Heaps

Tiziana Ligorio

Announcements

Please take a few minutes to fill in the teacher evaluation

You should have received a link invitation to your hunter email address

Smartphone: **www.hunter.cuny.edu/mobilete**

Computer: **www.hunter.cuny.edu/te**

Login using your **Hunter netID**

Thank you!!!

Heap

A **Heap** is a complete binary tree that is either

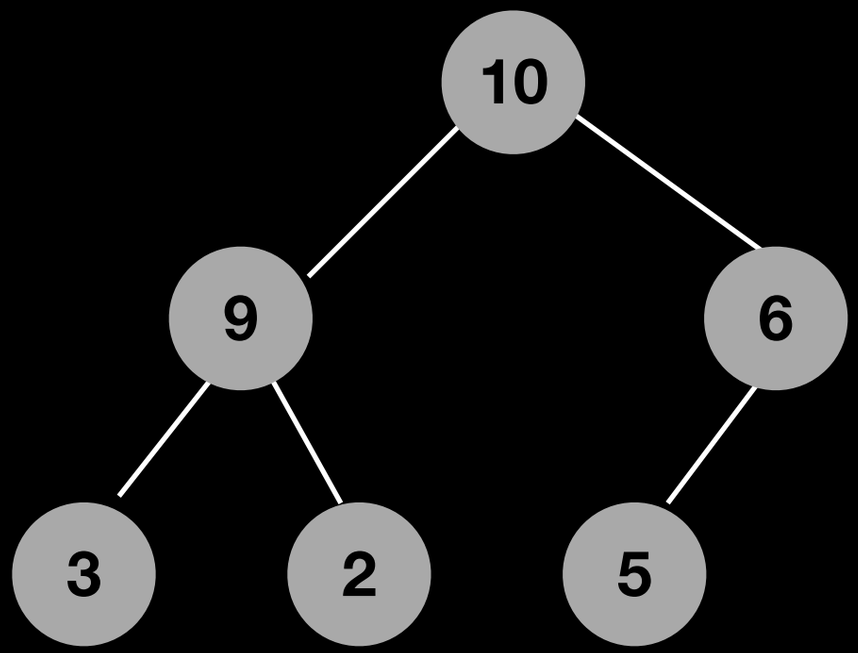
- Empty or
- Its **root** contains a value **≥** (or ≤) both of its **children** and has **heaps as subtrees**

Heap

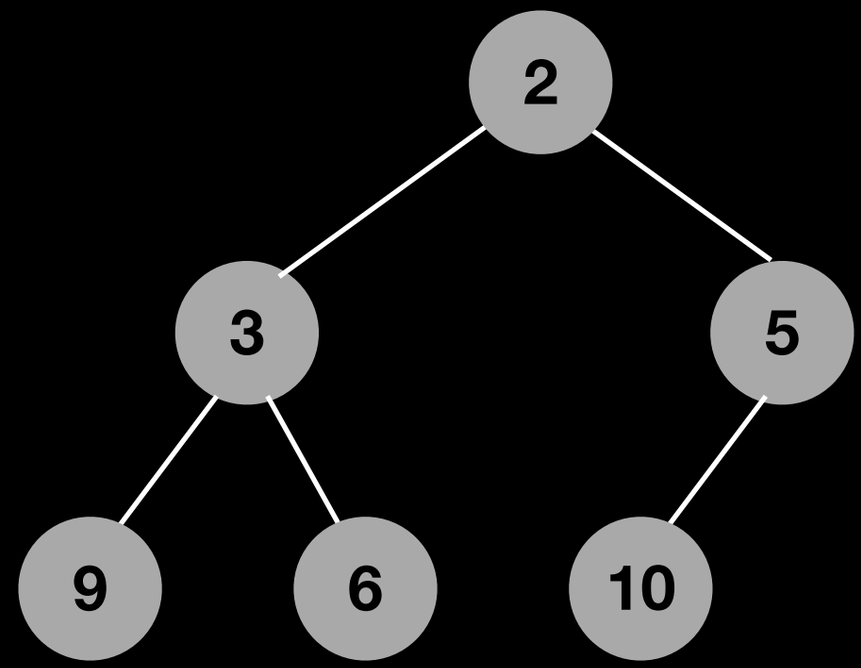
A special binary tree:

- Ordered in a weaker sense
- Always a **complete** binary tree

MaxHeap



MinHeap



Implementation

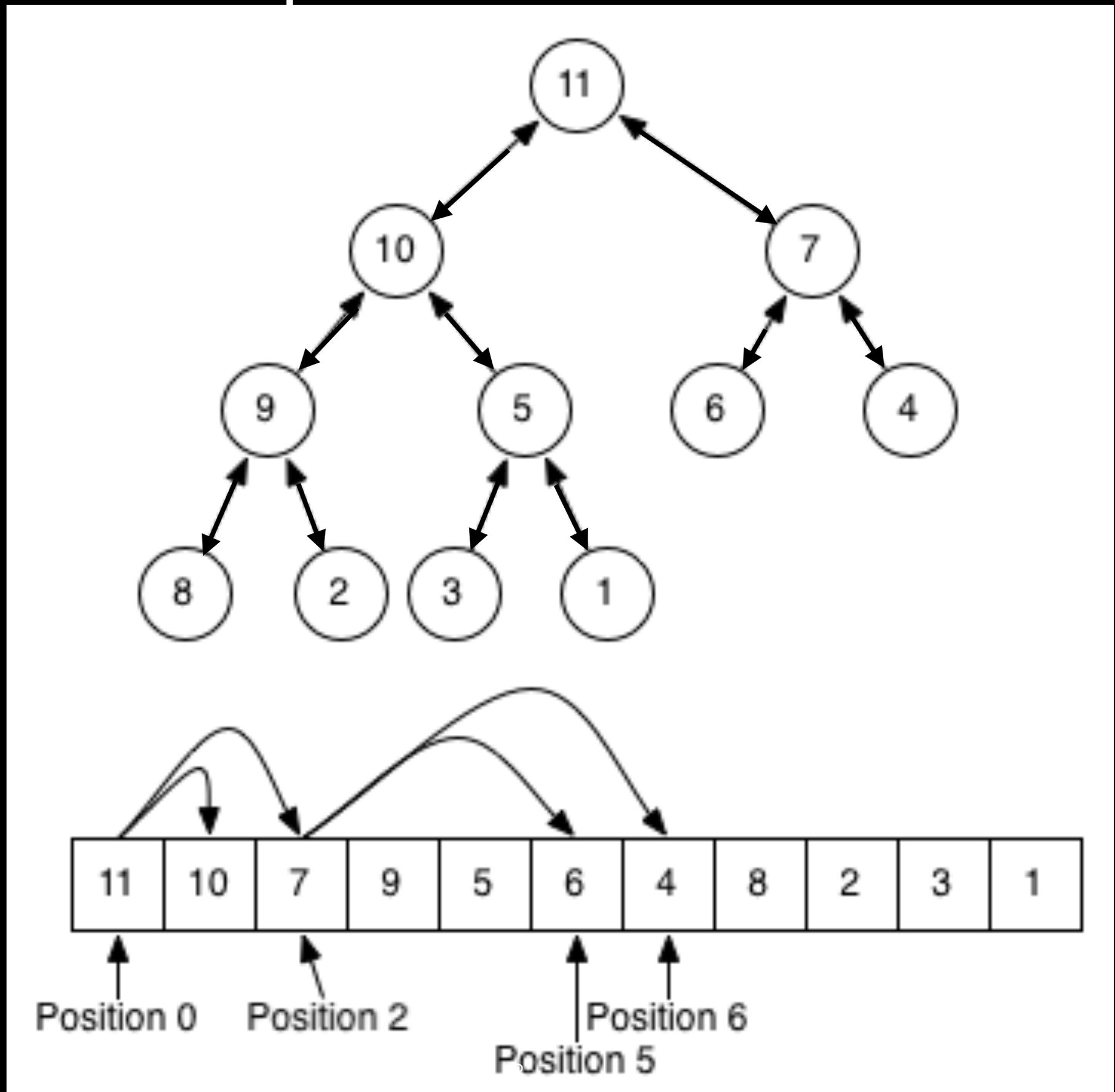
How would you implement it???

Implementation

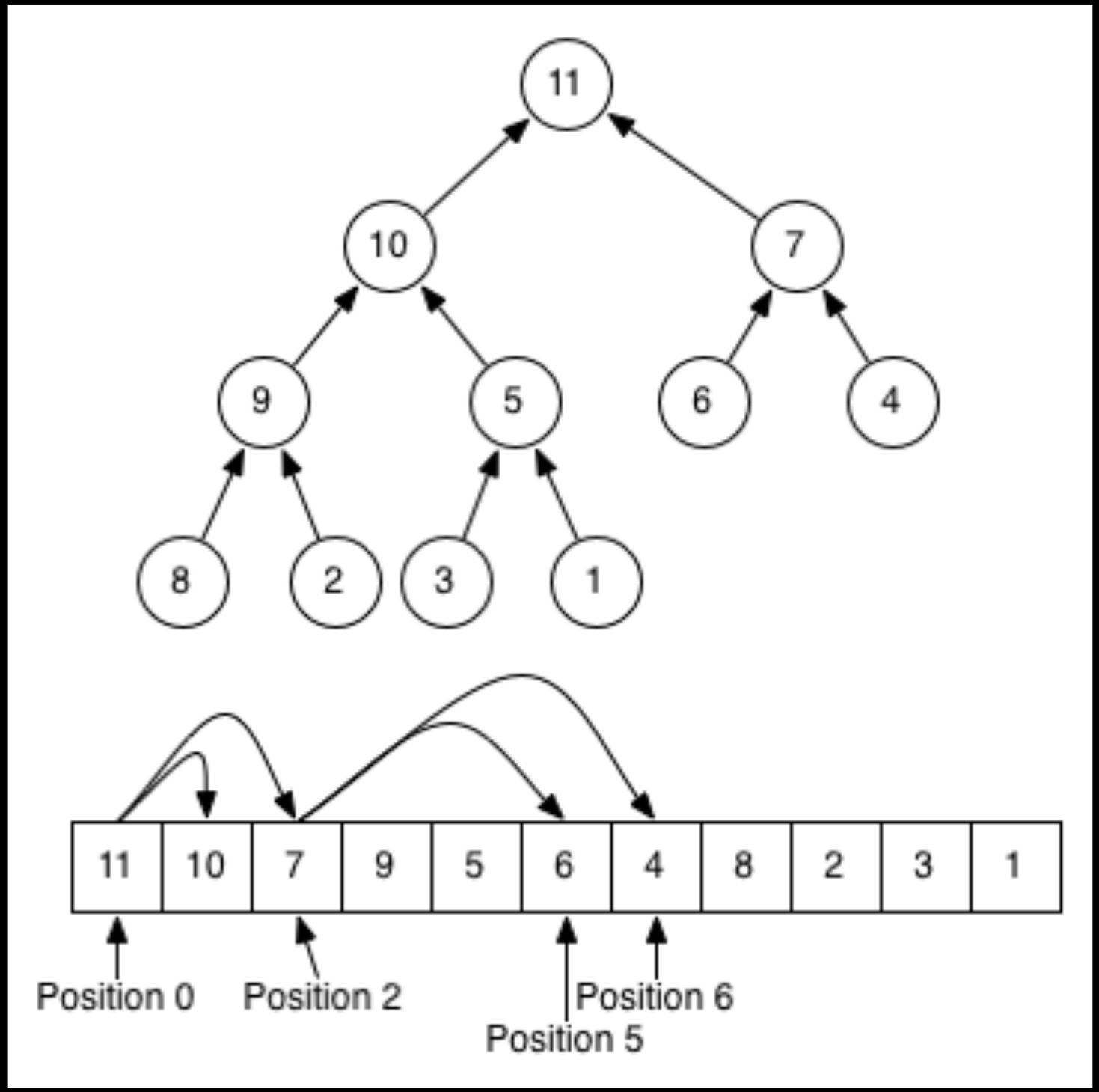
How would you implement it???

Insight: it is **always complete**

Implementation



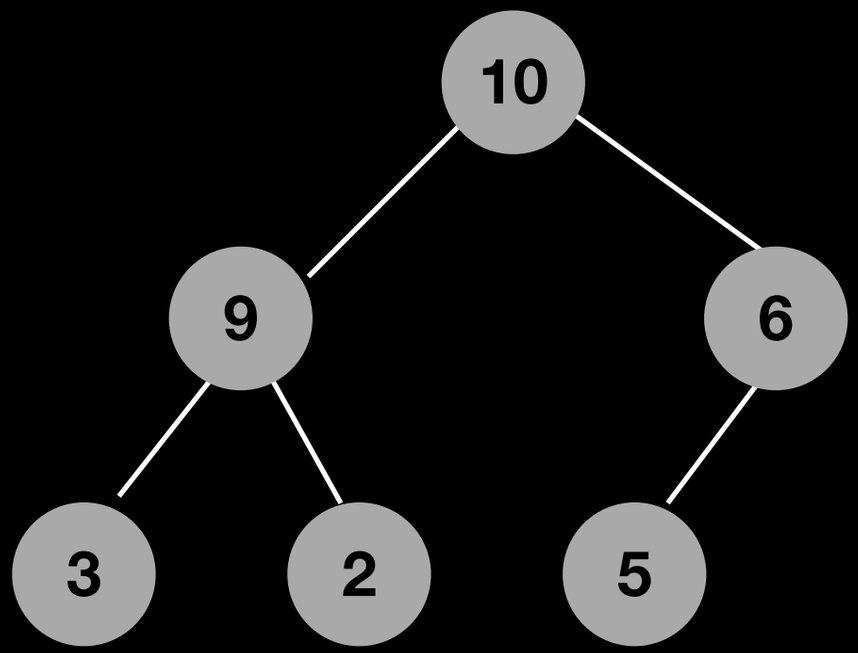
Implementation



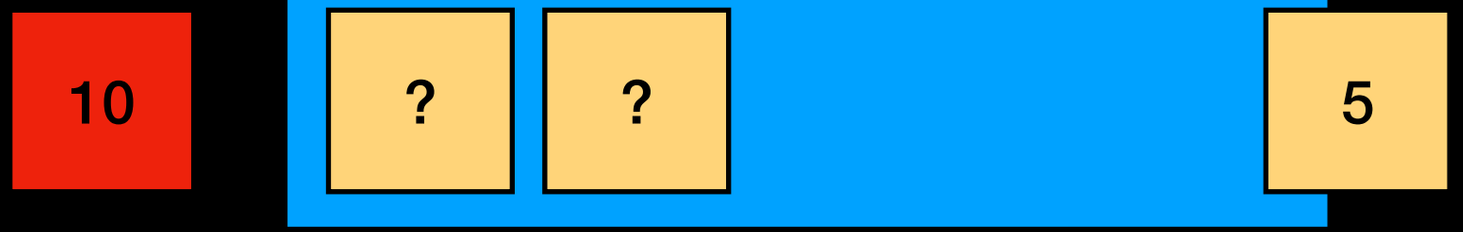

```
root_ = items_[0]
items_[i] left_child = items_[2 * i + 1]
items_[i] right_child = items_[ 2 * i + 2]
items_[i] parent = items_[(i-1)//2]
```

floor division

MaxHeap



Priority Queue



Retrieve

Can only retrieve max/min item

Stored at root

O(1)

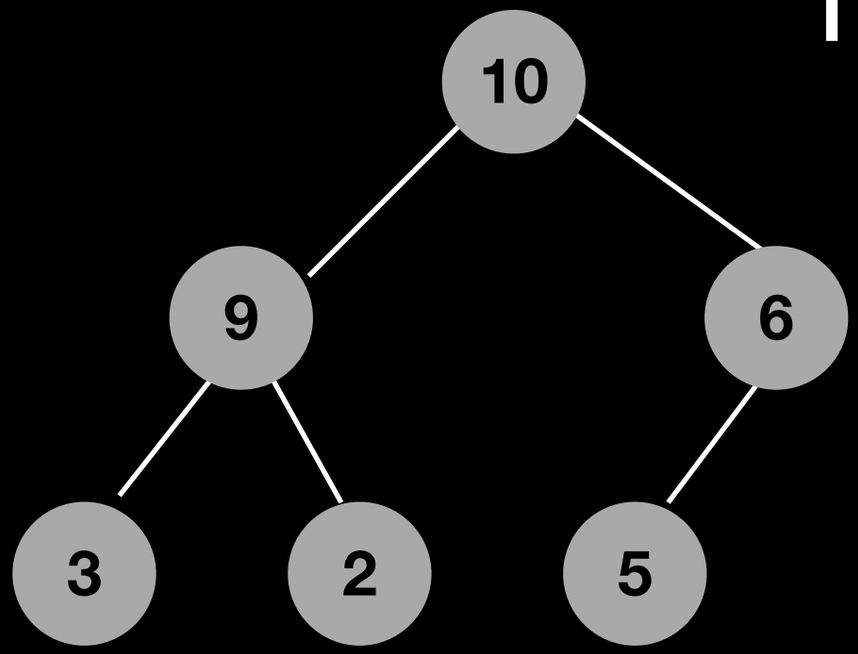
Remove

Remove max/min item (the root)

Must retain Heap

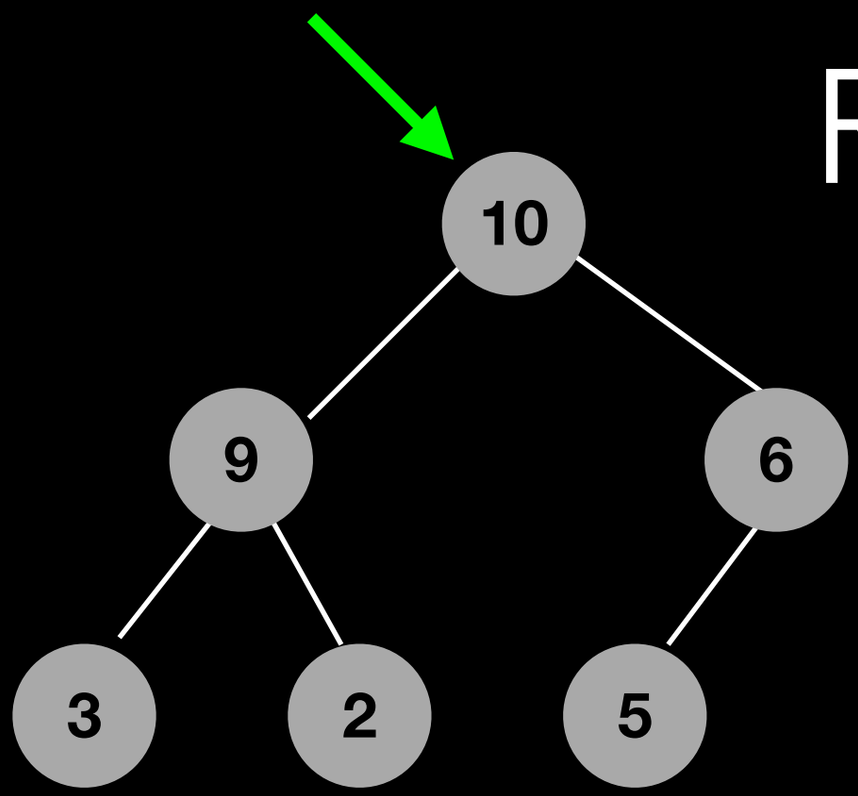
- Heap ordering property
- Complete

Remove



What element do we remove?

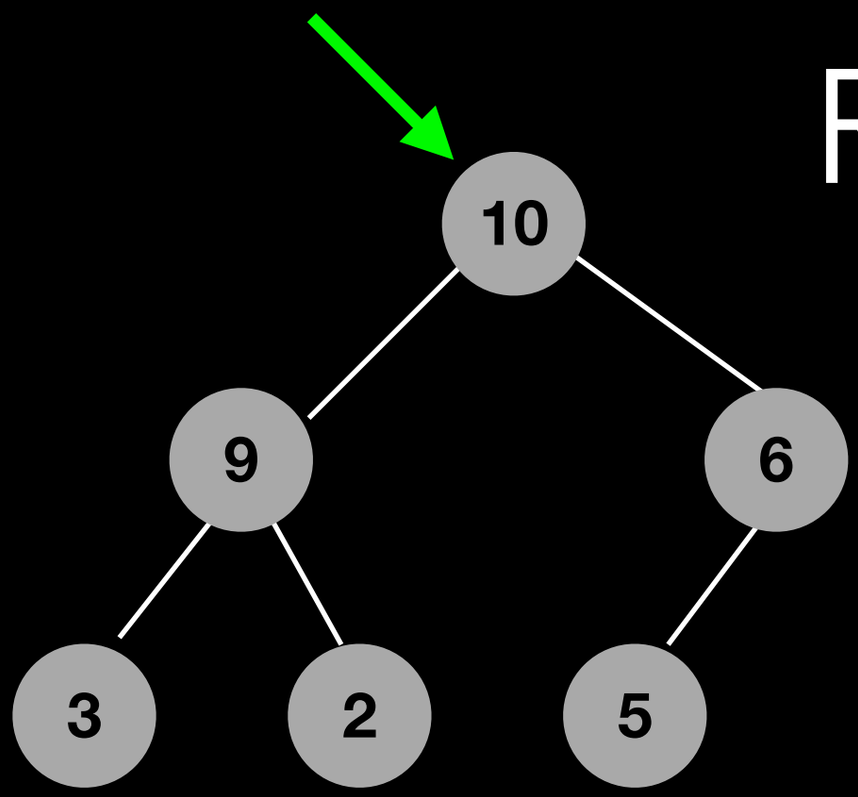
Remove



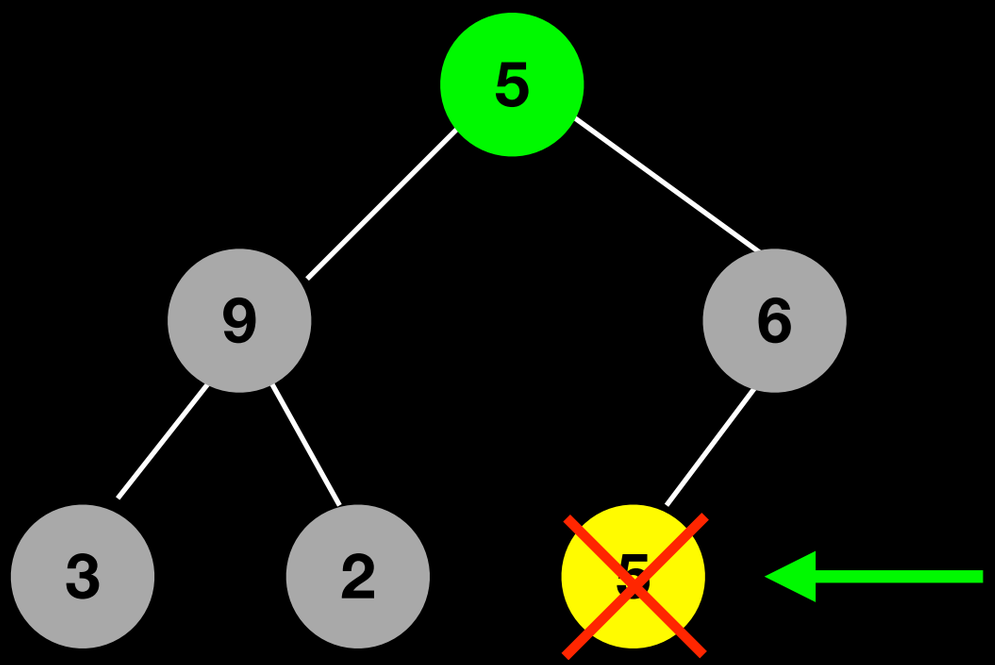
What element do we remove?

What node do we remove?

Remove

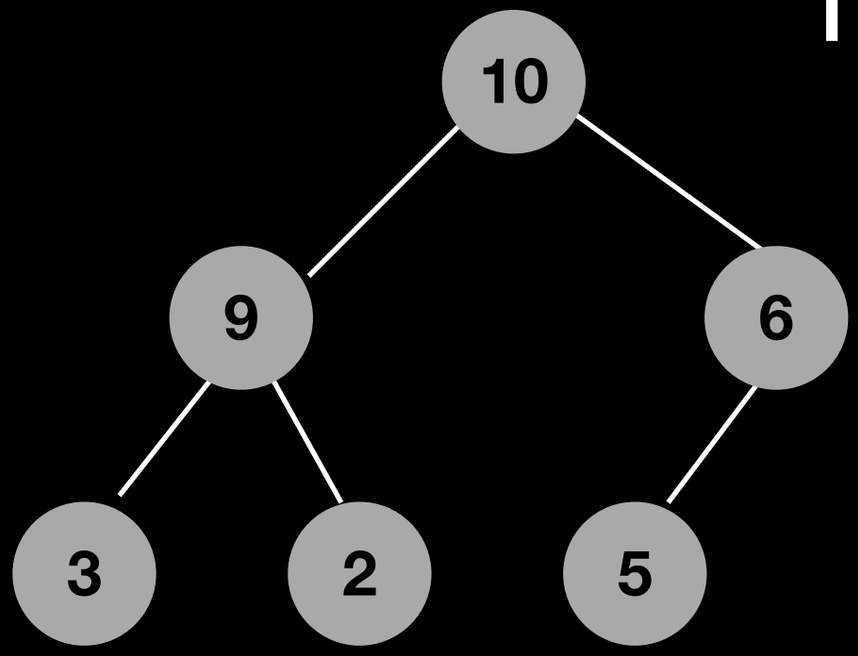


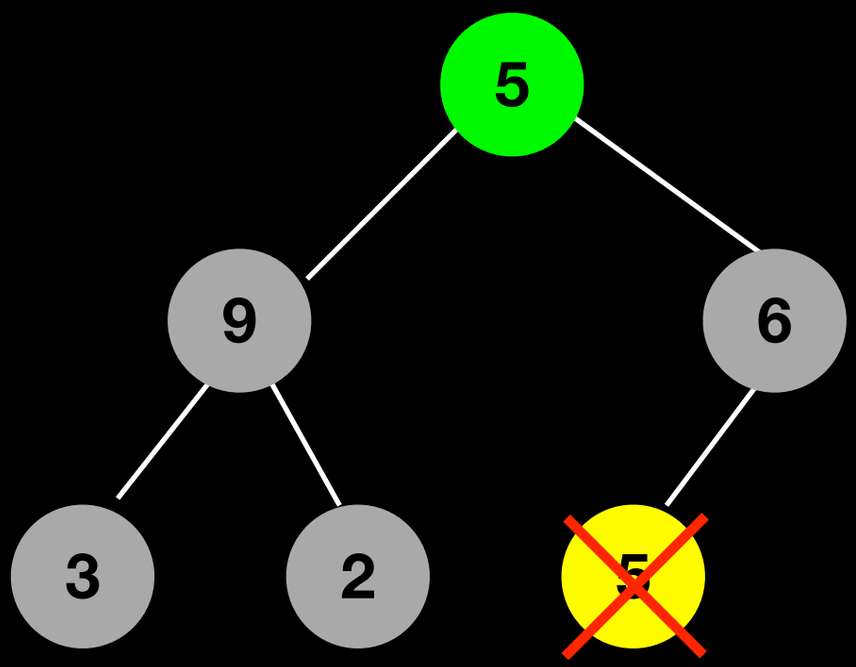
What element do we remove?



Remove this node form complete tree

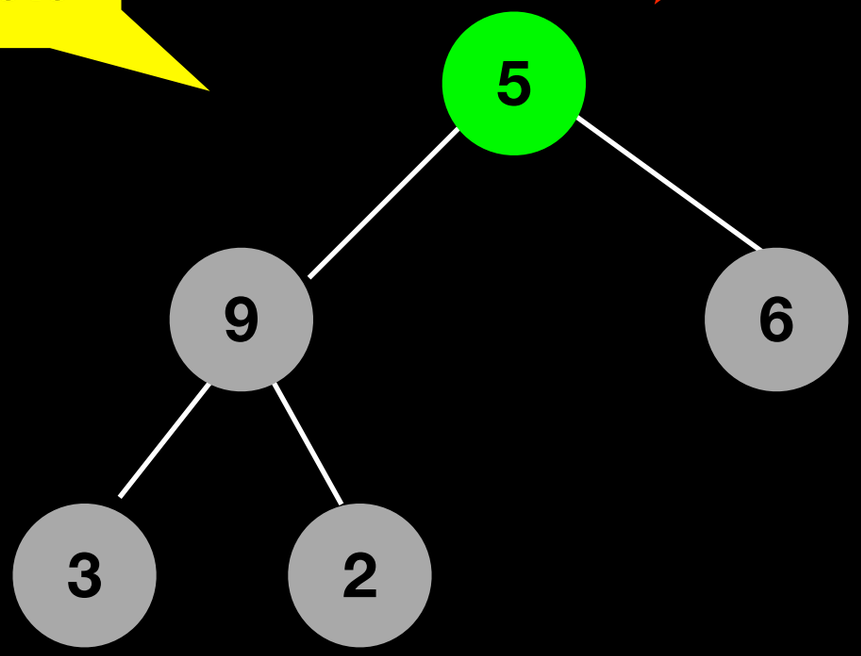
Remove





Complete

Not Heap

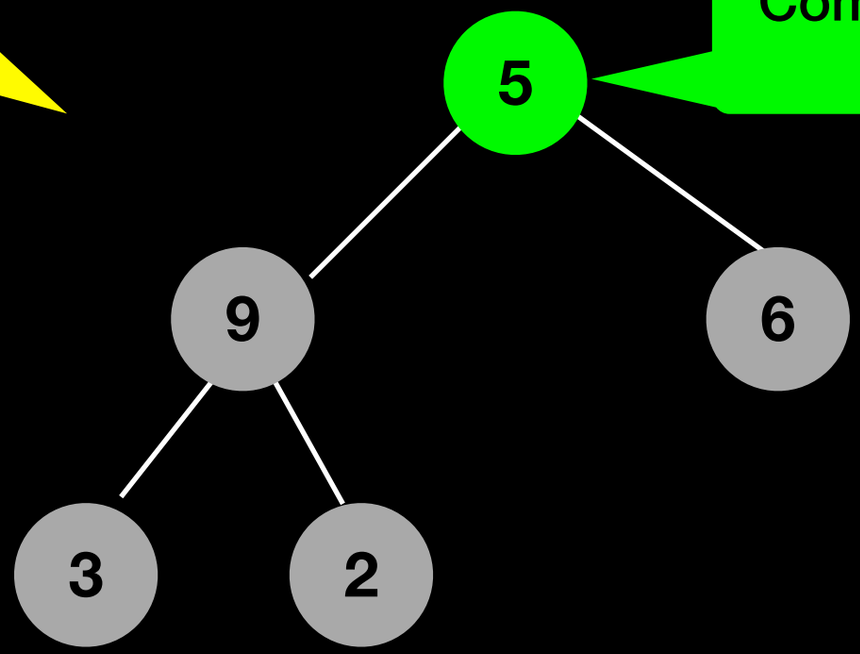


heapRebuild

Remove

Complete

Trickle down by swapping
Compare left and right



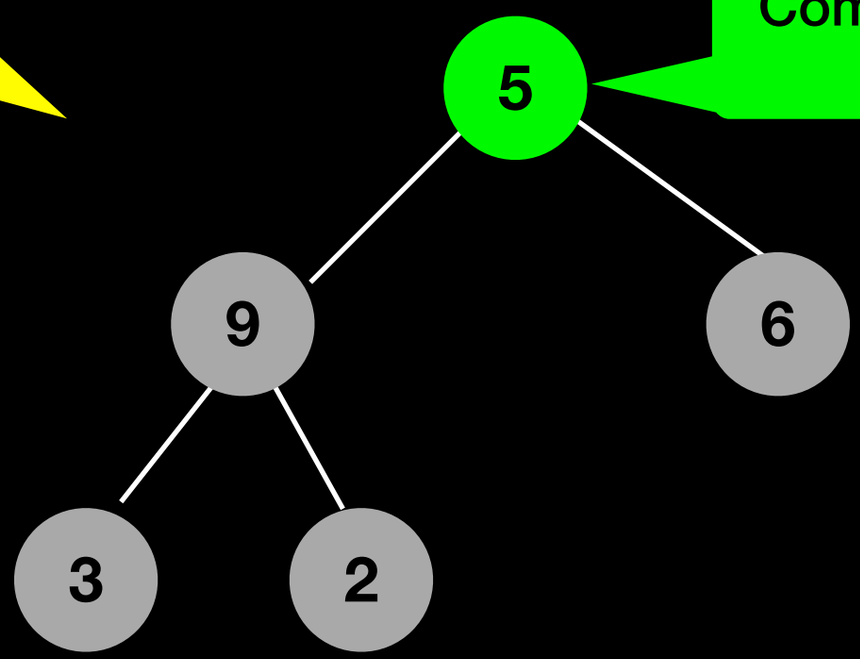
heapRebuild

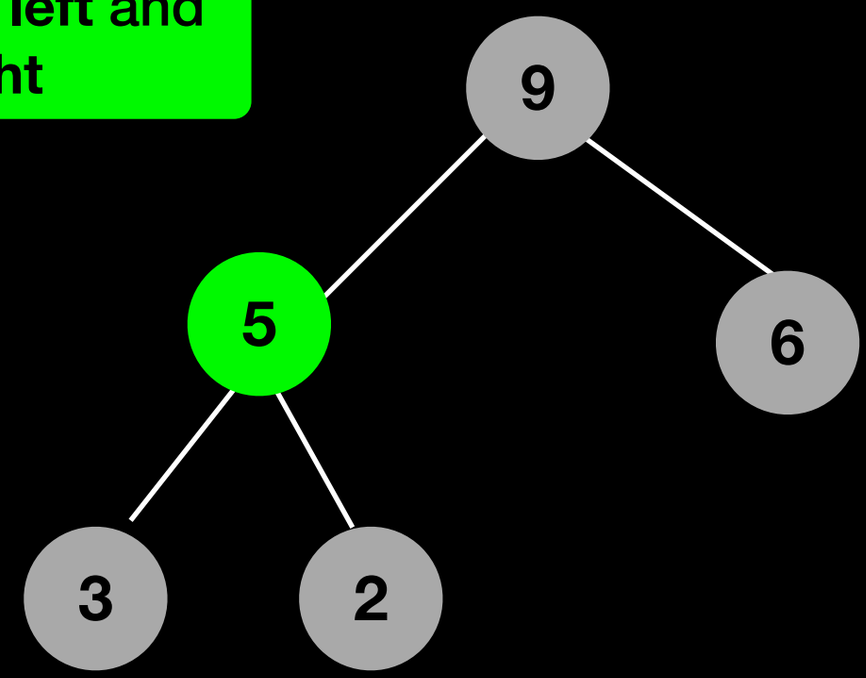
Remove

Complete

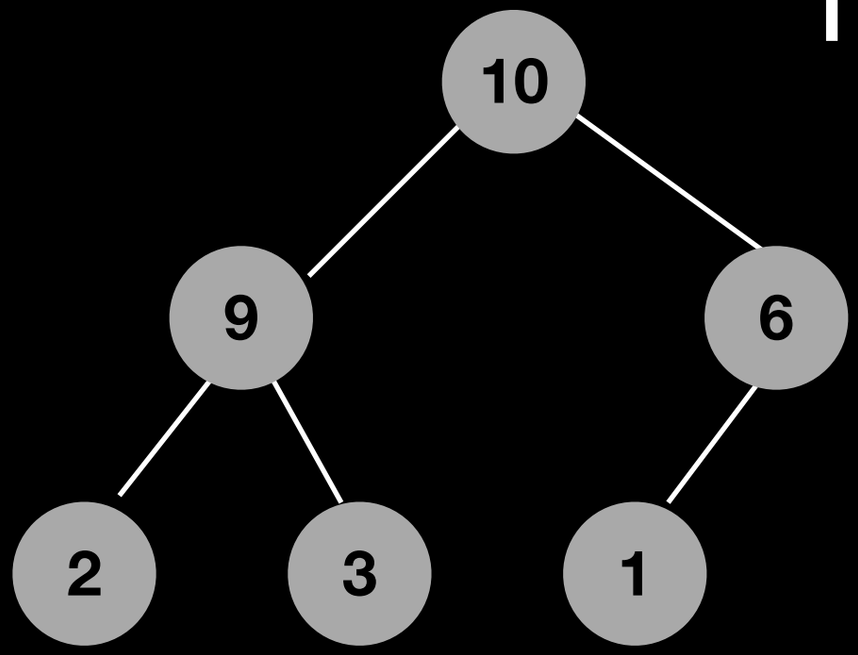
Trickle down by swapping
Compare left and right

Heap!

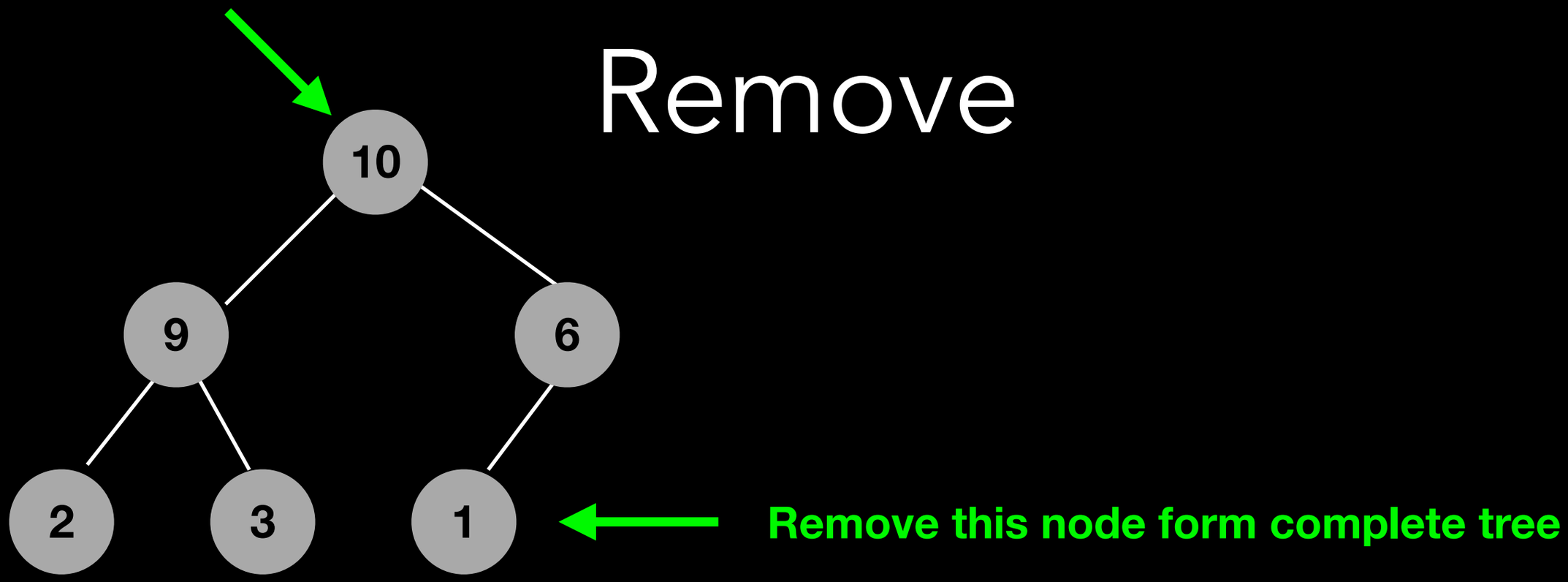




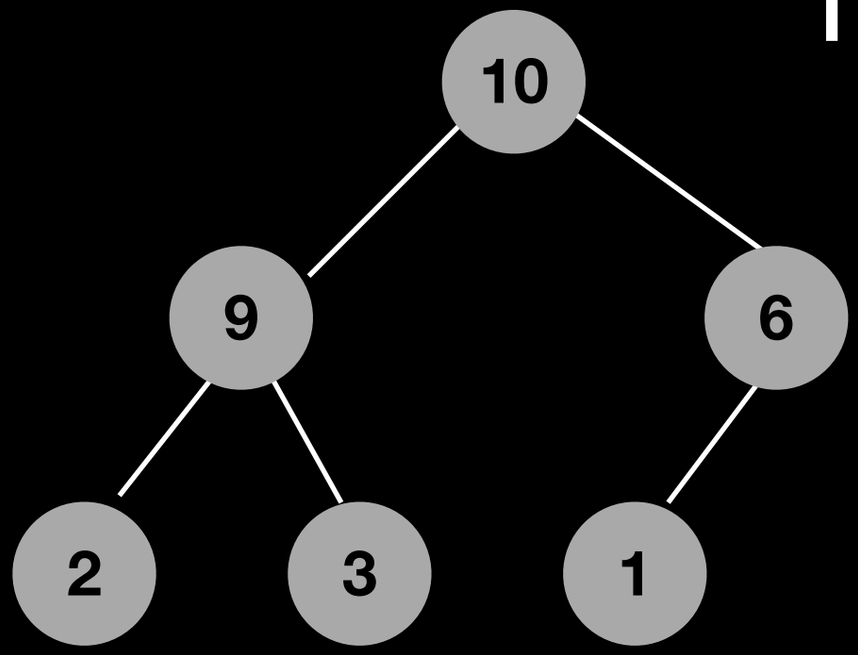
Remove

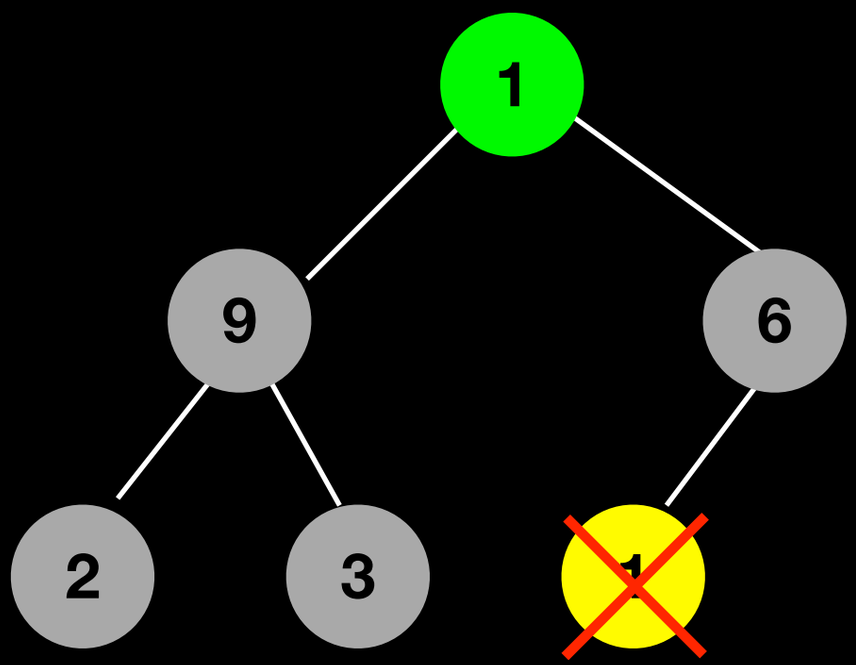


Remove

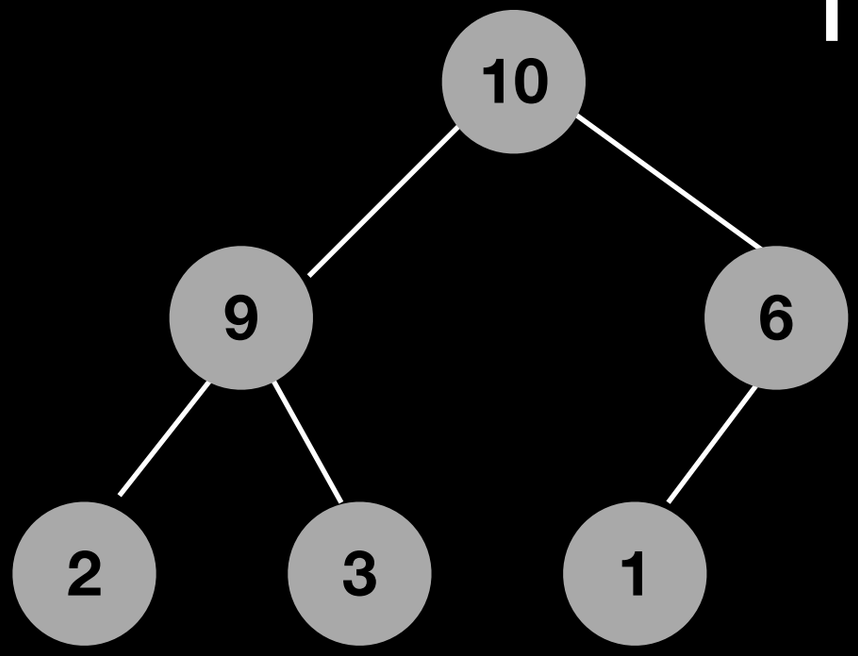


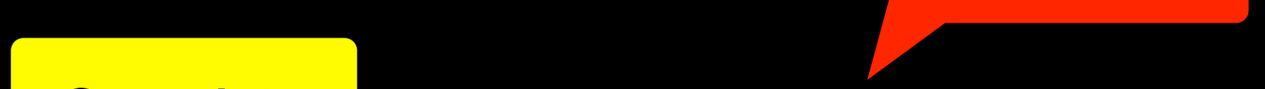
Remove

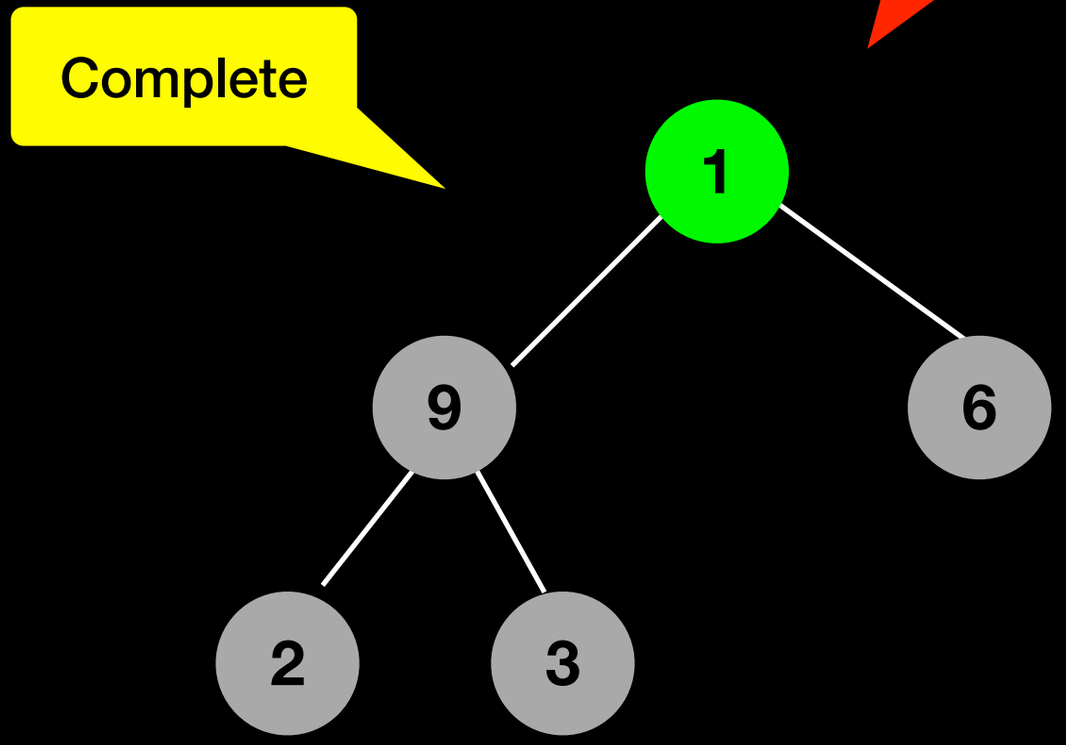


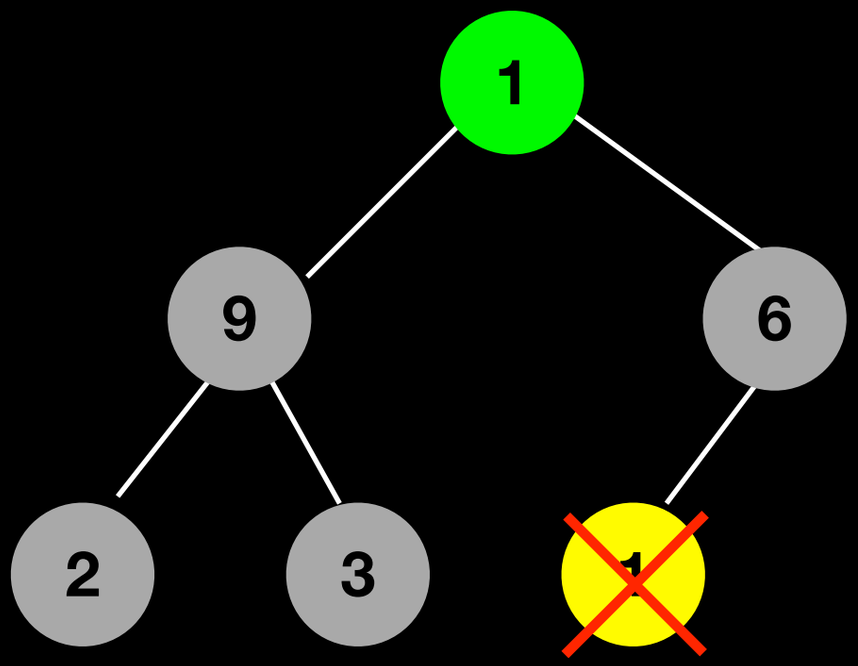


Remove







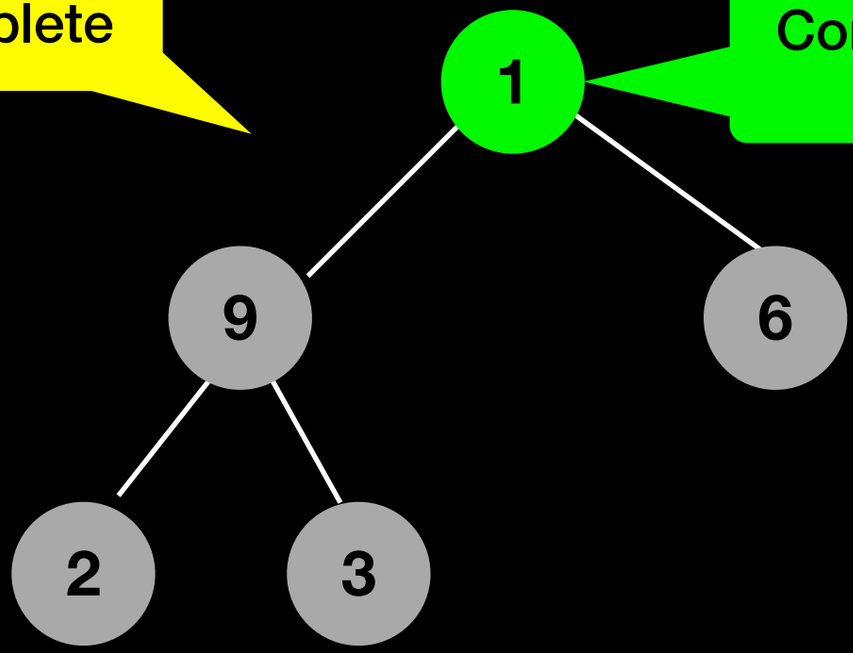


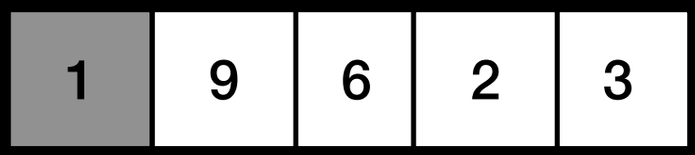
heapRebuild

Remove

Complete

Trickle down by swapping
Compare left and right



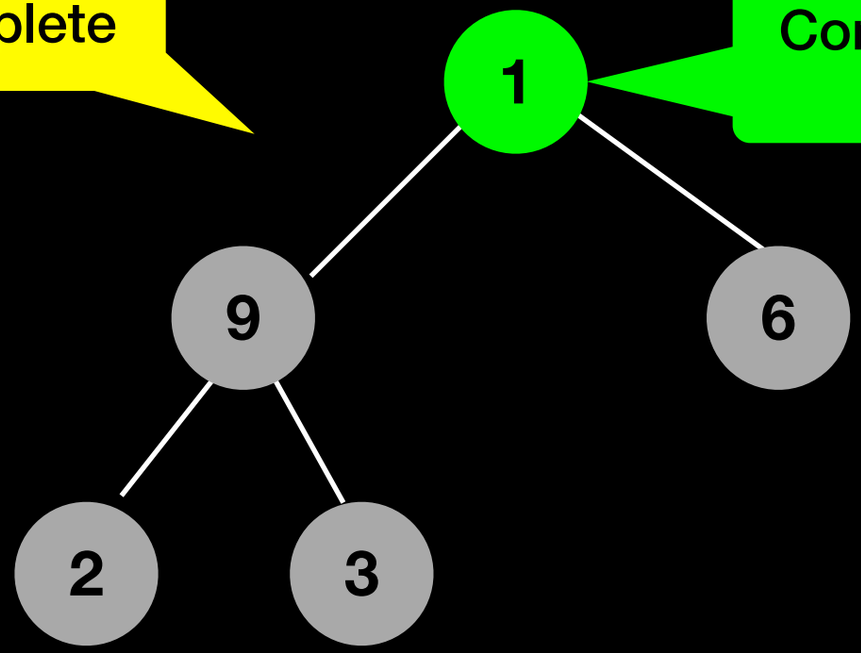



```
items_[i] left_child = items_[2 * i + 1]
items_[i] right_child = items_[ 2 * i + 2]
```

heapRebuild

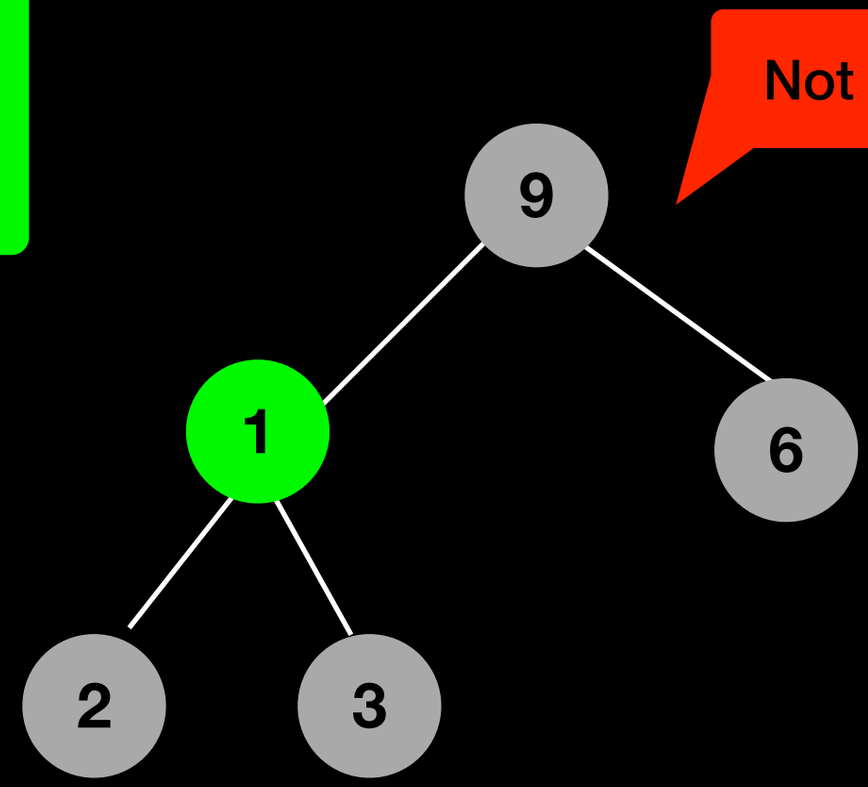
Remove

Complete



Trickle down by swapping
Compare left and right

Not Heap



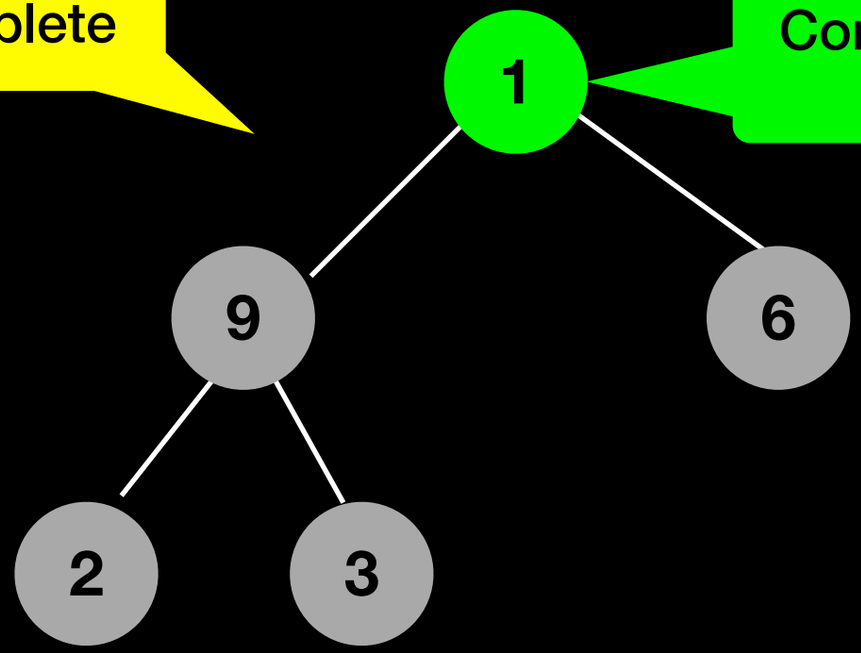
1	9	6	2	3
1	9	6	2	3

```
items_[i] left_child = items_[2 * i + 1]
items_[i] right_child = items_[ 2 * i + 2]
```

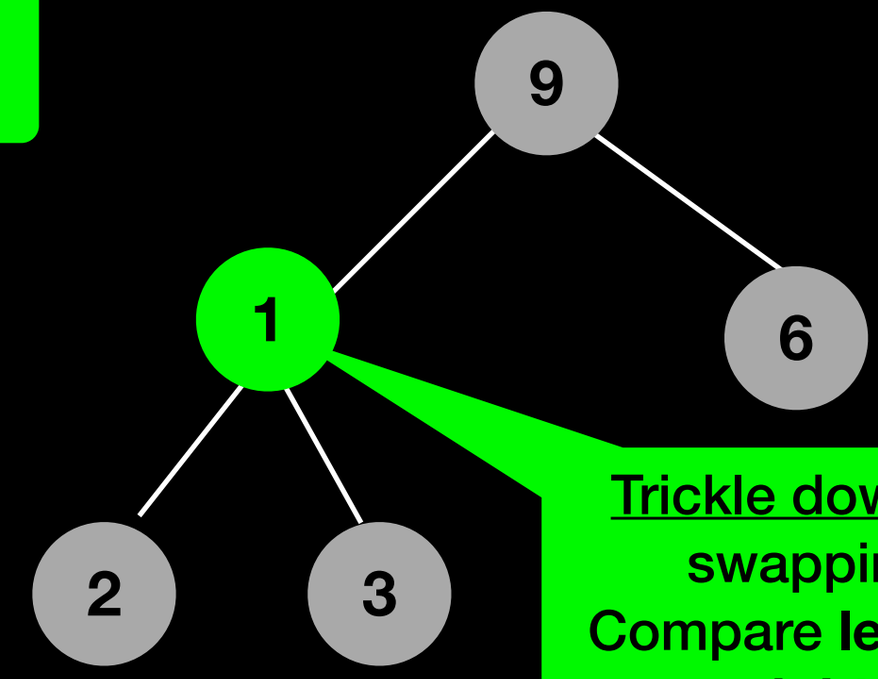
heapRebuild

Remove

Complete



Trickle down by swapping
Compare left and right



Trickle down by swapping
Compare left and right

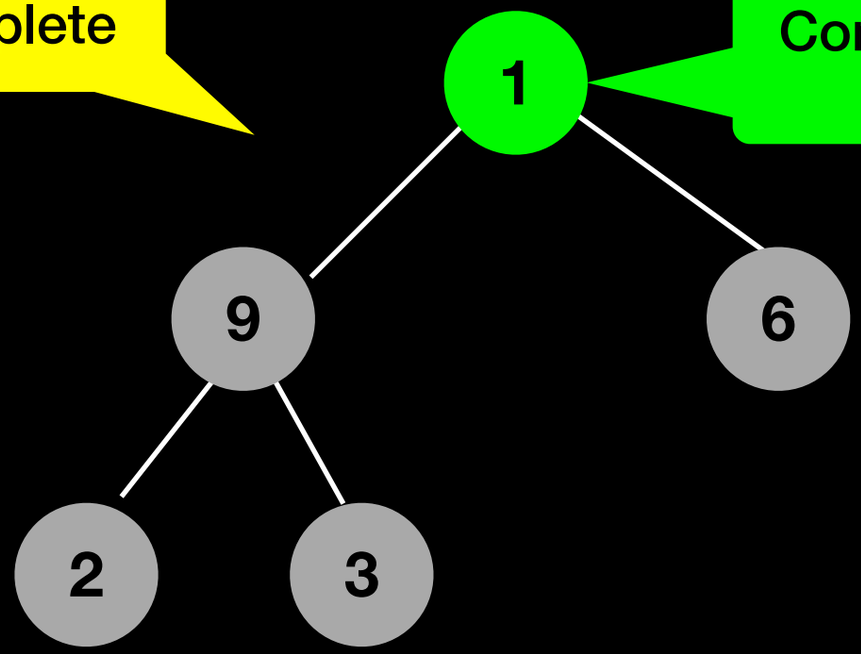
1	9	6	2	3
1	9	6	2	3

```
items_[i] left_child = items_[2 * i + 1]
items_[i] right_child = items_[ 2 * i + 2]
```

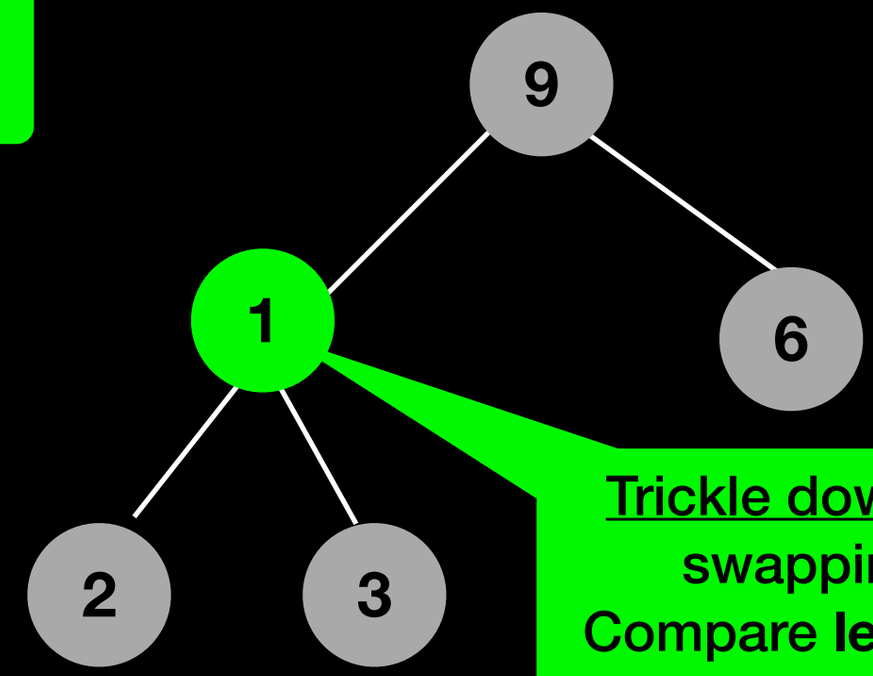
heapRebuild

Remove

Complete



Trickle down by swapping
Compare left and right



Trickle down by swapping
Compare left and right

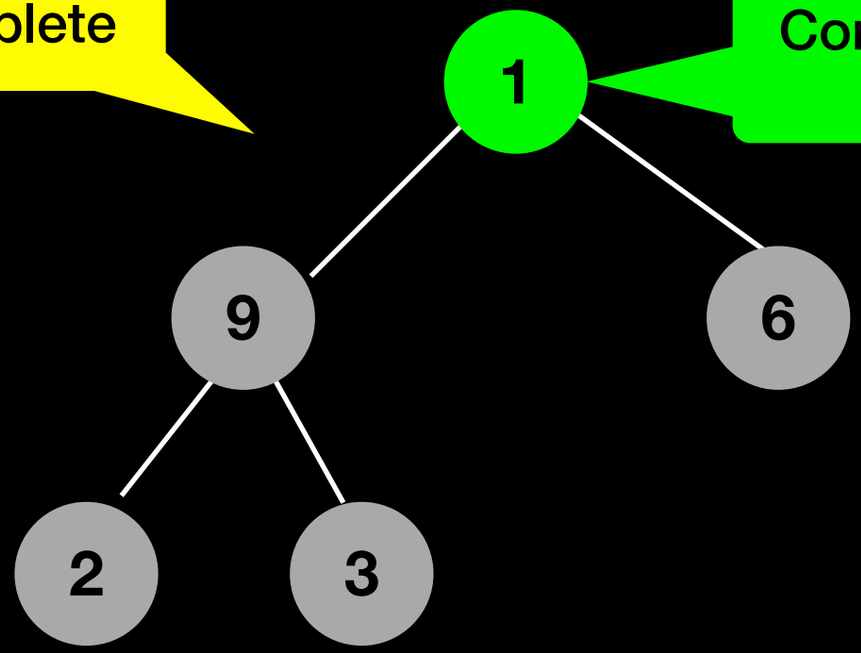
1	9	6	2	3
1	9	6	2	3
9	1	6	2	3

```
items_[i] left_child = items_[2 * i + 1]
items_[i] right_child = items_[ 2 * i + 2]
```

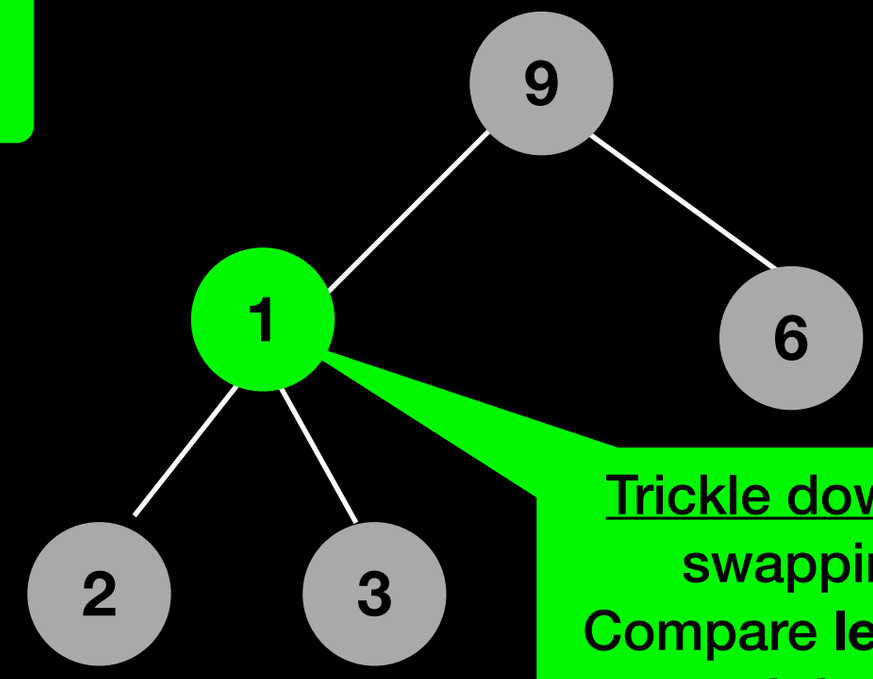
heapRebuild

Remove

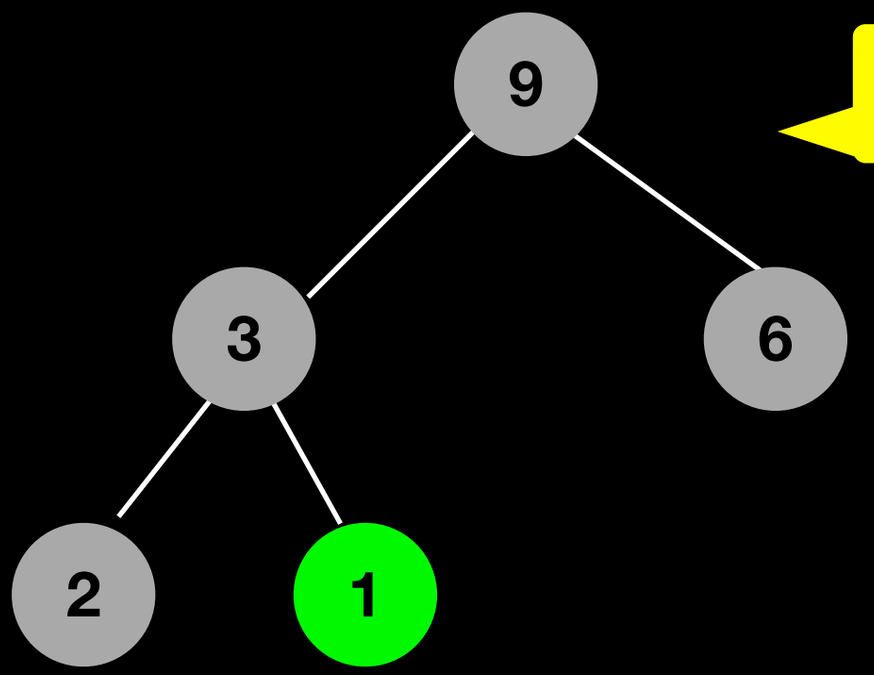
Complete



Trickle down by swapping
Compare left and right



Trickle down by swapping
Compare left and right



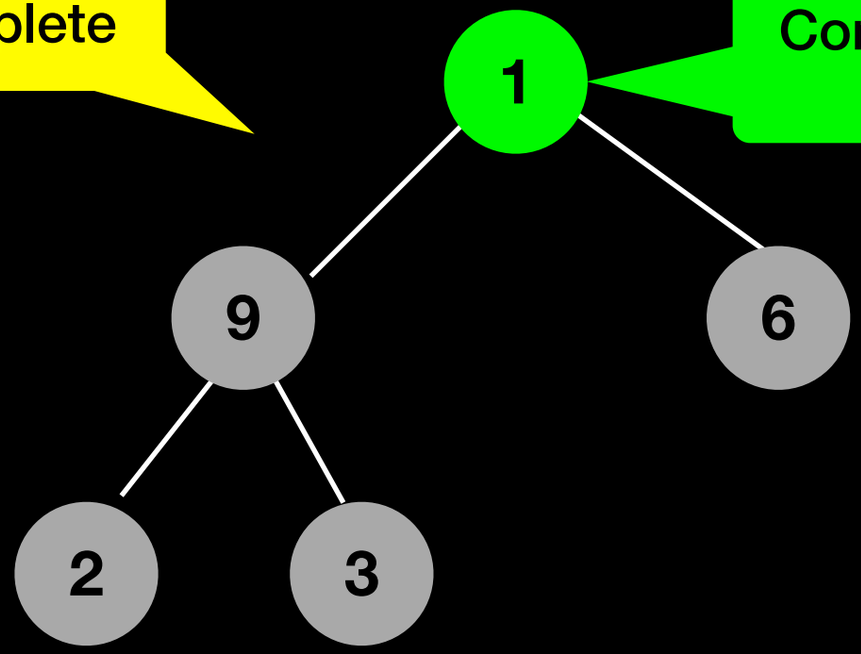
Heap!

1	9	6	2	3
1	9	6	2	3
9	1	6	2	3
9	3	6	2	1

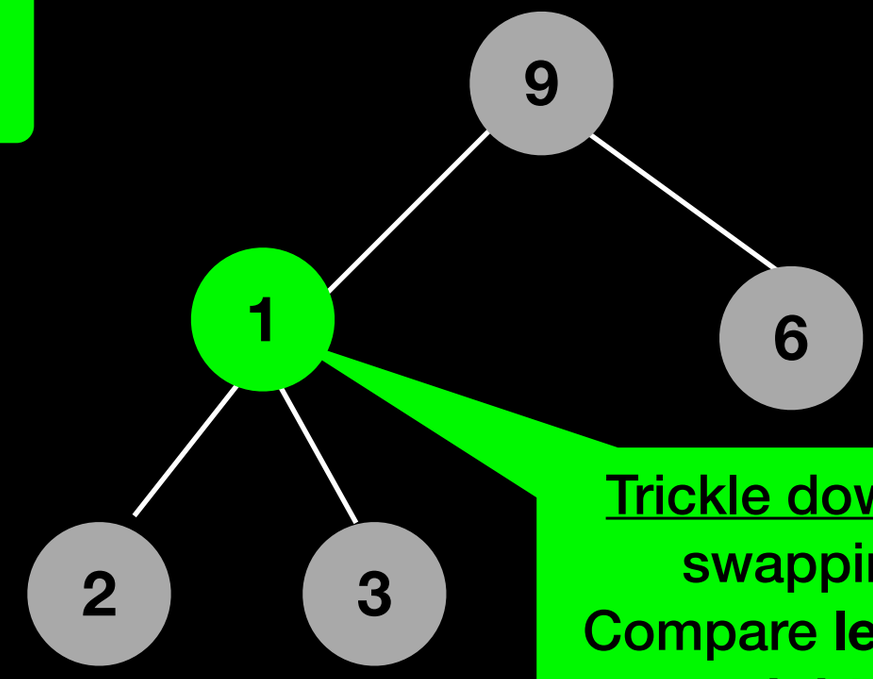
heapRebuild

Remove

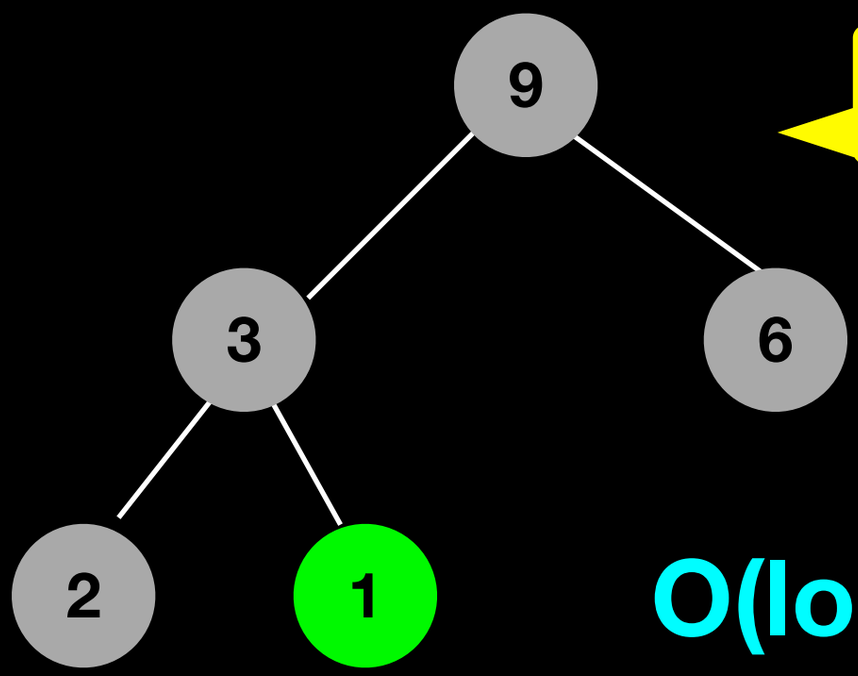
Complete



Trickle down by swapping
Compare left and right



Trickle down by swapping
Compare left and right

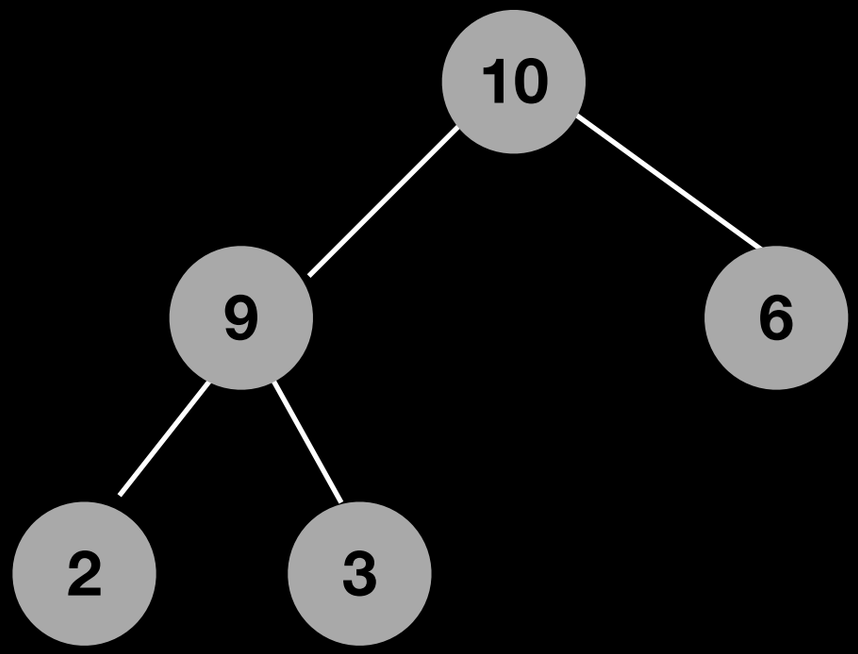


Heap!

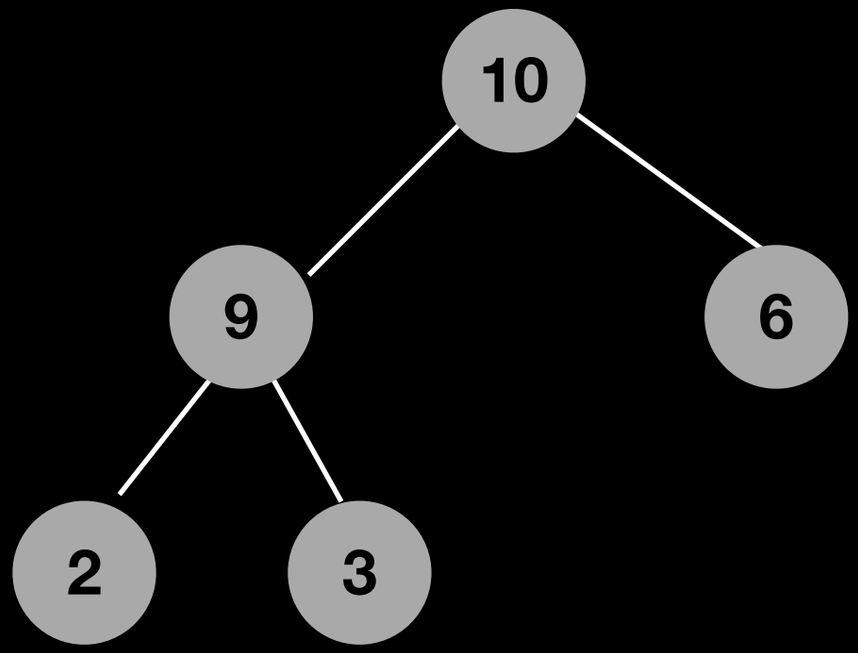
1	9	6	2	3
1	9	6	2	3
9	1	6	2	3
9	3	6	2	1

O(logn)

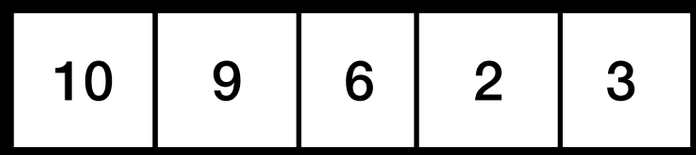
Add



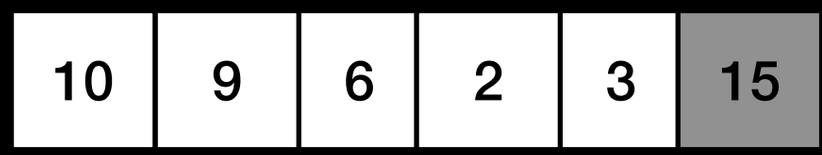
Add



15 **Where do we add?**



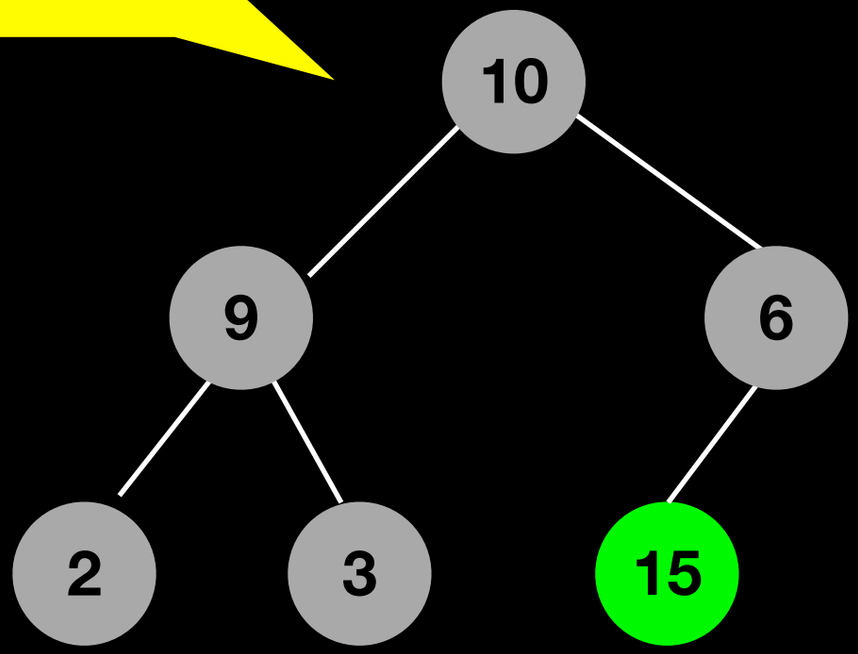
Add



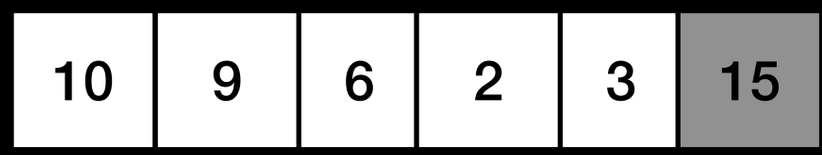

```
items_[i] left_child = items_[2 * i + 1]
```

Add

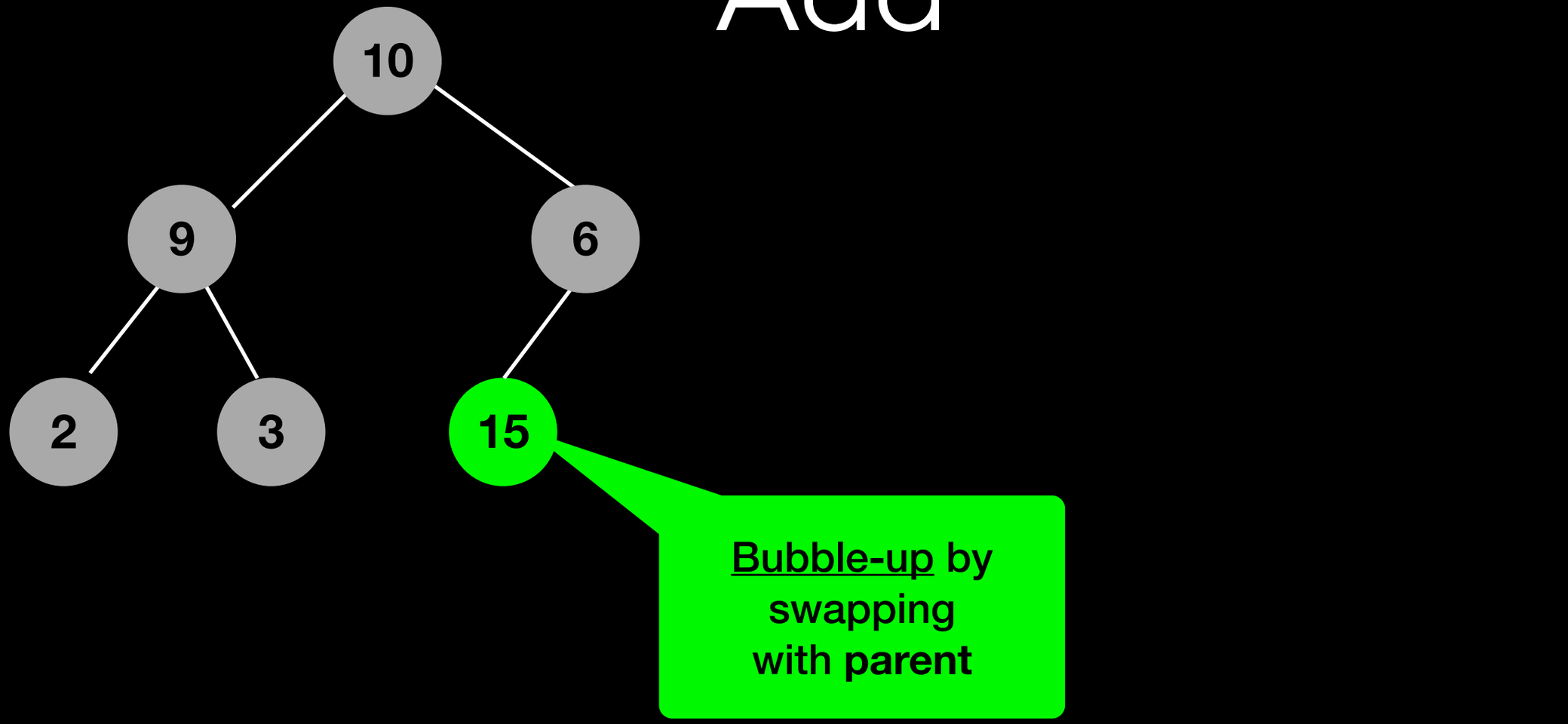
Complete

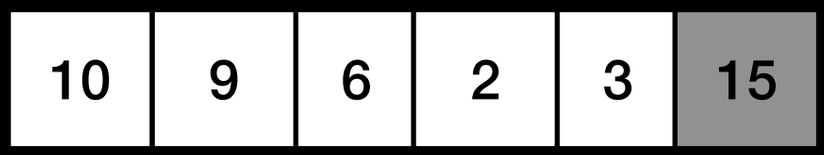


Not Heap



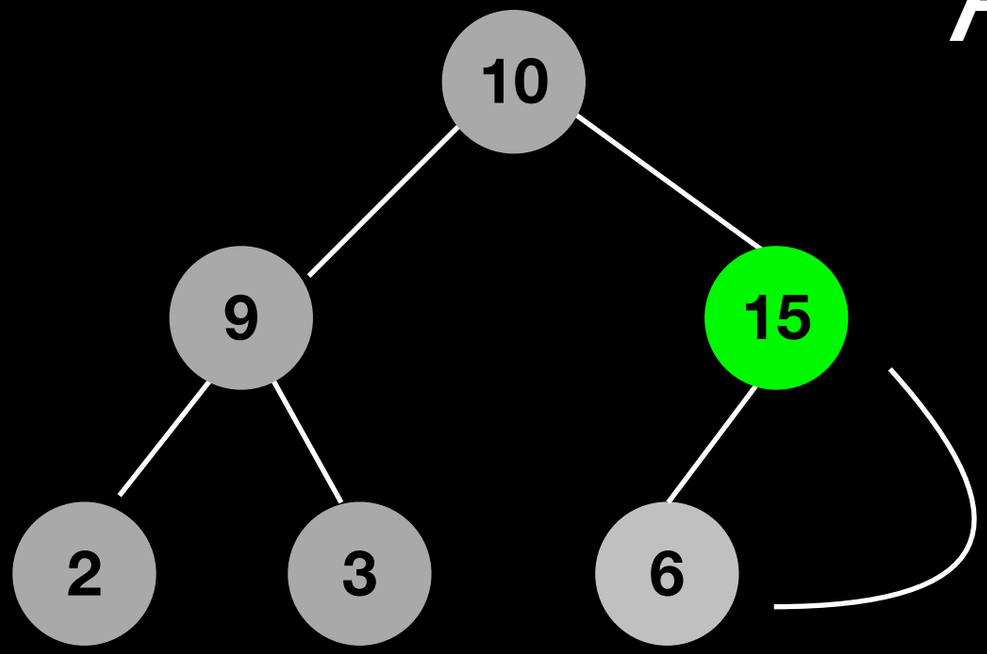
Add






```
items_[i] parent = items_[(i-1)//2]
```

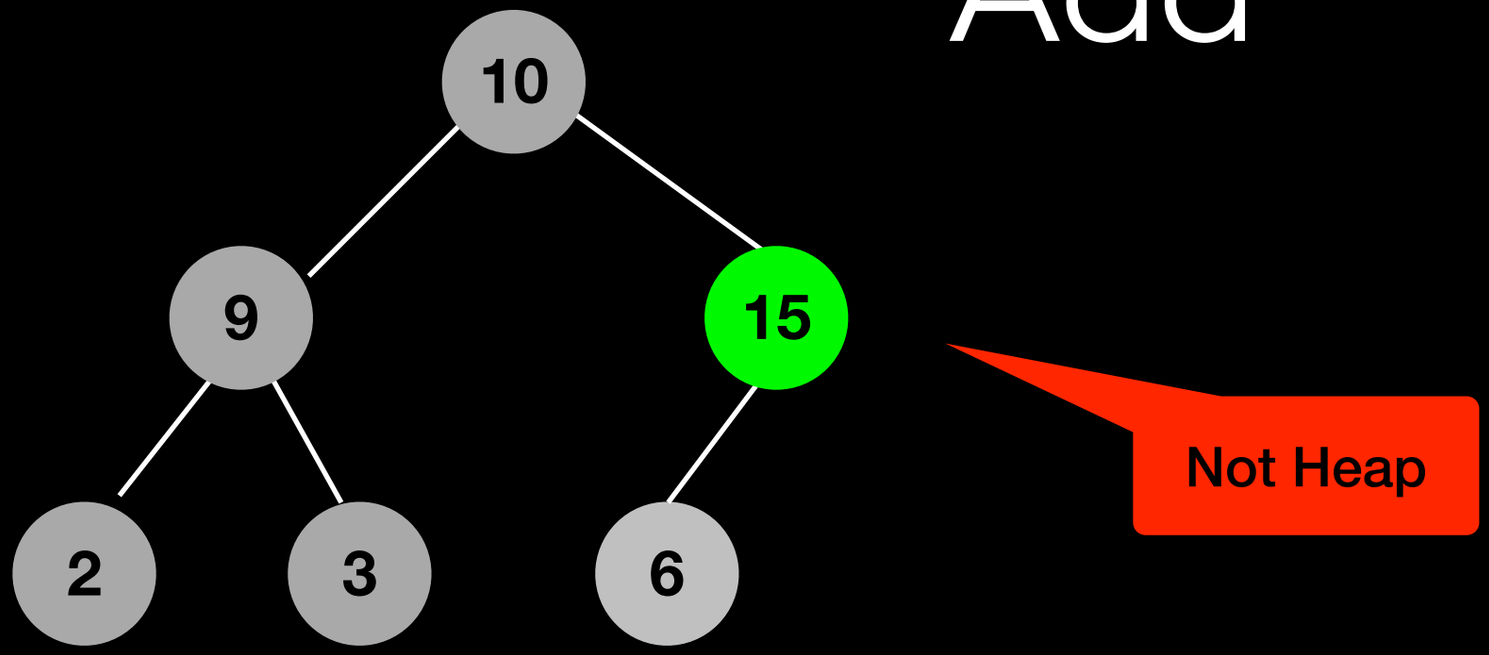
Add



10	9	6	2	3	15
10	9	15	2	3	6

```
items_[i] parent = items_[(i-1)//2]
```

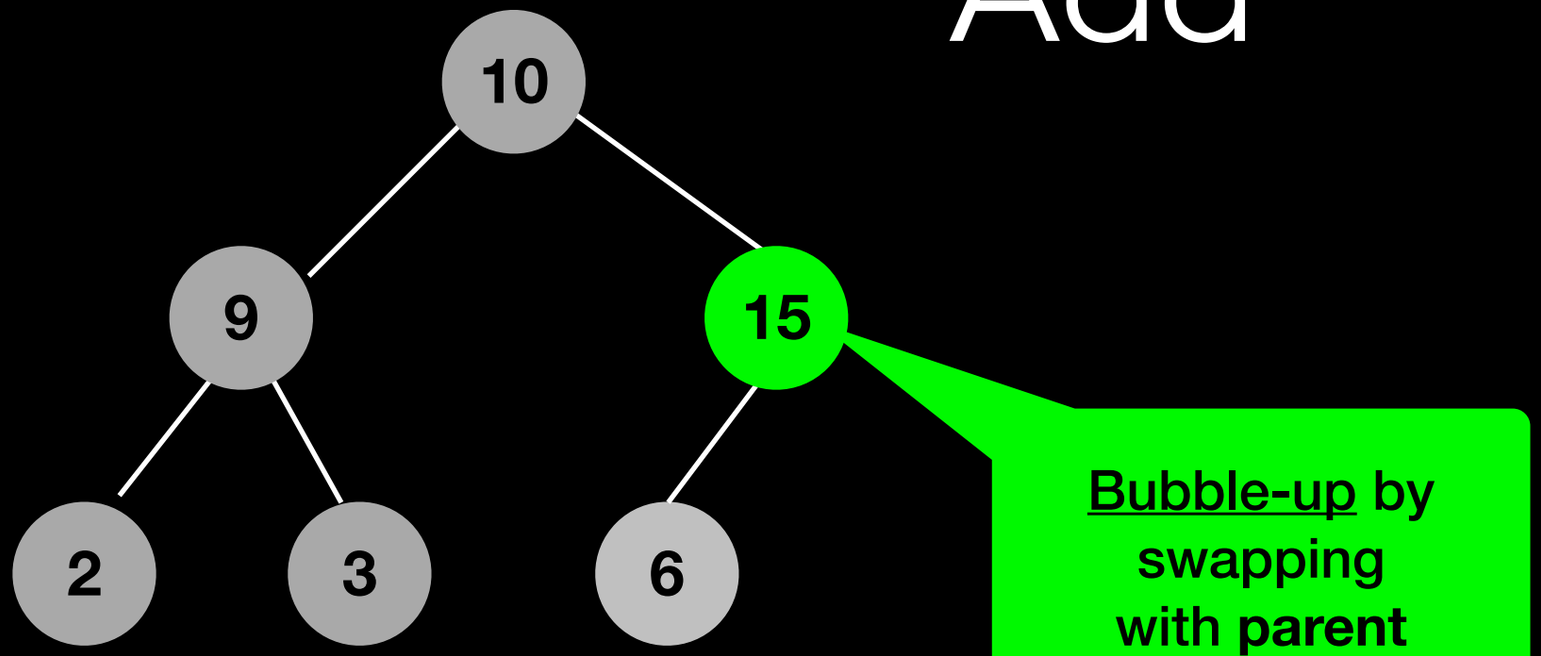
Add



10	9	6	2	3	15
10	9	15	2	3	6

```
items_[i] parent = items_[(i-1)//2]
```

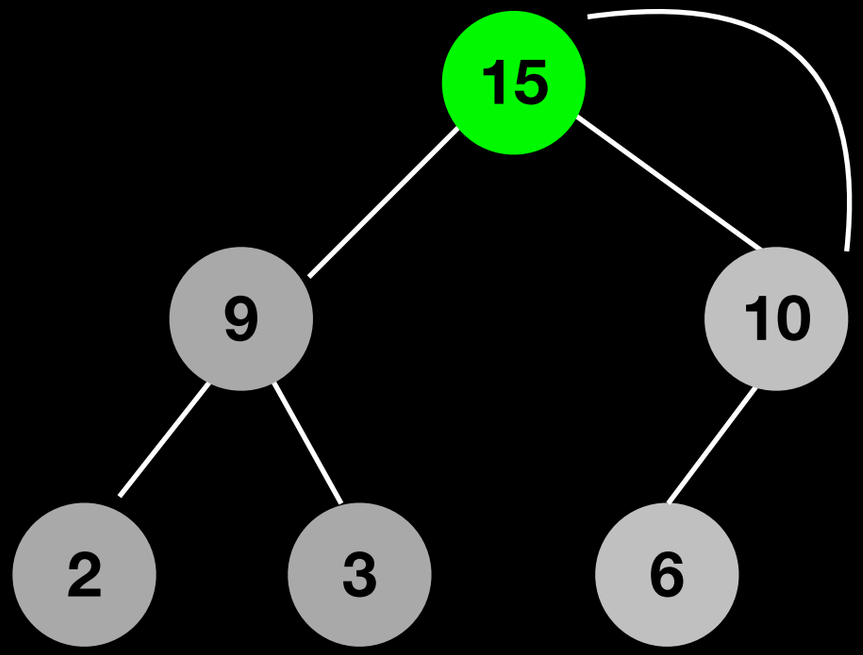
Add



10	9	6	2	3	15
10	9	15	2	3	6

```
items_[i] parent = items_[(i-1)//2]
```

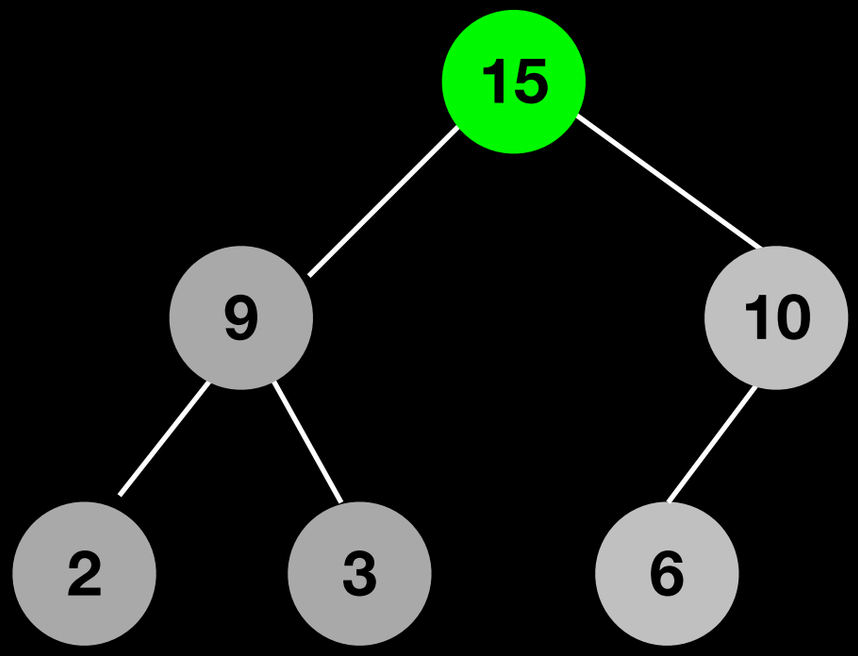
Add



10	9	6	2	3	15
10	9	15	2	3	6
15	9	10	2	3	6

```
items_[i] parent = items_[(i-1)//2]
```

Add



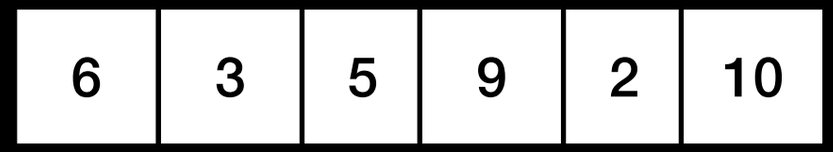
Heap!

O(logn)

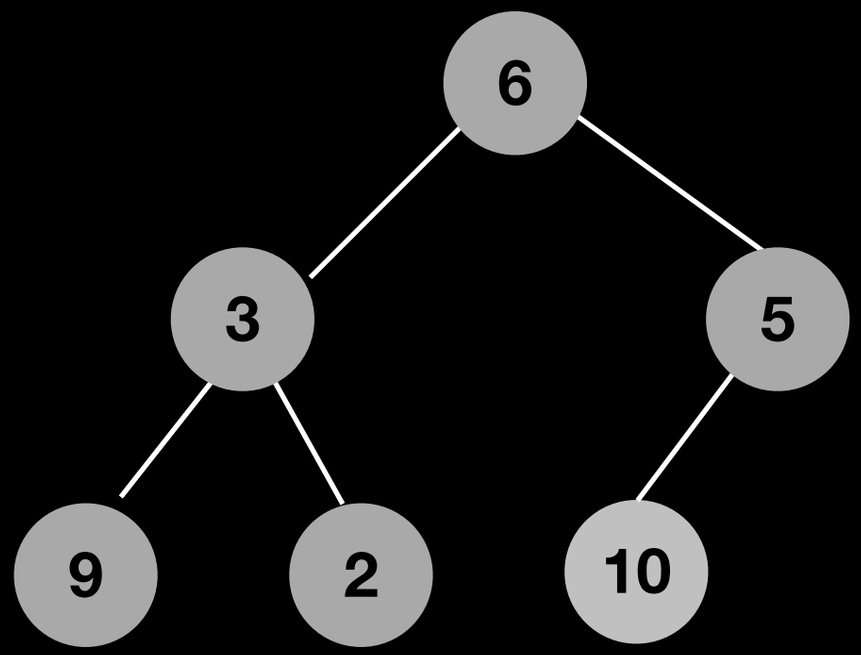
10	9	6	2	3	15
10	9	15	2	3	6
15	9	10	2	3	6

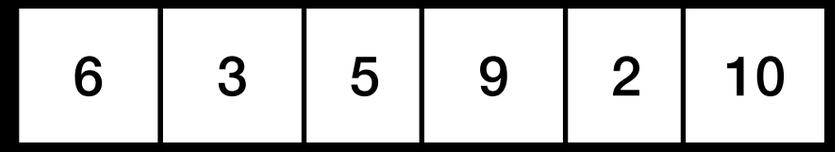
```
items_[i] parent = items_[(i-1)//2]
```

heapCreate



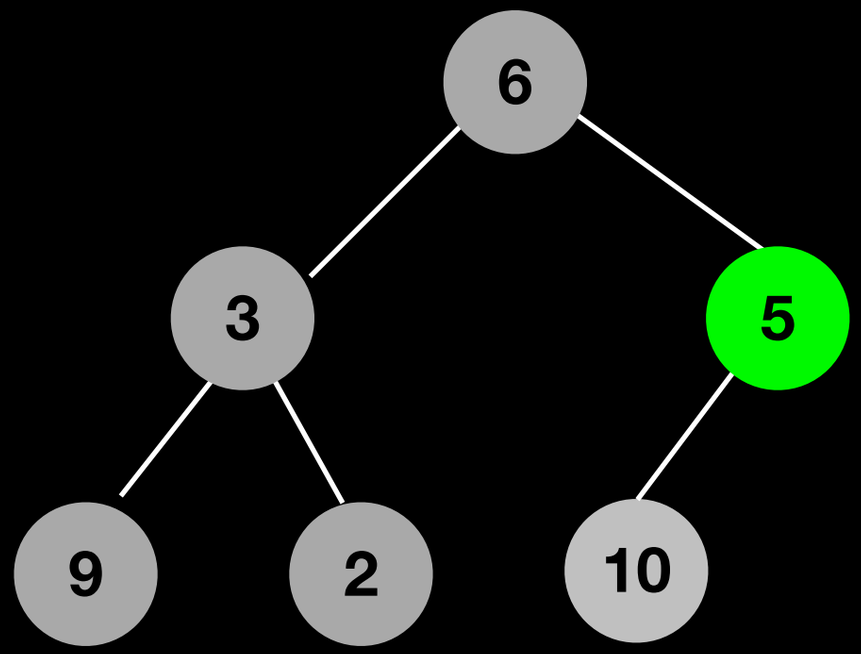
heapCreate

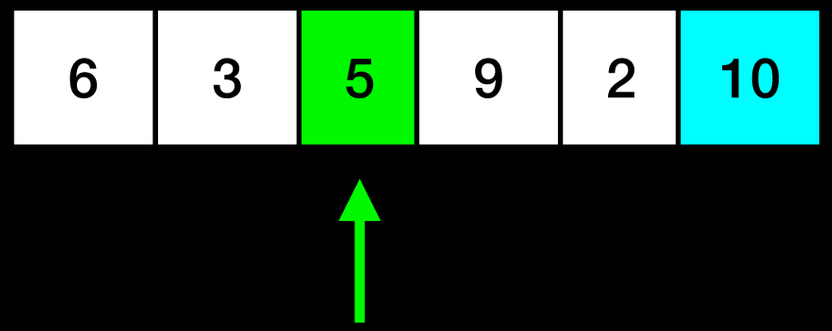





```
for(int i=(itemCount/2)-1; i >=0; i—)
{
    heapRebuild(index);
}
```

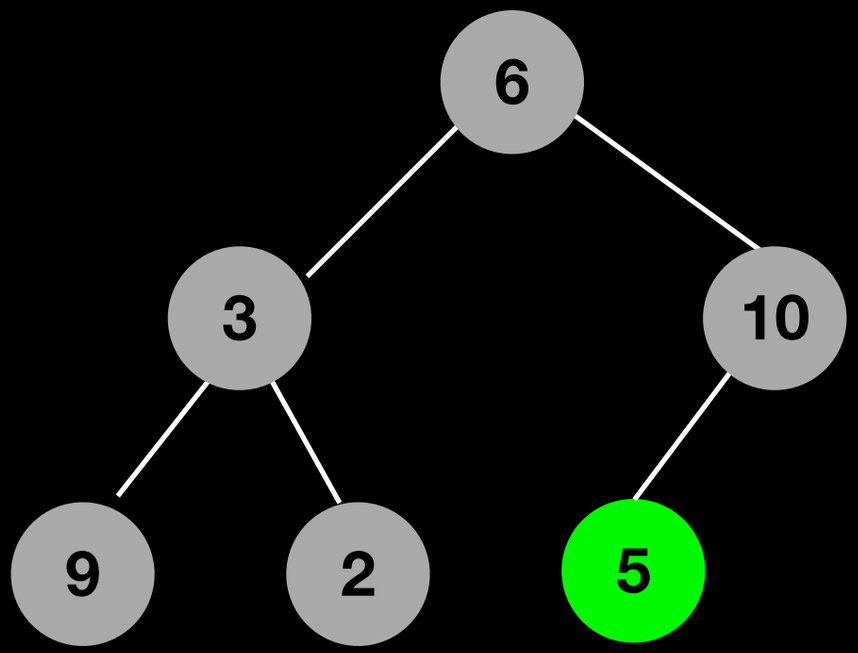
heapCreate

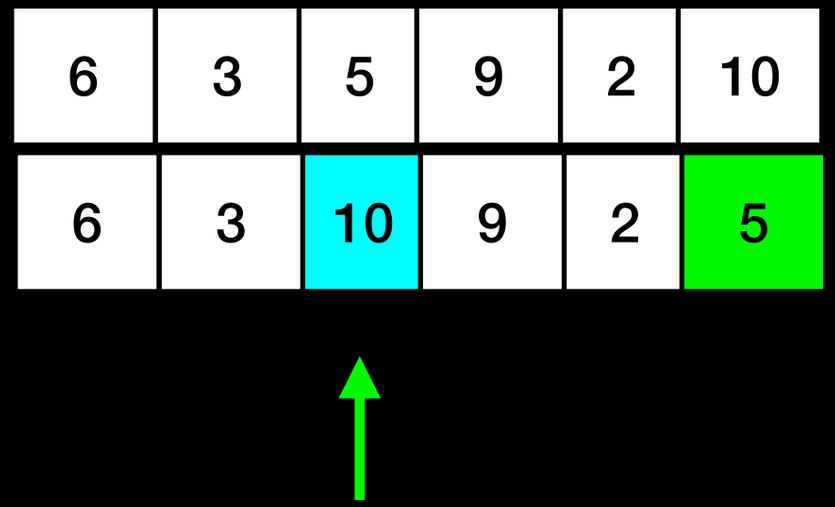





```
for(int i=(itemCount/2)-1; i >=0; i—)
{
    heapRebuild(index);
}
```

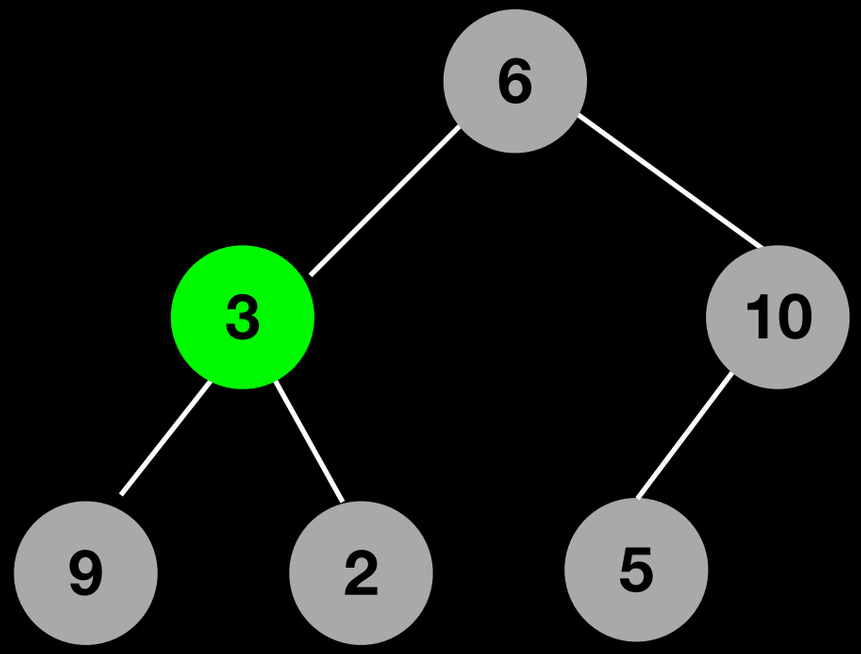
heapCreate

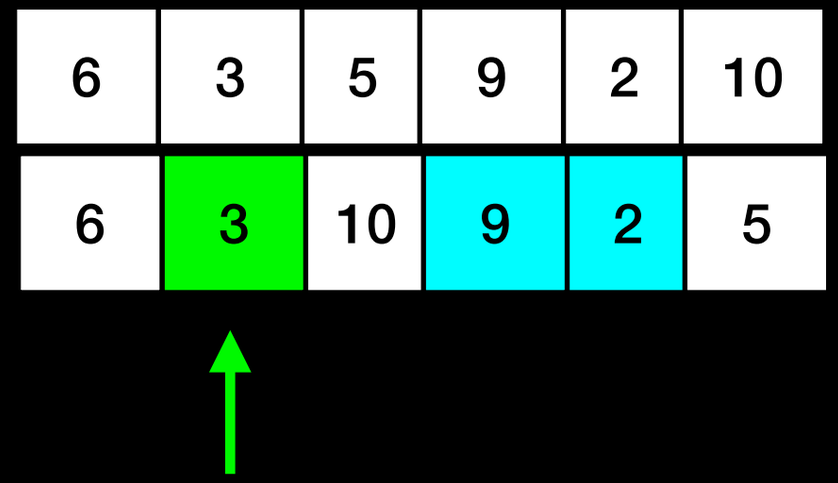





```
for(int i=(itemCount/2)-1; i >=0; i—)
{
    heapRebuild(index);
}
```

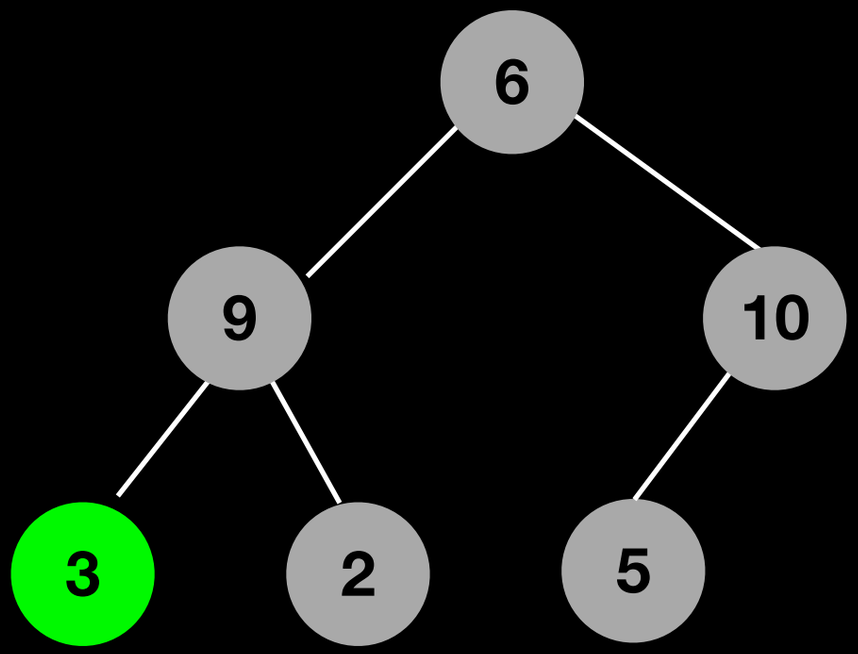
heapCreate



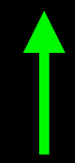



```
for(int i=(itemCount/2)-1; i >=0; i–)
{
    heapRebuild(index);
}
```

heapCreate

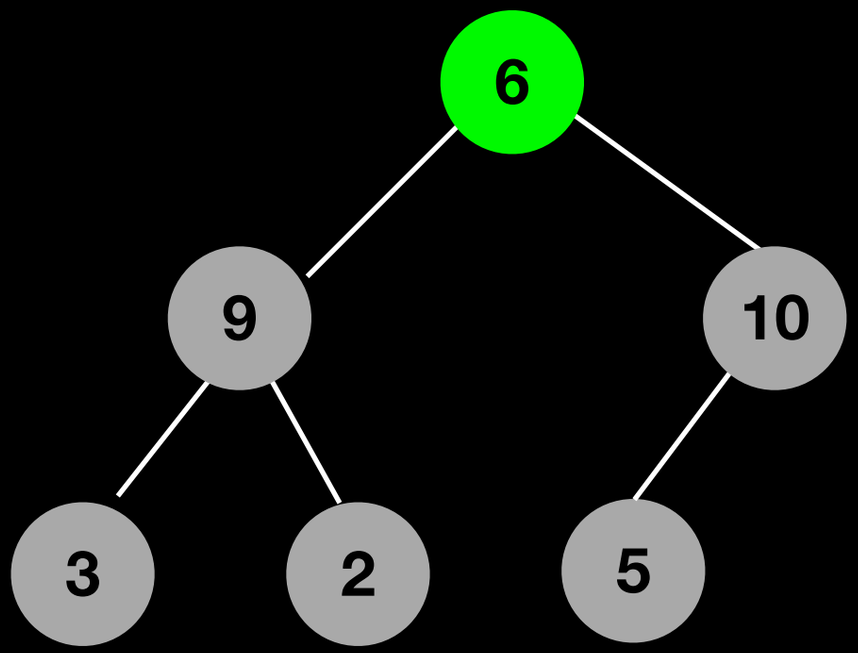


6	3	5	9	2	10
6	3	10	9	2	5
6	9	10	3	2	5

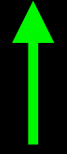


```
for(int i=(itemCount/2)-1; i >=0; i—)
{
    heapRebuild(index);
}
```

heapCreate

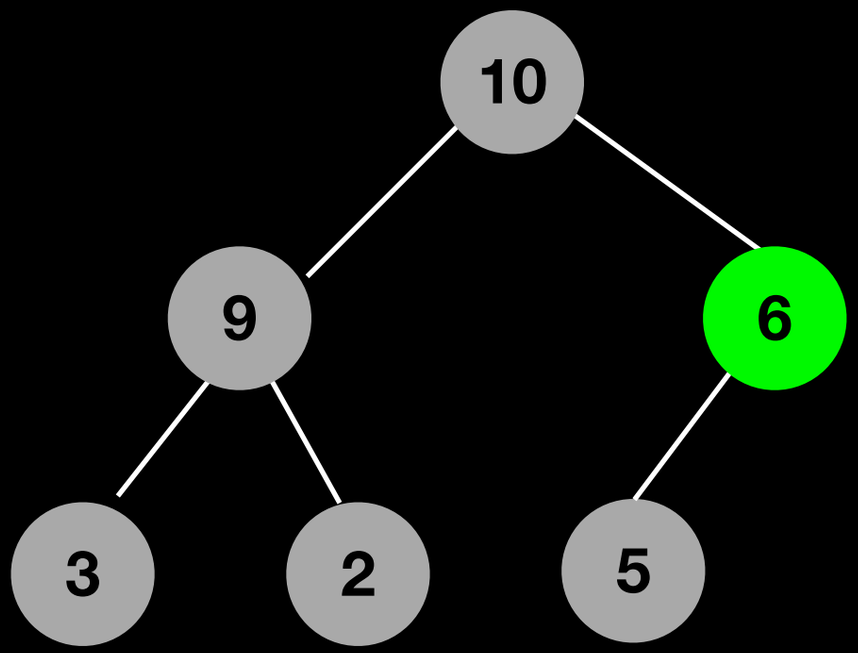


6	3	5	9	2	10
6	3	10	9	2	5
6	9	10	3	2	5

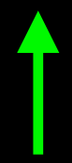


```
for(int i=(itemCount/2)-1; i >=0; i—)
{
    heapRebuild(index);
}
```

heapCreate

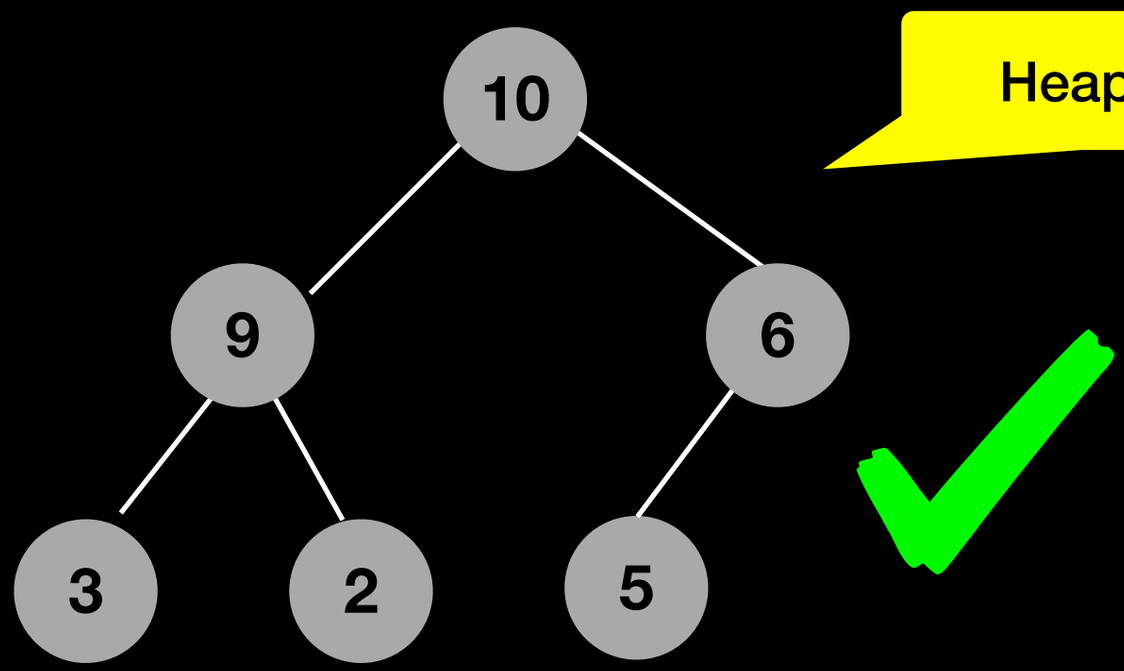


6	3	5	9	2	10
6	3	10	9	2	5
10	9	6	3	2	5



```
for(int i=(itemCount/2)-1; i >=0; i—)
{
    heapRebuild(index);
}
```

heapCreate



Heap!

6	3	5	9	2	10
6	3	10	9	2	5
10	9	6	3	2	5

n/2 calls to heapRebuild = O(n)

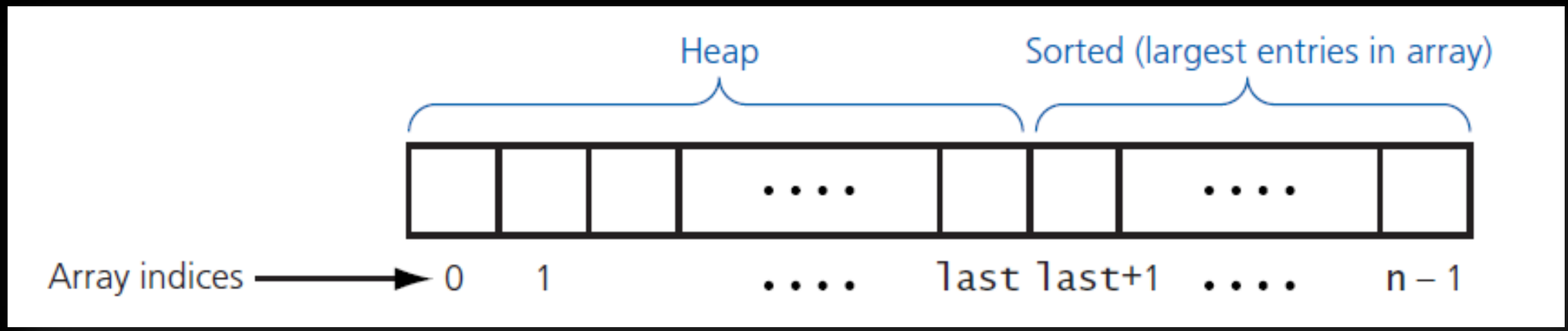
```
for(int i=(itemCount/2)-1; i >=0; i—)
{
    heapRebuild(index);
}
```

Heapsort

Heapsort

Given an unsorted array:

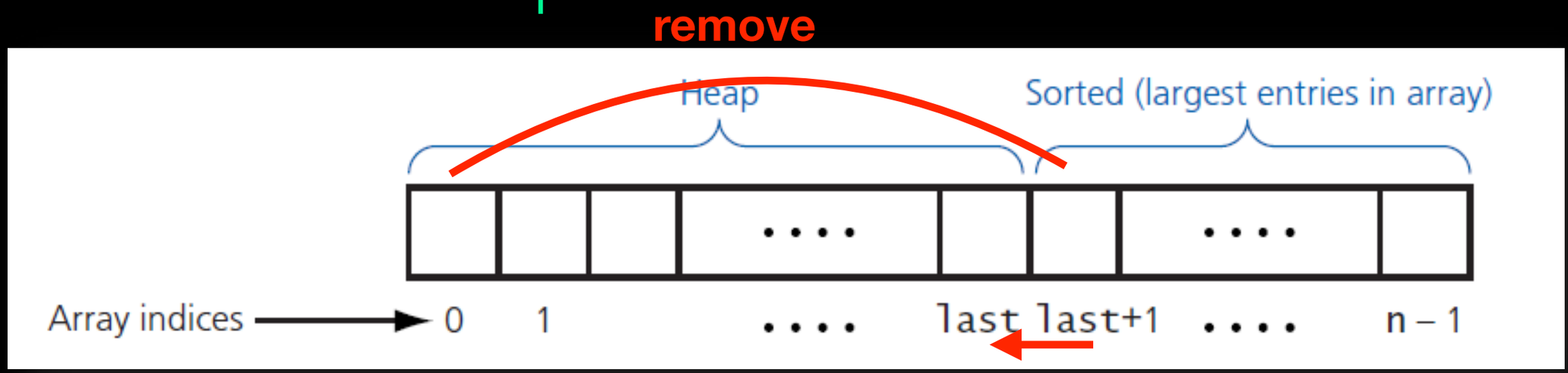
- heapCreate
- last = n - 2
- repeat:
    - swap items[0] with items[last+1]
    - last--
- rebuildHeap



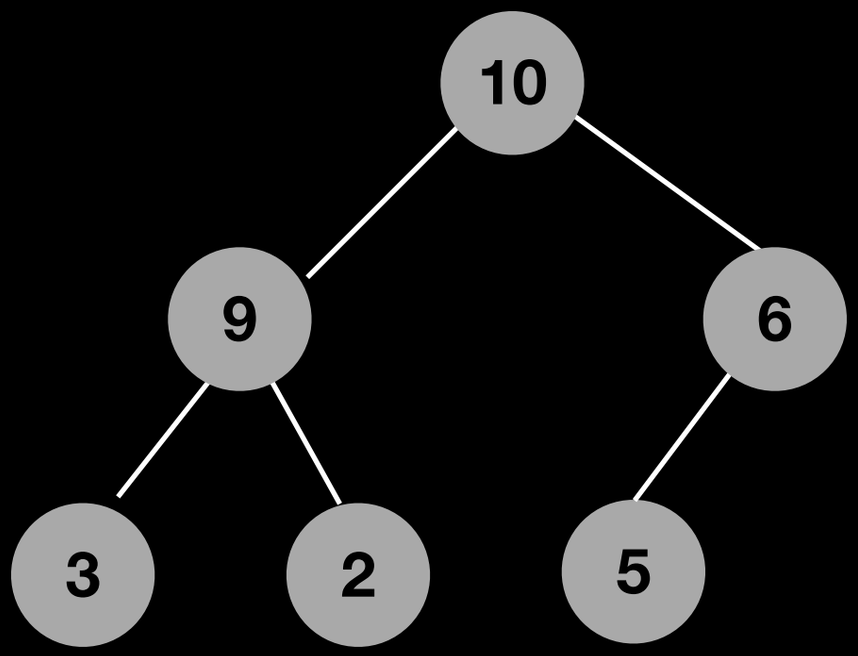
Heapsort

Given an unsorted array:

- heapCreate
- last = n - 2
- repeat:
    - swap items[0] with items[last+1]
    - last--
- rebuildHeap

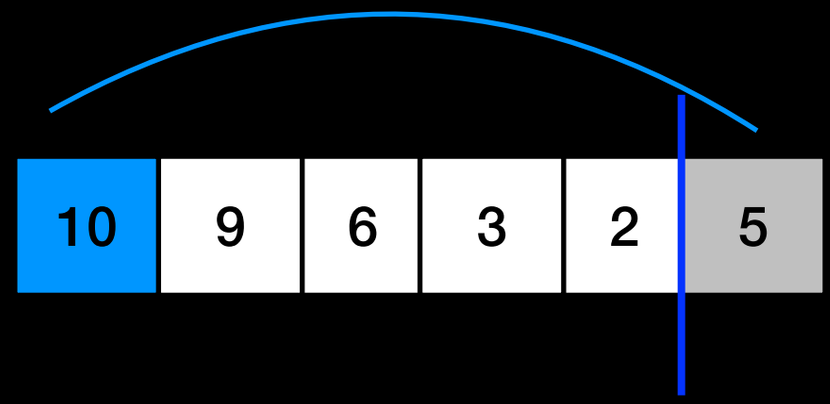


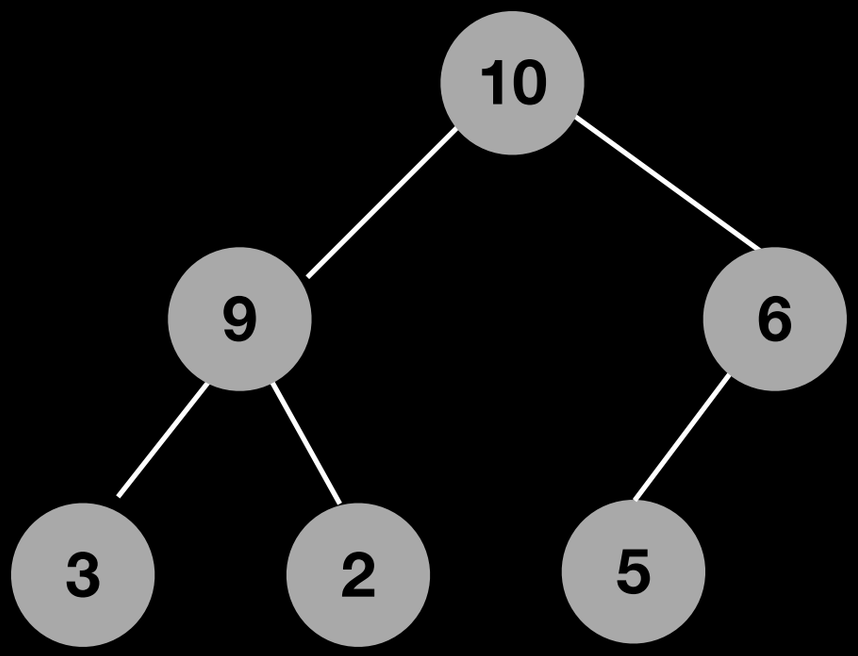
Heapsort



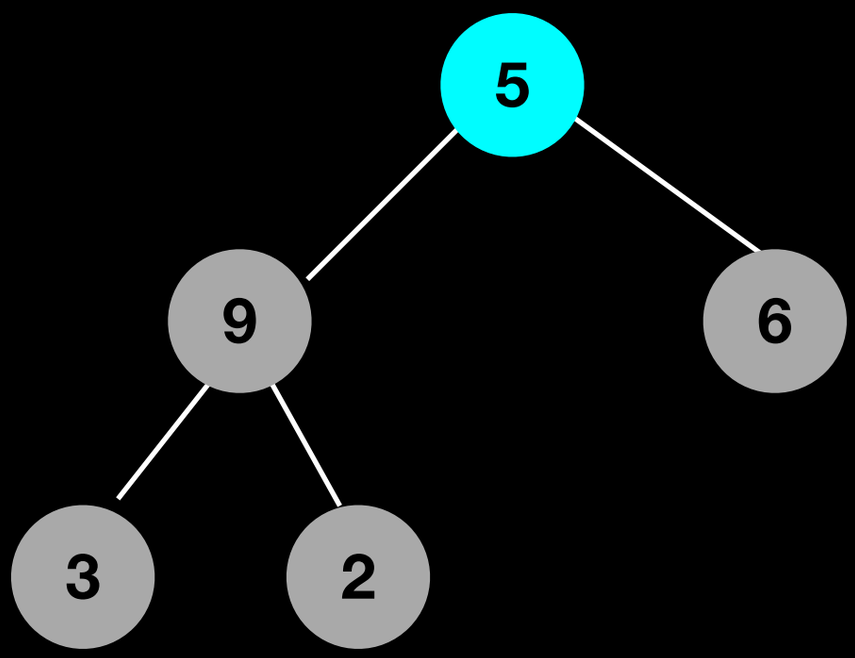
10	9	6	3	2	5

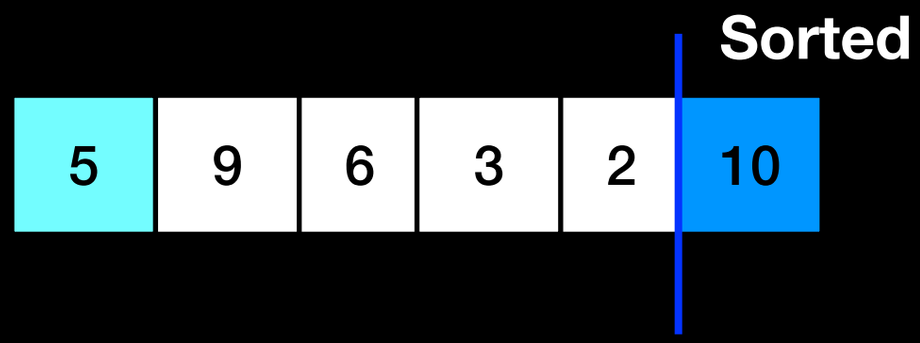
Heapsort





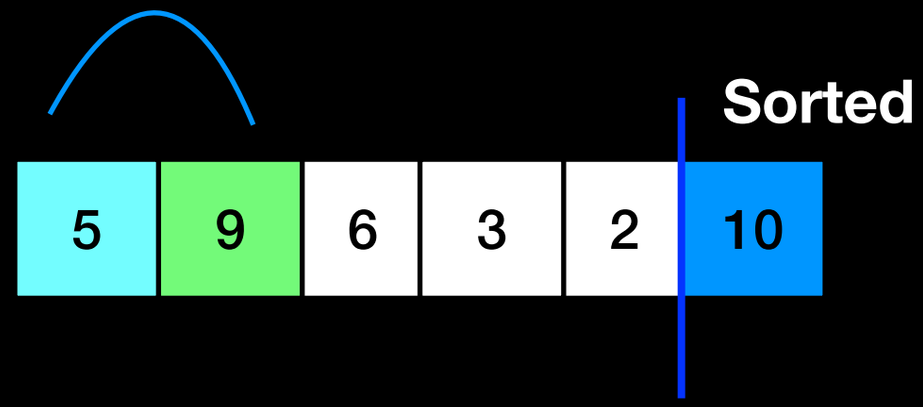
Heapsort

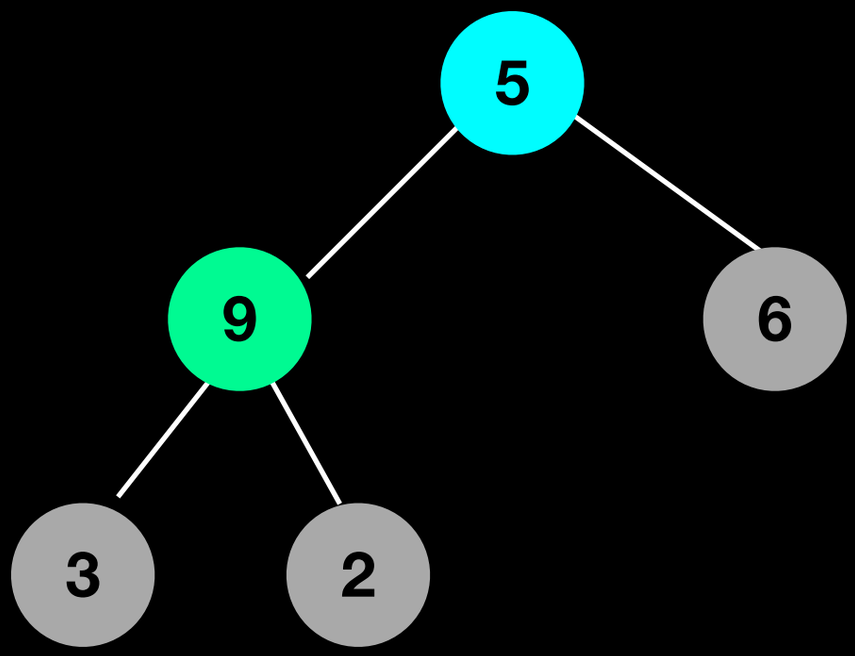




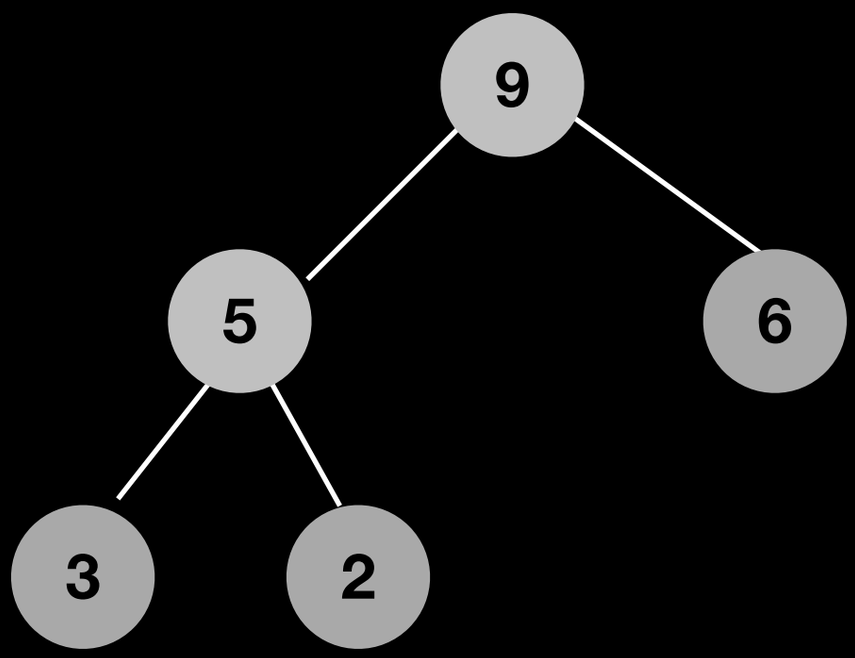
heapRebuild

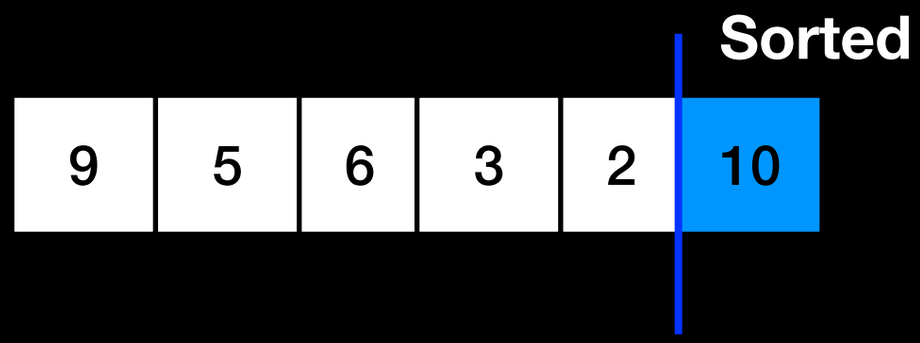
Heapsort



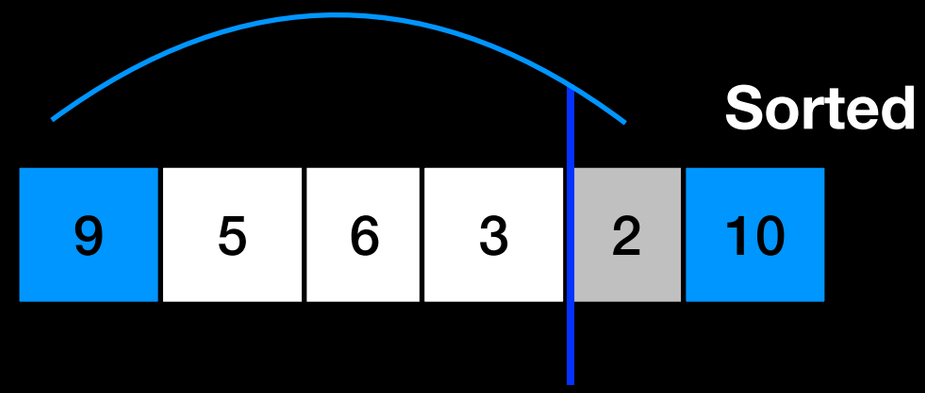


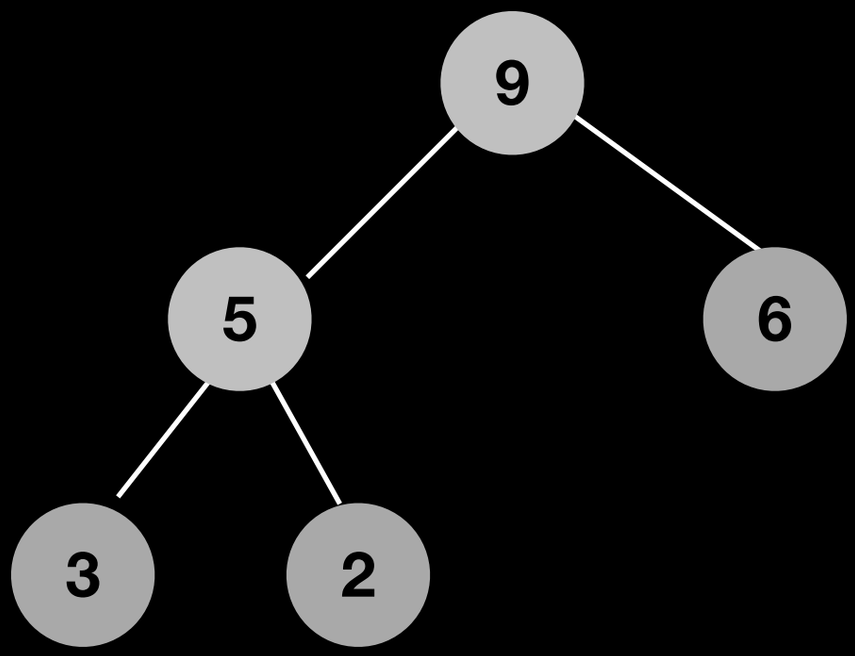
Heapsort





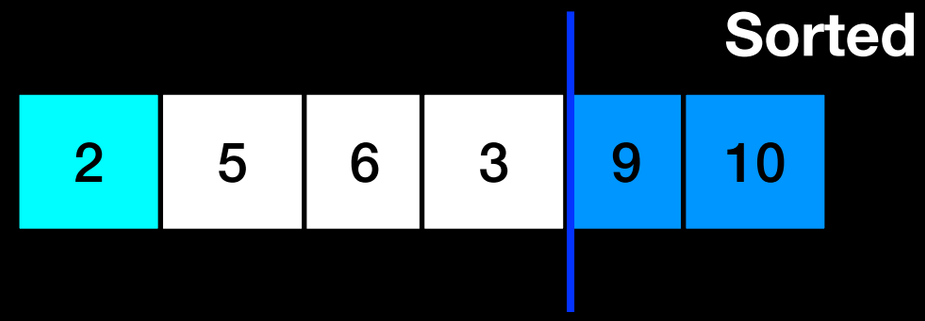
Heapsort

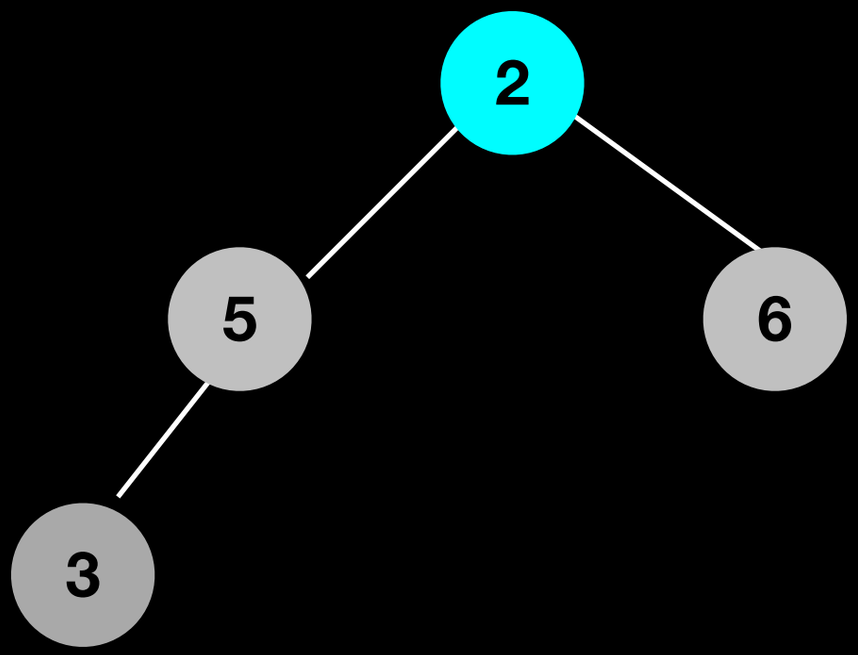




heapRebuild

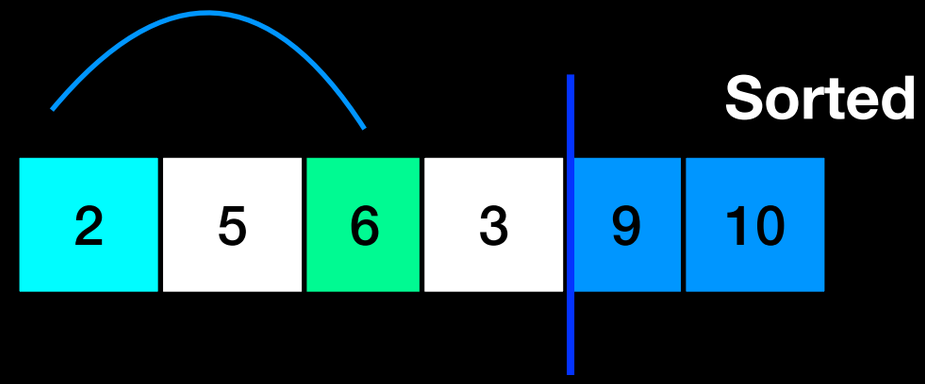
Heapsort

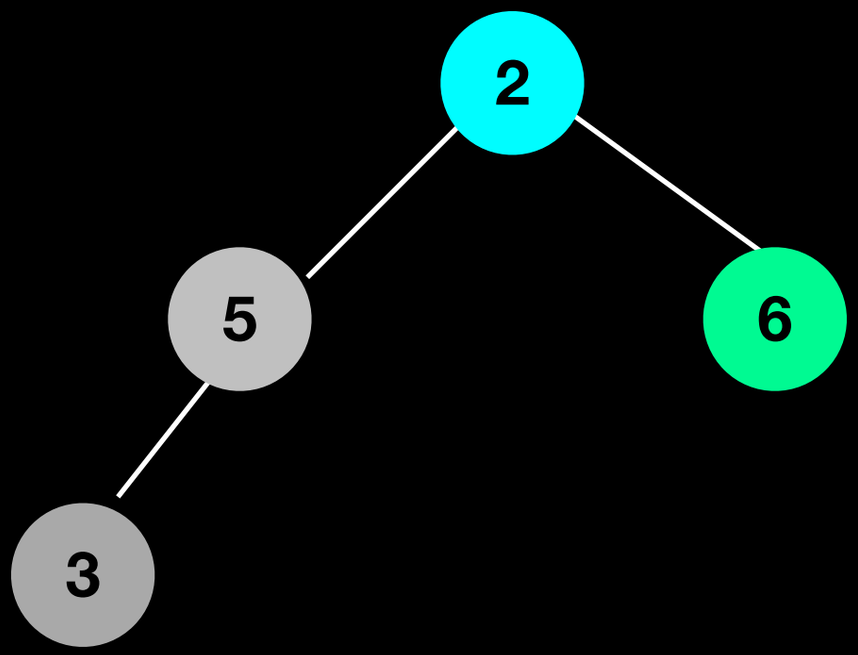




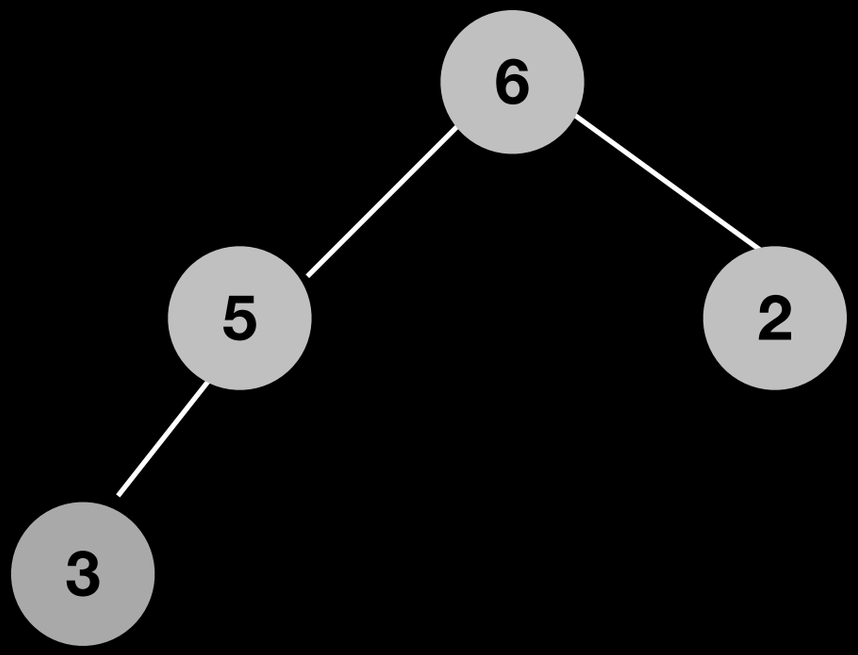
heapRebuild

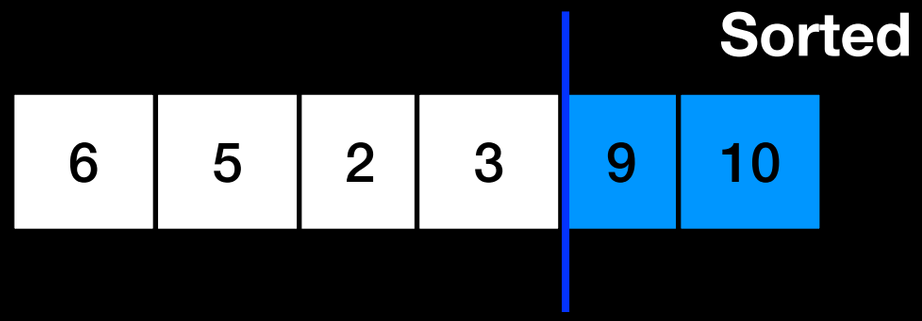
Heapsort



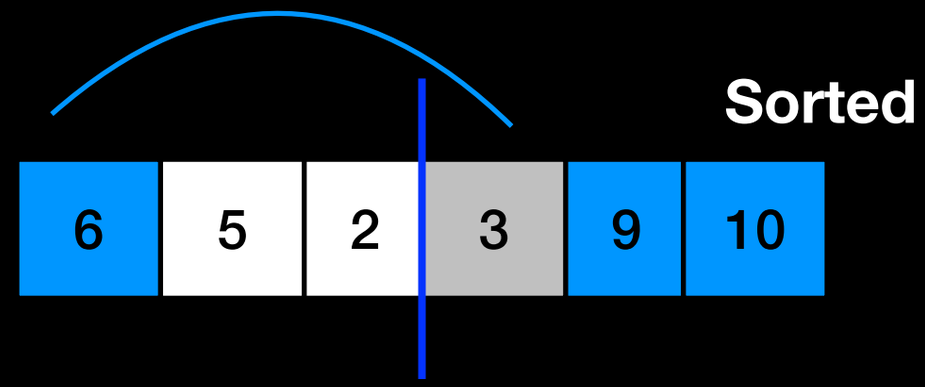


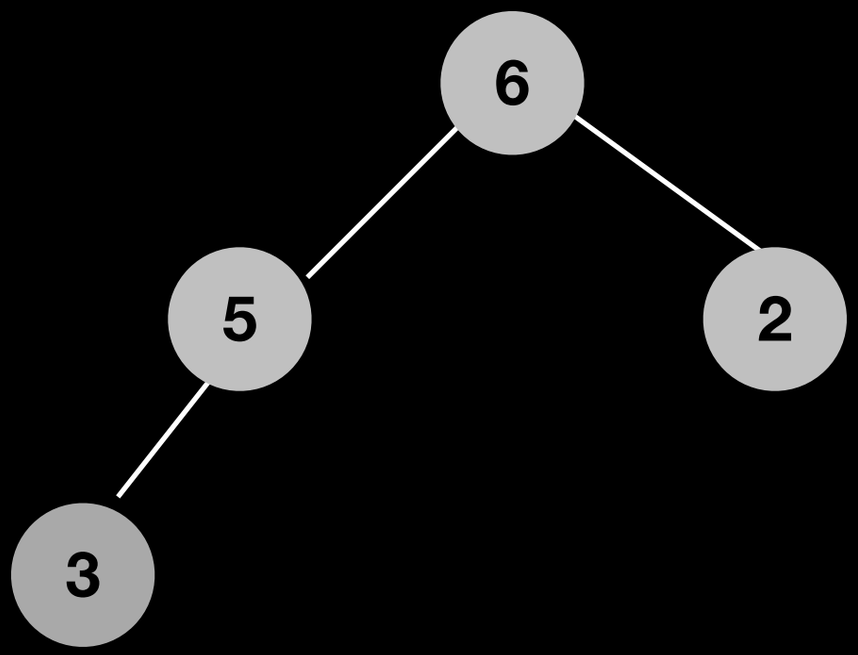
Heapsort





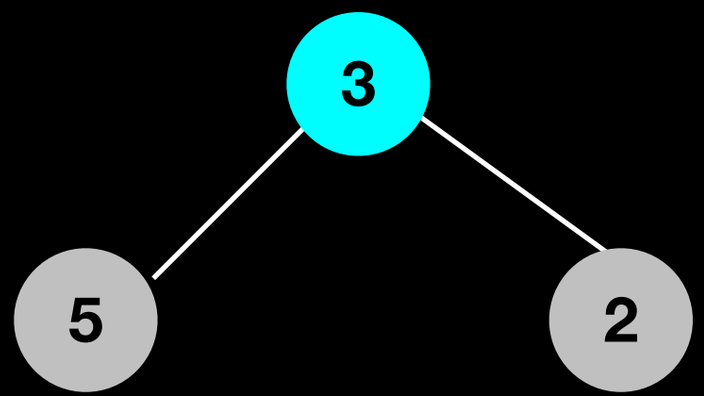
Heapsort

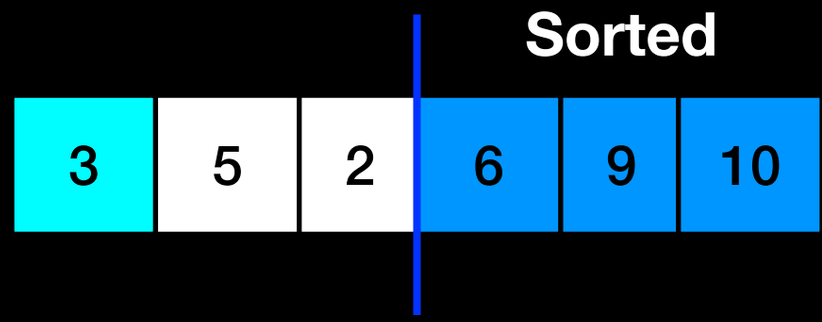




heapRebuild

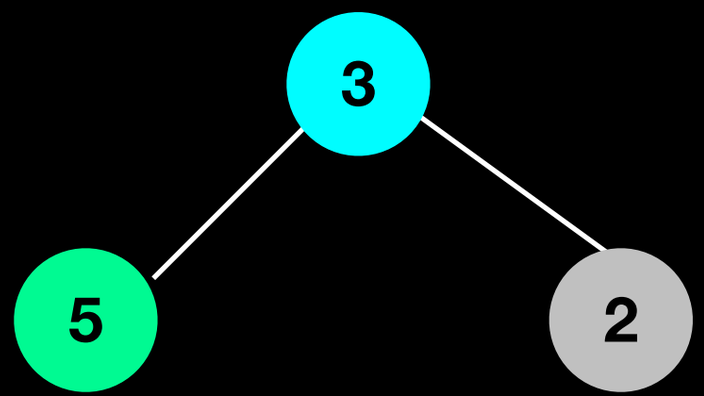
Heapsort

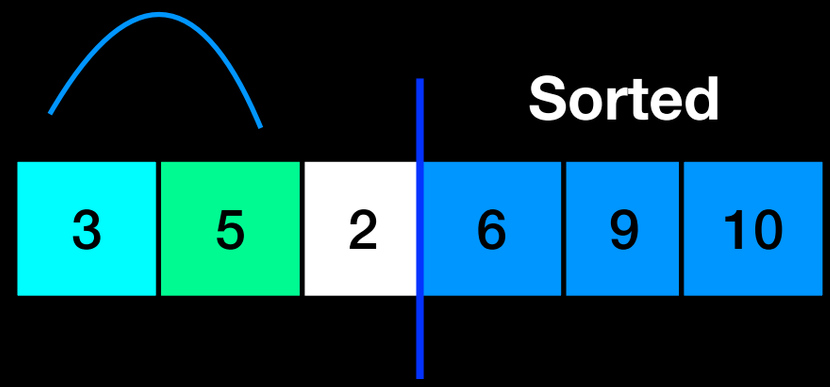




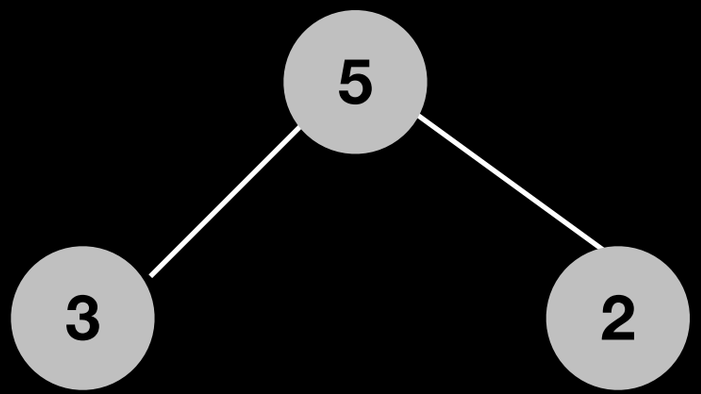
heapRebuild

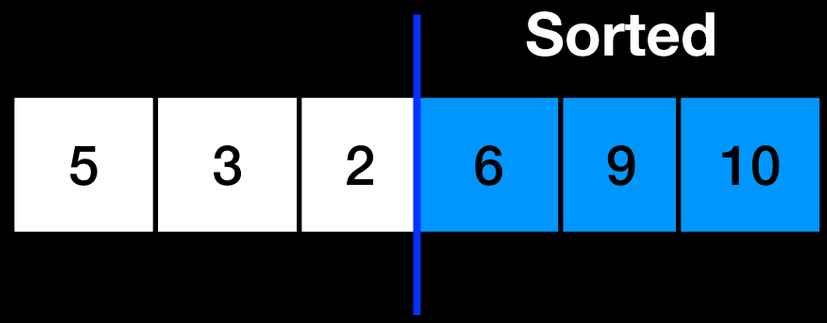
Heapsort



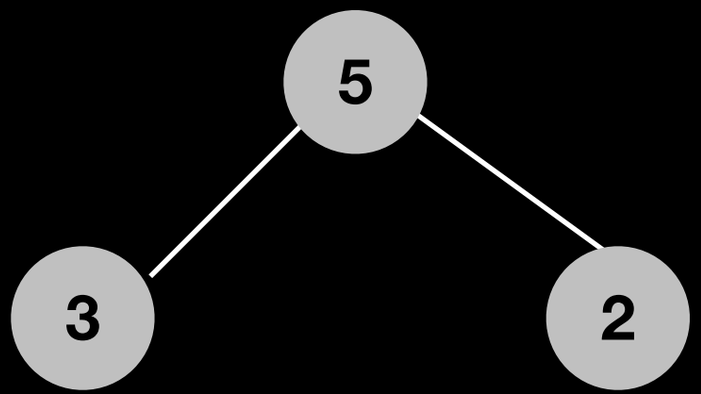


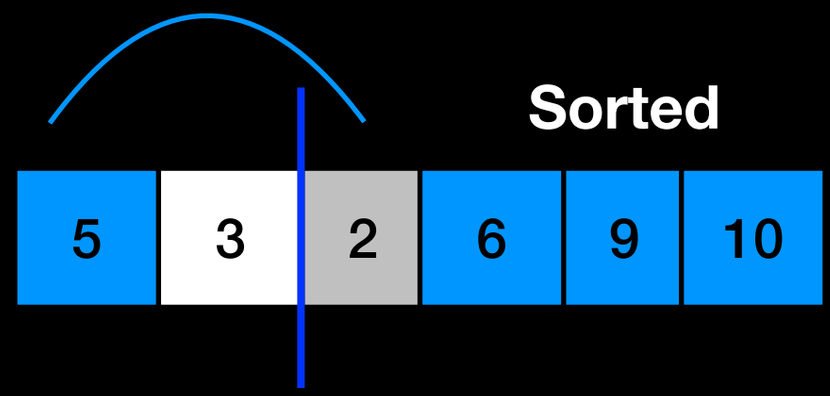
Heapsort





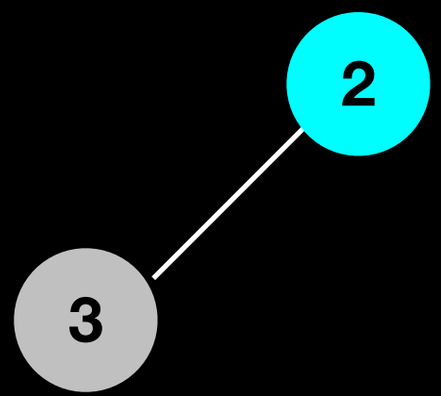
Heapsort

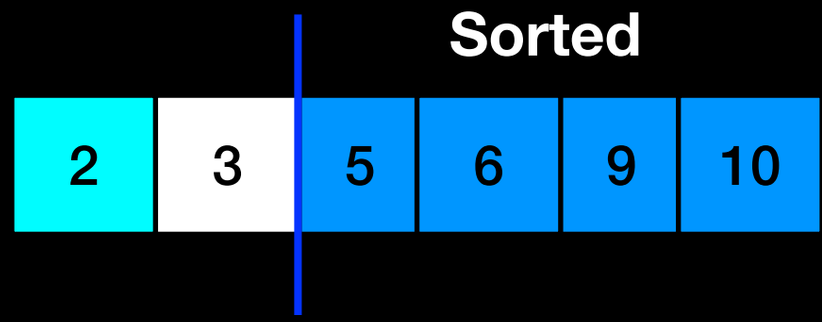




heapRebuild

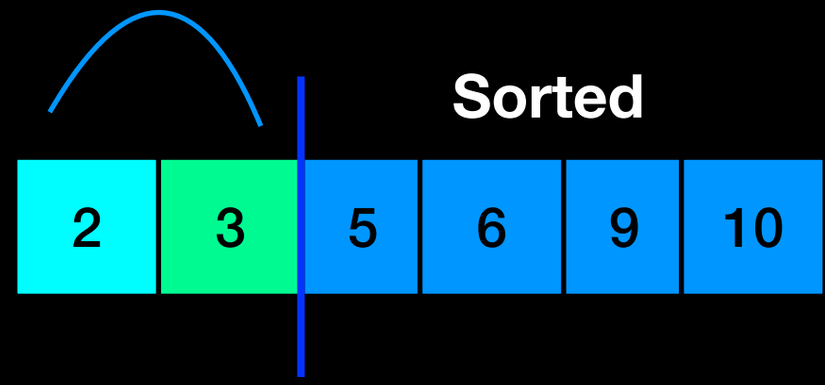
Heapsort

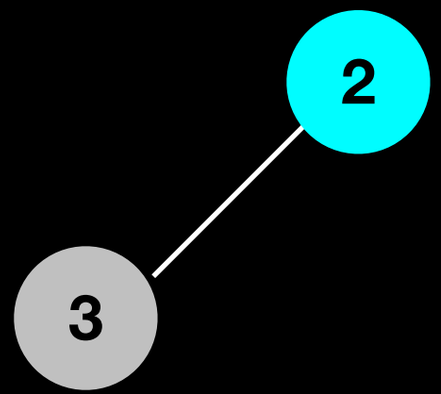




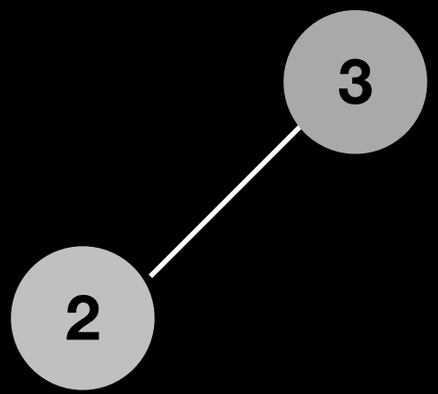
heapRebuild

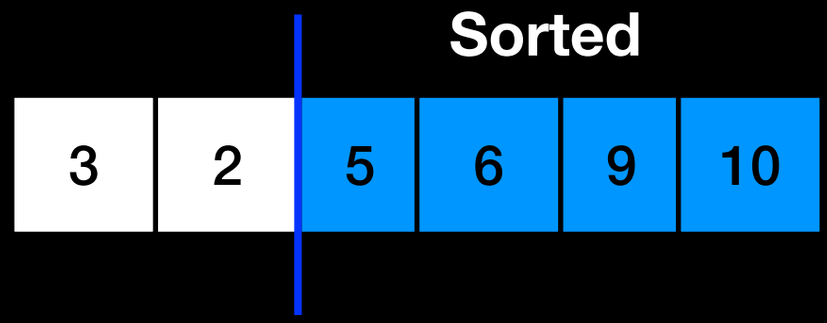
Heapsort



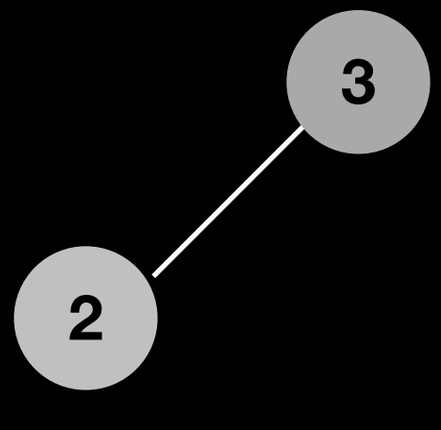


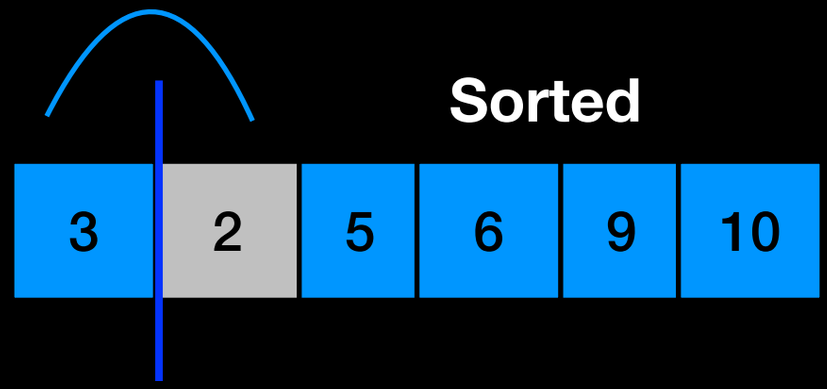
Heapsort



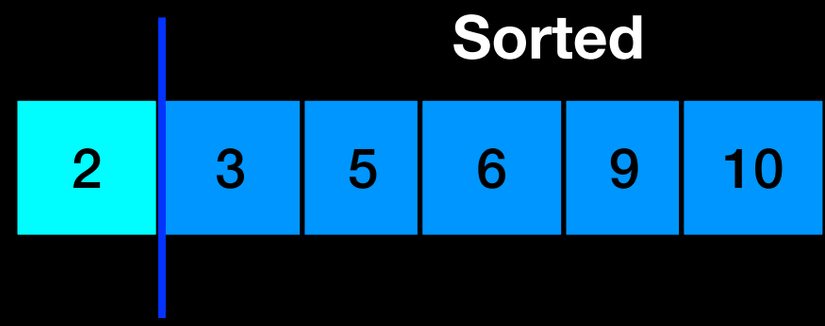


Heapsort





Heapsort



Heapsort

Sorted

2	3	5	6	9	10

Heapsort Analysis

1. heapCreate -> O(n)

2. heapRebuild -> O(logn) repeated for each of the n sorted items

O(n) + O(n logn) = **O(n logn)**

Like MergeSort but no extra space needed!