HashTable is a Dictionary

all operations fo O(1) (on average, and w/ good hash)

1. Storing k/v pairs in an array
2. Hashing fn: k → int
3. mod int to be in array bounds
4. Deal w/ "collisions".

Terms

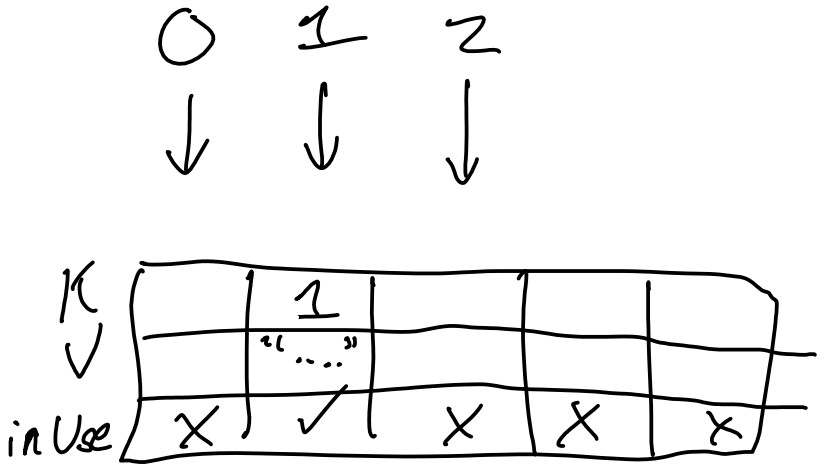
size — # of mappings

capacity — total allocated size of array

Collision Resolution

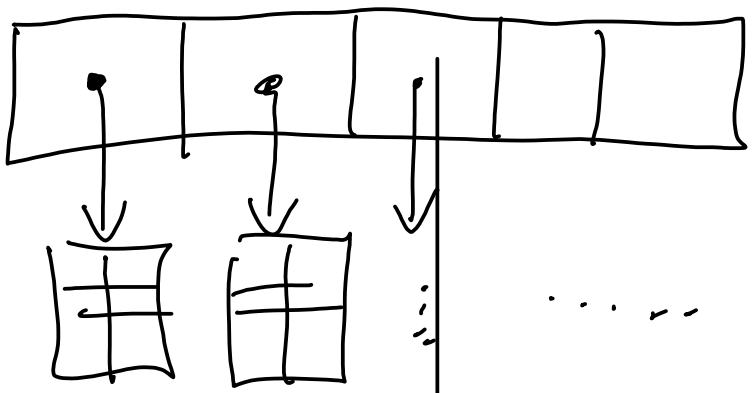
Linear probing (Tuesday)

if in use, go next slot



Other strategy:

Chaining

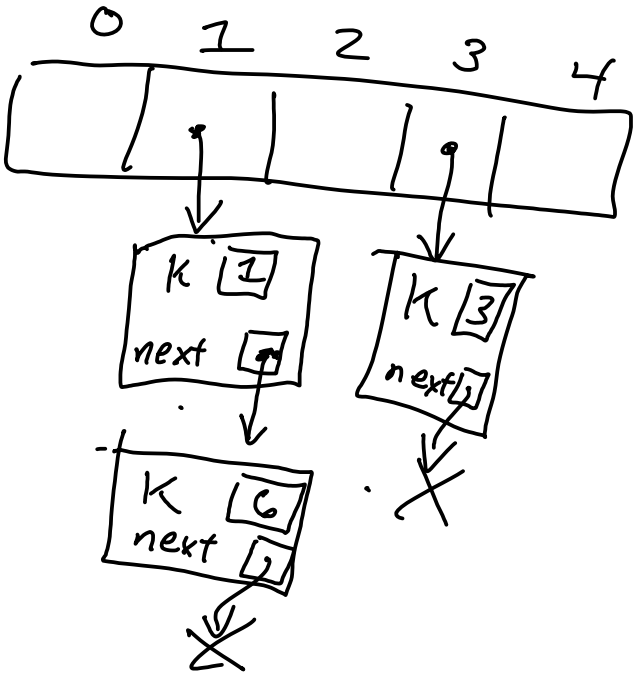


insert (1, ...)

insert (6, ...)

insert (3, ...)

insert (11, ...)



Dictionary impl:

LinearDictionary:

list (unsorted) of K/V pairs

Expanding HT capacity

load factor: $\frac{\text{size}}{\text{capacity}}$

max load factor

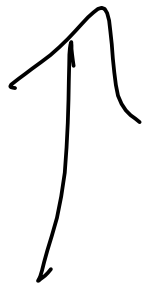
small	⟷	large
MLF		MLF
< collisions		> collisions
> mem usage		< memory usage

Method expandCapacity():

```
newArray ← new array of K/V pair lists; 2x size of prev
for each k/V pair in oldArray:
    h ← hash key
    i ← h % newArray capacity
    newArray[i].insertAtTail(pair)
update capacity
update array (to be newArray)
```

hash functions! $K \to int$

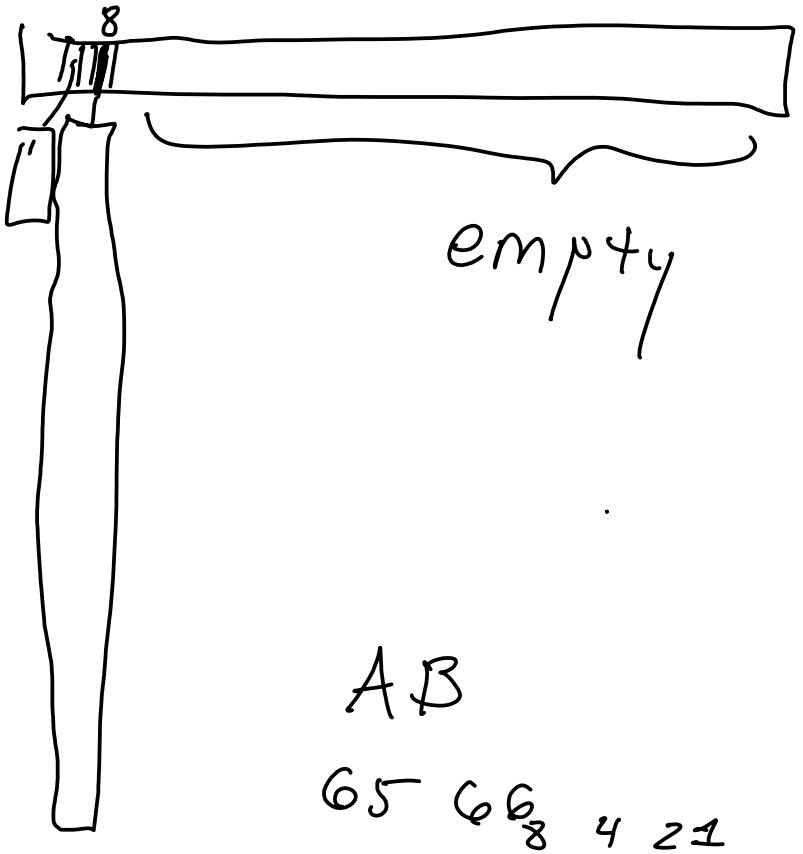
```
int hash(string s) {
    return 0;
}
```



NOT A GOOD HASH

good hash:
distribute K evenly over int

```
int hash(string s) {
    return s.length();
}
```



Better, but that's not saying much

AB

65 66 8 4 21

1000001

1 1 1 1 1 1

1 0 0 0 0 0 1

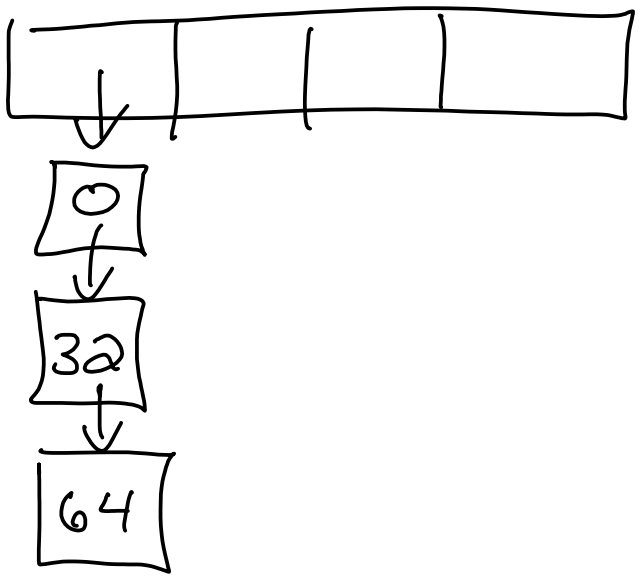
1 0 0 0 0 0 1

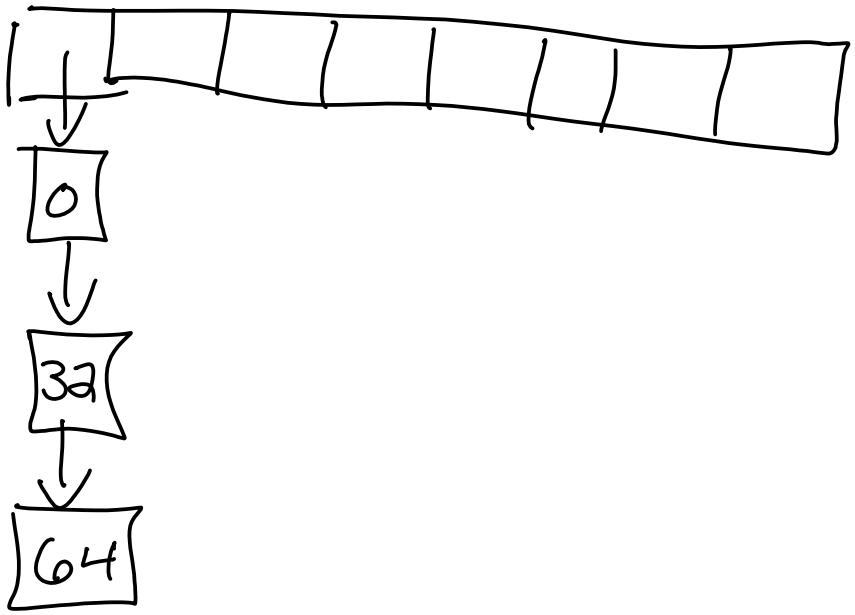
1 0 0 0 0 0 1

```
int hash(string s) {
    int n=0;
    for (int i=0; i<s.length(); i++) {
        n *=31;
        n += s[i];
    }
    return n;
}
```

```
int hash(int n) {
    return n;
}
```

insert(0, ...)
insert(32, ...)
insert(64, ...)
insert(96, ...)





Hash Table

get
insert
update
remove

- O(n) — worst — good hash
- O(1) — average
- O(n) — worst — bad hash