```
Function PosNums(n):
    a ← new ArrayList
    For i in 1 to n:
        a: insertAtTail(i)
```

worst case $O(n)$

amortized worst case $O(n)$

Complete binary trees

↓

ArrayHist

insert

$O(n)$ add

$O(\log n)$ bubble

$O(n)$

amort $O(1)$

$O(\log n)$

amort $O(\log n)$

Worst case $O(n)$

amortized worst case $O(1)$

avg complexity over a sequence

Hash Table (is a kind of Dictionary)

		unbalanced BST	AVL	HashTable (our goal)
	get	$O(n)$	$O(\log n)$	$O(1)$
put	insert	$O(n)$	$O(\log n)$	$O(1)$
put	update	$O(n)$	$O(\log n)$	$O(1)$
put	remove	$O(n)$	$O(\log n)$	$O(1)$

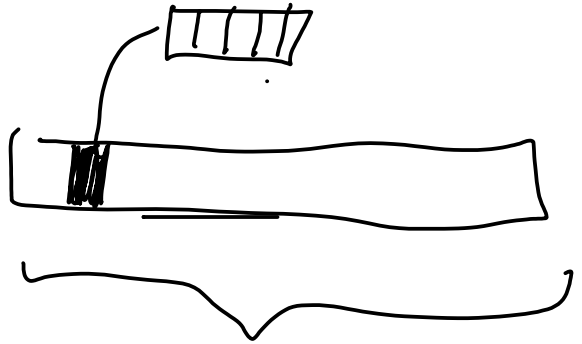
average

Dictionary

Constant time for all ops

1. all keys are int
2. all keys are positive
3. all keys < 1,000,000
4. ~~no keys will overlap~~

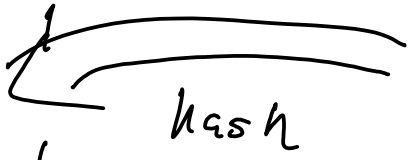
pair<pair<K, V>, bool>*



insert(2000005, "hi")

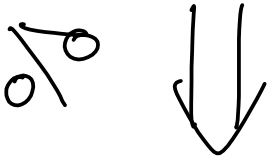
insert(5, "hi again")

→ "collision"



0. Any key

1. Any int key



2. Key in int range

hashing

Write hash func:

K → int

hash

"hello how are"

5023

3

Hashing Functions: $K \longrightarrow int$

when K is int:

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

Purpose of hashing function: turn key into an int

when K is string

```
int hash(string s) {
    int acc = 0;
    for (int i = 0; i < 4 && i < s.length(); ++i) {
        acc *= 256;
        acc += s[i];
    }
    return acc;
}
```

"hi"
"happy"
"bears"
"happiness"

```
stoi
int acc = 0;
for each c in str:
    acc *= 10;
    acc += c as num;
                (c - '0')
```

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

"hello" → 5

"bears"

Collisions: Linear Probing | Chaining

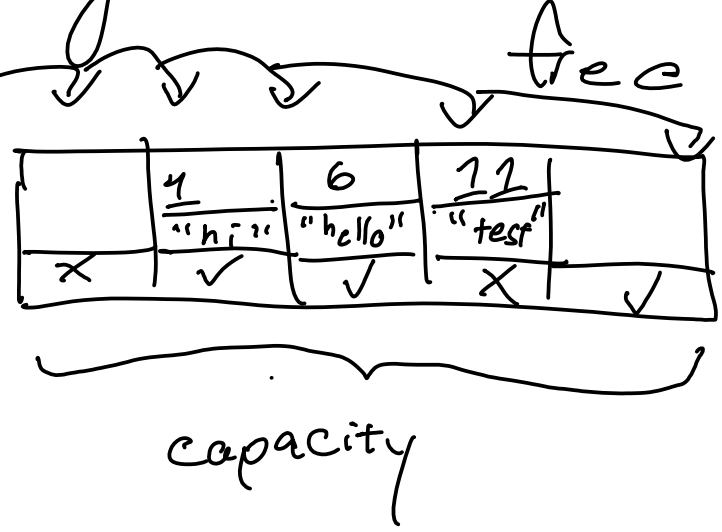
Hash Tables:

1. Turn key into int
2. Mod int into array index
3. Use array to store data
4. Deal w/ collisions

Linear Probing: on collision, just use next free box

3 things per bucket

- K key
- V value
- bool inUse



insert(1, "hi")
insert(6, "hello")
insert(3, "bye")
insert(11, "test")
remove(3)

Linear probing Hash Table

array of triples K, V, inUse-bool

```
insert(K key, V value):
    h ← hash(key)
    i ← h % capacity
    while array[i].inUse:
        i++
        if (i ≥ capacity)
            i ← 0
    array[i].key ← key
    array[i].value ← value
    array[i].inUse ← true
```

```
get(K key)
    h ← hash(key)
    i ← h % capacity
    while array[i].inUse:
        if array[i].key == key
            return array[i].value
        i++
        if (i ≥ capacity)
            i ← 0
    throw exception
```

remove(K key)

figure out start index i

look for matching k

remove matching key

as long as next box has key does not hash to its idx

move that box left