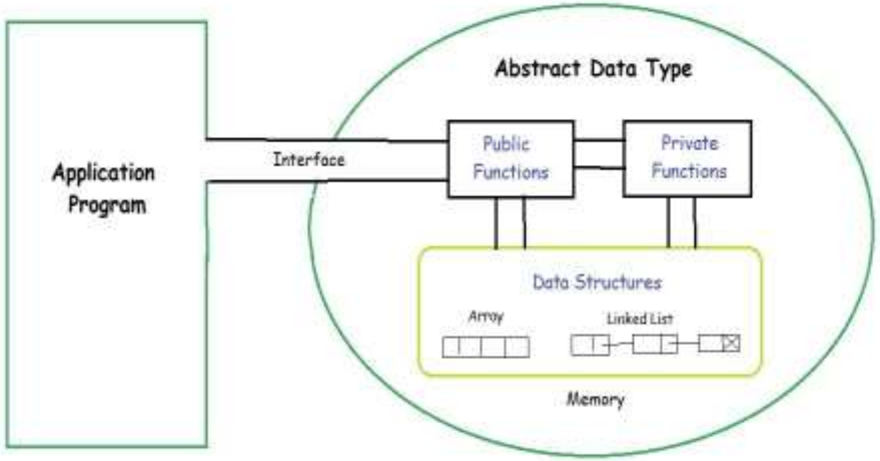
Hash Tables

COURSE CODE: 21AI33

Abstract Data Types (ADTs)

- Data and Operations are binded
- The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented.
- It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- **It is called "abstract" because it gives an implementation-independent view.**
- **Examples: List ADT, Stack ADT, Queue ADT, Vectors, Maps, Sets, etc.**



Abstract Data Types (ADTs)

ADT has three components:

1. **A name**
2. **A set of required operations (with contracts)**
3. **A set of expected properties**

ADT Name: Set

ADT operations:

addElt : Set element -> Set

remElt : Set element -> Set

size : Set -> integer

hasElt : Set element -> boolean

ADT properties (vague):

1. The set has no duplicates

2. The items within the set are unordered

Set ADT (Abstract Data Type)

- **A set is *an abstract data type that can store unique values, without any particular order***
- **Maintains a set *S* under the following three operations:**

1. Insert(*x*): Add the key *x* to the set.

2. Delete(*x*): Remove the key *x* from the set.

3. Search(*x*): Determine if *x* is contained in the set, and if so, return a pointer to *x*.

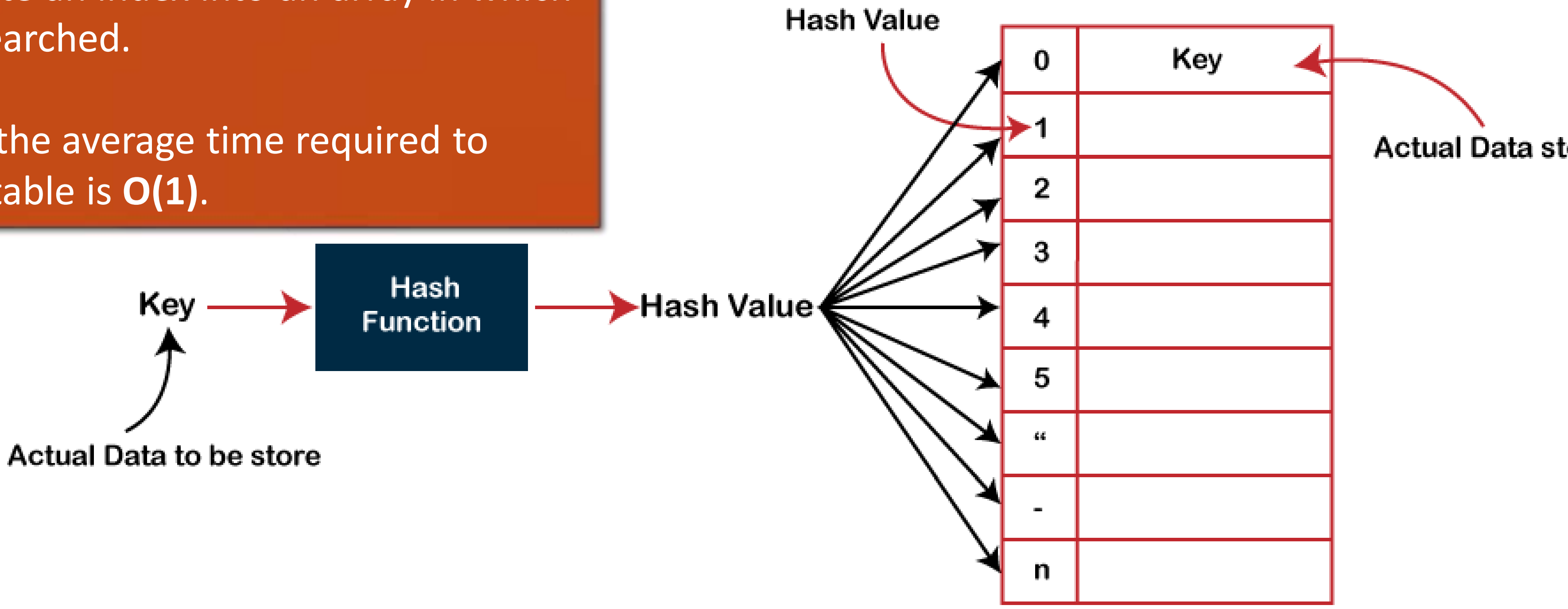
- **Hash Tables are used to implement set ADT**

Hash Tables Basics

A hash table is a data structure that is used to store keys/value pairs.

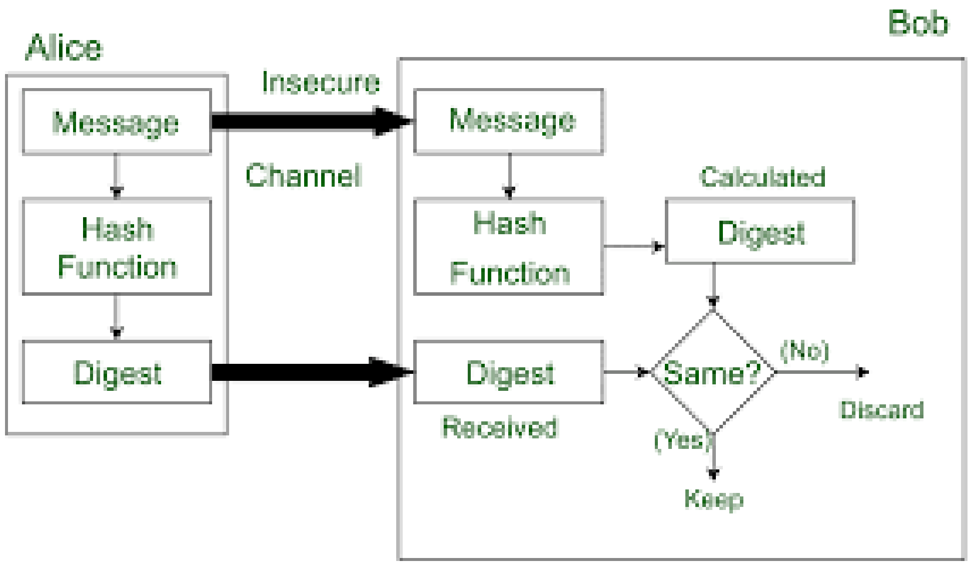
It uses a hash function to compute an index into an array in which an element will be inserted or searched.

Under reasonable assumptions, the average time required to search for an element in a hash table is **O(1)**.

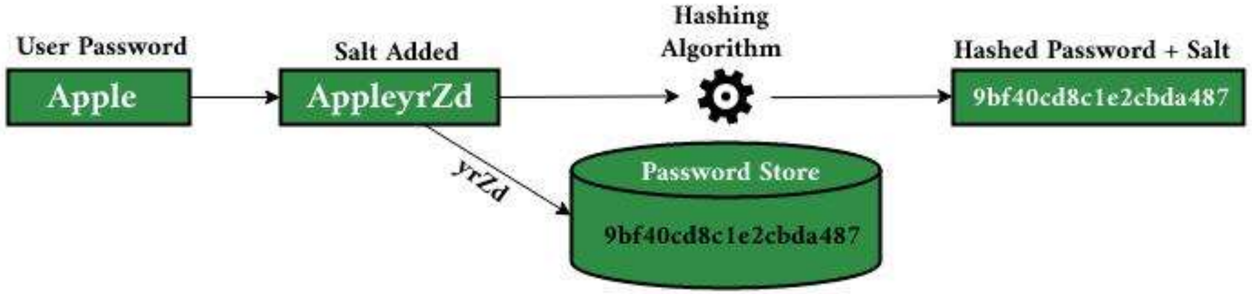


Hash Tables Applications

- Creating Message Digests in Cryptography



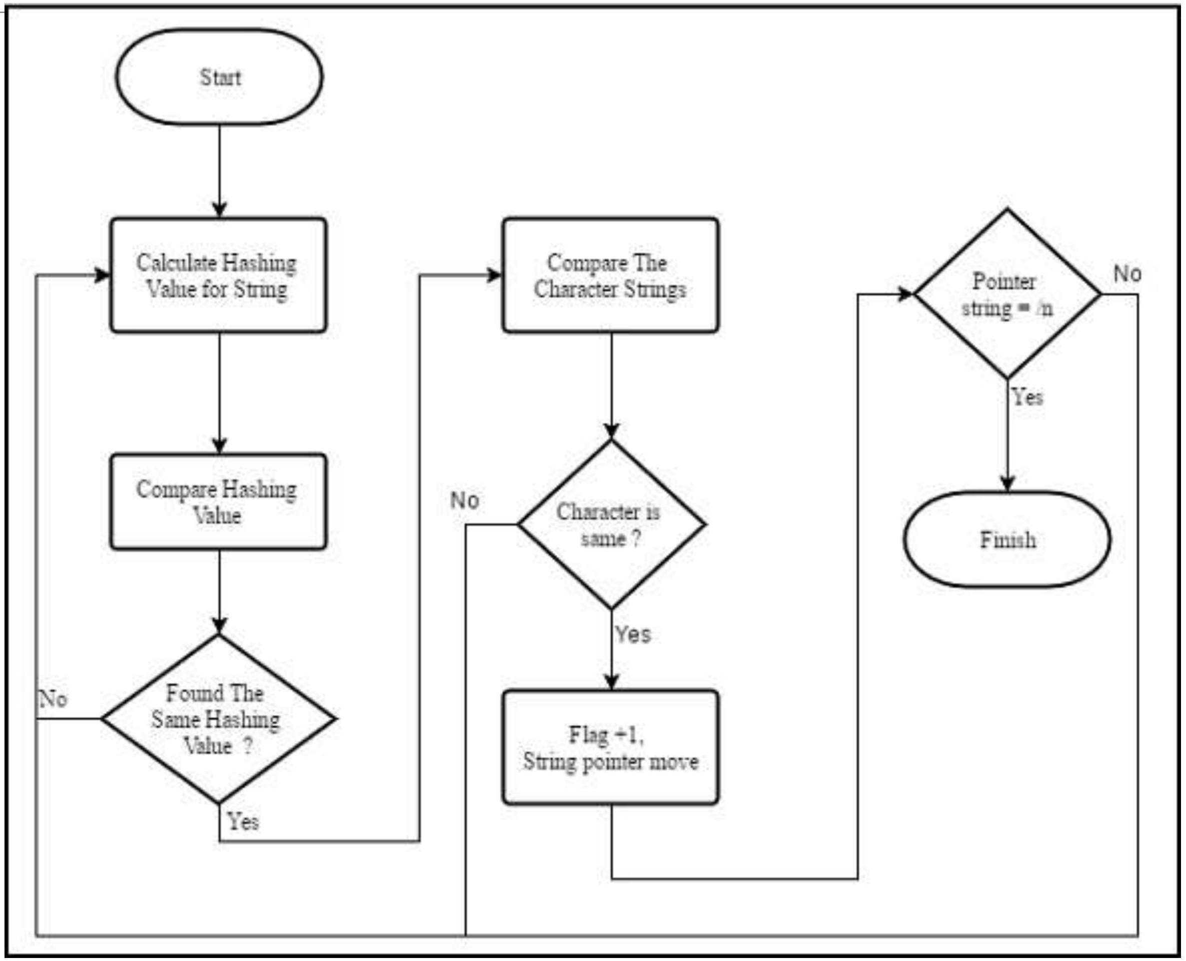
- Password Verification



Hash Tables Applications

- Rabin Karp Algorithm (Plagiarism Checking)

Location	Comparison	Hashing Values	Check Needed?	Result
0	**a b a** b a a c **b a b**	H("aba") = 292 H("bab") = 293	No	
1	a **b a b** a a c **b a b**	H("bab") = 293 H("bab") = 293	Yes	**Match**
2	a b **a b a** a c **b a b**	H("aba") = 292 H("bab") = 293	No	
3	a b a **b a a** c **b a b**	H("baa") = 292 H("bab") = 293	No	
4	a b a b **a a c** **b a b**	H("aac") = 293 H("bab") = 293	Yes	Mismatch



Integer Universe assumption

- All elements stored in the hash table come from the universe $U = \{0, . . . , u-1\}$.

- The goal is to design a hash function

$h : U \rightarrow \{0, . . ., m - 1\}$ so that for each $i \in \{0, . . ., m - 1\}$, the number of elements $x \in S$ such that $h(x) = i$ is as small as possible.

- Ideally, the hash function h would be such that each element of S is mapped to a unique value in $\{0, . . ., m-1\}$.

Random Probing assumption

Each element x that is inserted into a hash table is a black box that comes with an infinite random *probe sequence* $x_0, x_1, x_2, \ldots$

where each of the x_i is independently and uniformly distributed in $\{0, \ldots, m - 1\}$.

Hash Tables for Integer Keys

A *hash function h* is a function whose domain is *U* and whose range is the set $\{0, . . ., m - 1\}$, $m \leq u$.; key values *x* come from the universe $U = \{0, . . . , u - 1\}$.

A hash function *h* is said to be a *perfect hash function* for a set $S \subseteq U$ if, for every $x \in S$, $h(x)$ is unique.

A perfect hash function *h* for *S* is *minimal* if $m = |S|$, i.e., *h* is a bijection between *S* and $\{0, . . ., m - 1\}$.

A minimal perfect hash function for *S* is desirable since it allows us to store all the elements of S in a single array of length n.

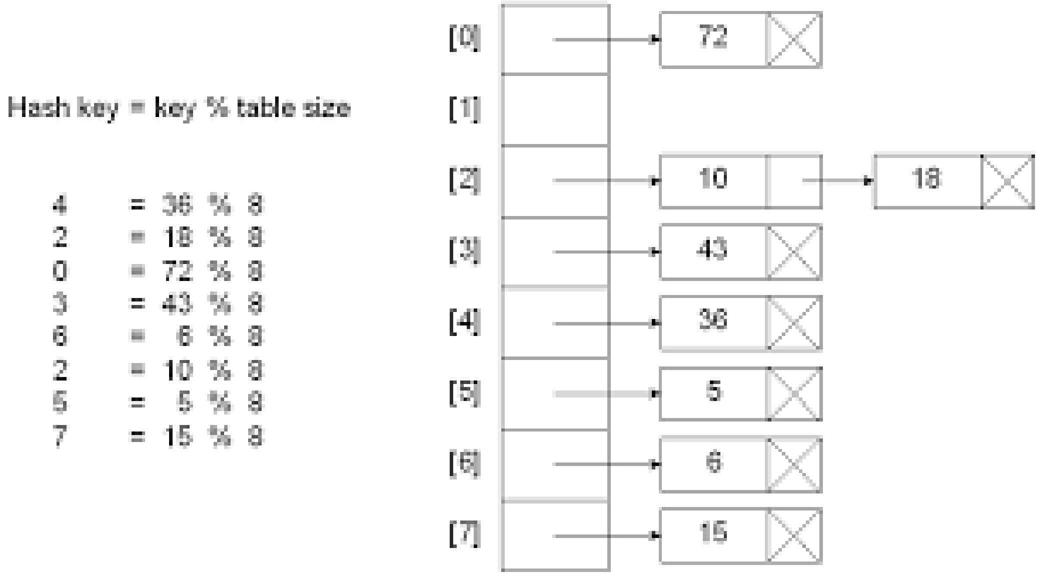
Hashing by Division

In *hashing by division*, we use the hash function

$\boldsymbol{h(x) = x \bmod m}$ **.**

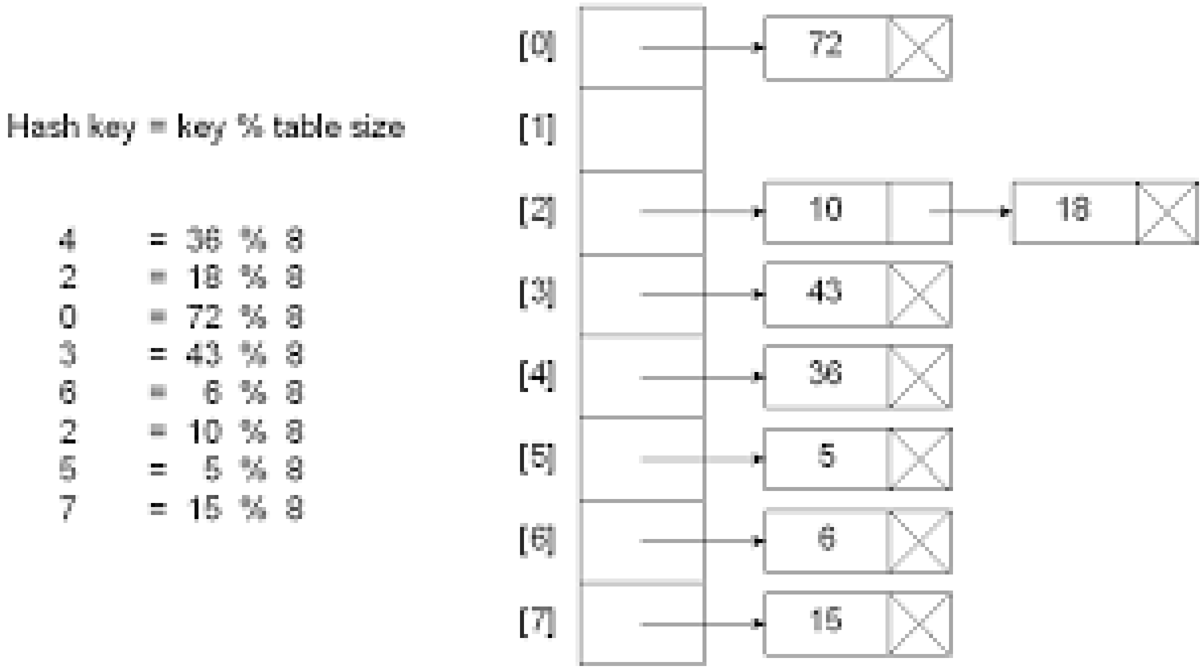
To use this hash function in a data structure, we maintain an array $A[0], \ldots, A[m-1]$ where each element of this array is a pointer to the head of a linked list.

The linked list L_i pointed to by the array element $A[i]$ contains all the elements x such that $h(x) = i$.



Hashing by Division

If the elements of *S* are uniformly and independently distributed in *U* and *u* is a multiple of *m* then the expected size of any list *Li* is only *n/m*.



Hashing by Division

- Inserting an element *x* takes *O*(1) time
- Searching for and/or deleting an element *x* is not so easy. We have to compute *i* = *h*(*x*) and then traverse the list *Li* until we either find *x* or reach the end of the list.
- If Set *S* consists of the elements 0,*m*, 2*m*, 3*m*, . . ., *nm* then all elements are stored in the list *L*0 and searches and deletions take linear time.
- If the elements of *S* are uniformly and independently distributed in *U* and *u* is a multiple of *m* then the expected size of any list *Li* is only *n*/*m*.
- Choice of m plays important role; $m \neq 2^p$

Hashing by Multiplication

The hash function

$h(x)$ = floor(m*x*A) mod m

Here A is a real-valued constant.

The advantage of the multiplication method is that **the value of *m* is not critical**.

We can take m to be a power of 2, which makes it convenient for use on binary computers.

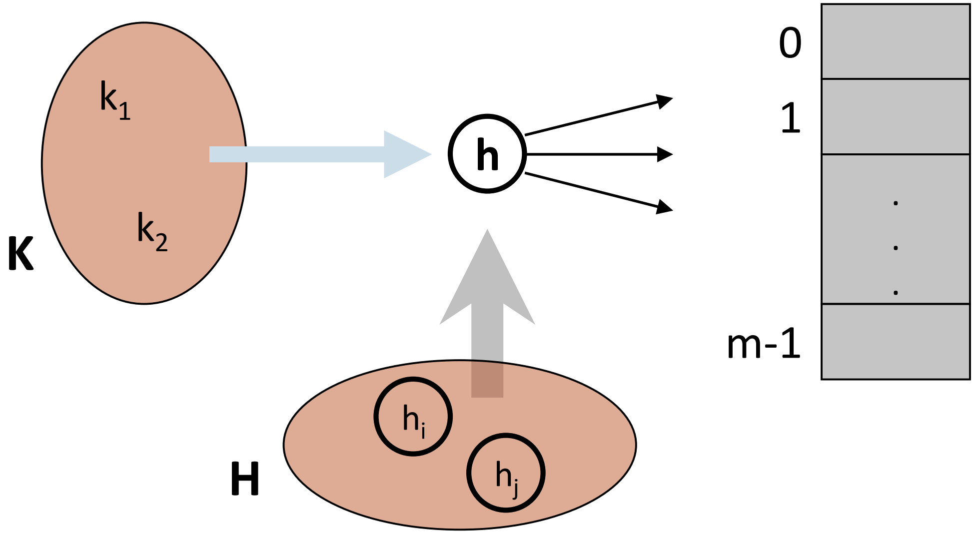
Use Golden Ratio for A (Knuth) (A=0.618…)

Choose A and B so that $(A+B)/A=A/B$.

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.618.$$

Universal Hashing

Suppose we have a set K of possible keys, and a *finite* set H of hash functions that map keys to entries in a hash table of size m.



Universal Hashing

- If the table size **m** is much smaller than the universe size **u** then for any hash function there is some large (of size at least **u/m**) subset of U that has the same hash value.
- To get around this difficulty we need a collection of hash functions from which we can choose one that works well for S.
- Let H be a collection of hash functions, i.e., functions from U onto $\{0, . . ., m - 1\}$.
- We say that H is universal if, for each $x, y \in U$ the number of $h \in H$ such that $h(x) = h(y)$ is at most $|H|/m$.
- Consider some value $x \in U$. The probability that any key $y \in S$ has the same hash value as x is only $1/m$.

Universal Hashing

- The expected number of keys in S, not equal to x, that have the same hash value as x is only

$$n_{h(x)} = \begin{cases} (n-1)/m & \text{if } x \in S \\ n/m & \text{if } x \notin S \end{cases}$$

-

Idea of universal hashing:

Choose hash function h randomly

H finite set of hash functions

$$h \in H : U \to \{0,\dots m-1\}$$

Definition: H is universal, if for arbitrary $x,y \in U$:

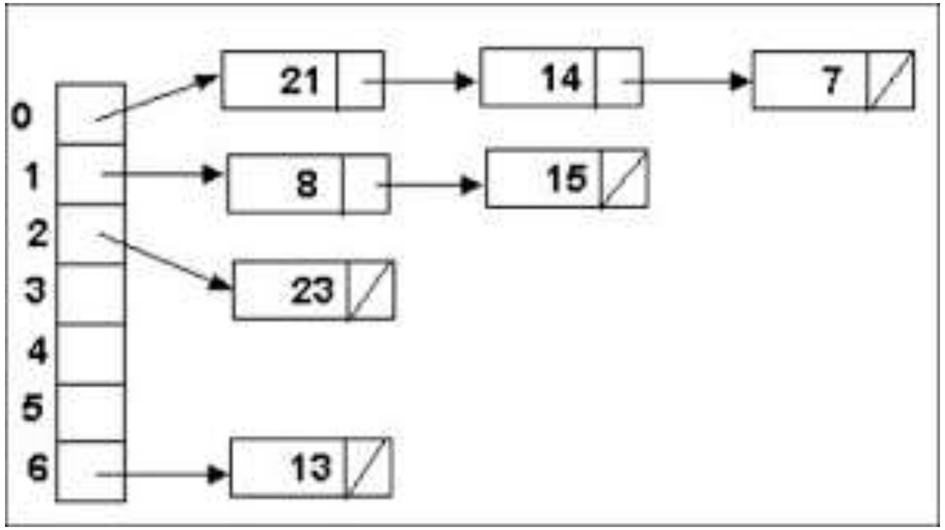
$$\frac{\{h \in H \mid h(x) = h(y)\}}{|H|} \leq \frac{1}{m}$$

Hence: if x, $y \in U$, H universal, $h \in H$ picked randomly

$$\Pr_H(h(x) = h(y)) \leq \frac{1}{m}$$

Random Probing-Hashing with Chaining

- Allows more than one element to be stored at each position (it is a collision resolution technique)
- Each entry in the array A is a pointer to the head of a linked list.
- To insert the value x, we simply append it to the list A[x0].
- To search for the element x, we perform a linear search in the list A[x0].
- To delete the element x, we search for x in the list A[x0] and delete it out.



Random Probing-Hashing with Chaining

- Insertions take $O(1)$ time, even in the worst case.
- For searching and deletion, the running time is proportional to a constant plus the length of the list stored at $A[x0]$.
- Each of the at most n elements not equal to x is stored in $A[x0]$ with probability $1/m$, so the expected length of $A[x0]$ is either $\alpha = n/m$ (if x is not contained in the table) or $1 + (n - 1)/m$ (if x is contained in the table).
- Thus, the expected cost of searching for or deleting an element is $O(1 + \alpha)$.
- Hashing with chaining supports the set ADT operations in O(1) expected time per operation, as long as the occupancy, α, is a constant.

Random Probing-Hashing with Chaining

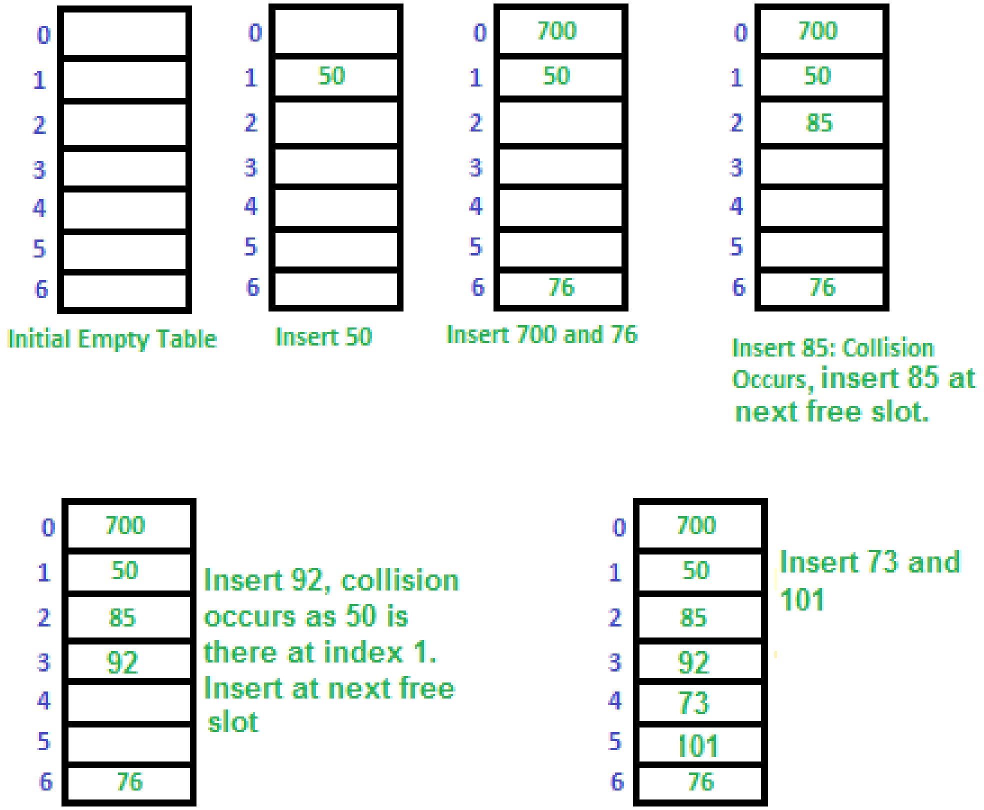
- The *worst-case search time* defined as

$W = \max\{\text{length of the list stored at } A[i] : 0 \leq i \leq m - 1\}$

- *The length of each list A[i] is a binomial(n, 1/m) random variable.*

Random Probing-Hashing with Open Addressing

- Each table position *A*[*i*] is allowed to store only one value.
- When a collision occurs at table position *i*, one of the two elements involved in the collision must move on to the next element in its probe sequence.
- **Working of insertion, deletion and search**



Random Probing-Quadratic Probing

Quadratic probing-example

Insert: 89, 18, 49, 58, 9 to table size=10, hash function is: %tablesize

0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89