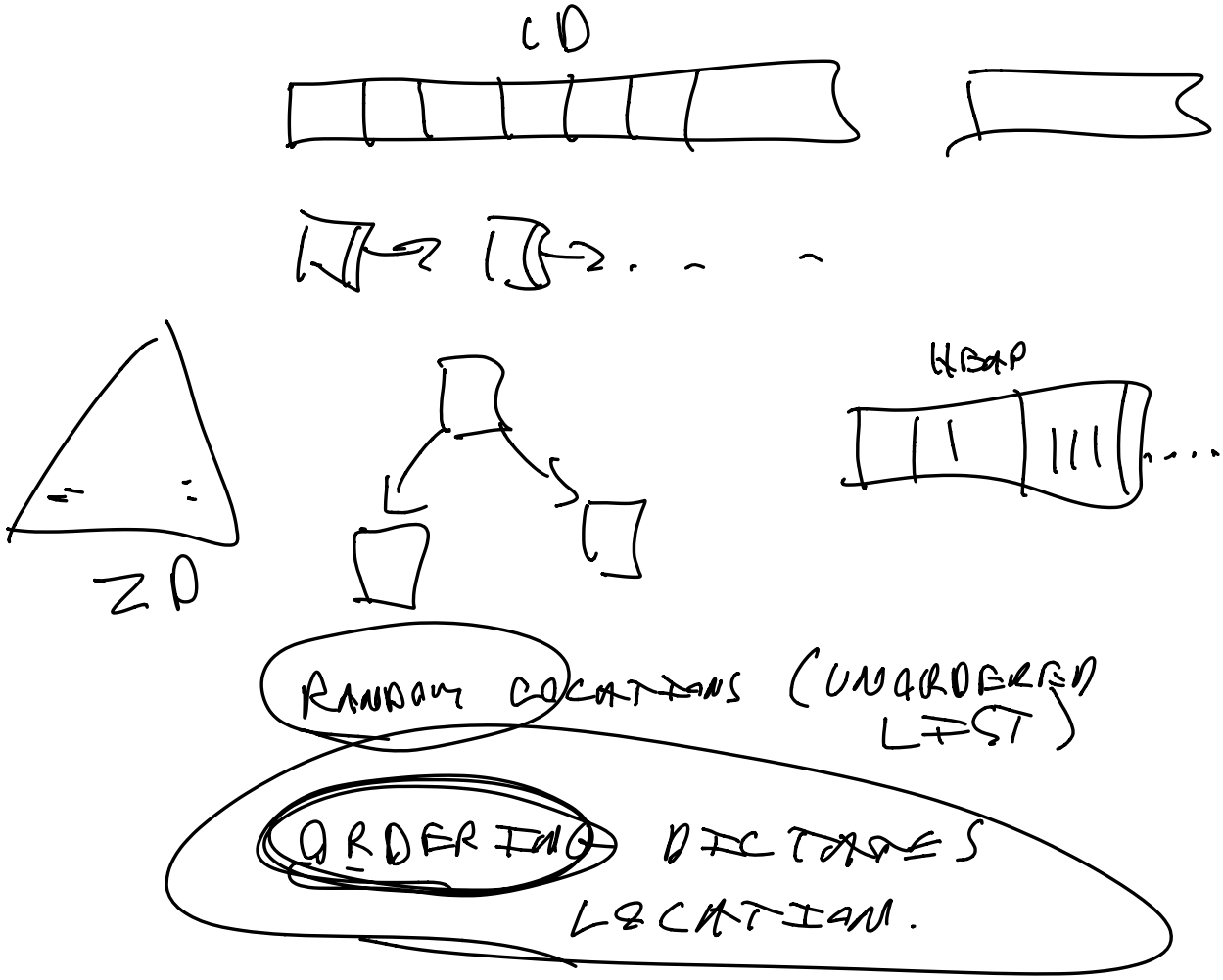


HASH TABLE

EXPLOITS "POWER OF RANDOMNESS"

WORST CASE TIME $O(N)$

AVG CASE $O(1)$

PERFECT DATA STRUCTURE ?
(SAY FOR DICTIONARY)

INSERT
MEMBER(FIND
~~SIZE~~
DELETE
~~ISEMPTY~~

FOR INTS

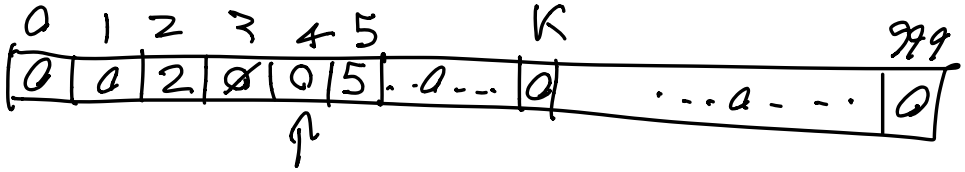
CHARACTERISTICS

✓ EFFICIENT O(1) FOR ALL OPERATIONS

✓ EASY TO IMPLEMENT & UNDERSTAND

✓ NOT USE EXTRA STORAGE

```
INT SIZE = 100000000;
INT[]A = INT{SIZE}; // DEFAULT: FILLED W. 0'S
```




```
VOID INSERT(INT K){
    A[K] = K;
```

INSERT(3)
INSERT(5)

```
}
VOID DELETE(INT K){
    A[K] = 0;
}
BOOLEAN MEMBER(INT K){
    RETURN (A[K]==K);
}
```

MEMBER(4)?
DELETE(3)

2ND ATTEMPT

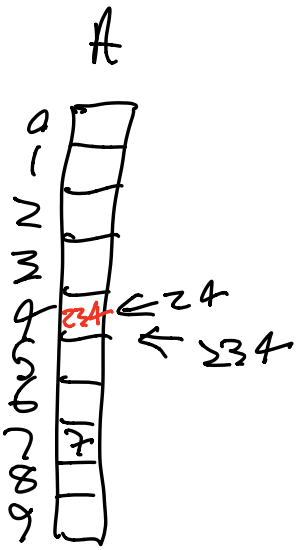
MAIN IDEAS:

(1) STORE KEY IN APPROPRIATE ARRAY SLOT

(2) STORE KEY IN A[KEY % SIZE]

```
INT SIZE = 10;
INT[]A = INT{SIZE};
```

```
VOID INSERT(INT K){
    A[K % SIZE] = K;
}
```




```
VOID DELETE(INT K){
    A[K%SIZE] = 0;
}

BOOLEAN MEMBER(INT K){
    RETURN (A[K%SIZE] == K)
}
```

INSERT(7) O(1)

MEMBER(7)

INSERT(24)

MEMBER(24) ✓

INSERT(234)

MEMBER(234) ✓

MEMBER(24) ✗

SIZE == 10

K % 10 ⇒ LAST DIGIT OF K

PROBLEM: CAN TRY TO INSERT 2 OR MORE KEYS IN 1 LOCATION.

CALLED A COLLISION

HASH TABLES w. SEPARATE CHAINING

```
INT SIZE = 10;
NODE[] T = NEW NODE[SIZE];
INT HASH(INT K) {
    RETURN (K % SIZE);
}
VOID INSERT(INT K) {
    T[HASH(K)] = INSERTH(K, T[HASH(K)]);
}
```

HEAD → [ ] → [ ] → ...

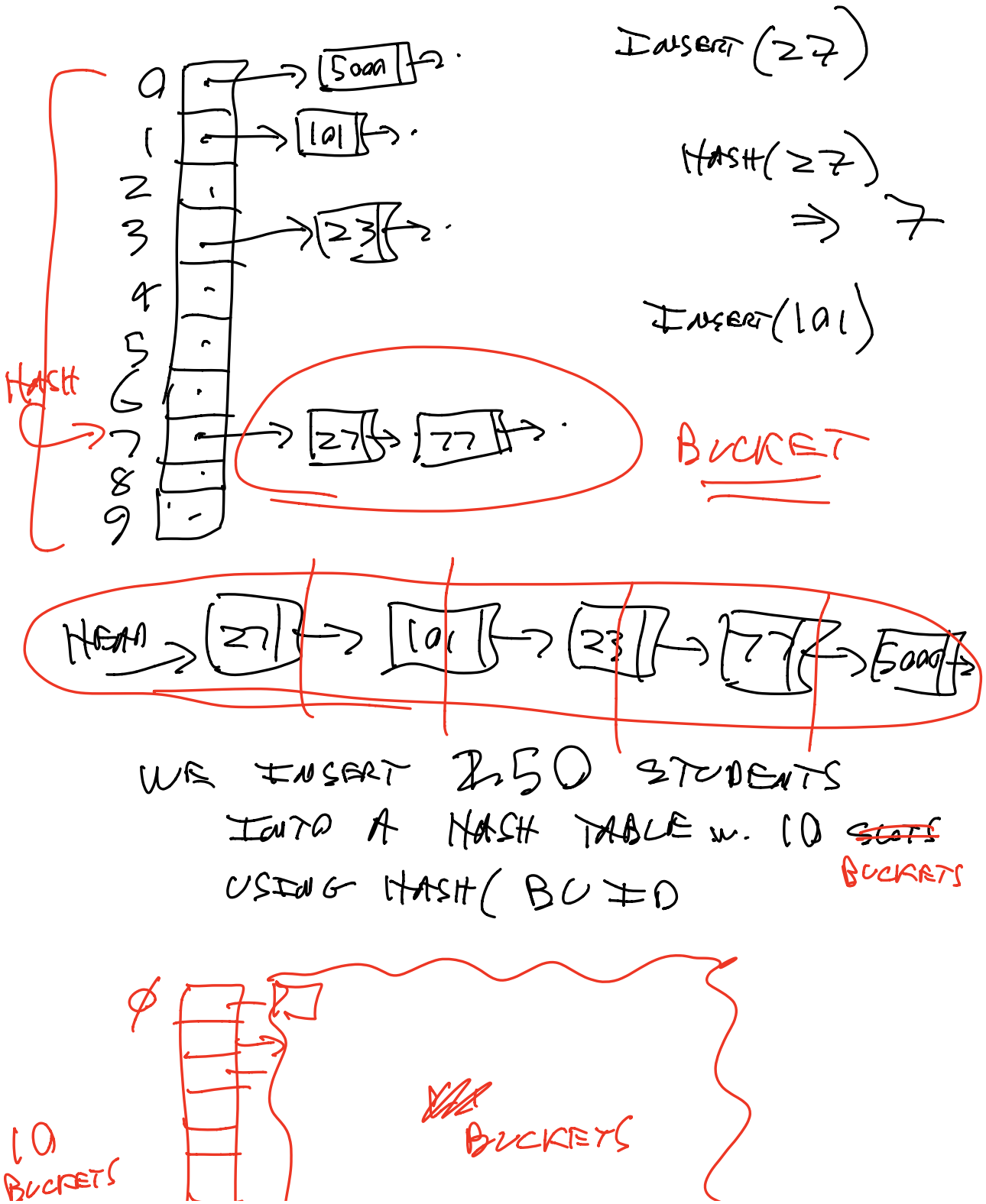
```
VOID DELETE(INT K) {
    T[HASH(K)] = DELETEH(K, T[HASH(K)]);
}
```

HEAD HEAD

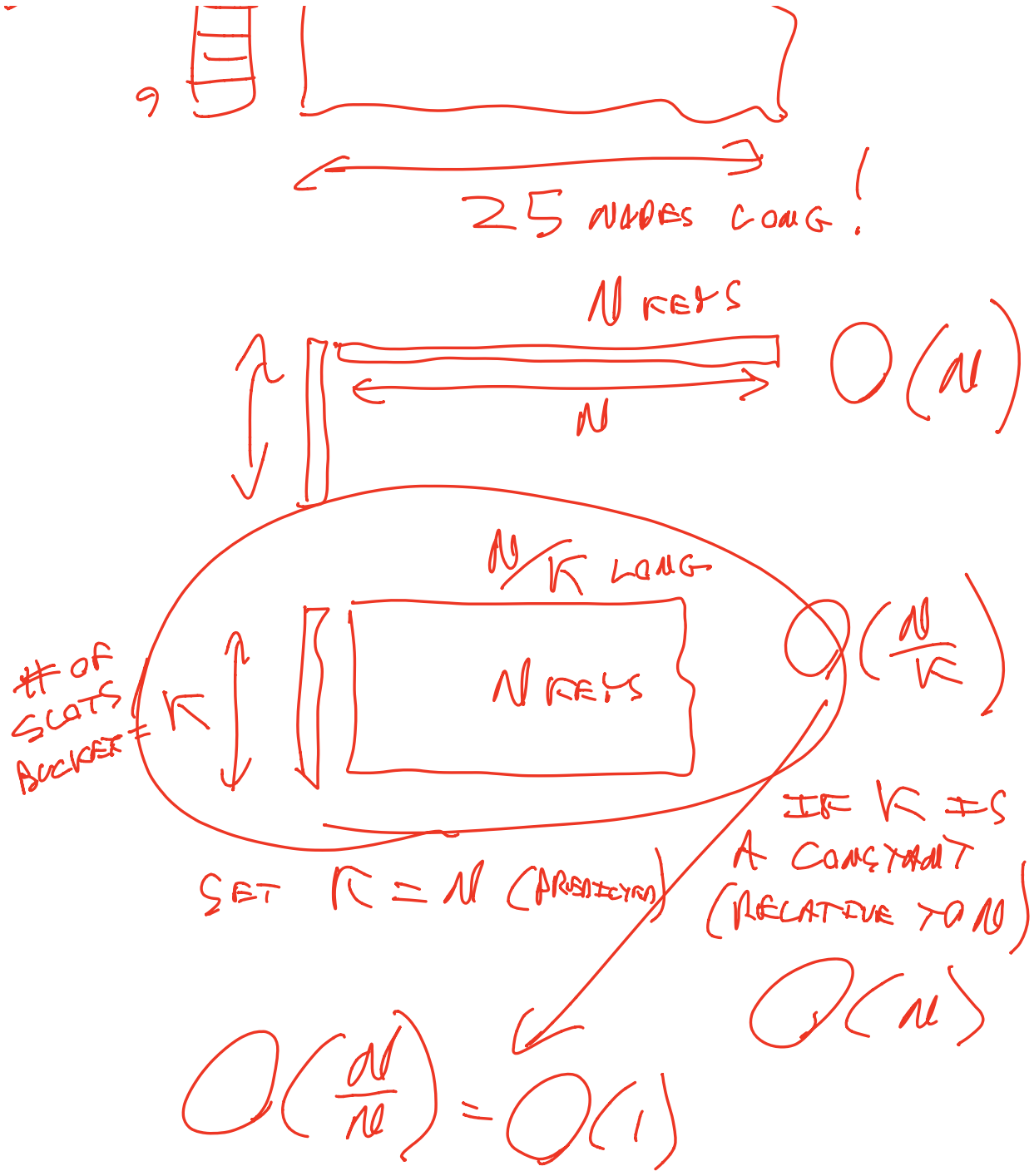
```
BOOLEAN MEMBER(INT K) {
    RETURN MEMBERH(K, T[HASH(K)]);
}
```

```
CLASS NODE {
    INT KEY;
    ;
    NODE NEXT;
}

NODE INSERTH(INT K, NODE P) {
    IF (P == NULL)
        RETURN NEW NODE(K, NULL)
    ;
}

NODE DELETEH(INT K, NODE P) {
    ;
}
```

```
BOOLEAN MEMBERH(INT K, NODE P) {
    ;
}
```



WE INSERT 250 STUDENTS
INTO A HASH TABLE W. 10 ~~SLOTS~~ BUCKETS
USING HASH(BUCKET


25 NODES LONG!
N KEYS
N
O(N)
N/K LONG
OF SLOTS BUCKET = K
N KEYS
O(N/K)
IF K IS A CONSTANT (RELATIVE TO N)
O(N)
SET K = N (PROBLEM)
O(N/N) = O(1)

CENTRAL ISSUE:

FINDING A HASH FUNCTION WHICH ~~SPREAD~~ ARRANGES KEYS "EVENLY" THROUGH OUT THE BUCKETS.

MATH PROBABILITY OF A RANDOM KEY GOING INTO A PARTICULAR BUCKET IS $\frac{1}{\text{SIZE}}$.

GOOD HASH FUNCTIONS HAVE THIS PROPERTY.

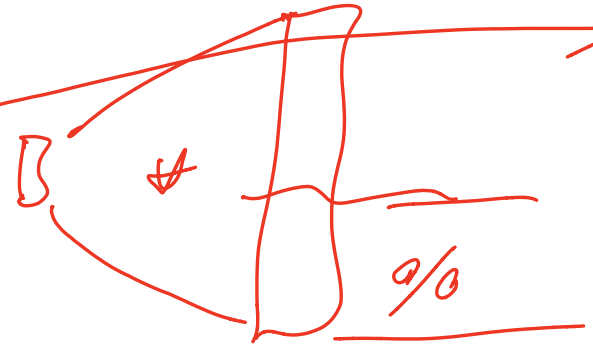
A SIMPLE WAY TO CREATE GOOD HASH FUNCTIONS IS TO USE 2 LARGE PRIME #S

SIZE = AS LARGE A PRIME AS YOU NEED FOR DATA

MULTIPLIER = A LARGE PRIME (OR; RELATIVELY PRIME TO SIZE)

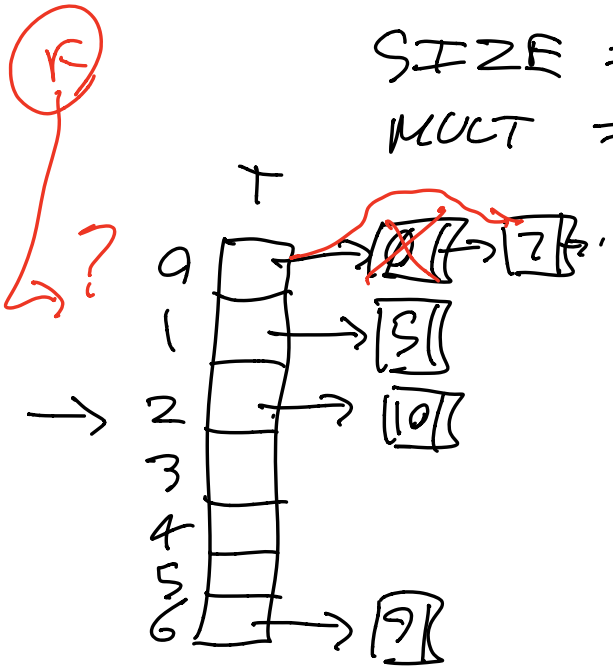
LINEAR CONGRUENTIAL METHOD.

```
INT HASH(INT KEY)
{
    RETURN (MULTIPLIER * KEY % SIZE)
}
```

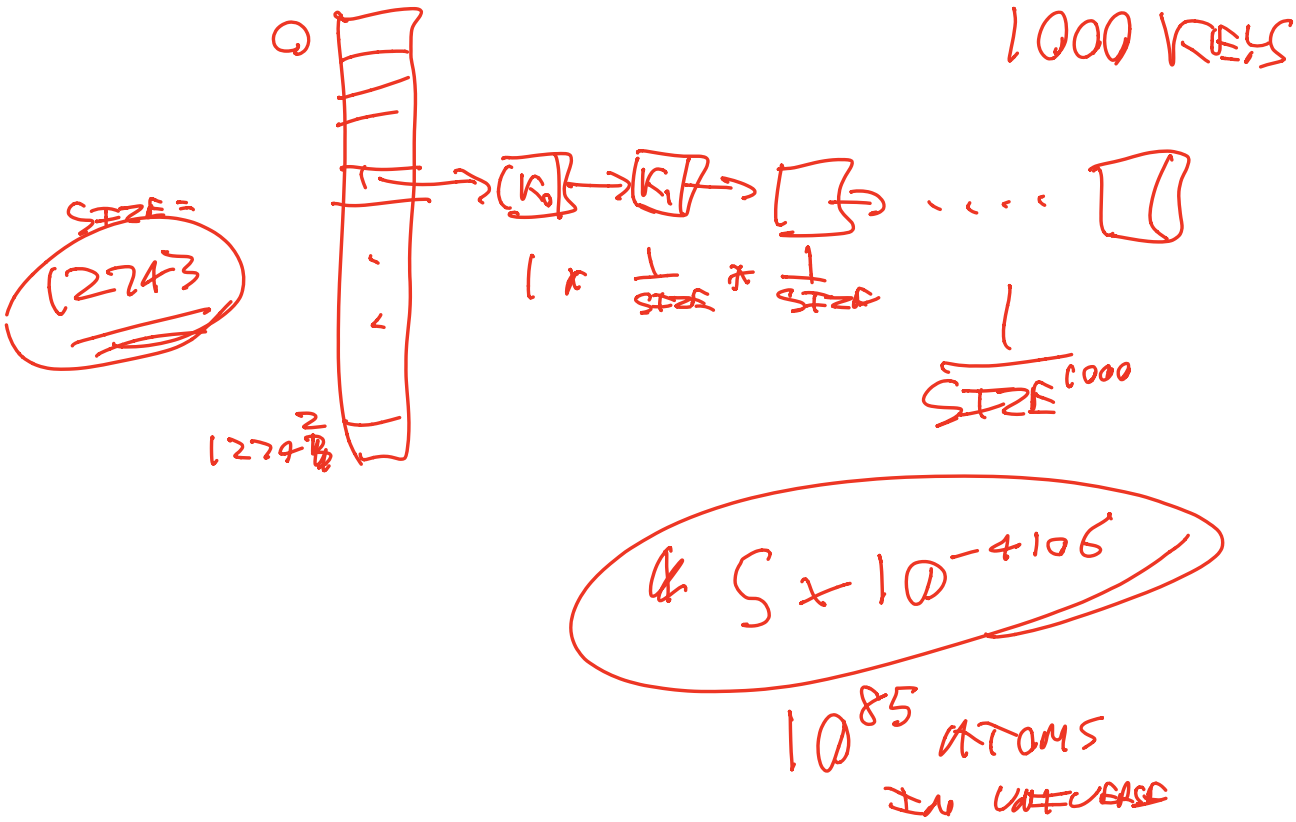


SIZE = 7

MULT = 3



INSERT	(3 * KEY) % 7
5	15 1
10	30 2
~~0~~	0 0
7	21 ~~0~~
9	27 6



$O(N)$ WORST CASE
EXTREMELY UNLIKELY

$O(1)$ ACCESS TIME ON AVG.