Tutorial 04 - PQ ADT

CS2040S Semester 1 2023/2024

By Wu Biao, adapted from previous slides

Set real display name

https://pollev.com/rezwanarefin430

Binary Heap Operations

Q1 Priority Queue (PQ) ADT

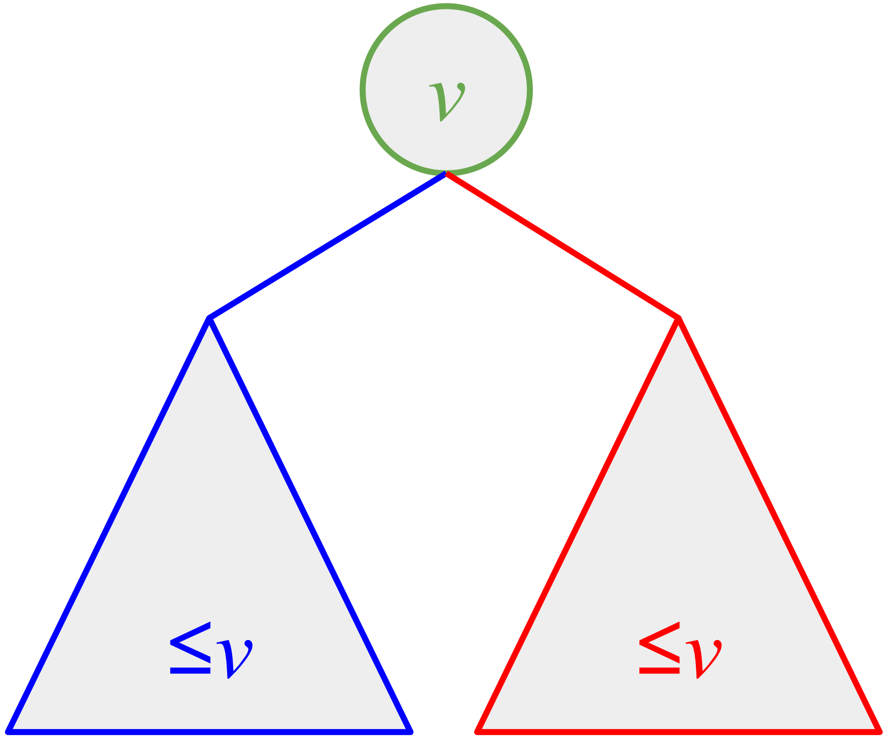
Common PQ ADT operations

`insert(v)`	Insert item with value `v` into the PQ.
`ExtractMax()`	Return the the item with highest priority from PQ.
`create(A)`	Create a PQ from the given array `A`.
`HeapSort(A)`	Return the sorted order of items in given array `A`.

Binary (Max) heap

- A *binary heap* is a standard implementation for PQ ADT.
- **It is a complete binary tree.**
- Therefore every vertex in a binary heap is itself also a binary heap (with the root being the vertex).
- *Max-heap property*: The value of every vertex in a binary max-heap is ≥ every value of its children. i.e. the root's value is the maximum of all values in the heap.
- *Min-heap property* is just the reverse of above using ≤.

A vertex in a binary max heap



Binary heap data-structure

Implement using an **array** in breadth-first order as implicit representation!

In the next slide, we will use

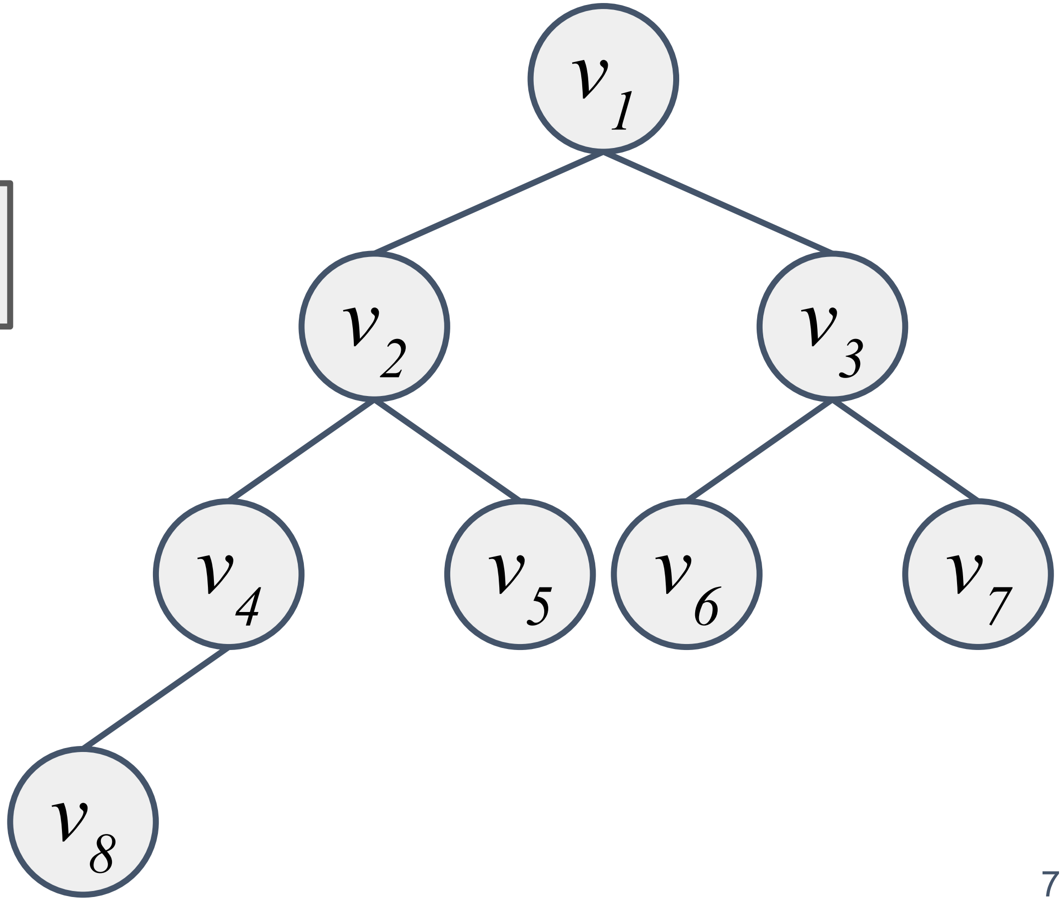
- $\mathbf{v_i}$ to represent value at index $\mathbf{i}$.
- $\mathbf{L_i}$ to represent left child index of index $\mathbf{i}$.
- $\mathbf{R_i}$ to represent right child index of index $\mathbf{i}$.
- $\mathbf{P_{i,j}}$ to represent parent index of index i and $\mathbf{j}$.

Note that for mathematical simplicity in illustration, we will use 1-based indexing.

Binary heap data-structure

We use the left compact array to implicitly represent the right heap!

Index	1	2	3	4	5	6	7	8
Value	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
		L_1	R_1	L_2	R_2	L_3	R_3	L_4
	$P_{2,3}$	$P_{4,5}$	$P_{6,7}$	P_8				



Array member relationship formulas

For array element at index i

1-based indexing

L_i : index $2i$
R_i : index $2i + 1$
P_i : index $\lfloor i / 2 \rfloor$

We demonstrated with 1-based indexing so that the math becomes clearer!

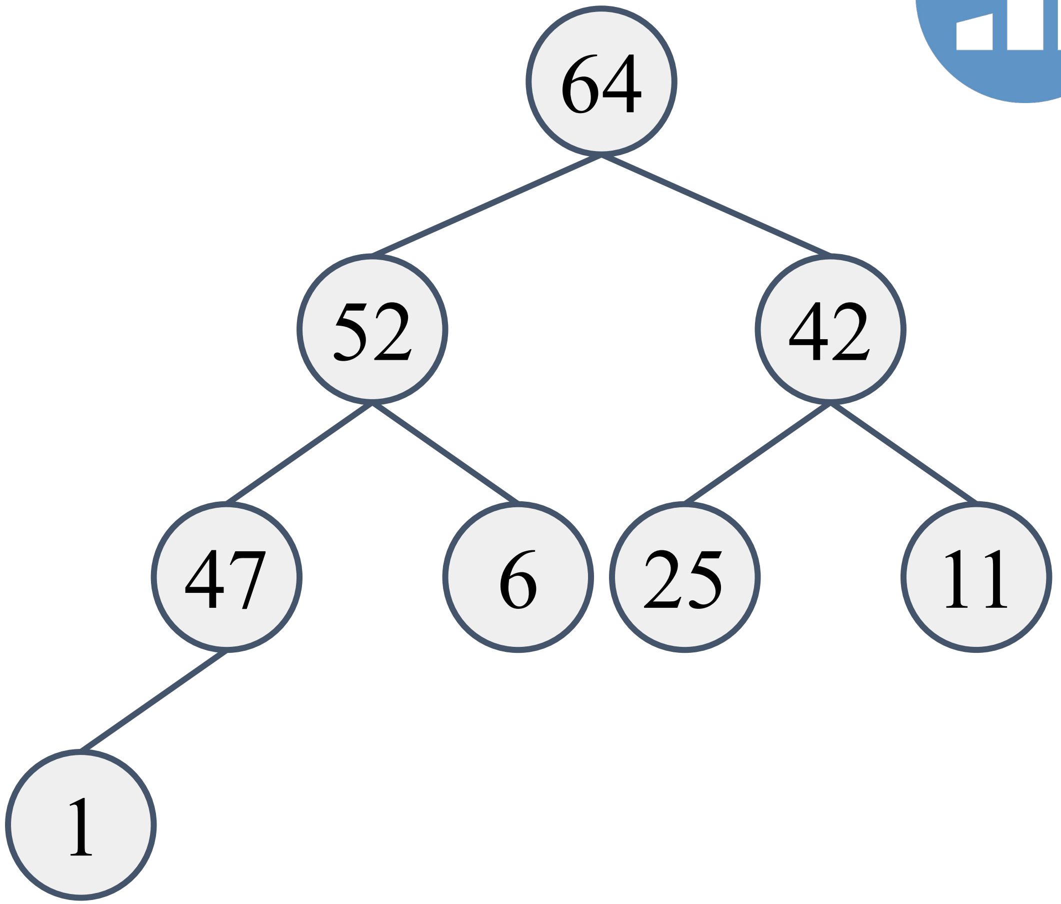
0-based indexing

L_i : index $2(i + 1) - 1$
R_i : index $2(i + 1)$
P_i : index $\lfloor (i + 1) / 2 \rfloor - 1$

Basically first convert to 1-based indexing by adding 1 to i then converting back to 0-based by deducting 1 from the output

Test yourself

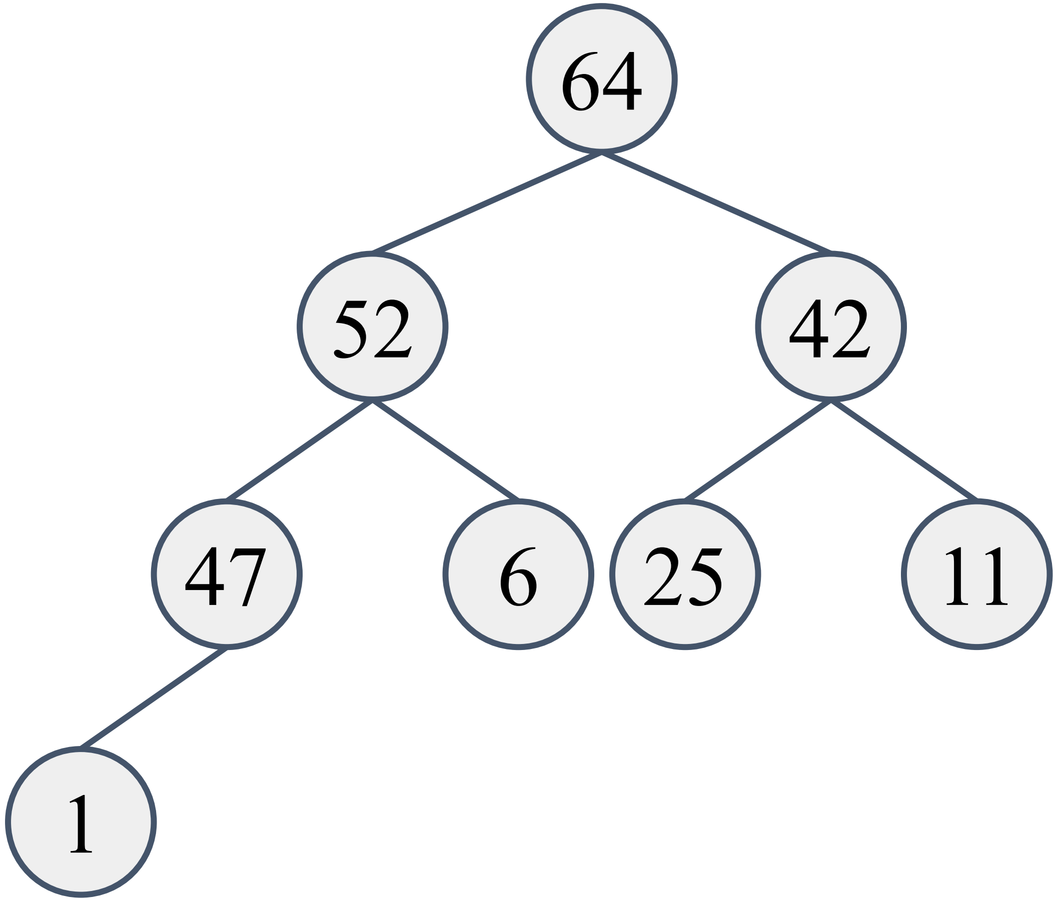
Is the value at the top of the max heap is ≥ all other values in the heap?



Test yourself

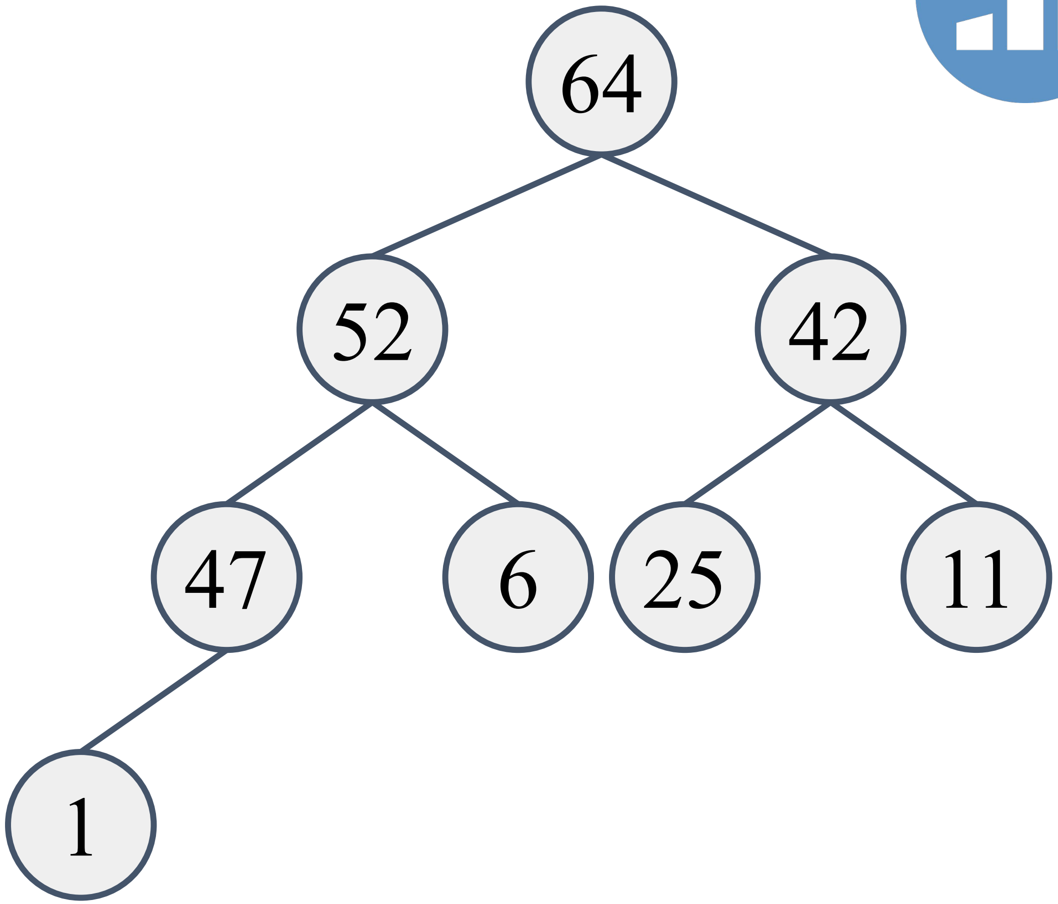
Is the value at the top of the max heap is ≥ all other values in the heap?

Yes! By definition!



Test yourself

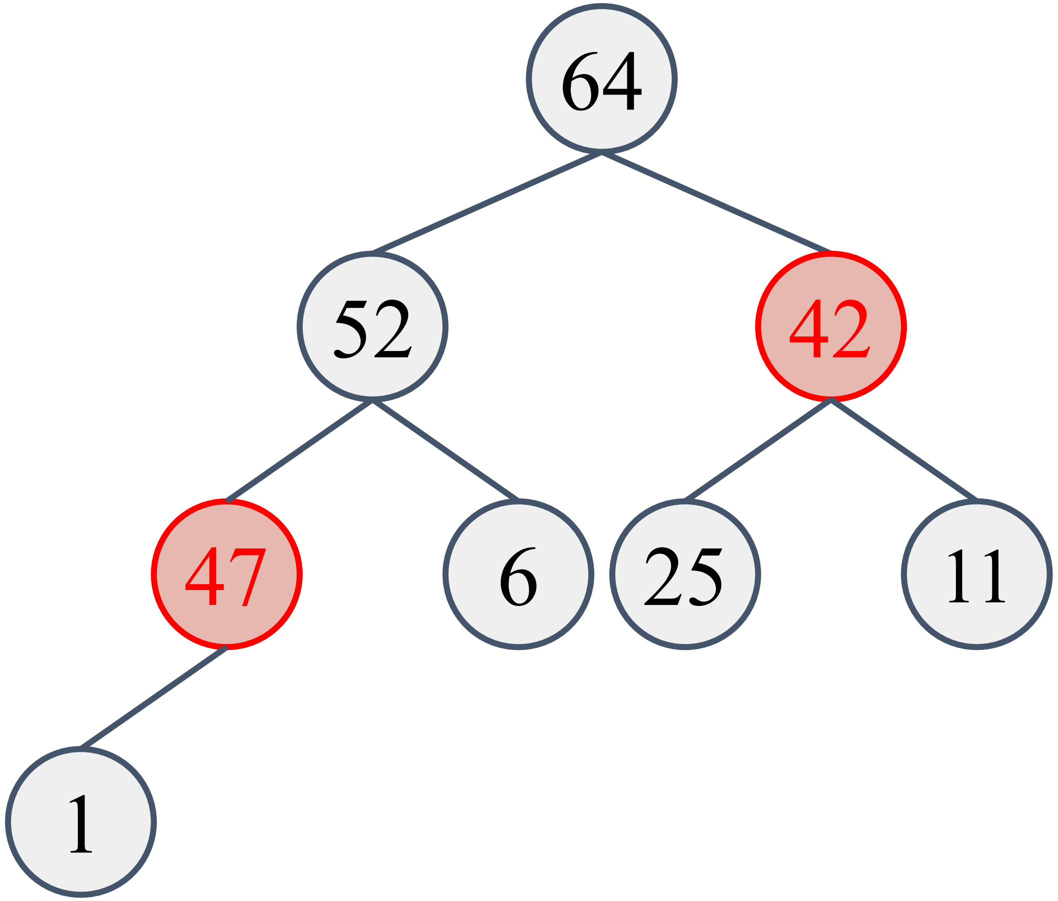
Must the value at a higher level be ≥ all values from lower levels?



Test yourself

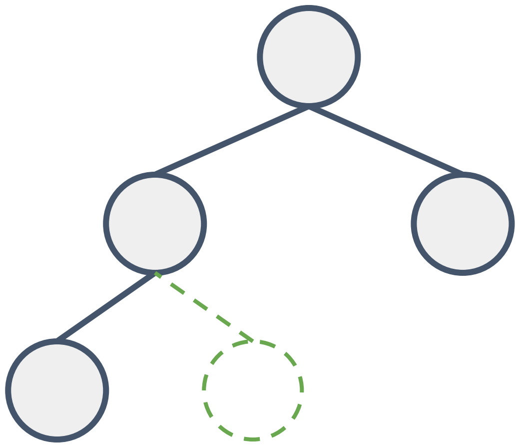
Must the value at a higher level be ≥ all values from lower levels?

Not necessarily!



`Insert(v)` (VisuAlgo slide)

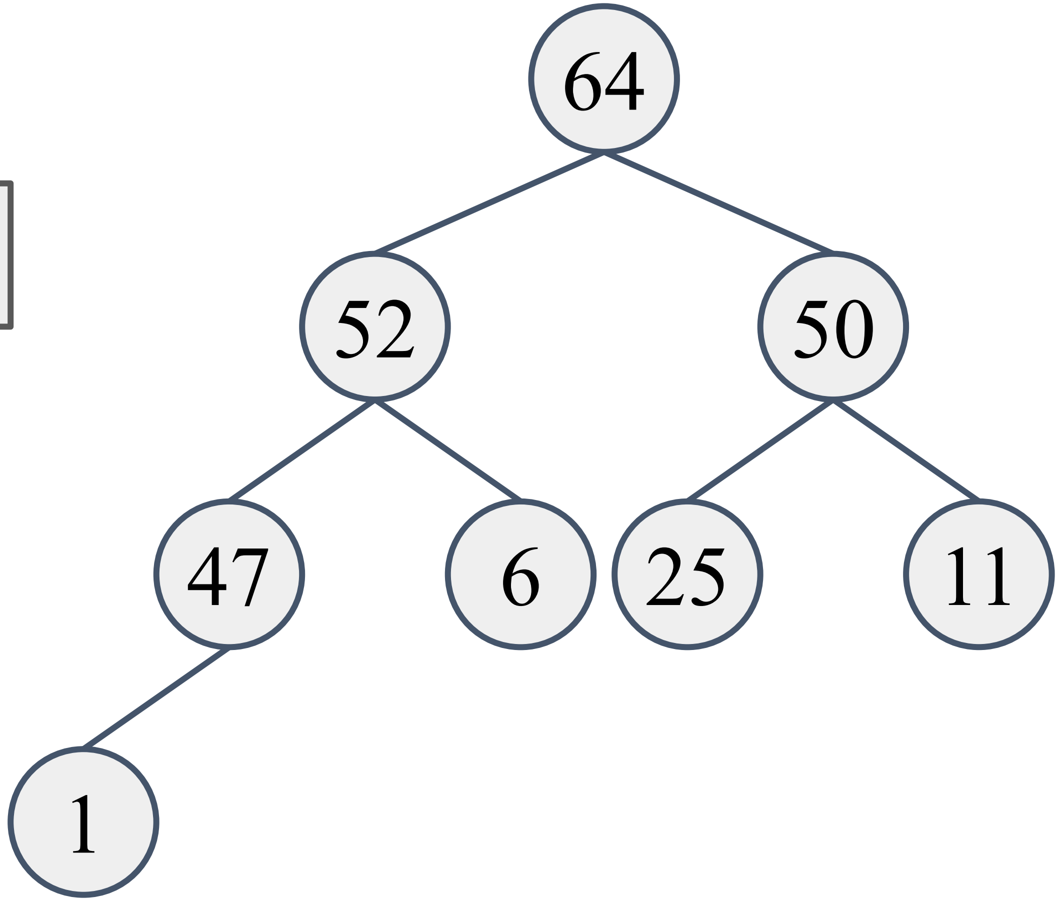
1. Attach a new vertex with value `v` and insert it into the **leftmost position at the bottommost level**. In the **1**-based indexed array, this value will be inserted at index $\mathbf{N+1}$.
2. Shift up (A.K.A swap up, bubble up) that value up until a valid spot in the heap is reached.



Complexity: $\mathbf{O(\log N)}$

`Insert(v)` (VisuAlgo slide)

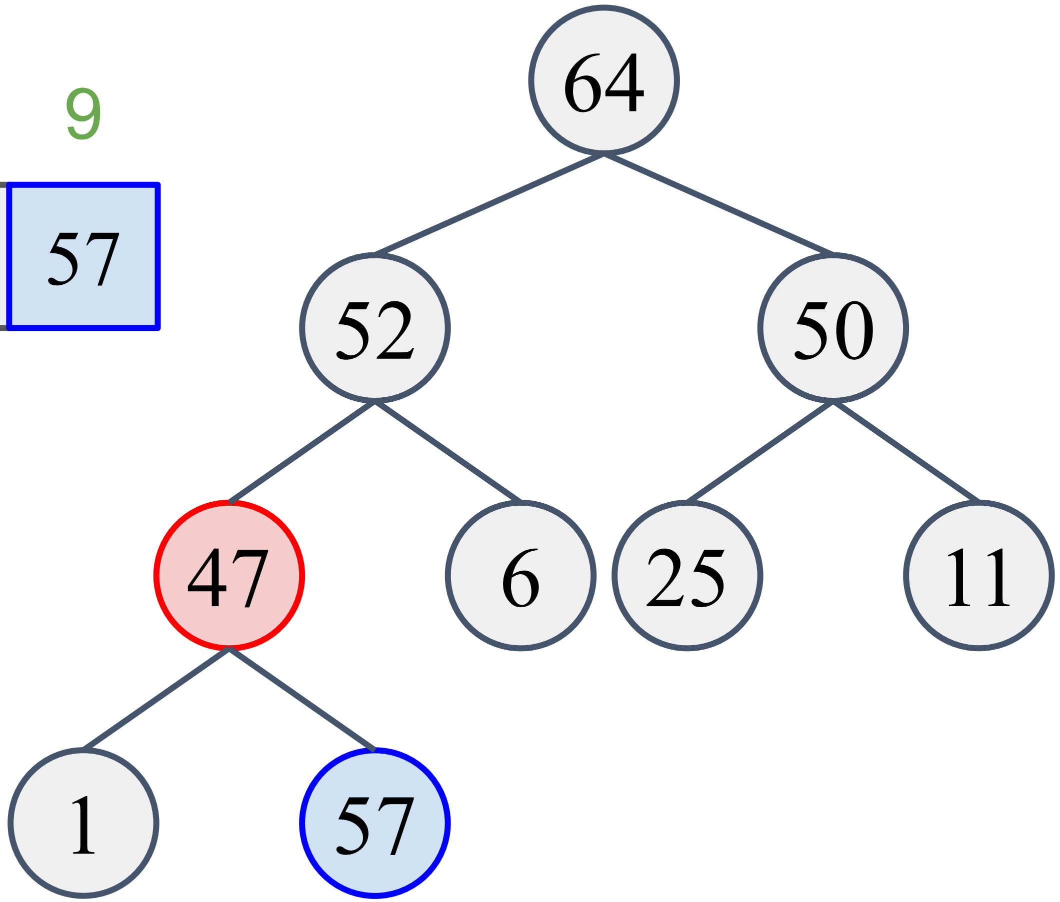
index	1	2	3	4	5	6	7	8
value	64	52	50	47	6	25	11	1



Say for instance, we insert the value 57 into this max-heap.

`Insert(v)` (VisuAlgo slide)

index	1	2	3	4	5	6	7	8	9
value	64	52	50	47	6	25	11	1	57

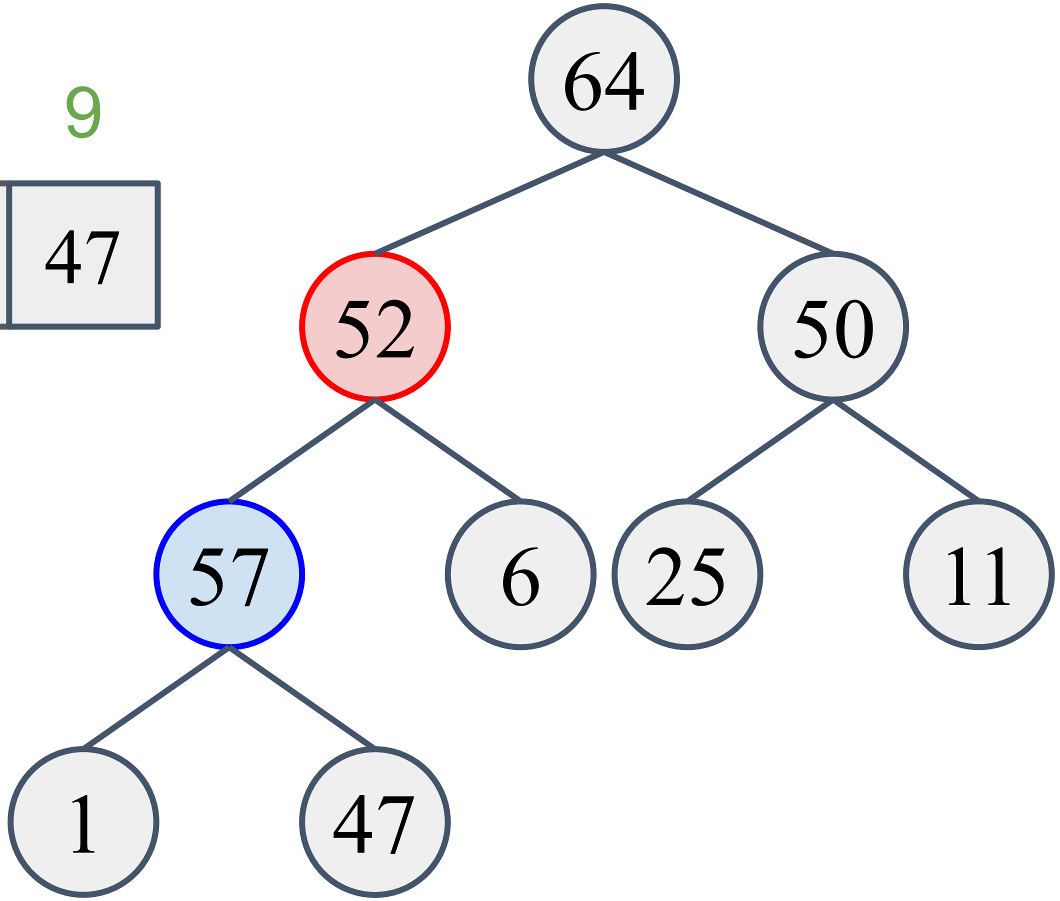


Insert 57 at last position in max-heap.

57 is greater than its parent 47, so we will shift up!

`Insert(v)` (VisuAlgo slide)

index	1	2	3	4	5	6	7	8	9
value	64	52	50	57	6	25	11	1	47

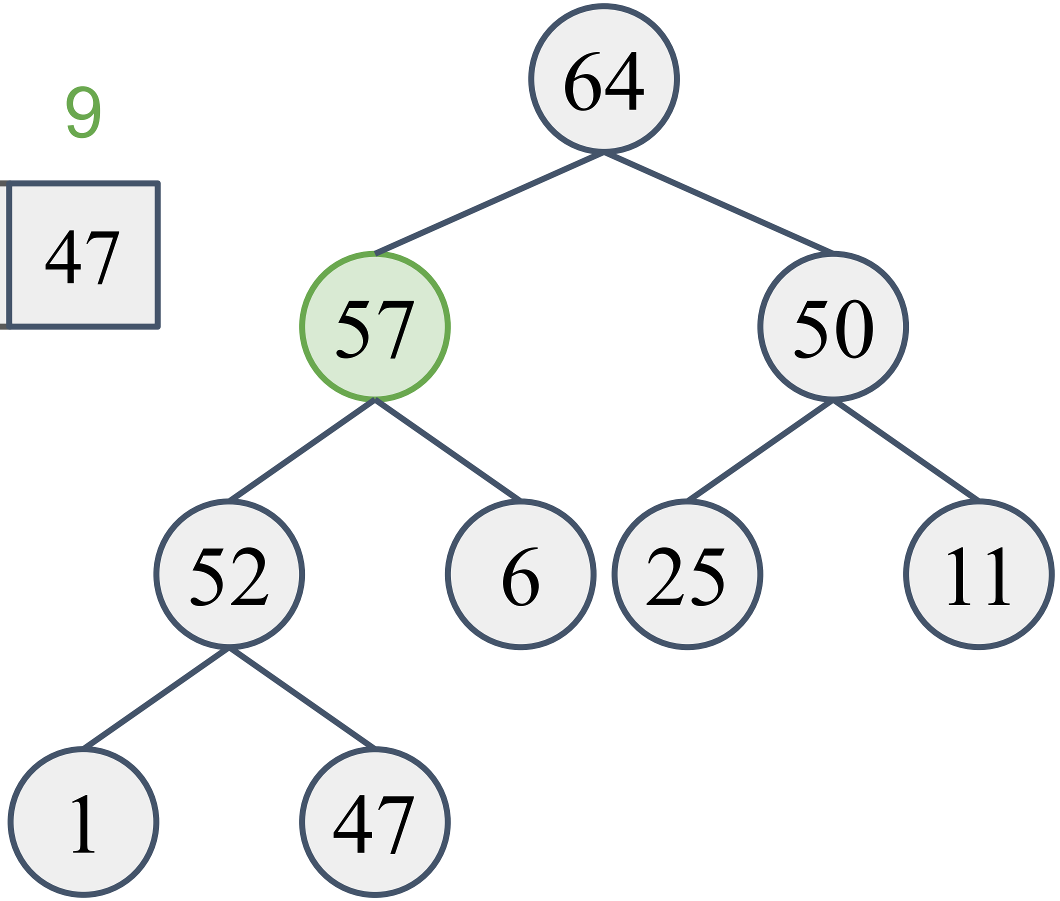


47 is now in its rightful spot.

57 is still greater than its new parent 52, so we will shift up!

`Insert(v)` (VisuAlgo slide)

index	1	2	3	4	5	6	7	8	9
value	64	57	50	52	6	25	11	1	47



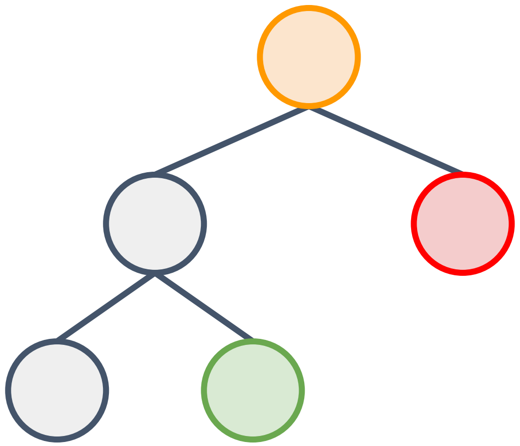
52 is now in its rightful spot.

57 is no longer greater than its new parent 64, so we are done!

`ExtractMax()` ([VisuAlgo slide])

Save **topmost vertex** in max-heap.

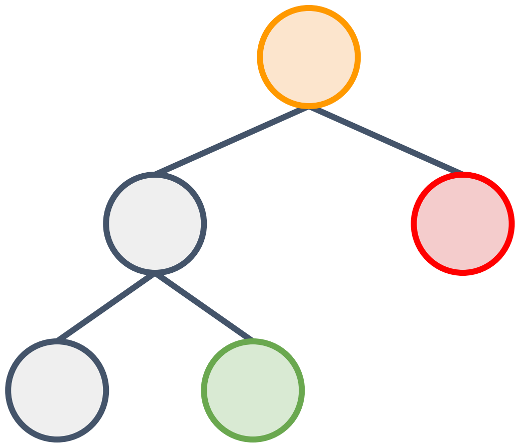
1. Replace topmost vertex with **rightmost vertex on the bottommost level***. In the **1**-based indexed array, this value is at index $\mathbf{N}$.



`ExtractMax()` (VisuAlgo slide)

Save **topmost vertex** in max-heap.

1. Replace topmost vertex with **rightmost vertex on the bottommost level***. In the **1**-based indexed array, this value is at index $\mathbf{N}$.

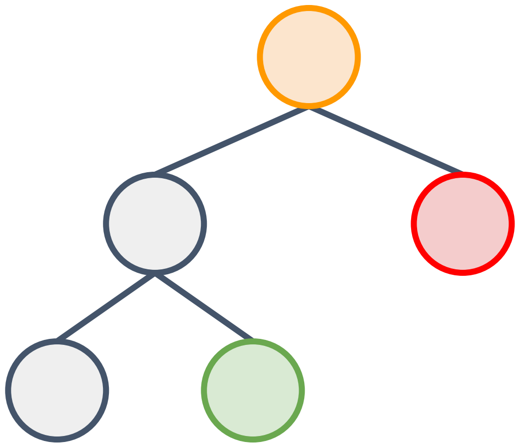


* This may not be the 2nd maximum element. 2nd maximum element is actually one of the two childs of root.

`ExtractMax()` (VisuAlgo slide)

Save **topmost vertex** in max-heap.

1. Replace topmost vertex with **rightmost vertex on the bottommost level***. In the **1**-based indexed array, this value is at index **N**.
2. Shift down (A.K.A swap down, bubble down) that value until a valid spot is reached.
3. Return saved topmost vertex.

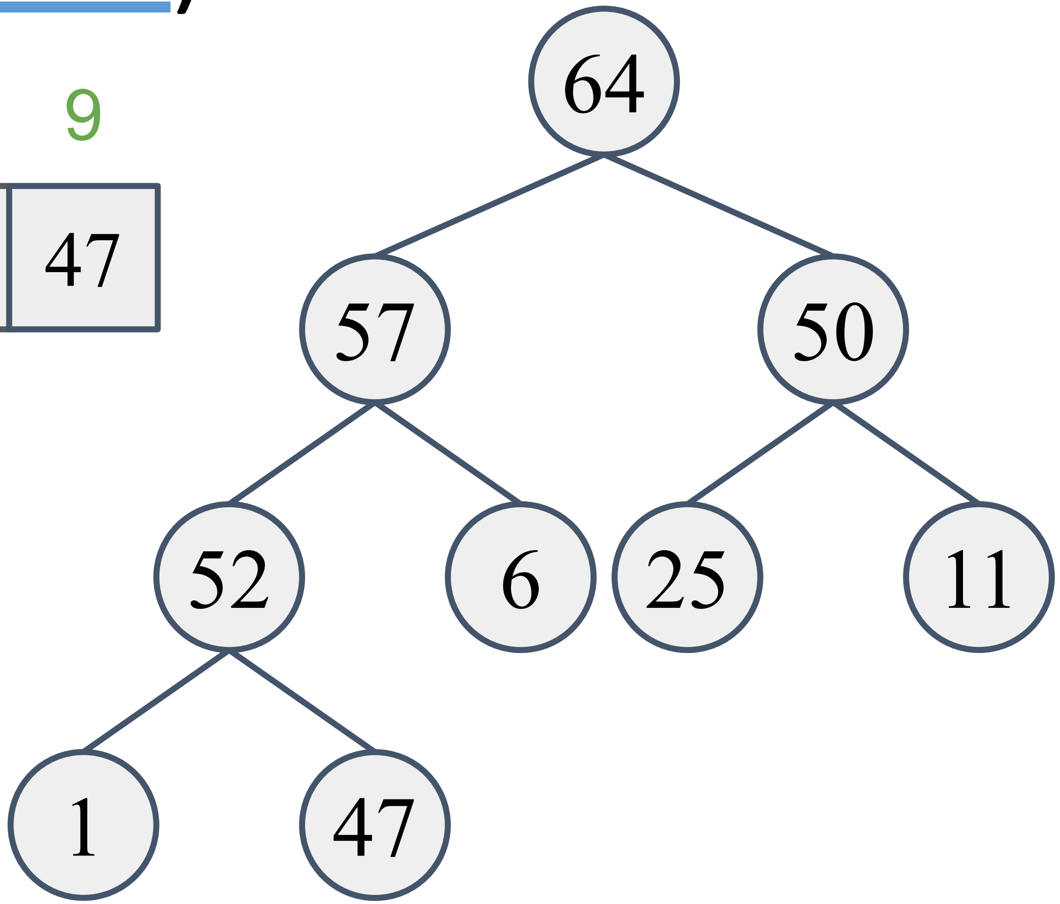


Complexity: **O(log N)**

* This may not be the 2nd maximum element. 2nd maximum element is actually one of the two childs of root.

`ExtractMax()` ([VisuAlgo slide]())

index	1	2	3	4	5	6	7	8	9
value	64	57	50	52	6	25	11	1	47

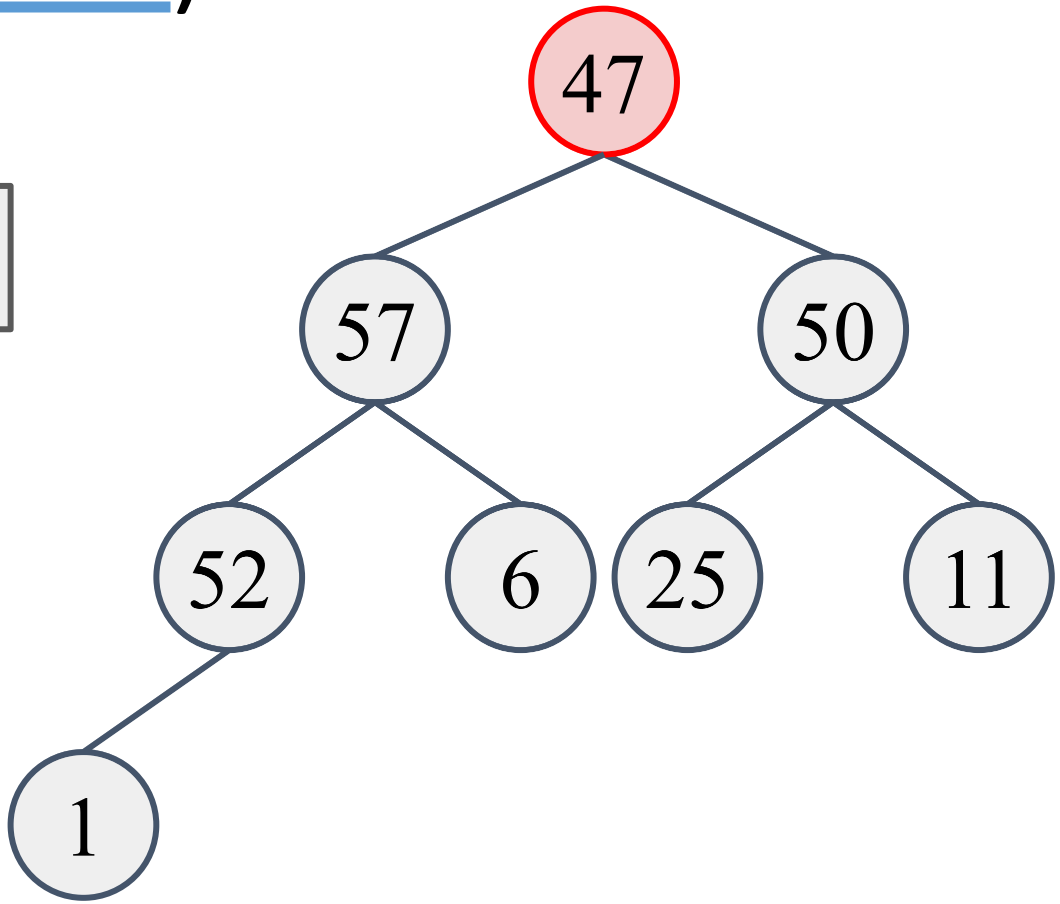


Say for instance, we call `ExtractMax()` on this max-heap.

We'll remove 64 and replace it with 47

`ExtractMax()` (VisuAlgo slide)

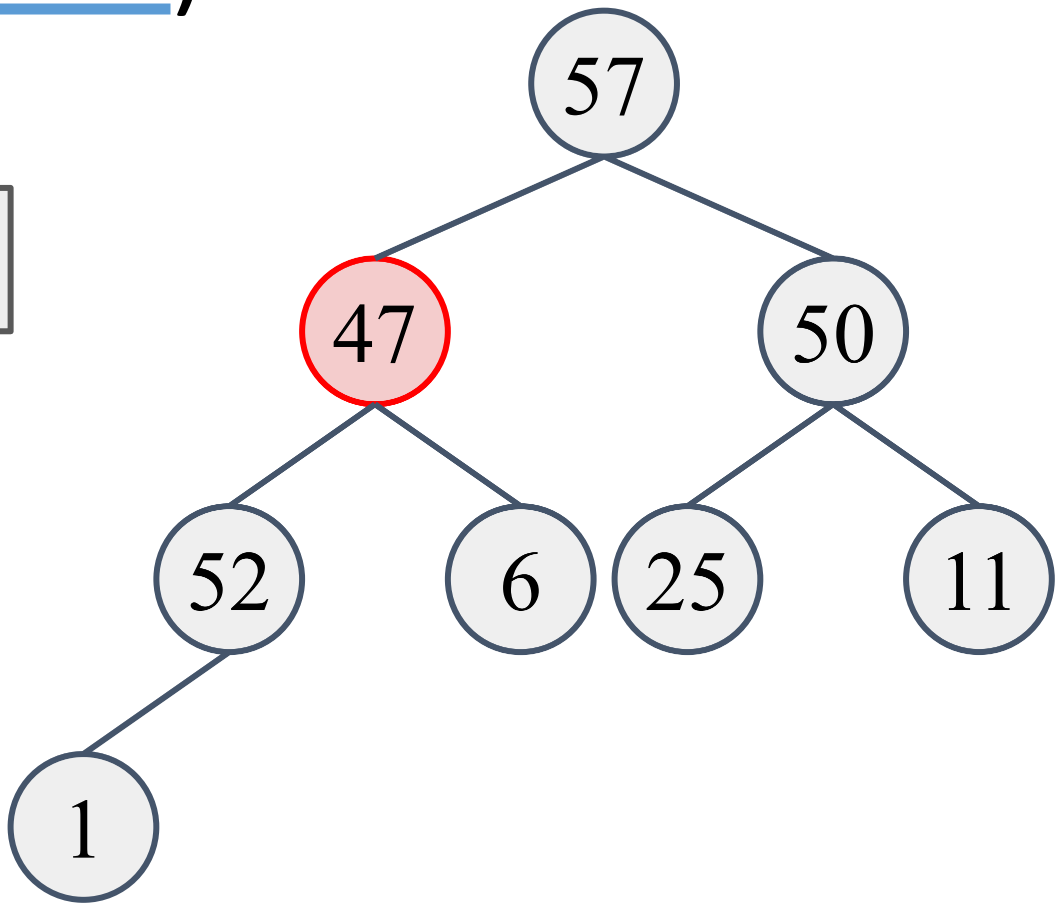
index	1	2	3	4	5	6	7	8
value	47	57	50	52	6	25	11	1



47 is lower than both its children, so we will shift it down with 57, the greater of the two.

`ExtractMax()` (VisuAlgo slide)

index	1	2	3	4	5	6	7	8
value	57	47	50	52	6	25	11	1

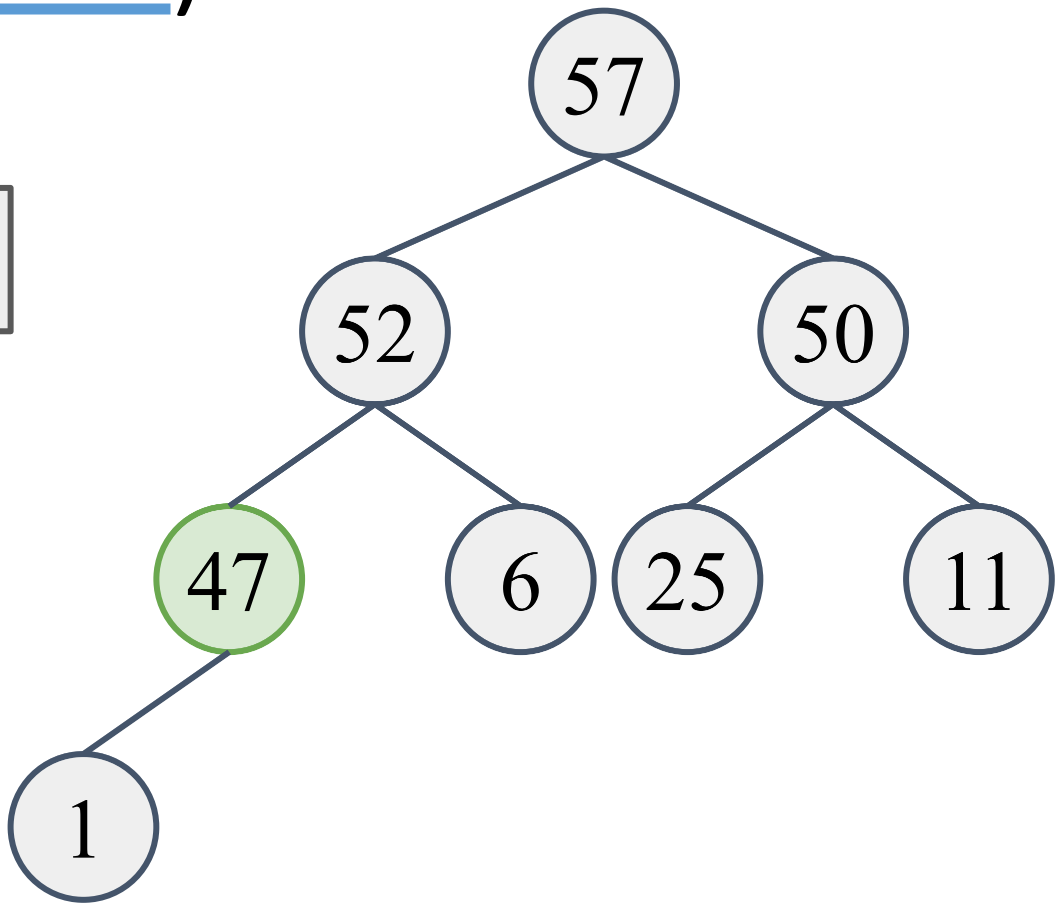


57 is now in its rightful spot.

47 is lower than one of its new children, so we will shift it down with 52, the greater of the two.

`ExtractMax()` ([VisuAlgo slide])

index	1	2	3	4	5	6	7	8
value	57	52	50	47	6	25	11	1



52 is now in its rightful spot.

47 is now greater than all its children, so we are done!

`Create(A)` (VisuAlgo slide)

Two versions:

1. **O(N log N)** version (VisuAlgo slide)
2. **O(N)** version (VisuAlgo slide)

`Create(A)` O(N log N) version (VisuAlgo slide)

Approach: `Insert` each and every value from array `A` into the heap that is being built.

N values total and each value potentially bubbled-up **log N** levels so total time complexity is **O(N log N)**.

A more mathematical analysis:

$$\log 1 + \log 2 + \log 3 + \ldots + \log N \leqslant \log N + \log N + \log N + \ldots + \log N$$

$$\log 1 + \log 2 + \log 3 + \ldots + \log N \leqslant N \log N$$

$$O(\log 1 + \log 2 + \log 3 + \ldots + \log N) = O(N \log N)$$

`Create(A)` $O(N)$ version (VisuAlgo slide)

Invented by Robert W. Floyd in 1964.

Approach: Take in entire compact array as a raw complete binary tree and heapify it level-by-level from the "bottom-up".
https://visualgo.net/en/heap?slide=7-2

Pseudocode:

For each vertex $\mathbf{v}_\mathbf{i}$ from $\mathbf{v}_\mathbf{N}$ down to $\mathbf{v}_\mathbf{1}$:
    shift-down($\mathbf{v}_\mathbf{i}$)

`Create(A)` $O(N)$ version analysis

But why is it **$O(N)$**? A loose analysis using same reasoning as insertion method seem to also suggest complexity **$O(N \log N)$**!

It turns out that the tight bound is not **$O(N \log N)$**.

The key difference here is that bubble-up and bubble-down (heapify) operations **have different time complexities depending on where they start from**:

- Bubble-up from a leaf: **$O(\log N)$** comparisons in the worst case when all levels traversed
- Bubble-down from a given vertex: **$O(h)$** comparisons in the worst case where h is the height of the subheap rooted at the vertex we wish to bubble-down.

`Create(A)` $O(N)$ version analysis

$N = 2^{h+1} - 1$ for where h is the height

For a heap which is a full binary tree (worst case) with $\mathbf{N}$ vertices and C being a constant number of comparison at each vertex:

- It has $(N + 1) / 2$ leaves. Since all leaves are by themselves valid subheaps, heapifying that level takes 0 comparisons
- Next level up has $(N + 1) / 4$ vertices
  - Each subheap rooted at them have a height of 1
  - Heapifying that level takes $((N\ 1\ ) / 4)(1)(C)$ comparisons
- Next level up has $(N + 1) / 8$ vertices
  - Each subheap rooted at them have a height of 2
  - Heapifying that level takes $((N + 1) / 8)(2)(C)$ comparisons
- And so on… until we reach the root node ($1 = (N + 1) / (N+1)$ vertices)
  - It has height of $h_{max} = \log_2(N+1) - 1$
  - Heapifying it takes $(1)(h_{max})(C)$ comparisons

`Create(A)` $O(N)$ version analysis

Total complexity is therefore a summation of aggregated heapify costs across every level in the heap. This is captured by the expression:

Summation across all heights from leaves with 0 height to root node with maximum height h_{max}.

$$\boxed{\sum_{h=0}^{h_{max}}}\boxed{\left(\frac{N+1}{2^{h+1}}\right)}\boxed{(ch)}$$

Cost to heapify from height h, where c is just a constant

Number of vertices at height h
See slide 29

`Create(A)` $O(N)$ version analysis

$$O\left(\sum_{h=0}^{h_{max}} \frac{(N+1)(ch)}{2^{h+1}}\right)$$

Take *Big O* on expression

$$= O\left((N+1)(c)\sum_{h=0}^{h_{max}} \frac{h}{2^h}\right)$$

Factor out non-h terms

$$= O\left((N+1)\sum_{h=0}^{h_{max}} \frac{h}{2^h}\right)$$

Throw away constant C

$$= O\left((N+1)\sum_{h=0}^{\infty} h\left(\tfrac{1}{2}\right)^h\right)$$

Sum to infinity (since convergent series)

$$= O\left((N+1)\left(\frac{\frac{1}{2}}{(1-\frac{1}{2})^2}\right)\right)$$

Using the formula $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ where $k=h$, $x=½$

$$= O(2N+2)$$

$$= O(N)$$

Q.E.D

Question 2

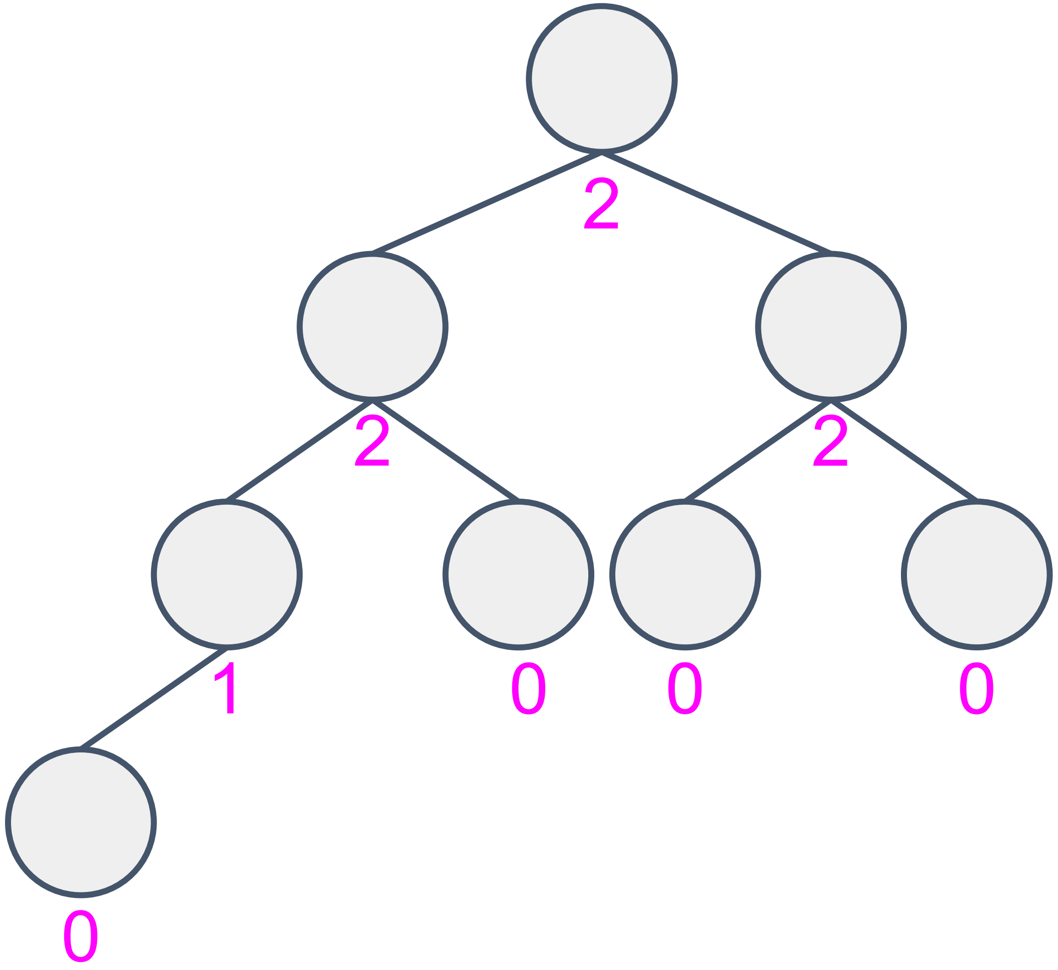
What is the **minimum** and **maximum** number of comparisons between Binary Heap elements required to construct a Binary (Max) Heap of arbitrary n elements using the $\mathbf{O(N)}$ `Create(A)`?

Realize that for `heapify`, each vertex must have make the number of comparisons equal to its number of children. For a vertex with 2 children, it must first check which is the greater of the 2 children, then it must check if its value is lower than the greater of its 2 children.

Question 2

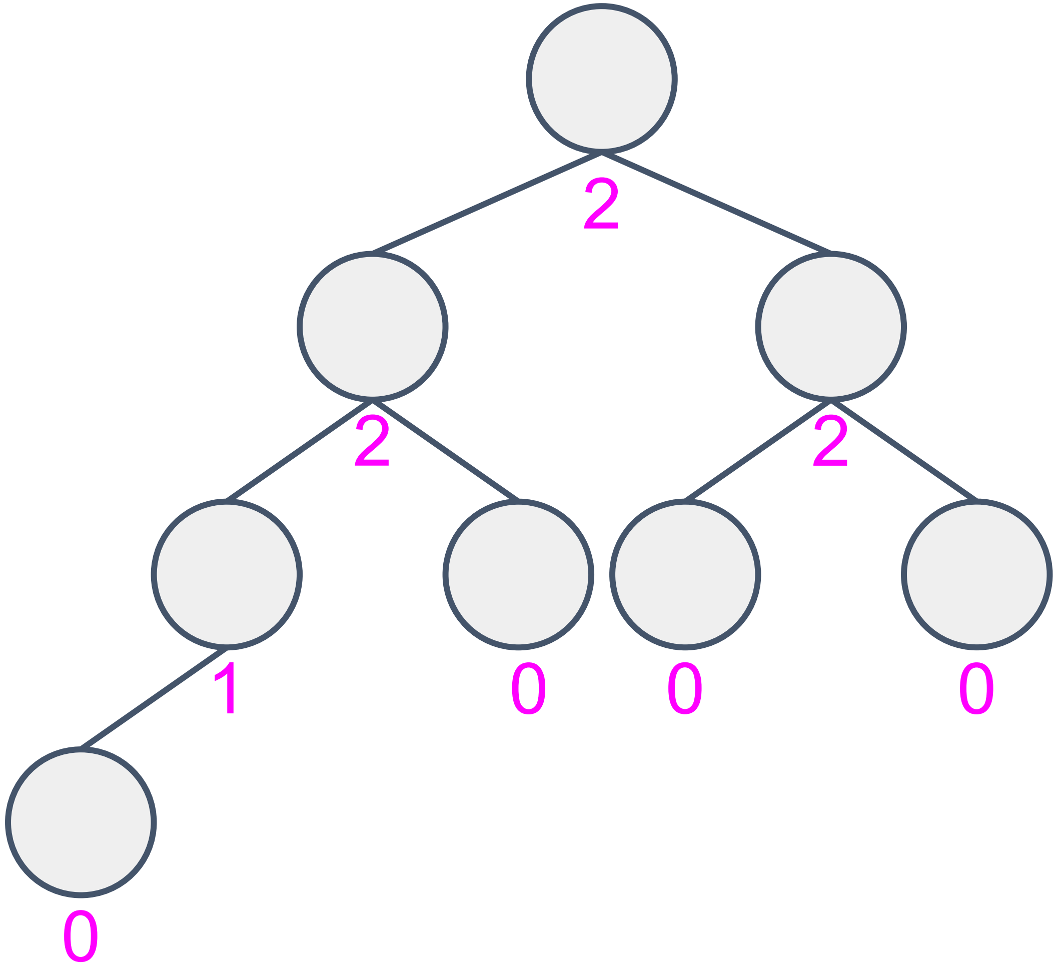
Take for instance a heap with 8 vertices.

Number of comparisons needed at each vertex displayed below vertex in pink



Question 2

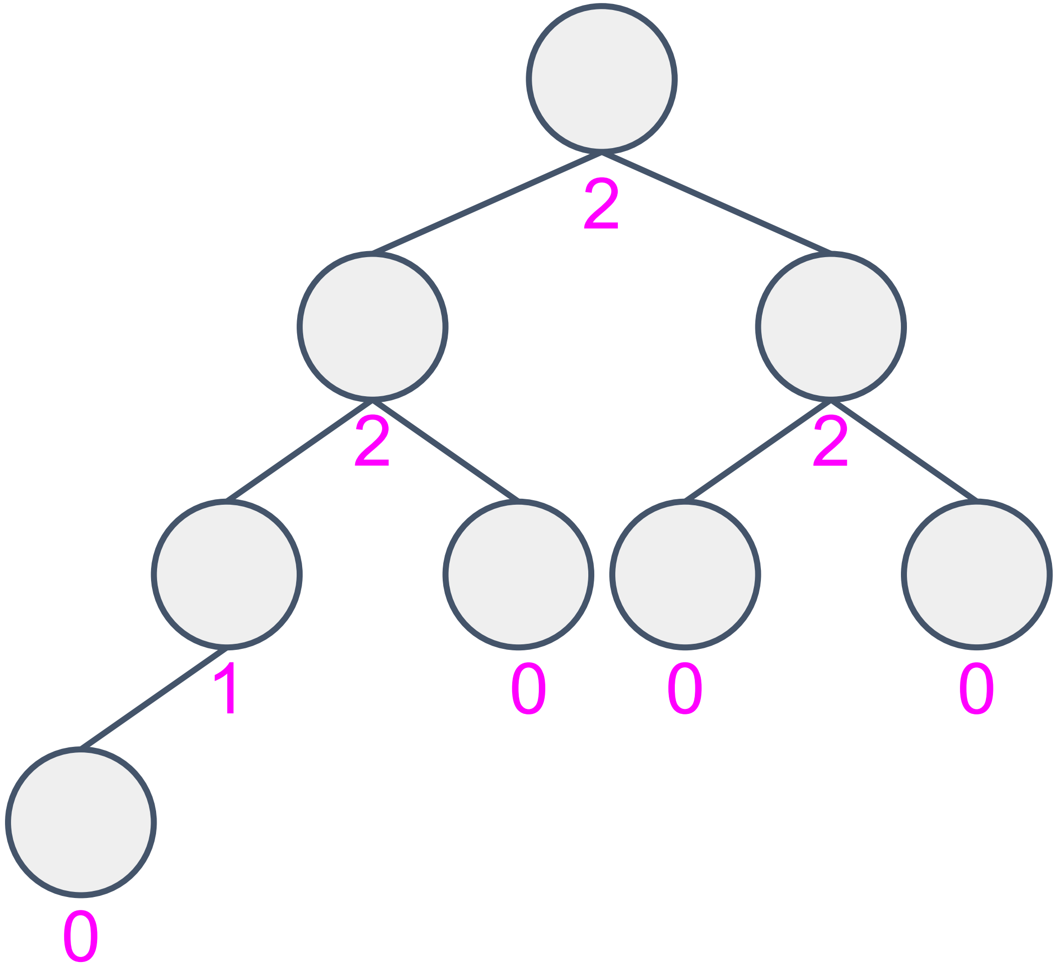
We incur minimum number of total comparisons if the array is already in heap order. So each vertex just need to conduct its own comparisons once and determine that no further heapify is required.



So minimum comparisons is $1 + 2 + 2 + 2 = 7$.

Question 2

We incur maximum number of total comparisons if each vertex need to heapify down its entire height to the leaves, incurring comparisons at each vertex along the way. i.e. We were given a min-heap!



So max comparisons is $\mathbf{1 + (2 + 1) + 2 + (2 + (2 + 1)) = 11}$.

Question 3

Give algorithm to count all vertices that have value $>x$ in a max-heap of size $\mathbf{n}$.

Your algorithm must run in $\mathbf{O(k)}$ time where $\mathbf{k}$ is the number of vertices in the output.

Key lesson: This is a new algorithm analysis type for most of you as the time complexity of the algorithm does not depend on the input size $\mathbf{n}$ but rather the output size $\mathbf{k}$.

Observation

“A binary heap is made up of many binary subheaps”.

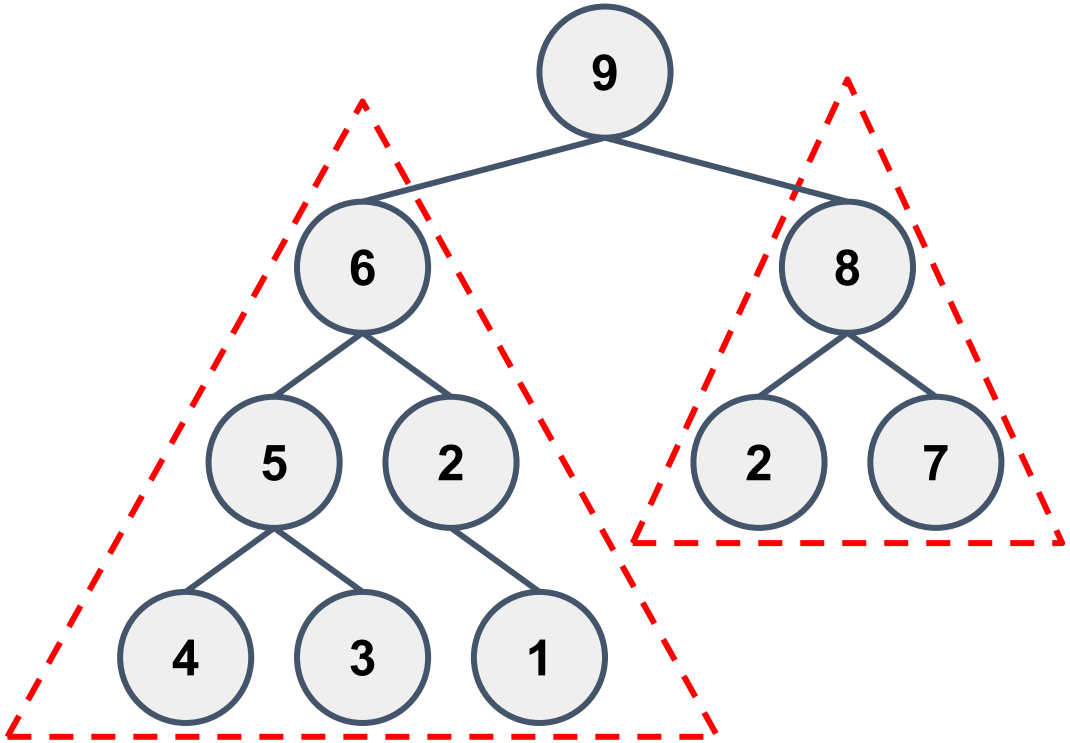
More specifically, a binary max-heap is a complete binary tree where the root vertex has value greater than both its children and each of its children is also a binary heap.

This recursive definition should give you a hint that the solution is likely recursive in nature too.

In fact, most tree-related algorithms are recursive because trees are defined recursively!

Observation

These two are subtrees that are by themselves binary heaps!



Observation

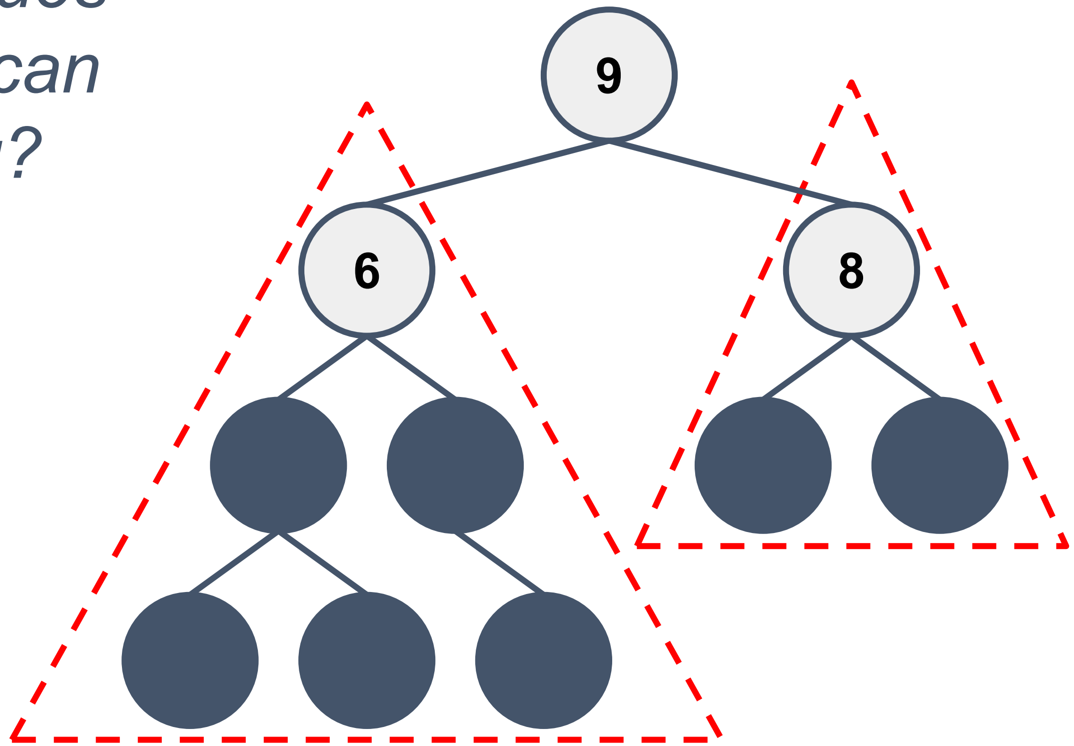
Recall for a binary heap, the root vertex has the greatest value in the entire heap.

i.e. All other vertices (i.e. non-root) `v` in the binary heap, have `v.value <= root.value`

Example

Let's say I want to find all vertices with value >7.

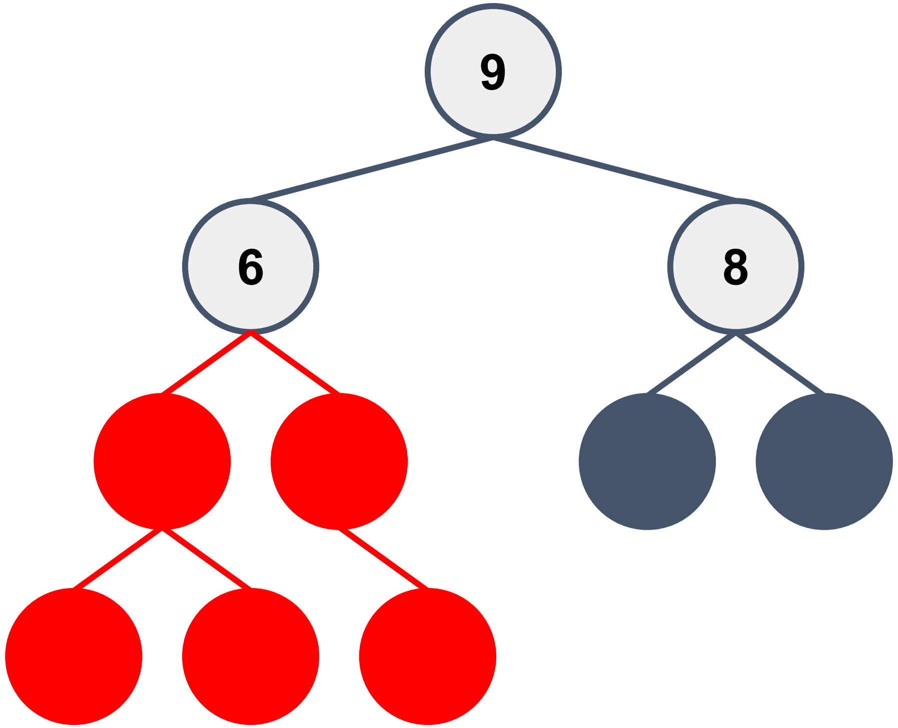
If I hide all the values except the roots, can you infer anything?



Example

Let's say I want to find all vertices with value >7.

You can infer that all these nodes are ≤6 and therefore less than what we want!



Approach

Traverse the tree, for every vertex `v` that we encounter:

- If the value of the vertex `v` is lesser than or equal to `x`, then we can stop the search.
- Else we output `v` and continue searching down its children.

Solution

We can implement the solution recursively

```
void findVerticesBiggerThanX(vertex v, int x) {
    if (v.value > x) {
        System.out.println(v.value);          // Output
        findVerticesBiggerThanX(v.left, x);   // Recurse left
        findVerticesBiggerThanX(v.right, x); // Recurse right
    }
}
```

Test yourself!

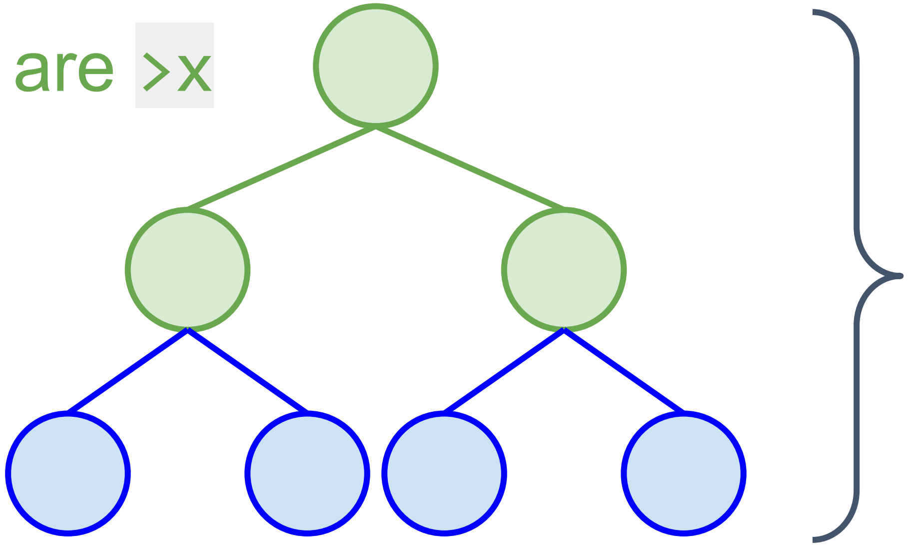
If the answer has $\mathbf{K}$ outputs, what is the total number comparisons done?

Test yourself!

If the answer has $\mathbf{K}$ outputs, what is the total number comparisons done?

Answer: $\leq \mathbf{2K}$ (i.e. at most **2** comparisons for every vertex with value >x)

K vertices compared that are >x



Additional vertices compared

Time complexity

Total time complexity is the number of output operations plus the number of vertices compared so,

$$\mathbf{O(K + 2K) = O(K)}$$

Question 4

Show an easy way to convert a Binary Max Heap of a set integers into a Binary Min Heap without changing the underlying data structure at all.

Question 4

Show an easy way to convert a Binary Max Heap of a set integers into a Binary Min Heap without changing the underlying data structure at all.

Insert negative of the numbers. Negate again when extracting.

Question 5

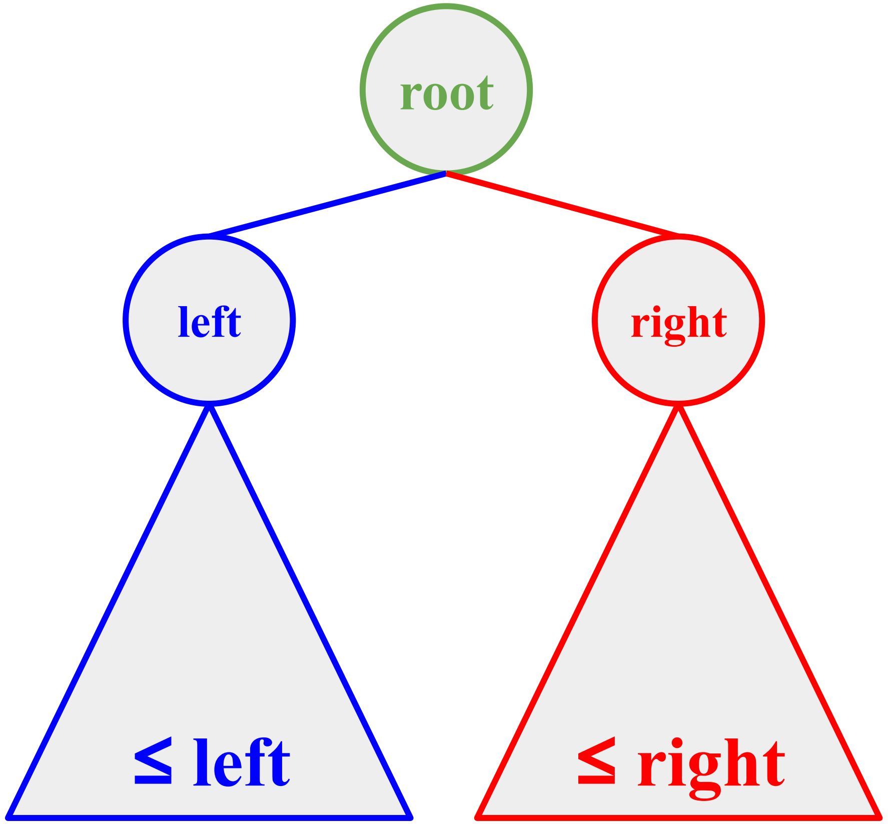
Claim: The second largest element in a max heap with more than two elements is always one of the children of the root. Is this true? If yes, show a simple proof. Otherwise, show a counter example.

For simplicity please assume that all values are unique!

Note that this kind of (simple) proof may appear in future CS2040S written tests.

Question 5: Proof by contradiction

1. It cannot be root since it's largest
2. Assume that 2^{nd} largest is neither **left** nor **right,** and Realize both **left** and **right** must be lesser than 2^{nd} largest.
3. That means 2^{nd} largest is a descendant of either **left** or **right.**
4. But descendants cannot be greater than the ancestors in a max-heap! So this is not possible!
5. We reached a contradiction and therefore 2^{nd} largest **must be** either **left** or **right**!



Tree Traversals

- Actually, this simple recursive algorithm is what we call a depth first search (DFS), a classic tree traversal algorithm!
- More specifically, it is a (pruned) pre-order traversal.
- The different types of tree traversals will become more important in *later weeks* of CS2040C.
- For now, just try to appreciate and understand what we mean. (Self-read the next few slides!)

Tree Traversals (self-read)

There are 3 common types of tree traversals:

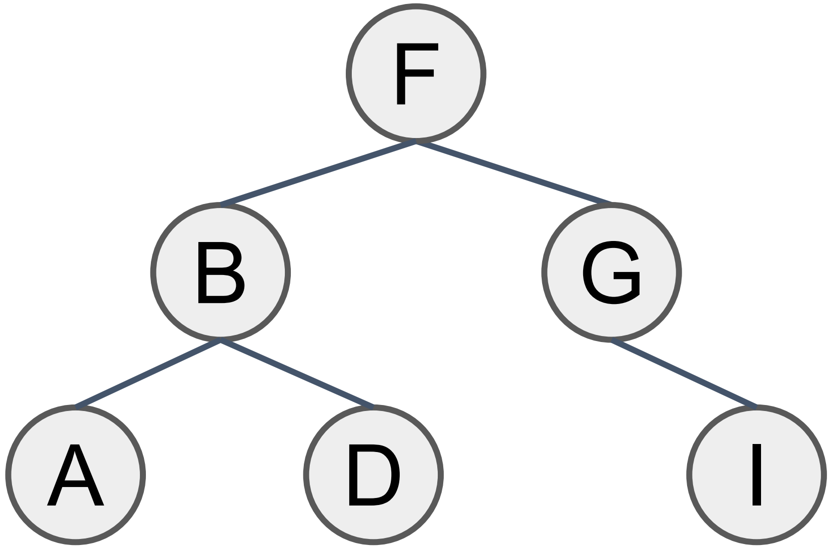
1. Pre-order traversal
2. In-order traversal
3. Post-order traversal

Their name comes from where the “current operation” in each level of the recursion is done in relation to the recursive calls of that level. “Current operation” can simply be printing out the value of the current vertex being visited.

Pre-order Traversal (self-read)

1. Current operation
2. Recruse left
3. Recruse right

```
void pre_order(vertex v) {
  if (v) {
    System.out.println(v.value);
    pre_order(v.left);
    pre_order(v.right);
  }
}
```



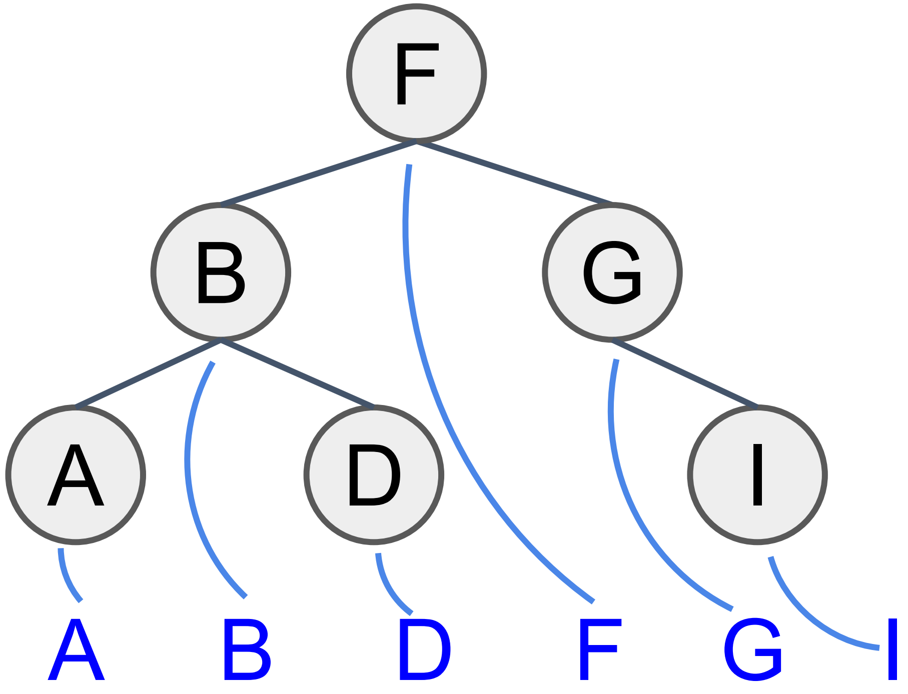
F B A D G I

Print order in blue

In-order Traversal (self-read)

1. Recurse left
2. Current operation
3. Recruse right

```
void in_order(vertex v) {
  if (v) {
    in_order(v.left);
    System.out.println(v.value);
    in_order(v.right);
  }
}
```

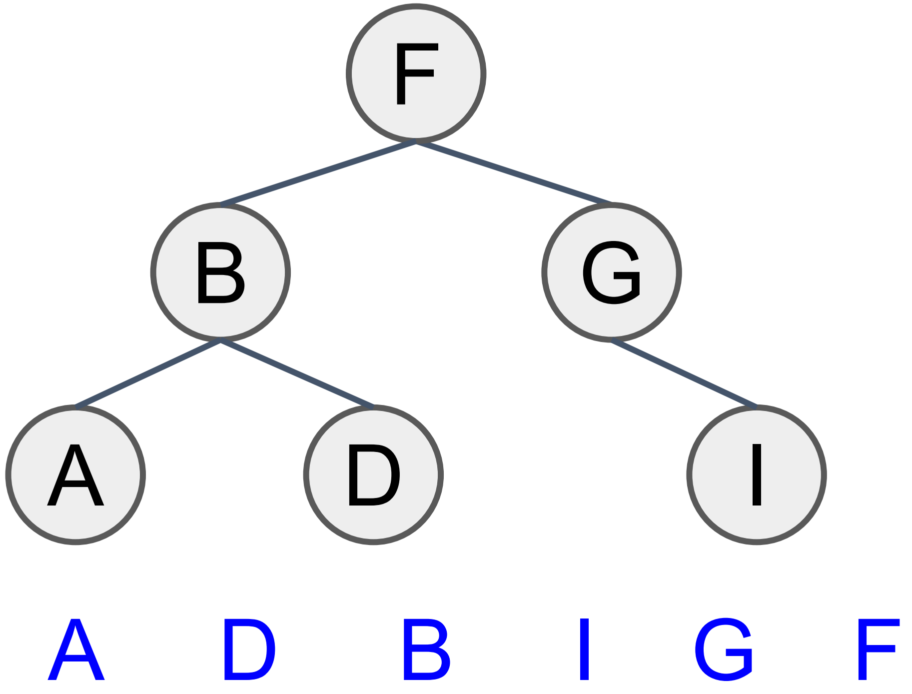


Print order in blue
Resembles "falling leaves" of a tree where lines don't cross each other!

Post-order Traversal (self-read)

1. Recurse left
2. Recurse right
3. Current operation

```
void post_order(vertex v) {
  if (v) {
    post_order(v.left);
    post_order(v.right);
    System.out.println(v.value);
  }
}
```



Print order in blue

Question 6

Question 6a

Java PriorityQueue doesn't have `DecreaseKey(old_v, new_lower_v)`. How to implement it?

This discussion shall also suffice for analysis of `Increase` as well as generalised `UpdateKey(old_v, new_v)` since they are analogous.

Questions to ask

- How do we locate `old_v` and how fast can we do it?

Questions to ask

- How do we locate `old_v` and how fast can we do it?
  - **O(N)** DFS. Worst case when it is the smallest value in heap.
  - **O(1)** using Hash Table! Key is vertex value, Value is vertex index in compact array.
- Will it affect Complete Binary Tree (compact array) property? If so how do we handle?

Questions to ask

- How do we locate `old_v` and how fast can we do it?
  - **O(N)** DFS. Worst case when it is the smallest value in heap.
  - **O(1)** using Hash Table! Key is vertex value, Value is vertex index in compact array.
- Will it affect Complete Binary Tree (compact array) property? If so how do we handle?
  - No because no gaps created.
- Will it affect heap property? If so how do we handle?

Questions to ask

- How do we locate `old_v` and how fast can we do it?
  - **O(N)** DFS. Worst case when it is the smallest value in heap.
  - **O(1)** using Hash Table! Key is vertex value, Value is vertex index in compact array.
- Will it affect Complete Binary Tree (compact array) property? If so how do we handle?
  - No because no gaps created.
- Will it affect heap property? If so how do we handle?
  - Yes it will. After decreasing value, shift down. **O(log N)**.
- What’s the time complexity?
  - **O(log N)**.

Question 6b

Java doesn't have `DeleteKey(v)` where `v` is not necessarily the max element. How to implement it?

Solution 1: Building upon `UpdateKey`

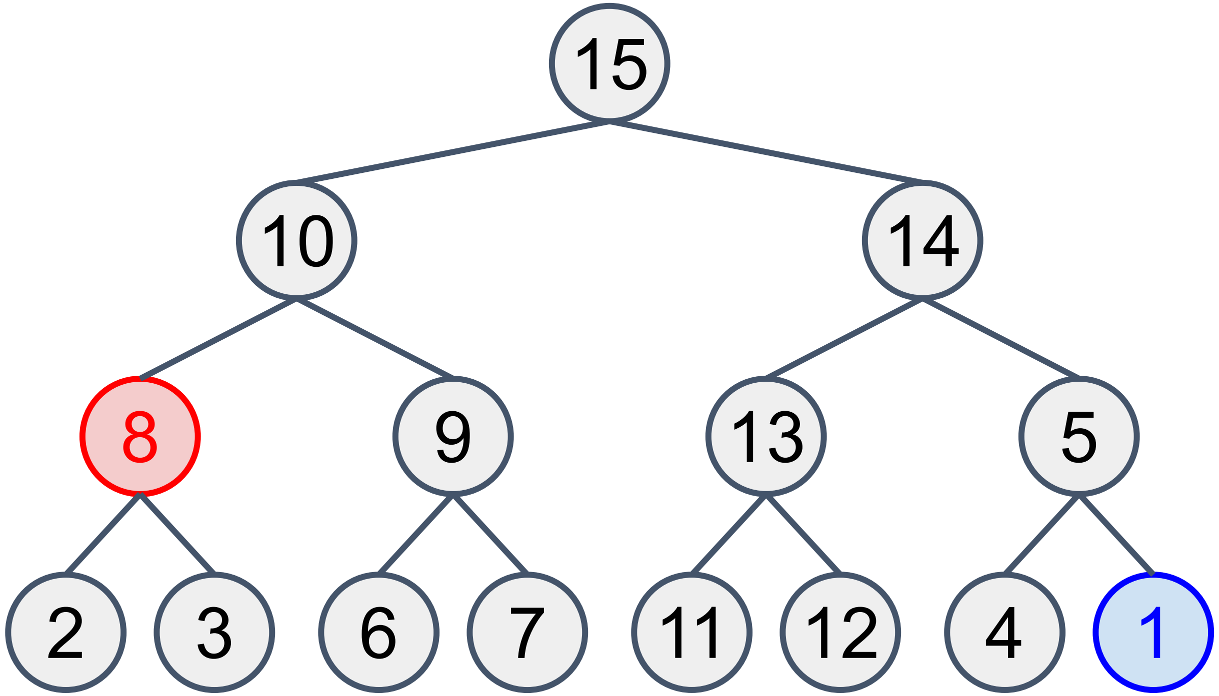
1. `UpdateKey(v, +∞)`
2. `ExtractMax()`

Solution 2: Generalising `ExtractMax()`

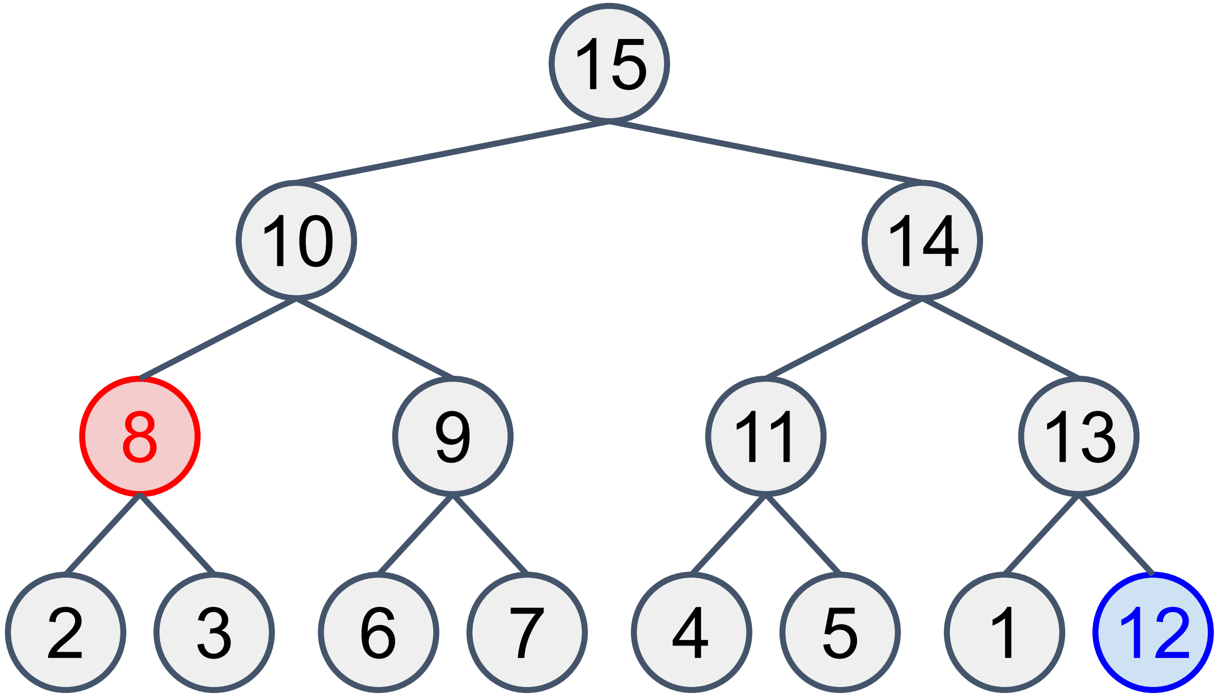
1. Find index of `i` vertex with value `v` in compact array
2. Move value at index `N` into value at index `i`
3. Then `shift_down` or `shift_up`?

Solution 2: Generalising `ExtractMax()`

Consider `DeleteKey(8)` for the two heaps below



This will need `shift_down`



This will need `shift_up`

Break

Attendance

Questions

Java PriorityQueue Documentation

link

https://visualgo.net/training?diff=Medium&n=5&tl=5&module=heap

PS3 Discussion

PS3: /sim

- Read the documentation of Java ListIterator: https://docs.oracle.com/javase/8/docs/api/java/util/ListIterator.html
- Every operation in the problem can be done in 1-2 lines using ListIterator.

PS3: /janeeyre

- Maintain a PriorityQueue containing all the books in stock.
  - Books in the PriorityQueue need to be sorted by title.
- Take the top book from PriorityQueue every time.
- However, some additional books may have arrived. Add them to PriorityQueue first before taking out a book.

Past Midterm Discussion

Thank You!

Anonymous Feedback:
https://forms.gle/MkETeXdUT53Vhh896