

Data Structures (II)

- Binary Heap
- Binary Search Tree
- Hash Table

Lau Chi Yung (Steven)

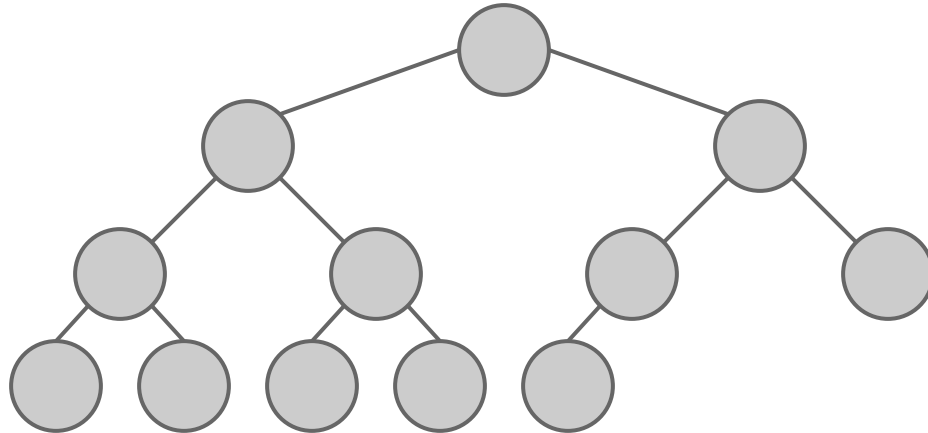
Binary Heap?

- Want a data structure which supports
 - insert element
 - remove minimum

	insert element	remove minimum
plain array	$O(1)$	$O(N)$
sorted array	$O(N)$	$O(1)$
binary heap	$O(\log(N))$	$O(\log(N))$

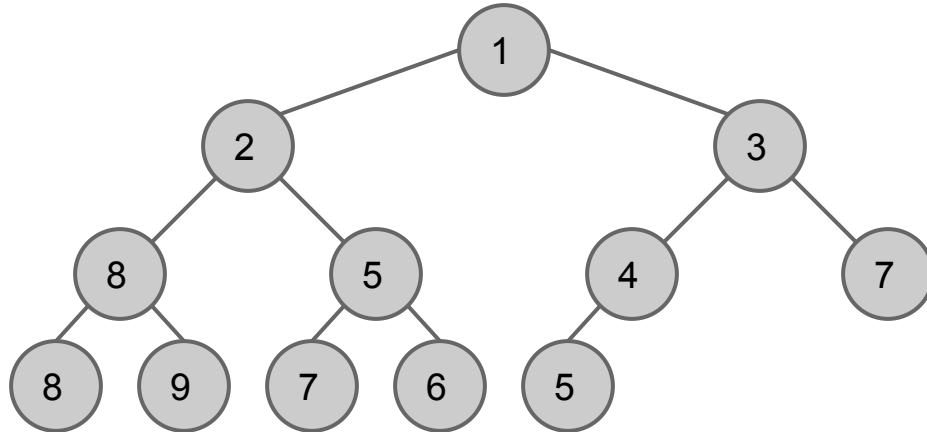
Binary Heap - shape

- Complete binary tree
 - all levels (except possibly the bottom) are fully filled
 - the bottom is filled from left to right



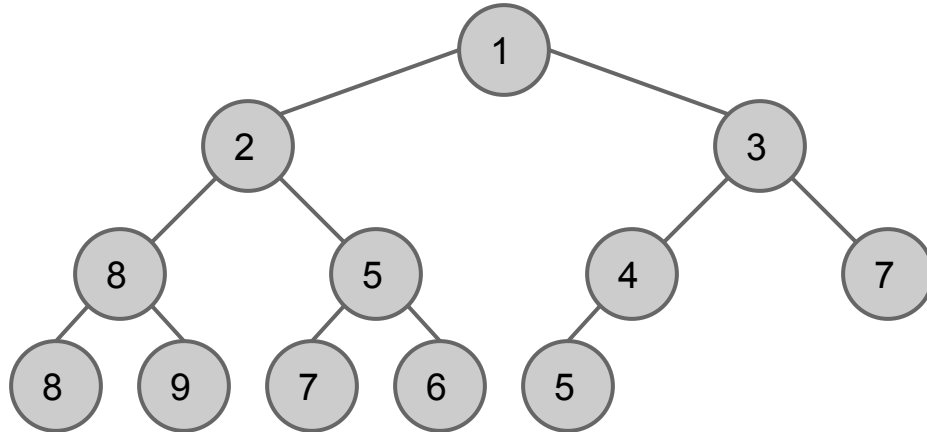
Binary Heap - heap property

- All nodes are greater than or equal to parent



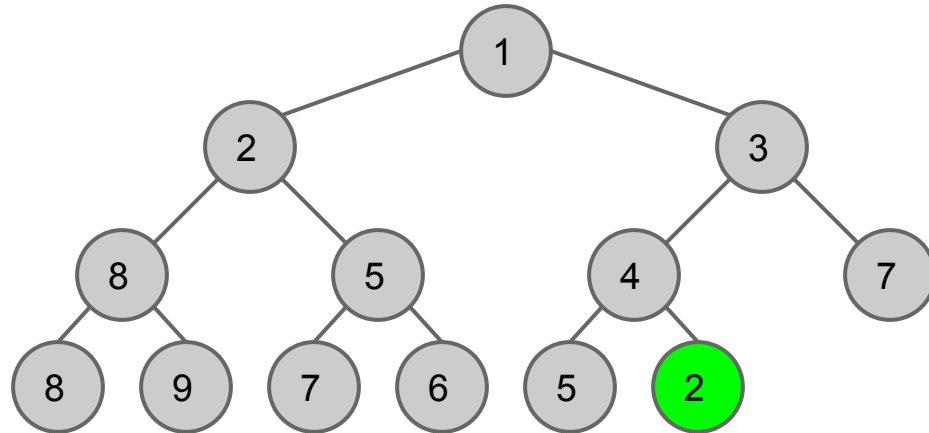
Binary Heap

- Insert element
 - insert at the bottom



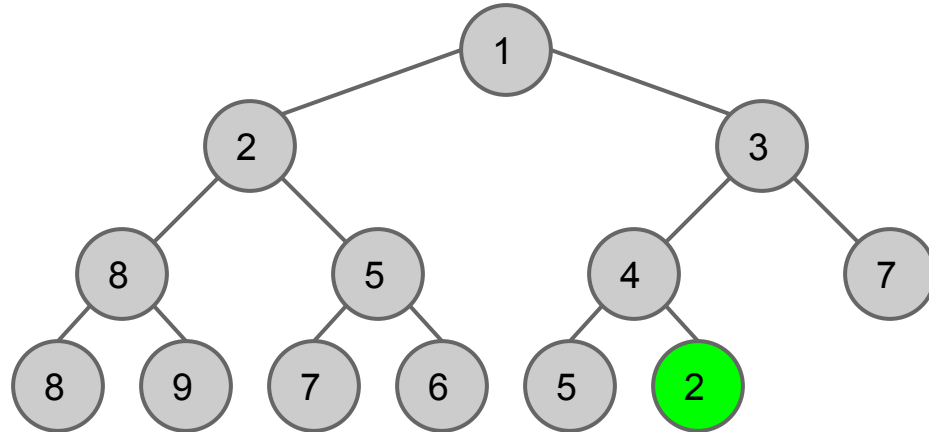
Binary Heap

- Insert element
 - insert at the bottom



Binary Heap

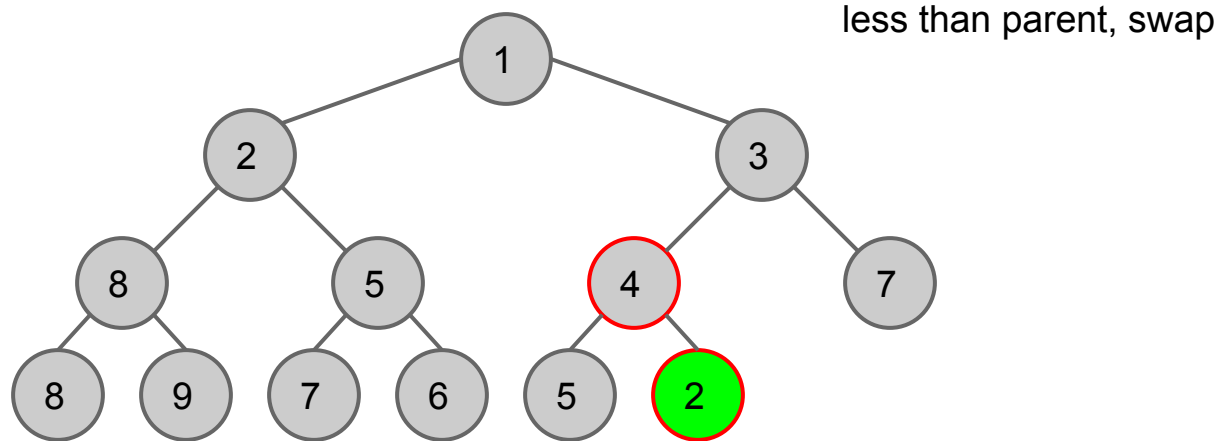
- Insert element
 - insert at the bottom
 - heap property violated; do *sift-up* to recover



Binary Heap

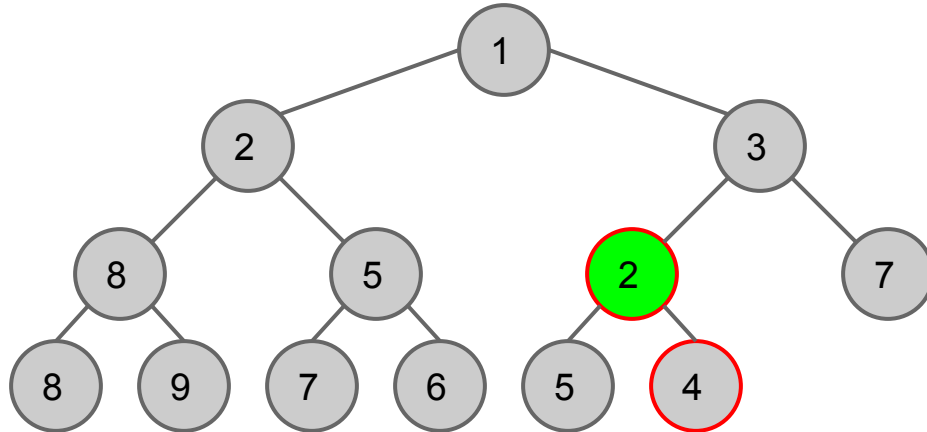
- Insert element

- insert at the bottom
- heap property violated; do *sift-up* to recover



Binary Heap

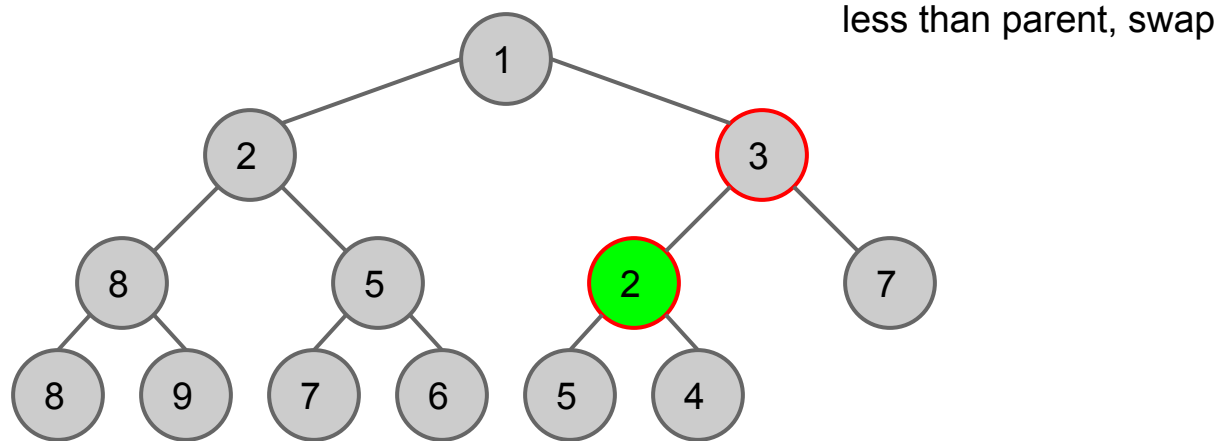
- Insert element
 - insert at the bottom
 - heap property violated; do *sift-up* to recover



Binary Heap

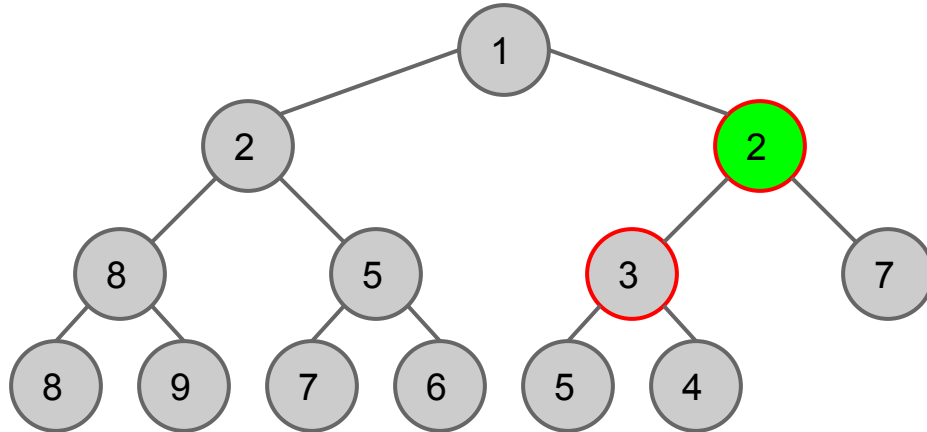
- Insert element

- insert at the bottom
- heap property violated; do *sift-up* to recover



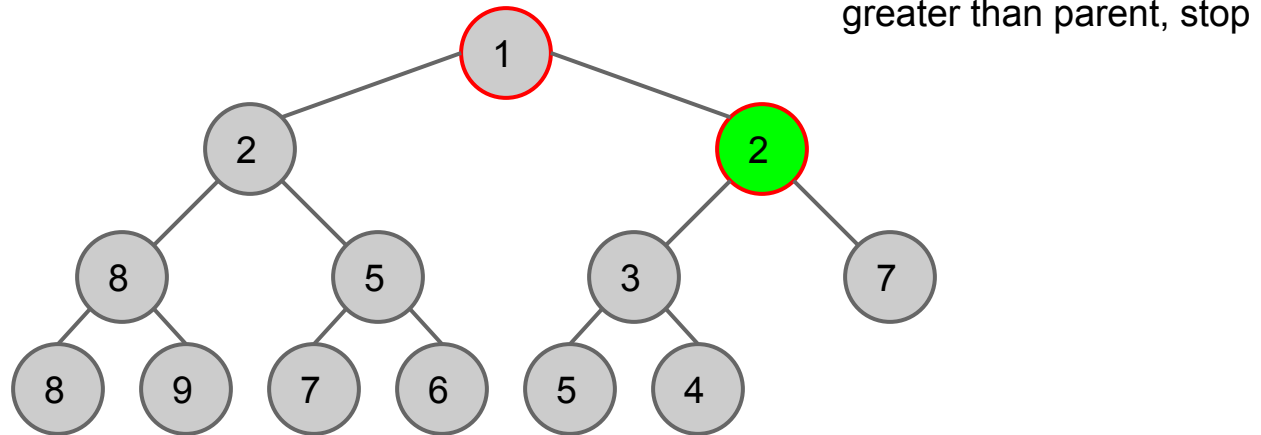
Binary Heap

- Insert element
 - insert at the bottom
 - heap property violated; do *sift-up* to recover



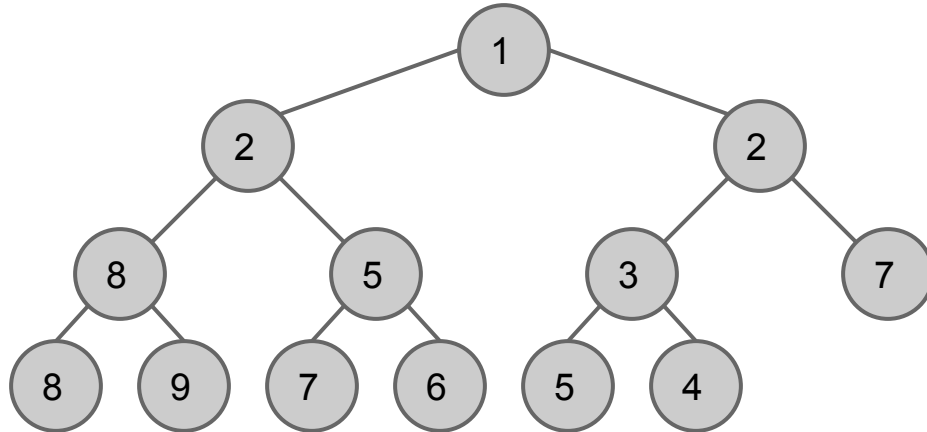
Binary Heap

- Insert element
 - insert at the bottom
 - heap property violated; do *sift-up* to recover



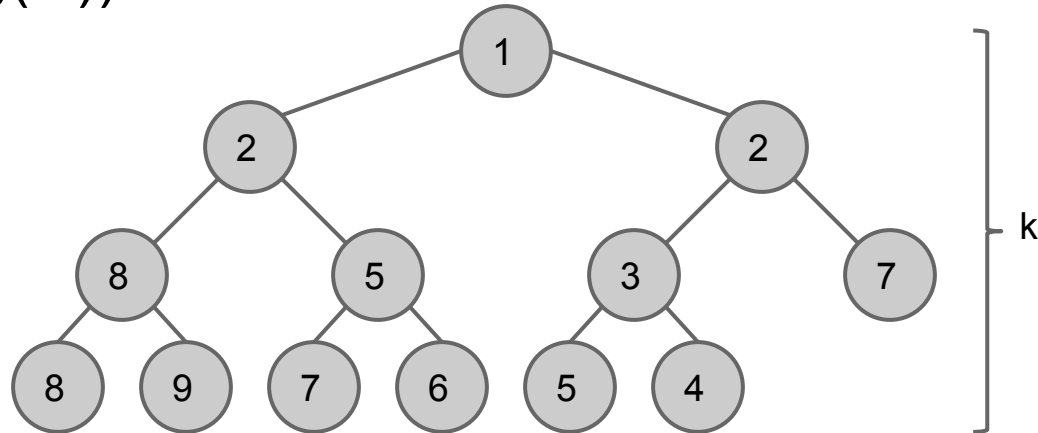
Binary Heap

- Insert element
 - insert at the bottom
 - heap property violated; do *sift-up* to recover



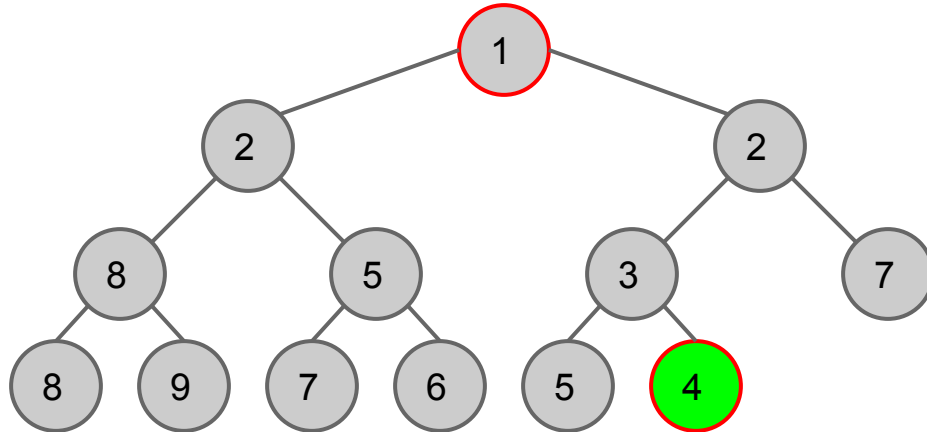
Binary Heap

- Insert element - time complexity
 - maximum number of swaps = $k-1$
 - $2^{k-1} \leq N \Rightarrow k-1 \leq \log_2 N$
 - $O(\log(N))$



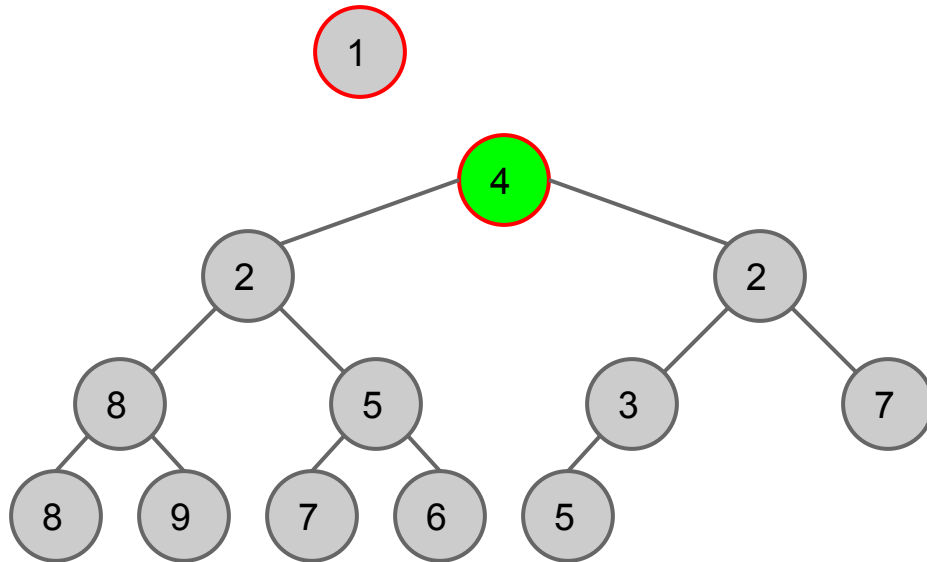
Binary Heap

- Remove minimum
 - replace the root with the last node at the bottom



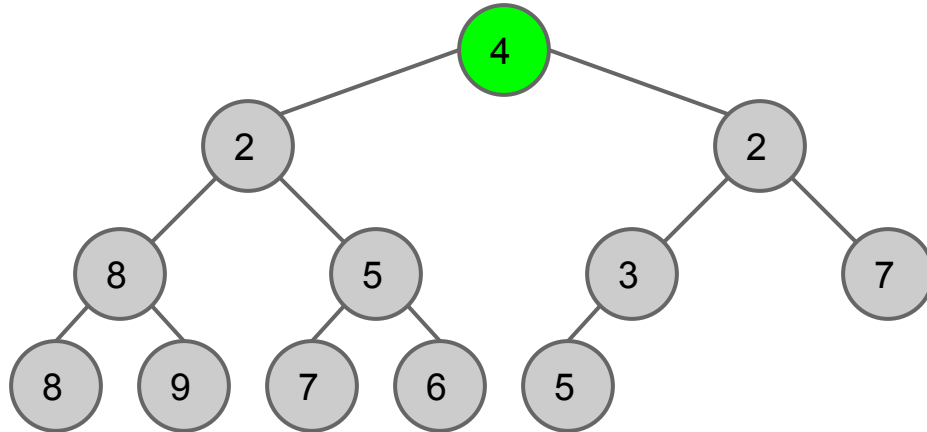
Binary Heap

- Remove minimum
 - replace the root with the last node at the bottom



Binary Heap

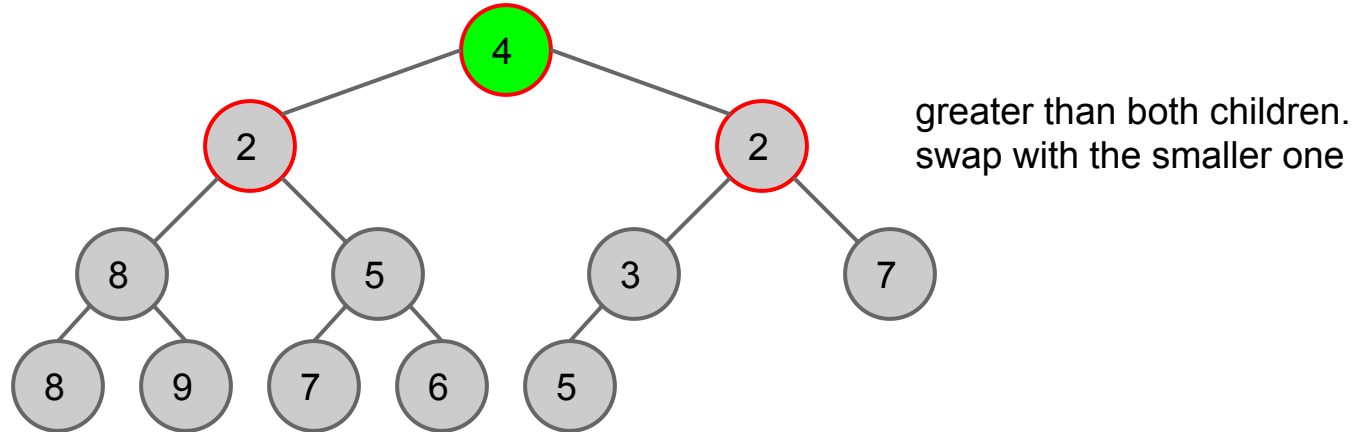
- Remove minimum
 - replace the root with the last node at the bottom
 - heap property violated; do *sift-down* to recover



Binary Heap

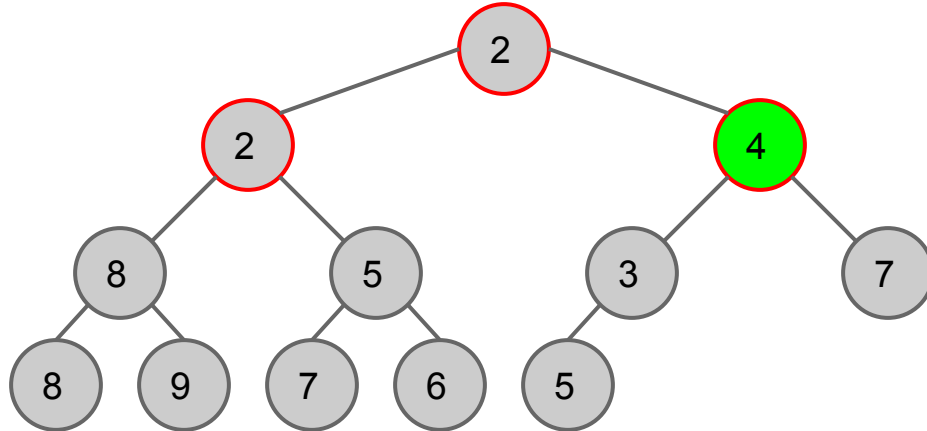
- Remove minimum

- replace the root with the last node at the bottom
- heap property violated; do *sift-down* to recover



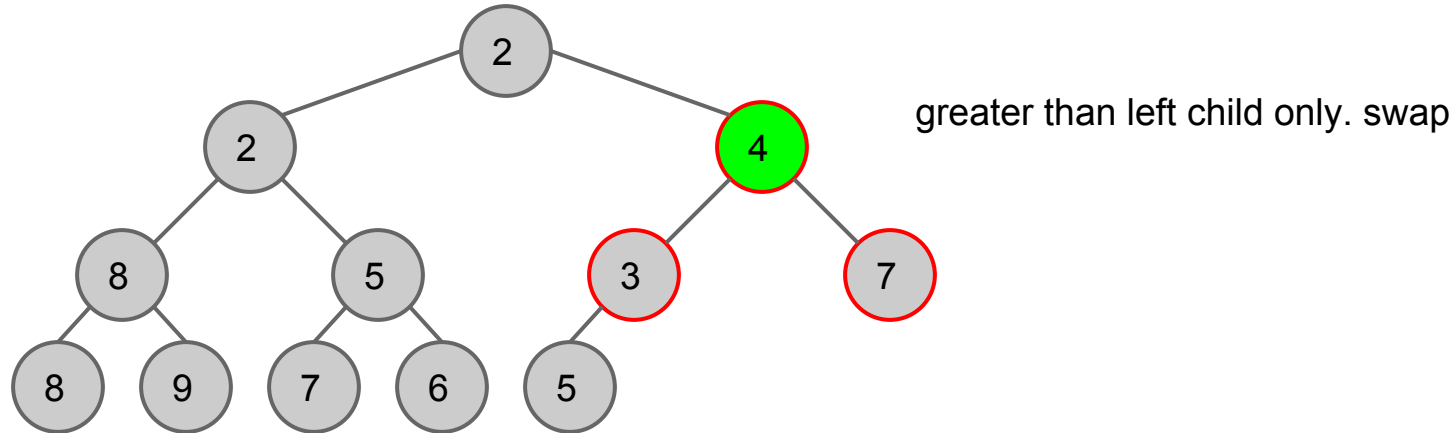
Binary Heap

- Remove minimum
 - replace the root with the last node at the bottom
 - heap property violated; do *sift-down* to recover



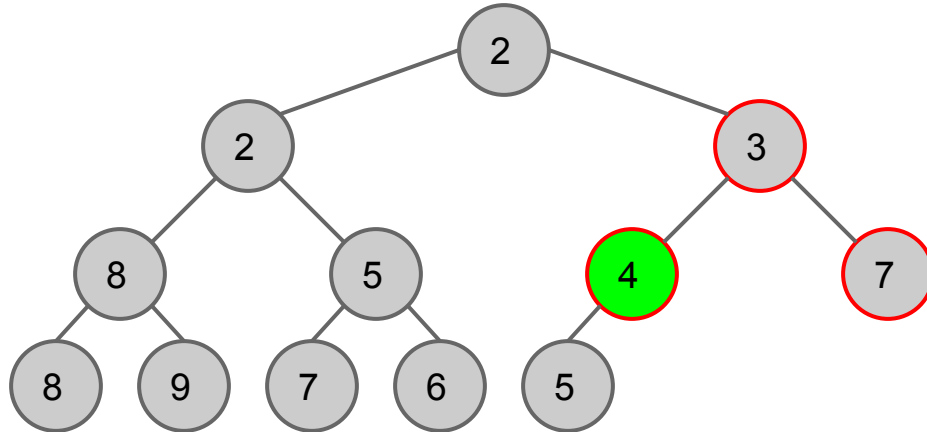
Binary Heap

- Remove minimum
 - replace the root with the last node at the bottom
 - heap property violated; do *sift-down* to recover



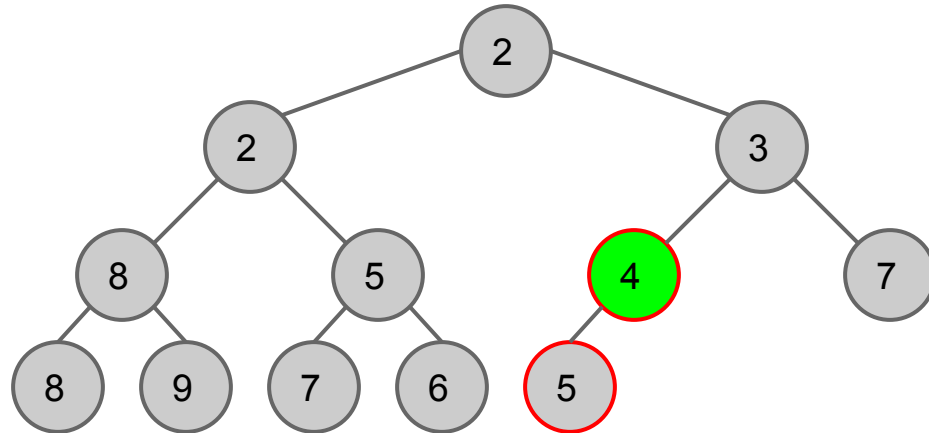
Binary Heap

- Remove minimum
 - replace the root with the last node at the bottom
 - heap property violated; do *sift-down* to recover



Binary Heap

- Remove minimum
 - replace the root with the last node at the bottom
 - heap property violated; do *sift-down* to recover

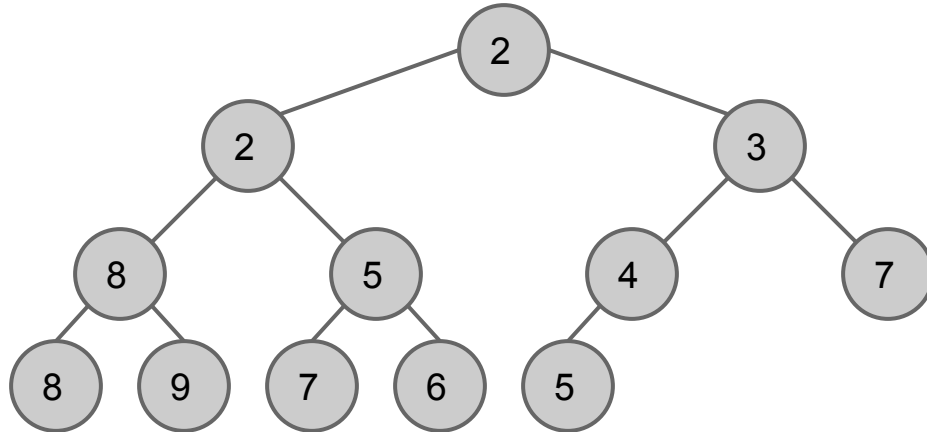


no smaller children. stop

Binary Heap

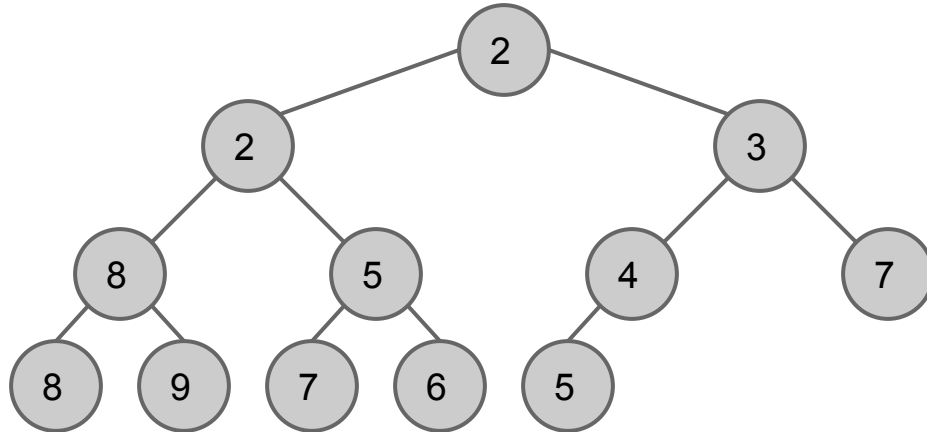
- Remove minimum

- replace the root with the last node at the bottom
- heap property violated; do *sift-down* to recover
- recovered



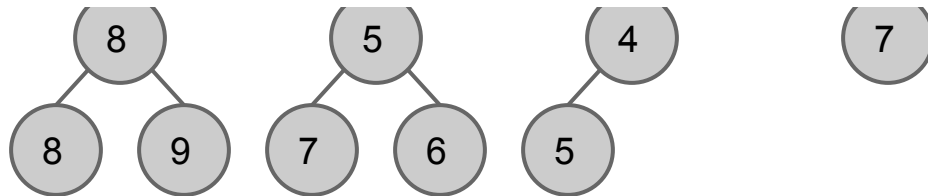
Binary Heap

- Remove minimum
 - Time complexity: $O(\log(N))$

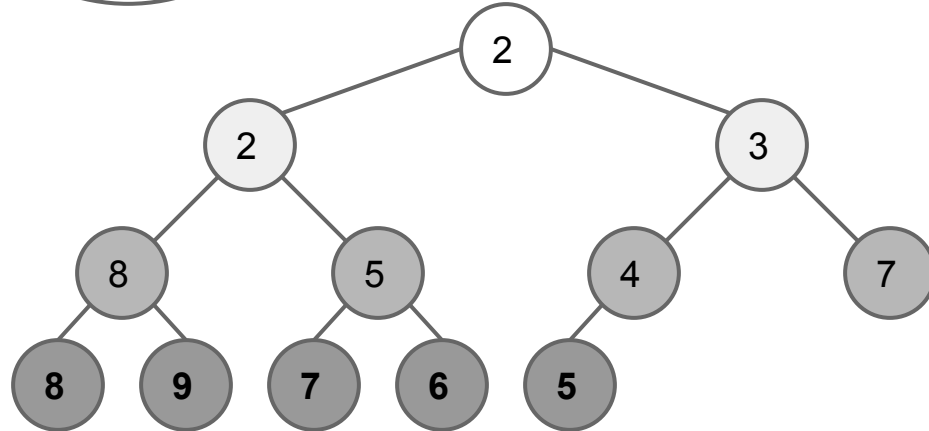
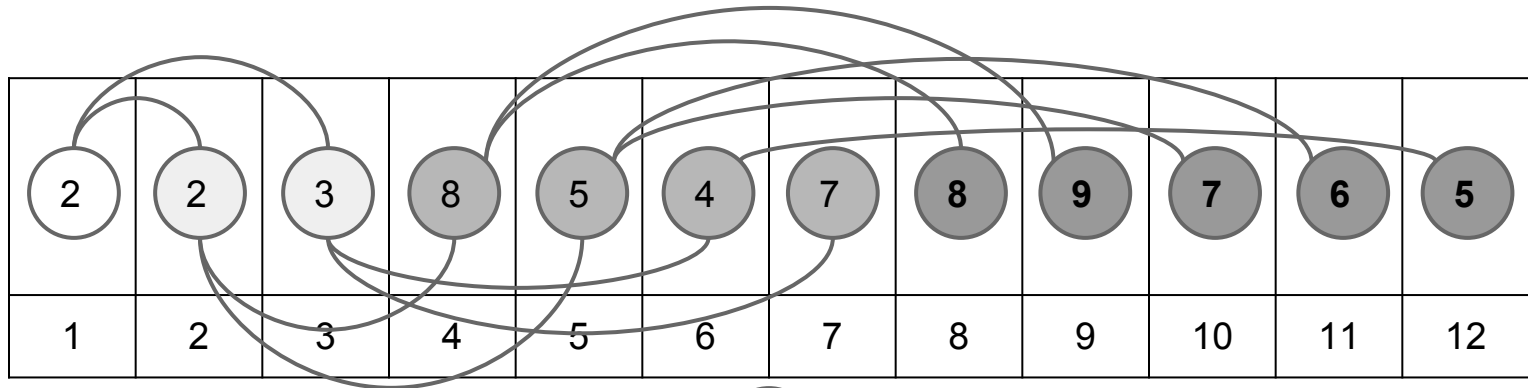


Binary Heap

- Implement in array
 - $\text{array}[1..N]$
 - root: $\text{array}[1]$
 - for $\text{array}[i]$,
 - left child: $\text{array}[i * 2]$
 - right child: $\text{array}[i * 2 + 1]$

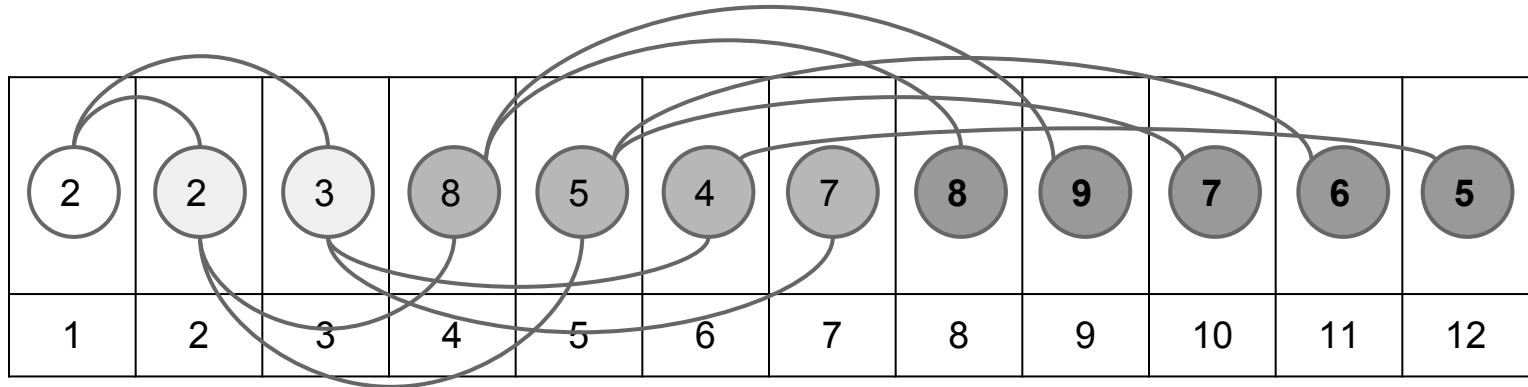


Binary Heap



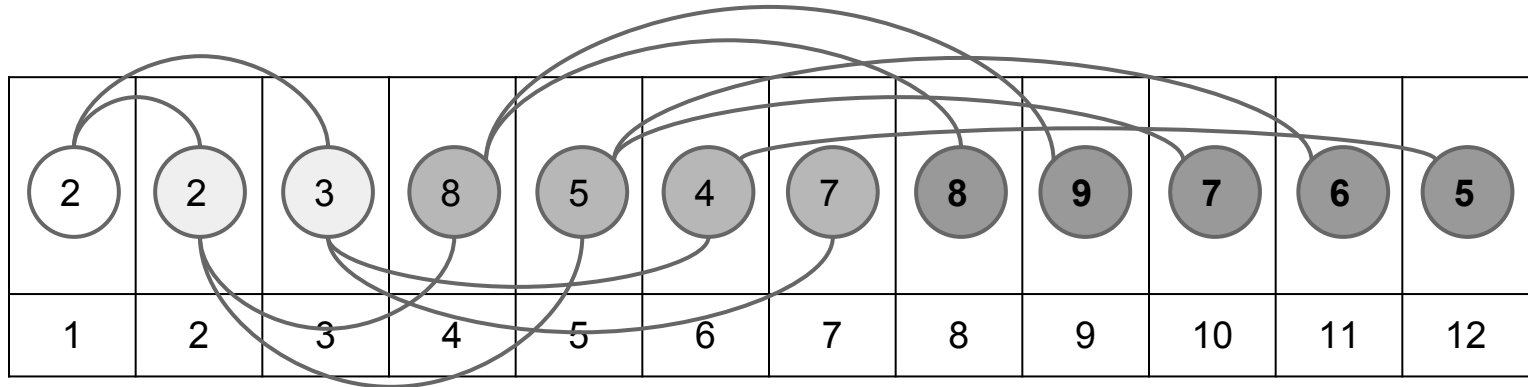
Binary Heap

- for $\text{array}[i]$,
 - left child: $\text{array}[i * 2]$
 - right child: $\text{array}[i * 2 + 1]$
 - parent: ?



Binary Heap

- for $\text{array}[i]$,
 - left child: $\text{array}[i * 2]$
 - right child: $\text{array}[i * 2 + 1]$
 - parent: $\text{array}[i / 2]$



Binary Heap

- C++ STL: `priority_queue`
- Max-heap,
 - all nodes are **less than or equal to** parent

```
#include <queue>
...
std::priority_queue<int> q;
q.push(4); q.push(7); q.push(3);
int x = q.top(); //x = 7
q.pop(); x = q.top(); //x = 4
```

Binary Heap

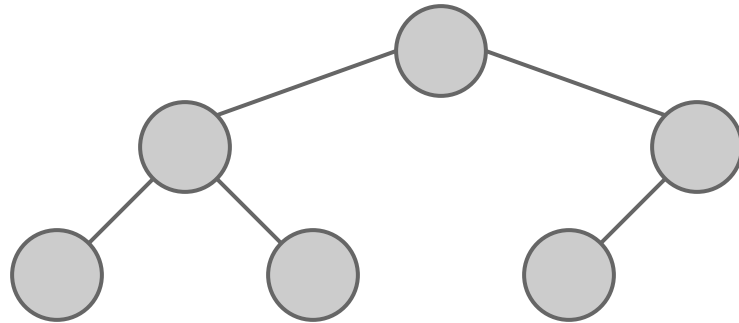
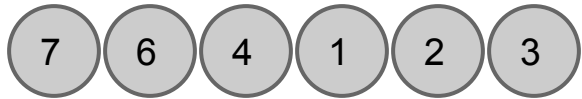
- Heapsort
 - insert everything into heap: $O(N\log(N))$
 - repeatedly remove minimum: $O(N\log(N))$
 - overall: $O(N\log(N))$

Binary Heap

- Heapsort
 - build heap: $O(N)$
 - repeatedly remove minimum: $O(N\log(N))$
 - overall: $O(N\log(N))$

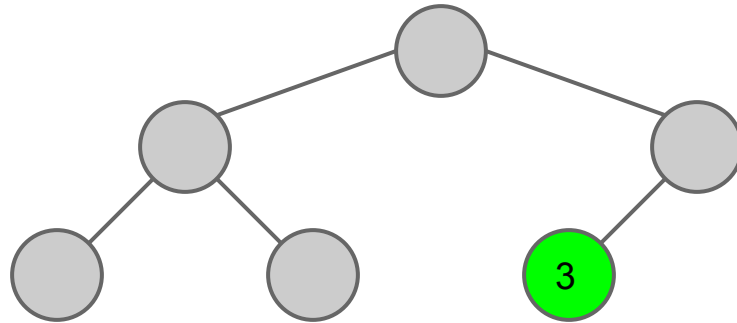
Binary Heap

- $O(N)$ build heap
- insert from the lowest level, then sift-down



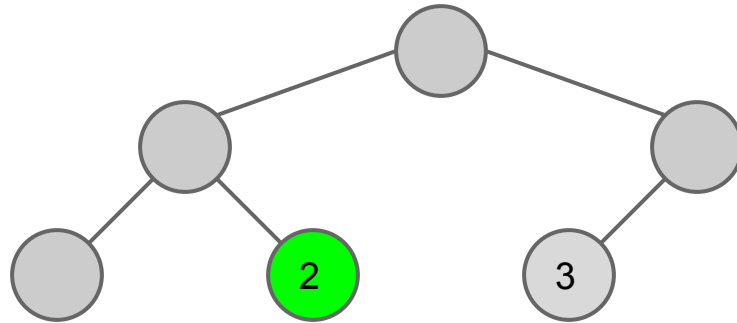
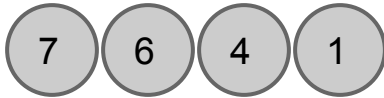
Binary Heap

- $O(N)$ build heap
- insert from the lowest level, then sift-down



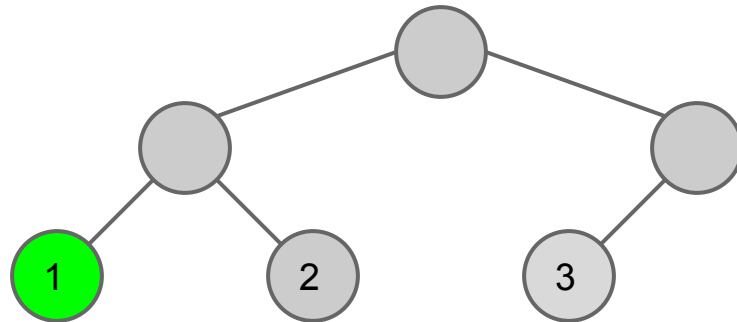
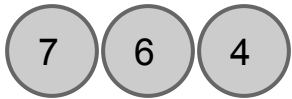
Binary Heap

- $O(N)$ build heap
- insert from the lowest level, then sift-down



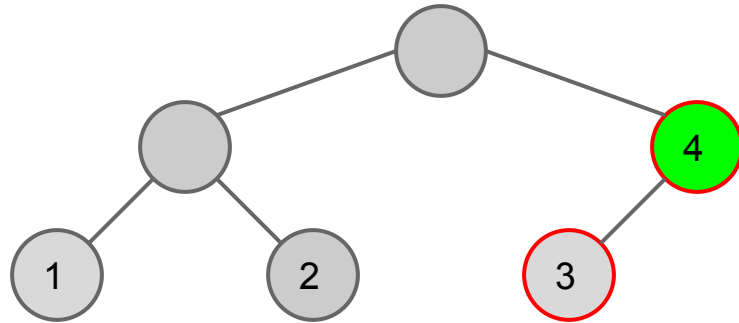
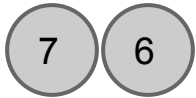
Binary Heap

- $O(N)$ build heap
- insert from the lowest level, then sift-down



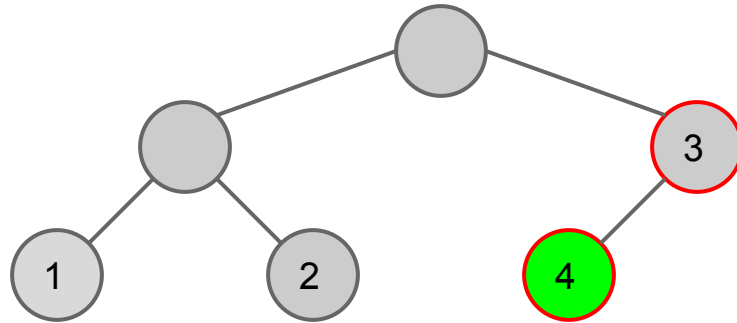
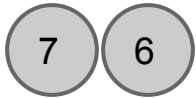
Binary Heap

- $O(N)$ build heap
- insert from the lowest level, then sift-down



Binary Heap

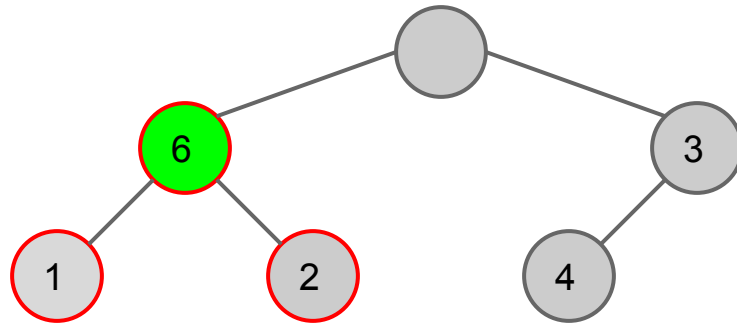
- $O(N)$ build heap
- insert from the lowest level, then sift-down



Binary Heap

- $O(N)$ build heap
- insert from the lowest level, then sift-down

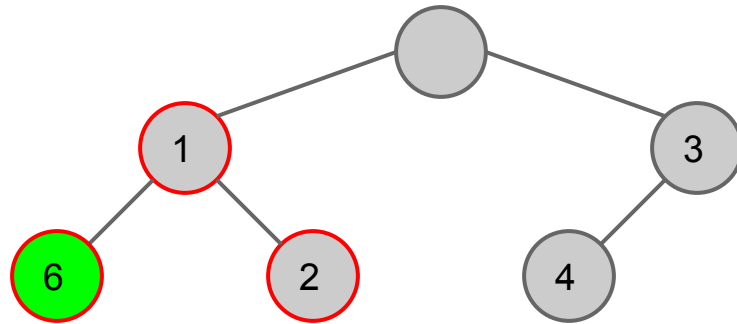
7



Binary Heap

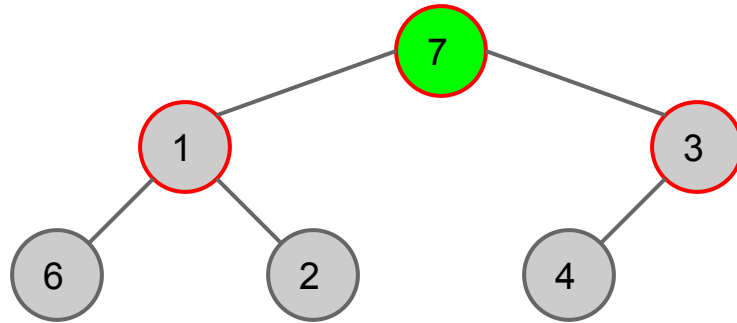
- $O(N)$ build heap
- insert from the lowest level, then sift-down

7



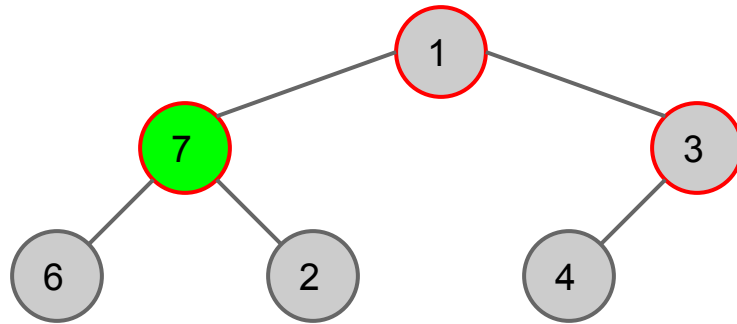
Binary Heap

- $O(N)$ build heap
- insert from the lowest level, then sift-down



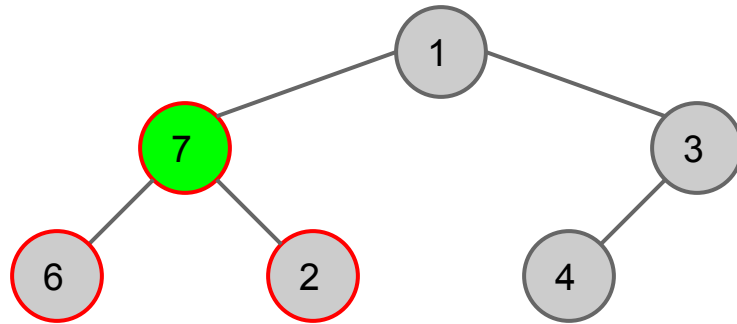
Binary Heap

- $O(N)$ build heap
- insert from the lowest level, then sift-down



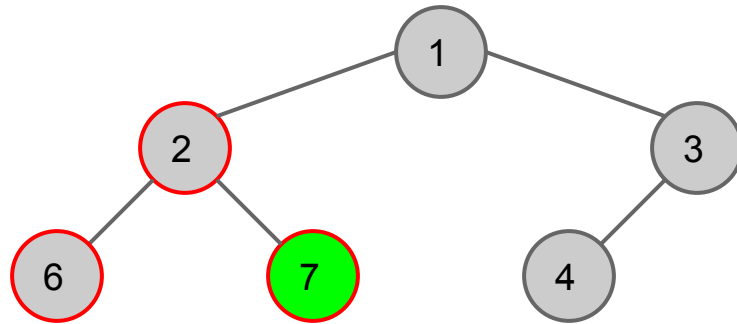
Binary Heap

- $O(N)$ build heap
- insert from the lowest level, then sift-down



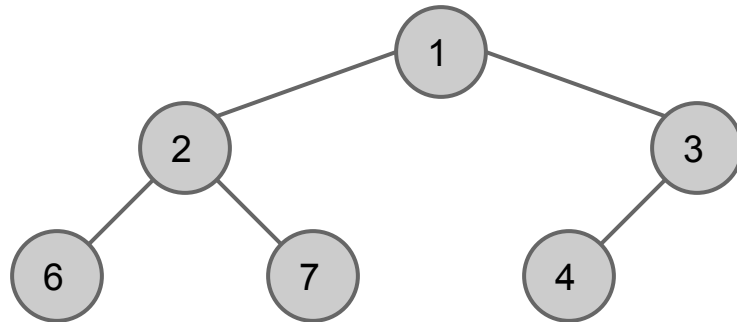
Binary Heap

- $O(N)$ build heap
- insert from the lowest level, then sift-down



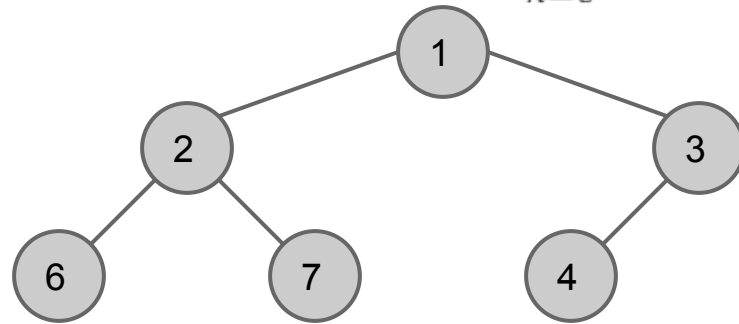
Binary Heap

- $O(N)$ build heap
- insert from the lowest level, then sift-down
- proof?



Binary Heap

- $O(N)$ build heap
- insert from the lowest level, then sift-down
- proof from wikipedia



$$\begin{aligned} \sum_{h=0}^{\lceil \log n \rceil} \frac{n}{2^{h+1}} O(h) &= O \left(n \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^{h+1}} \right) \\ &= O \left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n) \end{aligned}$$

Binary Heap - summary

- Time complexities:
 - insert : $O(\log(N))$
 - remove minimum : $O(\log(N))$
 - build heap : $O(N)$
- Space complexity:
 - $O(N)$

	Binary heap	Leftist heap	Binomial heap
merge heap	$O(N)$	$O(\log(N))$	$O(\log(N))$

Data Structures (II)

- Binary Heap
- Binary Search Tree
- Hash Table

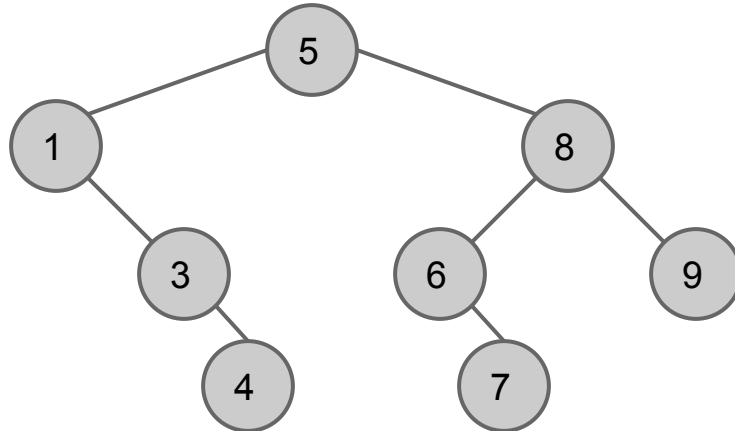
Binary Search Tree

- binary search is so fast

	Insert	Find	Remove
Plain array	$O(1)$	$O(N)$	$O(N)$
Sorted array	$O(N)$	$O(\log(N))$	$O(N)$
Binary search tree	$O(\log(N))$ average	$O(\log(N))$ average	$O(\log(N))$ average

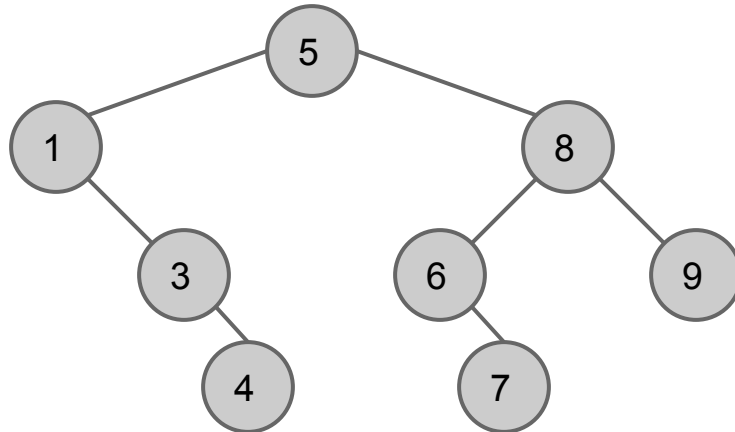
Binary Search Tree

- Binary tree
- Each node
 - $>$ all left descendants
 - $<$ all right descendants



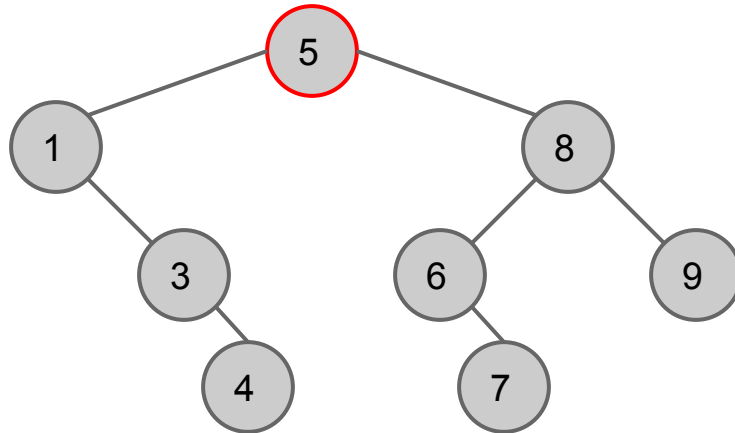
Binary Search Tree

- find a node
 - is '6' in the tree?
 - binary search



Binary Search Tree

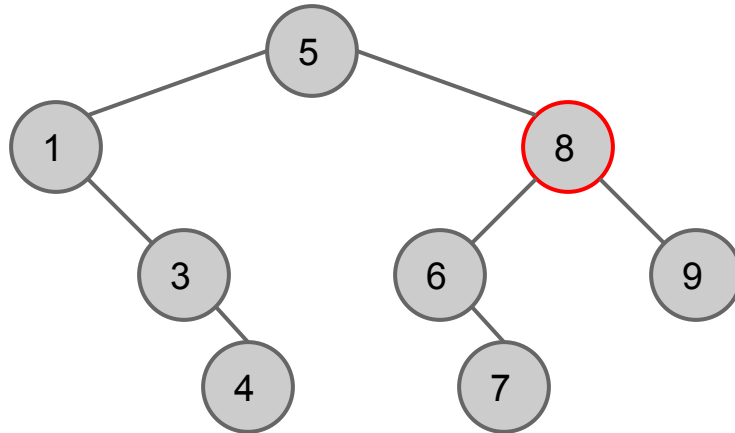
- find a node
 - is '6' in the tree?
 - binary search



'6' is not here.
'6' won't be on the left, so
let's go to the right

Binary Search Tree

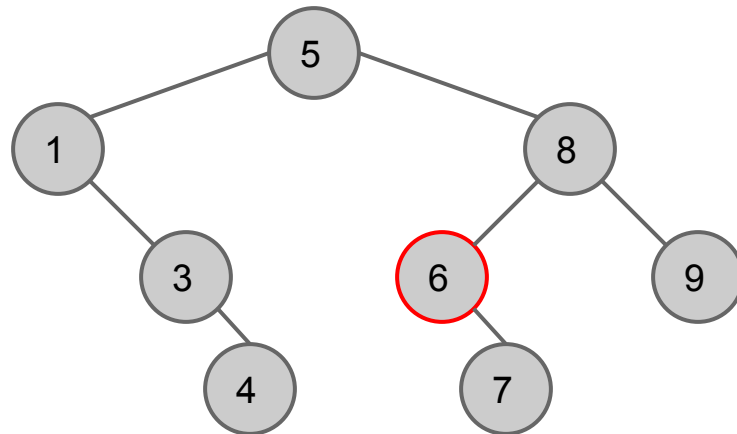
- find a node
 - is '6' in the tree?
 - binary search



'6' is not here either.
'6' won't be on the right, so
let's go to the left

Binary Search Tree

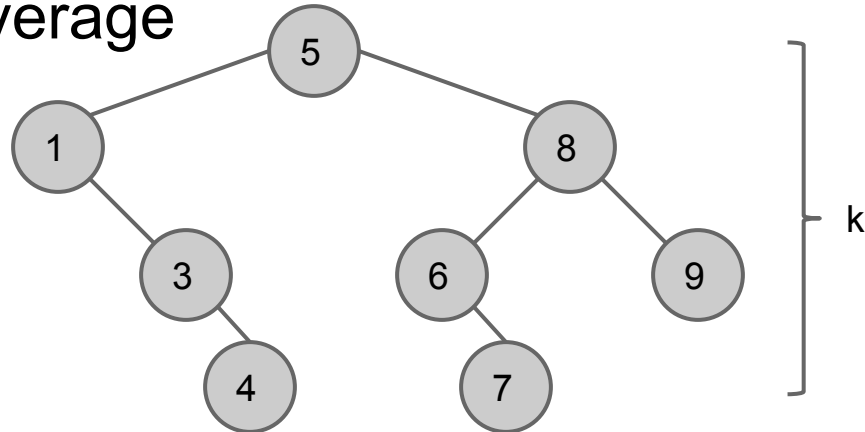
- find a node
 - is '6' in the tree?
 - binary search



'6' is in the tree!

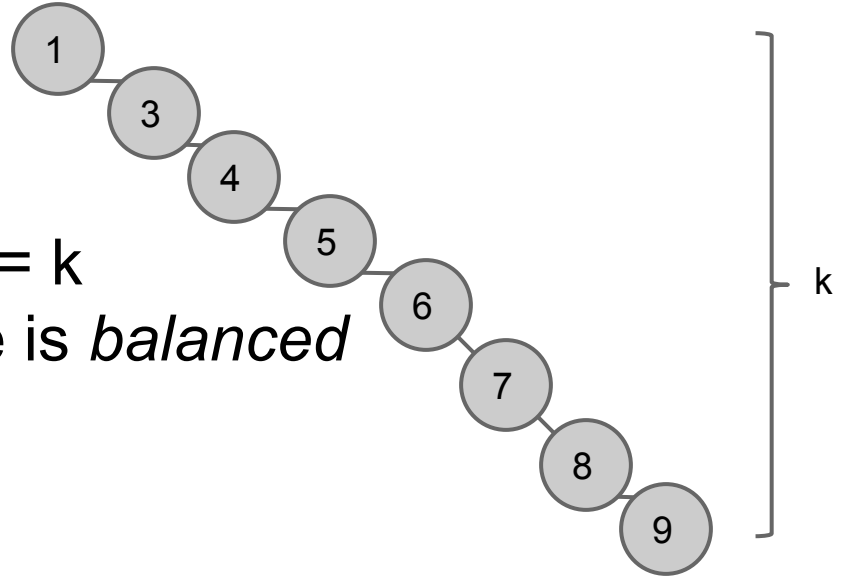
Binary Search Tree

- find a node
 - Time complexity?
 - max. # of nodes visited = k
 - $k \approx \log(N)$ when the tree is *balanced*
 - $O(\log(N))$ average



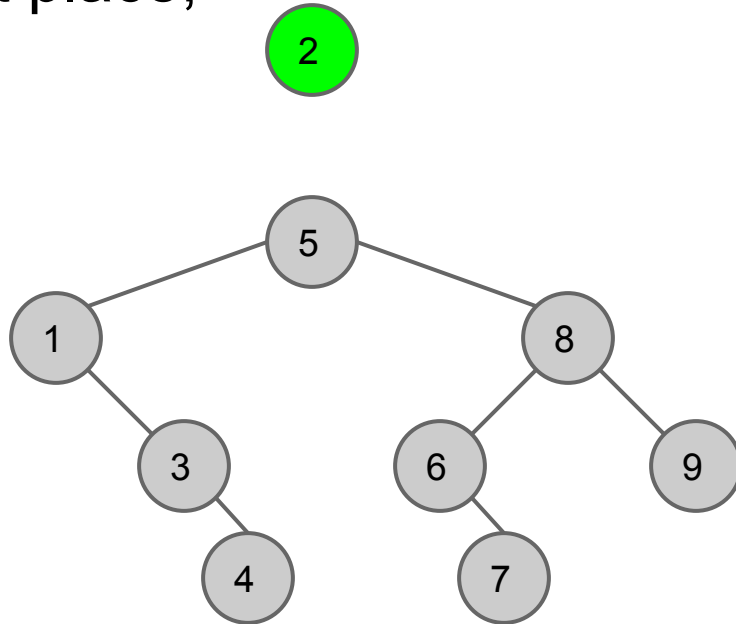
Binary Search Tree

- find a node
 - Time complexity?
 - max. # of nodes visited = k
 - $k \approx \log(N)$ when the tree is *balanced*
 - $O(\log(N))$ average
 - $O(N)$ worst case



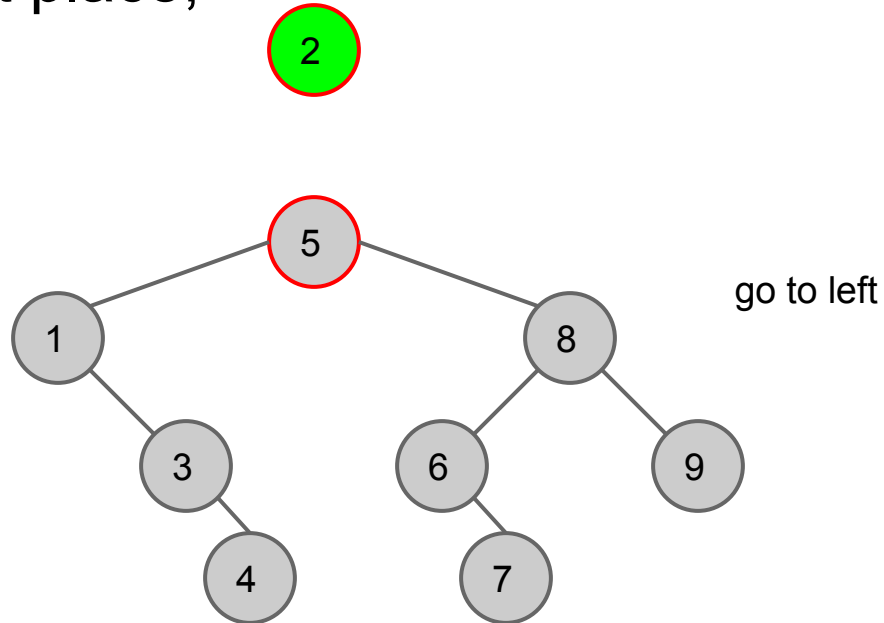
Binary Search Tree

- Insert a node
 - find the right place,
 - insert.



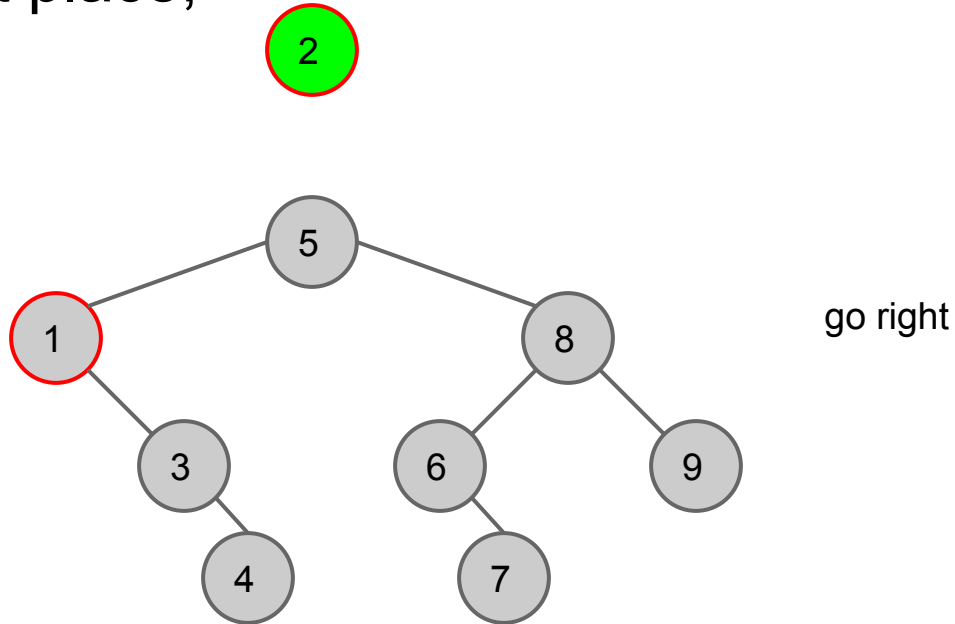
Binary Search Tree

- Insert a node
 - find the right place,
 - insert.



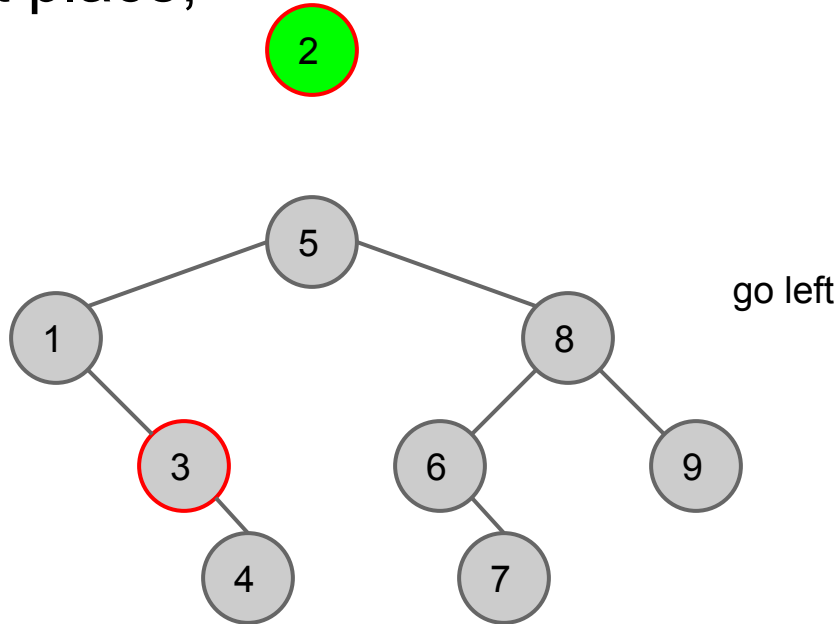
Binary Search Tree

- Insert a node
 - find the right place,
 - insert.



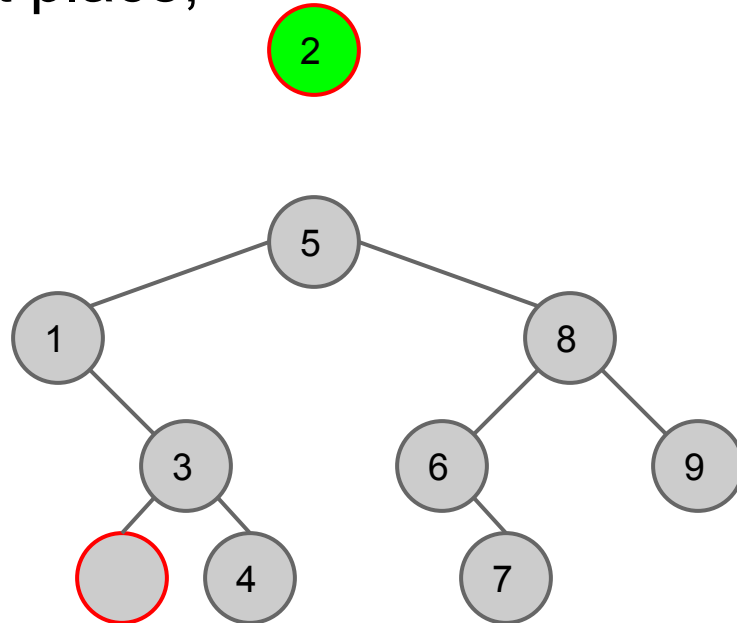
Binary Search Tree

- Insert a node
 - find the right place,
 - insert.



Binary Search Tree

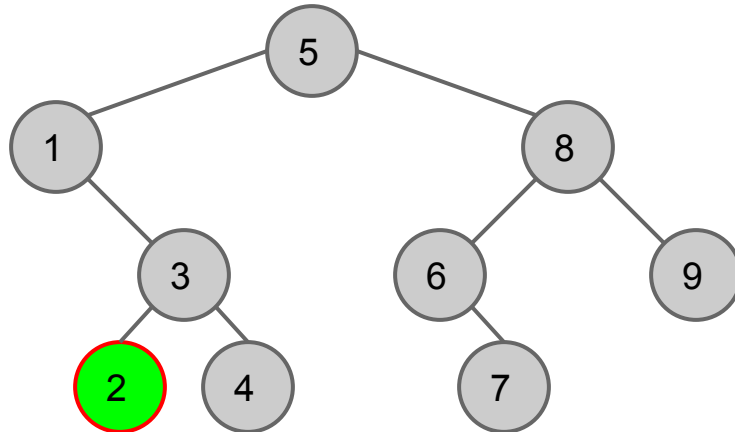
- Insert a node
 - find the right place,
 - insert.



left child is empty,
create new node

Binary Search Tree

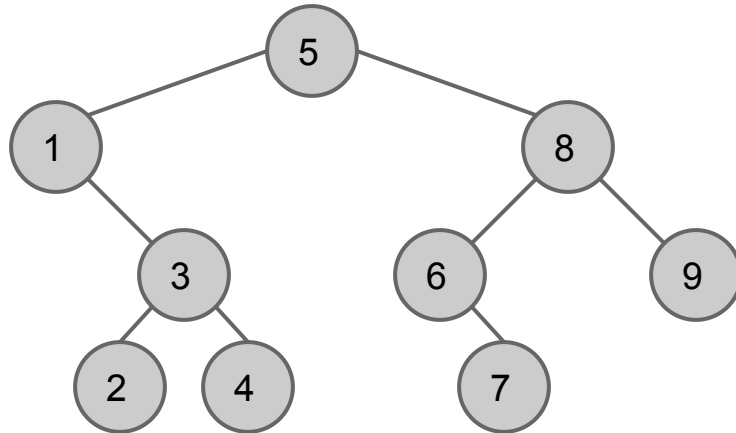
- Insert a node
 - find the right place,
 - insert.



insert.

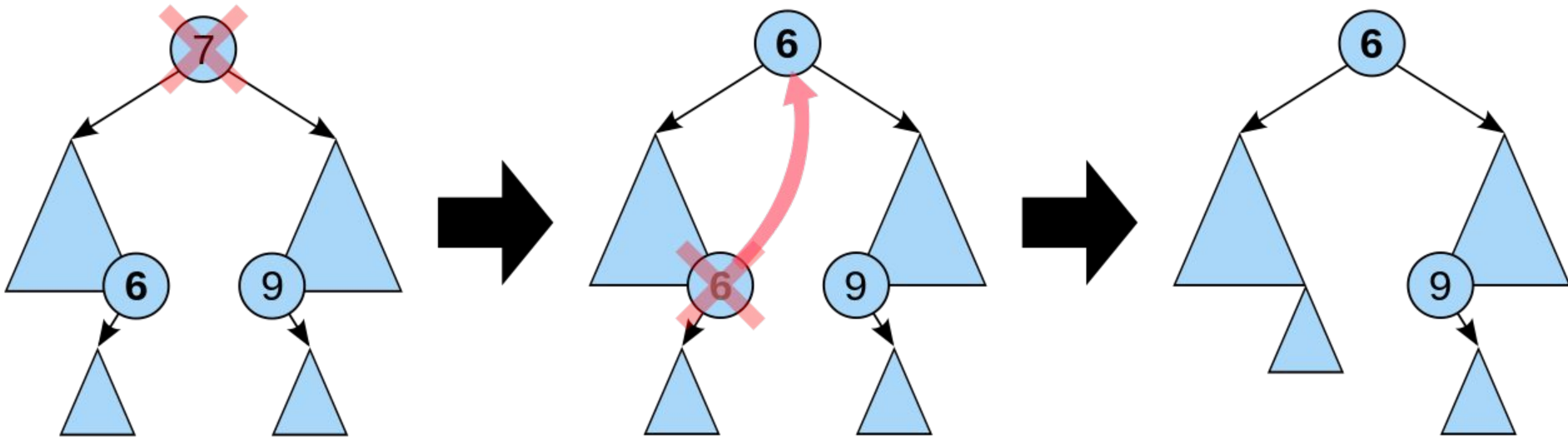
Binary Search Tree

- Insert a node
 - Time complexity
 - $O(\log(N))$ average
 - $O(N)$ worst case



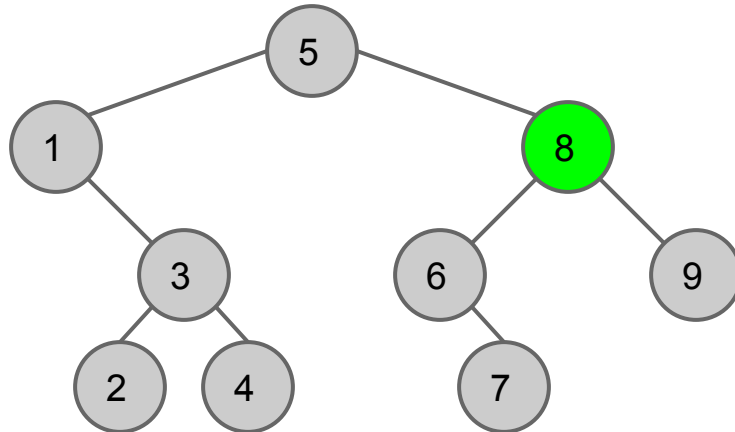
Binary Search Tree

- Remove a node
 - replace that node with the node *just* less than it
 - (image from wikipedia)



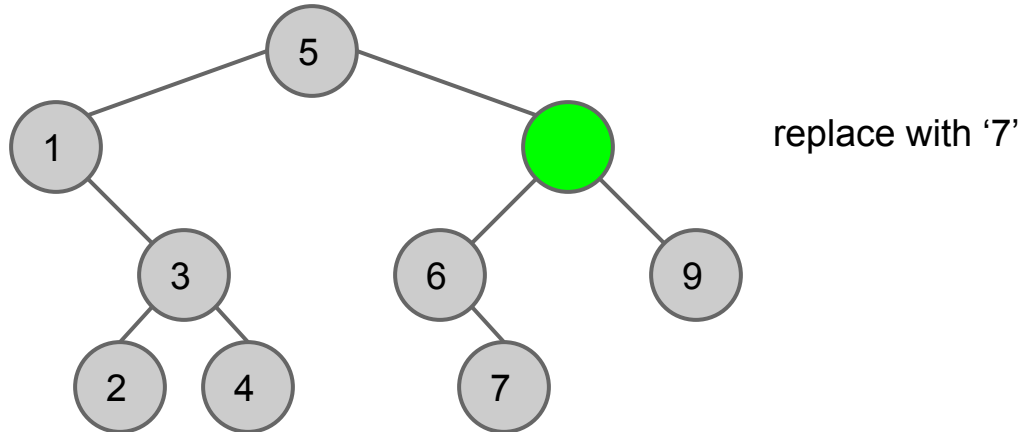
Binary Search Tree

- Remove a node
 - replace that node with the node *just* less than it



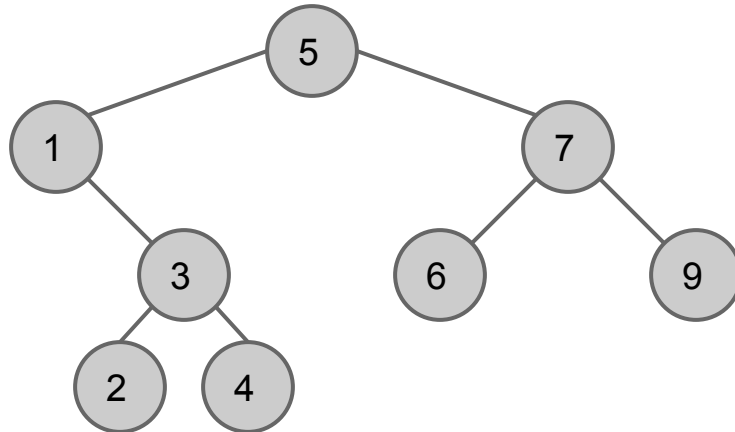
Binary Search Tree

- Remove a node
 - replace that node with the node *just* less than it



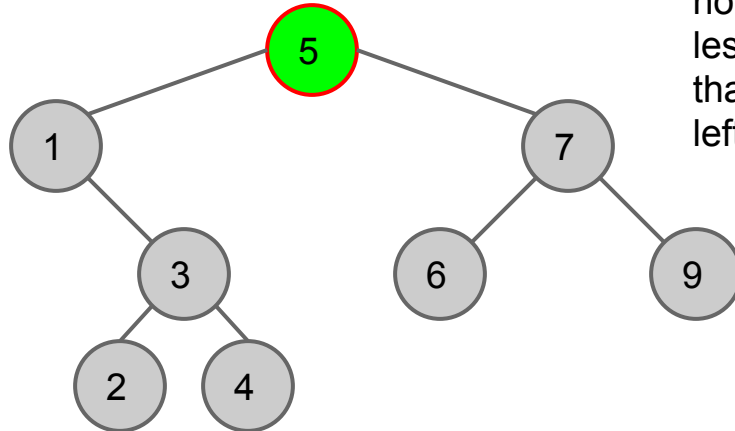
Binary Search Tree

- Remove a node
 - replace that node with the node *just* less than it



Binary Search Tree

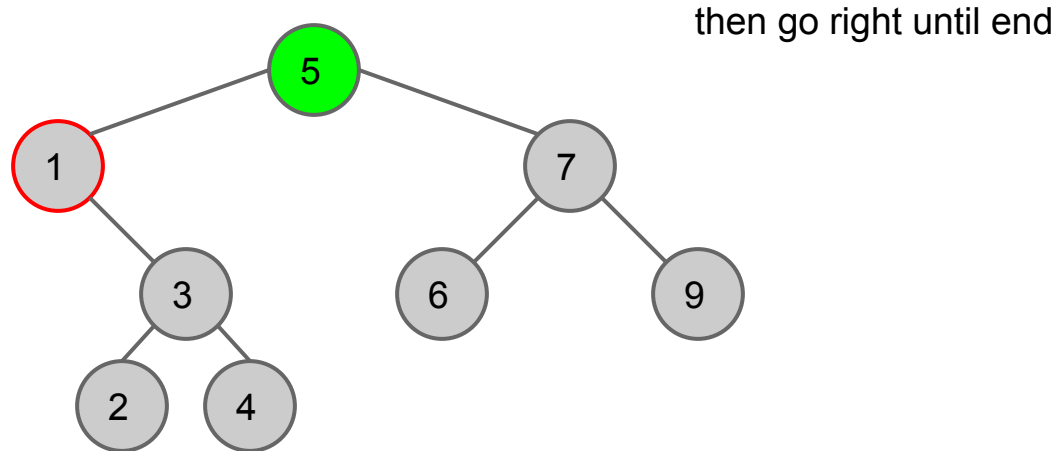
- Remove a node
 - replace that node with the node *just* less than it



how to find the node just less than '5'?
that node must be on the left. go left first

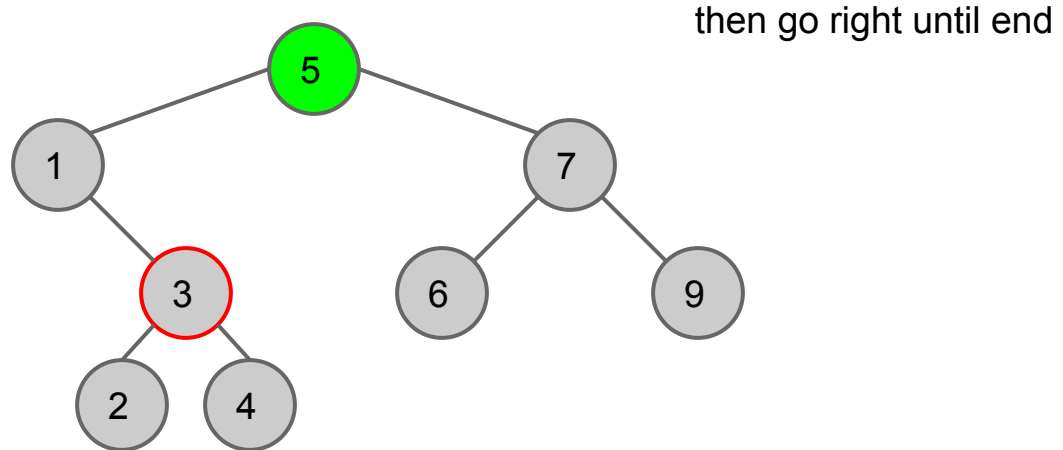
Binary Search Tree

- Remove a node
 - replace that node with the node *just* less than it



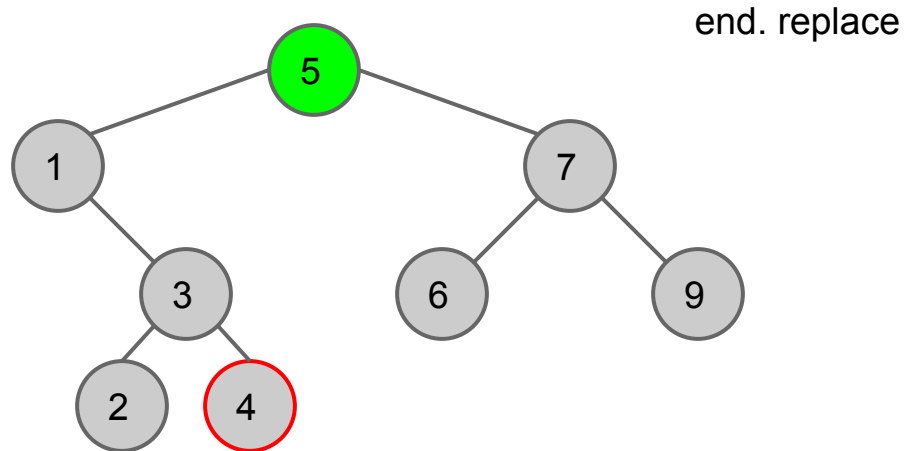
Binary Search Tree

- Remove a node
 - replace that node with the node *just* less than it



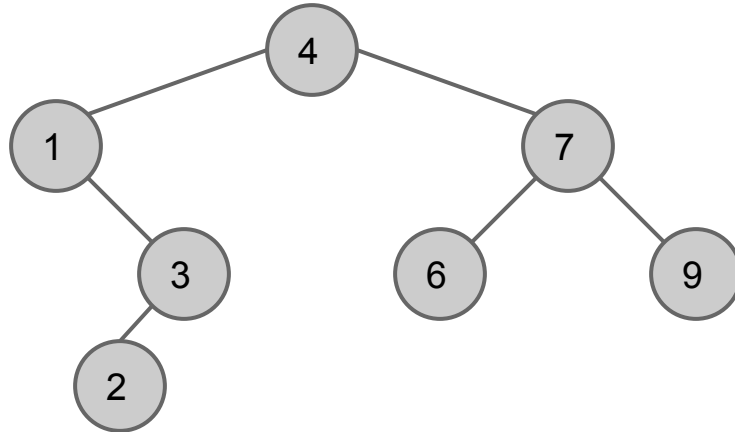
Binary Search Tree

- Remove a node
 - replace that node with the node *just* less than it



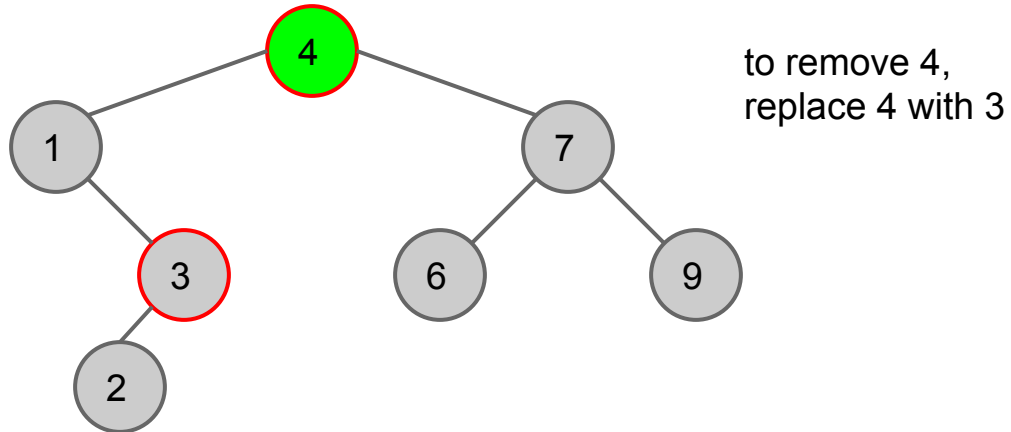
Binary Search Tree

- Remove a node
 - replace that node with the node *just* less than it



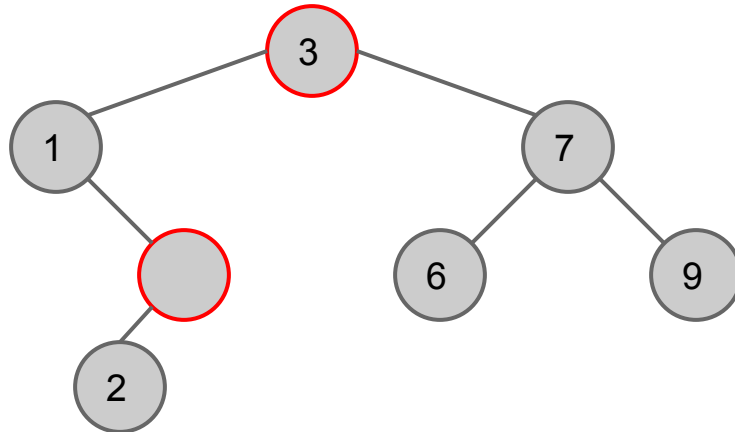
Binary Search Tree

- Remove a node
 - replace that node with the node *just* less than it



Binary Search Tree

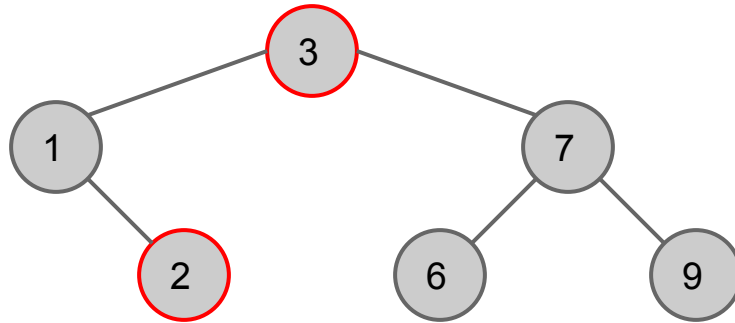
- Remove a node
 - replace that node with the node *just* less than it



remember to bring up 2

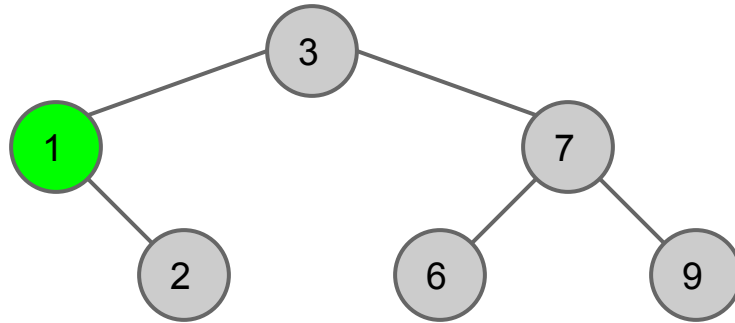
Binary Search Tree

- Remove a node
 - replace that node with the node *just* less than it



Binary Search Tree

- Remove a node
 - replace that node with the node *just* less than it

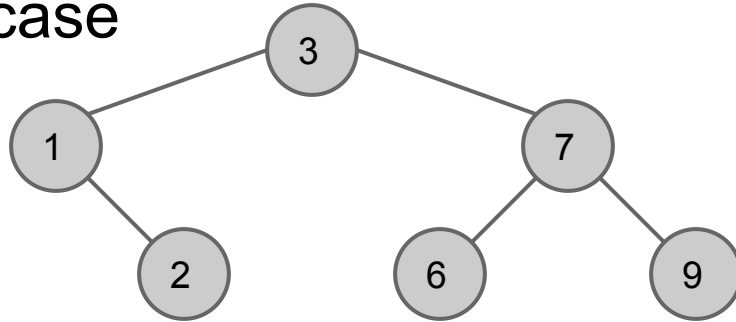


how to remove 1?

Binary Search Tree

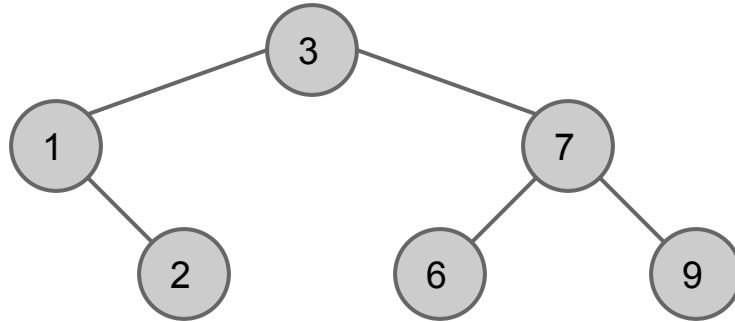
- Remove a node

- locate the node to be removed: $O(\log(N))$ average
- locate the node *just* less than it: $O(\log(N))$ average
- overall, $O(\log(N))$ average
- $O(N)$ worst case



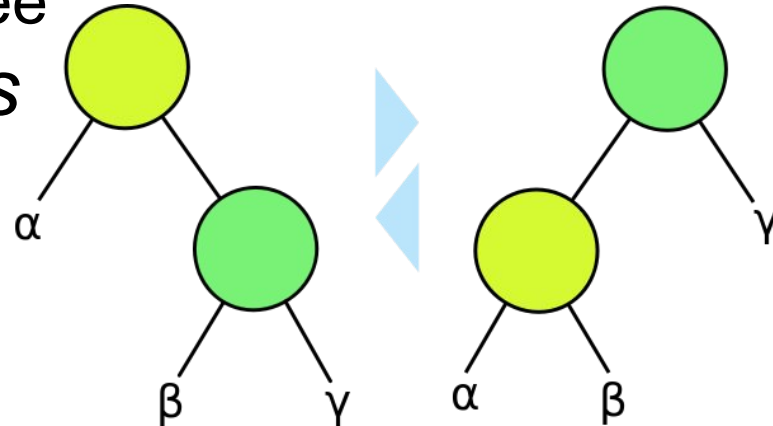
Binary Search Tree

- Summary
- Find, Insert, Remove
 - $O(\log(N))$ average
 - $O(N)$ worst case



Binary Search Tree

- How to get rid of $O(N)$ worst case?
- self-balancing binary search trees
 - AVL tree, Red-black tree, AA tree, Treap, Size-balanced tree, Splay tree
- make use of *tree rotations*
- $O(\log(N))$ worst case



Binary Search Tree

- Implementation

```
struct BST {  
    int value;  
    BST *left, *right;  
} *root, memory[100000], memory_i;
```

Binary Search Tree

- C++ STL: set, map
- Red-black tree internally, so worst case $O(\log(N))$

```
#include <set>
...
std::set<int> s;
s.insert(456);
int x = s.count(123); //0
s.erase(456);
```


Data structures (II)

- Binary Heap
- Binary Search Tree
- Hash Table

Hash Table

- Binary search tree is too complicated
- Hash table is much simpler, it is just an array

	Insert	Find	Remove
Plain array	$O(1)$	$O(N)$	$O(N)$
Sorted array	$O(N)$	$O(\log(N))$	$O(N)$
Binary search tree	$O(\log(N))$ average	$O(\log(N))$ average	$O(\log(N))$ average
Hash table	$O(1)$ average	$O(1)$ average	$O(1)$ average

Hash Table

- Array too short, value too big

972

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Hash Table

- Array too short, value too big
- *Hash* it



$$972 \% 17 = 3$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Hash Table

- Array too short, value too big
- *Hash* it

257

$$257 \% 17 = 2$$

			972													
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Hash Table

- Array too short, value too big
- *Hash* it

339

$$339 \% 17 = 16$$

		257	972													
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Hash Table

- Array too short, value too big
- *Hash* it

376

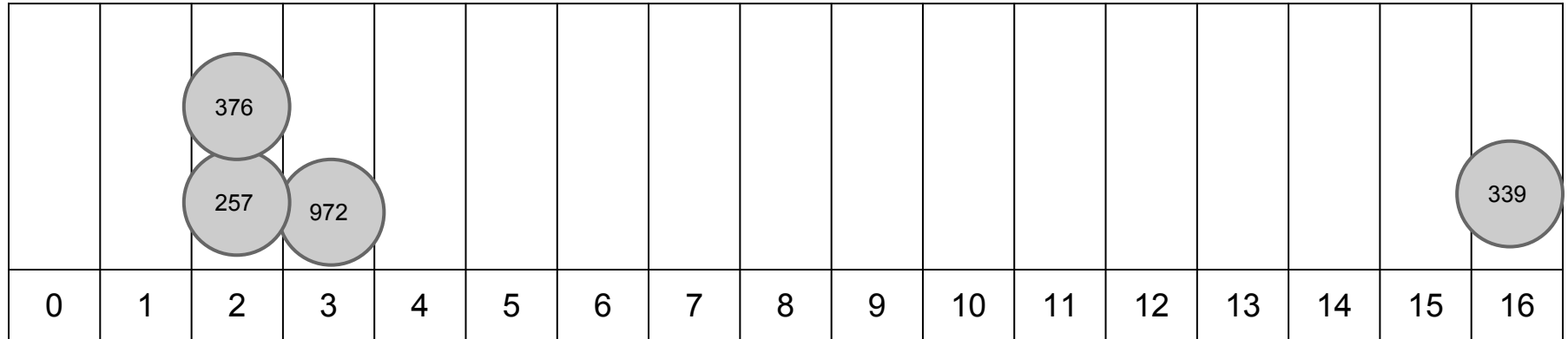
$$376 \% 17 = 2$$

oops occupied

		257	972													339
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Hash Table

- Hash collision resolution 1:
 - *separate chaining*
 - array of linked lists



Hash Table

- Hash collision resolution 2:
 - *open addressing*
 - find the first empty slot on the right
 - usually slower

		257	972	376												339
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Hash Table

- Insert
 - Time complexity
 - $O(1)$ average
 - $O(N)$ worst case
 - a good hash function gets better time

		257	972	376												339
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Hash Table

- Find (separate chaining)
 - traverse the linked list on the hashed slot

		376														
		257	972													339
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

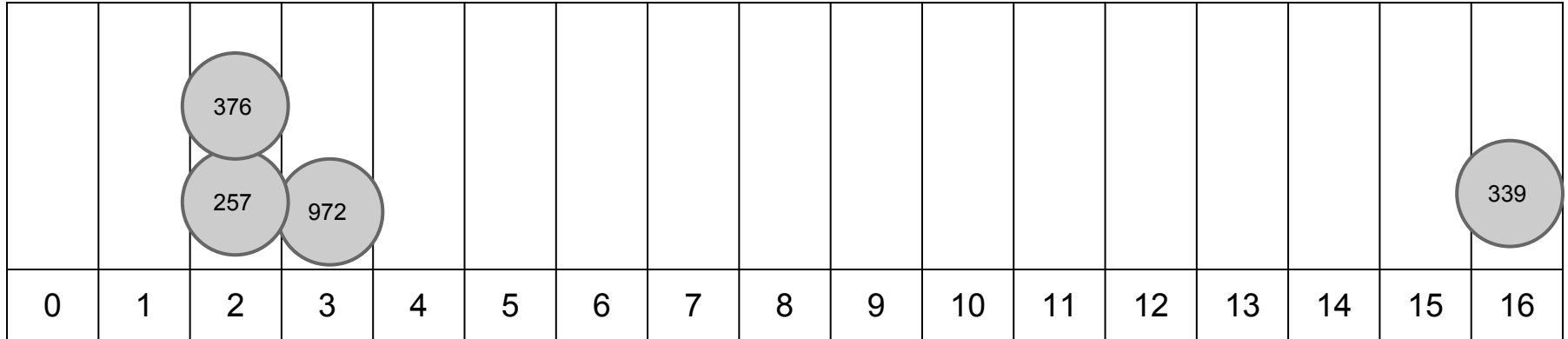
Hash Table

- Find (open addressing)
 - scan from the hashed slot to the right until the required value or an empty slot is found

		257	972	376												339
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Hash Table

- Remove (separate chaining)
 - remove from the linked list on the hashed slot



Hash Table

- Remove (open addressing)
 - set the slot value to -1 (or other dummy value)

		257	972	-1												339
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Hash Table

- Summary
- Insert, Find, Remove
 - $O(1)$ average
 - $O(N)$ worst case