Hash Table

- Data structure has average O(1)* complexity for Dictionary operations (get, insert, update, remove)
- Backed using arrays
- (Tricks from ArrayList will be helpful)
- Each slot in array has a key-value mapping and some way of determining if the slot is in use
- (not interpreted as a tree)
- hash function K ⟼ int (should evenly distribute keys)
- Use hash of each key to decide where to put mapping
- Collision: when two keys want to be in the same slot (according to hash)

insert(1, "a")
insert(7, "b")
insert(5, "c")

K		5		7
V		"c"		"b"
bool	X	✓	X	✓

- One collision resolution strategy: (Linear probing)
  - Just use next slot
  - On get: check next slot too until I find it or slot is empty
  - On remove: check next slots to move collided keys back
- To prevent frequent collisions:
  - Calculate "load factor" $LF = \frac{\#\text{ mappings}}{\#\text{ slots}}$
  - If adding a mapping would increase LF beyond "max load factor", then double array size & rehash all mappings
  - Common MLF ≈ $1/2$, $3/4$, $2/3$

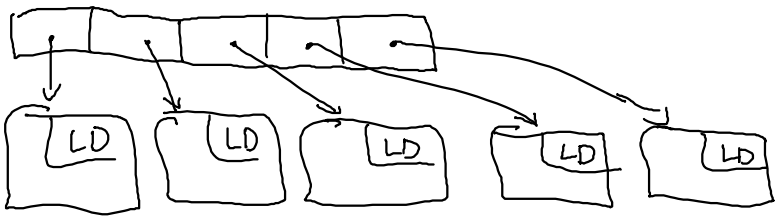
Collision resolution strategy: forward chaining

LinearDictionary: dictionary backed by ArrayList (not sorted)
get/insert/update/remove

```
Method get(K key):
    For each index i:
        If contents.get(i).first == key:
            Return contents.get(i).second
        EndIf
    EndFor
EndMethod
```

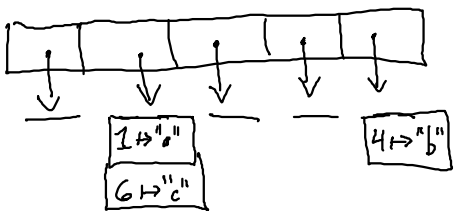
```
Method insert(K key  V value):
    For each index i:
        If contents.get(i).first == key:
        EndIf
    EndFor
    contents.insertLast((key, value))
EndMethod
```

Forward chaining:
Hash table contains an array of LinearDictionary objects

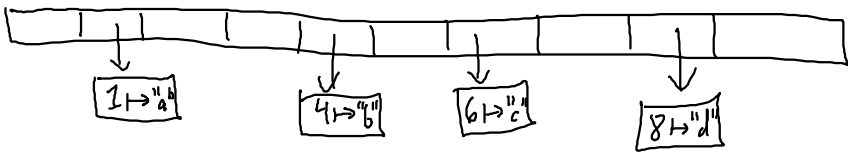


MLF = 3/4

insert(1, "a")
insert(4, "b")
insert(6, "c")
insert(8, "d")



exceed MLF: rehash everything



Hash table insert: WC O(n)

amortized average O(1)

expected (over chance)
amortized (over a sequence)
?? (over all possible inputs - keys)

There is always a set of keys that makes a Hashtable give O(n) performance