Hash function:

Dictionaries and Hash table

A computer dictionary is similar to a paper dictionary of words in the sense that both are used to look things up.

the main idea is that users can assign keys to elements and then use those keys later to look up or remove elements

thus, the dictionary abstract data type has methods for the insertion, removal, and searching of elements with keys.

Ex:
A conceptual illustration of the dictionary ADT.
Keys (labels) are assigned to elements (diskettes) by a users.
the resulting items (labeled diskettes) are inserted into the dictionary (file cabinet).
the keys can be used later to retrieve or remove the items.

The Unordered Dictionary ADT.
A dictionary stores key-elements pairs (k, e). which we call items,
where k is the key and e is the elements.
for example, in a dictionary storing student records (such as the student's name, address, and course grades).
the key might be the student's ID number.
So some applications, the key may be the element itself.

We distinguish two types of dictionaries,
the unordered dictionary
&
ordered dictionary.

In both the case, we use a key as an identifier that is assigned by an application or user to an associated element.

for the sake of generality, our definition allows a dictionary to some multiple items with the same key.

* Nevertheless, there are application in which we want to disallow items with the same key.
for example: in a dictionary storing student.

As an ADT, a dictionary D supports the following fundamental methods.

findElement(K): if D contains an item with key equal to K, then return the element of such as item, else return a special element NO_SUCH_KEY.

insertItem(K,e): Insert an item with elements e and key K into D.

removeElement(K): Remove from D an item with key Equal to K, and return its elements. If D has no such item, then return the special element NO_SUCH_KEY.

Note that if we wish to store an item e in a dictionary so that the item is itself its own key, then we would insert e with the method call insertItem(e,e).

In addition, a dictionary can implement other supporting methods, such as the usual size() and isEmpty() methods for containers. Moreover, we can include a method, element(), which returns the elements stored in D, and keys().

<u>log files:</u>

A simple way of realizing a dictionary D uses an unsorted sequence S, which in turn is implemented using a vector or list to store the key-element pairs. Such an implementation is often called a <u>log file</u> or <u>audit trail</u>.

a log file are situations where we wish to store small amounts of data or data that is unlikely to change much overtime.

→ We also refer to the log file implementation of D as an unordered sequence implementation.

* the space required for a log file is $\Theta(n)$, since both the vector and linked list data structures can maintain their memory usage to be proportional to their size.

* with a log file implementation of the dictionary ADT. we can realize operation insertItem (K,e) easily and efficiently, just by a single call to the insertLast method on S, which runs in O(1) time.

unfortunately, a findElement(K) operation must be performed by Scanning the entire sequence S,
examining each of its items.

* the worst case for the running time of this method clearly occurs when the search is unsuccessful, and we reach the end of the sequence having examined all of it's n items.

Thus, the findElement method runs in O(n) time.

Similarly, a linear amount of time is needed in the worst case to perform a removeElement(K) operation on D, for in order to remove an item with a given key.
We must first find it by scanning through the entire sequence S.

Hash Tables.

the keys associated with elements in a dictionary are often means as "addresses". for those elements.

Example: of such applications include a compiler's symbol table and a registry of environment variables.

Both of these structures consist of a collection of symbolic names where each name serves as the "address" for properties about a variable's type and value.

one of the most efficient ways to implement a dictionary in such circumstances is to use a hash table.

Although, as we will see, the worst-case running time of the dictionary ADT operations in O(n). when using a hash table, where n is the number of items in the dictionary.

A hash table can perform these operations in O(1) expected time.

It consists of two major components, the first of which is a bucket array.
the second part of a hash table structure is a function, h, called a hash function.

DSE- Hashing function:

1. why hashing
2. Ideal hashing
3. Modulus hash function
4. Drawbacks.
5. Solution.

Why hashing

↳ used for searching

1. Linean } O(n)
2. Binary } ↳ search.

↳ O(logn)

Keys 8, 3, 6, 10, 15, 18, 4.

A

8	3	6	10	15	18	4
0	1	2	3	4	5	6

⟶ Linear

A

3	4	6	8	10	15	18
0	1	2	3	4	5	6

⟶ Binary search.

H

-	-	-	3	4	-	6	-	8	-	10	-	-	-	-	15	-	-	18		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

the search time takes O(1).

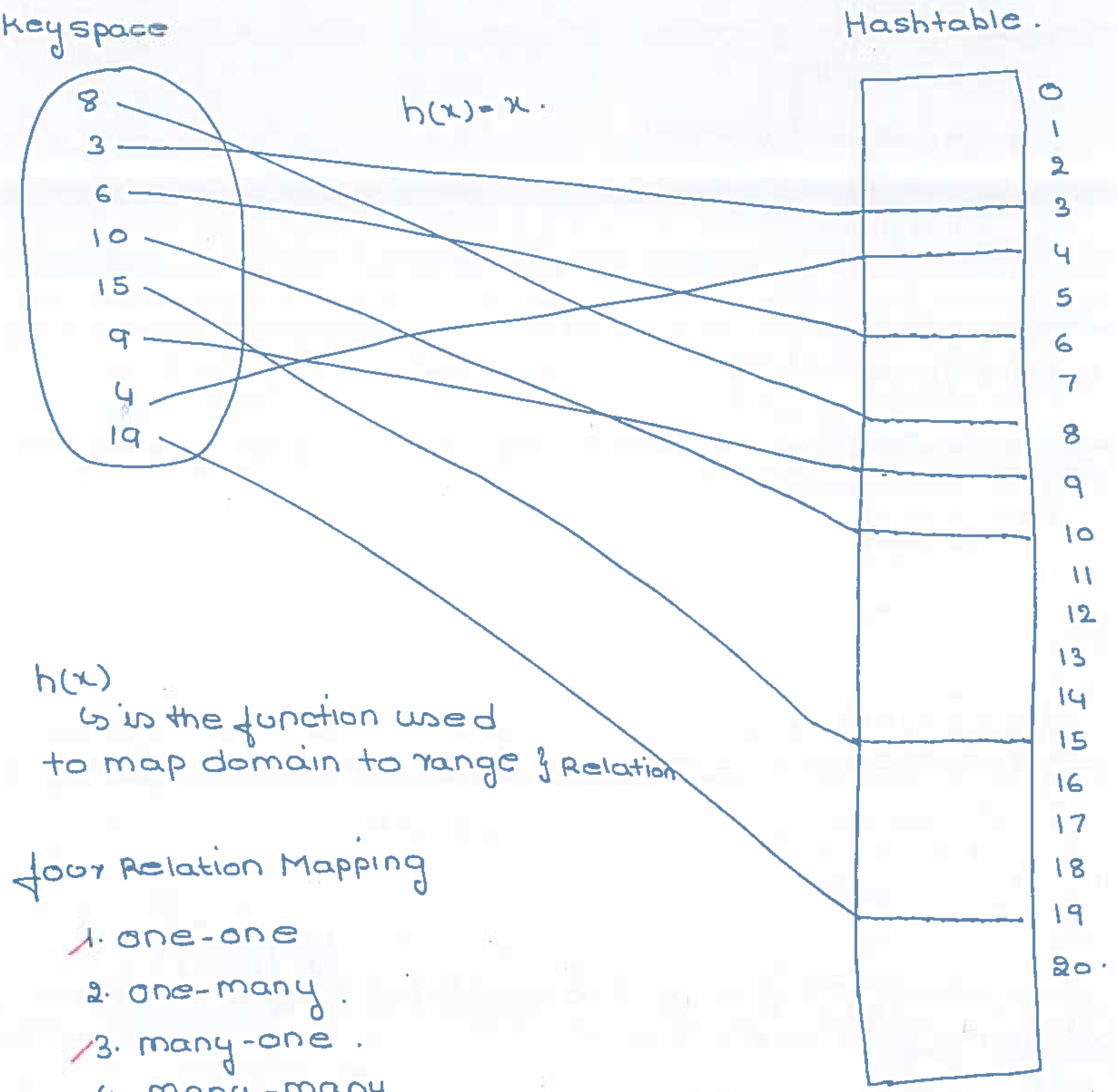
↳ as constant time

If anything is also missing

↳ space is used on effectly.

Mathematical Model of Hashing

key 8, 3, 6, 10, 15, 9, 4.



(this is array where key are stored).

h(x)
↳ is the function used to map domain to range { Relation

four Relation Mapping

1. one-one
2. one-many.
3. many-one.
4. many-many.

one-one & many-one are only function.

* The function that we used to mapping h(x).
hash function can be used for searching

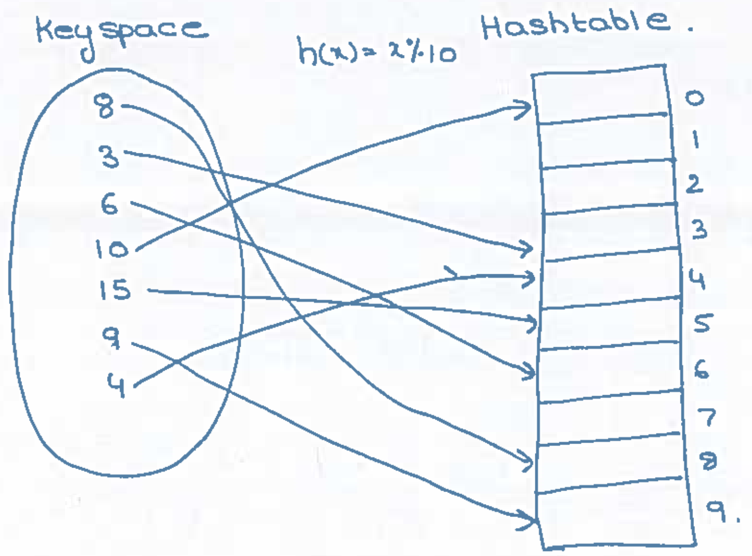
drawback. of ideal hashing
↳ space required is very hug inspite of number are low, who is responsable for this issue it's the hash function

* To reduce the space, and still have the hash we can modife the fonction

Let's modify the hash function to

$h(x) = x \% 10$ → mode.

↳ which will only have



drawback of mode function

Let assume we have value 25,

and 10)25(2
20
5 } where the hashtable 5 is mapped,

Now 25 and 15 are mapped to same key to location

* Two key mapped. one function we call it as collision 2 no more it is one-one function, it is now many to one. is due function

↳ How to resolve the collision

↳ open hashing
chaining { consume of more space

↳ closed hashing (open Addressing) { only the size of hash table remain same)

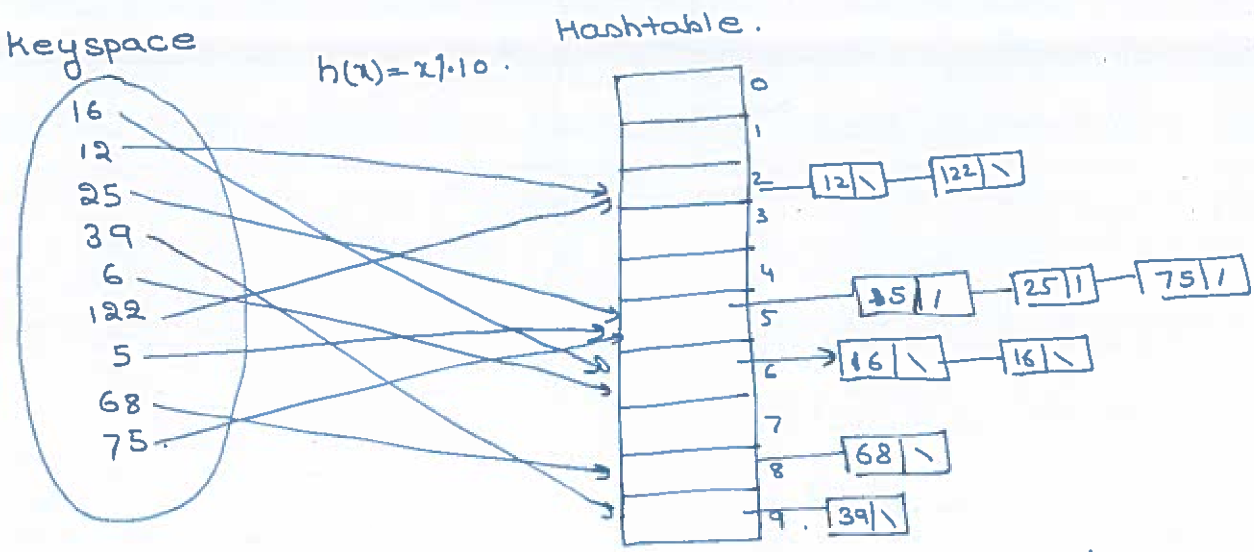
no Extra space.

1. linear probing
2. quadratic probing
3. Double Hashing

Hashing technique:

chaining

Key: 16, 12, 25, 39, 6, 122, 5, 68, 75



Array of chain (or) Link list

[Array of pointer]

1. Insert.
2. Search
3. Analysis
4) Delete - delete node like link list

searching

Key(12) -
Key 75 ✓
Key 65 ✗
Key 15 ✗.

Analysis of search

n=100, no of Key.

the size of hash table 10,

2 there is no upperlimit

(loading factor).

$\lambda = n/size$

* Analysis of the hashing is done on loading factor.

λ lamda $= 100/10 = 10$

which the 10, we can assume each location will have 10, key in

* Time take for successfuly search (Avg)

$t = 1 + \lambda/2$ (Average time of load).

Avg. unsuccessful search

$t = 1 + \lambda$ (maximum time).

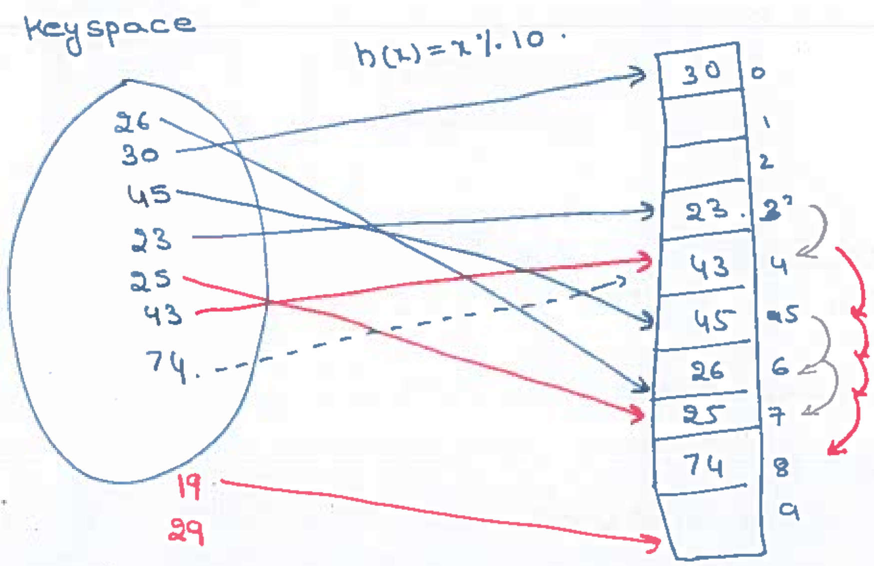
Delete the key: found, delete link del a node in link list

A unique problem:

key like

5
35
95
145
175
265
845
55
25.

Hashing technique (Linear probing).



to store 74, we had 4 - collision

after 2, collision the value is placed in the hashtable.

the formula used is

$$h'(x) = (h(x) + F(i)) \% 10$$

where $F(i) = i$

$i = 0, 1, 2, 3, 4 \ldots$

$h'(25) = (h(25) + f(0)) \% 10$

$= (5+0) \% 10$

$= 5$ here there is a collision

here there is a collision

$h'(25) = (h(25) + f(1)) \% 10$

$= (5+1) \% 10 = 6$, here there is a collision

$h'(25) = (h(25) + f(2)) \% 10$

$= (5+2) \% 10 = 7;$

$h'(29) = (h(29) + f(0)) \% 10$

$= (9+0) \% 10 = 9$

$= (9+1) \% 10 = 0$

$= (9+2) \% 10 = 1$

for search the same hash function is used

key = 45

key = 74 {it did 5-compression}

↳ it's more time complexity

key = 40

↳ search for element, if you find empty space then the element is not found.

Loading factor → $\lambda = n/\text{size} = \lambda = 9/10 = 0.9$.

Avg. successful search.

$$t = 1/\lambda \ln(1/1-\lambda)$$

Avg. unsuccessful search.

$$t = 1/1-\lambda$$

Loading factor, should be less than 0.5

$\lambda \leq 0.5$.

drawback:

↳ space waste,

↳ clustering of key can be formed

delete of element:

find & delete.

hashtable.

0	30
1	29
2	
3	23
4	43
5	45
6	26
7	~~25~~ → 25
8	74
9	19

after delete of 25, you should move 74, to 25 possition, if this is not done. when we try to find 74, this will shown as number not found.

To avoid this, we have to do rehashing which is really difficult.

So, in linear probing delete is not suggested

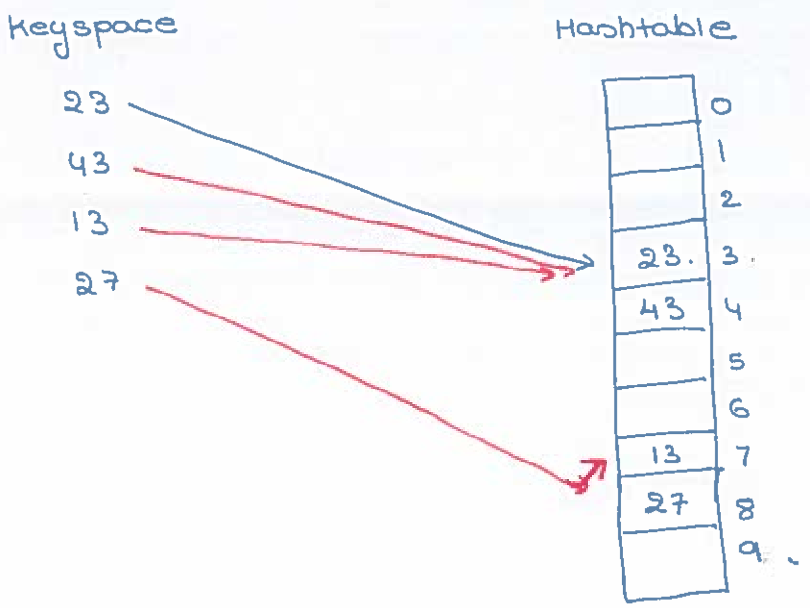
* Rehashing can be solved using flag.

Hashing technique (qudratic Probing)

$$h'(x) = (h(x) + f(i)) \% 10$$

where $f(i) = i^2$

$i = 0, 1, 2, \ldots$



so 43 is stored at '3'

for 43.

$h'(x) = (h(x) + f(i)) \% 10$

$= (3+0) \% 10 = 3$.

$h'(x) = (h(x) + f(i)) \% 10$

$= (3+1) \% 10$

$= 4$.

for 13.

$h'(x) = (h(x) + f(i)) \% 10$

$= (3+0) \% 10 = 3$.

$h'(x) = (h(x) + f(i)) \% 10$

$= (3+1) \% 10 = 4$.

$h'(x) = (h(x) + f(i)) \% 10$

$= (3+4) \% 10 = 7$.

Now 27

$h'(x) = (h(x) + f(1)) \% 10$

$= (7+0) \% 10 = 7$

$h'(x) = (h(x) + f(i)) \% 10$

$= (7+1) \% 10$

$= 8$

Hashing Technique (Double Hashing)

$h_1(x) = x \% 10$

$h_2(x) = R - (x \% R)$

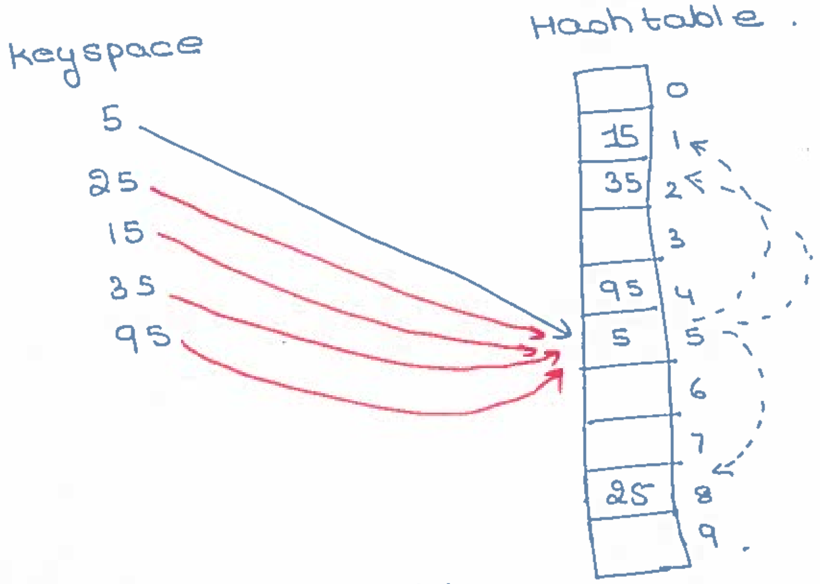
$h'(x) = (h_1(x) + i * h_2(x)) \% 10$

where $i = 0, 1, 2, \ldots$

where R is prime number

* condition of h_2 is it should not result 0.

2 it should try to cover all index.



$h'(25) = (h_1(x) + i * h_2(x)) \% 10$ $\quad$ $h_2(25)$

$= (5 + 1(3)) \% 10$ $\quad$ $= 7 - (25 \% 7)$

$= 8 \% 10$ $\quad$ $= 7 - 4 = 3.$

$= 8$

$h'(15) = (h_1(x) + i * h_2(x)) \% 10$ $\quad$ $h_2(15)$

$= (5 + 1 * (6)) \% 10$ $\quad$ $= 7 - (15 \% 7)$

$= (11) \% 10$ $\quad$ $= 7 - 1 = 6.$

$= 1$

$h'(35) = (h_1(x) + i * h_2(x)) \% 10$ $\quad$ $= 7 - (35 \% 7)$

$= (5 + 1 * 7) \% 10 = 2$ $\quad$ $= 7 - 0 = 7.$

$h'(95) = (h_1(x) + i * h_2(x)) \% 10$ $\quad$ $7 - (95 \% 7)$

$= (5 + 1 * (3)) \% 10$ $\quad$ $= 7 - 4 = 3.$

$= 8$ (the value is there)

now change $i = 2$.

$h'(95) = (5+2*3)\%10 = 1$ (again collesion)

now $i = 3$

$h'(95) = (5+3*3)\%10$

$= 4.$

Hash function

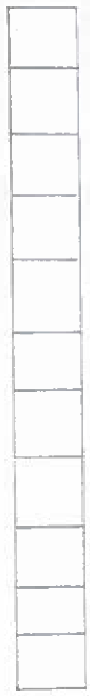
$h(x) = x \% \text{size}$

or

$h(x) = (x \% \text{size}) + 1$ if not start from "0"

↑ prime no

size = 11



1. mod.
2. mid square.
3. folding

Mid square

Key = 11

$= (11)^2 = 121$, now store 11 @ 2.

↑

key = 13

$= (13)^2 = 169$, now store 13 @ 6.

↑

If some square value are larger then above Example, still you can take mid number

if the size of square is like - - ‿ - - (6 digit value)

↑ now take this two values as the value of hash function

* per example if the mid value is not exists in hash table, then reapply hash function on the mid value that you selected

folding

Key = 12347

then

$$\begin{array}{r} 12 \\ +\ 33 \\ +\ 47 \\ \hline 92 \end{array}$$

92 → 9+2 → 11 @ now store the key in the function.

if key is string for example: "ABC"

A	B	C
65	66	67

$$\begin{array}{r} 65 \\ +\ 66 \\ +\ 67 \\ \hline \cancel{1}98 \end{array}$$

→ you can only use the last element