1. Amortization
2. Heapify
3. Data structure: hash table

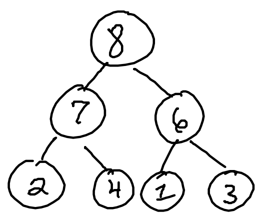
```
Function PosNums(n):          <- WC O(n)
    list <- new ArrayList
    For i in 1 to n:
        list.insertLast(i)    <- WC O(n), amortized WC O(1)
    EndFor
EndFunction
```

amortized : average over a sequence of operations

expected : average over all probabilities

Max Heap is a kind of complete binary tree

CBT: one size ⟷ one shape

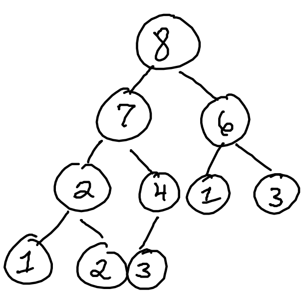


3, 7, 6, 2, 4, 1, 8

```
Function BadHeapify(nums):
    heap <- new MaxHeap
    For each num in nums:
        heap.insert(num, num)   <- takes O(log n)   } runs n times
    EndFor
EndFunction
```

```
Function Heapify(tree):
    For index from end of tree to start:
        BubbleDown(tree, index)
    EndFor
EndFunction
```

} O(n)



$$1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + 3 \cdot \frac{n}{8} + 4 \cdot \frac{n}{16} + \cdots +$$

$$= n \sum_{i=1}^{\log_2 n} \frac{i}{2^i} \leq n \sum_{i=1}^{\infty} \frac{i}{2^i}$$ — constant?

Most bubble downs are small

$$\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots$$

$$= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots$$

$$+ \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots$$

$$+ \frac{1}{8} + \frac{1}{16} + \cdots$$

$$1 + \frac{1}{2} + \frac{1}{4} + \cdots = 2$$

Dictionary ADT:

	BST	AVL	Hash Table
V get (K key)	$O(n)$	$O(\log n)$	average $O(1)^*$
void insert (K key, V value)	$O(n)$	$O(\log n)$	average $O(1)^*$
void update (K key, V value)	$O(n)$	$O(\log n)$	average $O(1)^*$
void remove (K key)	$O(n)$	$O(\log n)$	average $O(1)^*$

Hash Table

1. Assume all keys are integers.
2. Assume all keys are non-negative.
3. Assume all keys are less than 10.
4. Assume no collisions

$5 \% 10 = 5$

$8 \% 10 = 8$

$12 \% 10 = 2$

$27 \% 10 = 7$

$-1 \% 10 = 9$

Array of size 10
indices match key mod array size
array contains slots w/ key & value & bool

insert(7, "a")
insert(13, "b")
insert (3, "q")
collision
(put this aside)

	0	1	2	3	4	5	6	7	8	9
K				13				7		
V				"b"				"a"		
bool	X	X	X	✓	X	X	X	✓	X	X

```
Function hash(string key):
    Return 0
End Function
```

Bad

Good hash: tends to distribute keys to ints evenly

```
Function hash(string key):
    acc ← 0
    For each character c in key:
        acc ← acc * 31
        acc ← acc + c
    EndFor
    Return acc
EndFunction
```

Good

Non-integer key (e.g. string):
convert it into an integer and then mod
(hash) (might lose information)