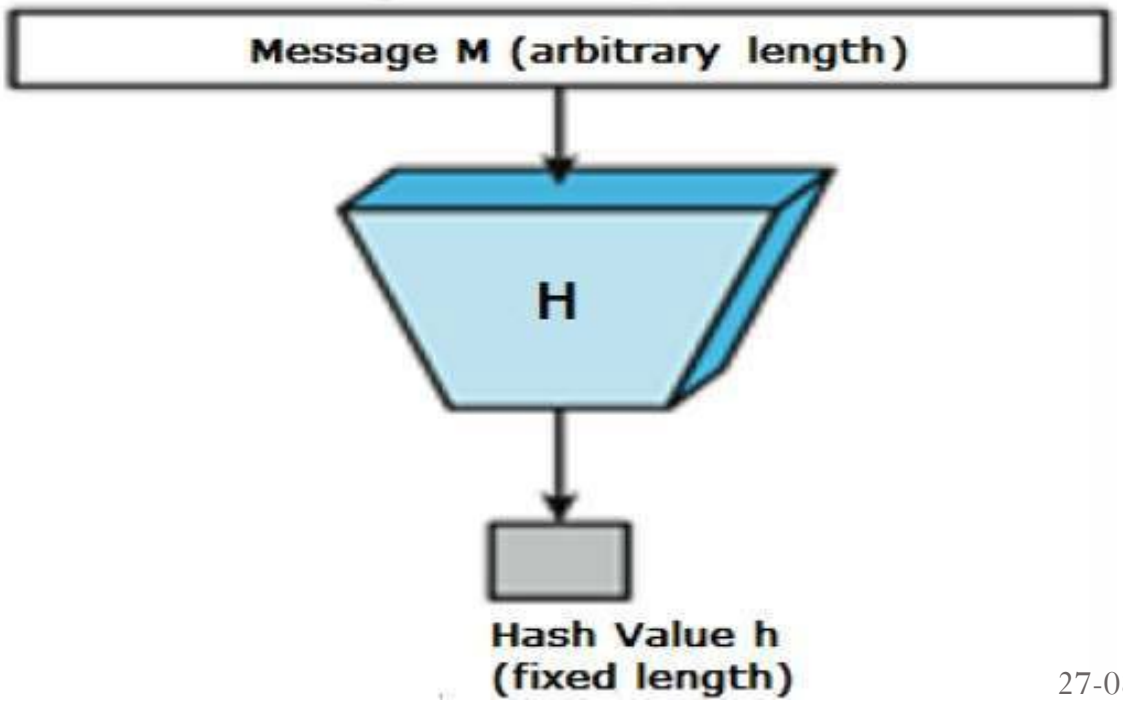


Cryptographic hash functions

- A hash function is a mathematical function that converts a numerical input value into another compressed numerical value. The input to the hash function is of arbitrary length but output is always of fixed length.
- A **cryptographic hash function** is an algorithm that takes an arbitrary amount of data input—a credential—and produces a fixed-size output of **enciphered text** called a hash value, or just "hash." That enciphered text can then be stored instead of the password itself, and later used to verify the user.

Cryptographic hash functions

- Hash functions are extremely useful and appear in almost all information security applications.
- Values returned by a hash function are called **message digest** or simply **hash values**. The following picture illustrated hash function –



Properties of cryptographic hash functions

- **Non-reversibility, or one-way function.** A good hash should make it very hard to reconstruct the original password from the output or hash.
- **Diffusion, or avalanche effect**. A change in just one bit of the original password should result in change to half the bits of its hash. In other words, when a password is changed slightly, the output of enciphered text should change significantly and unpredictably.
- **Determinism.** A given password must always generate the same hash value or enciphered text.
- **Collision resistance.** It should be hard to find two different passwords that hash to the same enciphered text.
- **Non-predictable.** The hash value should not be predictable from the password.

Features of Hash Functions

- **Fixed Length Output (Hash Value)**
  - Hash function coverts data of arbitrary length to a fixed length. This process is often referred to as **hashing the data**.
  - In general, the hash is much smaller than the input data, hence hash functions are sometimes called **compression functions**.
  - Since a hash is a smaller representation of a larger data, it is also referred to as a **digest**.
  - Hash function with n bit output is referred to as an **n-bit hash function**. Popular hash functions generate values between 160 and 512 bits.
- **Efficiency of Operation**
  - Generally for any hash function h with input x, computation of h(x) is a fast operation.
  - Computationally hash functions are much faster than a symmetric encryption.

Properties of Cryptographic Hash Functions

Pre-Image Resistance

- This property means that it should be computationally hard to reverse a hash function.
- In other words, if a hash function h produced a hash value z, then it should be a difficult process to find any input value x that hashes to z.
- This property protects against an attacker who only has a hash value and is trying to find the input.

Second Pre-Image Resistance

- This property means given an input and its hash, it should be hard to find a different input with the same hash.
- In other words, if a hash function h for an input x produces hash value h(x), then it should be difficult to find any other input value y such that $h(y) = h(x)$.
- This property of hash function protects against an attacker who has an input value and its hash, and wants to substitute different value as legitimate value in place of original input value.

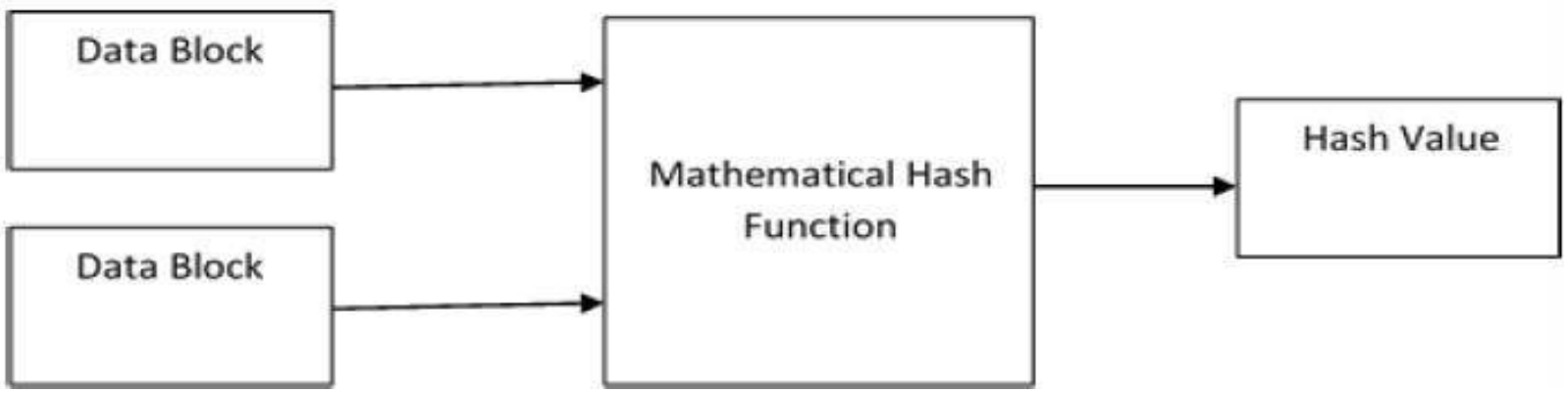
Properties of Cryptographic Hash Functions

Collision Resistance

- This property means it should be hard to find two different inputs of any length that result in the same hash. This property is also referred to as collision free hash function.
- In other words, for a hash function h, it is hard to find any two different inputs x and y such that $h(x) = h(y)$.
- Since, hash function is compressing function with fixed hash length, it is impossible for a hash function not to have collisions. This property of collision free only confirms that these collisions should be hard to find.
- This property makes it very difficult for an attacker to find two input values with the same hash.
- Also, if a hash function is collision-resistant **then it is second pre-image resistant.**

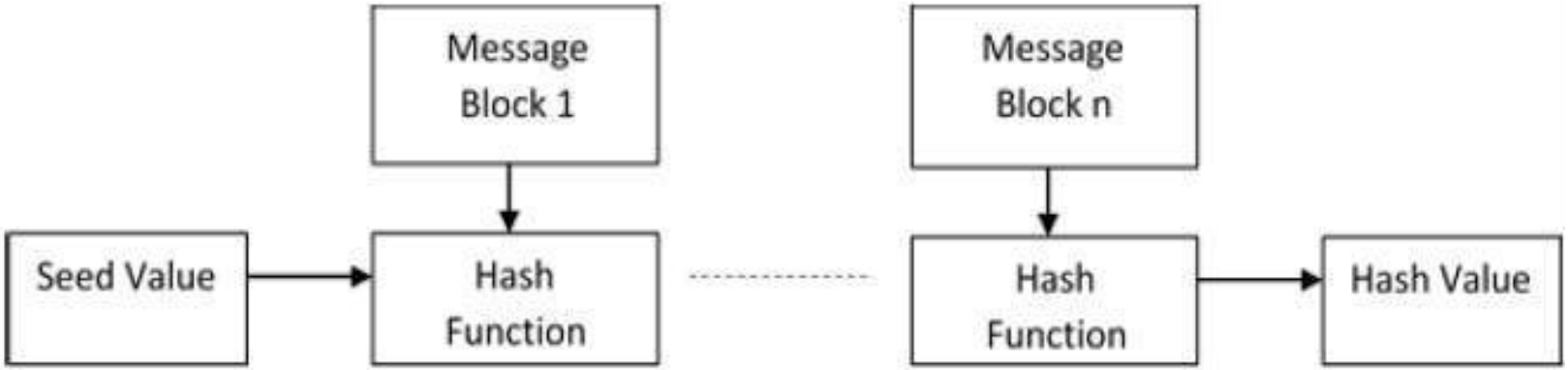
Design of Hashing Algorithms

- At the heart of a hashing is a mathematical function that operates on two fixed-size blocks of data to create a hash code. This hash function forms the part of the hashing algorithm.
- The size of each data block varies depending on the algorithm. Typically the block sizes are from 128 bits to 512 bits. The following illustration demonstrates hash function −



- Hashing algorithm involves rounds of above hash function like a block cipher. Each round takes an input of a fixed size, typically a combination of the most recent message block and the output of the last round

Design of Hashing Algorithms



- This process is repeated for as many rounds as are required to hash the entire message. Schematic of hashing algorithm is depicted above.
- Since, the hash value of first message block becomes an input to the second hash operation, output of which alters the result of the third operation, and so on. This effect, known as an **avalanche** effect of hashing.
- Avalanche effect results in substantially different hash values for two messages that differ by even a single bit of data.

Popular Cryptographic Hash Functions

Popular Cryptographic Hash Function

Message Digest (MD)

- MD5 was most popular and widely used hash function for quite some years.
- The MD family comprises of hash functions MD2, MD4, MD5 and MD6. It was adopted as **Internet Standard RFC 1321**. It is a 128-bit hash function.
- MD5 digests have been widely used in the software world to provide assurance about integrity of transferred file. For example, file servers often provide a pre-computed MD5 checksum for the files, so that a user can compare the checksum of the downloaded file to it.
- **In 2004, collisions were found in MD5**. An analytical attack was reported to be successful only in an hour by using computer cluster. This collision attack resulted in compromised MD5 and hence it is no longer recommended for use.

Popular Cryptographic Hash Function

Secure Hash Function (SHA)

- Family of SHA comprise of four SHA algorithms; SHA-0, SHA-1, SHA-2, and SHA-3. Though from same family, there are structurally different.
- The original version is SHA-0, a 160-bit hash function, was published by the National Institute of Standards and Technology (NIST) in 1993. It had few weaknesses and did not become very popular. Later in 1995, SHA-1 was designed to correct alleged weaknesses of SHA-0.
- SHA-1 is the most widely used of the existing SHA hash functions. It is employed in several widely used applications and protocols including Secure Socket Layer (SSL) security.
- In 2005, a method was found for uncovering collisions for SHA-1 within practical time frame making long-term employability of SHA-1 doubtful.
- SHA-2 family has four further SHA variants, SHA-224, SHA-256, SHA-384, and SHA-512 depending up on number of bits in their hash value. No successful attacks have yet been reported on SHA-2 hash function.
- Though SHA-2 is a strong hash function. Though significantly different, its basic design is still follows design of SHA-1. Hence, NIST called for new competitive hash function designs.
- In October 2012, the **NIST** chose the Keccak algorithm as the new SHA-3 standard. Keccak offers many benefits, such as efficient performance and good resistance for attacks.

Popular Cryptographic Hash Function

RIPEMD

- The RIPEMD is an acronym for RACE Integrity Primitives Evaluation Message Digest. This set of hash functions was designed by open research community and generally known as a family of European hash functions.
- The set includes RIPEMD, RIPEMD-128, and RIPEMD-160. There also exist 256, and 320-bit versions of this algorithm.
- Original RIPEMD (128 bit) is based upon the design principles used in MD4 and found to provide questionable security. RIPEMD 128-bit version came as a quick fix replacement to overcome vulnerabilities on the original RIPEMD.
- RIPEMD-160 is an improved version and the most widely used version in the family. The 256 and 320-bit versions reduce the chance of accidental collision, but do not have higher levels of security as compared to RIPEMD-128 and RIPEMD-160 respectively.

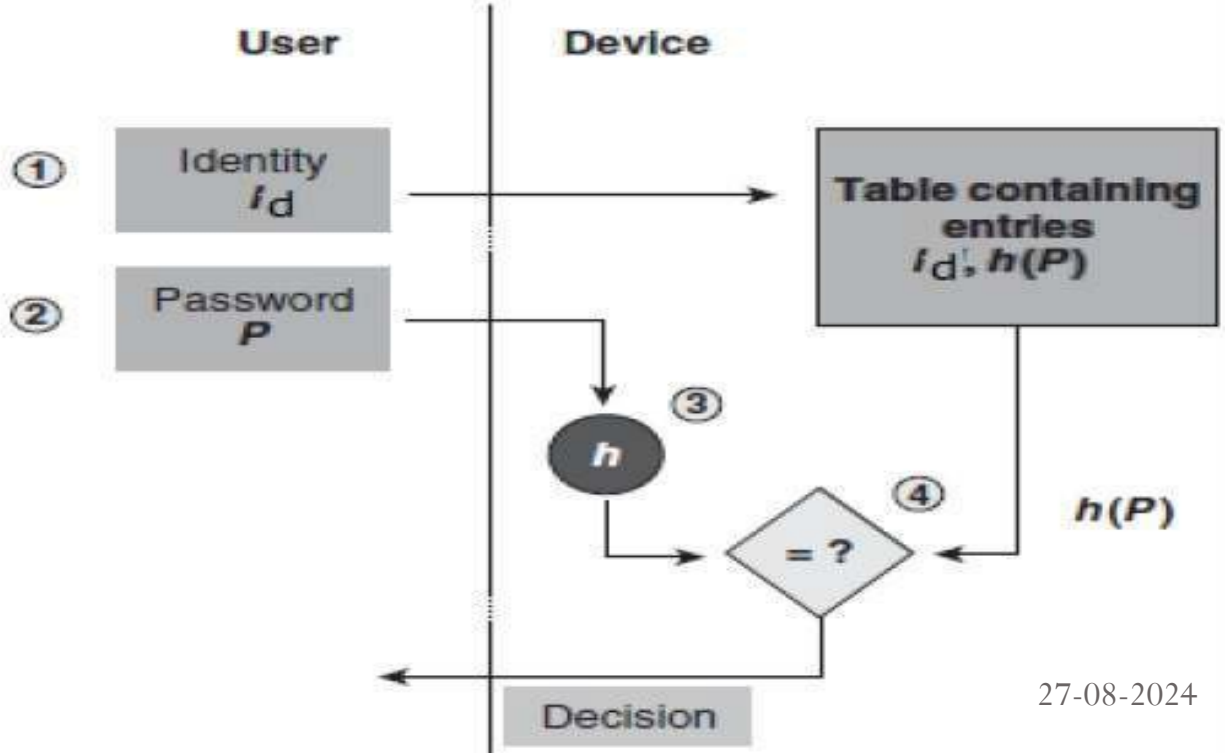
Popular Cryptographic Hash Function

Whirlpool

- This is a 512-bit hash function.
- It is derived from the modified version of Advanced Encryption Standard (AES). One of the designer was Vincent Rijmen, a co-creator of the AES.
- Three versions of Whirlpool have been released; namely WHIRLPOOL-0, WHIRLPOOL-T, and WHIRLPOOL.

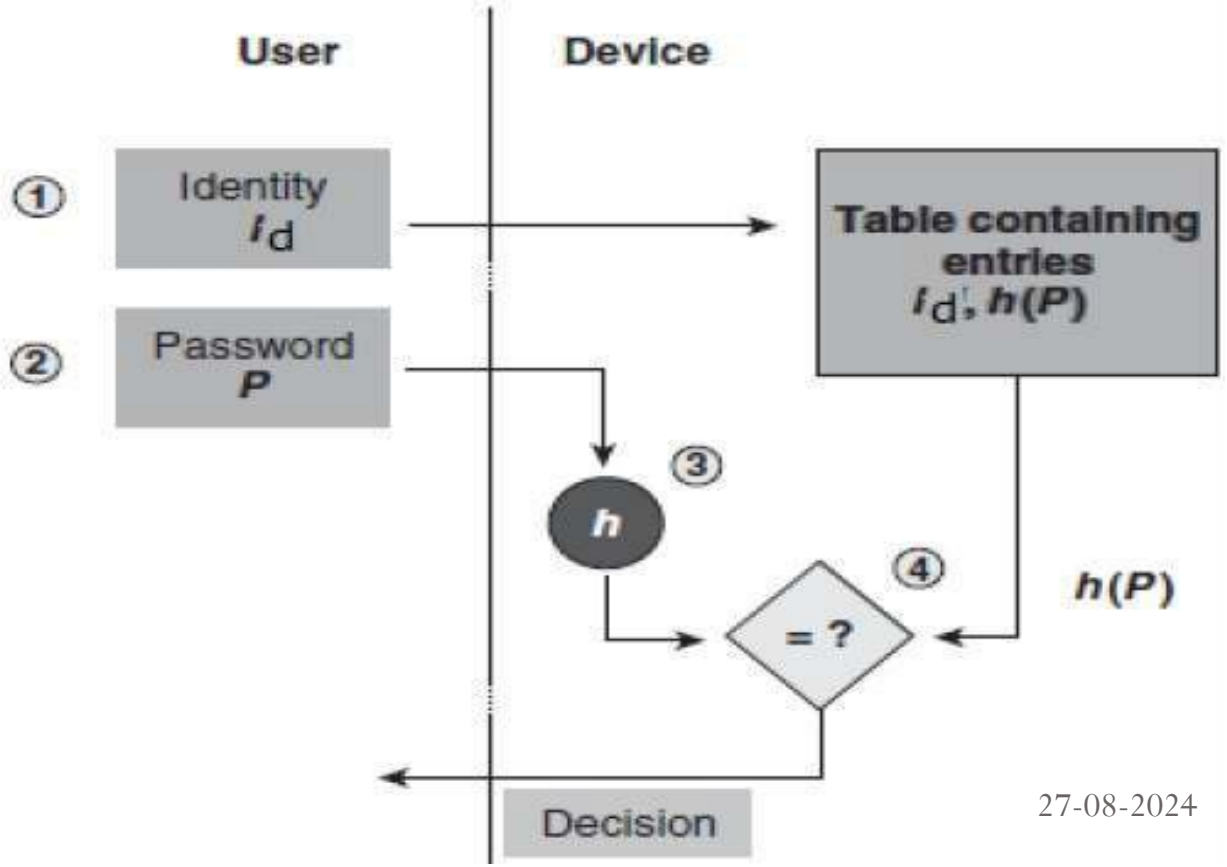
Applications of Hash Functions: password storage

- Hash functions provide protection to password storage.
- Instead of storing password in clear, mostly all logon processes store the hash values of passwords in the file.
- The Password file consists of a table of pairs which are in the form (user id, h(P)). The process of logon is depicted in the following illustration –



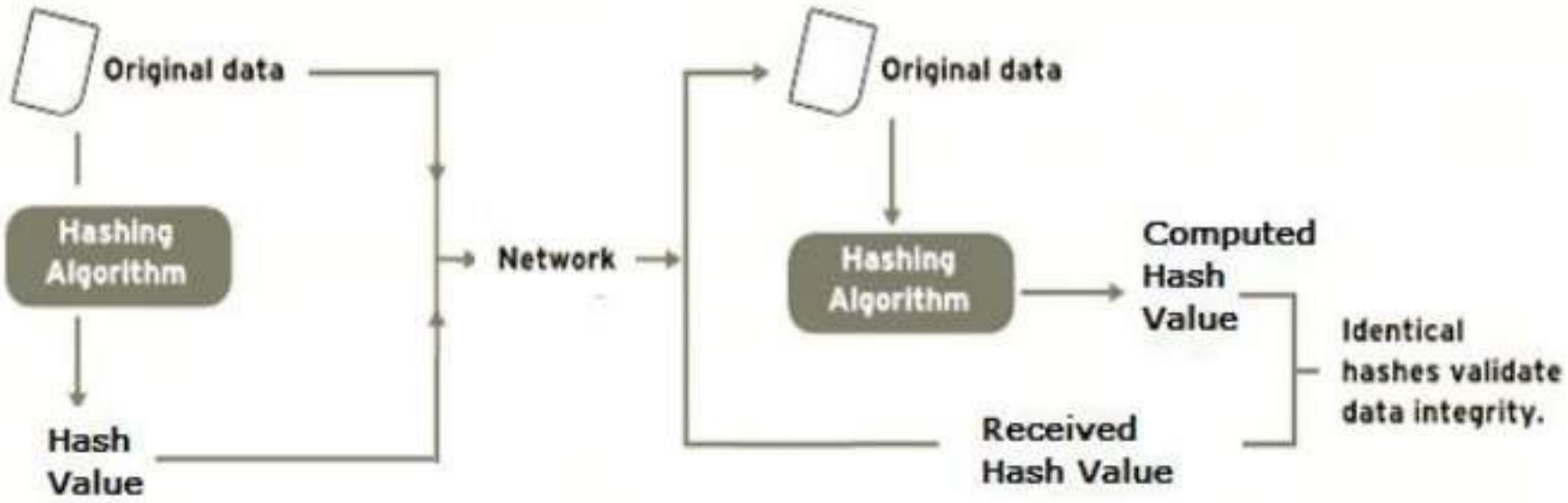
Applications of Hash Functions: password storage

- An intruder can only see the hashes of passwords, even if he accessed the password. He can neither logon using hash nor can he derive the password from hash value since hash function possesses the property of pre-image resistance.



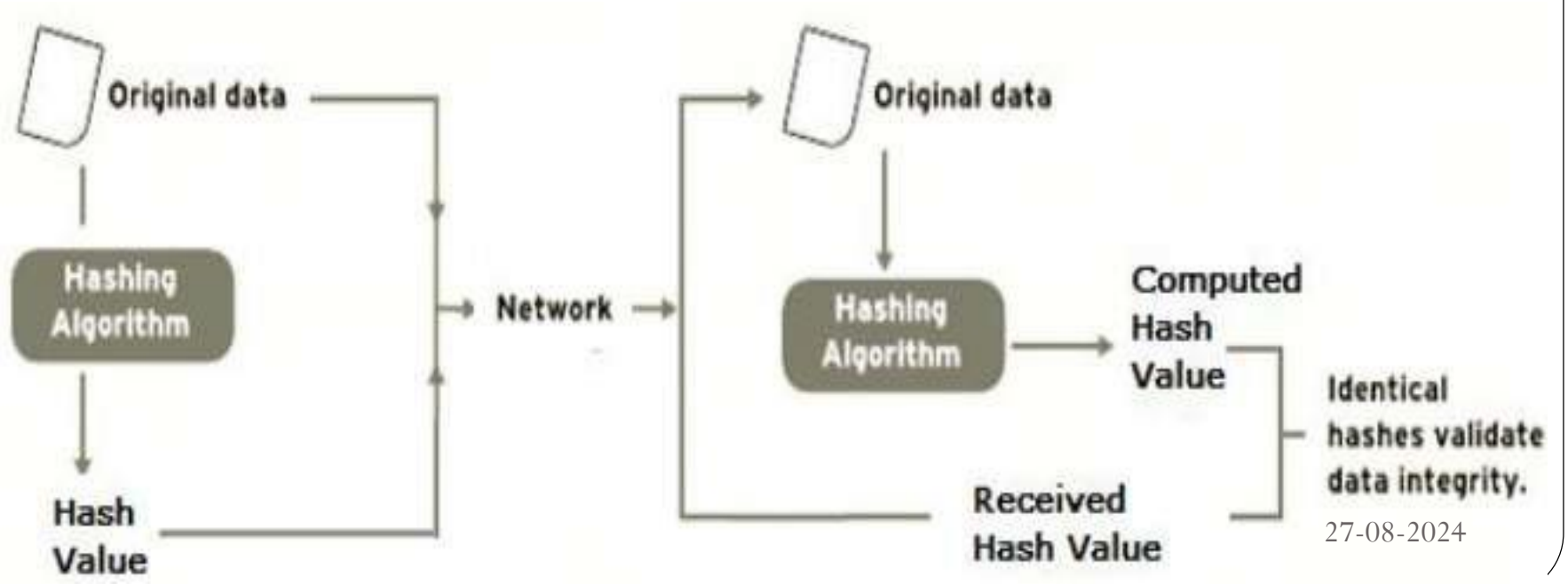
Applications of Hash Functions: Data Integrity Check

- Data integrity check is a most common application of the hash functions. It is used to generate the checksums on data files. This application provides assurance to the user about correctness of the data.
- The process is depicted in the following illustration –



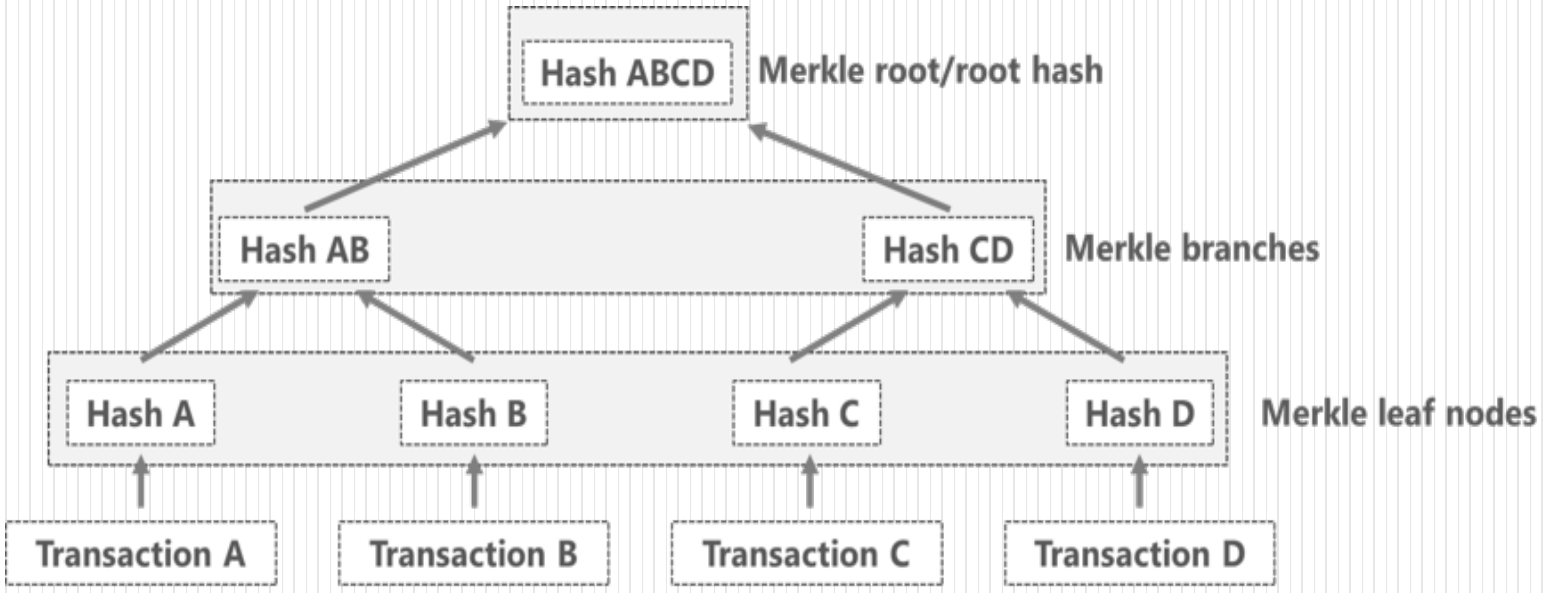
Applications of Hash Functions: Data Integrity Check

- The integrity check helps the user to detect any changes made to original file. It however, does not provide any assurance about originality.
- The attacker, instead of modifying file data, can change the entire file and compute all together new hash and send to the receiver. This integrity check application is useful only if the user is sure about the originality of file.



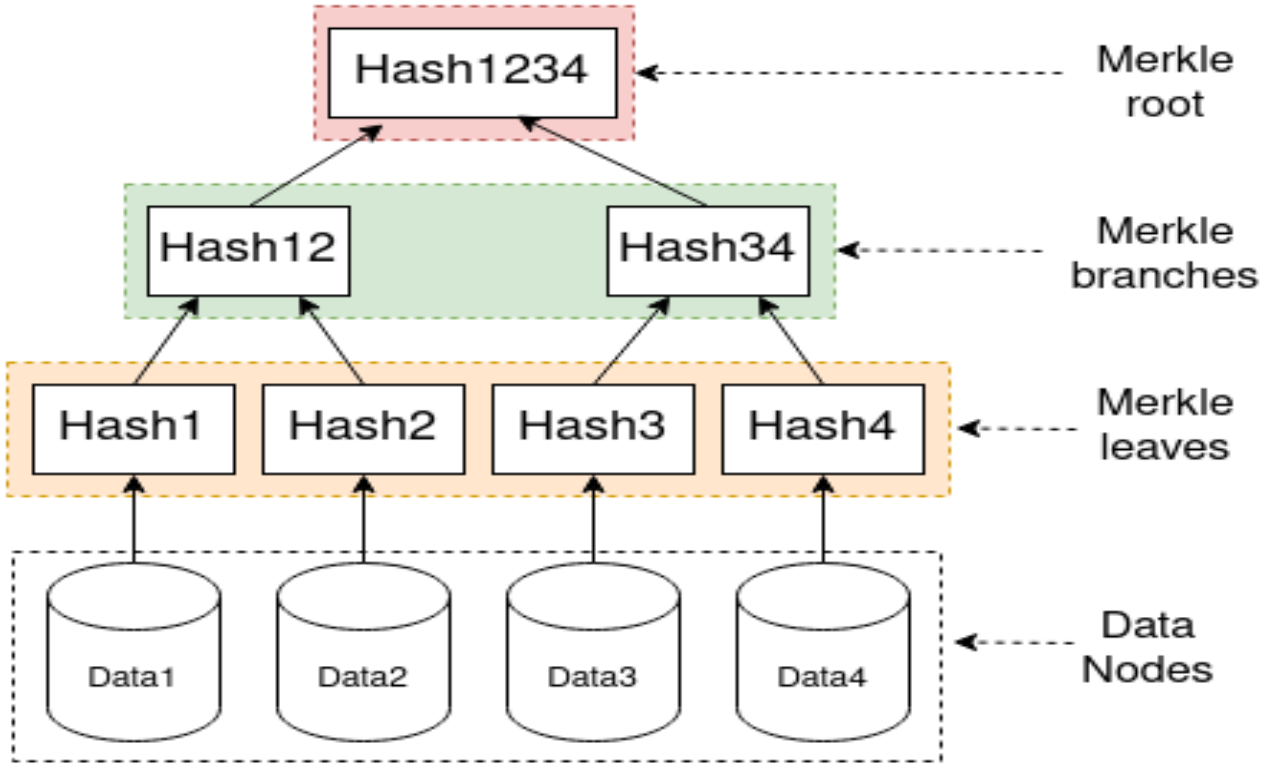
Merkle tree

[a hash-based data structure]



Merkle tree

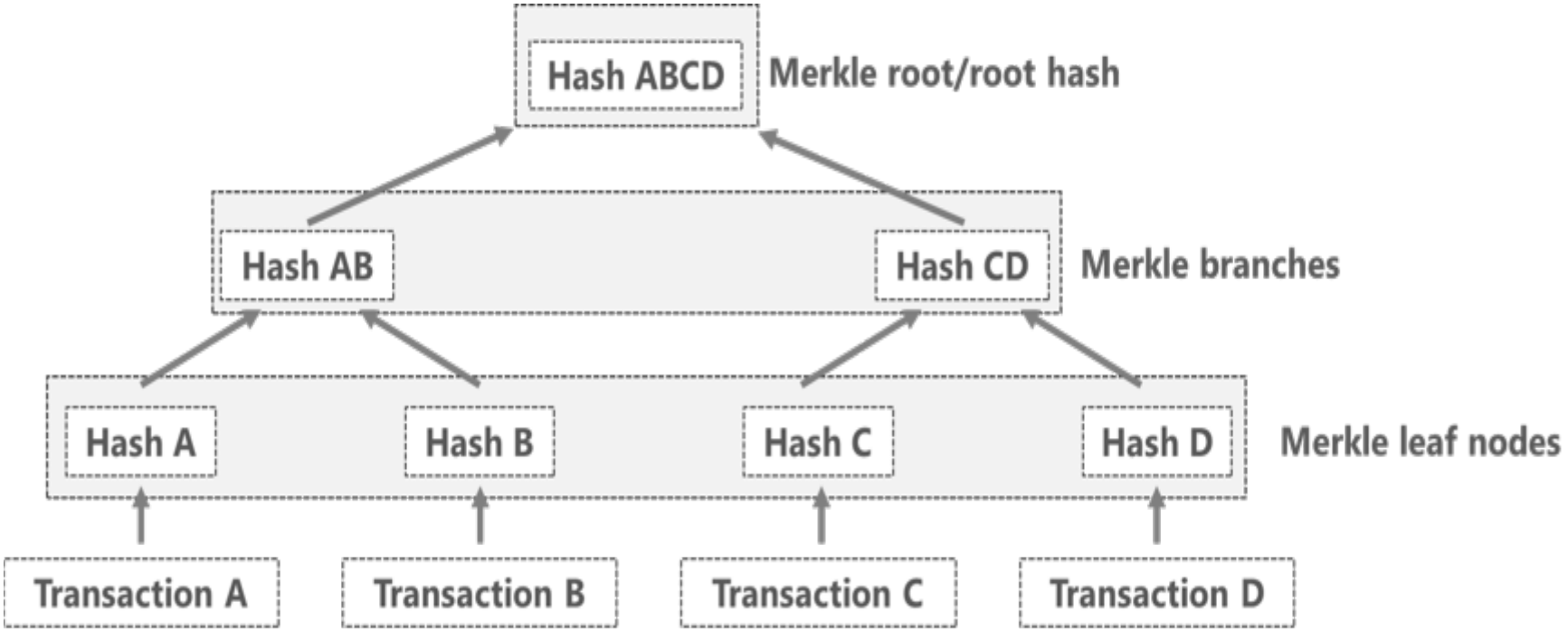
- Merkle tree is a mathematical **data structure** composed of hashes of different blocks of data, and which serves as a summary of all the transactions in a block.



Merkle tree: NIST Defination

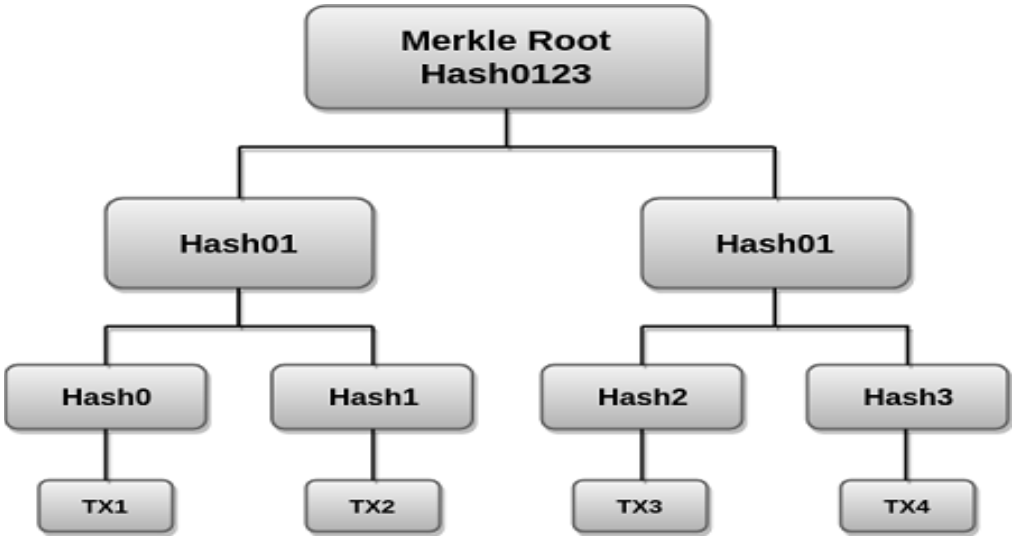
- **Definition:** A *tree* (usually a *binary tree*) in which each *internal node* has a *hash* of all the information in the *leaf* nodes under it.
- Specifically, each internal node has a hash of the information in its *children*.
- Each leaf has a hash of the block of information it represents. All leaf nodes are at the same *depth*. All nodes are as far left as possible.
- To add a block (leaf) to a *full* tree, a new *root* is created with the old root as its left child.
- Its right child is a degenerate tree (only left subtrees) all the way to the leaf level.

The Merkle tree of transactions A, B, C, and D; Hash ABCD is the Merkle root/root hash

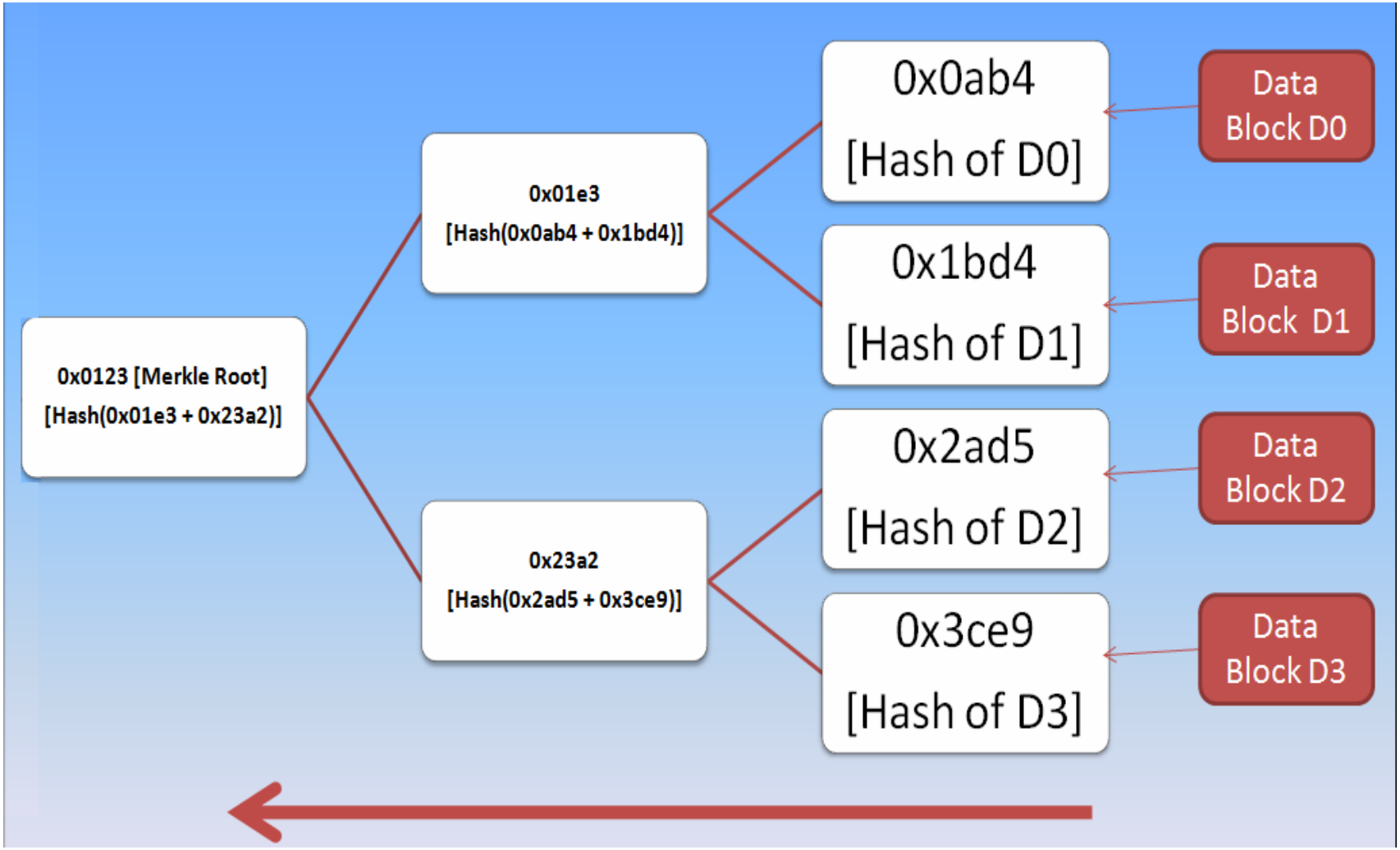


Merkle tree or Hash Tree

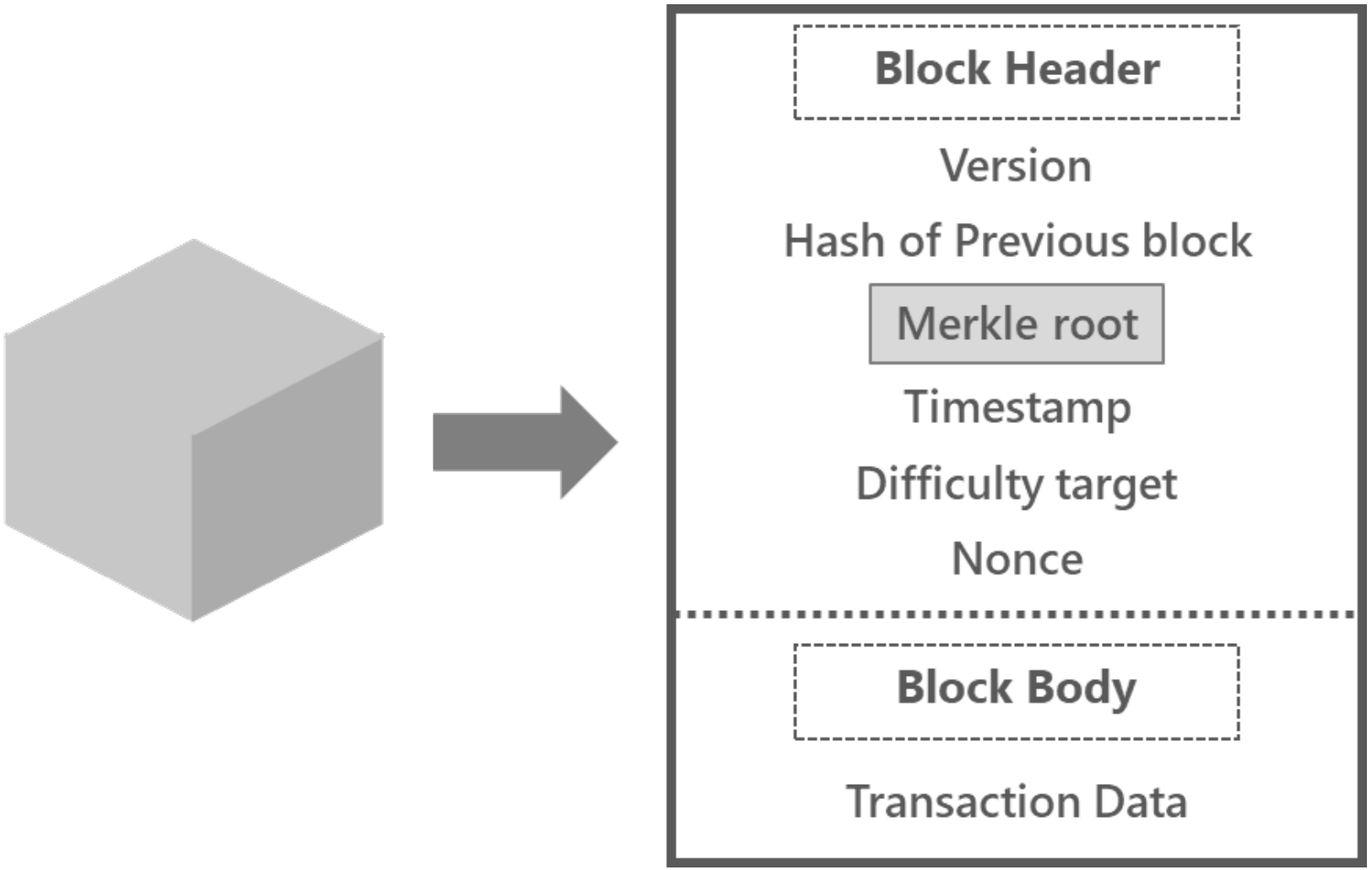
- Merkle tree is named after Ralph Merkle, it is a **tree data structure where non-leaf nodes are a hash of its child nodes and leaf nodes are a hash of a block of data**.
- It has a branching factor of 2 (it can have a maximum of 2 children).
- Merkel trees allow efficient and secure verification of the contents of a large data structure.



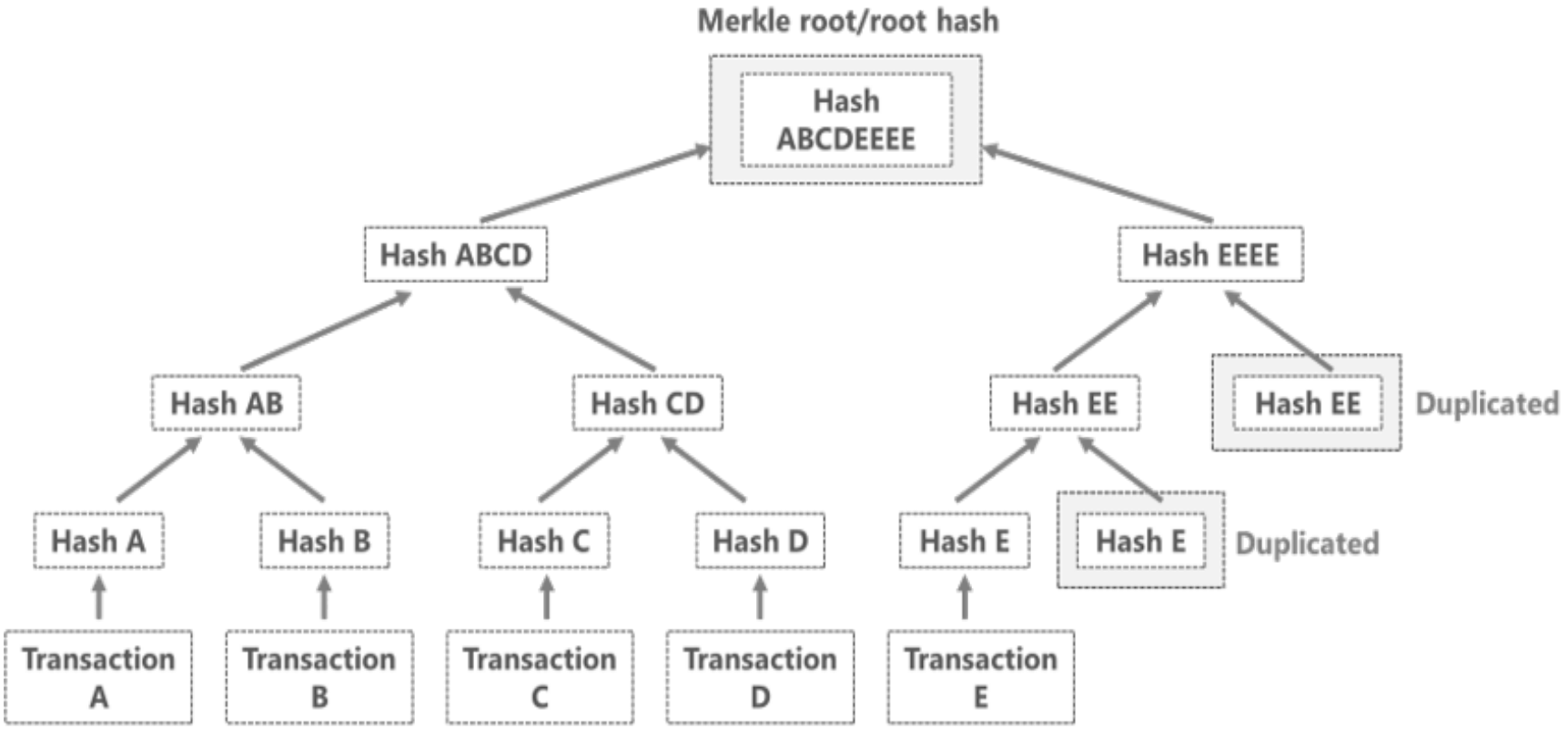
Note: Merkle Tree is a **USPTO patented Algorithm/ Data Structure** and hence, you cannot use it in production without permission or by paying royality to Ralph Merkle.


0x0123 [Merkle Root]
[Hash(0x01e3 + 0x23a2)]
0x01e3
[Hash(0x0ab4 + 0x1bd4)]
0x23a2
[Hash(0x2ad5 + 0x3ce9)]
0x0ab4
[Hash of D0]
0x1bd4
[Hash of D1]
0x2ad5
[Hash of D2]
0x3ce9
[Hash of D3]
Data Block D0
Data Block D1
Data Block D2
Data Block D3

The Merkle root is stored in the block header of the block



An Unbalanced Merkle tree with an odd number of transactions



The Merkle tree of transactions A, B, C, D, and E; Hash ABCDEEEE is the Merkle root/root hash

Hashing a Tree Structure

- https://www.baeldung.com/cs/hashing-tree
- https://decentralizedthoughts.github.io/2020-12-22-what-is-a-merkle-tree/

Hashed file organization
or
Direct file organization

File Organization

- The **File** is a collection of records. Using the primary key, we can access the records. The type and frequency of access can be determined by the type of file organization which was used for a given set of records.
- File organization is a logical relationship among various records. This method defines how file records are mapped onto disk blocks.
- File organization is used to describe the way in which the records are stored in terms of blocks, and the blocks are placed on the storage medium.
- The first approach to map the database to the file is to use the several files and store only one fixed length record in any given file. An alternative approach is to structure our files so that we can contain multiple lengths for records.
- Files of fixed length records are easier to implement than the files of variable length records.

File organization

- FILE ORGANIZATION It is the **methodology which is applied to structured computer files**.
- Files contain computer records which can be documents or information which is stored in a certain way for later retrieval. File organization refers primarily to the logical arrangement of data in a file system.
- Some types of File Organizations are :

❑ Sequential File Organization

❑ Index sequential

❑ Indexed [Heap File Organization , B+ Tree File Organization, ….]

❑ Direct or Random or Hash File Organization

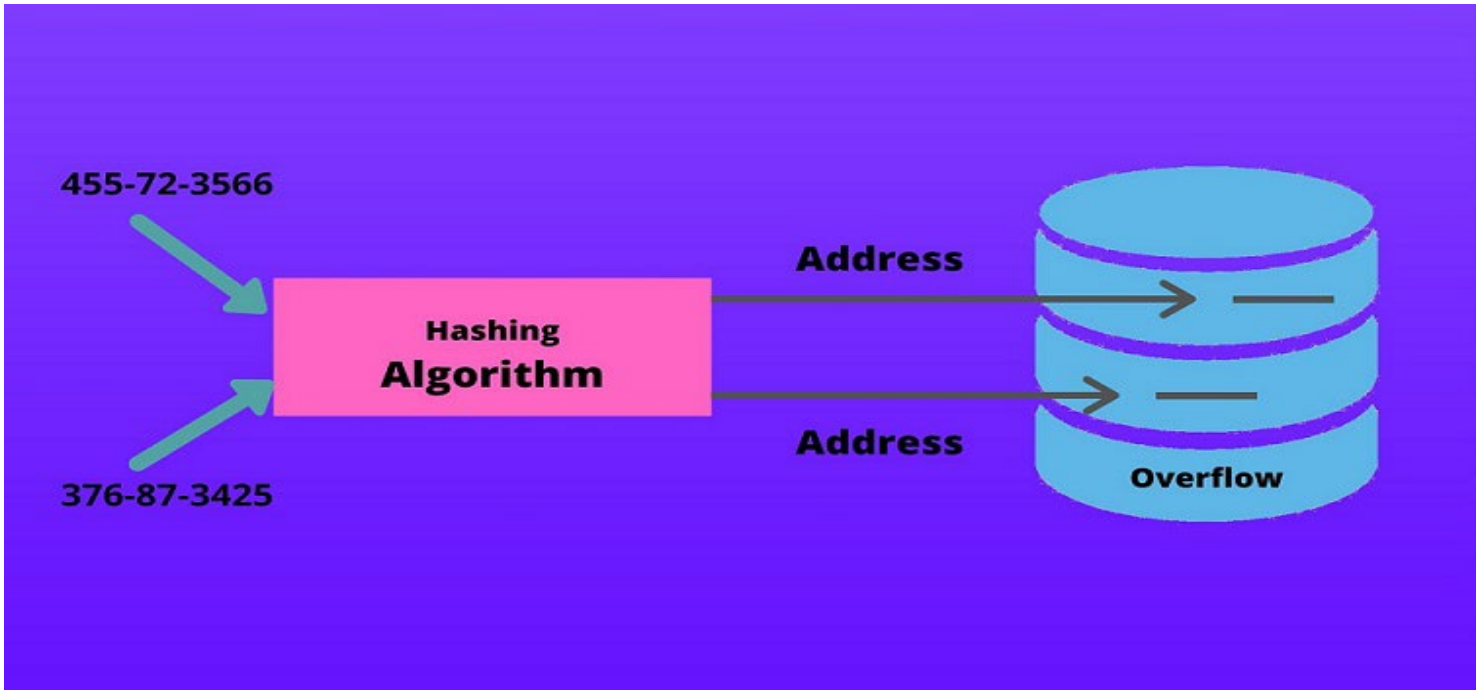
❑ Clustered File Organization

Hashed file organization

- **Hashed file organization** is also called **a direct file organization**.
- In direct file organization, the **key value is mapped directly to the storage location**.
- The usual method of direct mapping by performing some arithmetic manipulation of the key value. In this method, for storing the records a hash function is calculated, which provides the address of the block to store the record. This process is called **hashing**.

- Let us consider a hash function ***h*** that maps the key value ***k*** to the value ***h(k)***. The value ***h(k)*** is used as an address and for our application we require that this value be in some range. If our address are for the records lies between S_1 and S_2, that for all values of k it should generate values between S_1 and S_2.

Hashing in Direct File organization

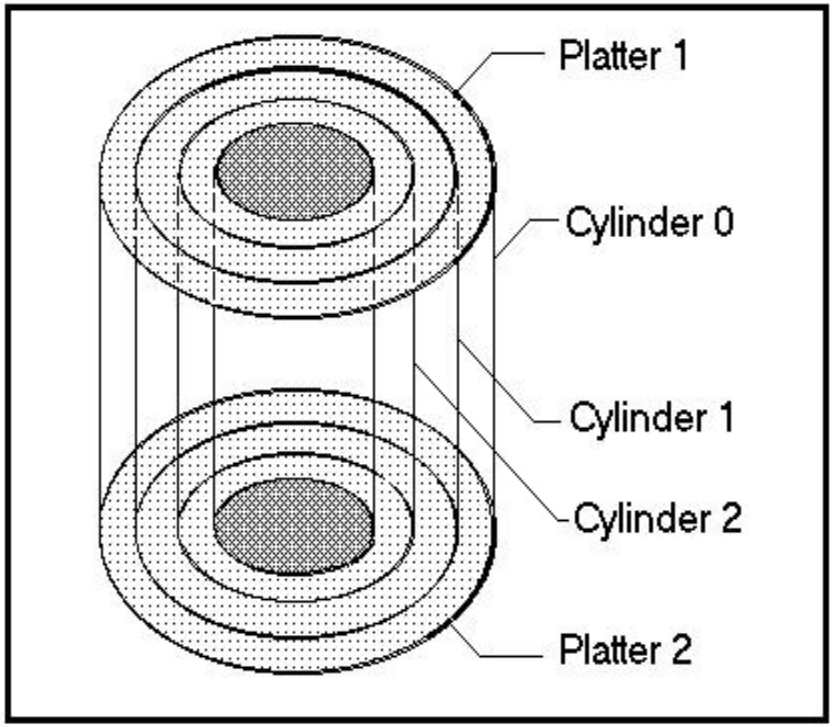


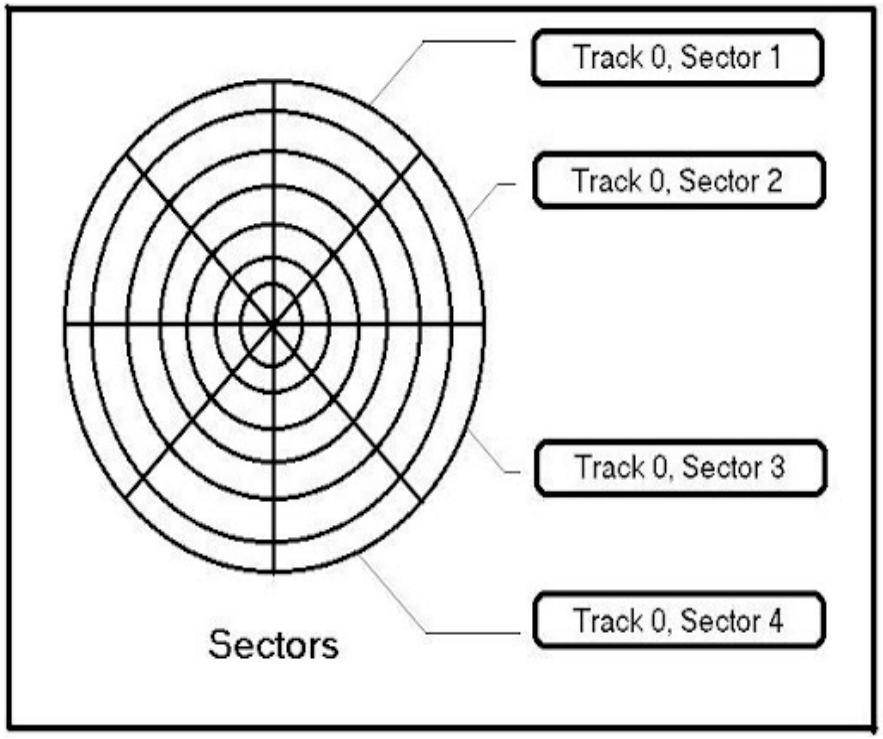
Direct-access storage device (DASD)

- A direct-access storage device (DASD) is another name for secondary storage devices that **store data in discrete locations with a unique address**, such as hard disk drives, optical drives and most magnetic storage devices.
- Modern DASDs are internal and external hard disk drives that connect directly to the host computer via an IDE, SATA, eSATA, USB or FireWire interface. Unlike network-attached storage (NAS), DASDs become inaccessible once the device they are connected to goes offline.
- The DASD *device block* (or *sector*) level is the level at which a processing unit can request low-level operations on a device block address basis. Typical low-level operations for DASD are read-sector, write-sector, read-track, write-track, and format-track.
- By using direct access storage, you can quickly retrieve information from random addresses as a stream of one or more blocks. Many DASDs perform best when the blocks to be retrieved are close in physical address to each other.

Direct-access storage device (DASD)

- A DASD consists of a set of flat, circular rotating platters. Each platter has one or two sides on which data is stored. Platters are read by a set of nonrotating, but positionable, read or read/write heads that move together as a unit.





Direct-access storage device (DASD)

sector

- An addressable subdivision of a track used to record one block of a program or data. On a DASD, this is a contiguous, fixed-size block.
- Every sector of every DASD is exactly 512 bytes.

track

- A circular path on the surface of a disk on which information is recorded and from which recorded information is read; a contiguous set of sectors.
- A track corresponds to the surface area of a single platter swept out by a single head while the head remains stationary.
- A DASD contains at least 17 sectors per track. Otherwise, the number of sectors per track is not defined architecturally and is device-dependent. A typical DASD track can contain 17, 35, or 75 sectors.
- A DASD may contain 1024 tracks. The number of tracks per DASD is not defined architecturally and is device-dependent.

Direct-access storage device (DASD)

head

- A head is a positionable entity that can read and write data from a given track located on one side of a platter.
- Usually a DASD has a small set of heads that move from track to track as a unit. There must be at least 43 heads on a DASD.
- Otherwise, the number is not defined architecturally and is device-dependent. A typical DASD has 8 heads.

cylinder

- The tracks of a DASD that can be accessed without repositioning the heads.
- If a DASD has *n* number of vertically aligned heads, a cylinder has *n* number of vertically aligned tracks.

- The Disks are comprised of tracks and further sectors, each of which has a **unique address**.
- The "Logical block addressing" ("LBA") is a common design that is used to mention the location of the block of data information stored on computer system storage devices like hard disks.

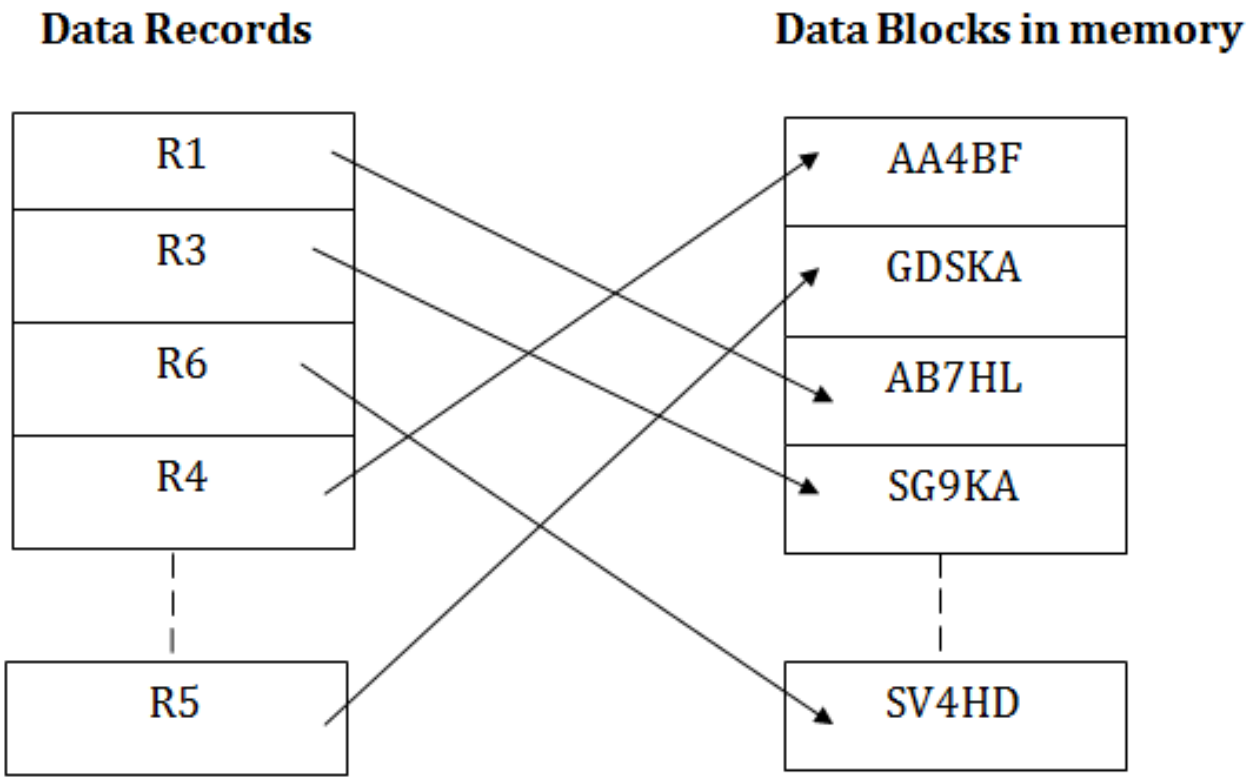
Data Block address in memory

CTR for cylinder-track-record or CHR

- The operating system uses a four byte relative track and record (TTR) for some access methods and for others an eight-byte extent-bin-cylinder-track-record block address, or **MBBCCHHR**, Channel programs address DASD using a six byte seek address (**BBCCHH**) and a five byte record identifier (**CCHHR**).
  - **M** represents the extent number within the allocation
  - **BB** representing the **B**in (from 2321 data cells),
  - **CC** representing the **C**ylinder,
  - **HH** representing the **H**ead (or track), and
  - **R** representing the **R**ecord (block) number.
- When the 2321 data cell was discontinued in January 1975, the addressing scheme and the device itself was referred to as CHR or CTR for cylinder-track-record, as the bin number was always 0.

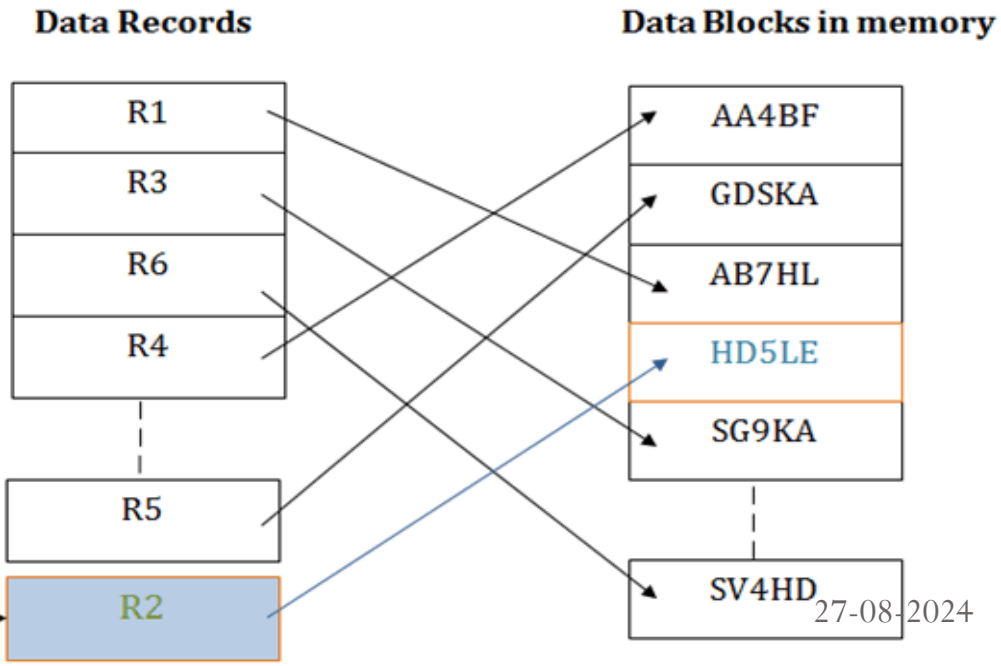
Hashed file organization

- Hash File Organization uses the computation of hash function on some fields of the records. The hash function's output determines the **location of disk block** where the records are to be placed.



Hashed file organization

- When a record has to be received using the hash key columns, then the address is generated, and the whole record is retrieved using that address. In the same way, when a new record has to be inserted, then the address is generated using the hash key and record is directly inserted. The same process is applied in the case of delete and update.
- In this method, there is no effort for searching and sorting the entire file. In this method, each record will be stored randomly in the memory(DASD).



Advantages & Disadvantages of Hash File Organization

Advantages of Hash File Organization

- Users can access the record at a fast speed because the address of the block is known by the hash function.
- It is the best method for online transactions like ticket booking, online banking, etc.

Disadvantages of Hash File Organization

- There is more chance of losing the data. For Example, In the employee table, when the hash field is on the Employee_Name, and there are two same names – 'Areena', then the same address is generated for both. In such a case, the record which is older will be overwritten by newer.
- This method of file organization is not correct in that situation when we are searching for a given range of data because each record in the database file will be stored at a random address. So, in this condition, searching for records is not efficient.
- If we search on those columns which are not hash columns, then the search will not find the correct data address because the search is done only on the hash columns.

Cuckoo hashing

Source code available in C++:

https://github.com/efficient/cuckoofilter

Cuckoo hashing [Open address]

- Cuckoo hashing is a form of open addressing in which each non-empty cell of a hash table contains a key or key–value pair.
- A hash function is used to determine the location for each key, and its presence in the table (or the value associated with it) can be found by examining that cell of the table.
- [Collision]The basic idea of cuckoo hashing is to resolve collisions by using two hash functions instead of only one. This provides two possible locations in the hash table for each key.
- In one of the commonly used variants of the algorithm, the hash table is split into two smaller tables of equal size, and each hash function provides an index into one of these two tables. It is also possible for both hash functions to provide indexes into a single table.

Introduction: Hashing

- Background : There are three basic operations that must be supported by a hash table (or a dictionary):

Lookup(key): return true if key is there in the table, else false

Insert(key): adds the item 'key' to the table if not already present

Delete(key): removes 'key' from the table

- Collisions are very likely even if we have a big table to store keys. Using the results from the birthday paradox: with only 23 persons, the probability that two people share the same birth date is 50%! There are 3 general strategies towards resolving hash collisions:
- **Closed addressing or Chaining**: store colliding elements in an auxiliary data structure like a linked list or a binary search tree.
- **Open addressing:** allow elements to over mow out of their target bucket and into other spaces.

Introduction: Cuckoo Hashing

- Although above solutions provide expected lookup cost as O(1), the expected worst-case cost of alook up in Open Addressing (with linear probing) is Ω(log n) and Θ(log n / log log n) in simple chaining (Source : Standford Lecture Notes).
- To close the gap of expected time and worst case expected time, two ideas are used:
- **Multiple-choice hashing**: Give each element multiple choices for positions where it can reside in the hash table
- **Relocation hashing**: Allow elements in the hash table to move after being placed
- **Cuckoo Hashing** : Cuckoo hashing applies the idea of multiple-choice and relocation together and guarantees O(1)worst case lookup time!

Cuckoo Hashing

- **Cuckoo Hashing** : Cuckoo hashing applies the idea of multiple-choice and relocation together and guarantees O(1)worst case lookup time!
- **Multiple-choice:** We give a key two choices h1(key) and h2(key) for residing.
- Otherwise, older key displaces another key. This continues until the procedure finds a vacant position, or enters a cycle. In case of cycle, new hash functions are chosen and the whole data structure is 'rehashed'. Multiple rehashes might be necessary before Cuckoo succeeds.
- Insertion is expected O(1) (amortized) with high probability, even considering the possibility rehashing, as long as the number of keys is kept below half of the capacity of the hash table, i.e. the load factor is below 50%.

Cuckoo Hashing

- Insertion is expected O(1) (amortized) with high probability, even considering the possibility rehashing, as long as the number of keys is kept below half of the capacity of the hash table, i.e. the load factor is below 50%.
- Deletion is O(1) worst-case as it requires inspection of just two locations in the hash table.

Cuckoo Hashing

- Example:

Input:

```
{20, 50, 53, 75, 100, 67, 105, 3, 36, 39}
```

Hash Functions:

```
h1(key) = key%11
h2(key) = (key/11)%11
```

https://www.geeksforgeeks.org/cuckoo-hashing/

Incomplete ……

C++ STL

- Hashing in C++ STL is a technique that maps a key with the associated hash value.

The syntax to create a template for hash class:

```
template <class key> struct hash;
```

Now, to create objects with the hash class, we can use the following syntax:

```
hash<class template> object-name
```

The hash class also has a single member function, operator(), that returns the object's hash value.

Example:

```
// Initializing character.
char ch = 'n';

// Creating object of the hash class.
hash<char> ch_hash;

// Using operator() to find hash value.
cout << ch_hash(ch);
```

Character Hashing

```cpp
// Program to demonstrate the character hashing.
#include <iostream>

// To avoid using std:: before each statement.
using namespace std;

// Function for character hashing.
void hashingChar() {

    char ch = 'n';

    // Instantiation of a hash object.
    hash<char> char_hash;

    // Using operator() to return hashed value.
    cout << "The hashed value of character 'n' is: " << char_hash(ch)
}

// Main function.
int main() {
    // Calling function for character hashing.
    hashingChar();
}
```

Building an Efficient Hash Table on the GPU

Bibliographic Details

DOI10.1016/b978-0-12-385963-1.00004-6

AUTHOR(S)Dan A. Alcantara; Vasily Volkov; Shubhabrata Sengupta; Michael Mitzenmacher; John D. Owens; Nina Amenta

PUBLISHER(S)Elsevier BV

Efficient Hash Table on the GPU

- A straightforward algorithm for parallel hash table construction on the graphical processing unit (GPU).
- It constructs the table in global memory and use atomic operations to detect and resolve collisions.
- Construction and retrieval performance are limited almost entirely by the time required for these uncoalesced memory accesses, which are linear in the total number of accesses; so the design goal is to minimize the average number of accesses per insertion or lookup.

Efficient Hash Table on the GPU

- The hash functions $h_i(k)$ should ideally be tailored for the application so that they distribute all of the items as evenly as possible throughout the table. However, it is difficult to guarantee this for every given input. To achieve fast construction times, we instead rely on randomly generated hash functions of the form:

$$\mathbf{h_i(k)=(a_i \cdot k+b_i) \bmod p \bmod tableSize}$$

- Here, $p = 334214459$ is a prime number, and *tableSize* is the number of slots in the table. Each h_i uses its own randomly generated constants a_i and b_i.
- These constants produce a set of weak hash functions, but they limit the number of slots under heavy contention and allow the table to be built successfully in most of our trials.
- It is found that using an XOR operation worked better than multiplication at distributing the items for some datasets; the question of what implementable hash functions to use for cuckoo hashing for provable performance guarantees.

Incomplete ……

Exercises

- Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m = 11$ using open addressing with the primary hash function $h'(k) = k \bmod m$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with $c_1 = 1$ and $c_2 = 3$, and using double hashing with $h_2(k) = 1 + (k \bmod (m - 1))$.
- Write pseudocode for HASH-DELETE as outlined in the text, and modify HASH-INSERT and HASH-SEARCH to incorporate the special value DELETED.
- Suppose that we use double hashing to resolve collisions; that is, we use the hash function $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$. Show that the probe sequence $h(k, 0), h(k, 1), \ldots, h(k, m - 1)$ is a permutation of the slot sequence $0, 1, \ldots, m - 1$ if and only if $h_2(k)$ is relatively prime to m.

Exercises

- Consider an open-address hash table with uniform hashing and a load factor $\alpha = 1/2$. What is the expected number of probes in an unsuccessful search? What is the expected number of probes in a successful search? Repeat these calculations for the load factors 3/4 and 7/8.
- Suppose that we insert n keys into a hash table of size m using open addressing and uniform hashing. Let $p(n, m)$ be the probability that no collisions occur. Show that $p(n, m) \leq e^{-n(n-1)/2m}$. Argue that when n exceeds $\sqrt{m}$, the probability of avoiding collisions goes rapidly to zero.
- The bound on the harmonic series can be improved to

References

- http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap12.htm
- https://www.2brightsparks.com/resources/articles/introduction-to-hashing-and-its-uses.html
- https://www.scaler.com/topics/hashing-in-cpp-stl/


Thank you for your attention

Exercises

Exercises: Direct address Hashing

1) Consider a dynamic set S that is represented by a direct-address table T of length m. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?
2) A ***bit vector*** is simply an array of bits (0's and 1's). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements with no satellite data. Dictionary operations should run in $O(1)$ time.
3) Suggest how to implement a direct-address table in which the keys of stored elements do not need to be distinct and the elements can have satellite data. All three dictionary operations (INSERT, DELETE, and SEARCH) should run in $O(1)$ time. (Don't forget that DELETE takes as an argument a pointer to an object to be deleted, not a key.)
4) We wish to implement a dictionary by using direct addressing on a *huge* array. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct-address dictionary on a huge array. Each stored object should use $O(1)$ space; the operations SEARCH, INSERT, and DELETE should take $O(1)$ time each; and the initialization of the data structure should take $O(1)$ time. (*Hint:* Use an additional stack, whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.)

Exercises :Hash tables

1) Suppose we use a random hash function h to hash n distinct keys into an array T of length m. What is the expected number of collisions? More precisely, what is the expected cardinality of $\{(x,y): h(x) = h(y)\}$?
2) Demonstrate the insertion of the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \bmod 9$.
3) Argue that the expected time for a successful search with chaining is the same whether new elements are inserted at the front or at the end of a list. (*Hint:* Show that the expected successful search time is the same for *any* two orderings of any list.)
4) Professor Marley hypothesizes that substantial performance gains can be obtained if we modify the chaining scheme so that each list is kept in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?
5) Suggest how storage for elements can be allocated and deallocated within the hash table itself by linking all unused slots into a free list. Assume that one slot can store a flag and either one element plus a pointer or two pointers. All dictionary and free-list operations should run in $O(1)$ expected time. Does the free list need to be doubly linked, or does a singly linked free list suffice?
6) Show that if $|U| > nm$, there is a subset of U of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $O(n)$.

Exercises :Hash Function

1) Suppose we wish to search a linked list of length n, where each element contains a key k along with a hash value $h(k)$. Each key is a long character string. How might we take advantage of the hash values when searching the list for an element with a given key?

2) Suppose a string of r characters is hashed into m slots by treating it as a radix-128 number and then using the division method. The number m is easily represented as a 32-bit computer word, but the string of r characters, treated as a radix-128 number, takes many words. How can we apply the division method to compute the hash value of the character string without using more than a constant number of words of storage outside the string itself?

3) Consider a version of the division method in which $h(k) = k \bmod m$, where $m = 2^p - 1$ *and* k is a character string interpreted in radix 2^p. Show that if string x can be derived from string y by permuting its characters, then x and y hash to the same value. Give an example of an application in which this property would be undesirable in a hash function.

Exercises :Hash Function

1) Consider a hash table of size $m = 1000$ and the hash function $h(k) = \lfloor m\ (k\ A \bmod 1) \rfloor$ for $A = (\sqrt{5}-1)/2$. Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped.

2) Show that if we restrict each component a_i of a in equation (12.3) to be nonzero, then the set as defined in equation (12.4) is not universal. (*Hint:* Consider the keys $x = 0$ and $y = 1$.)

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m\ . \qquad (12.3)$$

$$\mathcal{H} = \bigcup_{a} \{h_a\} \qquad (12.4)$$

Suggested Homework

- Draw the 11 entry hashtable for hashing the keys 12, 44, 13, 88, 23, 94, 11, 39, 20 using the function (2i+5) mod 11, closed hashing, linear probing
- Pseudo-code for listing all identifiers in a hashtable in lexicographic order, using open hashing, the hash function h(x) = first character of x. What is the running time?

Suggested Homework

- Draw the 11 entry hashtable for hashing the keys 12, 44, 13, 88, 23, 94, 11, 39, 20 using the function (2i+5) mod 11, closed hashing, linear probing
- Pseudo-code for listing all identifiers in a hashtable in lexicographic order, using open hashing, the hash function h(x) = first character of x. What is the running time?
- Draw a hash table with open addressing and a size of 9. Use the hash function "k%9". Insert the keys: 5, 29, 20, 0, 27 and 18 into your table (in that order).
- Suppose you are building an open address hash table with **double** hashing. The hash table capacity is n, so that the valid hash table indexes range from 0 to n. Fill in the blanks:
  - In order to ensure that every array position is examined, the value returned by the second hash function must be _____ with respect to n.
  - One way to ensure this good behavior is to make n be _____, and have the return value of the second hash function range from _____ to _____ (including the end points).

Suggested Homework

- Suppose that an open-address hash table has a capacity of 811 and it contains 81 elements. What is the table's load factor? (An approximation is fine.)
- I plan to put 1000 items in a hash table, and I want the average number of accesses in a successful search to be about 2.0.

  A. About how big should the array be if I use open addressing with linear probing? NOTE: For a load factor of **A**, the average number of accesses is generally ½(1+1/(1-**A**)).

  B. About how big should the array be if I use chained hashing? NOTE: For a load factor of **A**, the average number of accesses is generally (1+**A**/2).

Suggested Homework

- **Weiss 5.1** Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function h(x) = (x mod 10), show the resulting: (a) separate chaining hash table. (b) hash table using linear probing. 1 (c) hash table using quadratic probing. (d) hash table with second hash function h2(x) = 7 − (x mod 7).
- **Weiss 5.2** Show the result of rehashing the hash tables in the previous exercise. Rehash using a new table size of 19, and a new hash function h(x) = (x mod 19).
- Describe the advantages and disadvantages of Binary Search Trees versus Binary Heaps versus Hash Tables. What operations are efficient in some but not others? What are the tradeoffs between the three data structures? Give three example applications, one for each of these data structures, where it makes the most sense to use that structure and not the other two. Explain your choices.

Suggested Homework

- Which of the hash table collision-handling schemes could tolerate a load factor above 1 and which could not?
- What is the worst-case running time for inserting n key-value entries into an initially empty map M that is implemented with a list?
- What is the worst-case asymptotic running time for performing n (correct) erase() operations on a map, implemented with an ordered search table, that initially contains 2n entries?
- Describe how to use a skip-list map to implement the dictionary ADT, allowing the user to insert different entries with equal keys.
- Describe how an ordered list implemented as a doubly linked list could be used to implement the map ADT.

Suggested Homework

- What would be a good hash code for a vehicle identification that is a string of numbers and letters of the form "9X9XX99X9XX999999," where a "9" represents a digit and an "X" represents a letter?
- Draw the 11-entry hash table that results from using the hash function, h(i) = (3i+5) mod 11, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by chaining.
- What is the result of the previous exercise, assuming collisions are handled by linear probing?
- Give a pseudo-code description of an insertion into a hash table that uses quadratic probing to resolve collisions, assuming we also use the trick of replacing deleted items with a special "deactivated item" object.
- Describe a set of operations for an ordered dictionary ADT that would correspond to the functions of the ordered map ADT. Be sure to define the meaning of the functions so that they can deal with the possibility of different entries with equal keys.

Suggested Homework

- Explain why a hash table is not suited to implement the ordered dictionary ADT.
- What is the worst-case running time for inserting n items into an initially empty hash table, where collisions are resolved by chaining? What is the best case?
- What is the expected running time of the functions for maintaining a maxima set if we insert n pairs such that each pair has lower cost and performance than the one before it? What is contained in the ordered dictionary at the end of this series of operations? What if each pair had a lower cost and higher performance than the one before it?
- Describe how to perform a removal from a hash table that uses linear probing to resolve collisions where we do not use a special marker to represent deleted elements. That is, we must rearrange the contents so that it appears that the removed entry was never inserted in the first place.

Suggested Homework

- Suppose that each row of an n×n array A consists of 1's and 0's such that, in any row of A, all the 1's come before any 0's in that row. Assuming A is already in memory, describe a method running in O(nlogn) time (not $O(n^2)$ time!) for counting the number of 1's in A.
- Describe an efficient ordered dictionary structure for storing n elements that have an associated set of k < n keys that come from a total order. That is, the set of keys is smaller than the number of elements. Your structure should perform all the ordered dictionary operations in O(log k+ s) expected time, where s is the number of elements returned.

Project

Write a spell-checker class that stores a set of words, W, in a hash table and implements a function, spellCheck(s), which performs a ***spell check*** on the string s with respect to the set of words, W. If s is in W, then the call to spellCheck(s) returns an iterable collection that contains only s, since it is assumed to be spelled correctly in this case. Otherwise, if s is not in W, then the call to spellCheck(s) returns a list of every word in W that could be a correct spelling of s. Your program should be able to handle all the common ways that s might be a misspelling of a word in W, including swapping adjacent characters in a word, inserting a single character inbetween two adjacent characters in a word, deleting a single character from a word, and replacing a character in a word with another character. For an extra challenge, consider phonetic substitutions as well.

Programing assignment

- Perform a comparative analysis that studies the collision rates for various hash codes for character strings, such as various polynomial hash codes for different values of the parameter α. Use a hash table to determine collisions, but only count collisions where different strings map to the same hash code (not if they map to the same location in this hash table). Test these hash codes on text files found on the Internet.
- Perform a comparative analysis as in the previous exercise but for 10-digit telephone numbers instead of character strings.
- Design a C++ class that implements the skip-list data structure. Use this class to create implementations of both the map and dictionary ADTs, including location-aware functions for the dictionary.

Programing assignment-I

- Perform a comparative analysis that studies the collision rates for various hash codes for character strings, such as various polynomial hash codes for different values of the parameter α. Use a hash table to determine collisions, but only count collisions where different strings map to the same hash code (not if they map to the same location in this hash table). Test these hash codes on text files found on the Internet.
- Perform a comparative analysis as in the previous exercise but for 10-digit telephone numbers instead of character strings.
- Design a **C++ class** that implements the **skip-list data structure**. Use this class to create implementations of both the map and dictionary ADTs, including location-aware functions for the dictionary.

Programing assignment-II

- Implement a probing hash table and insert 10,000 randomly generated integers into the table and count the average number of probes used.
- This average number of probes used is the average cost of a successful search. Repeat this test 10 times and calculate minimum, maximum and average values of average cost.
- Run the test for both linear and quadratic probing and do it for final load factors $\lambda = 0.1, 0.3, 0.5, 0.7, 0.9$. Always choose the capacity of the table so that no rehashing is needed. For instance, for final load factor $\lambda = 0.5$, in order to insert 10,000 integers into the hash table, the size of the table should be approximately 20,000 (i.e., $10000/\lambda = 10000/0.5 = 20000$).
- You must make some adjustments to make table size a prime number. For instance, 20,011 is the prime number that is slightly larger than 20,000.

Programing assignment-II

- You can refer to the following link for the list of prime numbers: http://compoasso.free.fr/primelistweb/page/prime/liste_online_en.php At the end of your simulations, please fill in the following table.

Load Factor (λ)	Average Cost for a Successful Search					
	Linear Probing			Quadratic Probing		
	minimum	Maximum	average	minimum	Maximum	Average
0.1						
0.3						
0.5						
0.7						
0.9						

Programing assignment-III

- **Problem Statement:** Hash tables are very useful in situations where an individual wants to quickly find their data by the "value" or "search key". You could think of them as similar to an array, except the client program uses a "key" instead of an "index" to get to the data. The key is then mapped through the hash function which turns it into an index!
- You have decided to write a table abstract data type, using a hash table/function with chaining (for collision resolution), to support searching for interesting web sites. Your search key is the topic that someone is interested in and the list of appropriate web sites is displayed (just the link) that pertain to that particular subject matter. A web site may have many different subjects that it can be searched upon.

Programing assignment-III

Abstract Data Type:

- Write a C++ program that implements and uses a **table abstract data type** to store, search, and remove web site links and their subject lists.
- What does retrieve need to do? It needs to supply back to the calling routine information about all of the web sites that contain the search key. Retrieve, since it is an ADT operation, should not correspond with the user (i.e., it should not prompt, echo, input, or output data).
- Evaluate the performance of storing and retrieving items from this table. Monitor the number of collisions that occur for a given set of data that you select. Make sure your hash table is small enough that collisions actually do occur, even without vast quantities of data so that you can get an understanding of how chaining works. Try different table sizes, and evaluate the performance (i.e., the length of the chains!). Although, remember to make the hash table size a prime number.
- Implement the abstract data type using a separate header file (.h) and implementation file (.cpp). Only include header files when absolutely necessary. Never include .cpp files!

Syntax Requirements:

1) Use iostream.h for all I/O; do not use stdio.h

2) Implement the table ADT as a class; provide a constructor and a destructor.

3) Make sure to protect the data members so that the main() program does not have direct access.

4) Your main() should allow the user to interactively insert, delete, or retrieve items from the table.

Stack as an abstract data type

- A stack of elements of type T is a finite sequence of elements together with the operations

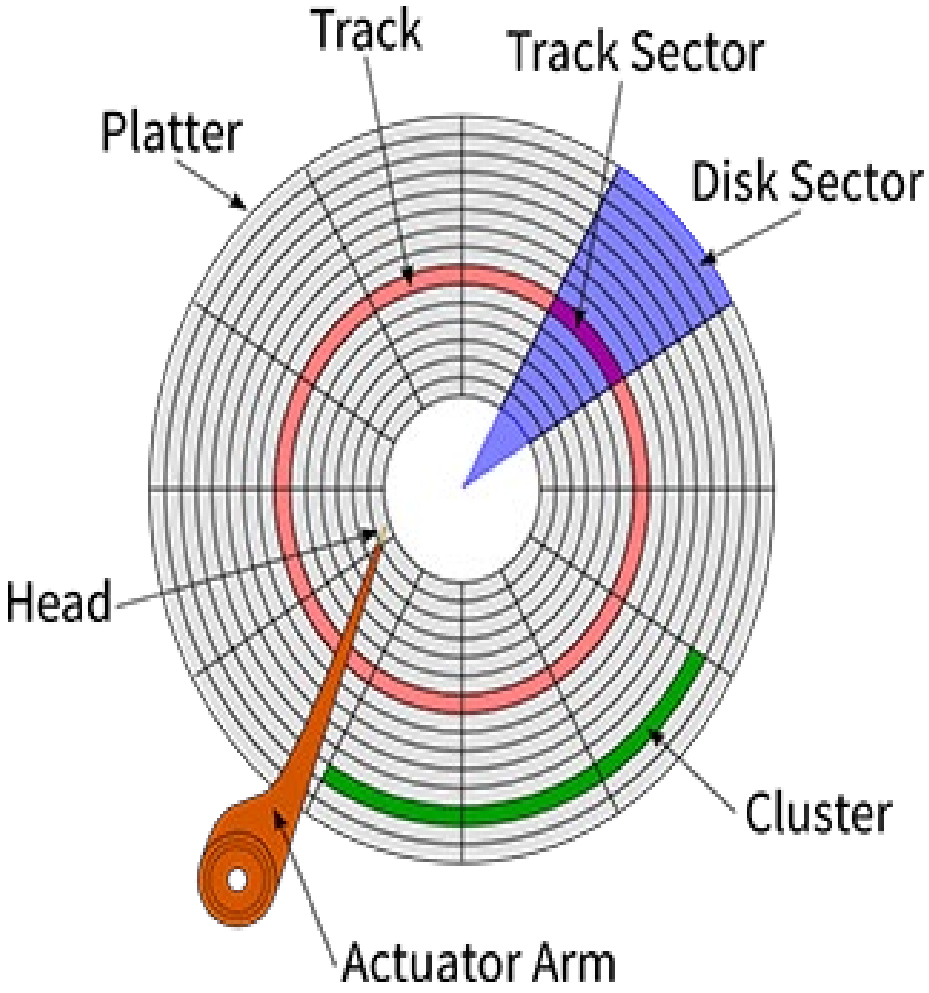
1. **CreateEmptyStack(S)**: create or make stack **S** be an empty stack
2. **Push(S,x):** Insert **x** at one end of the stack, called its **top**
3. **Top(S):** If stack **S** is not empty; then retrieve the element at its **top**
4. **Pop(S):** If stack **S** is not empty; then delete the element at its **top**
5. **IsFull(S):** Determine if **S** is full or not. Return **true** if **S** is full stack; return **false** otherwise
6. **IsEmpty(S)**: Determine if **S** is empty or not. Return **true** if **S** is an empty stack; return **false** otherwise.

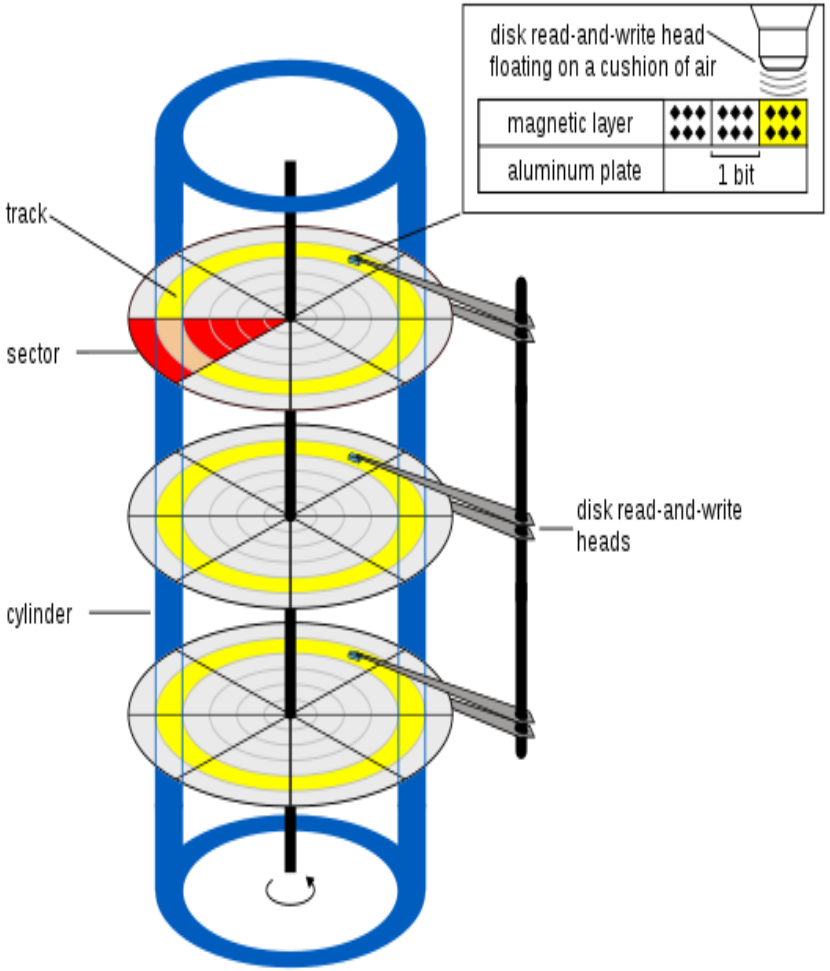
13

DASD: direct access storage device

Appendix-I

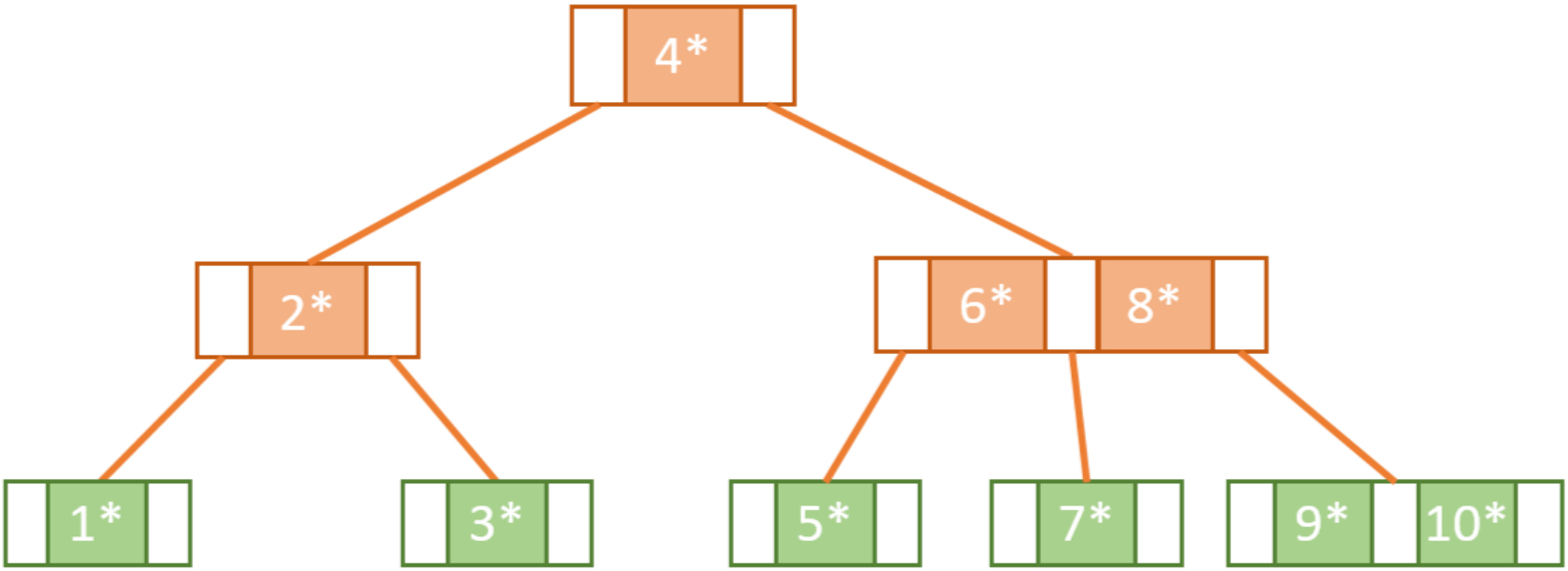
DASD: direct access storage device





B-tree of order 3

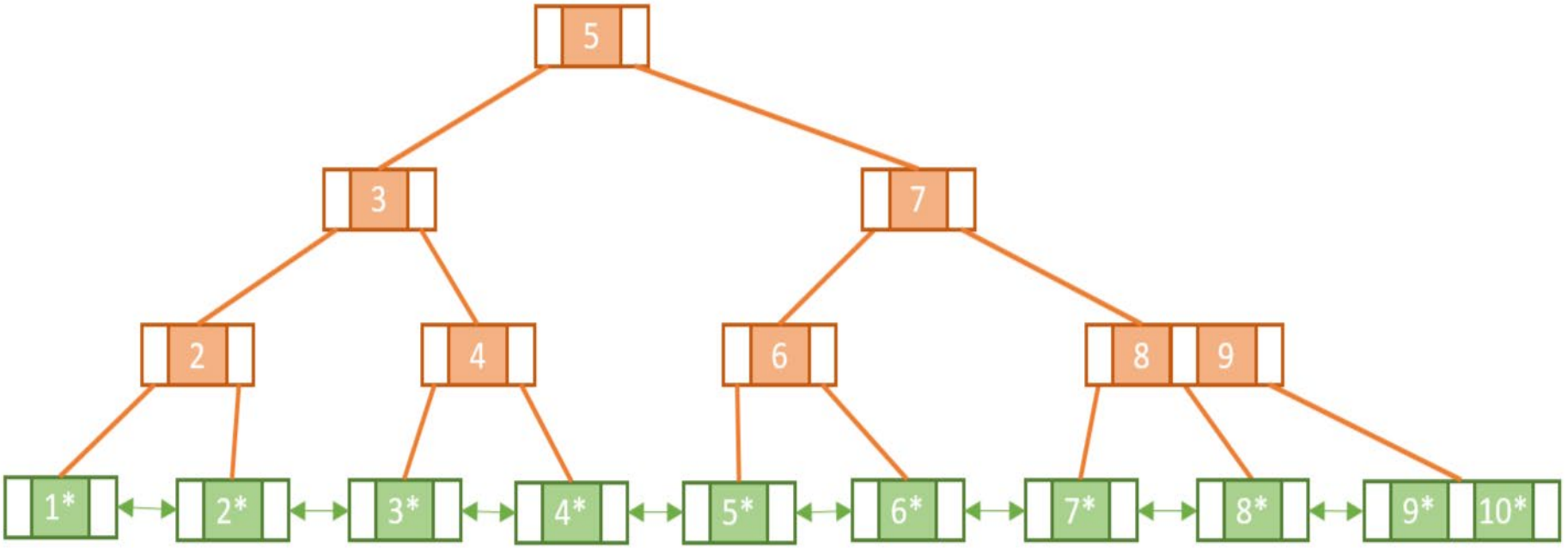
- All the leaves are on the same level, and the root and internal nodes have a minimum of two children fulfilling all the conditions of a B-tree



B+ tree

What distinguishes the B+tree from the B-tree are two main aspects:

- all leaf nodes are linked together in a doubly-linked list
- satellite data is stored on the leaf nodes only. Internal nodes only hold keys and act as routers to the correct leaf node



B tree of order 5