Data Structure (II)

Steven Lau

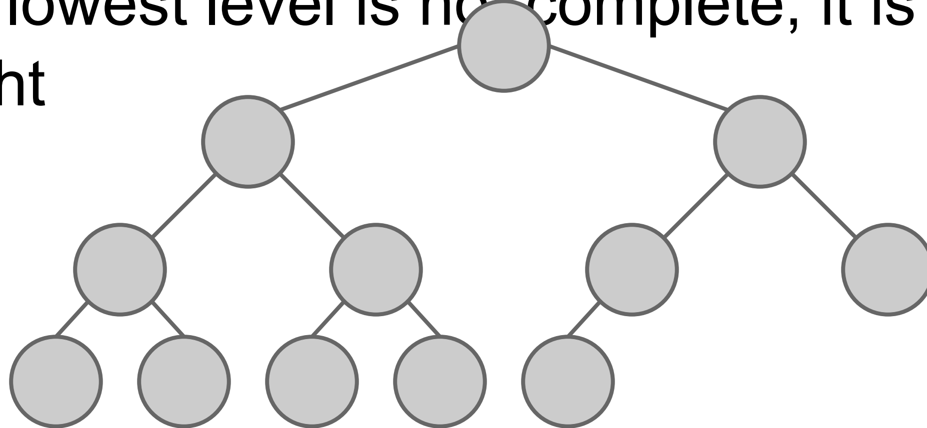
7th Feb, 2015 10:00am

Content

- Binary Heap
- Binary Search Tree
- Hash Table
- XOR Linked List

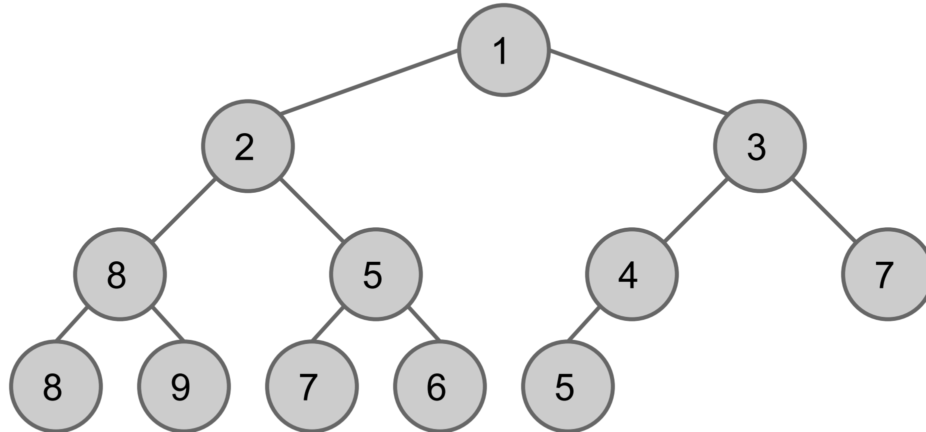
Binary Heap properties

- Complete binary tree
  - all levels (except possibly the lowest level) are fully filled
  - if the lowest level is not complete, it is filled from left to right



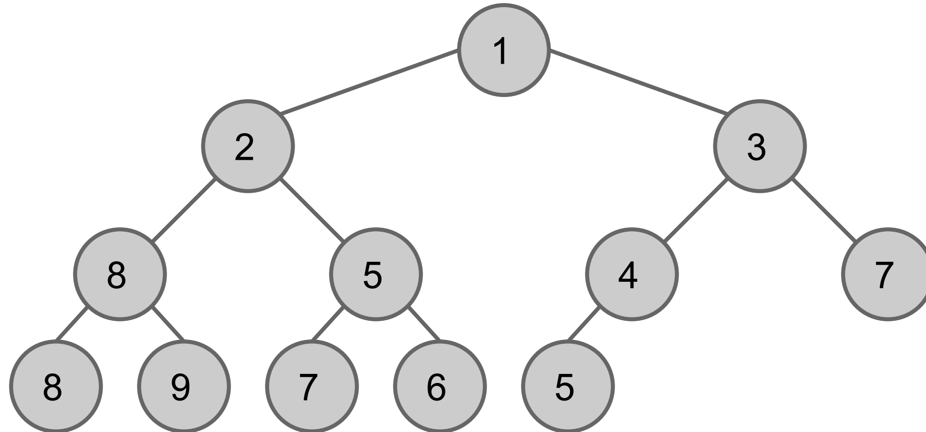
Binary Heap properties

- Heap property
  - All nodes are less than or equal to each of its children



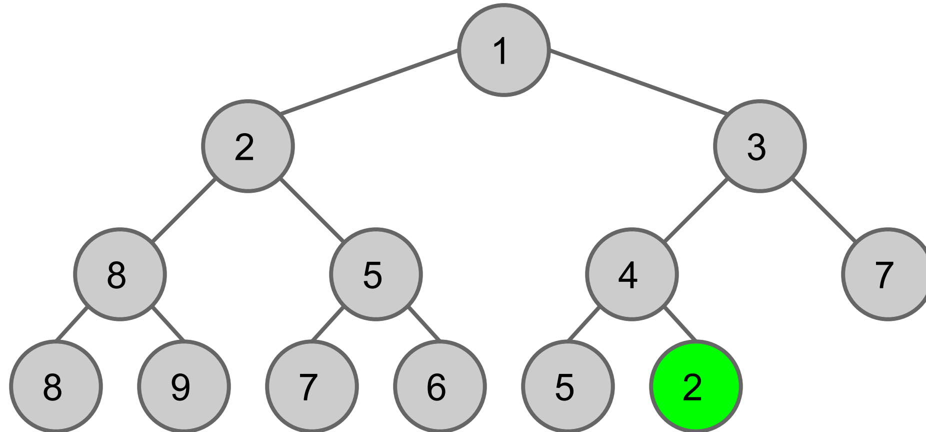
Binary Heap

- Insert a node
  - add a node in the lowest level



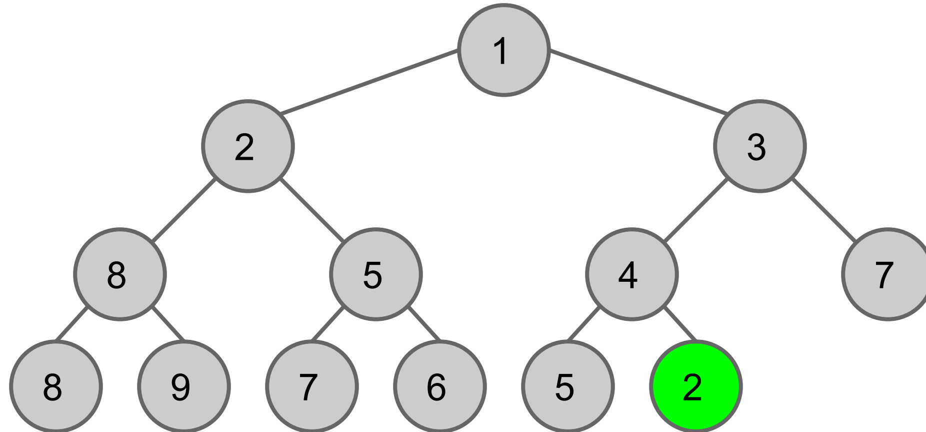
Binary Heap

- Insert a node
  - add a node in the lowest level



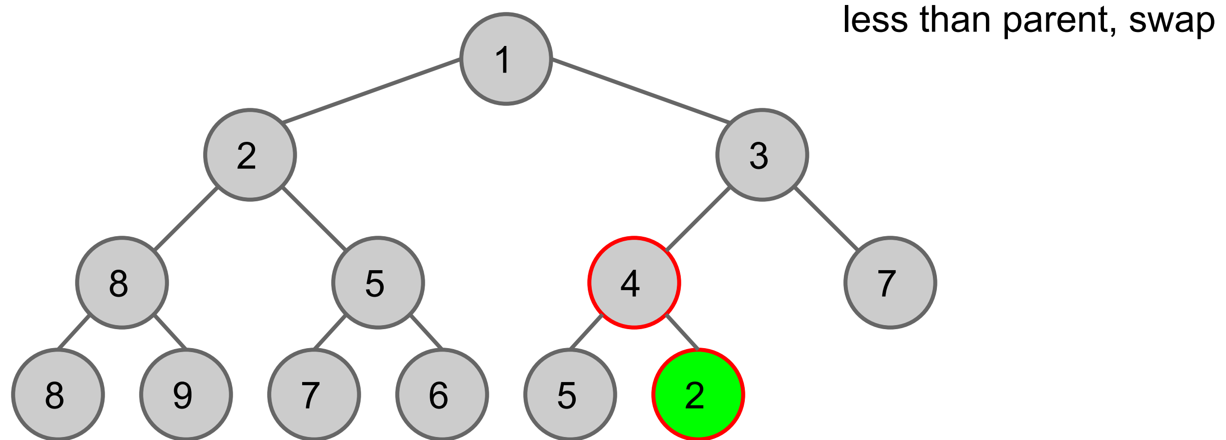
Binary Heap

- Insert a node
  - add a node in the lowest level
  - heap property violated. do *sift-up* to recover



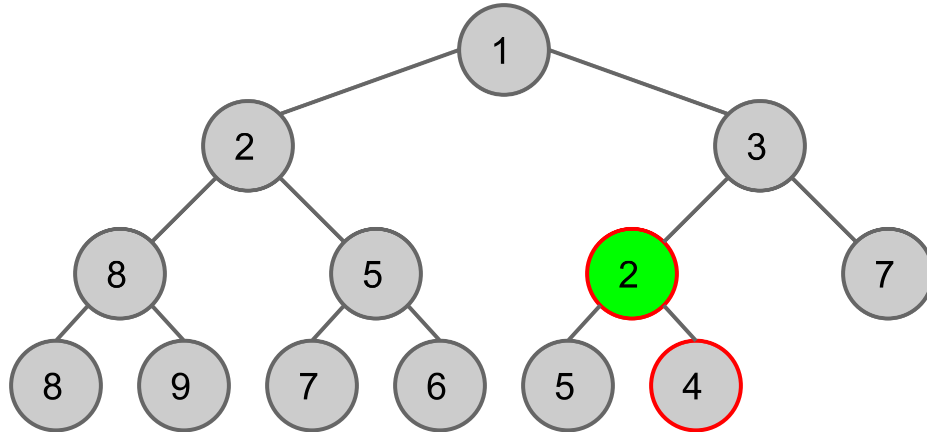
Binary Heap

- Insert a node
  - add a node in the lowest level
  - heap property violated, do *sift-up* to recover



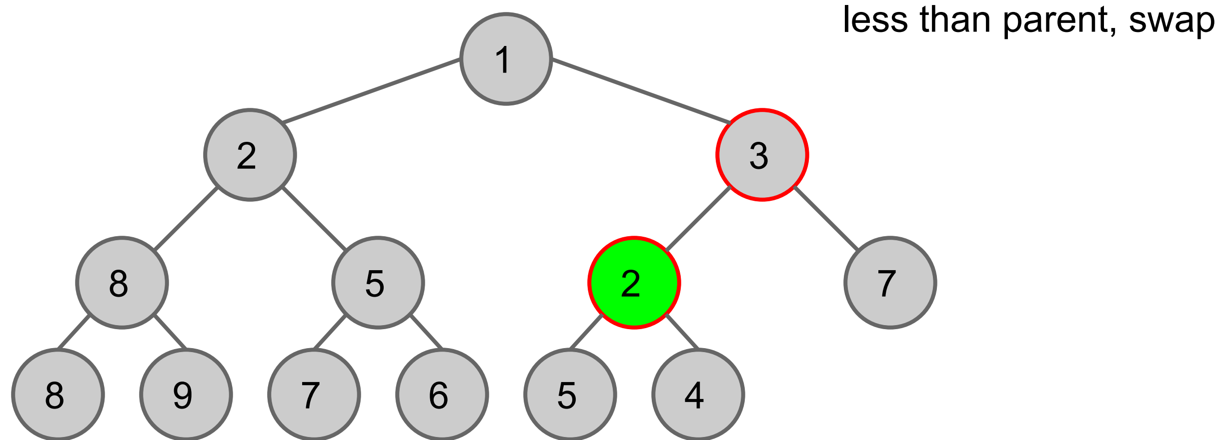
Binary Heap

- Insert a node
    - add a node in the lowest level
    - heap property violated, do *sift-up* to recover



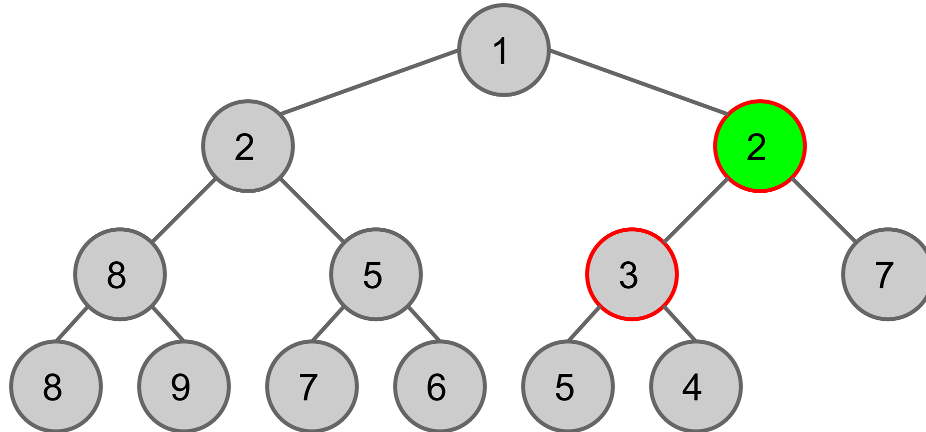
Binary Heap

- Insert a node
  - add a node in the lowest level
  - heap property violated, do *sift-up* to recover



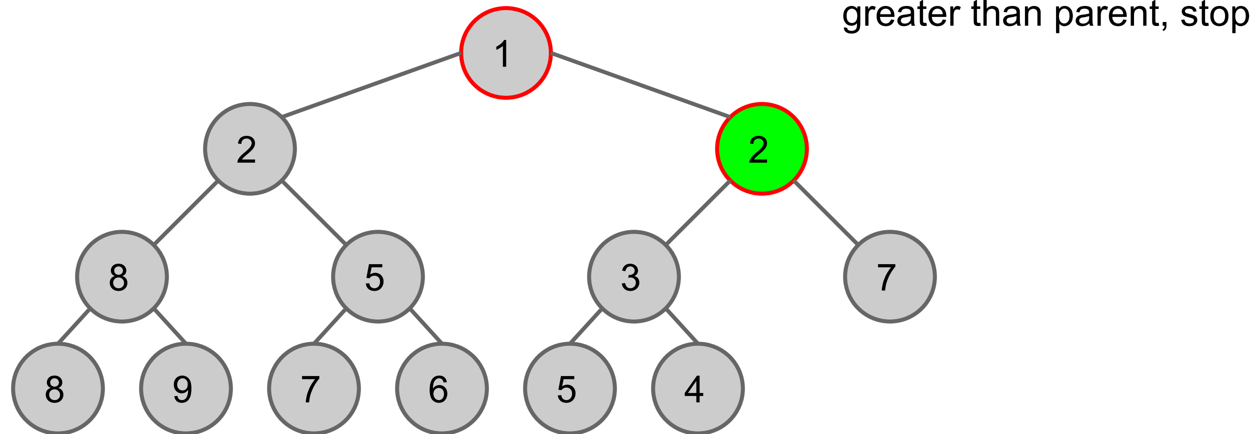
Binary Heap

- Insert a node
  - add a node in the lowest level
  - heap property violated, do *sift-up* to recover



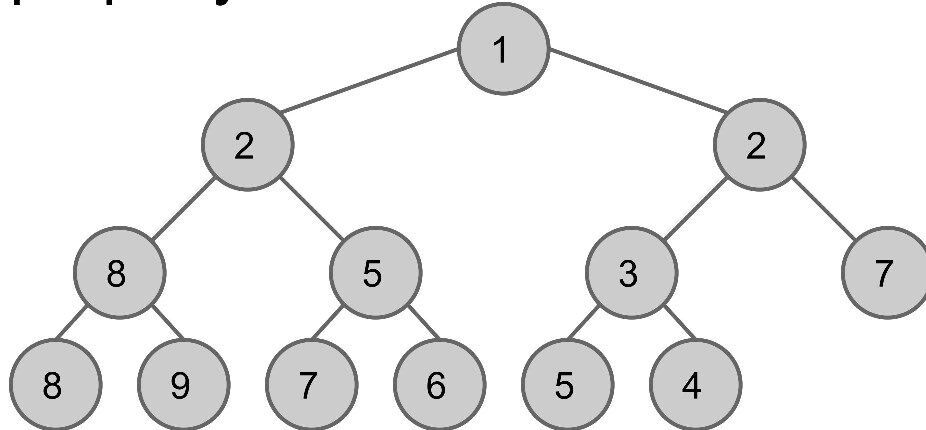
Binary Heap

- Insert a node
  - add a node in the lowest level
  - heap property violated, do *sift-up* to recover



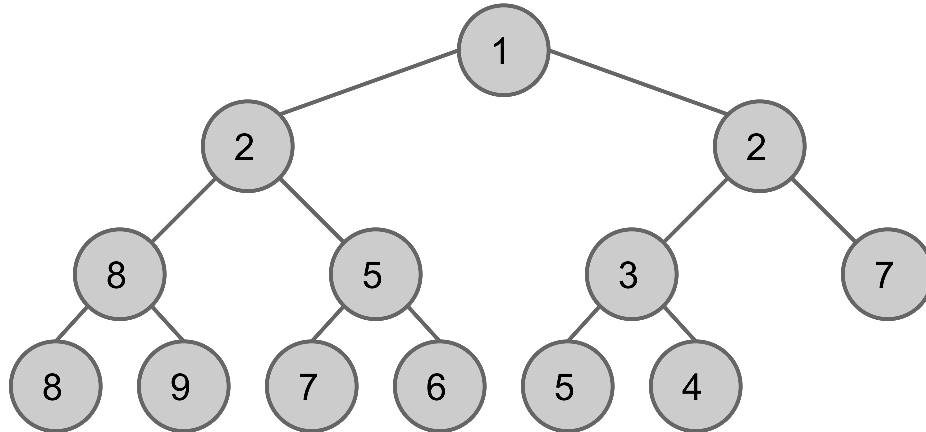
Binary Heap

- Insert a node
  - add a node in the lowest level
  - heap property violated, do *sift-up* to recover
  - heap property recovered



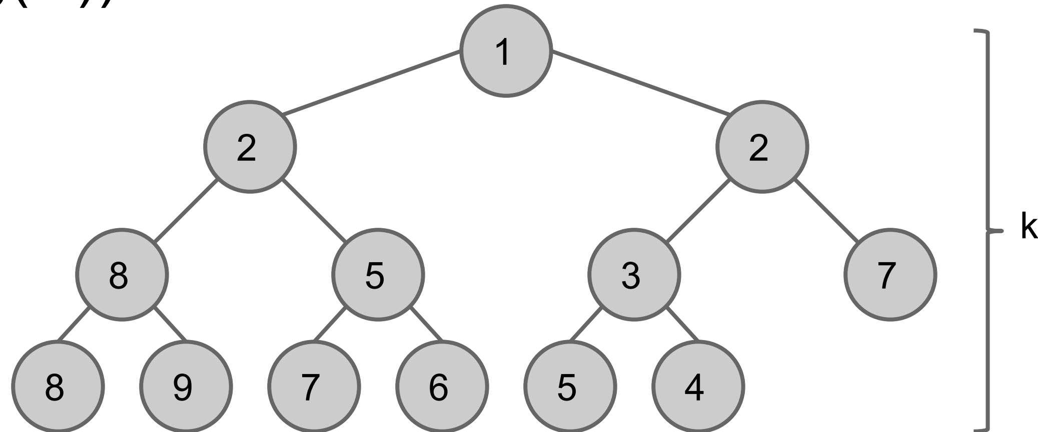
Binary Heap

- Insert a node
  - Time complexity?



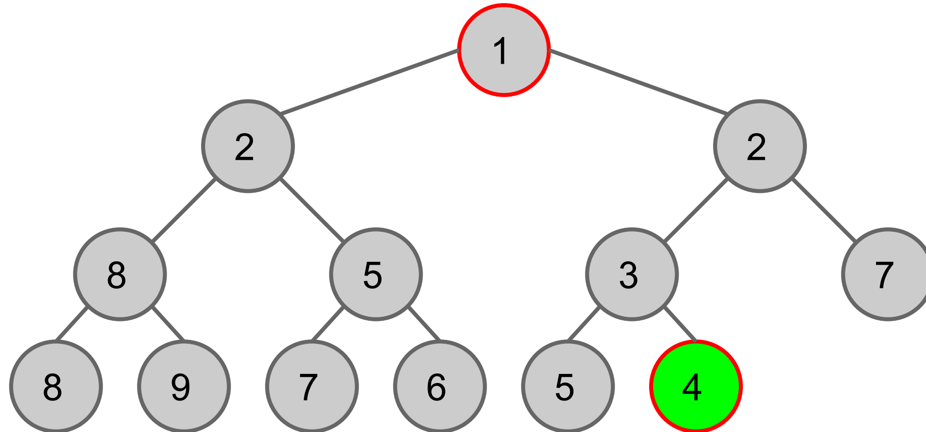
Binary Heap

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



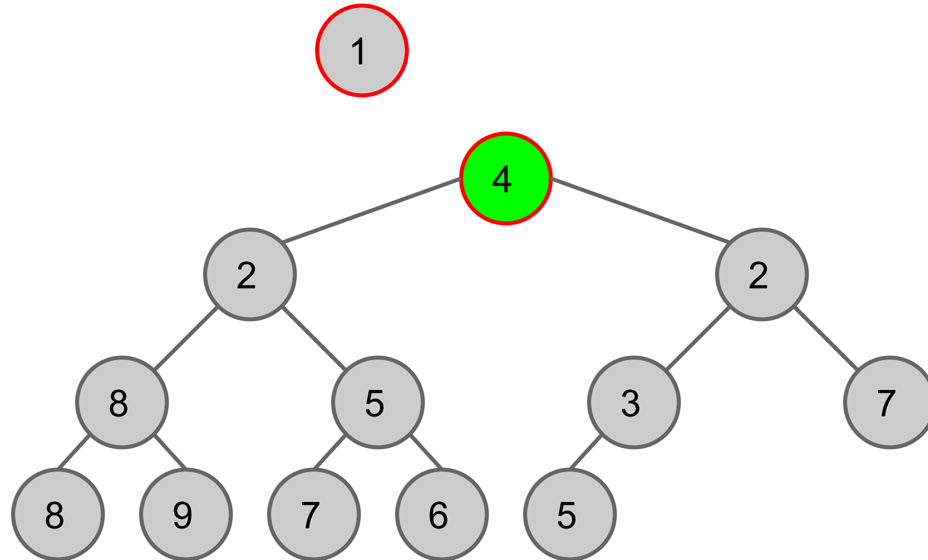
Binary Heap

- Remove minimum
  - Replace the root with the last node in the lowest level



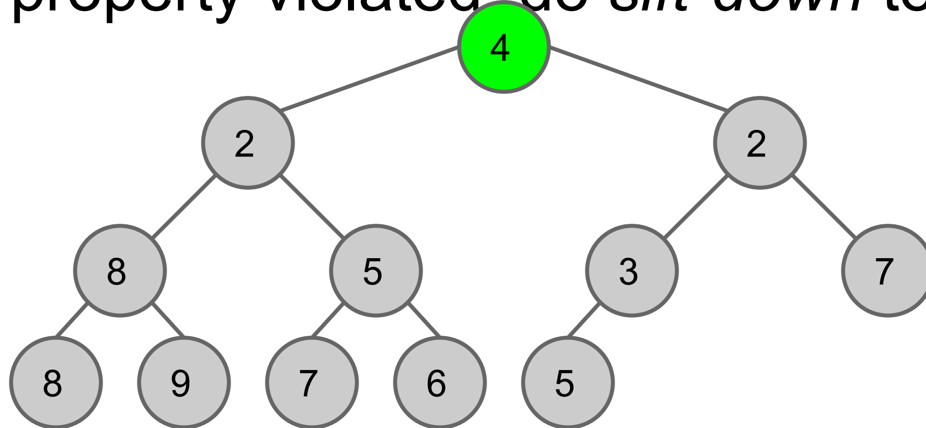
Binary Heap

- Remove minimum
  - Replace the root with the last node in the lowest level



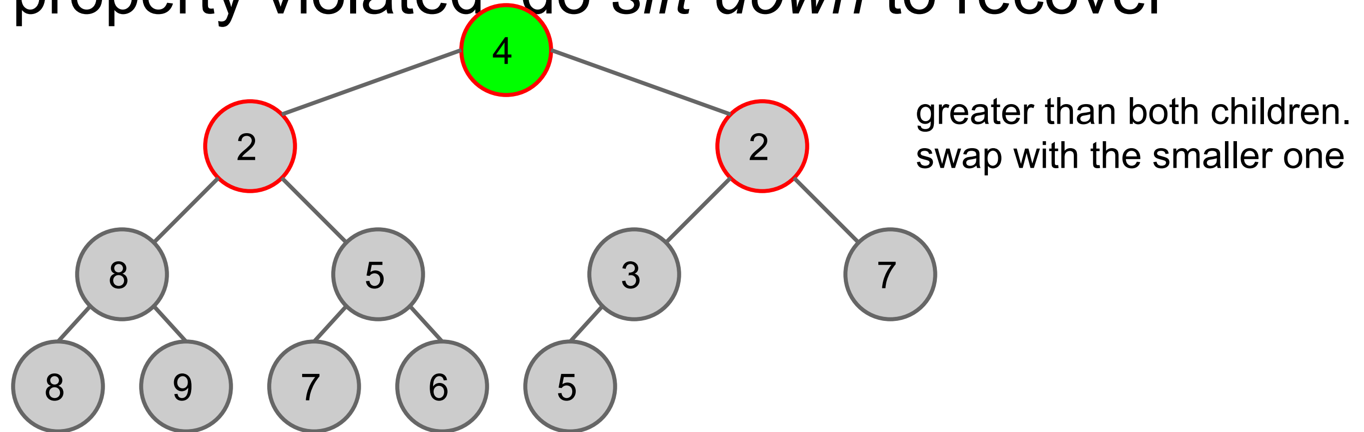
Binary Heap

- Remove minimum
  - Replace the root with the last node in the lowest level
  - heap property violated, do *sift-down* to recover



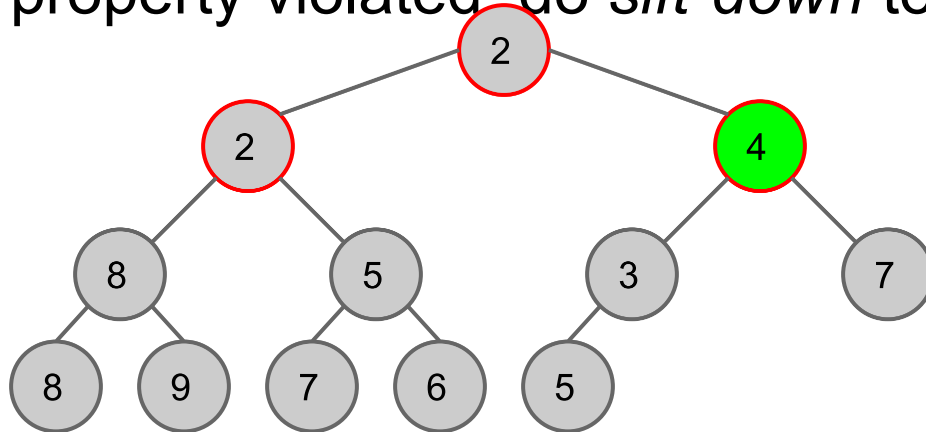
Binary Heap

- Remove minimum
  - Replace the root with the last node in the lowest level
  - heap property violated, do *sift-down* to recover



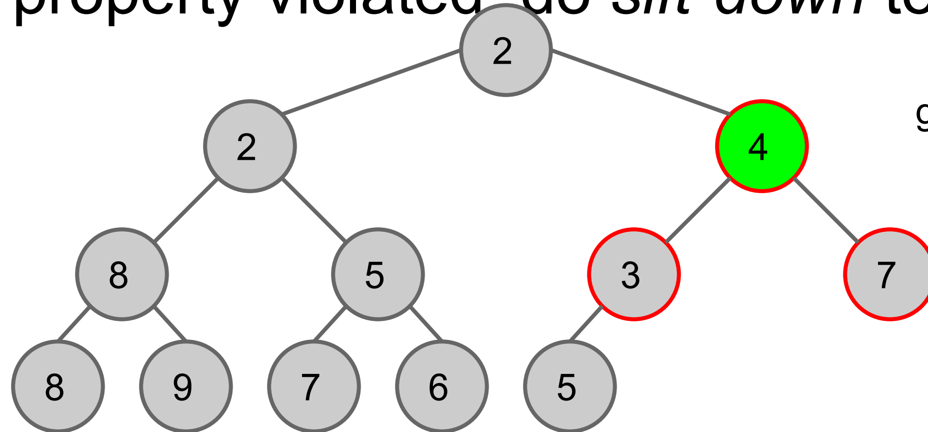
Binary Heap

- Remove minimum
  - Replace the root with the last node in the lowest level
  - heap property violated, do *sift-down* to recover



Binary Heap

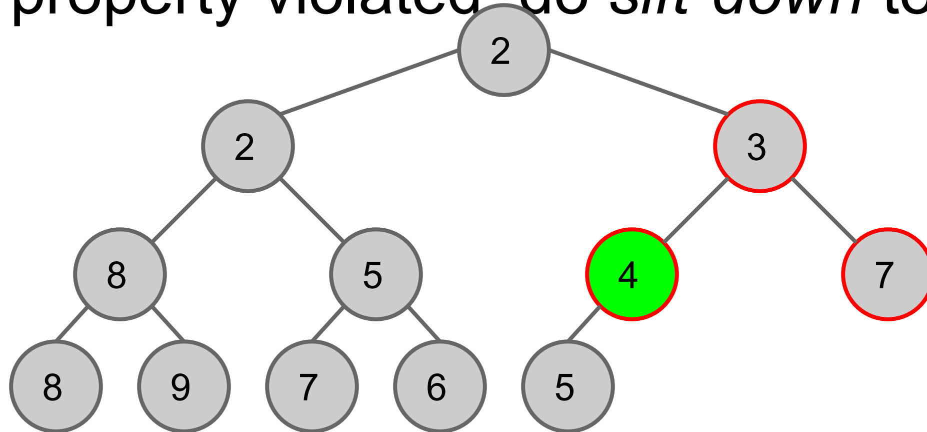
- Remove minimum
  - Replace the root with the last node in the lowest level
  - heap property violated, do *sift-down* to recover



greater than left child only. swap

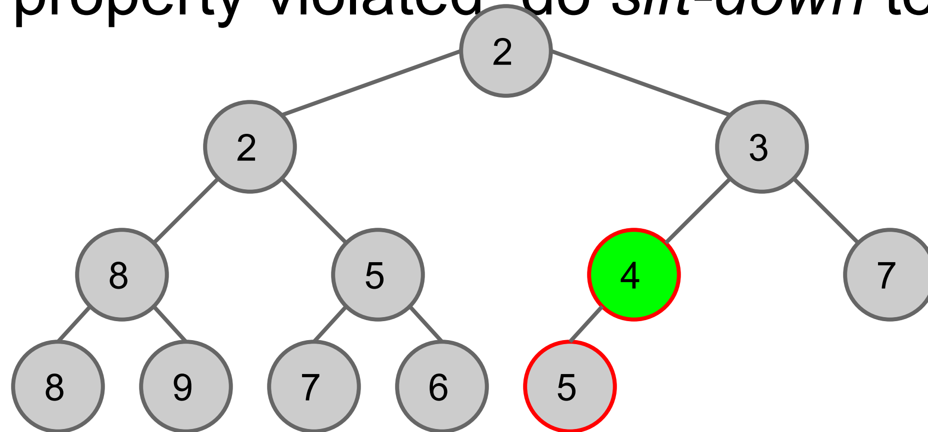
Binary Heap

- Remove minimum
  - Replace the root with the last node in the lowest level
  - heap property violated, do *sift-down* to recover



Binary Heap

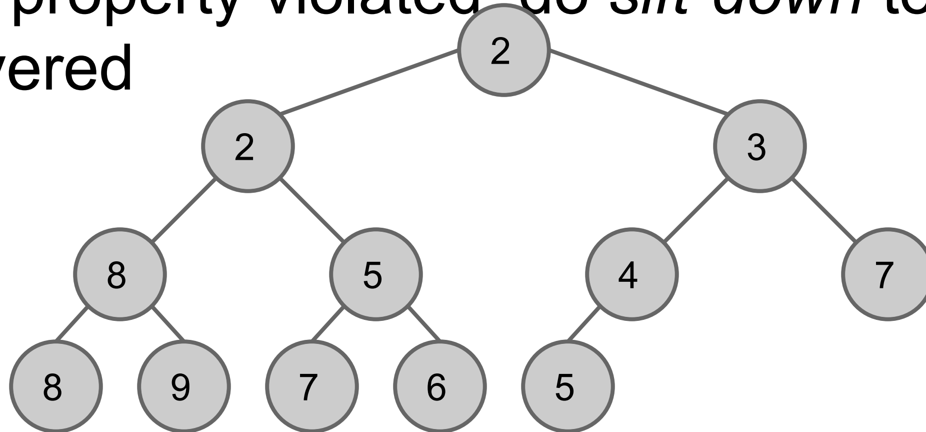
- Remove minimum
  - Replace the root with the last node in the lowest level
  - heap property violated, do *sift-down* to recover



no smaller children. stop

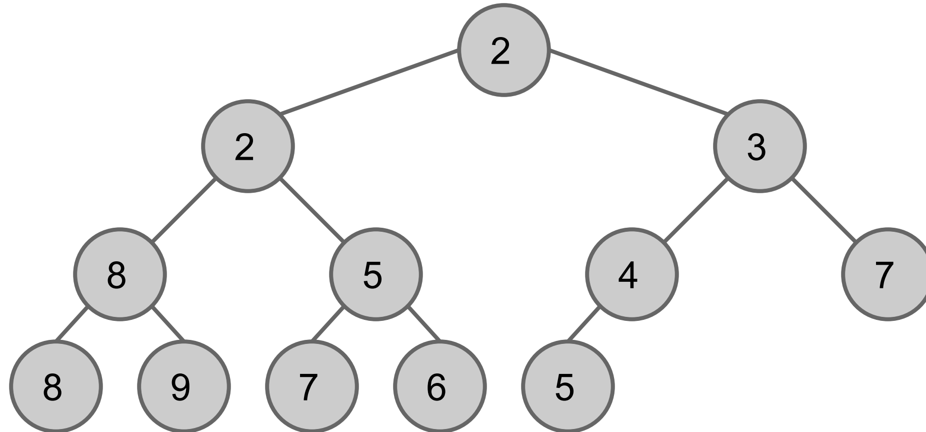
Binary Heap

- Remove minimum
  - Replace the root with the last node in the lowest level
  - heap property violated, do *sift-down* to recover
  - recovered



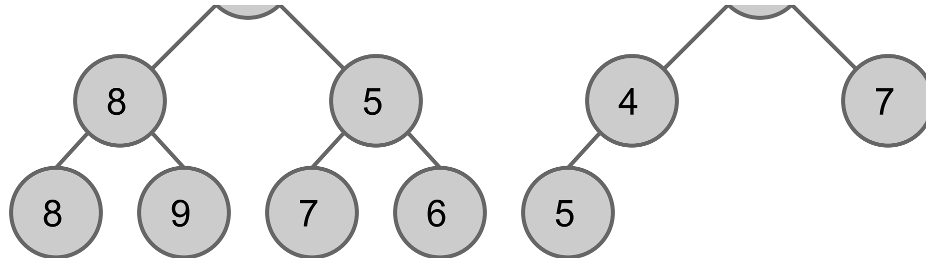
Binary Heap

- Remove minimum
  - Time complexity: O(log(N))
  - Why?

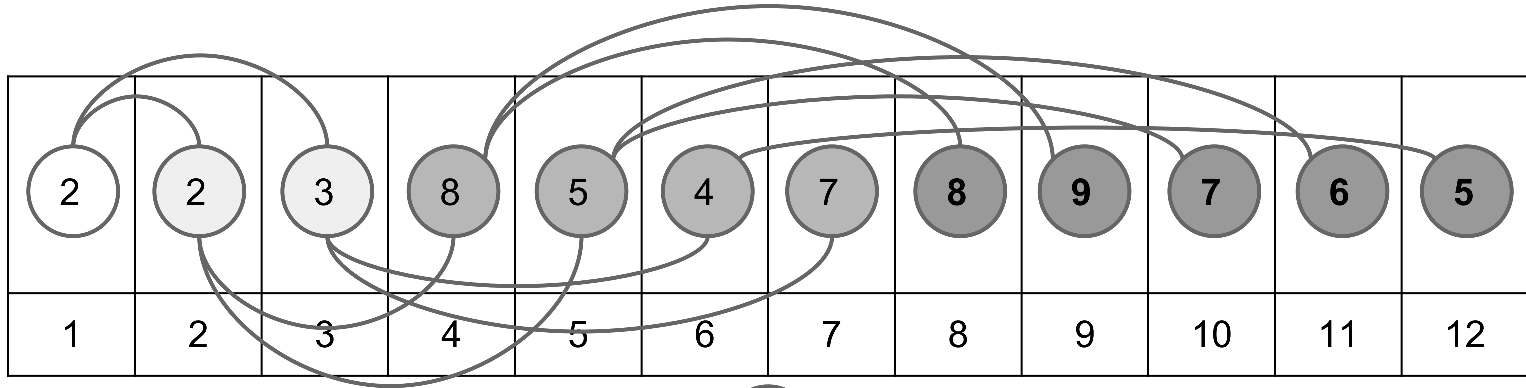


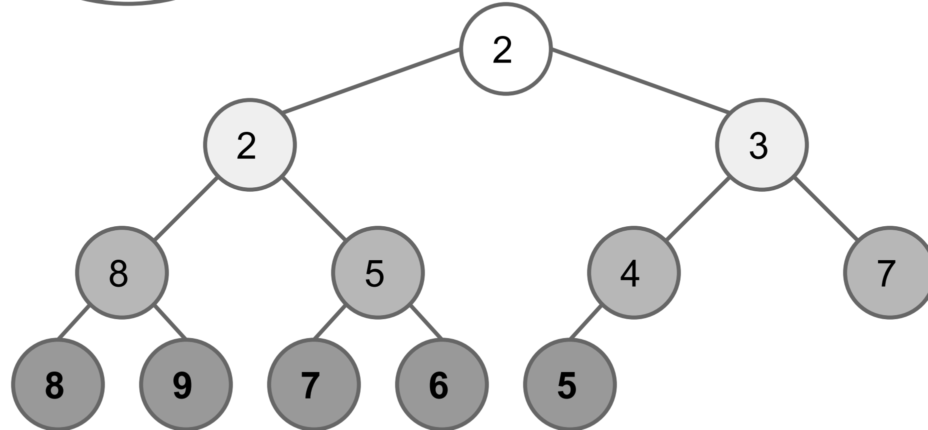
Binary Heap

- Implementation as array
  - array[1..N]
  - root: array[1]
  - for array[i],
    - left child: array[i * 2]
    - right child: array[i * 2 + 1]



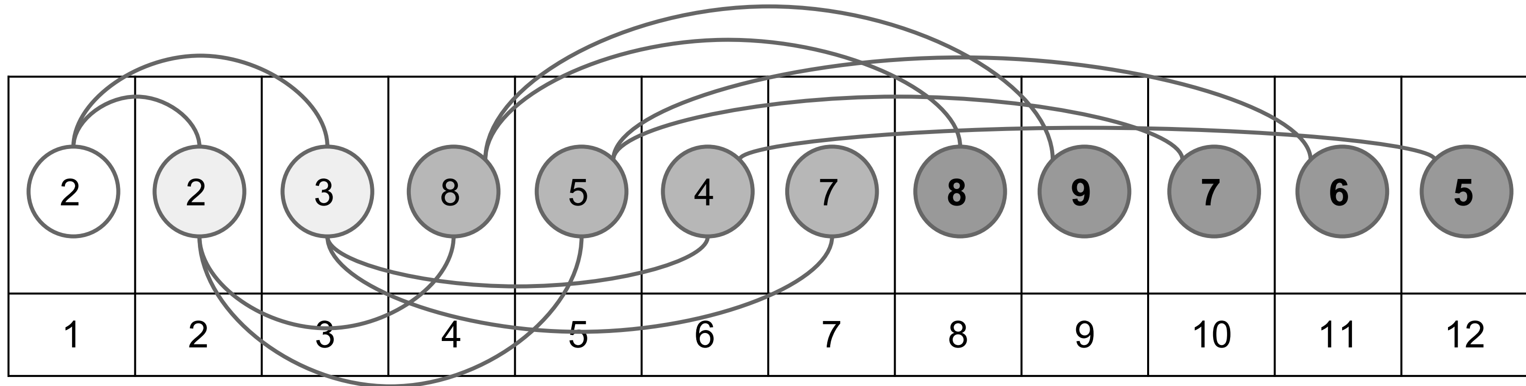
Binary Heap





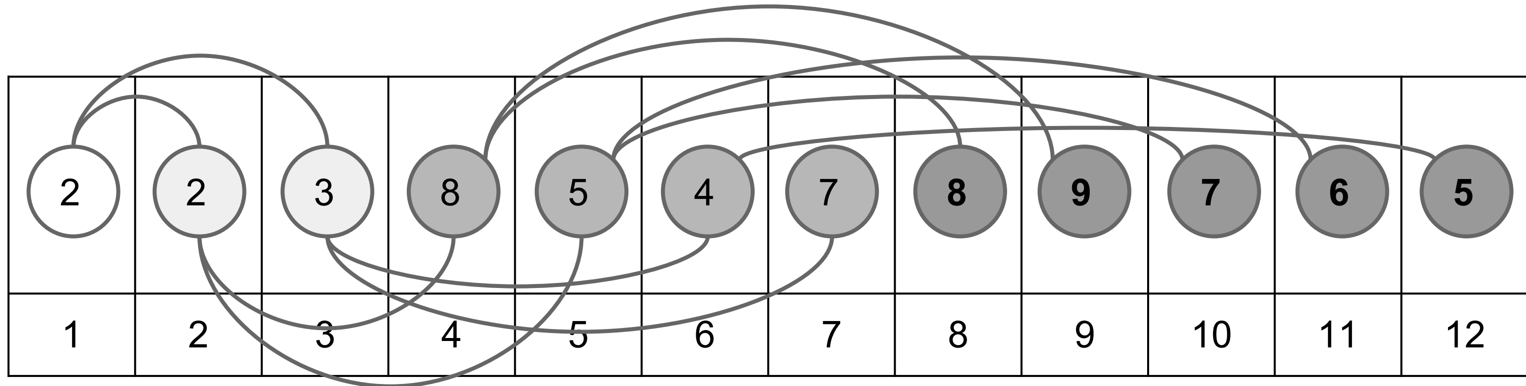
Binary Heap

- for array[i],
  - left child: array[i * 2]
  - right child: array[i * 2 + 1]
  - parent: ?



Binary Heap

- for array[i],
  - left child: array[i * 2]
  - right child: array[i * 2 + 1]
  - parent: array[i / 2]



Binary Heap

- C++ STL: priority_queue
- Note that it is a max-heap,
  - all nodes are **greater than or equal to** each of its children
  - #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

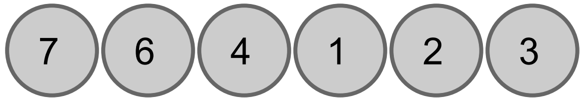
- Heapsort
  - insert everything into heap: O(Nlog(N))
  - repeatedly remove minimum: O(Nlog(N))
  - overall: O(Nlog(N))

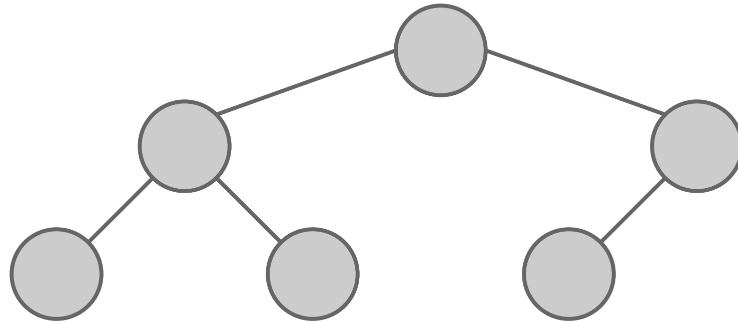
Binary Heap

- Heapsort
  - build heap: O(N)
  - repeatedly remove minimum: O(Nlog(N))
  - overall: O(Nlog(N))

Binary Heap

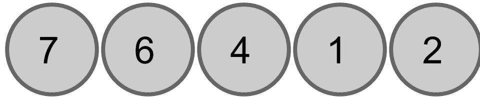
- O(N) build heap
- insert from the lowest level, then sift-down

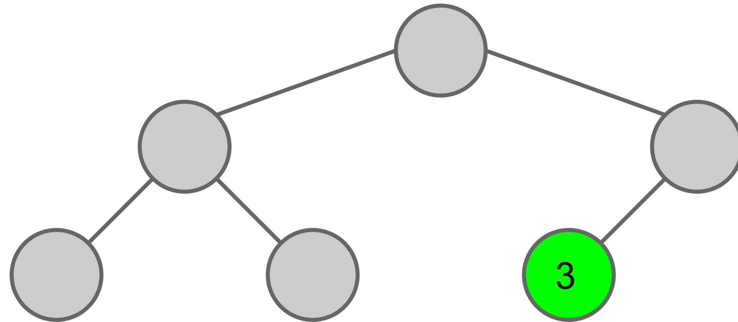




Binary Heap

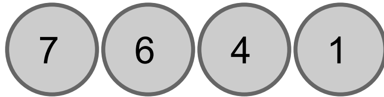
- O(N) build heap
- insert from the lowest level, then sift-down

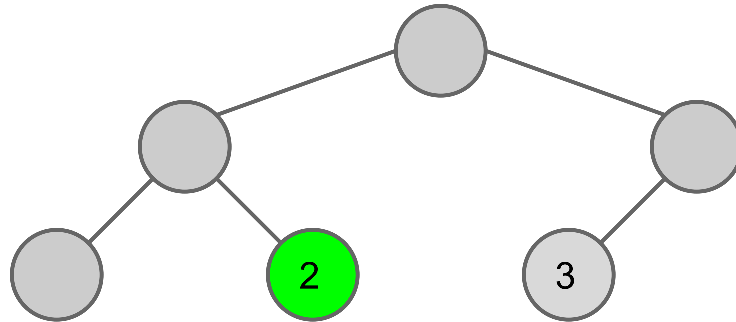




Binary Heap

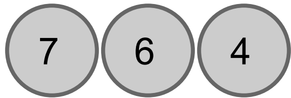
- O(N) build heap
- insert from the lowest level, then sift-down

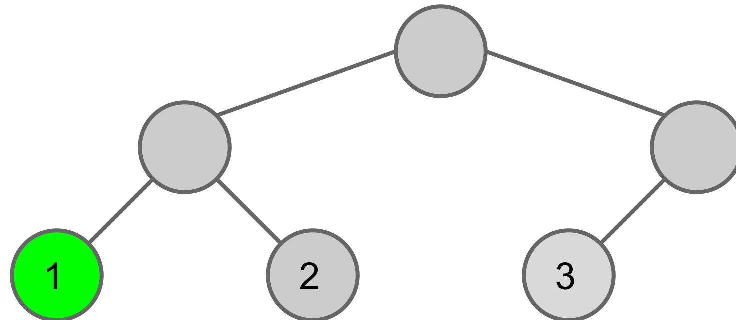




Binary Heap

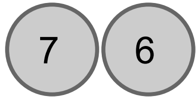
- O(N) build heap
- insert from the lowest level, then sift-down

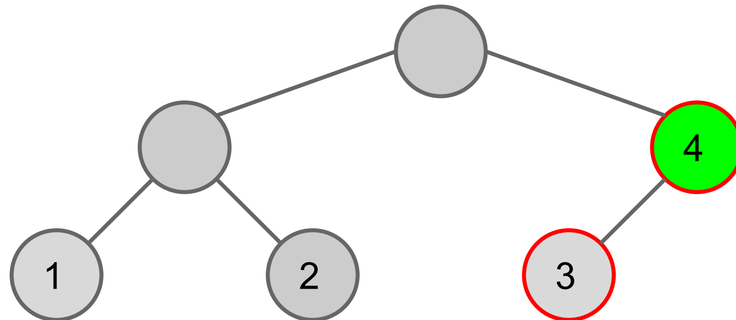




Binary Heap

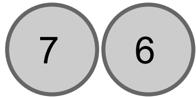
- O(N) build heap
- insert from the lowest level, then sift-down

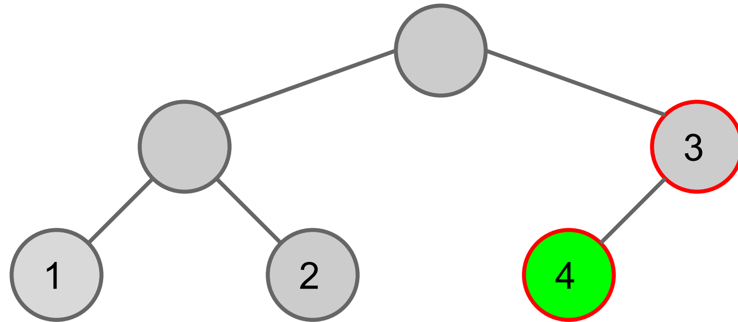




Binary Heap

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

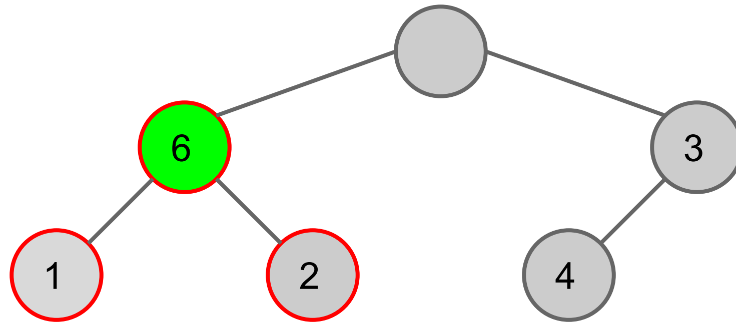




Binary Heap

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

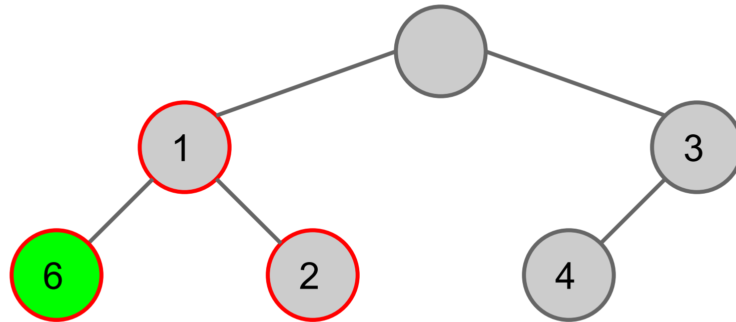
7



Binary Heap

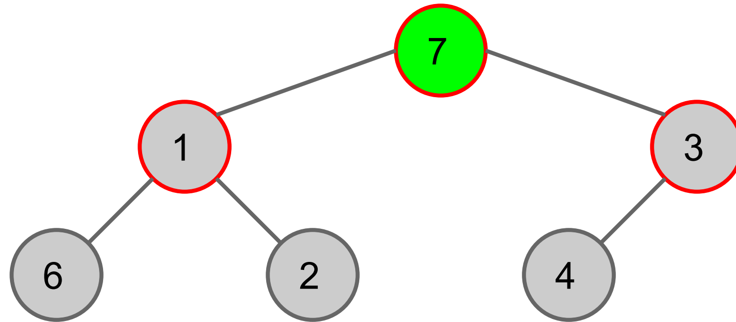
- O(N) build heap
- insert from the lowest level, then sift-down

7



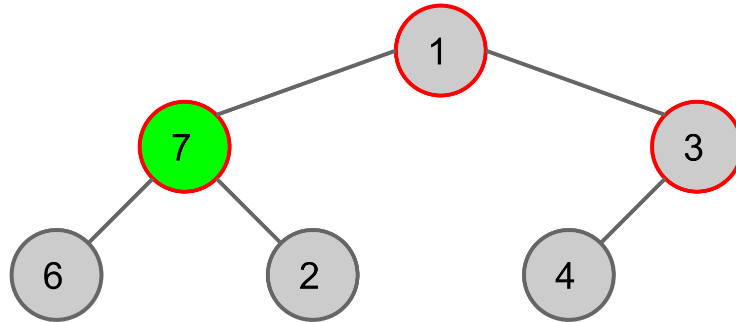
Binary Heap

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



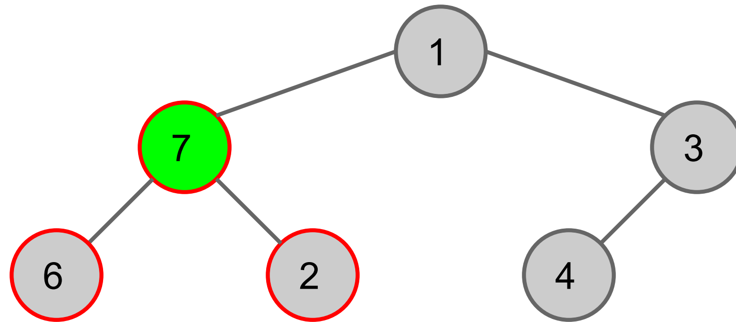
Binary Heap

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



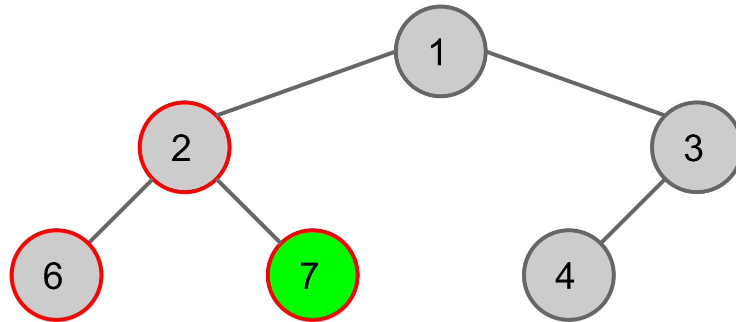
Binary Heap

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



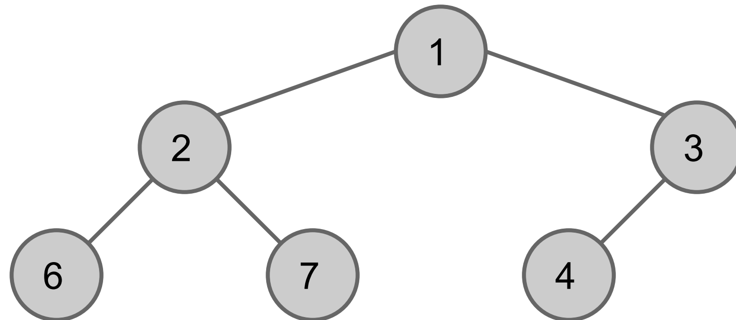
Binary Heap

- 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
- for proof of time complexity, see wikipedia

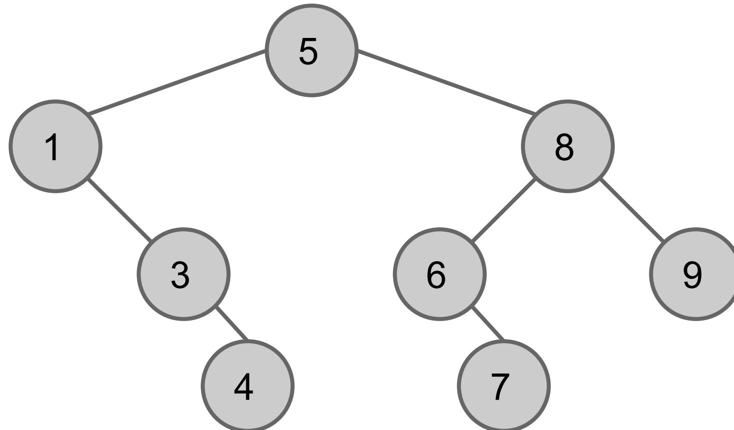


Content

- Binary Heap
- Binary Search Tree
- Hash Table
- XOR Linked List

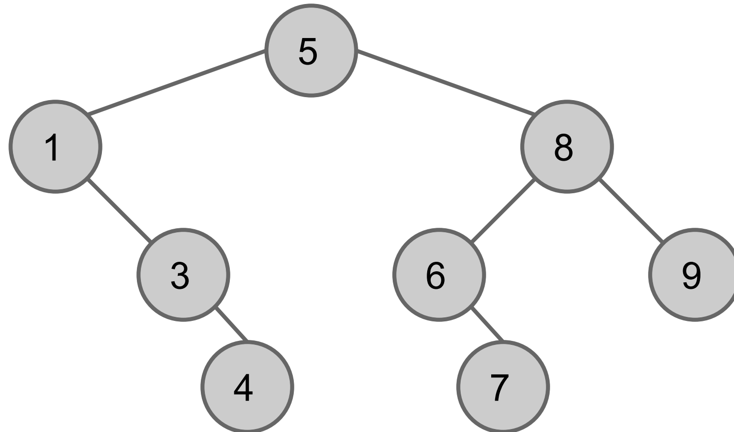
Binary Search Tree

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



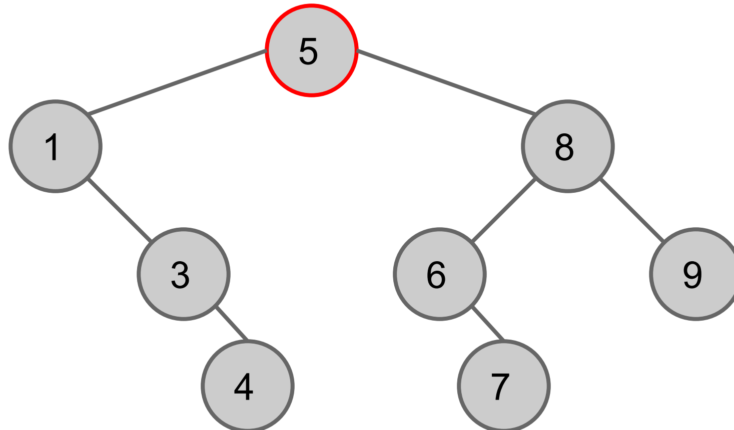
Binary Search Tree

- find a node
    - is ‘6’ in the tree?
    - starting from root, walk to left, walk to right.



Binary Search Tree

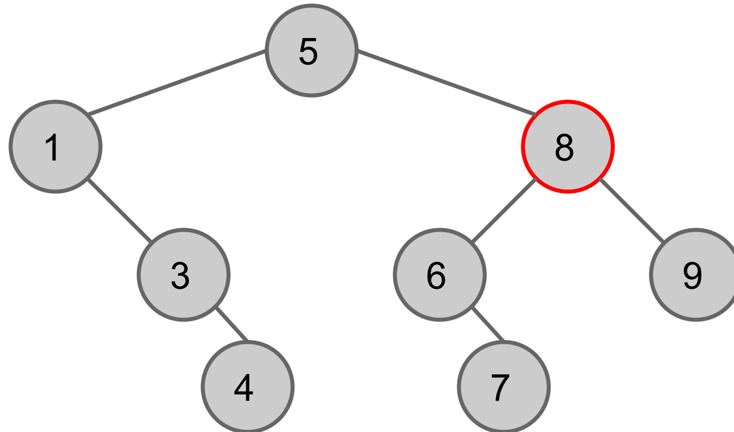
- find a node
  - is ‘6’ in the tree?
  - starting from root, walk to left, walk to right.



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

Binary Search Tree

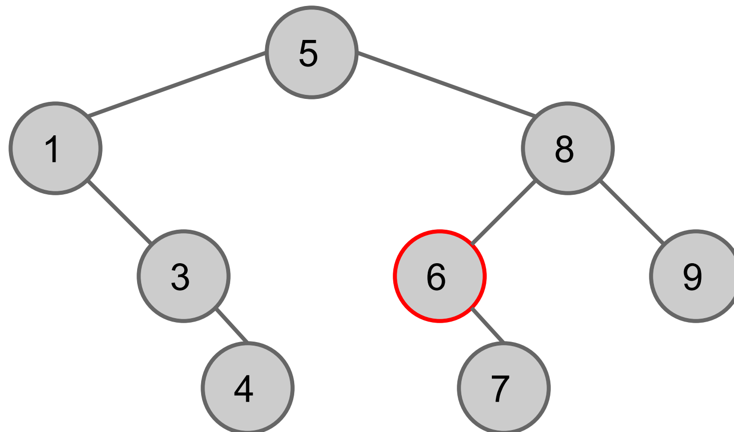
- find a node
  - is ‘6’ in the tree?
  - starting from root, walk to left, walk to right.



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

Binary Search Tree

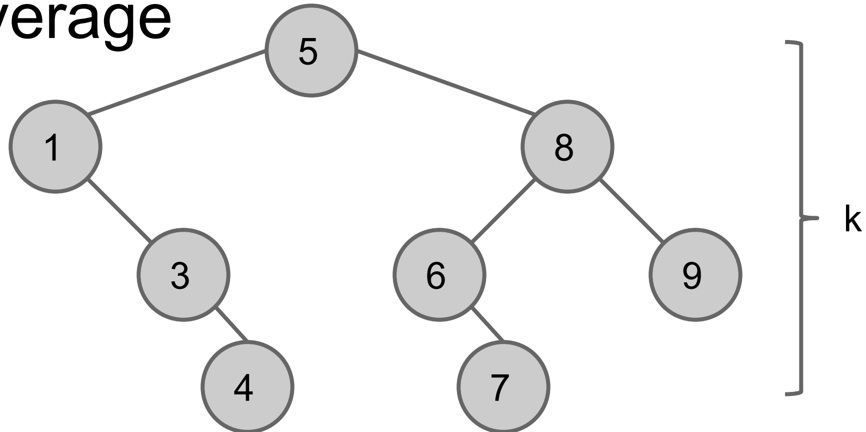
- find a node
  - is ‘6’ in the tree?
  - starting from root, walk to left, walk to right.



‘6’ is in the tree!

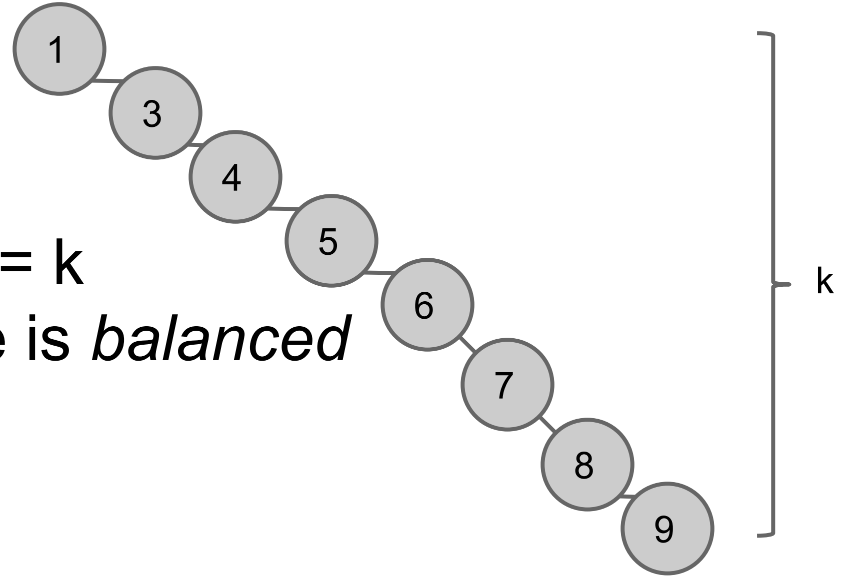
Binary Search Tree

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



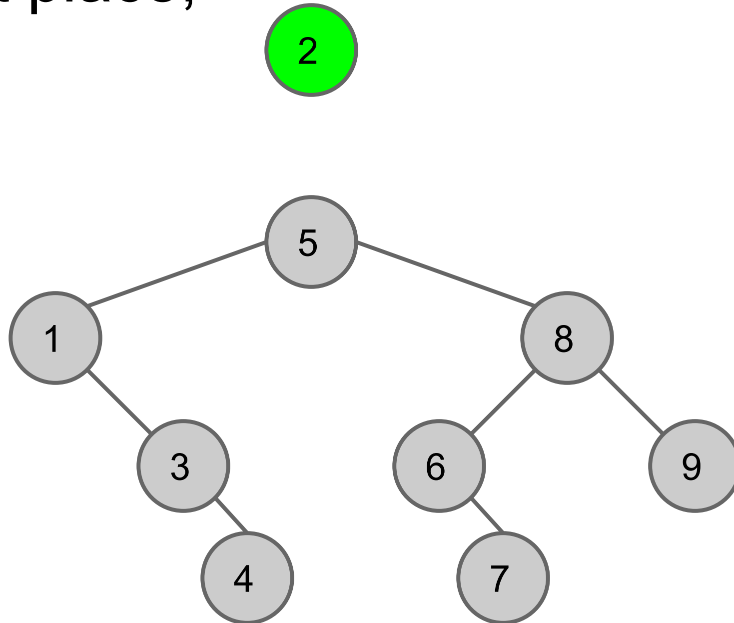
Binary Search Tree

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



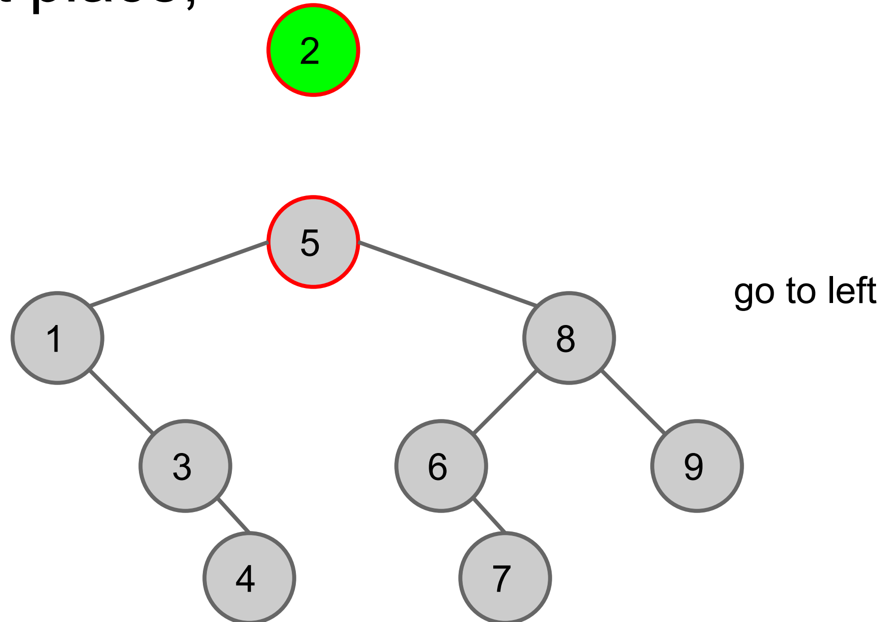
Binary Search Tree

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



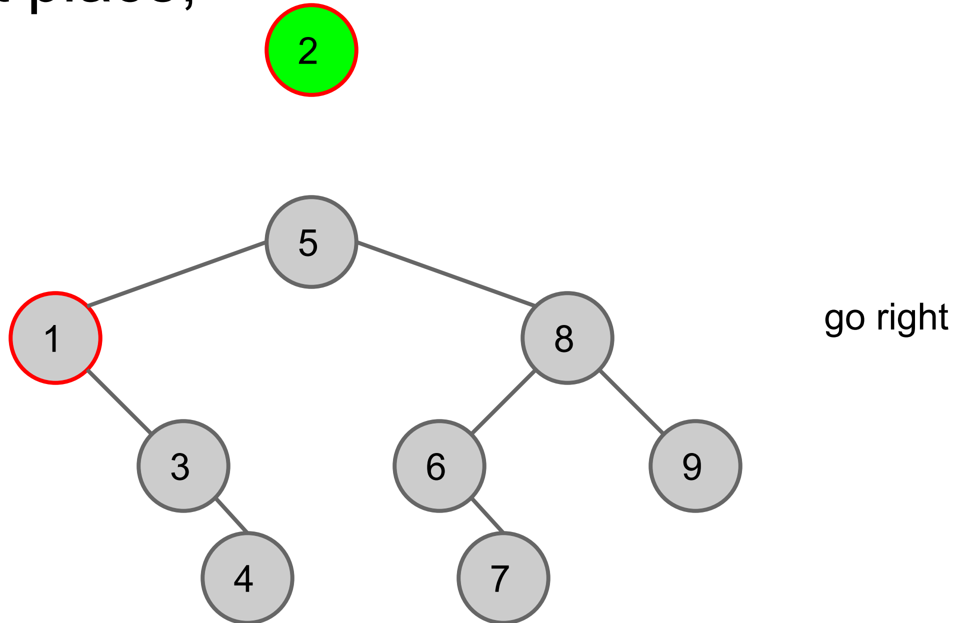
Binary Search Tree

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



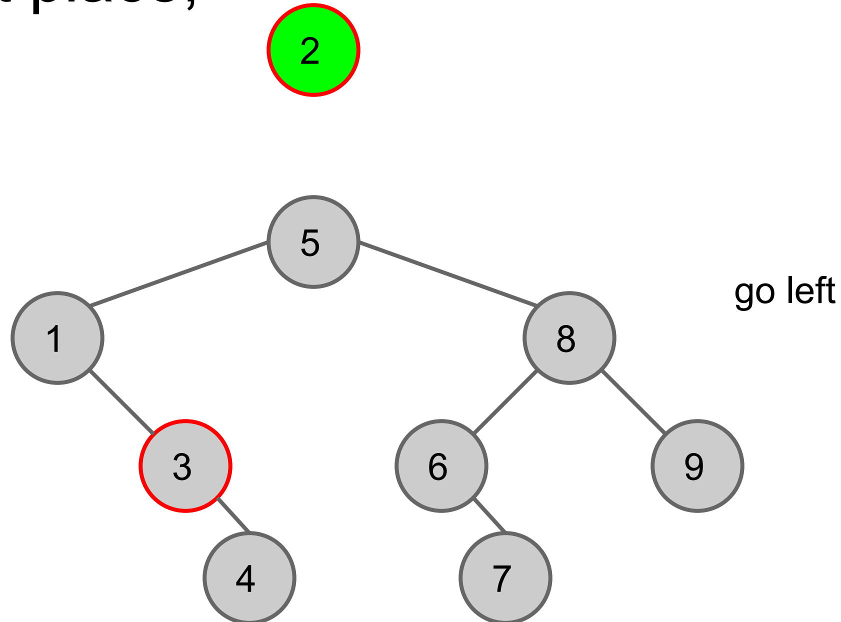
Binary Search Tree

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



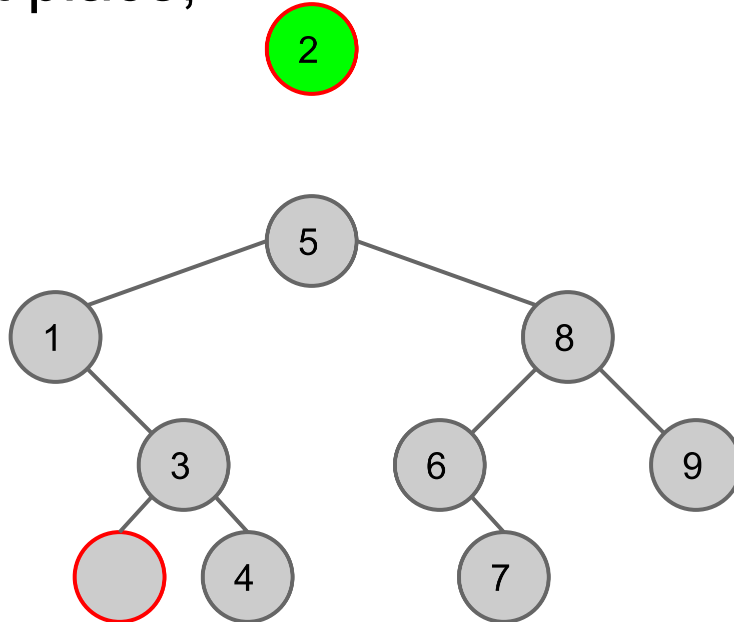
Binary Search Tree

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



Binary Search Tree

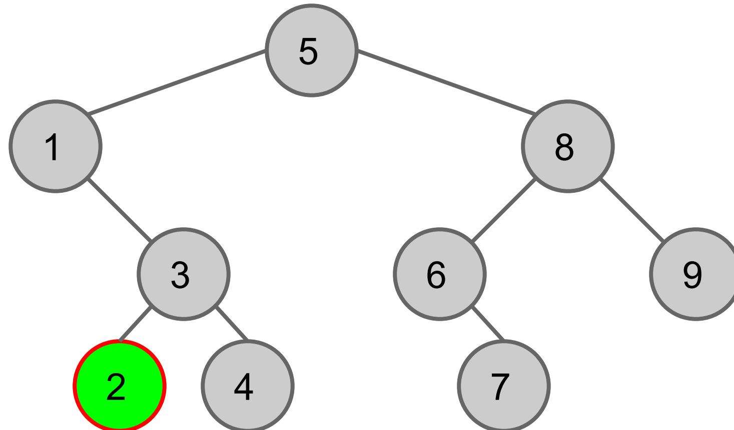
- Insert a node
  - find the right place,
  - insert.



left child is empty,
create new node

Binary Search Tree

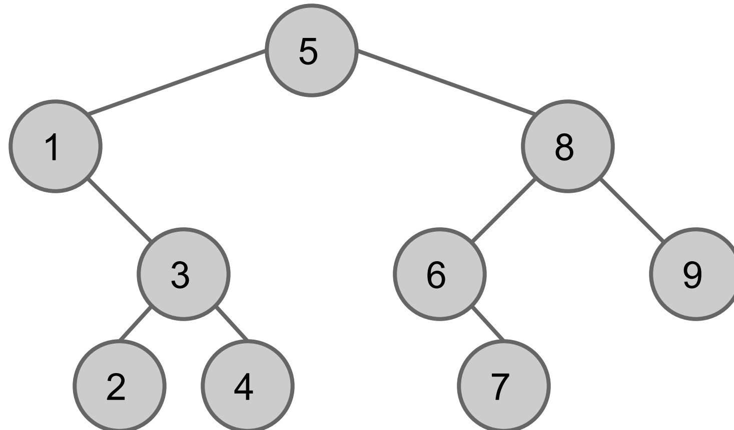
- Insert a node
  - find the right place,
  - insert.



insert.

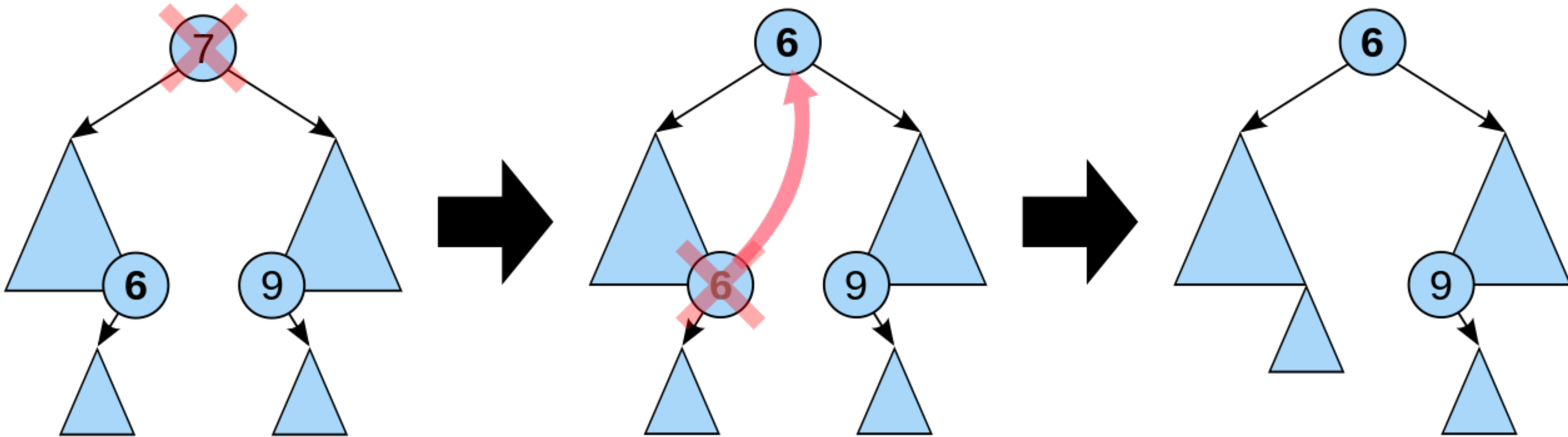
Binary Search Tree

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



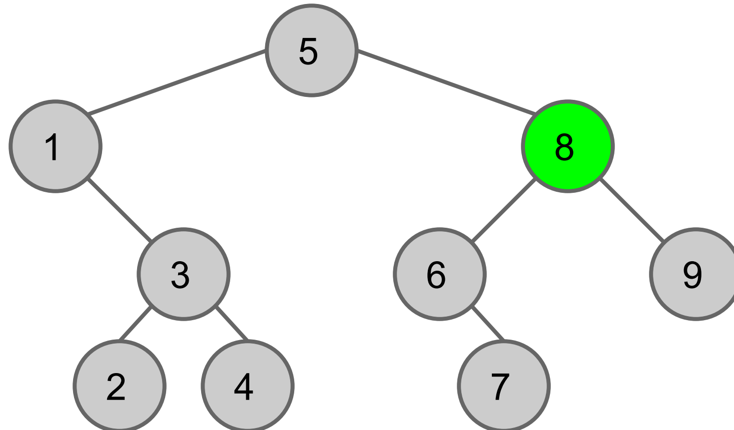
Binary Search Tree

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



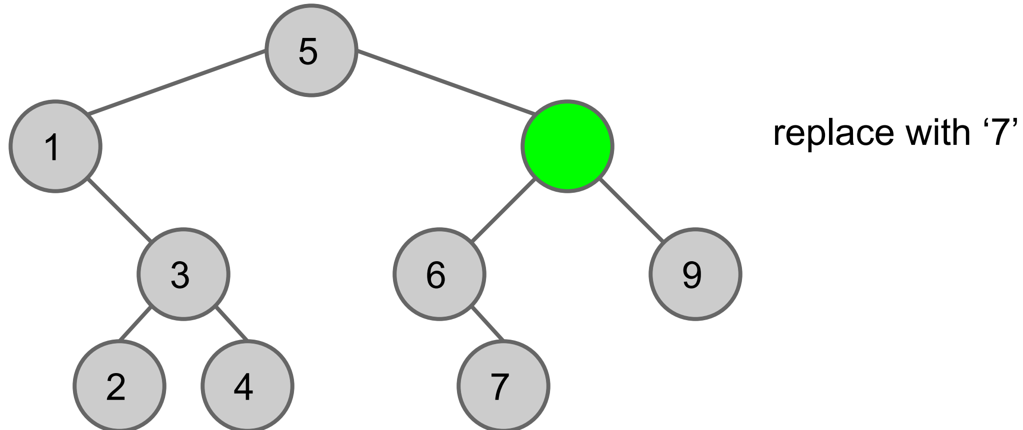
Binary Search Tree

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



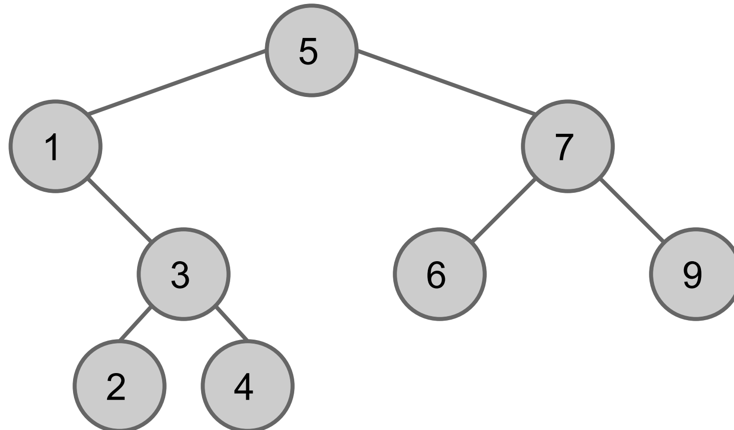
Binary Search Tree

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



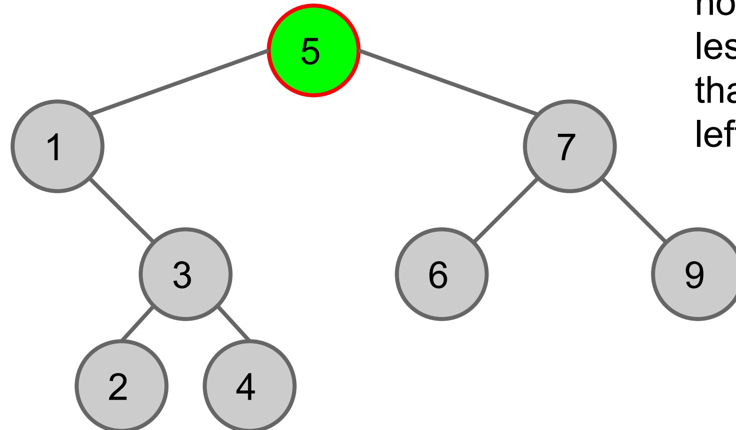
Binary Search Tree

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



Binary Search Tree

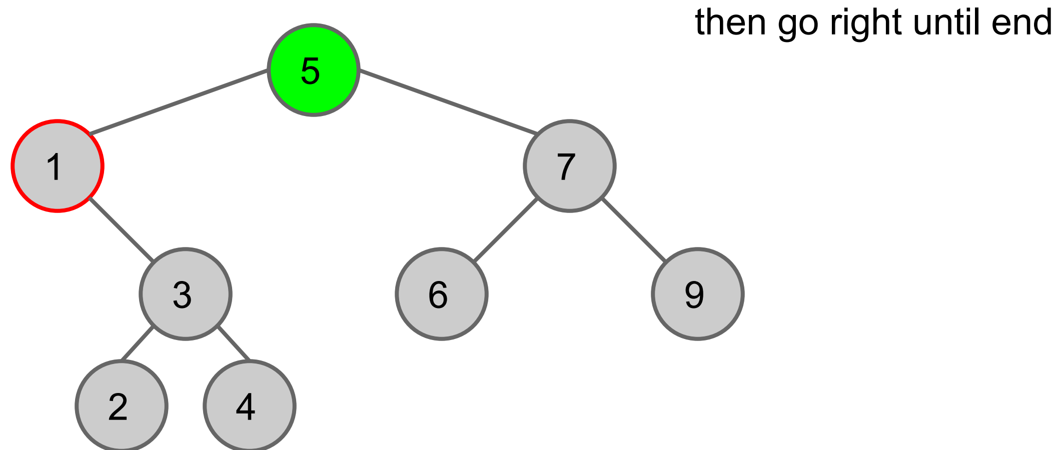
- Remove a node
  - replace that node with the node *just* less than it



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

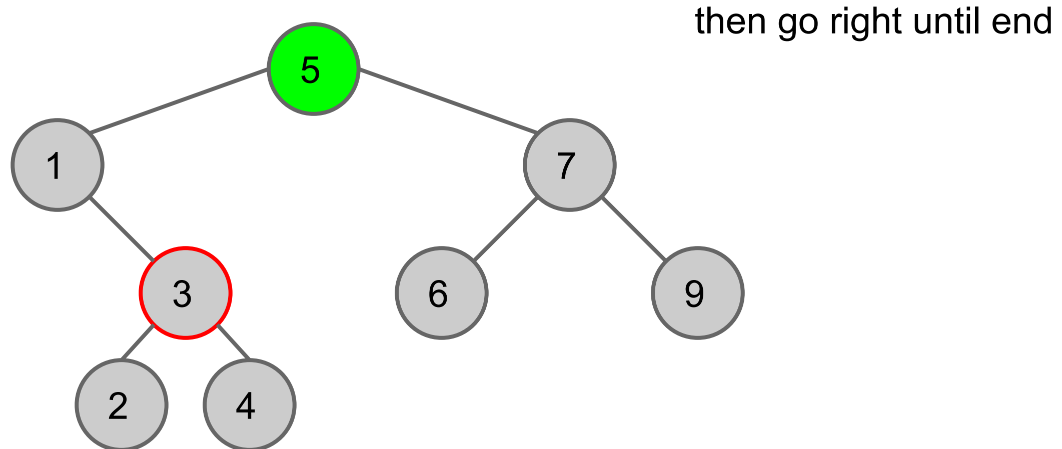
Binary Search Tree

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



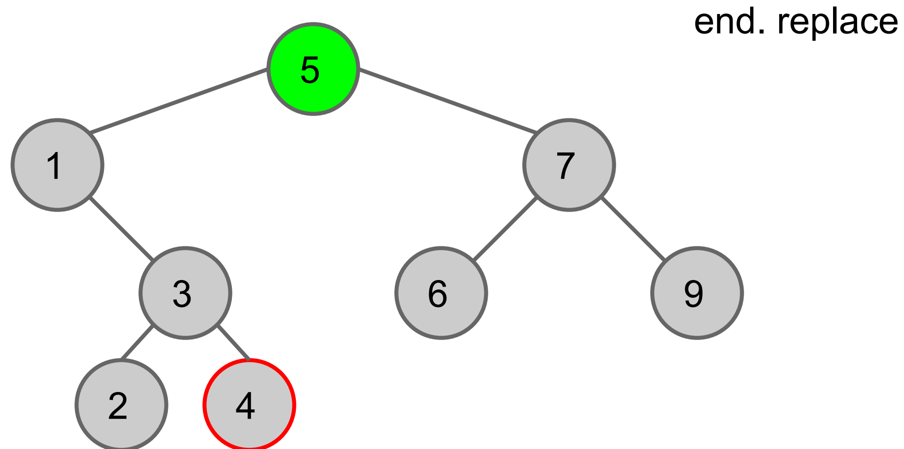
Binary Search Tree

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



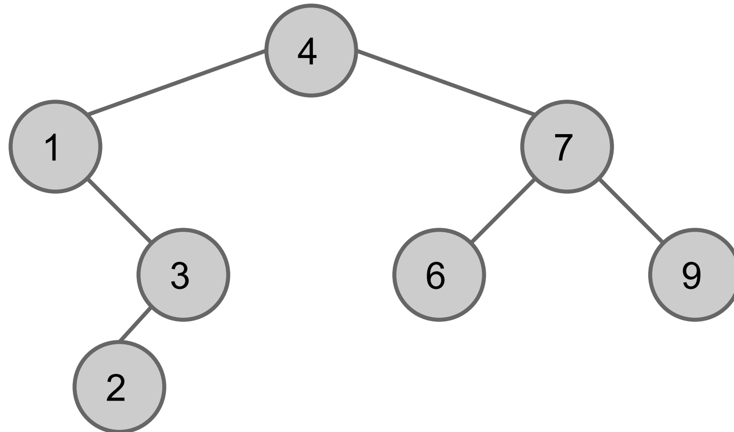
Binary Search Tree

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



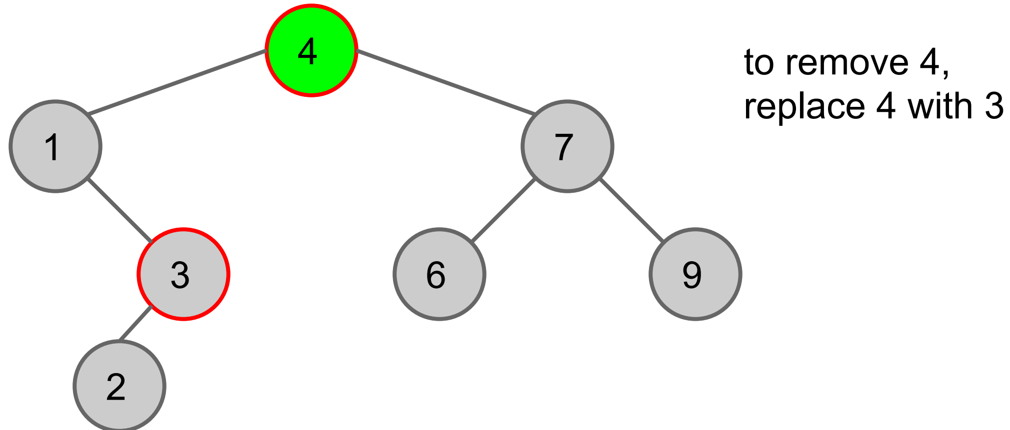
Binary Search Tree

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



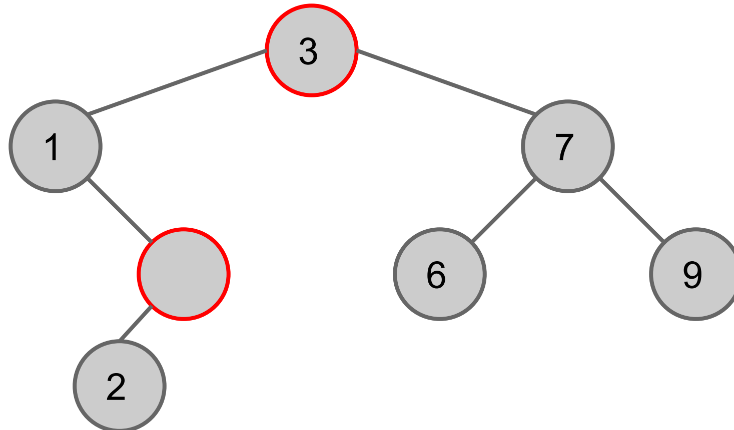
Binary Search Tree

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



Binary Search Tree

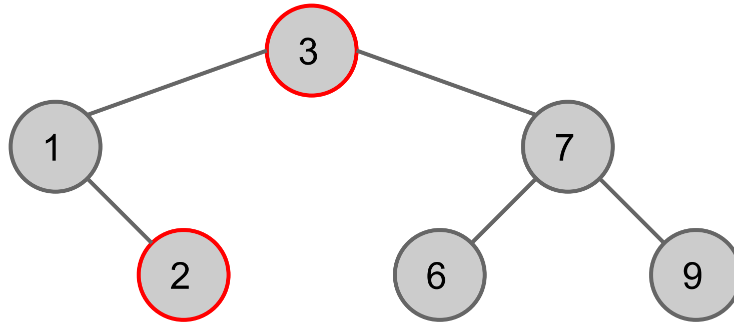
- Remove a node
    - replace that node with the node *just* less than it



remember to bring up 2

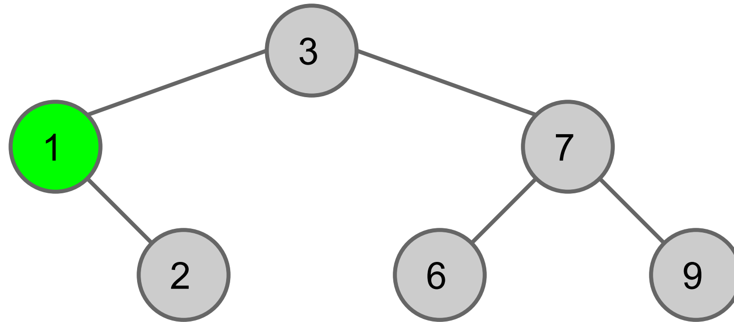
Binary Search Tree

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



Binary Search Tree

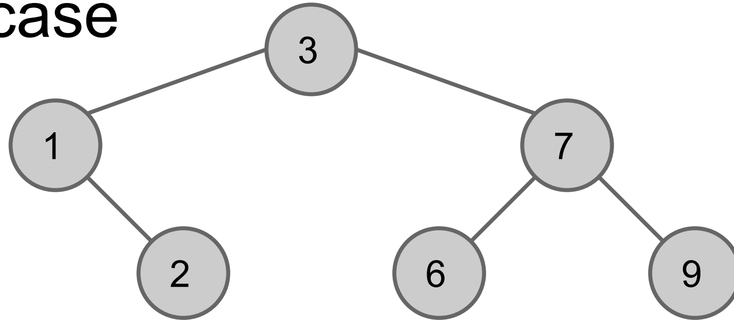
- Remove a node
    - replace that node with the node *just* less than it



how to remove 1?

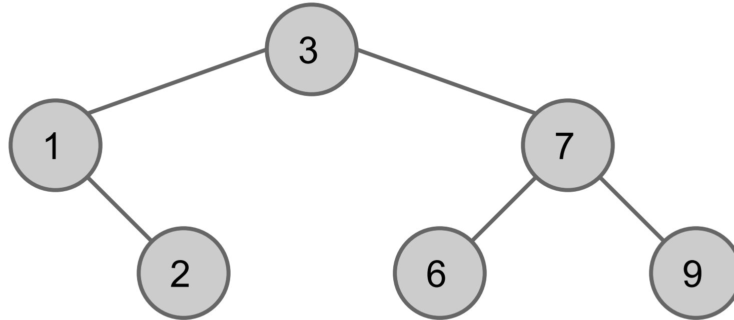
Binary Search Tree

- Remove a node
  - locate the node to be removed: O(log(N))
  - locate the node *just* less than it: O(log(N))
  - overall, O(log(N))
  - 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, Treap, Size-balanced Tree
- 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);

Content

- Binary Heap
- Binary Search Tree
- Hash Table
- XOR Linked List

Hash Table

- Plain array
- A value is stored in a slot calculated by the *hash function*
- The *hash function* is usually in the form
- int h(int x) {
-   return x % 17; //note that 17 is prime
- }

Hash Table

- Insert
  - put a value in the slot calculated by h()

972

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

Hash Table

- Insert
  - put a value in the slot calculated by h()

972

972 % 17 = 3

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

Hash Table

- Insert
  - put a value in the slot calculated by h()

257

257 % 17 = 2

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

Hash Table

- Insert
  - put a value in the slot calculated by h()

339

339 % 17 = 16

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

Hash Table

- Insert
  - put a value in the slot calculated by h()

376

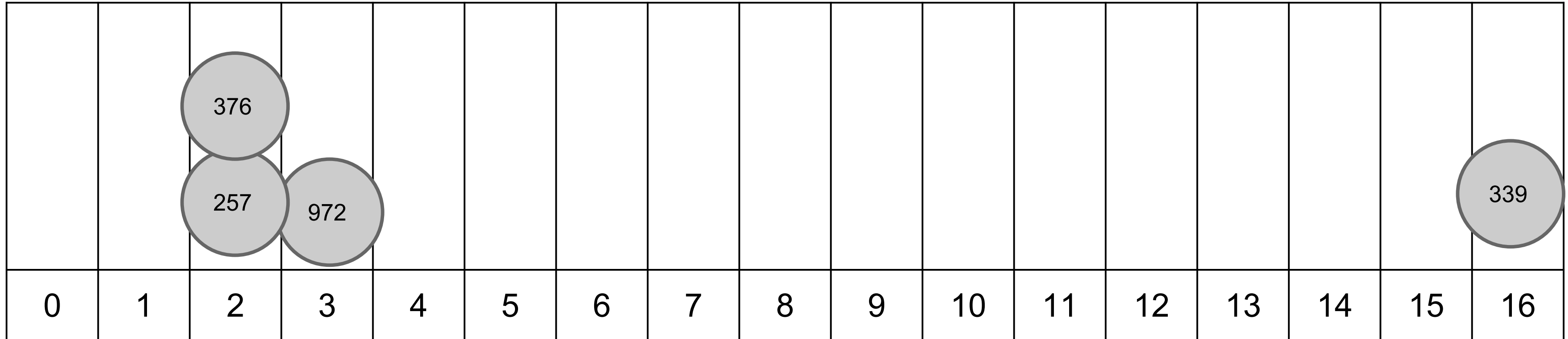
376 % 17 = 2

oops occupied

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

Hash Table

- Insert
  - *collision* resolution 1: separate chaining
  - each slots are linked lists



Hash Table

- Insert
  - *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
  - for separate chaining,
    - traverse the linked list at slot calculated by h()
  - for open addressing,
    - scan from slot h() 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
  - for separate chaining,
    - remove from the linked list at the slot h()
  - for 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

Content

- Binary Heap
- Binary Search Tree
- Hash Table
- XOR Linked List