ANNOUNCEMENTS
today is Pre-Recoreded Lecture Monday ☹️
see you back in person on Wednesday!

WARMUP
imagine searching for a tube of toothpaste in the real world
how would find one as quickly as possible?

TODAY
binary search trees

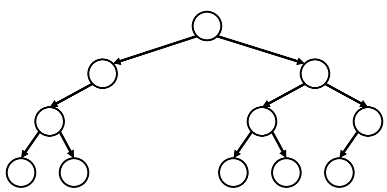
840

binary trees

841

binary tree

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



842

🧠 is a linked list a binary tree?

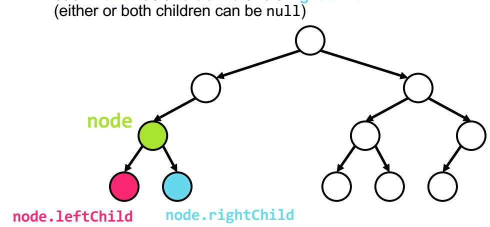
843

technically, yes
all nodes have <= 2 children
(tail has 0 children, all other nodes have 1 child)

844

(ordered) binary tree

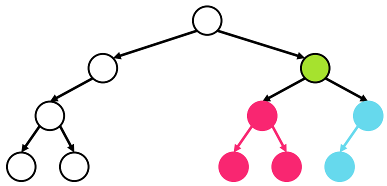
- each node has a left child and a right child
  (either or both children can be `null`)



845

(ordered) binary tree

- each node has a **left subtree** and a **right subtree**



846

sort & search

847

sort

848

sorted

- a list / array 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)

849

search

850

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 search(int[] array, int value) { ... }
```

851

linear search
(of a list/array)

852

linear 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;
    }
    ```

853

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

854

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

855

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

856

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

857
    ```

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

858

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

859

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

860

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

861

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

862

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

863

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

864

865

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

866

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

867

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

868

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

869

example: linear search for 17

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

870

example: linear search for 17

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

871

example: linear search for 17

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

872

example: linear search for 17

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

873

example: linear search for 17

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

874

example: linear search for 17

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

875

example: linear search for 17

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

876

example: linear search for 17

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

877

example: linear search for 17

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

878

example: linear search for 17

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

879

example: linear search for 17

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

880

example: linear search for 17

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

881

example: linear search for 17

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

882

example: linear search for 17

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

883

example: linear search for 17

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

884

example: linear search for 17

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

885

example: linear search for 17

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

886

example: linear search for 17

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

887

example: linear search for 17

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

888

example: linear search for 17

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

889

example: linear search for 17

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

890

example: linear search for 17

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

891

example: linear search for 17

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

892

if we know that a list is sorted...
can we search it faster?

893

yes 🚀

894

binary search
(of a list/array)

895

binary search

- **binary search** "cuts the list in half" over and over
  - **binary search only works when the list is sorted**
  - binary search is **O(log(n))** 😀

896

example: binary search for 17

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

897

example: binary search for 17

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

10 < 17

898

example: binary search for 17

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

899

example: binary search for 17

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

16 < 17

900

example: binary search for 17

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

901

example: binary search for 17

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

17 < 18

902

example: binary search for 17

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

903

example: binary search for 17

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

17 == 17

904

example: binary search for 17

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

905

implementating binary search is actually pretty tricky!

make sure you test thoroughly!

906

✨ **hint:** binary search

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

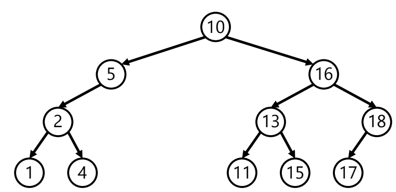
907

binary search tree

908

binary search tree

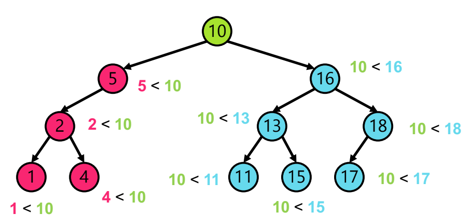
- in a **binary search tree** (sorted binary tree) **every** node follows: "a node's value is **greater than the values of all nodes in its left subtree** and **less than the values of all nodes in its right subtree**"



909

binary search tree

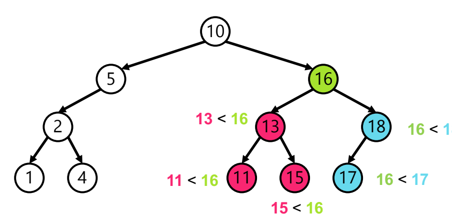
- in a **binary search tree** (sorted binary tree) every node follows: "a node's value is **greater than the values of all nodes in its left subtree** and **less than the values of all nodes in its right subtree**"



910

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**"



911

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**"

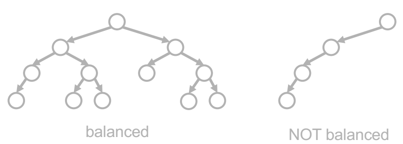
912

binary search
(of a binary search tree)

913

binary search

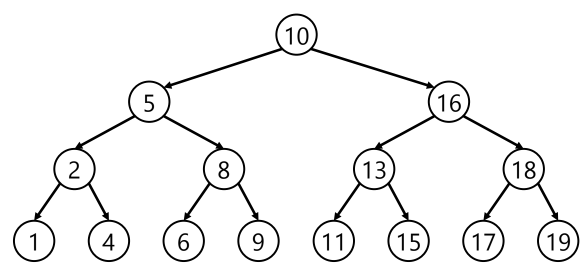
- **binary search** "cuts the tree in half" over and over
  - **binary search only works when the binary tree is sorted (is a binary search tree)**
  - binary search is **O(log(n))** if the tree is **balanced** 😀
    - balanced means each node's children are similar in **height** (we will talk more about balance on Friday)



914

example: binary search for 17

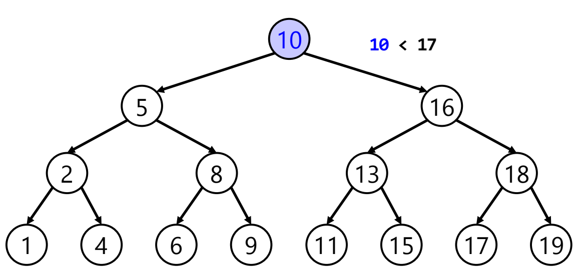
`[1  2  4  5  6  8  9  10  11  13  15  16  17  18  19]`



915

example: binary search for 17

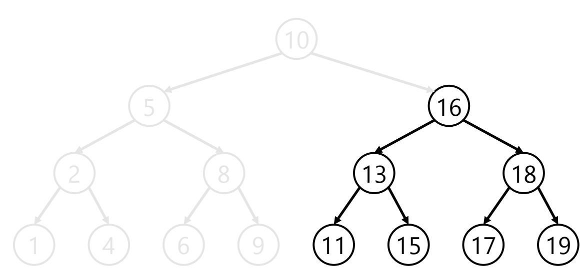
`[1  2  4  5  6  8  9  10  11  13  15  16  17  18  19]`



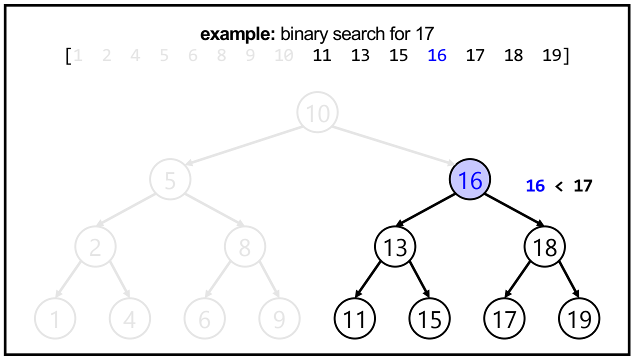
916

example: binary search for 17

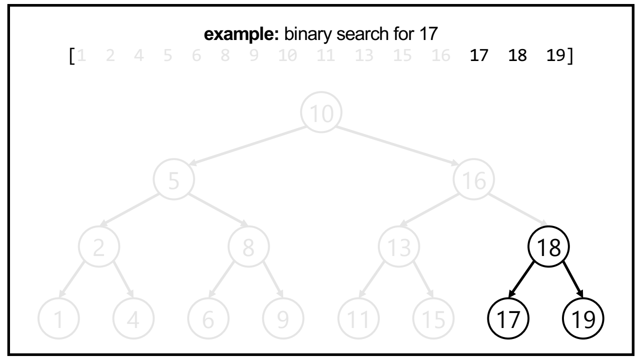
`[1  2  4  5  6  8  9  10  11  13  15  16  17  18  19]`



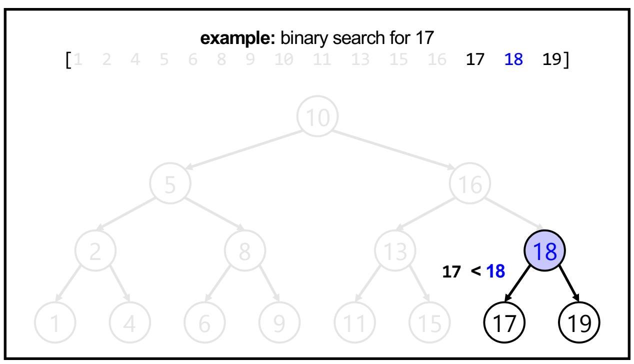
917



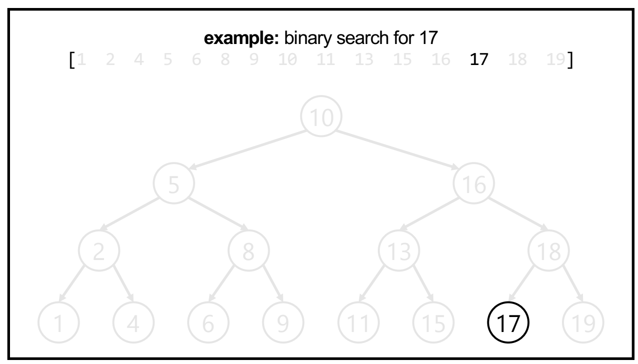
918



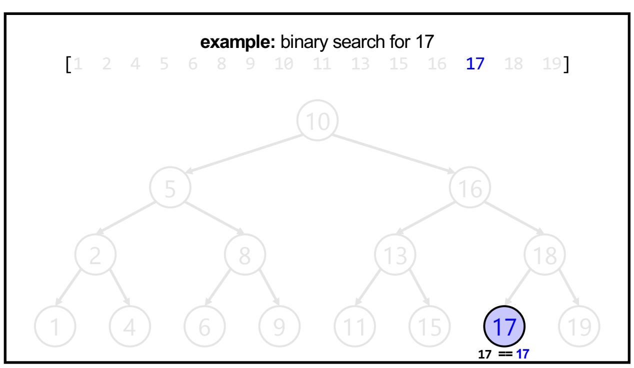
919



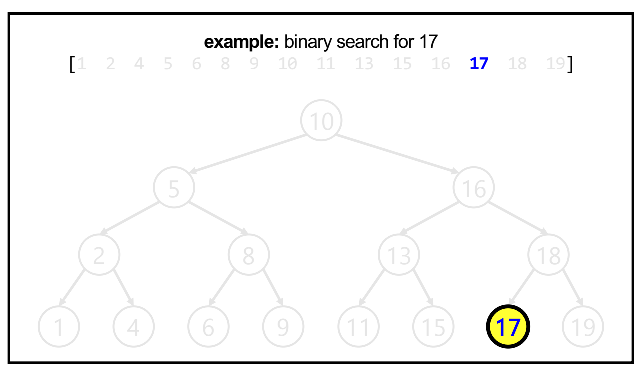
920



921



922



923

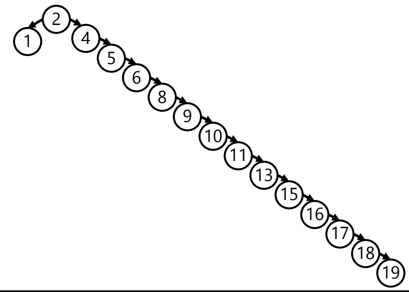
note: there are many different binary search trees that represent the same data

let's look at another one!

924

example: binary search for 17

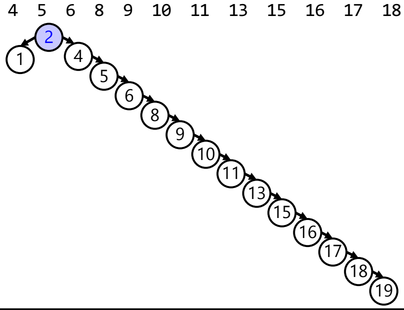
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]



925

example: binary search for 17

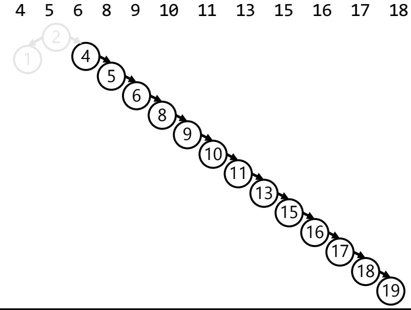
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]



926

example: binary search for 17

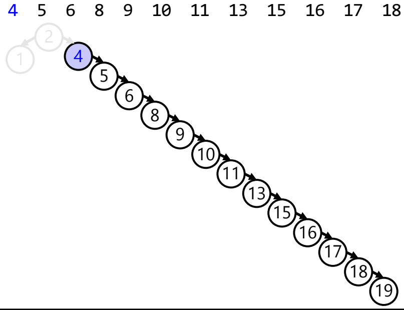
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]



927

example: binary search for 17

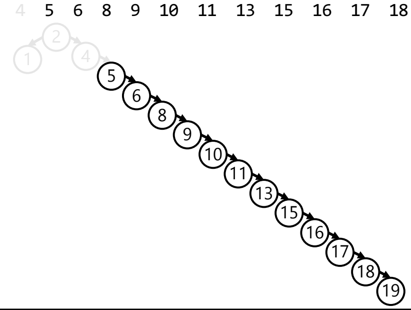
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]



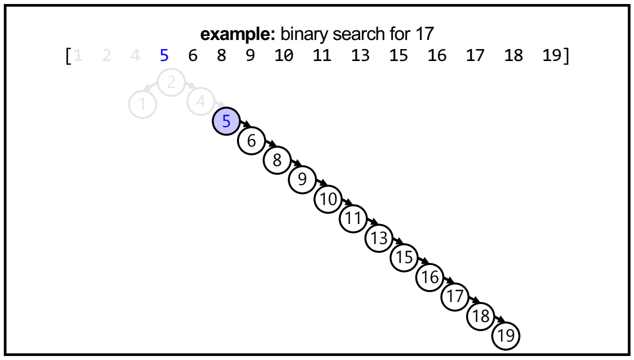
928

example: binary search for 17

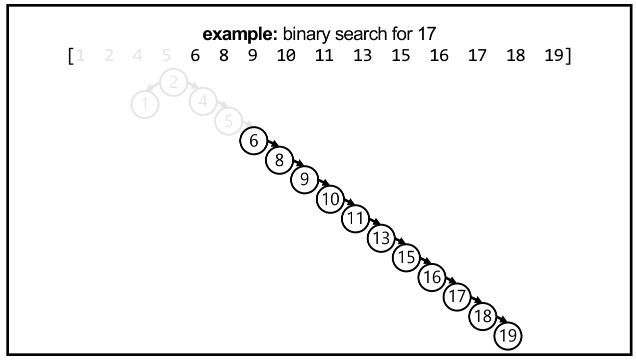
[1 2 4 5 6 8 9 10 11 13 15 16 17 18 19]



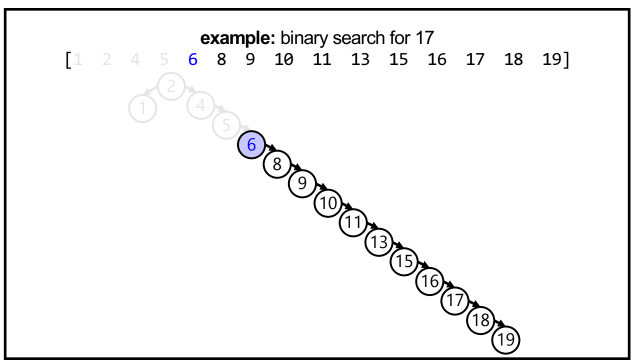
929



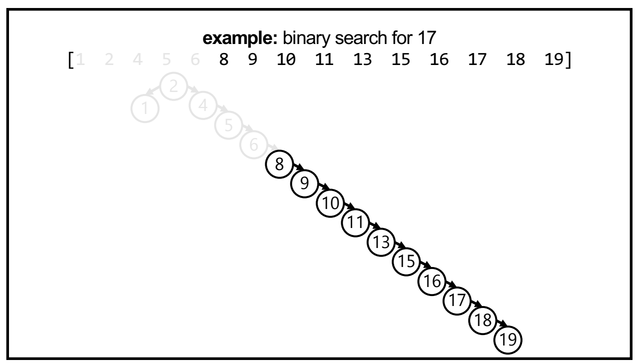
930



931



932



933

...

934

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

(otherwise things starts to look a lot like linear search on a linked list 😟)

935

adding a new node to a binary search tree

(the simplest version)

936

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

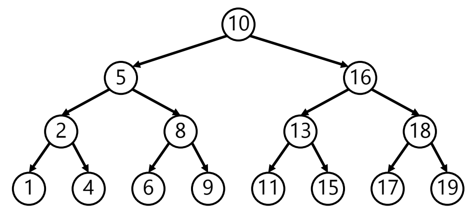
937

just keep going until you hit a `null` node

make it the new node 🙂 👍

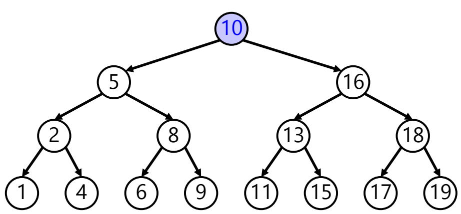
938

example: adding 7



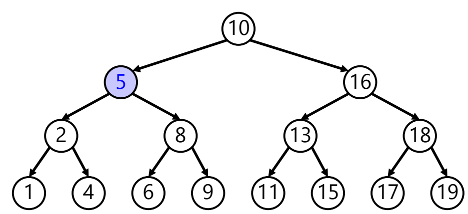
939

example: adding 7

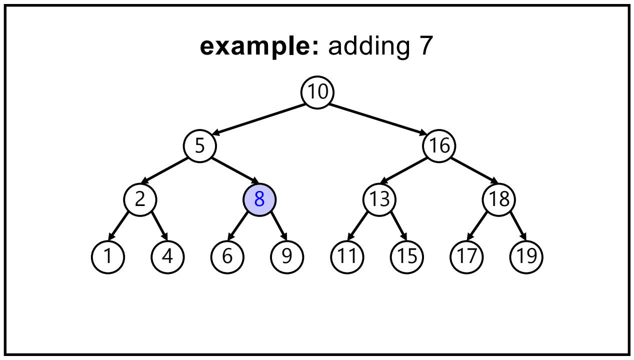


940

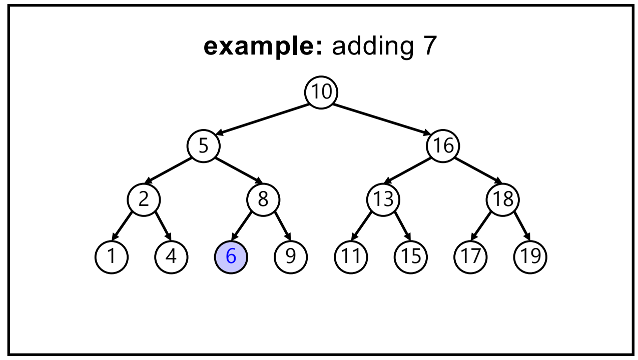
example: adding 7



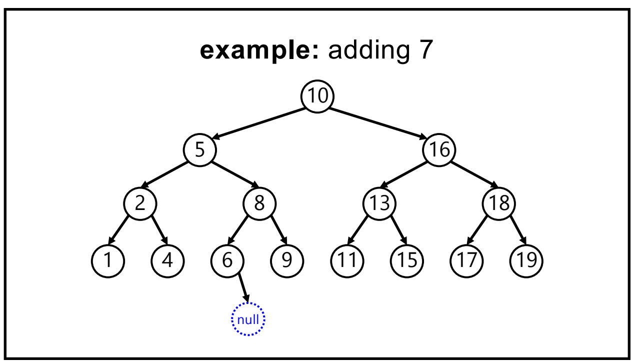
941



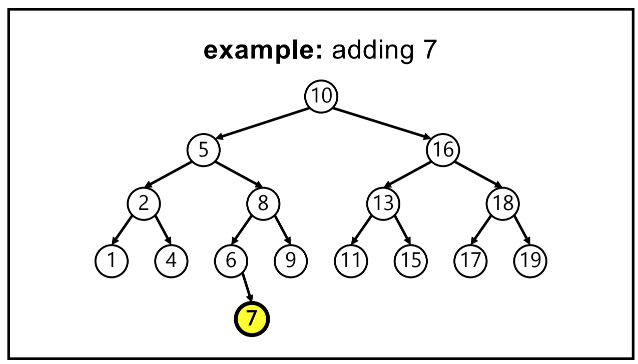
942



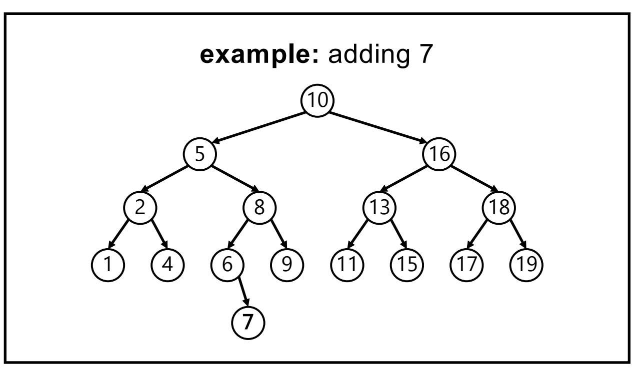
943



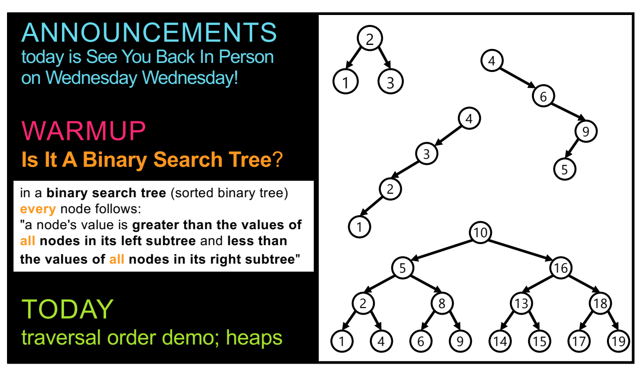
944



945



946



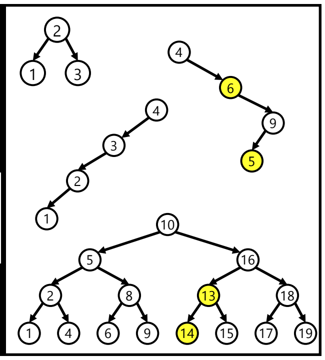
947

ANNOUNCEMENTS
today is See You Back In Person on Wednesday Wednesday!

WARMUP
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**"

TODAY
traversal order demo; heaps



948

record LEC-02

949

something to ponder:
what will this pseducode do?

```
binarySearchTree = BinarySearchTree()
array = [ 1, 2, 5, 6, 7, 9, 12, 17 ]
for element in array:
    binarySearchTree.add(element)
```

950

uses of binary search trees

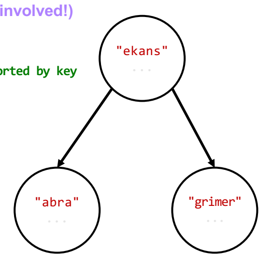
951

tree map (implement a map)

- implement a **map as a binary search tree**
  (NOT a hash map!—no hashing will be involved!)

```
class Node {
    String key; // the binary search tree is sorted by key
    Integer value;
    Node leftChild;
    Node rightChild;
}

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



952

bucket in a hash map

- use a binary search tree **as a bucket in a hash map** (with separate chaining)

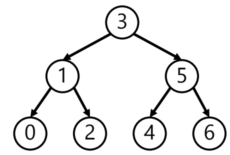
953

binary tree depth-first traversal orders

954

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



955

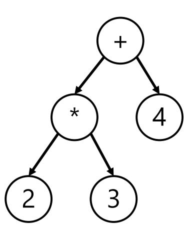
TODO (Jim): Live-code traversal orders
(Feel free to follow along or race.)

956

```
void _recurse(Node self) {
    // rearranging these 3 lines gives you all
    // 3! = 6 traversal orders
    System.out.print(self.value + " ");
    if (self.right != null) { _recurse(self.right); }
    if (self.left != null) { _recurse(self.left); }
}
```

957

2 * 3 + 4



who's that traversal order?

in-order!

958

heaps

959

always-complete max binary heap

960

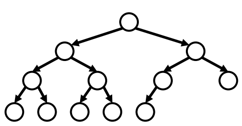
always-complete max binary heap

- 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"**

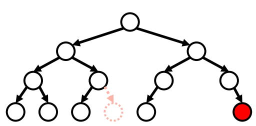
961

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)

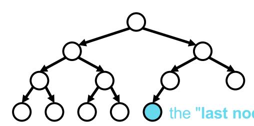


balanced but **NOT complete**

962

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"



the **"last node"**

complete
(and balanced)

963

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

964

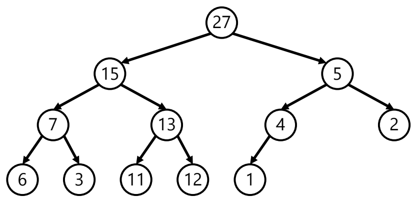
always-complete max binary heap

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

965

always-complete max binary heap

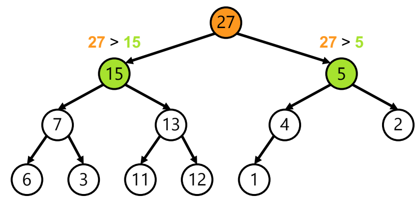
- in a **max binary heap**, **every node** follows: "a node's value is greater than the value of its left child and the value of its right child"



966

always-complete max binary heap

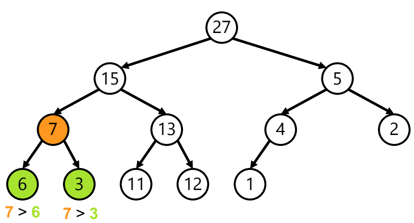
- in a **max binary heap**, **every node** follows: "a node's value is greater than the value of its left child and the value of its right child"



967

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"



968

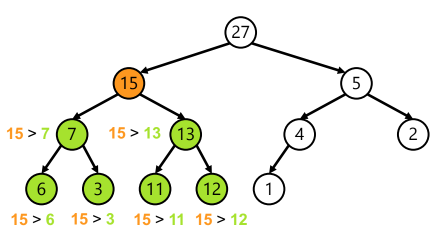
always-complete max binary heap

- 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

969

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"



970

heap interface

add(...) & remove()

🐡 ⚓

971

heap interface

- `// Add this value to the heap.`
  `void add(ValueType value) { ... }`

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

972

`add(...)`

973

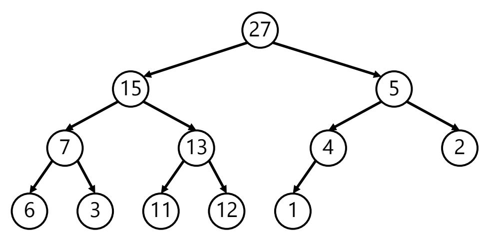
`void add(ValueType value);`

- 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"?

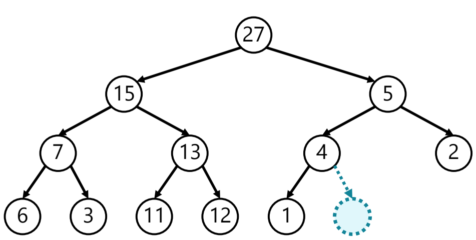
974

example: adding 10



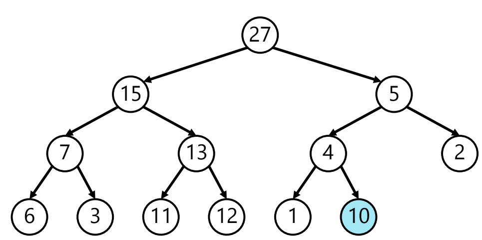
975

example: adding 10

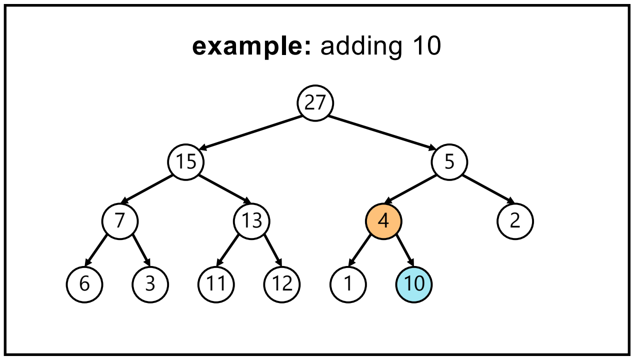


976

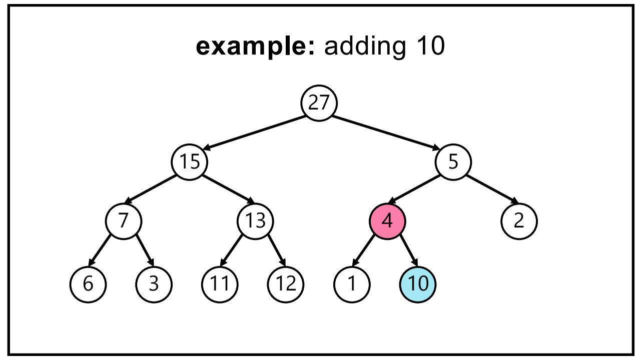
example: adding 10



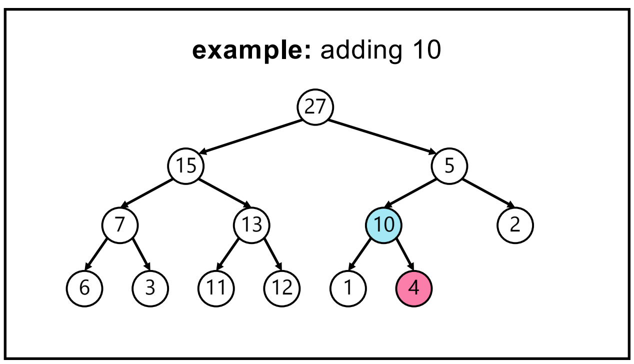
977



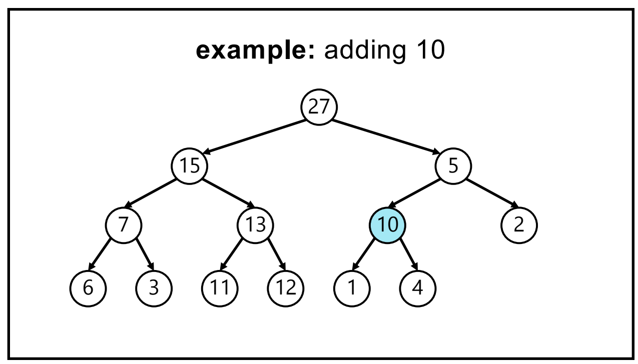
978



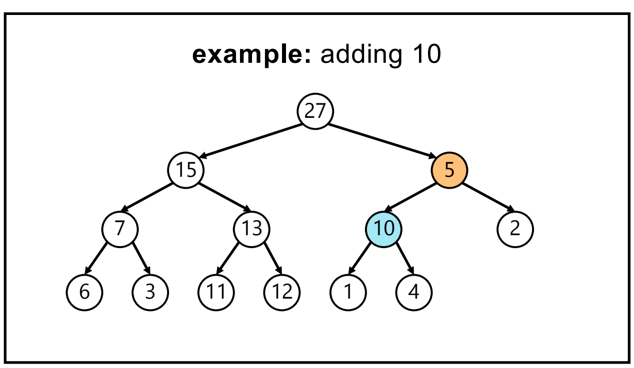
979



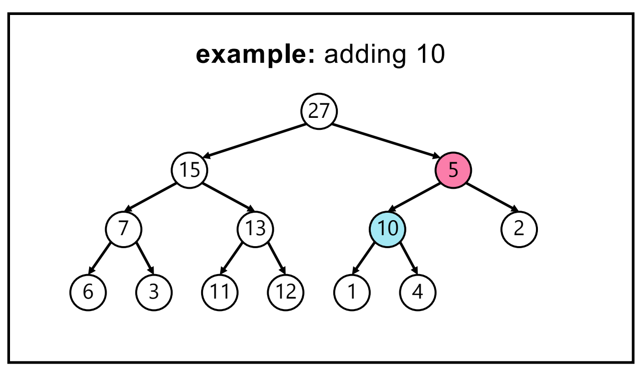
980



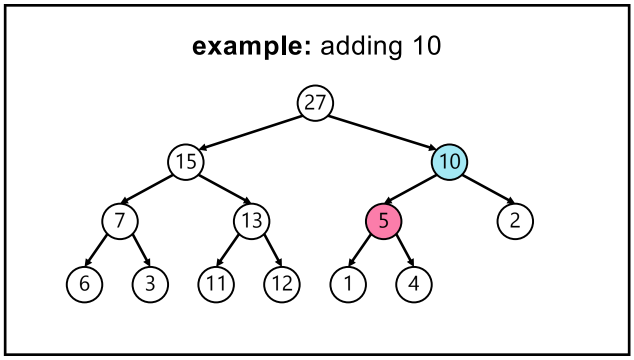
981



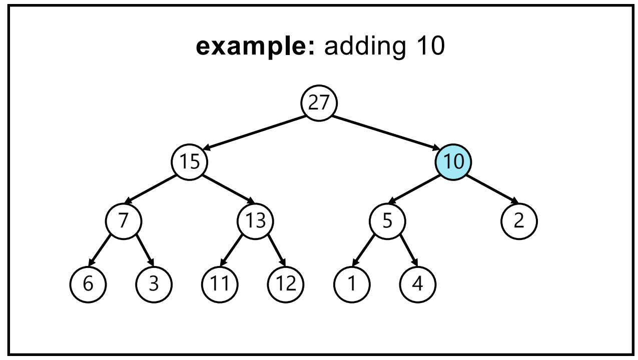
982



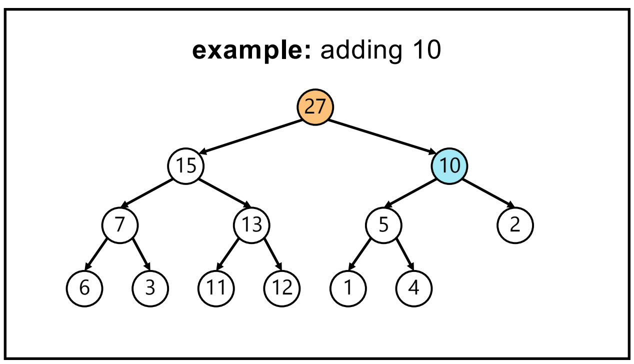
983



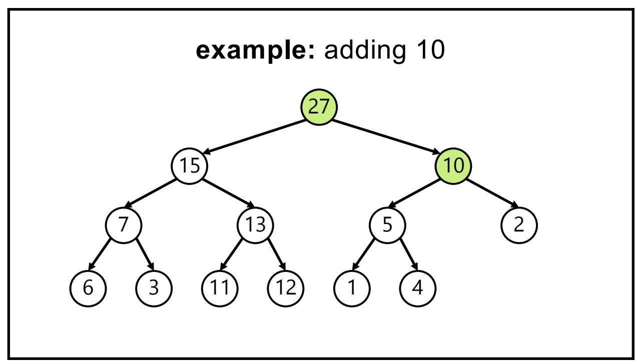
984



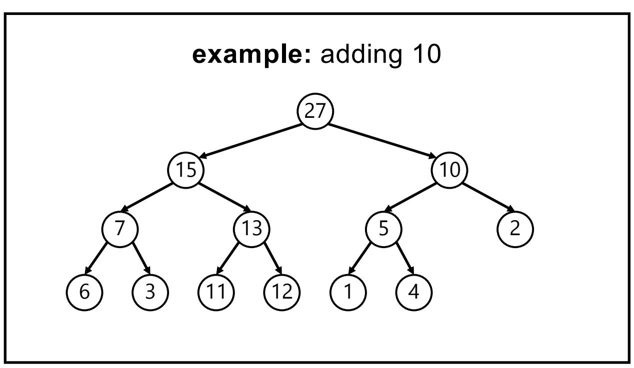
985



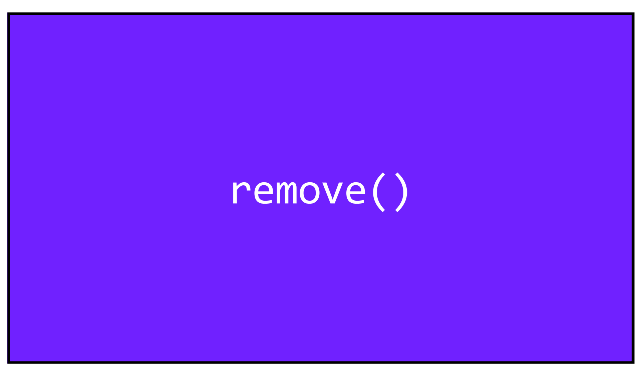
986



987



988



989

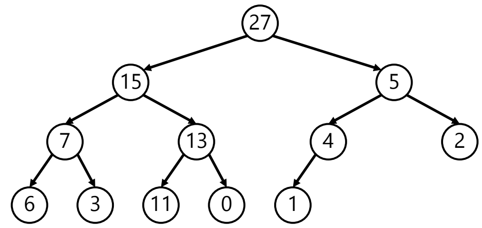
`ValueType remove();`

- 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 "dead" 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"?

990

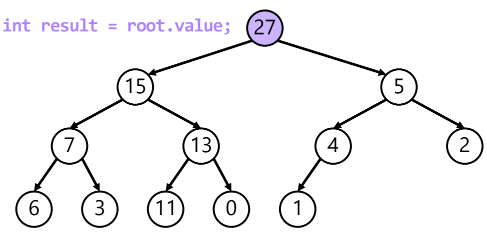
example: removing max node



991

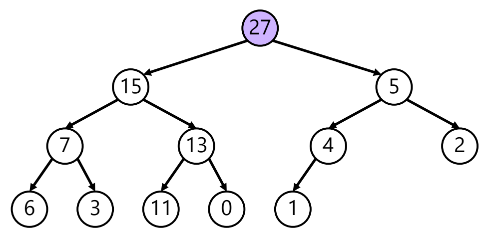
example: removing max node

`int result = root.value;`



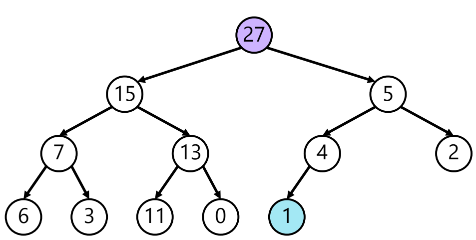
992

example: removing max node



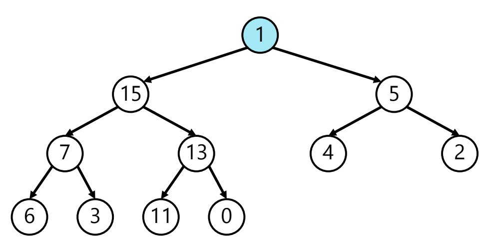
993

example: removing max node

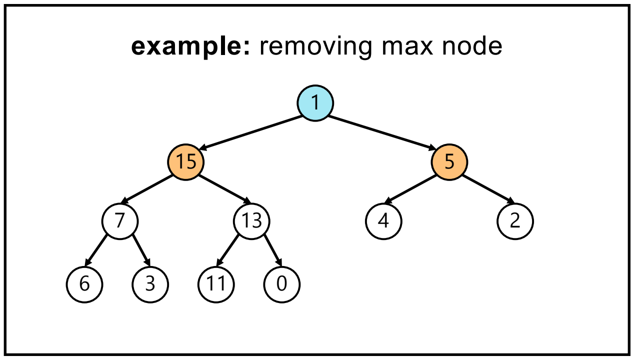


994

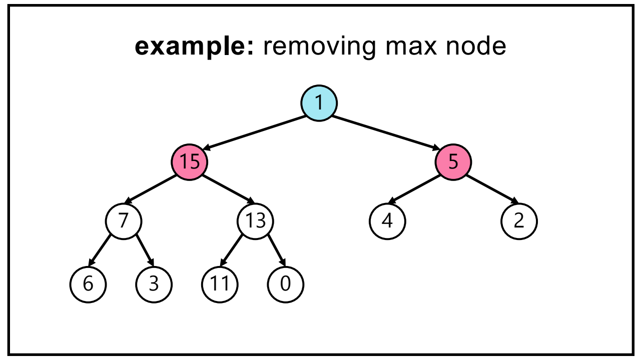
example: removing max node



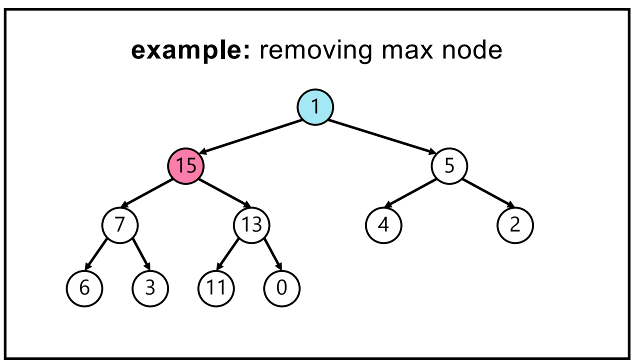
995



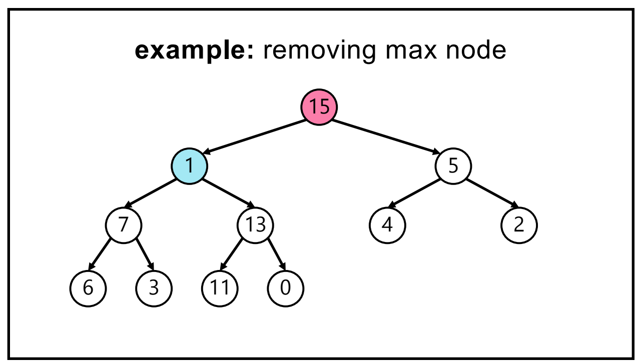
996



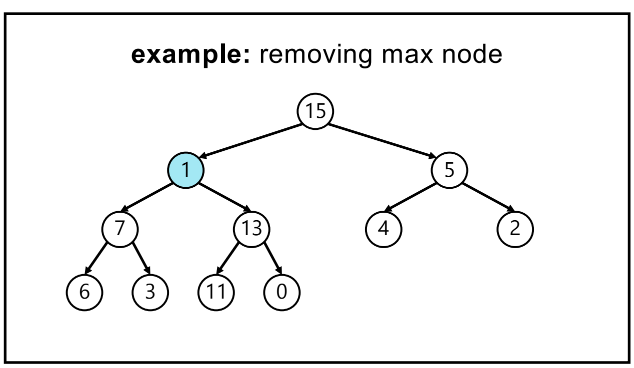
997



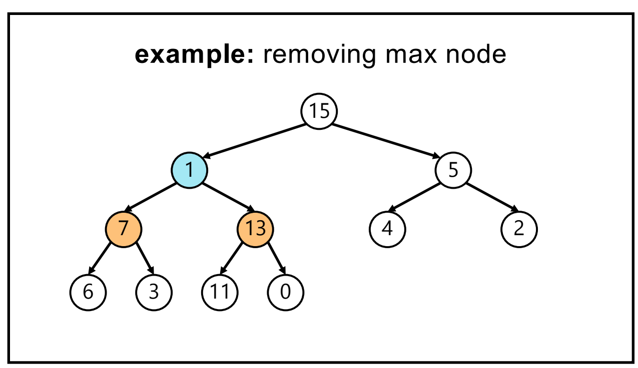
998



999

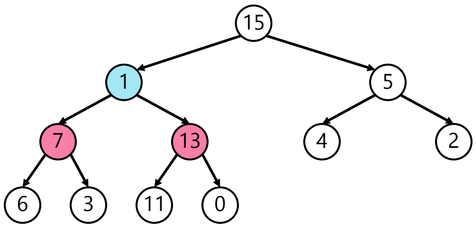


1000



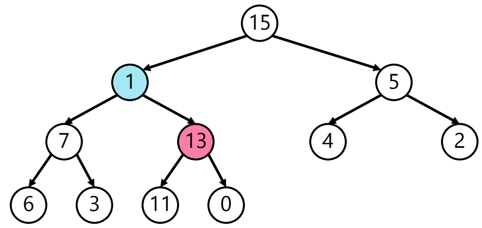
1001

example: removing max node



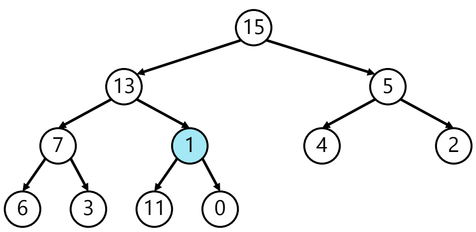
1002

example: removing max node



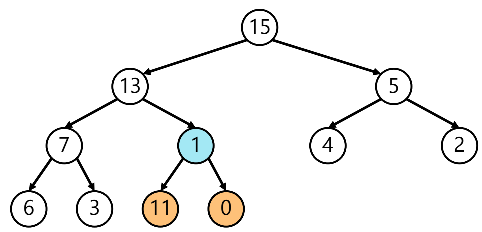
1003

example: removing max node



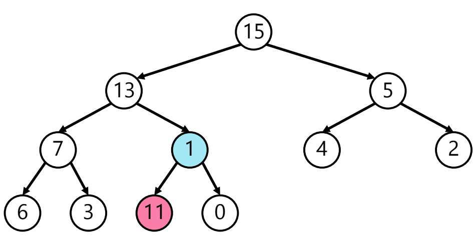
1004

example: removing max node



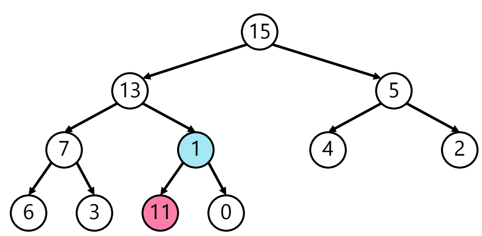
1005

example: removing max node

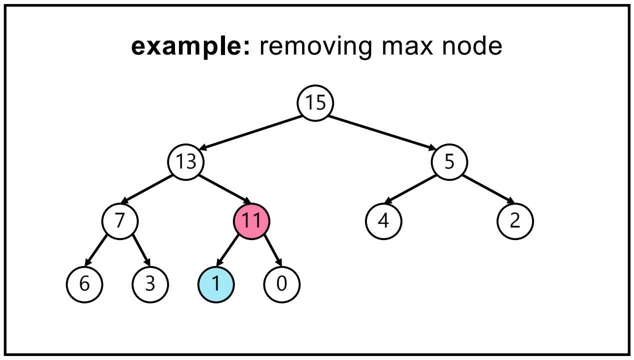


1006

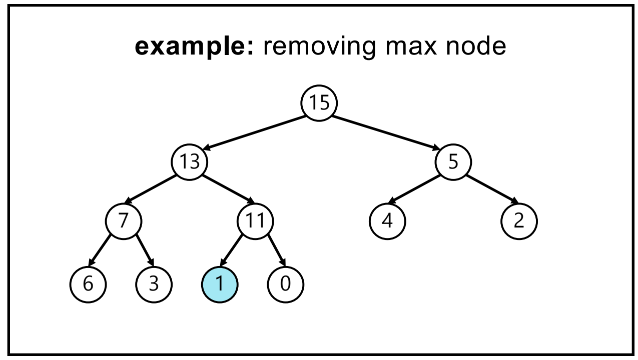
example: removing max node



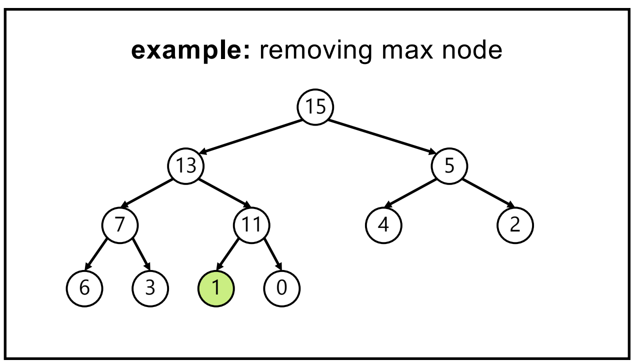
1007



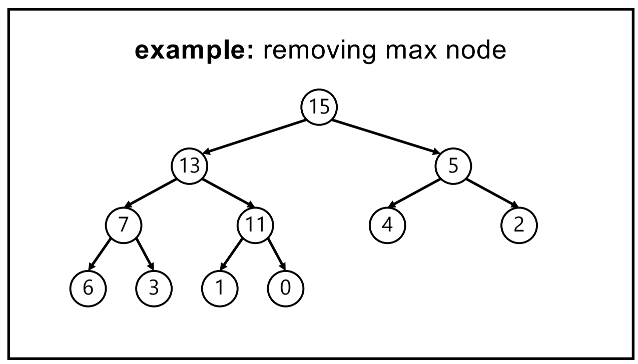
1008



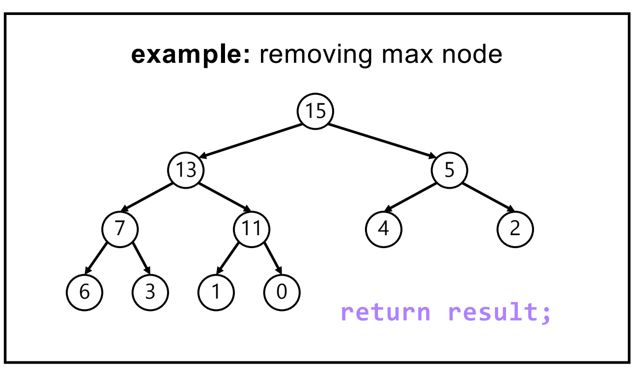
1009



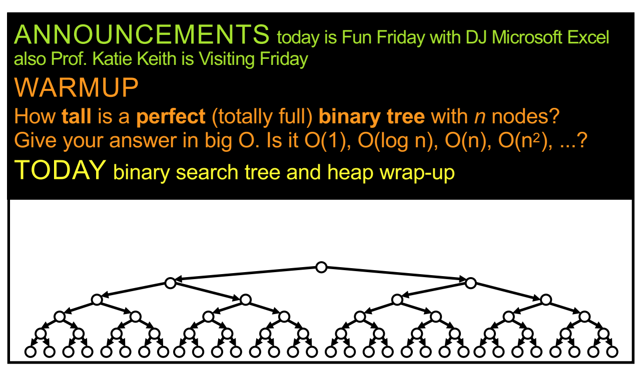
1010



1011



1012



1013

record LEC-02

1014

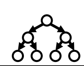
1 node → height 0

1015

3 nodes → height 1

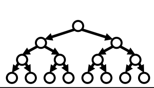
1016

7 nodes → height 2



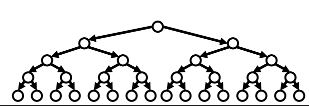
1017

15 nodes → height 3



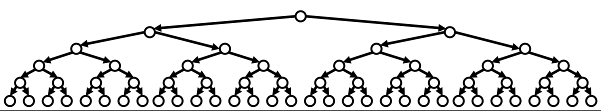
1018

31 nodes → height 4



1019

63 nodes → height 5



1020

summary

- (1 node, height 0)
- (3 nodes, height 1)
- (7 nodes, height 2)
- (15 nodes, height 3)
- (31 nodes, height 4)
- (63 nodes, height 5)
- ...

1021

summary

(1, 0)
(3, 1)
(7, 2)
(15, 3)
(31, 4)
(63, 5)
...

1022

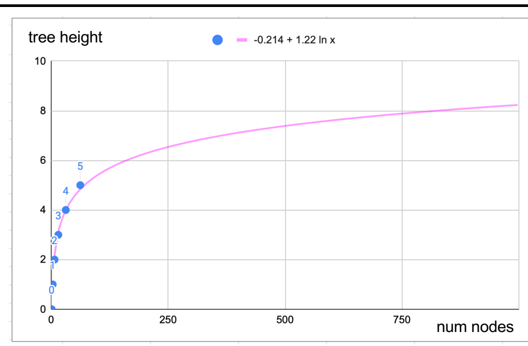
summary

1	0
3	1
7	2
15	3
31	4
63	5

1023

TODO (Jim): Let's make a plot.

1024



1025

log n

1026

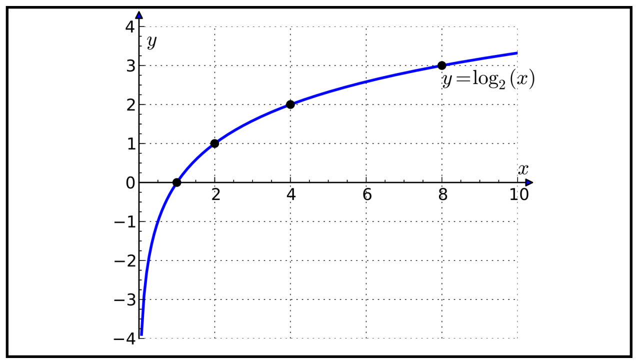
a **balanced binary tree**
has O(log n) height

note: the example we just did showed this for a "perfectly balanced" binary tree, but it is also true for just plain ol' balanced binary search trees

1027

what does log look like?

1029



1030

log is the inverse of exponential growth

1031

$y = 2^x$

solve for x.

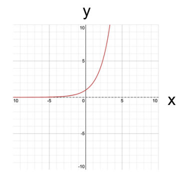
1032

$y = 2^x$

$x = \log_2 y$

1033

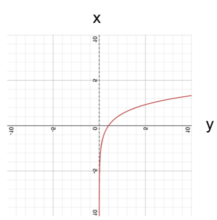
$y = 2^x$



$x = \log_2 y$

1034

$y = 2^x$



$x = \log_2 y$

1035

the change of base formula

implies that $O(\log_2 n) = O(\log_{10} n) = \ldots$

1036

$\log_b n = \log_d n \,/\, \log_d b$

1037

$\log_2 n = \log_{10} n \,/\, \log_{10} 2$

1038

$\log_2 n = \log_{10} n \,/\, \log_{10} 2$

this is a constant.

1039

$\log_2 n = c \log_{10} n$

1040

$O(\log_2 n) = O(c \log_{10} n)$

1041

$O(\log_2 n) = O(\log_{10} n)$

1042

$O(\log_2 n)$ and $O(\log_{10} n)$
are the exact same thing

1043

so you can just say $O(\log n)$
and not worry about it 🙄 👍

1044

binary search tree details

1045

self-balancing
binary search trees

1046

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

(otherwise things starts to look a lot like linear search on a linked list 😟)

1047

1048

self-balancing binary search trees are very cool but painful to implement

1049

heap details

1050

heap application:
priority queue

1051

review: queue

`remove();`

1052

review: queue

`remove();`

1053

review: queue

`add();`

1054

review: queue

`add();`

1055

review: queue

`remove();`

1056

review: queue

1057

extension: priority queue

1058

extension: priority queue

1059

extension: priority queue

1060

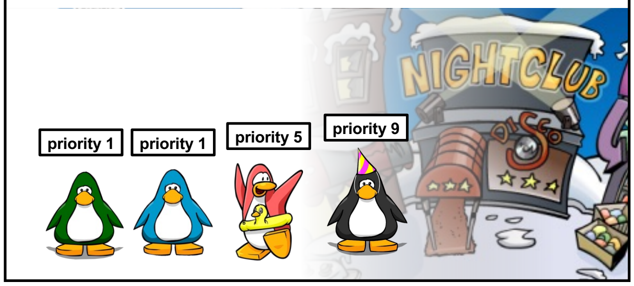
extension: priority queue

1061

extension: priority queue

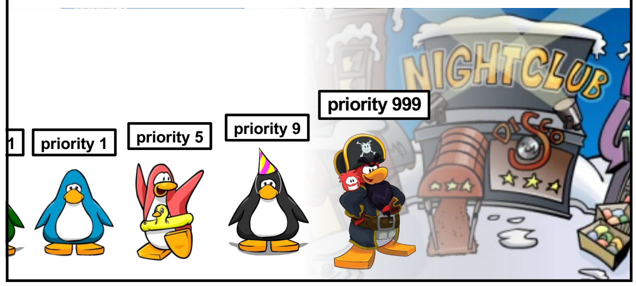
1062

extension: priority queue



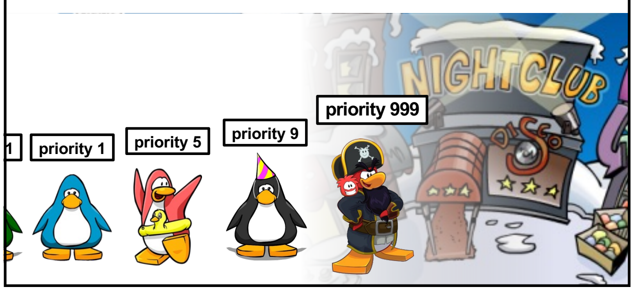
1063

extension: priority queue



1064

extension: priority queue



1065

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

1066

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

1067

TODO: club penguin meme

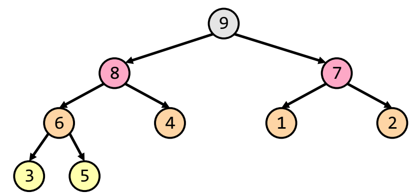
1068

heap application:
(implicit) heapsort

1069

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]



1070

this lets us do **in-place heapsort**!

(using only swaps)

1071

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

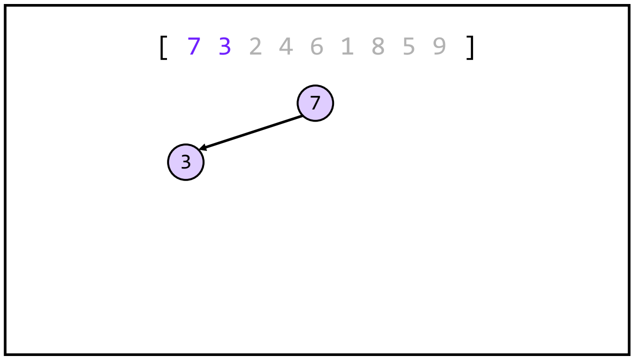
1072

[7 3 2 4 6 1 8 5 9]

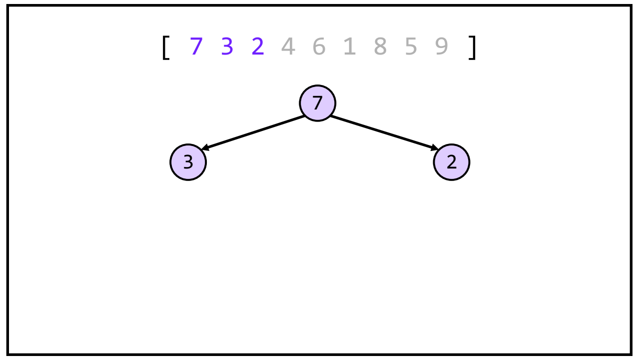
1073

[7 3 2 4 6 1 8 5 9]

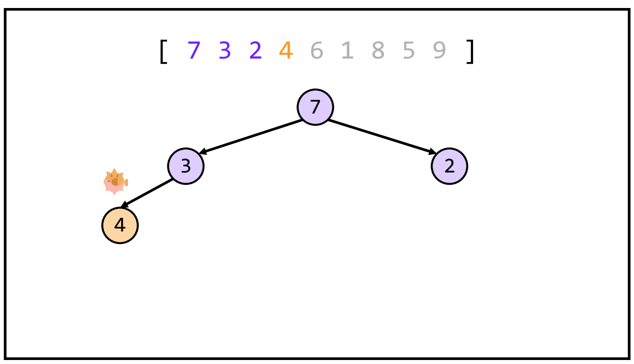
1074



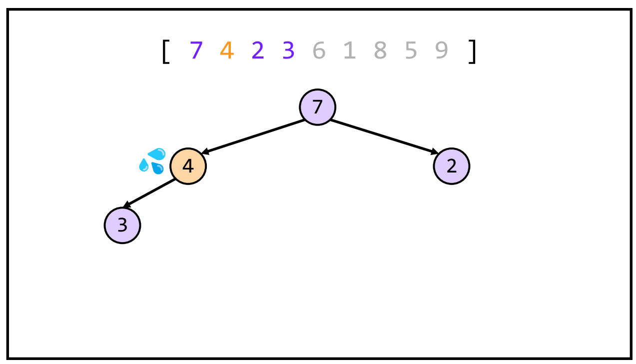
1075



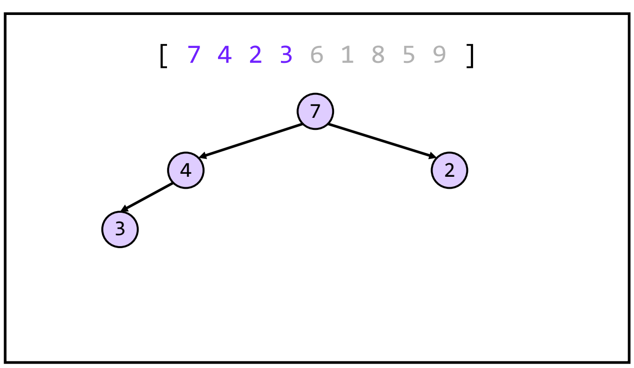
1076



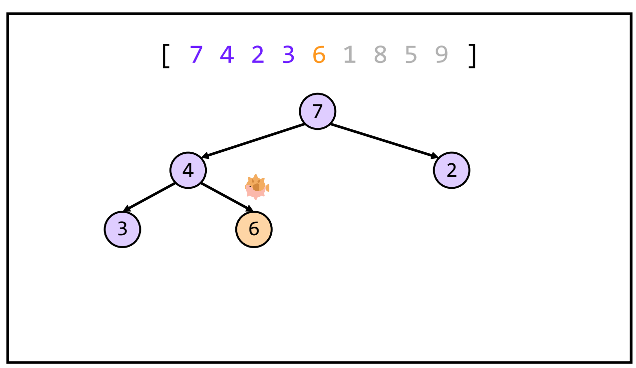
1077



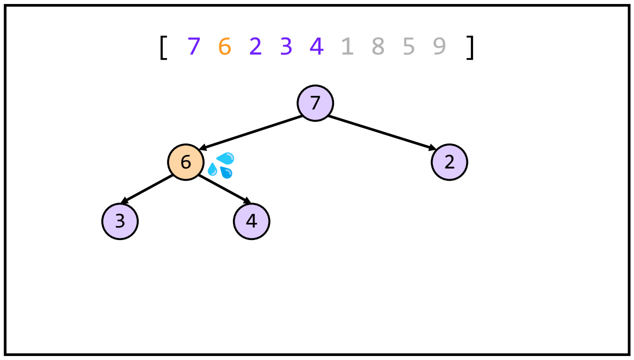
1078



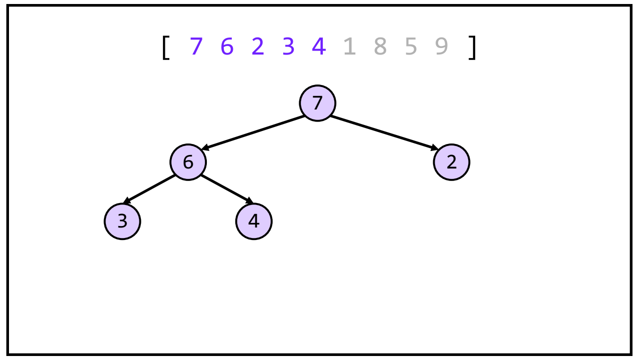
1079



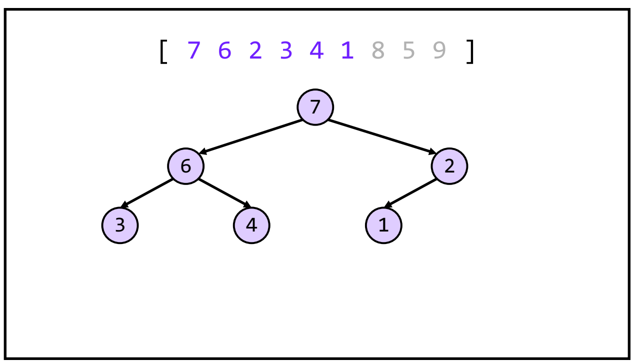
1080



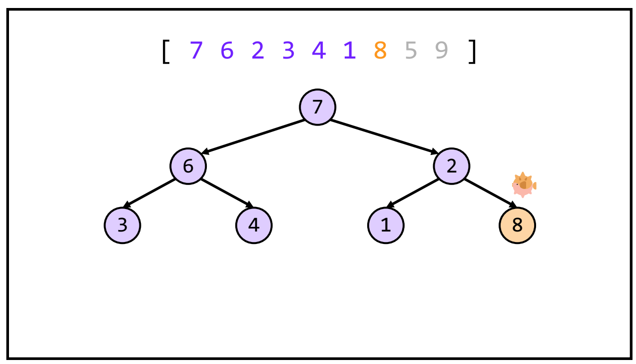
1081



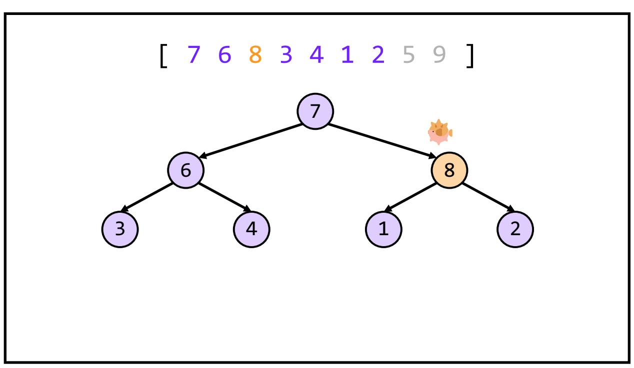
1082



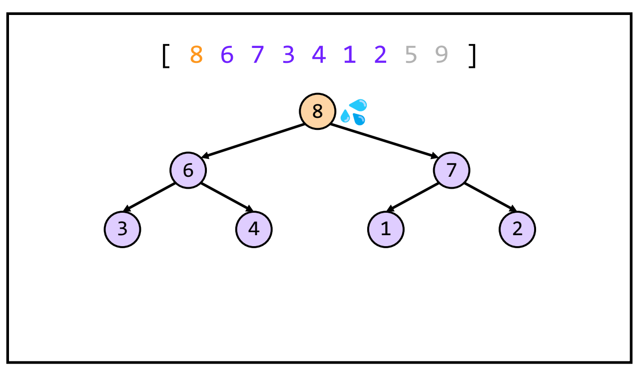
1083



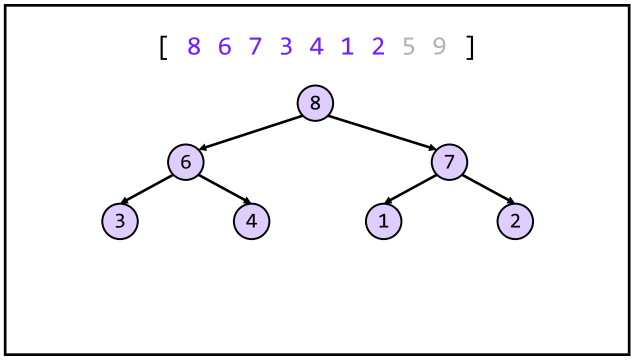
1084



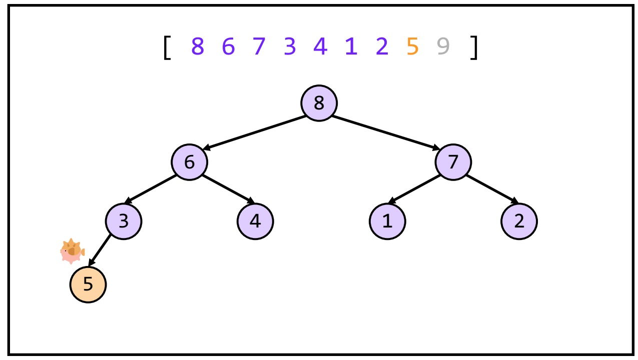
1085



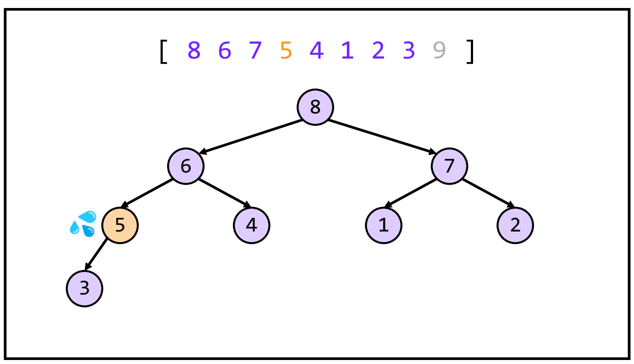
1086



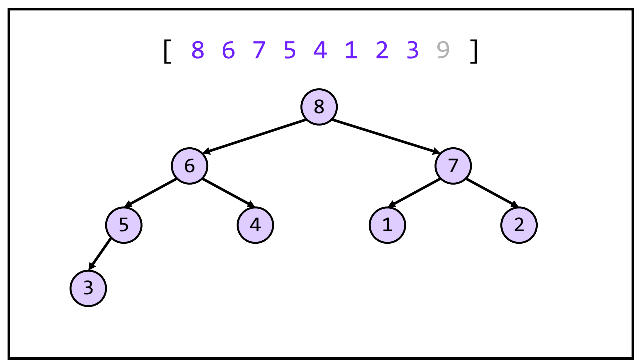
1087



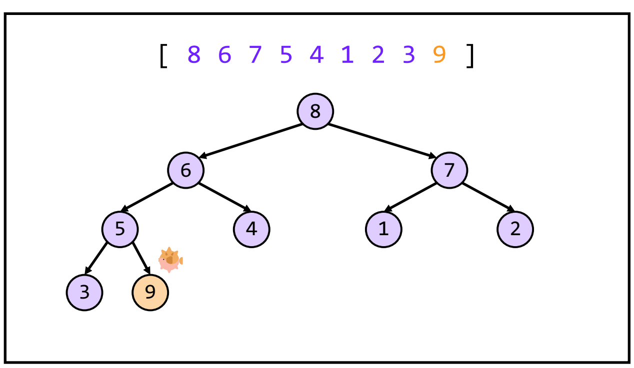
1088



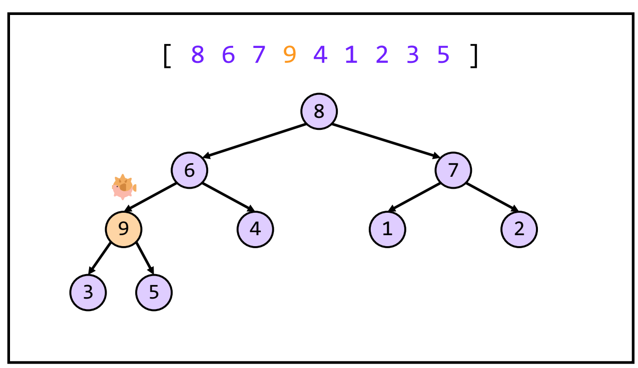
1089



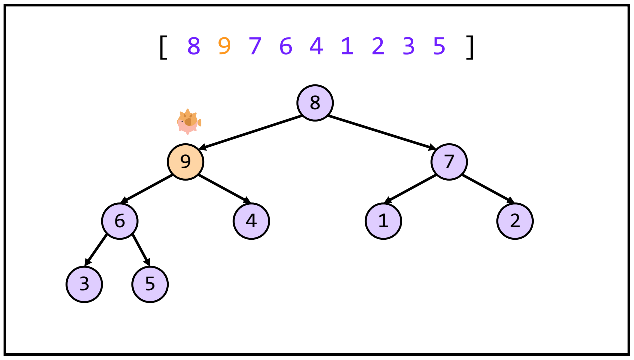
1090



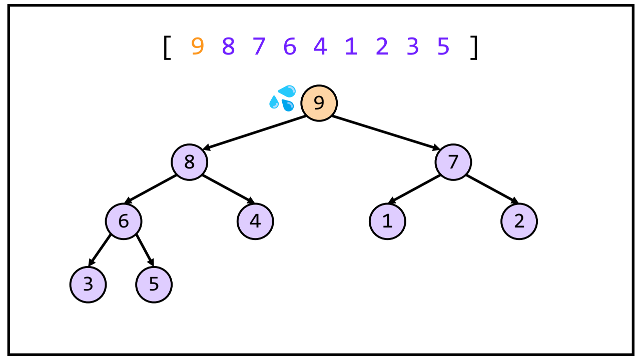
1091



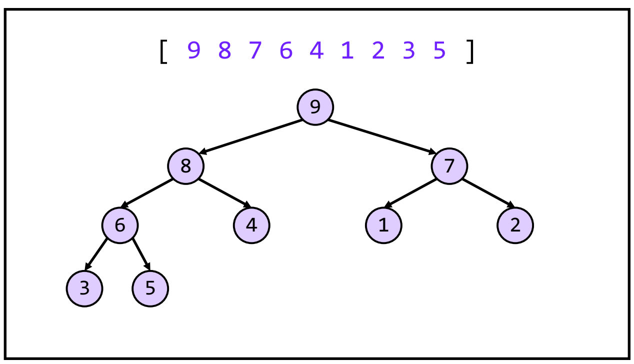
1092



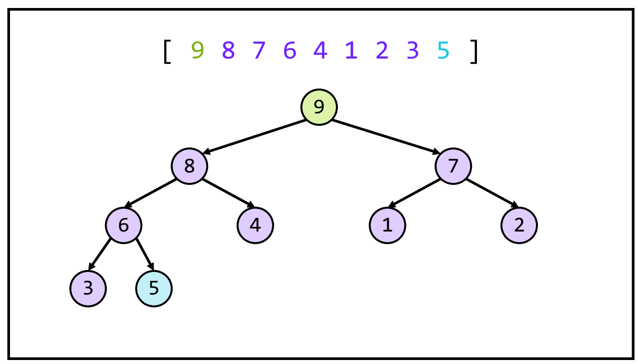
1093



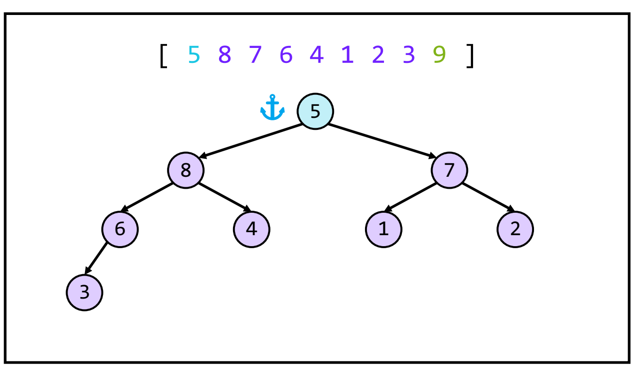
1094



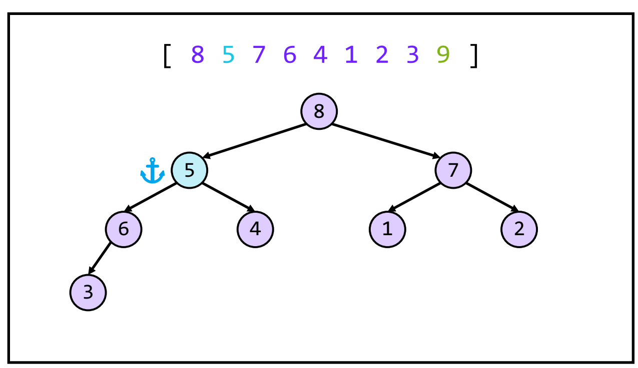
1095



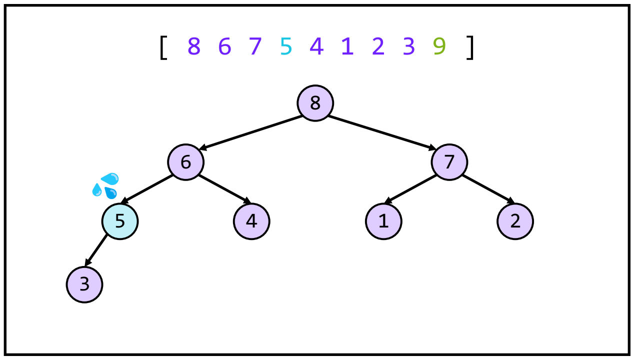
1096



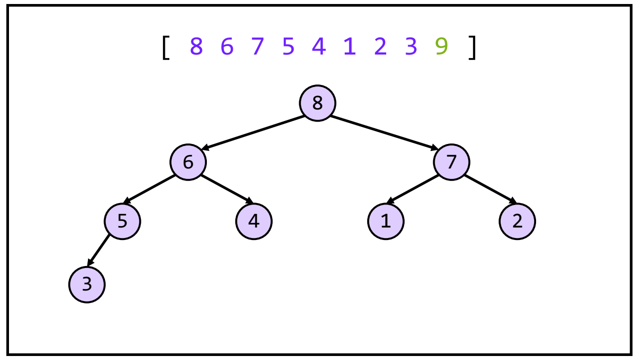
1097



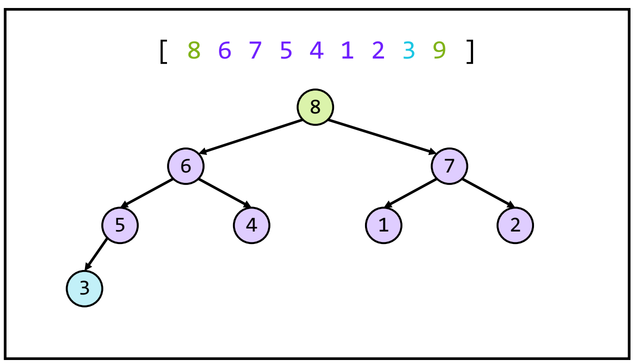
1098



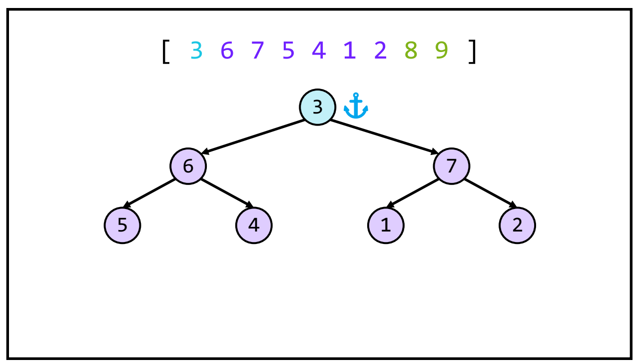
1099



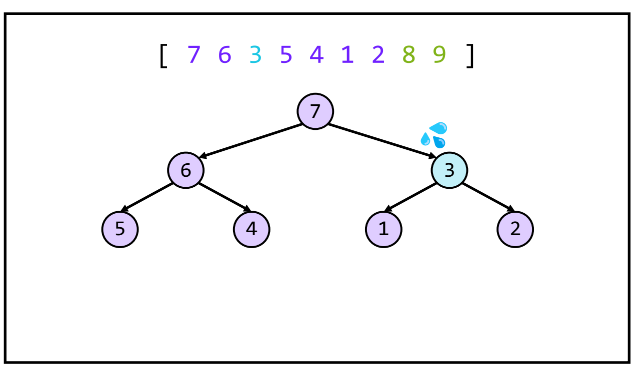
1100



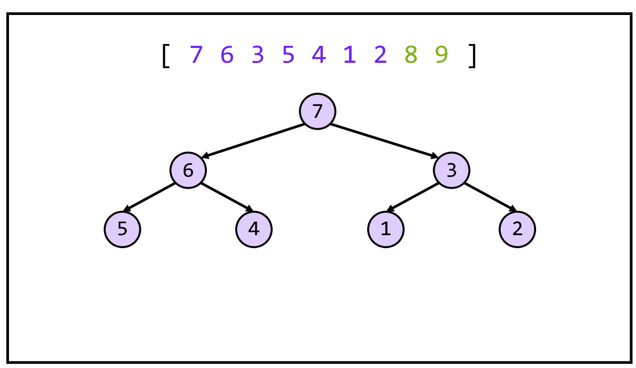
1101



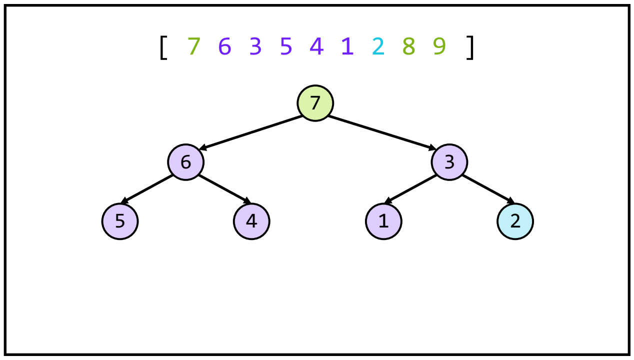
1102



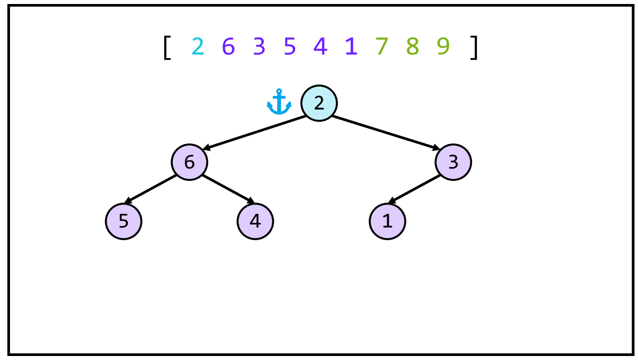
1103



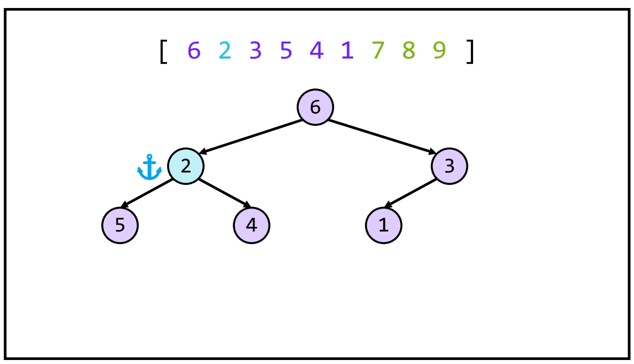
1104



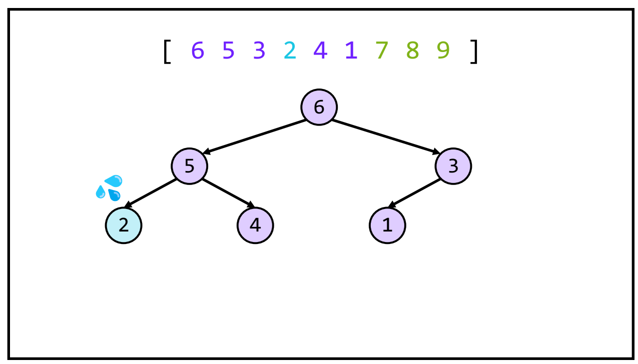
1105



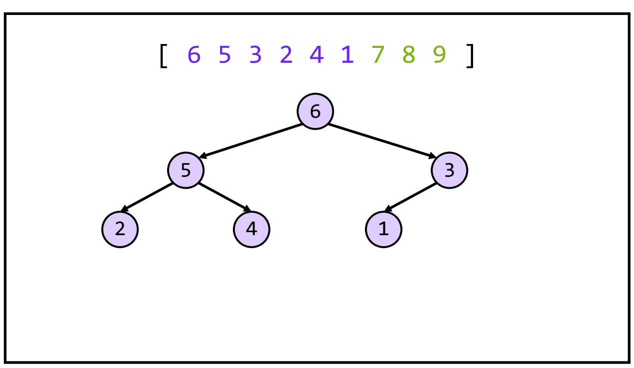
1106



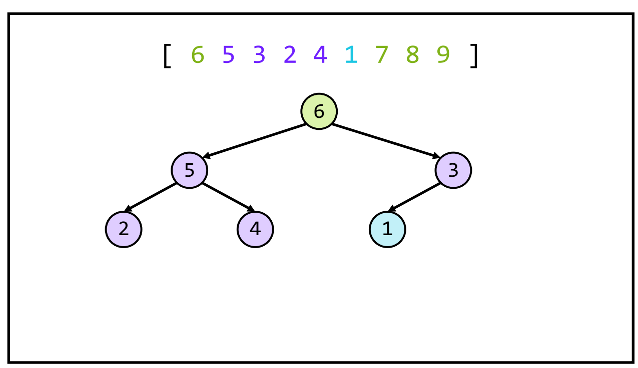
1107



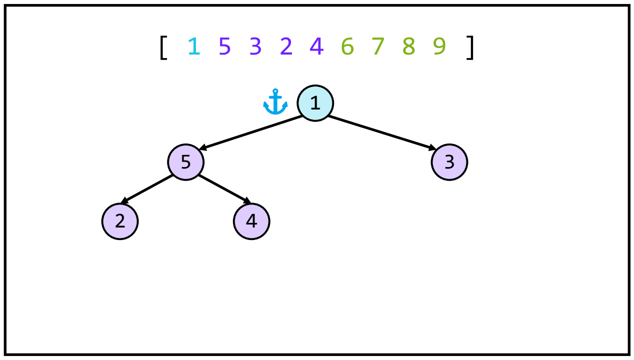
1108



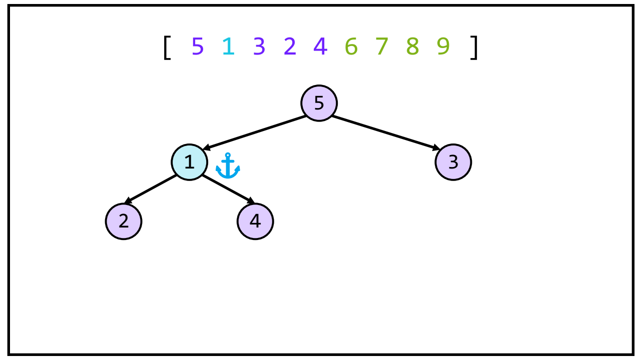
1109



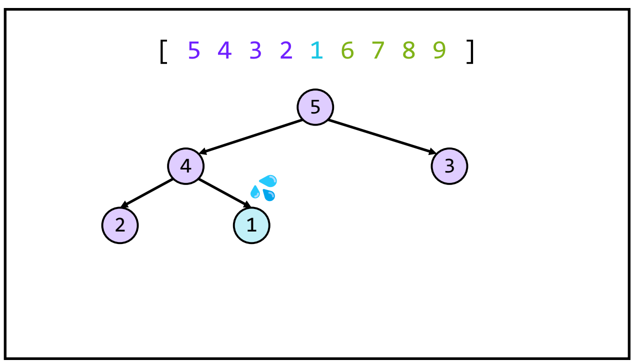
1110



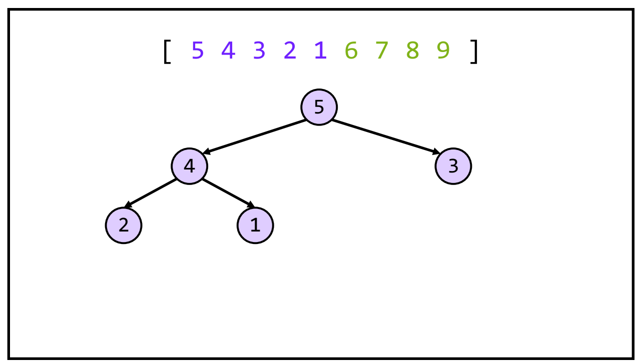
1111



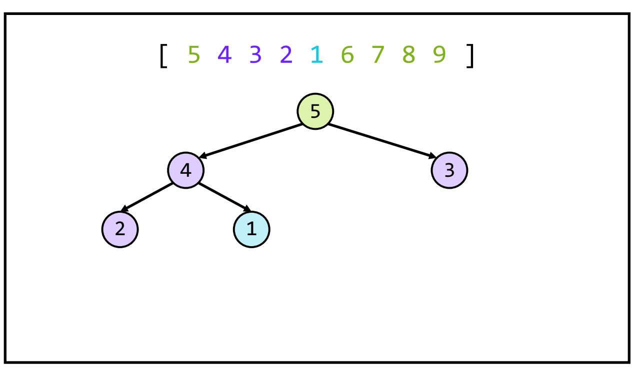
1112



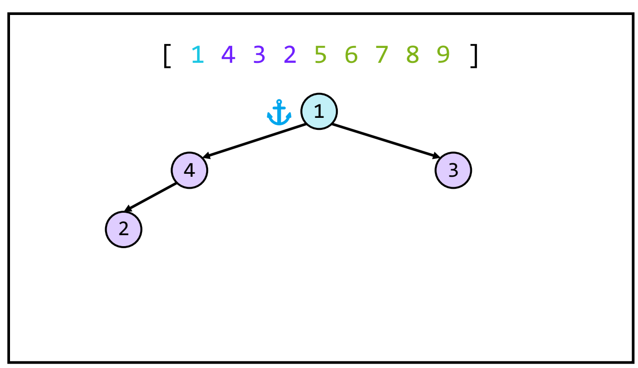
1113



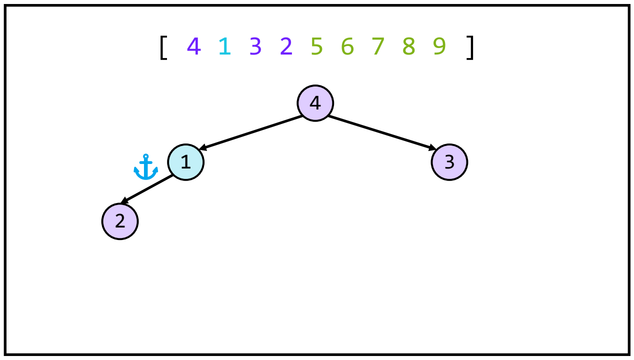
1114



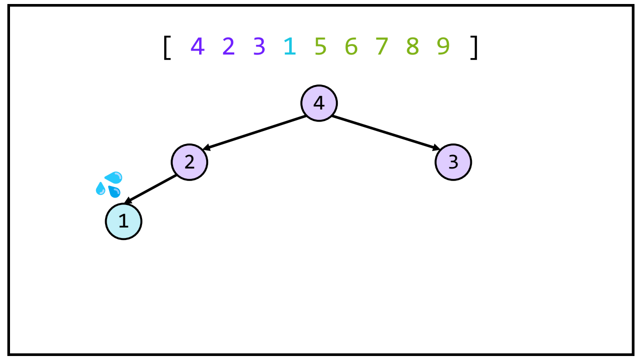
1115



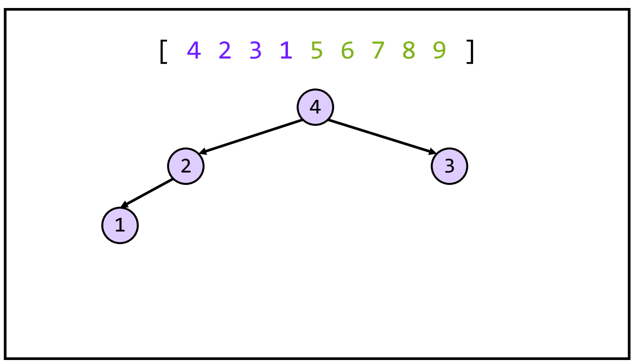
1116



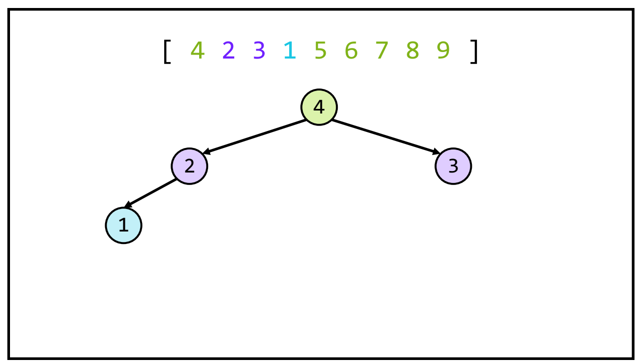
1117



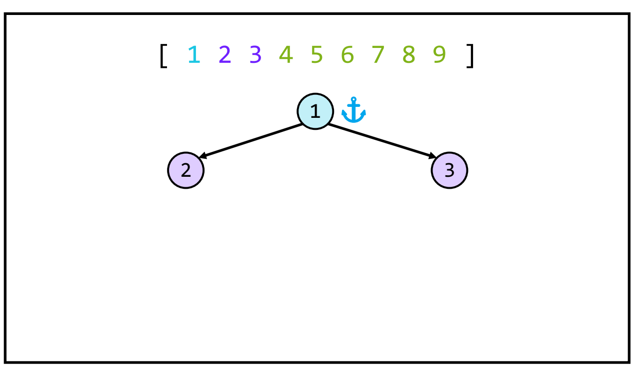
1118



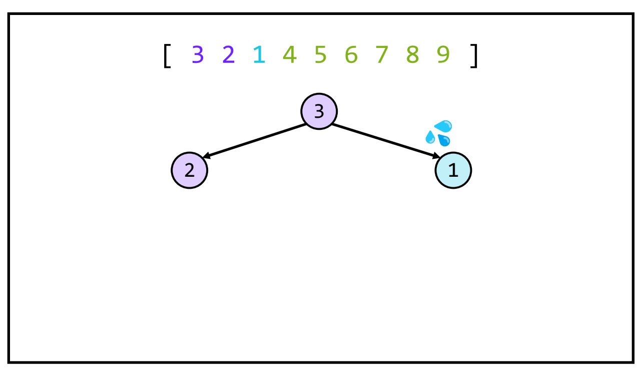
1119



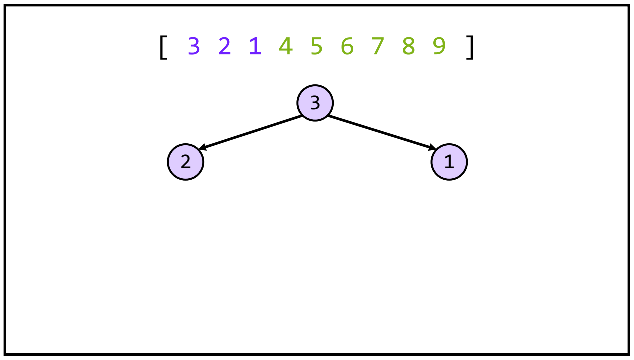
1120



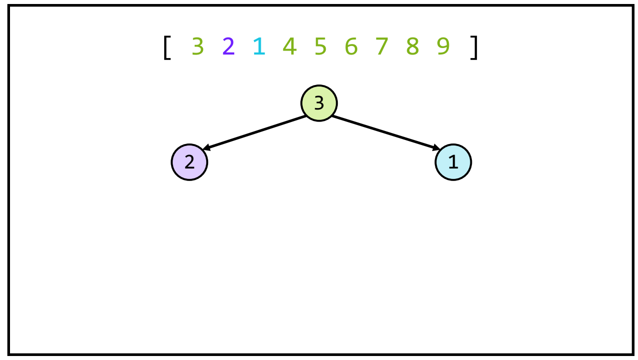
1121



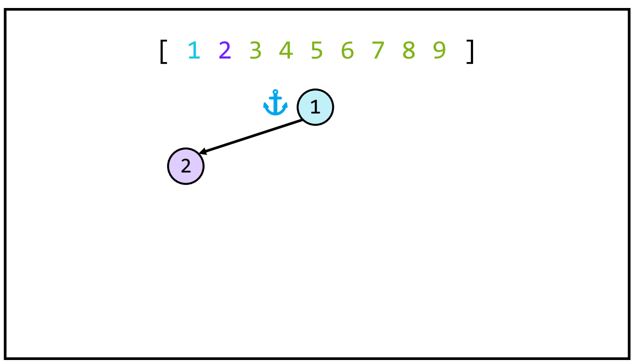
1122



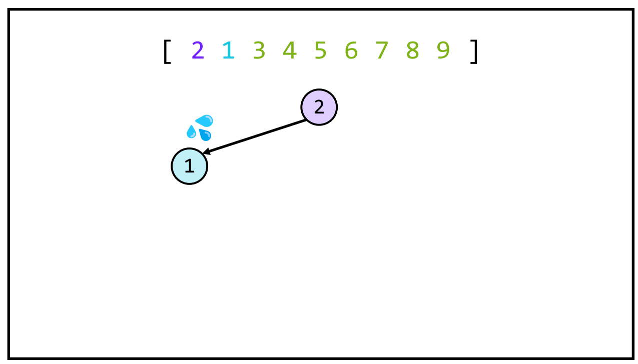
1123



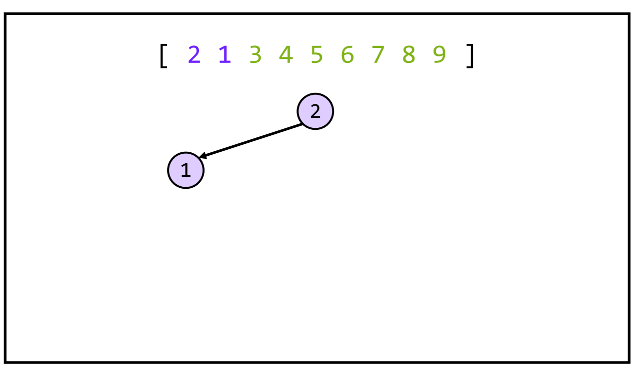
1124



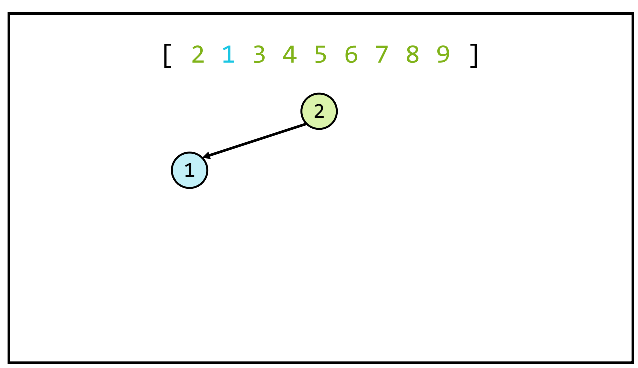
1125



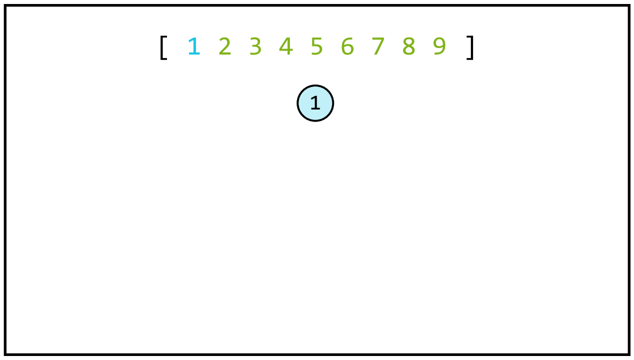
1126



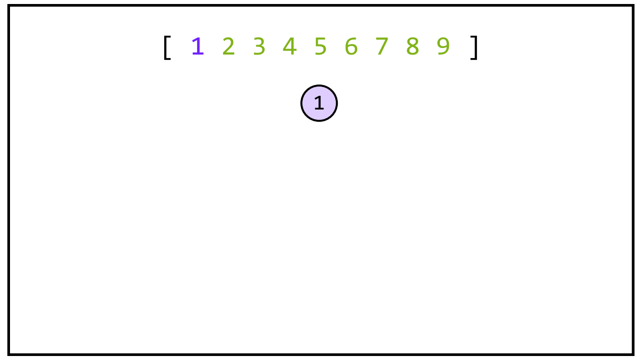
1127



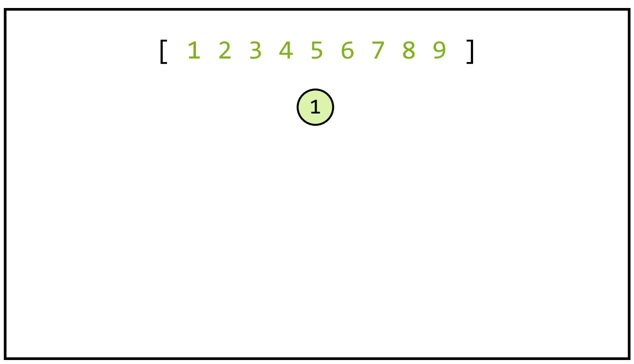
1128



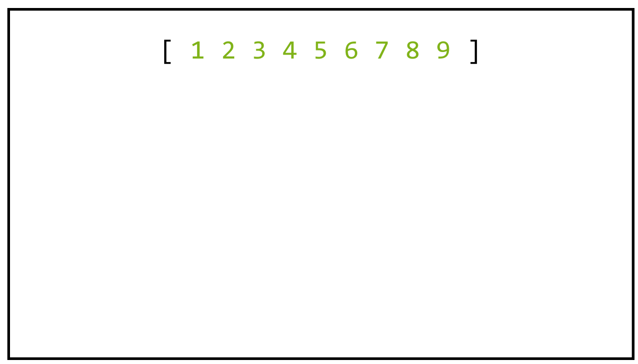
1129



1130



1131



1132

gamedev update

(switch to other laptop)

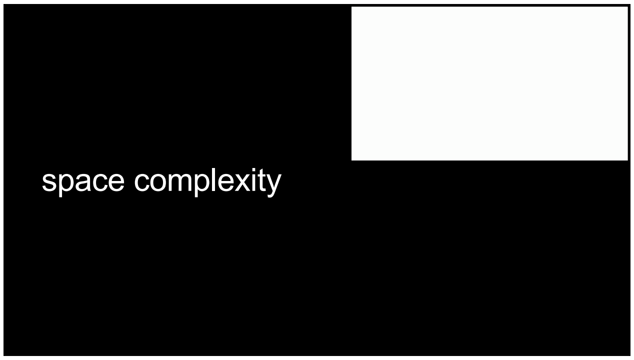
1133

avl tree
red black tree
anchor

n vs. log n
priority queue (club analogy)
log(n) time
implicit heap

robots/games

1134



1135

1136