typedef int Dictionary[SIZE];

* Closed Hashing

- linear hashing or circular array

Hash values

H(a) = 3
H(b) = 9
H(c) = 4
H(d) = 9
H(e) = $\boxed{9}$
H(f) = $\boxed{0}$
H(g) = 1
H(h) = 3

D

0	e
1	f
2	EMPTY
3	a
4	DELETED
5	d
6	EMPTY
7	EMPTY
8	EMPTY
9	b

*insert() is O(1) bcs it goes directly to location

* Key Terms

- Synonyms - elements with the same hash value
  - ex: a & d are synonyms
  - results in COLLISION

- collision - when an element is inserted in an already occupied space
  - solution: Linear Hashing

- Displacement - when an element is inserted in an already occupied space by a non-synonymous member
  - ex: f & e
  - solution: Linear Hashing

* Operations

1.) initialize() - set each cell of dictionary to EMPTY

2.) member () - search for element stops when:

> element is found

> EMPTY cell is encountered

> number of comparisons = MAX

↳ counter variable

3.) insert () - elements must be UNIQUE but two different elements can have same HV

- inserts element at EMPTY / DELETED cell but prioritizes SEARCH LENGTH.

4.) delete () - when element is found, mark as DELETED

---

* Advantages		* Disadvantages	
CH	OH	CH	OH
- exact location	- open space	- collision	- cant O(1)
- O(1)	- no collision		

* Linear Hashing

- looking for next available space in circular array
- uses % operator

FORMULA: " $H_1(x) = (H(x) + i) \% MAX$ "

* EMPTY and DELETED

When calling the isMember() fn. for closed hashing, it is: O(1) because hash fn() returns exact location

The function:

> STOP search at empty slot
> CONTINUE search when it encounters a deleted elem

So, differentiate EMPTY and DELETED using macros:

```
#define EMPTY 0
```

```
#define DELETED -1
```

* these "markers" should be the SAME data type as elements in dictionary.

//Exercise (Average search Length)

Search Length

SL = Actual location of x − Hash (x) + 1

Average Search Length

Ave SL = sum of SL / no. of elements / MAX

* SL is if dictionary is circular / doesn't rotate.

* Ave SL is to see if hash fn() is efficient & correct.

Exercise
(Average Search Length)

Hash Values
Hash(A) = 1
Hash(B) = 4
Hash(C) = 9
Hash(D) = 9
Hash(E) = 0
Hash(F) = 3
Hash(G) = 4
Hash(H) = 3

Dictionary D

0	EMPTY
1	EMPTY
2	EMPTY
3	EMPTY
4	EMPTY
5	EMPTY
6	EMPTY
7	EMPTY
8	EMPTY
9	EMPTY

Do the following:

1) Insert the elements A, B, C, D, E, F, G, and H in an initially empty dictionary with hash values 1, 4, 9, 9, 0, 3, 4, and respectively. Note: Solution for collisi[on] is linear hashing, i.e. next available space in the dictionary which is treate[d] as a circular array
2) Determine the search length of each element. Search length (SL) of element x:

   SL = Actual location of x − Hash(x) + 1
3) Determine the Average Search Length of all 8 elements

   Ave SL = sum of SL / no. of elements

SL = Actual location of x − Hash(x) + 1

Average Search Length

Ave SL = sum of SL / no. of elements / MAX

* SL is if dictionary is circular / doesn't rotate.

* Ave SL is to see if hash ...

Insert

H(x)

A = 1
B = 4
C = 9
D = 9 → 0
E = 0 → 2
F = 3
G = 4 → 5
H = 3 → 6

0	D
1	A
2	E
3	F
4	B
5	G
6	H
7	//////
8	//////
9	C

SL

A = (1−1) + 1 = 1
B = (4−4) + 1 = 1
C = (9−9) + 1 = 1
D = (0−9) + 1 = −8 (?)
E = (2−0) + 1 = 3
F = (3−3) + 1 = 1
G = (5−4) + 1 = 2
H = (6−3) + 1 = 4

5

Ave SL = 5 / MAX = 5/10 or 0.5

Hash Values

Hash(A) = 1
Hash(B) = 4
Hash(C) = 9
Hash(D) = 9
Hash(E) = 0
Hash(F) = 3
Hash(G) = 4
Hash(H) = 3

Dictior

0	EM
1	EM
2	EM
3	EM
4	EM
5	EM
6	EM
7	EMPTY
8	EMPTY
9	EMPTY

SL = Actual location of x − Hash(x) + 1

3) **Determine the Average Search Length of all 8 elements**

Ave SL = sum of SL / no. of elements

NO:

DATE:

* "Perfect" Hash fn()

- returns a unique value FOR EACH element
- has no collisions, no synonyms : O(1)
- "holy grail"

* Load factor / packing density

- ratio of no. of elements to be stored to no. of available spaces
- rule of thumb : 80%
  - ↳ more space = less likely for collision / synomyms

* Packing density & collision = DIRECTLY PROPORTIONAL

* "apple children" analogy

Packing Density

# of elems / x = 80%

x/0.80

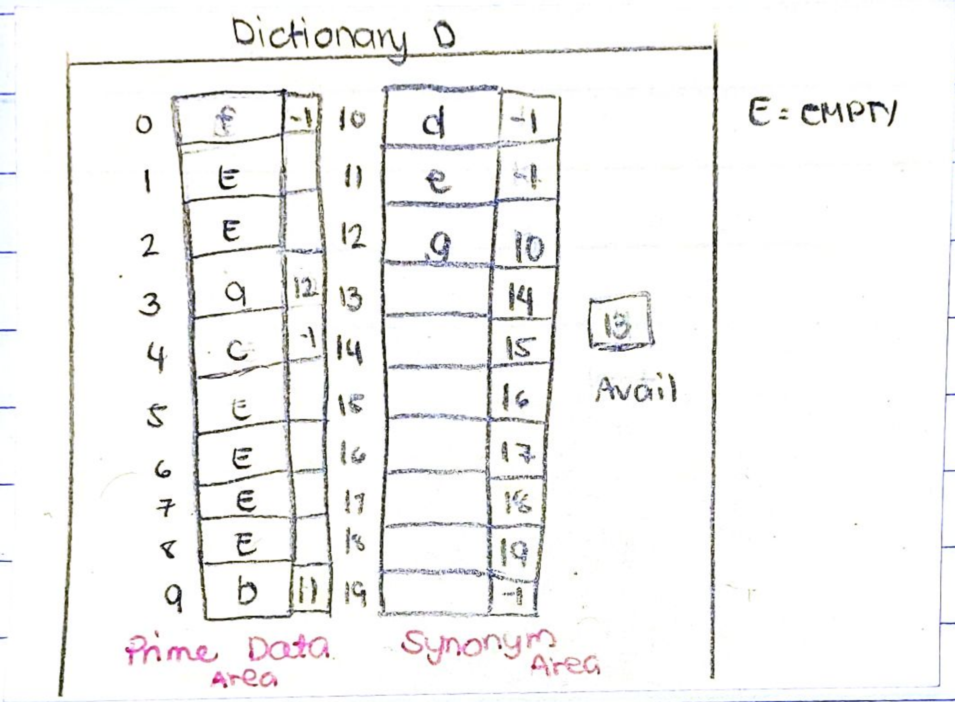
x is no. of spaces

- SOLUTIONS TO CO

1.) Linear Hashing — most common
2.) Group synonyms in one location — Variation 3
   ↳ can be : LL, CBL
3.) Double Hashing

* Variation 3 - the most efficient / "semi-open hashing"

Placing synonyms in separate area and synonym area cells are linked together during initialization. Last variable is changed to AVAIL.



* non synonym elems have link : -1

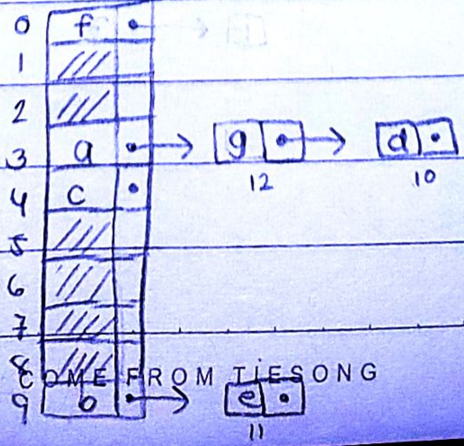
* If element has a synonym, it will be inserted in the synonym area, with link node updated to -1 to indicate its the last / links to original element. Original element in prime data area will have its link node updated to link to synonym.

* will look like an open hashing implementation

Insert	Hash Value (H(x))
a	3
b	9
c	4
d	3
e	9
f	0
g	3

Avail: 13

PRIME DATA AREA SYNONYM AREA



* this is insertFirst()!

NO:

DATE:

For deleting(),

- don't change link of element because it needs to still be connected
- just mark as DELETED
- return slot to Avail (if in synonym area)

* Operations

- code in C file -

- same as open hashing -

* Write definition of Dictionary.

* Operations: (closed Hashing)

> Init
> Member
> Insert
> Delete

NO:

DATE:

-CLOSED HASHING OPERATIONS
VARIATION 3

```
typedef struct node {
    char data;
    int link;
} nodetype;

typedef struct {
    nodetype Node[MAX];
    int avail;
} Dictionary;
```

//initialize()

```
#define MAX
#define EMPTY 0
#define DELETED -1

void init
```