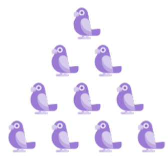
✨ **No Laptop Monday!** ✨

Week10a



- binary search
- binary search trees

WARMUP

how long does it take to **find** (**search** for) something in...
- an array list?
- a linked list?

what if the list is sorted?

[record lecture]

review: sorted

sorted

sorted

- a sequence is **sorted** if its elements are "in order"
  - by convention, this means ascending order (going up from left to right)
    - `[1, 2, 5, 6, 9, 13]` is sorted
    - `[9, 5, 1, 2, 6, 13]` is **unsorted** (NOT sored)

search

search

- to **search** means to look for something (in some data structure)
  - a simple search problem is finding a given value in an array / list
  - ```
    // get index of the first element in array with this value
    // returns -1 if value not found
    int find(int[] array, int value) { ... }
    ```
  - ```
    // Option B
    class FindResult {
        boolean success;
        int index;
    }
    FindResult find(int[] array, int value) { ... }
    ```

linear search

(of an array / array list)

linear search (brute force search)

- **linear search** looks at each element one by one
  - linear search works whether or not the list is sorted
  - linear search is **O(n)** (linear time) 😐
  - ```
    // get index of the first element in array with this value
    // returns -1 if value not found
    int linearSearch(int[] array, int value) {
        for (int i = 0; i < array.length; ++i) {
            if (array[i] == value) {
                return i;
            }
        }
        return -1;
    }
    ```

example: linear search for 17

[13 2 55 7 17 100 77]

example: linear search for 17

[13 2 55 7 17 100 77]

example: linear search for 17

[13 2 55 7 17 100 77]
    ```

example: linear search for 17

[13 2 55 7 17 100 77]

example: linear search for 17

[13 2 55 7 17 100 77]

example: linear search for 17

[13 2 55 7 17 100 77]

example: linear search for 17

[13 2 55 7 17 100 77]

example: linear search for 17

[13 2 55 7 17 100 77]

example: linear search for 17

[13 2 55 7 17 100 77]

example: linear search for 17
[13 2 55 7 17 100 77]

example: linear search for 17
[13 2 55 7 17 100 77]

example: linear search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: linear search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

if we know that an array is sorted...
can we search it faster?

yes 🚀

unless it's a linked list
💀

binary search
(of a array / array list)

binary search

- **binary search** is an O(log n) algorithm for searching a sorted array 😀
  - binary search works by "cutting the array in half" over and over
  - ☠️ **binary search only applies if the array is sorted**

example: binary search for 17

```
[1  2  4  5  6  8  9  10  11  13  15  16  17  18  19]
```

example: binary search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

10 < 17

example: binary search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: binary search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

16 < 17

example: binary search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: binary search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

17 < 18

example: binary search for 17

[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

example: binary search for 17

```
[1  2  4  5  6  8  9  10  11  13  15  16  17  18  19]
                                          17 == 17
```

example: binary search for 17

```
[1  2  4  5  6  8  9  10  11  13  15  16  17  18  19]
```

☠️

implementating binary search is surprisingly tricky!

make sure you test thoroughly!

✨ **hint:** binary search

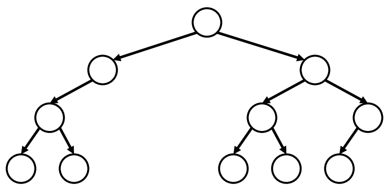
```
[1  2  4  5  6  8  9  10  11  13  15  16  17  18  19]
 ^                                                ^
 |                                                |
 i = 0                                            j
```

review: binary tree

binary tree

binary tree

- a **binary tree** is a tree in which each node has <= 2 children



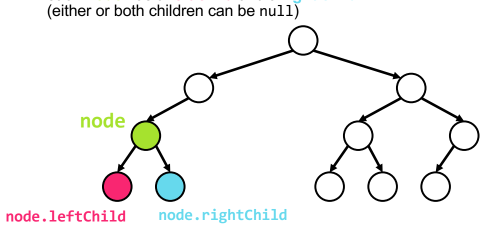
🧠 can a linked list be seen as a binary tree?

technically, yes (assuming no cycles)
all nodes have <= 2 children
(tail has 0 children, all other nodes have 1 child)

ordered binary tree

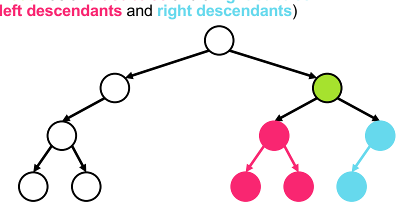
(ordered) binary tree

- each **node** has a **left child** and a **right child**
  (either or both children can be `null`)



(ordered) binary tree

- each node has a **left subtree** and a **right subtree**
  (and **left descendants** and **right descendants**)

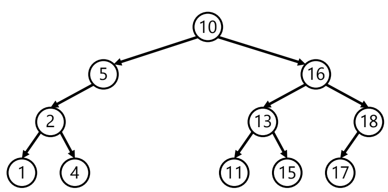


binary search tree

binary search tree

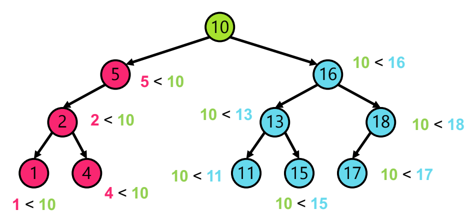
binary search tree

– in a **binary search tree** (**BST**, sorted binary tree) **every** node follows:
"a node's value is **greater than the values of all nodes in its left subtree**
and **less than the values of all nodes in its right subtree**"



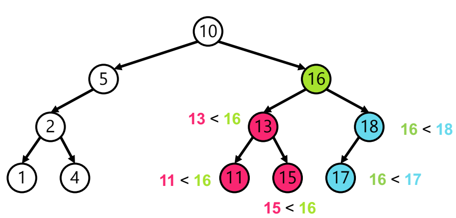
binary search tree

– in a **binary search tree** (**BST**, sorted binary tree) every node follows:
"a node's value is **greater than the values of all nodes in its left subtree**
and **less than the values of all nodes in its right subtree**"



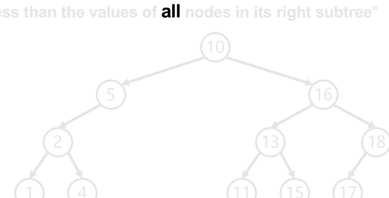
binary search tree

– in a **binary search tree** (**BST**, sorted binary tree) every node follows:
"a node's value is **greater than the values of all nodes in its left subtree**
and **less than the values of all nodes in its right subtree**"



binary search tree

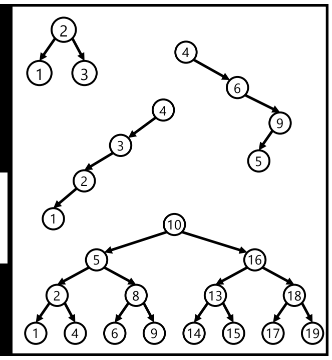
– in a **binary search tree** (**BST**, sorted binary tree) **every** node follows:
"a node's value is **greater than the values of all nodes in its left subtree**
and **less than the values of all nodes in its right subtree**"



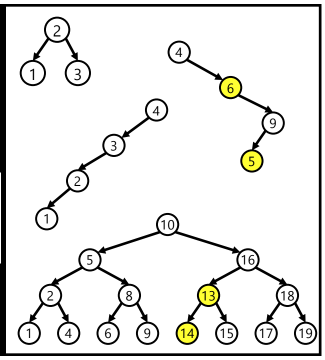
is it a BST?

Is It A Binary Search Tree?

in a **binary search tree** (sorted binary tree)
every node follows:
"a node's value is **greater than the values of all nodes in its left subtree** and **less than the values of all nodes in its right subtree**"



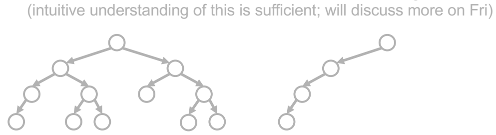
in a **binary search tree** (sorted binary tree)
every node follows:
"a node's value is **greater than the values of all nodes in its left subtree** and **less than the values of all nodes in its right subtree**"



binary search
(of a binary search tree)

binary search

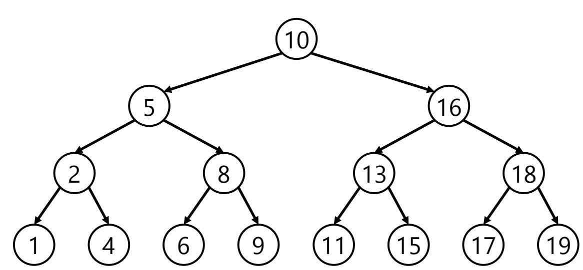
- **binary search** "cuts a BST in half" over and over
  - **binary search only works when the binary tree is sorted (is a binary search tree)**
  - binary search is **O(log n)** if the tree is "balanced" 😊
    - **balanced** means each node's children are similar in height (intuitive understanding of this is sufficient; will discuss more on Fri)

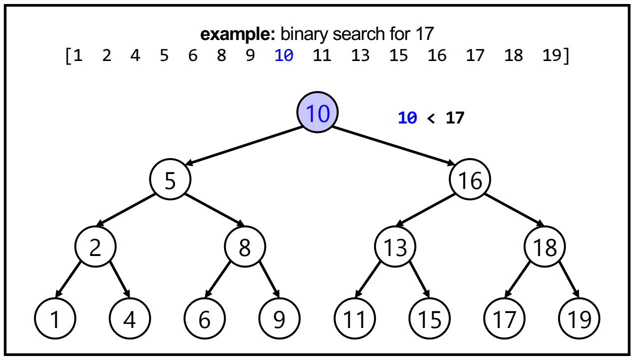


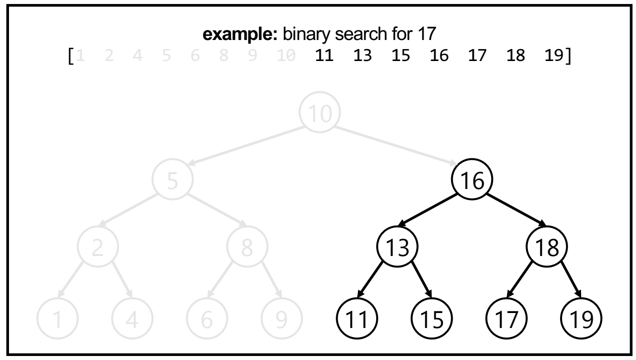
balanced NOT balanced

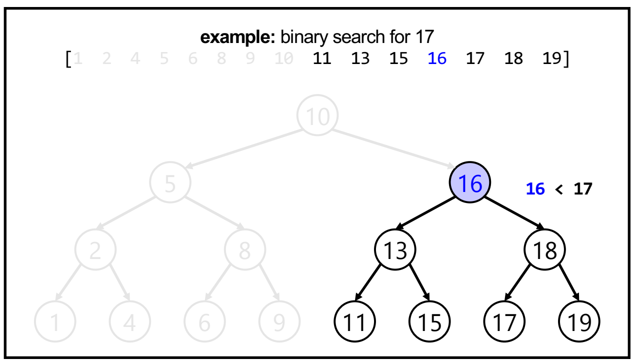
example: binary search for 17

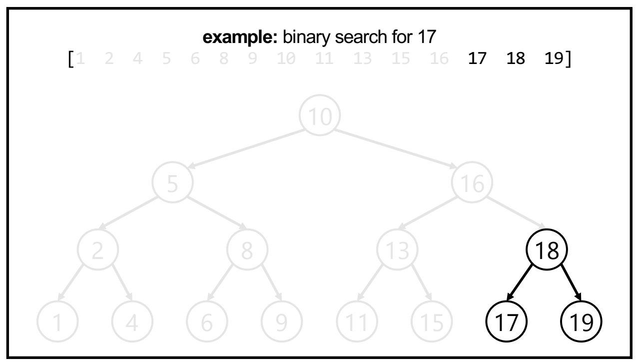
```
[1  2  4  5  6  8  9  10  11  13  15  16  17  18  19]
```

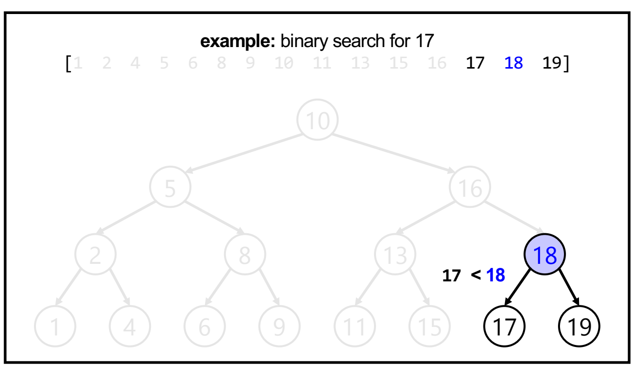


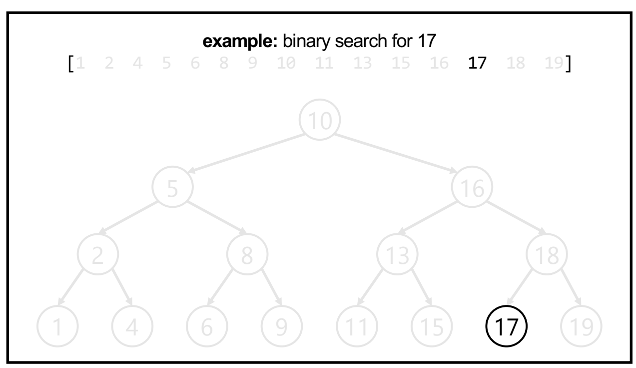

example: binary search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]
10 < 17
10 5 16 2 8 13 18 1 4 6 9 11 15 17 19


example: binary search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]
10 5 16 2 8 13 18 1 4 6 9 11 15 17 19


example: binary search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]
16 < 17
10 5 16 2 8 13 18 1 4 6 9 11 15 17 19

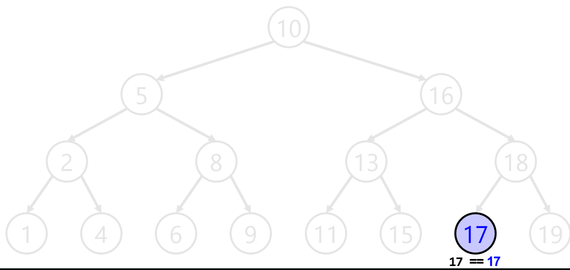

example: binary search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]
10 5 16 2 8 13 18 1 4 6 9 11 15 17 19


example: binary search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]
17 < 18
10 5 16 2 8 13 18 1 4 6 9 11 15 17 19


example: binary search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]
10 5 16 2 8 13 18 1 4 6 9 11 15 17 19

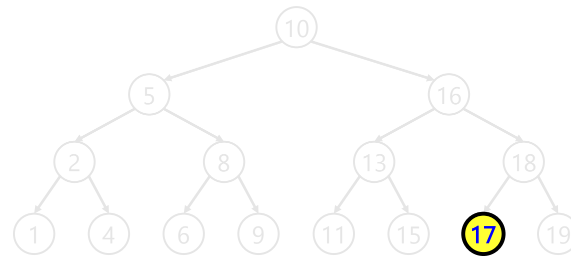
example: binary search for 17

`[1  2  4  5  6  8  9  10  11  13  15  16  17  18  19]`



example: binary search for 17

`[1  2  4  5  6  8  9  10  11  13  15  16  17  18  19]`

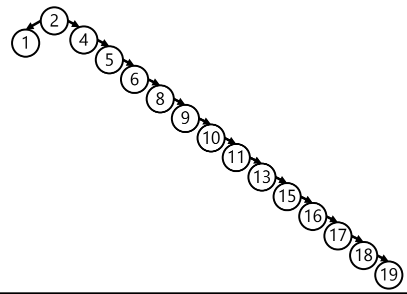


note: a binary search tree is NOT unique

let's look at another BST for the same data!

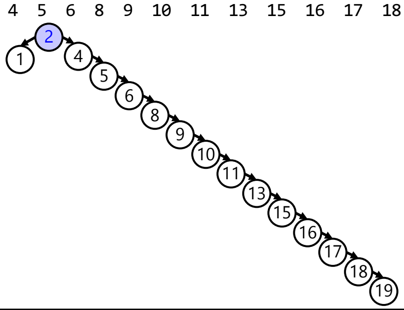
example: binary search for 17

`[1  2  4  5  6  8  9  10  11  13  15  16  17  18  19]`



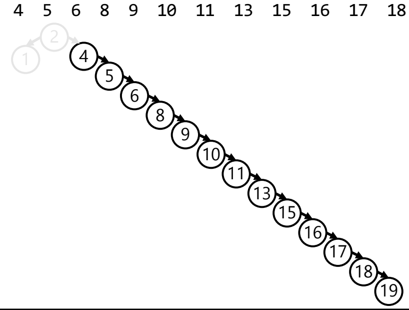
example: binary search for 17

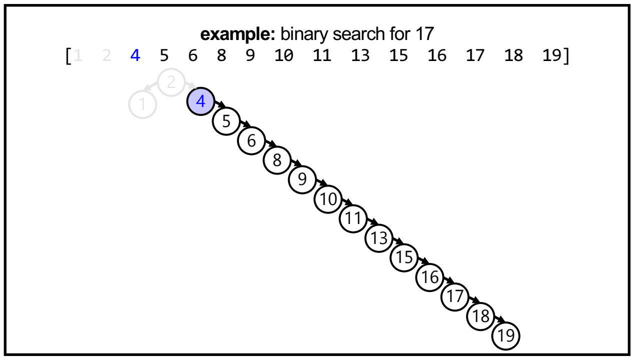
`[1  2  4  5  6  8  9  10  11  13  15  16  17  18  19]`

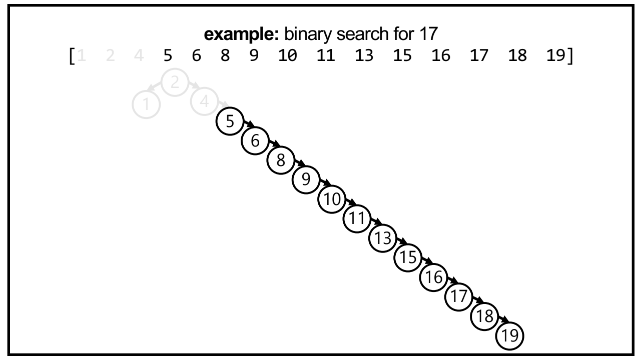


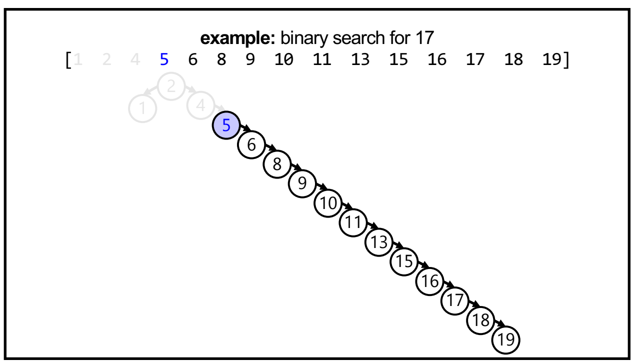
example: binary search for 17

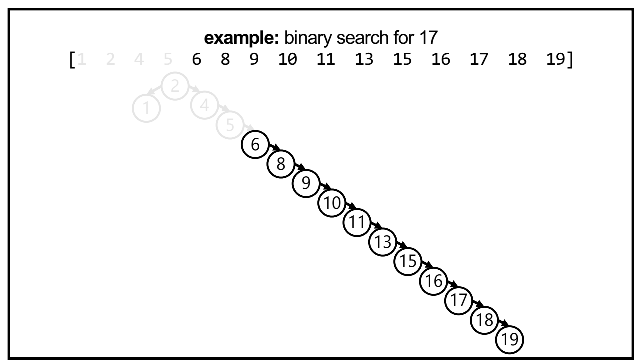
`[1  2  4  5  6  8  9  10  11  13  15  16  17  18  19]`

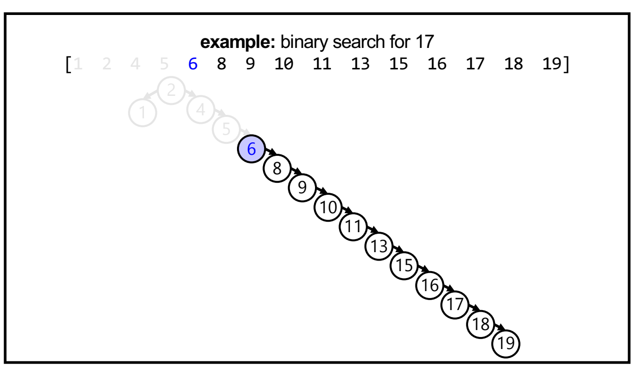


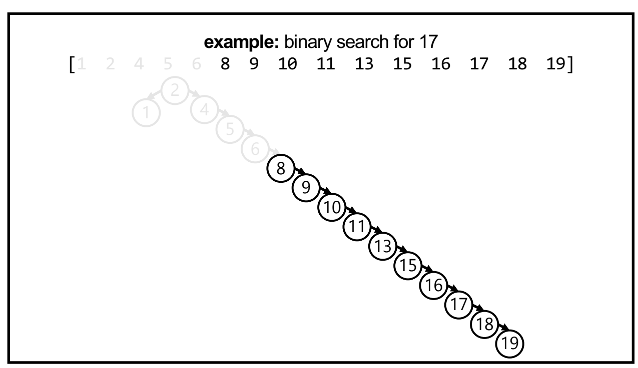

example: binary search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]


example: binary search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]


example: binary search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]


example: binary search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]


example: binary search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]


example: binary search for 17
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]

...

life lesson: it is important that your binary search tree is ✨ balanced ✨

(otherwise your "binary search" degrades into linear search of a linked list 😦)

adding a new node to a binary search tree

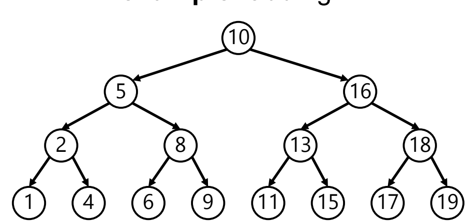
(the naive (simple but bad) method)

for a binary search tree, **`add`** starts out just like **`search`**

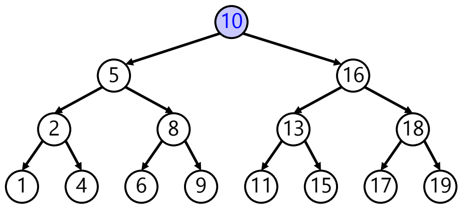
just keep going until you hit a `null` node

put the new node there 🙂 👍

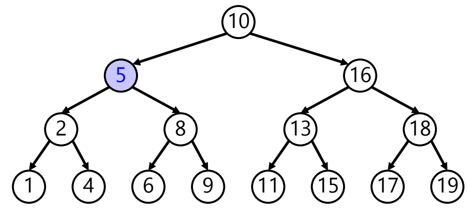
example: adding 7



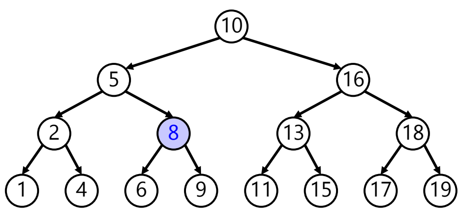
example: adding 7



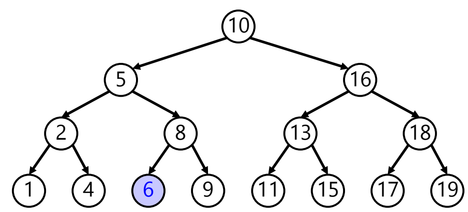
example: adding 7



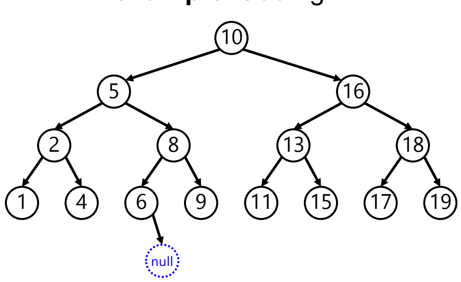
example: adding 7



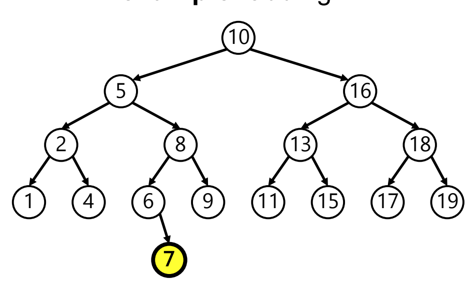
example: adding 7



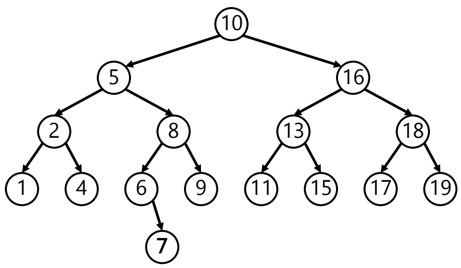
example: adding 7



example: adding 7



example: adding 7



is this approach to adding a new node good?

hint: no.

self-balancing
binary search trees

note: hard to implement

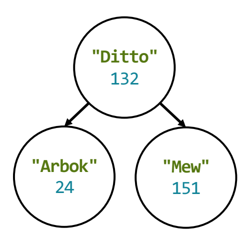
uses of binary search trees

tree map (implement a map)

- you can implement the map interface using a BST
  NOTE: this is NOT a hash map!—no hashing is involved!

```
class Node {
    String key; // NOTE: BST is sorted by key
    Integer value;
    Node leftChild;
    Node rightChild;
}

class TreeMap {
    Node root;
    ValueType get(KeyType) { ... }
    void put(ValueType, KeyType) { ... }
}
```



✨ Wednesday! ✨

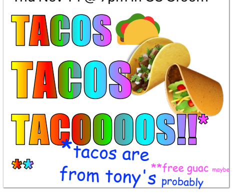
Week10b

- BST traversal order review
- heaps, heapsort

WARMUP
what does "sink or swim" mean?

ANNOUNCEMENTS
Jim out of town Wed-Fri
(TA's will run lab)
Tacos Tacos Tacos
tomorrow at 7 pm

CoSSAC Kick-Off Event
Thu Nov 14 @ 7pm in CS Croom



---

review: binary tree depth-first traversal orders

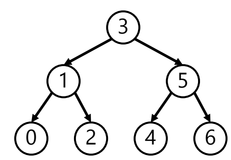
---

binary tree depth-first traversal orders

---

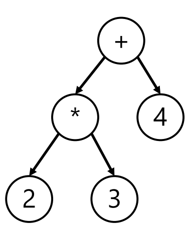
depth-first traversal orders (of a binary tree)

- **pre-order** = self, left, right
  - 3, 1, 0, 2, 5, 4, 6
- **in-order** = left, self, right
  - 0, 1, 2, 3, 4, 5, 6
- **post-order** = left, right, self
  - 0, 2, 1, 4, 6, 5, 3
- **reverse pre-order** = self, right, left
  - 3, 5, 6, 4, 1, 2, 0
- **reverse in-order** = right, self, left
  - 6, 5, 4, 3, 2, 1, 0
- **reverse post-order** = right, left, self
  - 6, 4, 5, 2, 0, 1, 3



---

2 * 3 + 4



who's that traversal order?

in-order!

---

the most satisfying function in CS 136

```
void recurse(Node self) {
    // NOTE: rearranging these 3 lines gives you all
    // 3! ("three factorial") = 6 traversal orders
    if (self.leftChild  != null) { recurse(self.leftChild);  }
    System.out.print(self.value + " ");
    if (self.rightChild != null) { recurse(self.rightChild); }
}
```

[Live-code traversal orders]
(Feel free to follow along or race.)

heaps

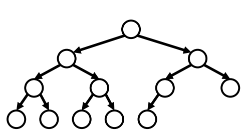
always-complete max binary heap

always-complete max binary heap

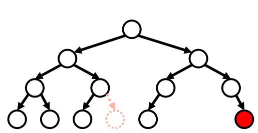
- in this class, when we say "**heap**" or "**max heap**", we mean an "**always-complete max binary heap**"
  - we might occasionally mention a "**min heap**", which means an "**always-complete min binary heap**"

always-complete max binary heap

- in a **complete binary tree**, all levels (depths, rows) are "full of nodes", except for possibly the bottom level, in which all nodes are "as far to to the left as possible"



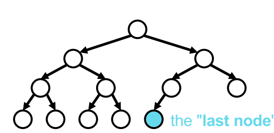
complete
(and balanced)



balanced but **NOT complete**

always-complete max binary heap

- in a **complete binary tree**, all levels (depths) are "full of nodes", except for possibly the bottom level, in which all nodes are "as far to to the left as possible"



complete
(and balanced)

always-complete max binary heap

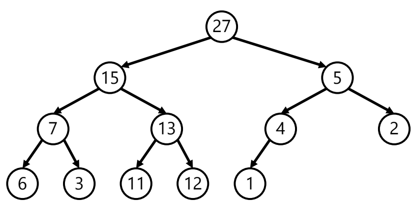
- "**always-complete**" means that every function in the Heap interface `(add(...)` & `remove())` "preserves the completeness of the heap"
  - the heap **was complete** before calling `add`...
  - ...and the heap **is still complete** after `add` returns
  - the heap is always complete
  - always complete.

always-complete max binary heap

- a **binary heap** is another special kind of binary tree
  - **a binary heap is NOT, in general, a binary search tree**

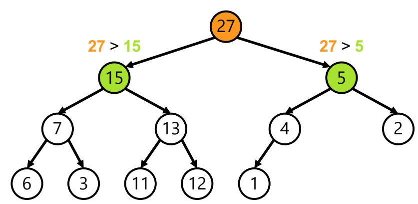
always-complete max binary heap

- in a **max binary heap**, **every node** follows: "a node's value is greater than the value of its left child and the value of its right child"



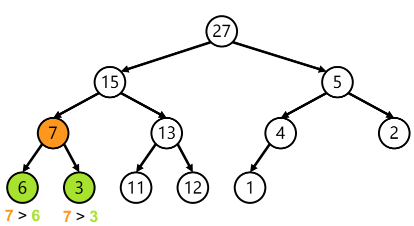
always-complete max binary heap

- in a **max binary heap**, **every node** follows: "a node's value is greater than the value of its left child and the value of its right child"



always-complete max binary heap

- in a **max binary heap**, **every node** follows: "a node's value is greater than the value of its left child and the value of its right child"

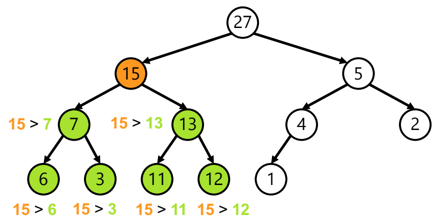


always-complete max binary heap

- in a **max binary heap**, **every node** follows: "a node's value is greater than the value of its left child and the value of its right child"
- is the "**max heap property**" above equivalent to...
  - in a **max binary heap**, **every node** follows: "a node's value is greater than the values of all its descendants"
    - yes.
      - **idea:** apply definition recursively
        - node's value is greater than the values of its children... ...which are greater than the values of their children...
- which node always has the max value of all nodes in the heap?
  - the root

always-complete max binary heap

- in a **max binary heap**, **every node** follows: "a node's value is greater than the values of all its descendants"



heap interface

`add(...)` & `remove()`

a heap is like a benthic trawl net (ish)

heap interface

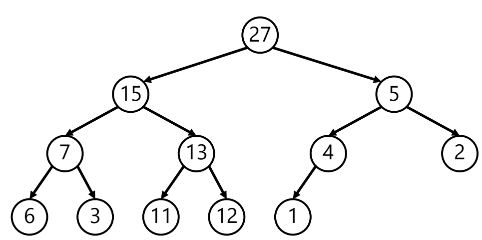
- ```
  // Add this value to the heap.
  void add(ValueType value) { ... }
  ```
- ```
  // Remove the max value from the heap, and return it.
  ValueType remove() { ... }
  ```

add(...)

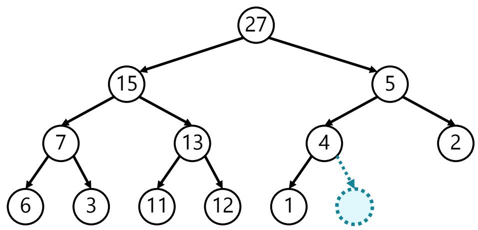
`void add(ValueType value);`

- to **add** a new node with a given value to a max binary heap...
  - add the new node so that the heap is still complete (add into "the next empty slot")
  - while that node violates the max heap property...
    - swap it with its parent
- the node **"swims up"**
  - "**sifts up**"
  - "heap up"?

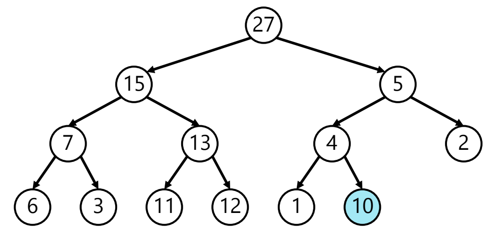
example: adding 10



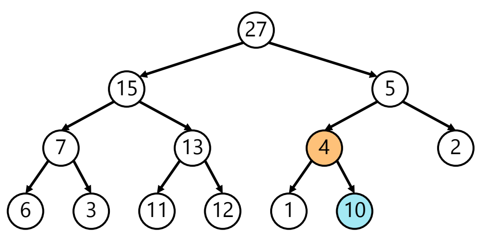
example: adding 10



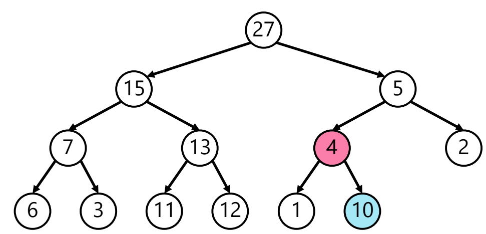
example: adding 10



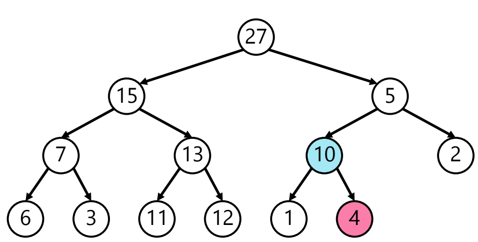
example: adding 10



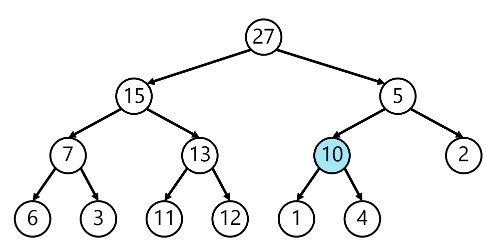
example: adding 10



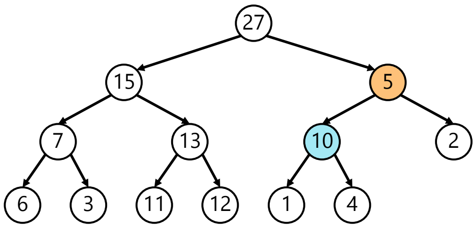
example: adding 10



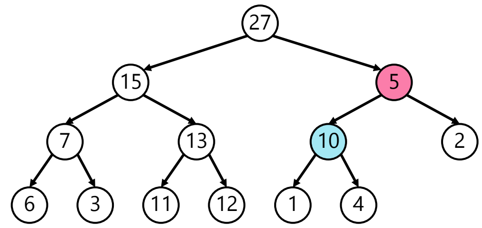
example: adding 10



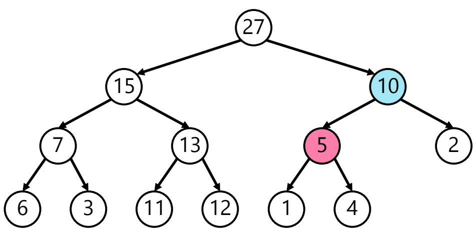
example: adding 10



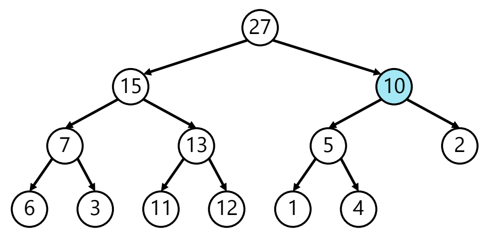
example: adding 10



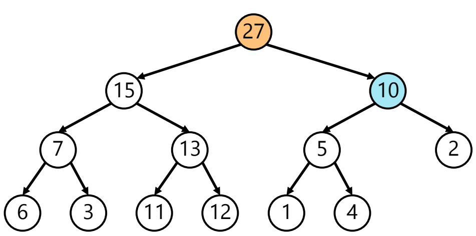
example: adding 10



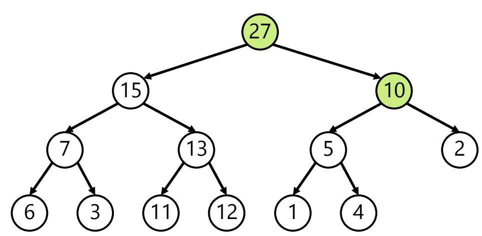
example: adding 10



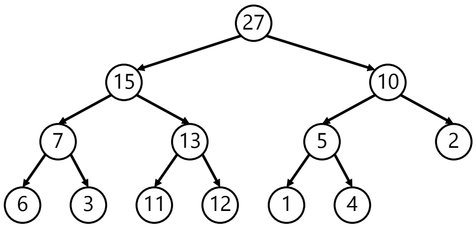
example: adding 10



example: adding 10



example: adding 10

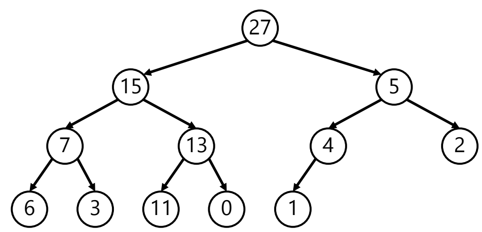


`remove()`

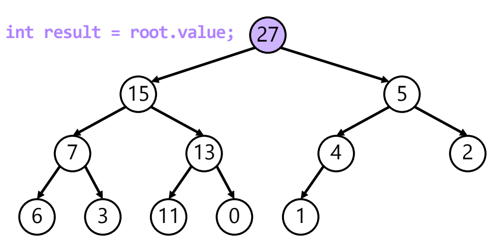
`ValueType remove();`

- to **remove** the node with max value (the root) from a max binary heap...
  - save the root's value in a temporary variable called `result`
  - replace the root with the **last node** (rightmost node in the bottom level) (the old root is now "garbage" and ready to be garbage collected 🚛)
  - while that node violates the max heap property...
    - swap it with its larger child
  - `return result;`
- the node "**sinks down**" ⚓
  - "**sifts down**"
  - "heap down"?

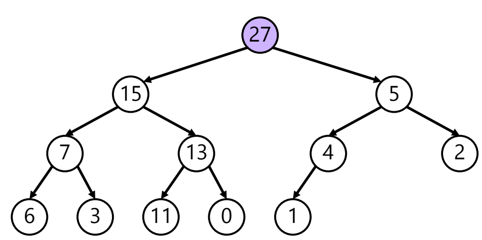
example: removing max node



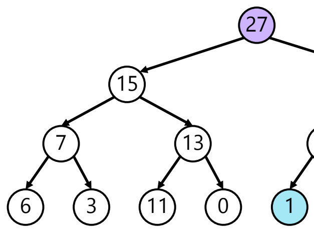
example: removing max node



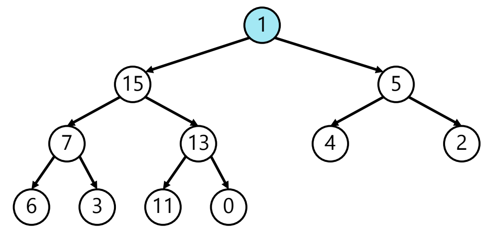
example: removing max node



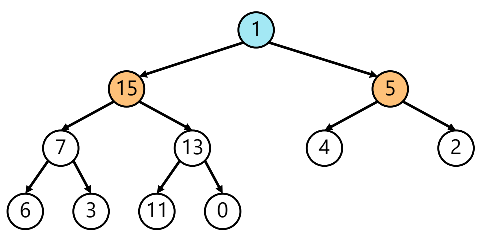
example: removing max node



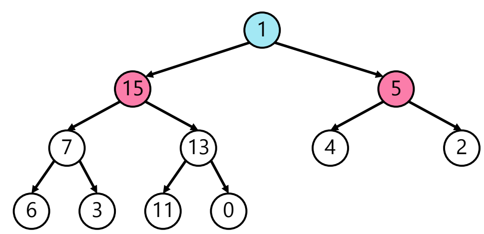
example: removing max node



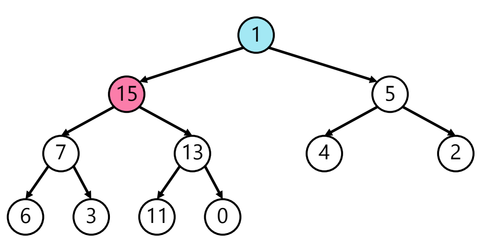
example: removing max node



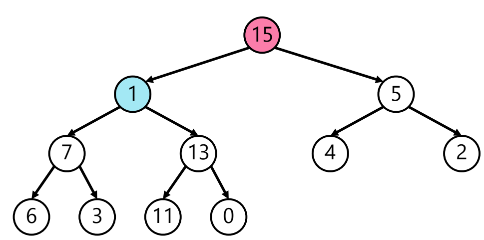
example: removing max node



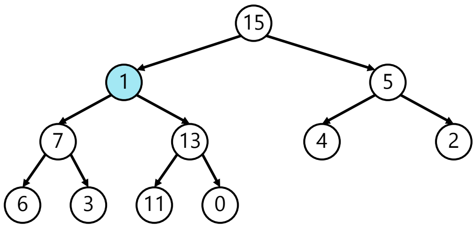
example: removing max node



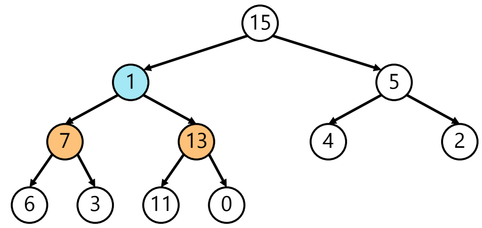
example: removing max node



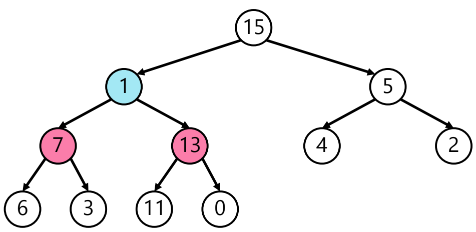
example: removing max node



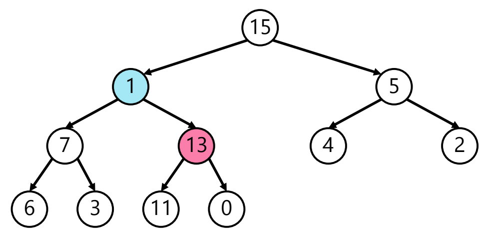
example: removing max node



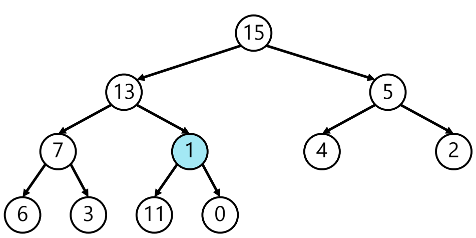
example: removing max node



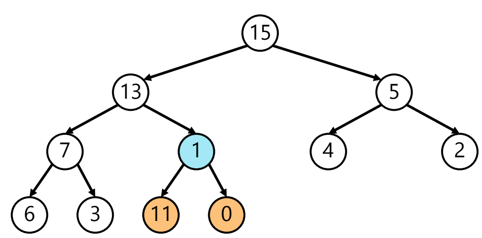
example: removing max node



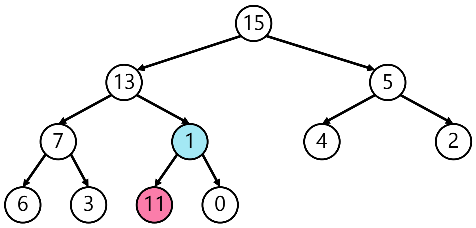
example: removing max node



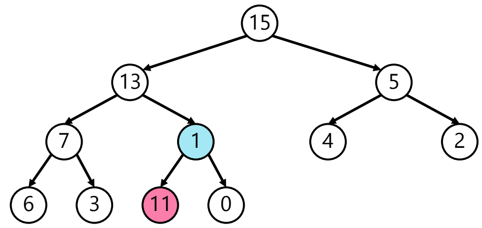
example: removing max node



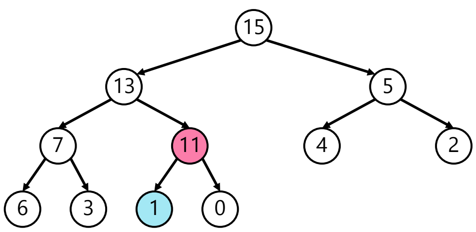
example: removing max node



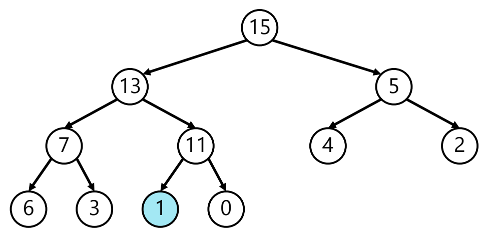
example: removing max node



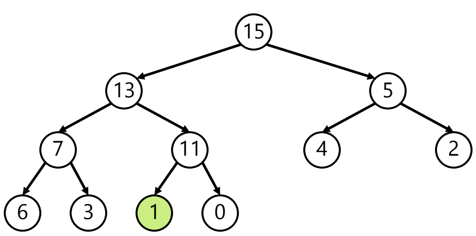
example: removing max node



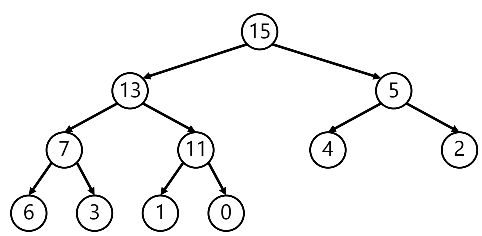
example: removing max node



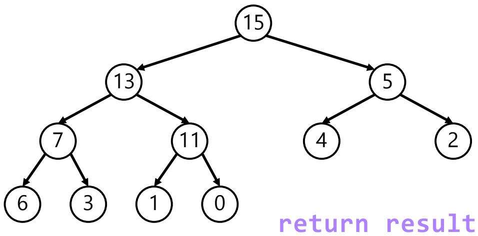
example: removing max node



example: removing max node



example: removing max node

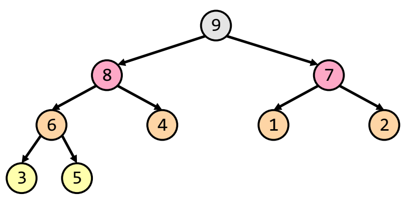


`return result;`

(implicit) heapsort

because a heap is an always-complete binary tree, **we can store a heap "implicitly" as an array** using a breadth-first (level-order) traversal!

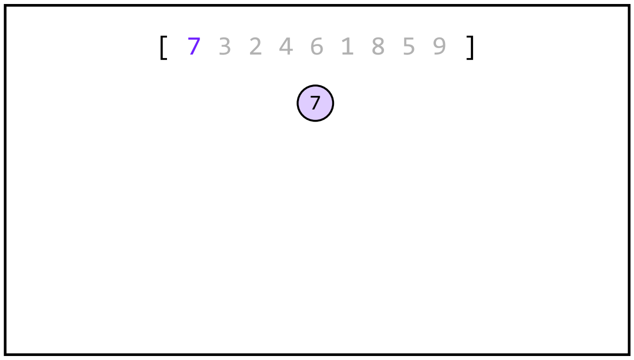
[9 8 7 6 4 1 2 3 5]

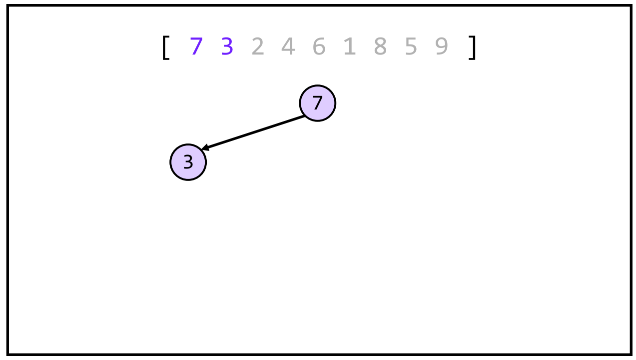


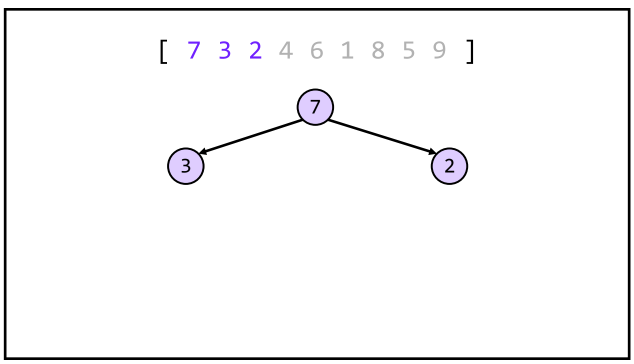
this lets us do (in-place) **implicit heapsort**!
(using only swaps)

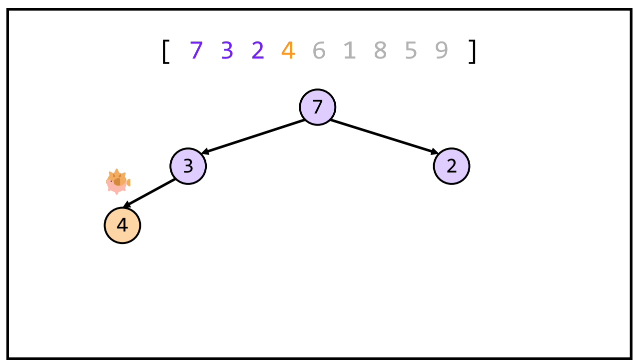
1. build a heap by calling **add(...)** over and over
2. deconstruct the heap by calling **remove()** over and over

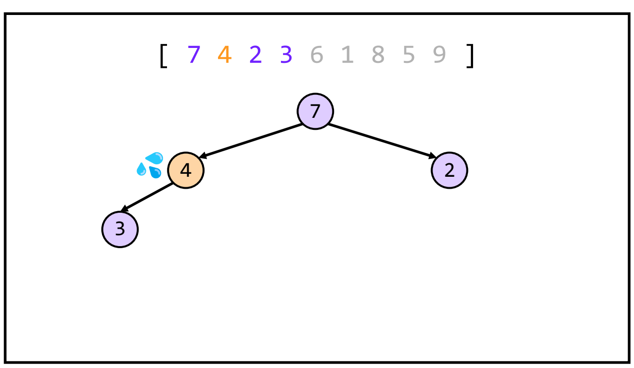
[7 3 2 4 6 1 8 5 9]

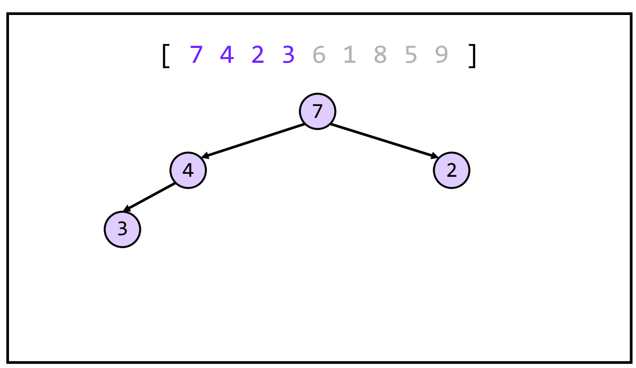

[7 3 2 4 6 1 8 5 9]
7

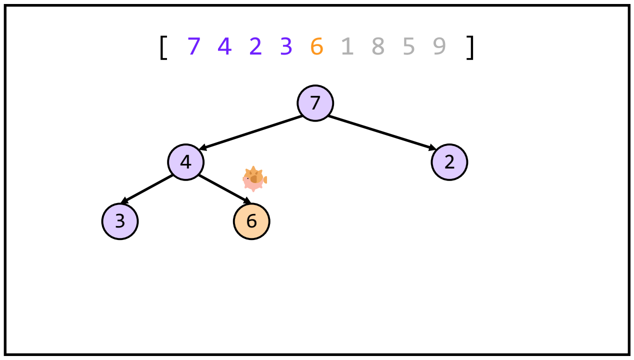

[7 3 2 4 6 1 8 5 9]
7
3

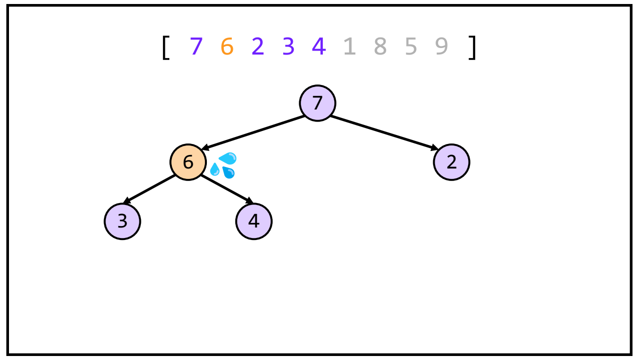

[7 3 2 4 6 1 8 5 9]
7
3
2

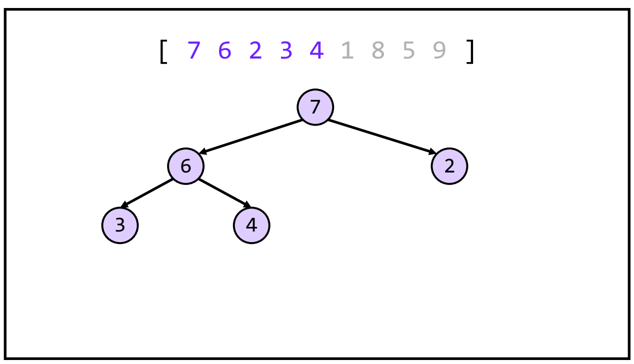

[7 3 2 4 6 1 8 5 9]
7
3
2
4

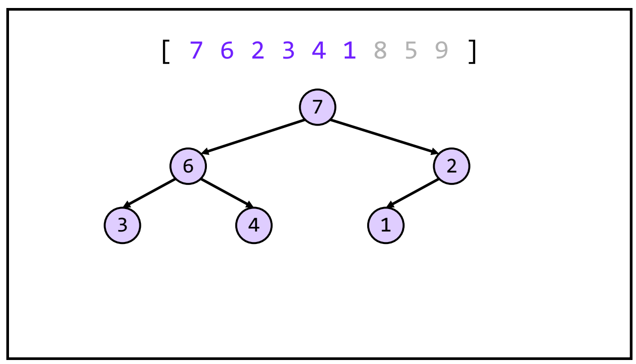

[7 4 2 3 6 1 8 5 9]
7
4
2
3

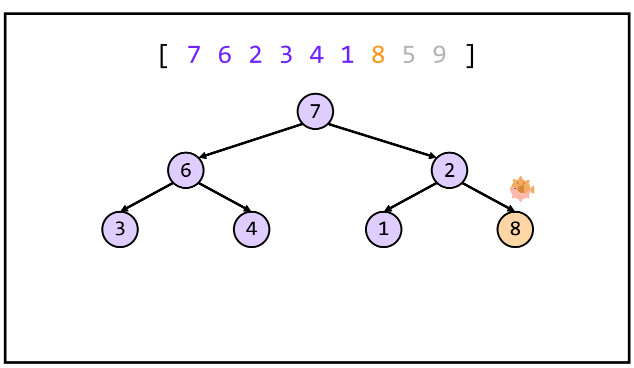

[7 4 2 3 6 1 8 5 9]
7
4
2
3

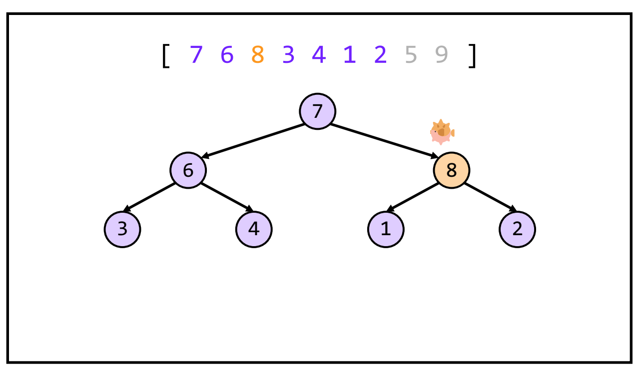

[7 4 2 3 6 1 8 5 9]
7
4
2
3
6

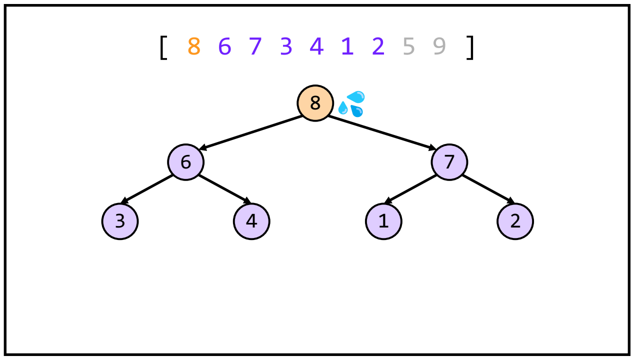

[7 6 2 3 4 1 8 5 9]
7
6
2
3
4

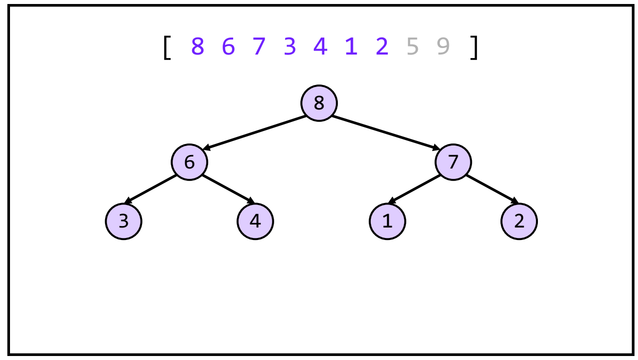

[7 6 2 3 4 1 8 5 9]
7
6
2
3
4

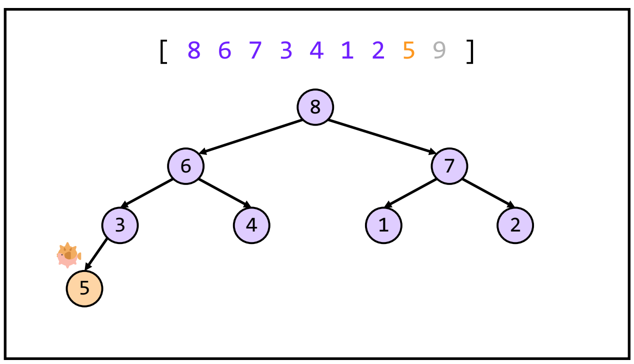

[7 6 2 3 4 1 8 5 9]
7
6
2
3
4
1

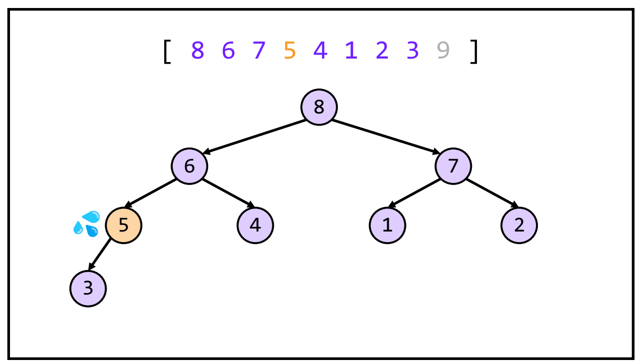

[7 6 2 3 4 1 8 5 9]
7
6
2
3
4
1
8

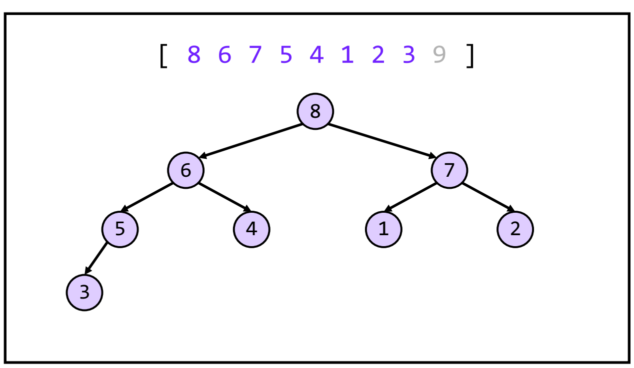

[7 6 8 3 4 1 2 5 9]
7
6
8
3
4
1
2

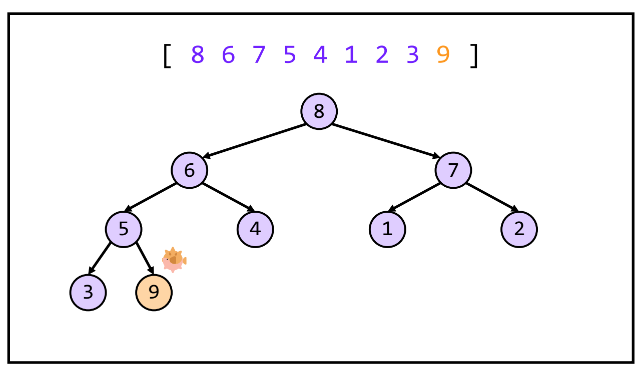

[8 6 7 3 4 1 2 5 9]

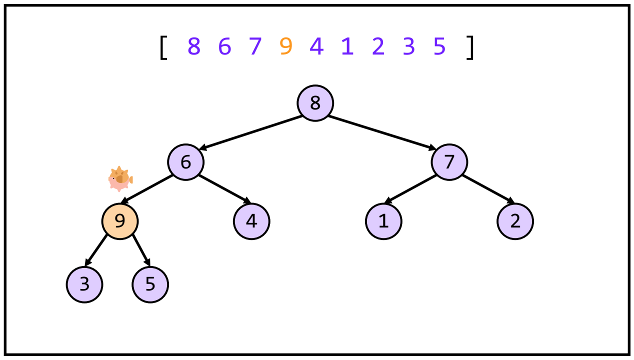

[8 6 7 3 4 1 2 5 9]

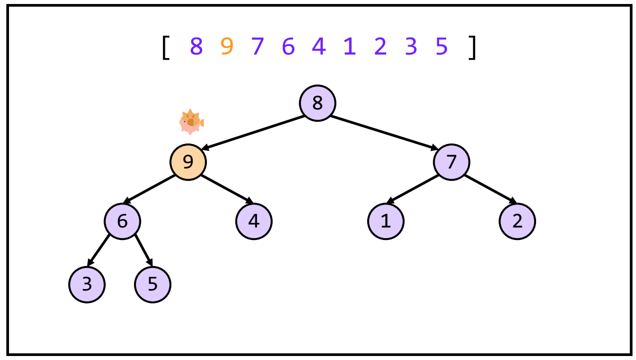

[8 6 7 3 4 1 2 5 9]

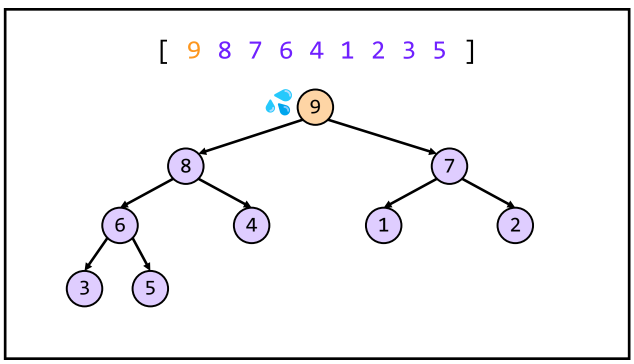

[8 6 7 5 4 1 2 3 9]

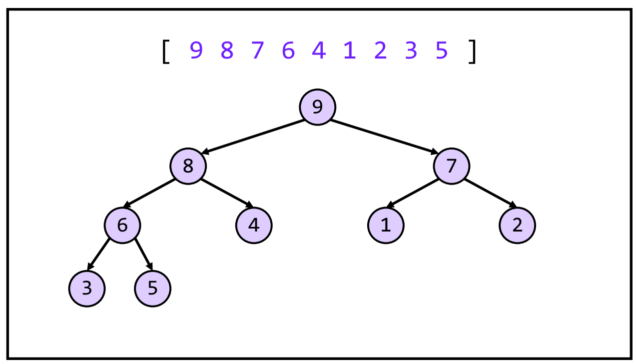

[8 6 7 5 4 1 2 3 9]

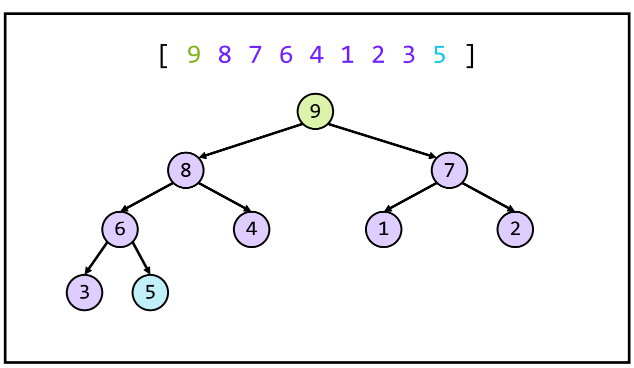

[8 6 7 5 4 1 2 3 9]

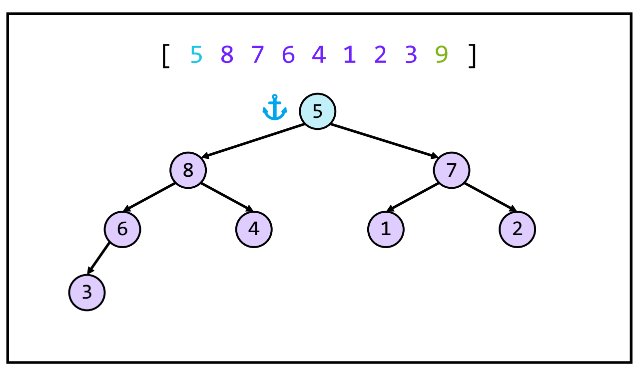

[8 6 7 9 4 1 2 3 5]

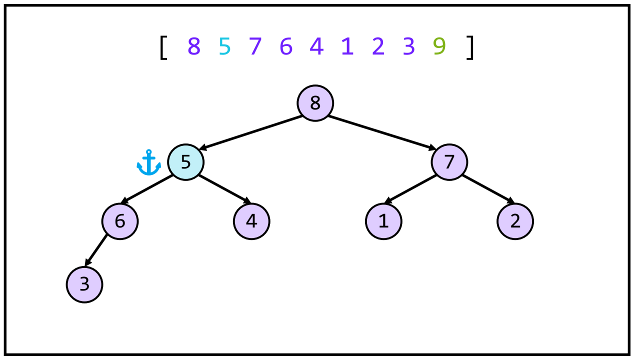

[8 9 7 6 4 1 2 3 5]

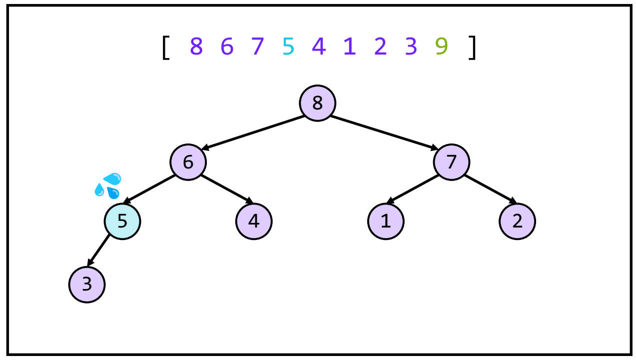

[9 8 7 6 4 1 2 3 5]

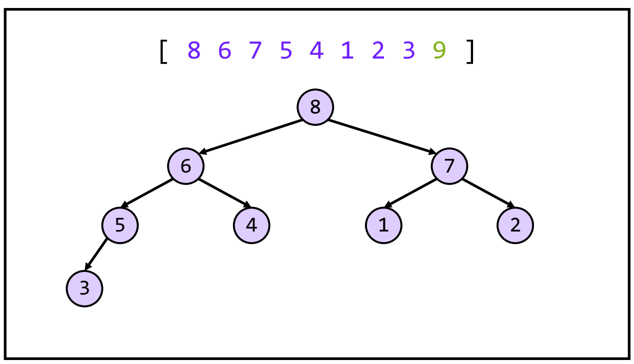

[9 8 7 6 4 1 2 3 5]

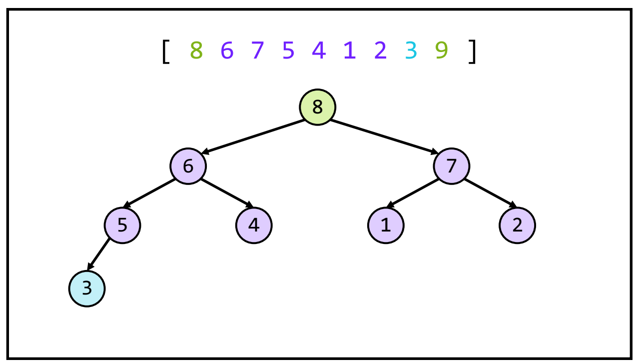

[9 8 7 6 4 1 2 3 5]

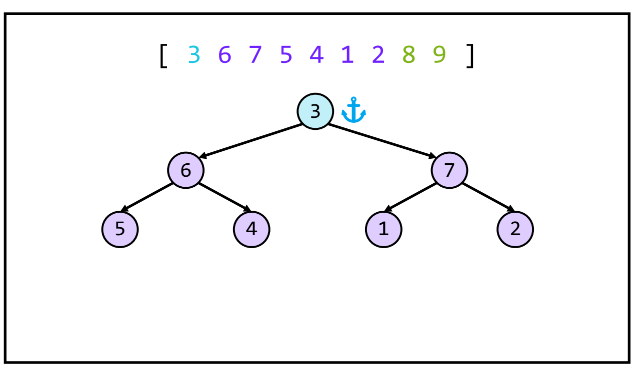

[5 8 7 6 4 1 2 3 9]

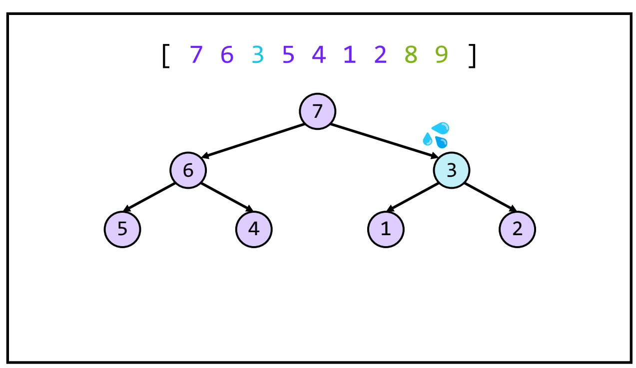

[8 5 7 6 4 1 2 3 9]
8
5
7
6
4
1
2
3

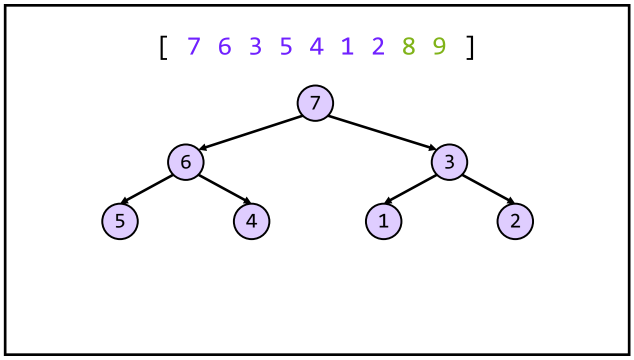

[8 6 7 5 4 1 2 3 9]
8
6
7
5
4
1
2
3

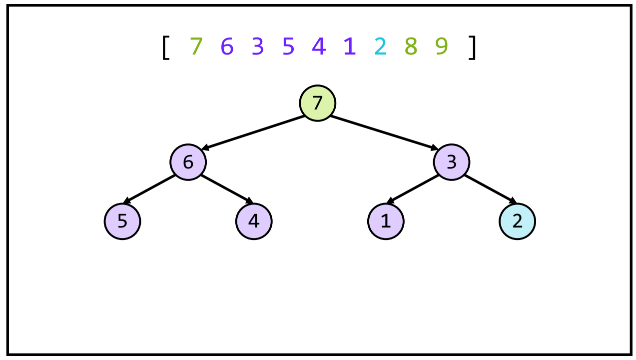

[8 6 7 5 4 1 2 3 9]
8
6
7
5
4
1
2
3

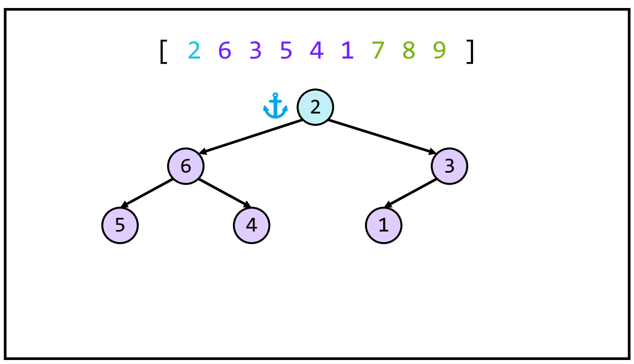

[8 6 7 5 4 1 2 3 9]
8
6
7
5
4
1
2
3

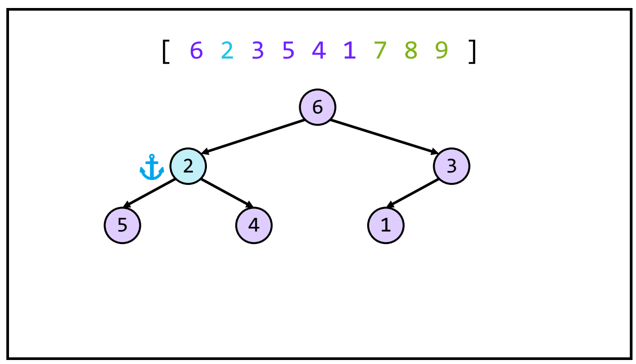

[3 6 7 5 4 1 2 8 9]
3
6
7
5
4
1
2

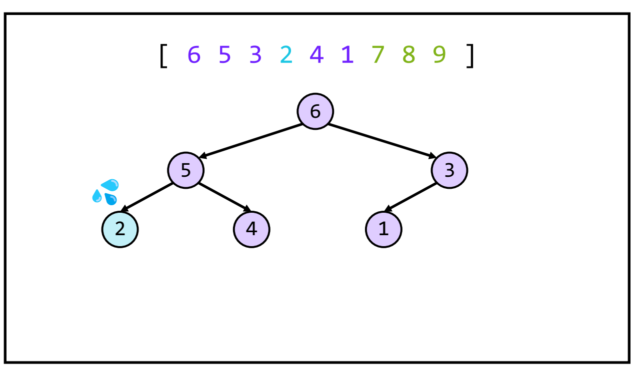

[7 6 3 5 4 1 2 8 9]
7
6
3
5
4
1
2

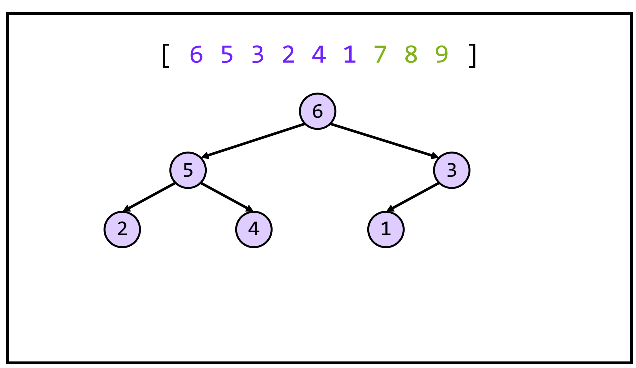

[7 6 3 5 4 1 2 8 9]

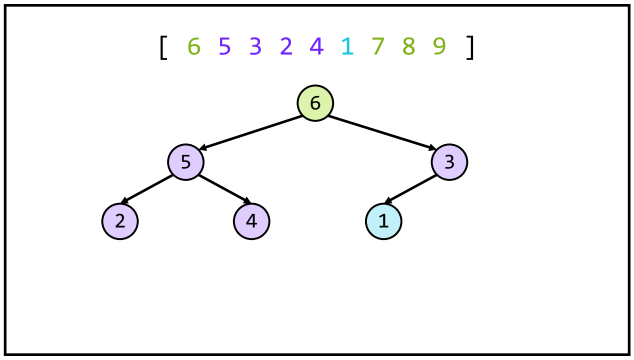

[7 6 3 5 4 1 2 8 9]

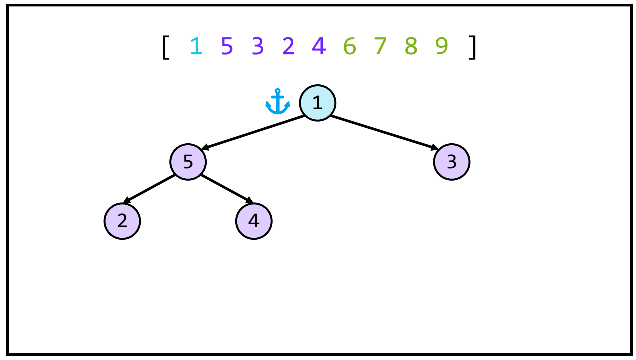

[2 6 3 5 4 1 7 8 9]

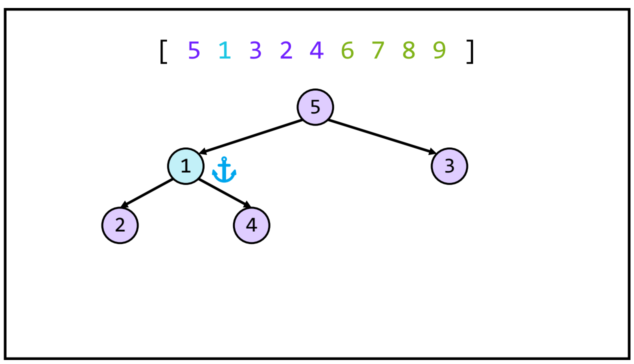

[6 2 3 5 4 1 7 8 9]

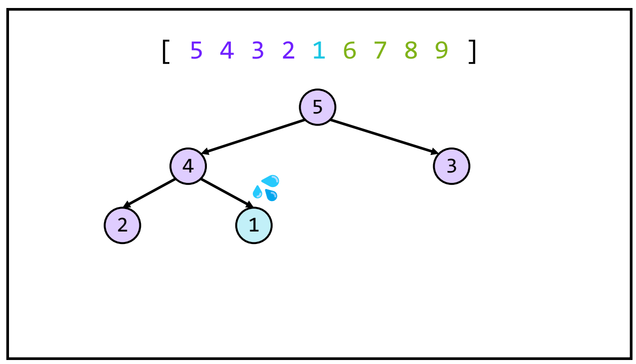

[6 5 3 2 4 1 7 8 9]

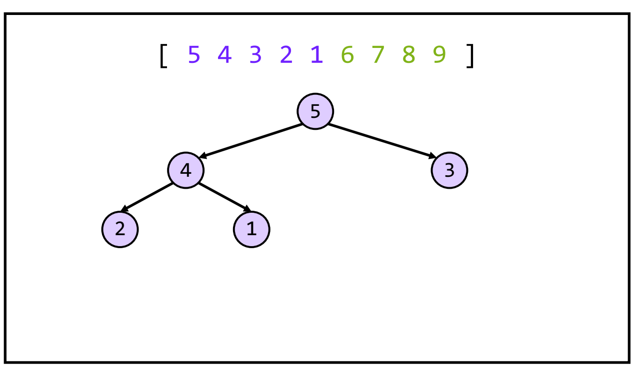

[6 5 3 2 4 1 7 8 9]

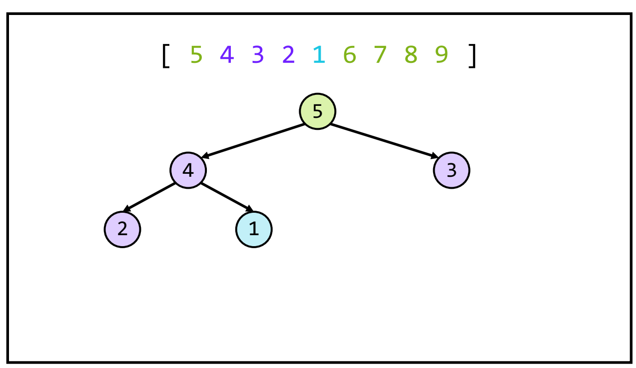

[6 5 3 2 4 1 7 8 9]
6
5
3
2
4
1

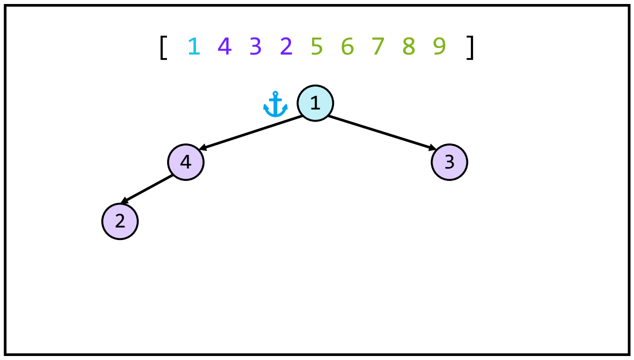

[1 5 3 2 4 6 7 8 9]
1
5
3
2
4

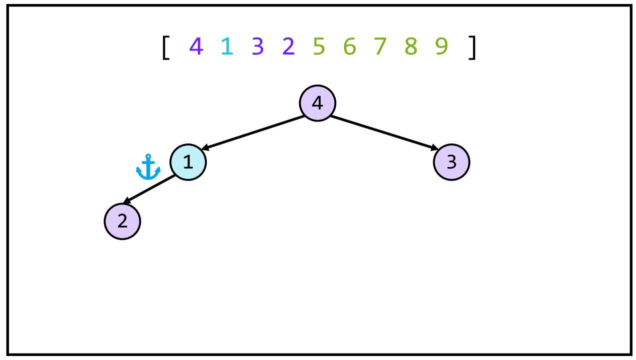

[5 1 3 2 4 6 7 8 9]
5
1
3
2
4

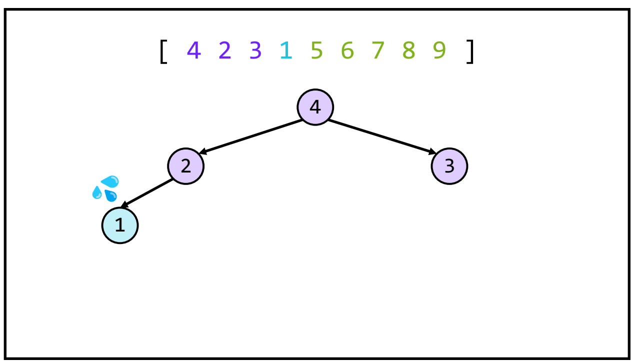

[5 4 3 2 1 6 7 8 9]
5
4
3
2
1

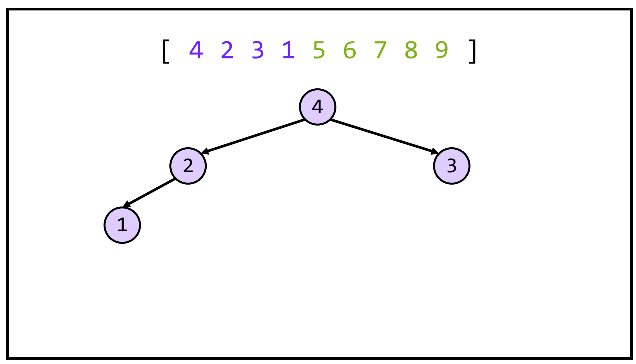

[5 4 3 2 1 6 7 8 9]
5
4
3
2
1

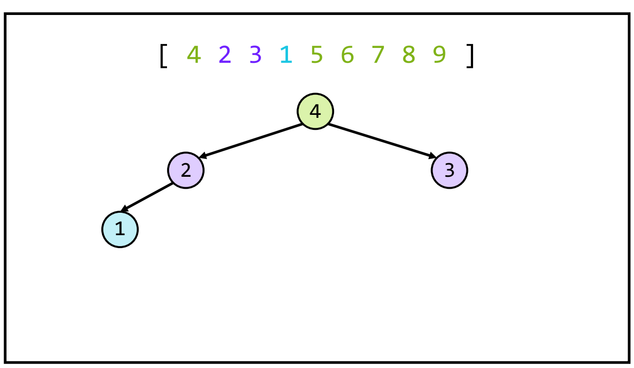

[5 4 3 2 1 6 7 8 9]
5
4
3
2
1

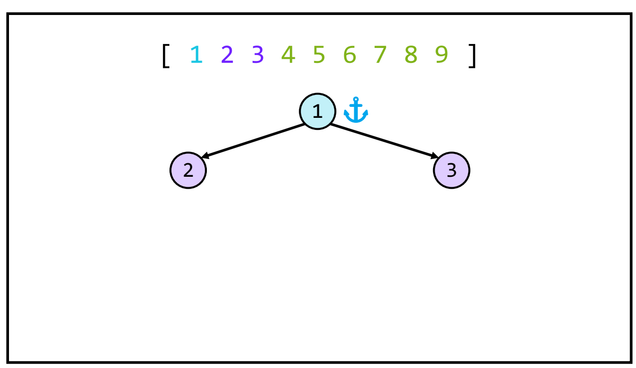

[1 4 3 2 5 6 7 8 9]
1
4
3
2

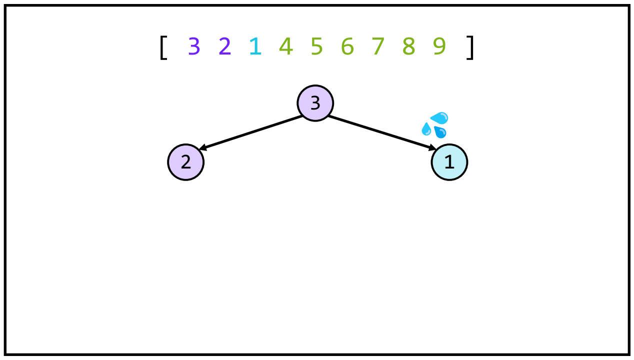

[4 1 3 2 5 6 7 8 9]
4
1
3
2

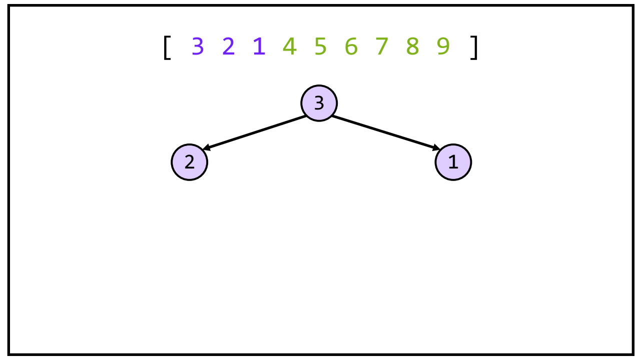

[4 2 3 1 5 6 7 8 9]
4
2
3
1

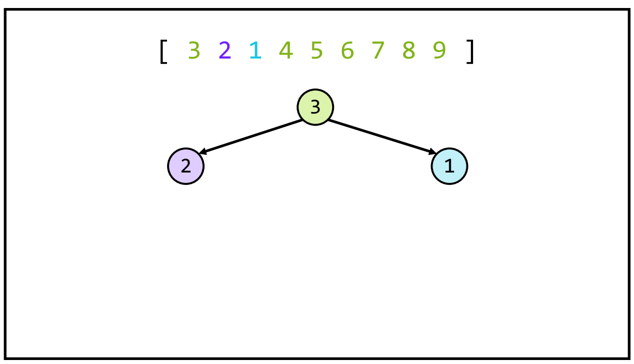

[4 2 3 1 5 6 7 8 9]
4
2
3
1

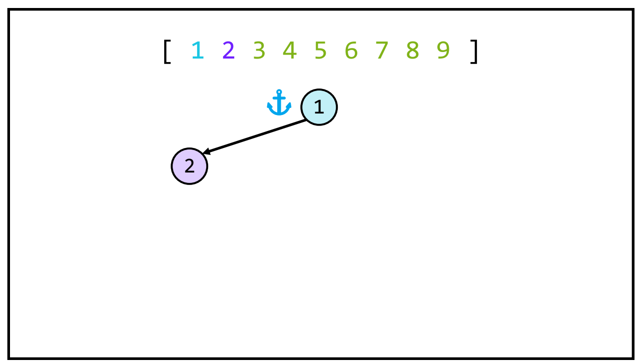

[4 2 3 1 5 6 7 8 9]
4
2
3
1

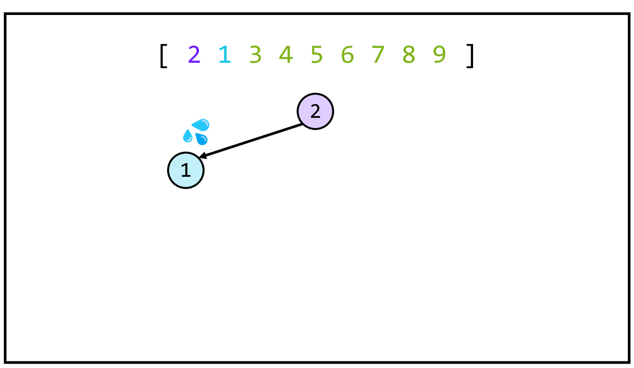

[1 2 3 4 5 6 7 8 9]
1
2
3

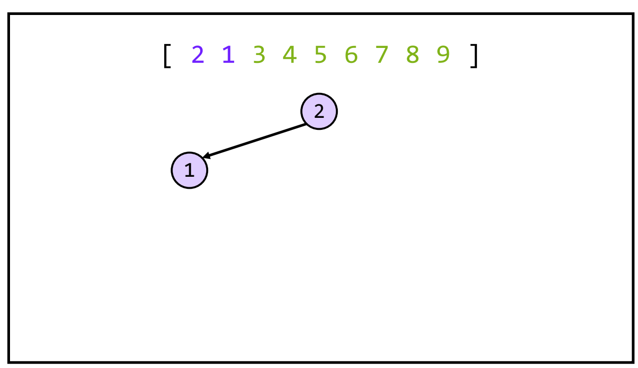

[3 2 1 4 5 6 7 8 9]
3
2
1


[3 2 1 4 5 6 7 8 9]
3
2
1

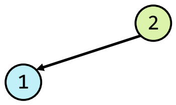

[3 2 1 4 5 6 7 8 9]
3
2
1


[1 2 3 4 5 6 7 8 9]
1
2


[2 1 3 4 5 6 7 8 9]
2
1


[2 1 3 4 5 6 7 8 9]
2
1

[2 1 3 4 5 6 7 8 9]



[1 2 3 4 5 6 7 8 9]

[1 2 3 4 5 6 7 8 9]

[1 2 3 4 5 6 7 8 9]

[1 2 3 4 5 6 7 8 9]

heap application:
priority queue

review: queue

remove();

review: queue

remove();

review: queue

add();

review: queue

add();

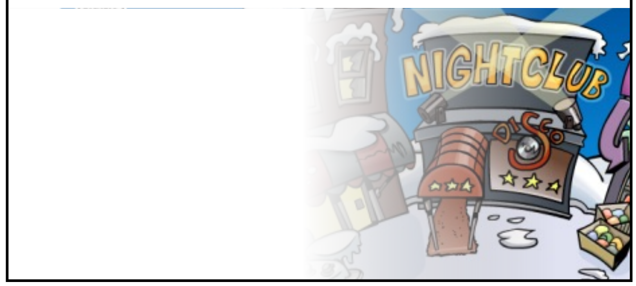
review: queue

remove();

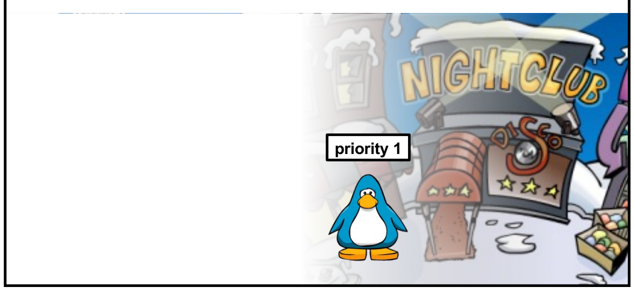
review: queue

extension: priority queue

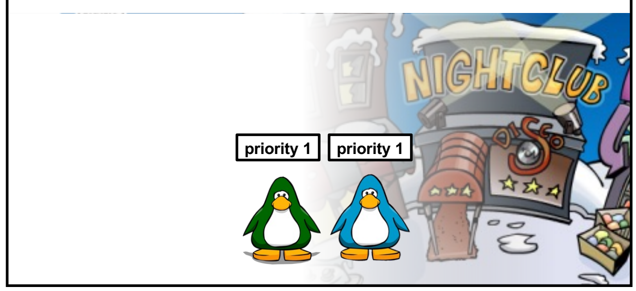
extension: priority queue



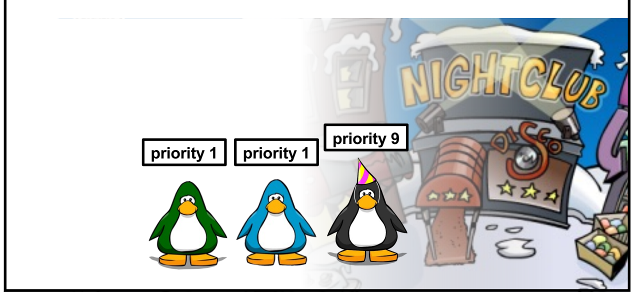
extension: priority queue



extension: priority queue

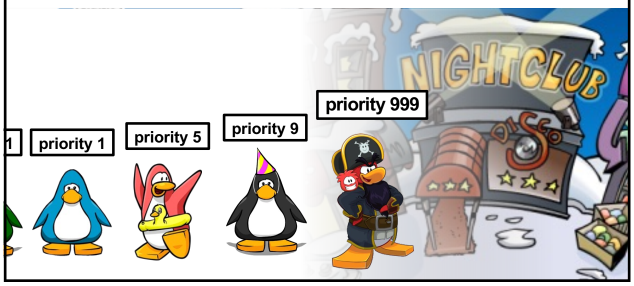


extension: priority queue

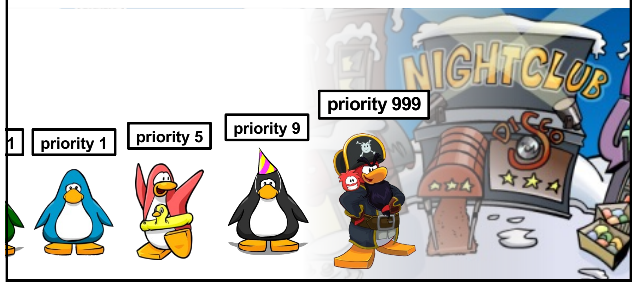


extension: priority queue

extension: priority queue



extension: priority queue



a **priority queue**'s remove() function removes the element with **highest priority**

a **max heap**'s remove() function removes the node with **maximum value**

✨ a max heap is a natural way to implement a priority queue ✨