Tutorial 04 - PQ ADT

CS2040C Semester 1 2020/2021

By Matthew and Jin Zhe, AY19/20 S1, adapted from slides by Ranald+Si Jie, AY1819 S2

Key Points

- Binary Heap as Priority Queue
- Some properties
- Some PQ operations

Basic Binary Heap Stuffs

Priority Queue (PQ) ADT

Common PQ ADT operations

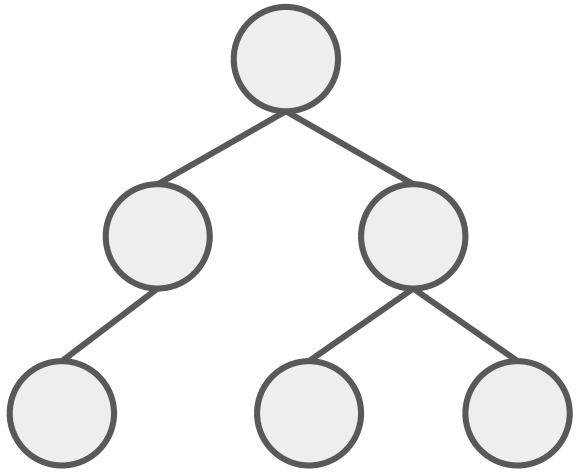
`insert(v)`	Insert item with value `v` into the PQ.
`ExtractMax()`	Return the the item with highest priority from PQ.
`create(A)`	Create a PQ from the given array `A`.
`HeapSort(A)`	Return the sorted order of items in given array `A`.

Binary tree

A binary tree **is a tree** where every vertex in the tree

- has at most 2 children (therefore *binary*)
- is itself a binary *subtree* rooted at that vertex

Example of a binary tree



The **minimum height** of a binary tree with a total of *N* vertices is *O(log N)*. Why? What's the maximum height?

Full binary tree

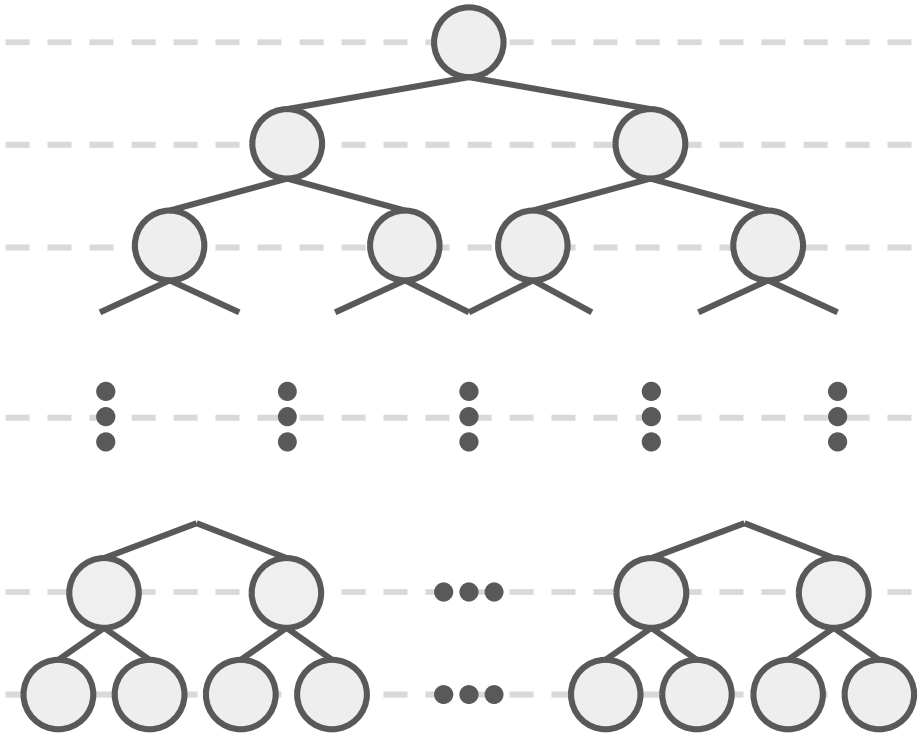
A full binary tree **is a complete binary tree** (more on this in a bit) in which every node other than the leaves has two children. i.e. It is completely filled.

In the next 2 slides, we shall highlight some noteworthy properties which will be useful for analysis later on.

There’s no need for you to memorize them, just appreciate will do!

Full binary tree

Full binary tree with N vertices



Level	Height	Vertices
0	$log_2(N+1)-1$	2^0
1	$log_2(N+1)-2$	2^1
2	$log_2(N+1)-3$	2^2
ℓ	h	v
$log_2(N+1)-2$	1	$(N+1)/4$
$log_2(N+1)-1$	0	$(N+1)/2$

Relationship between level ℓ, height h and vertices v

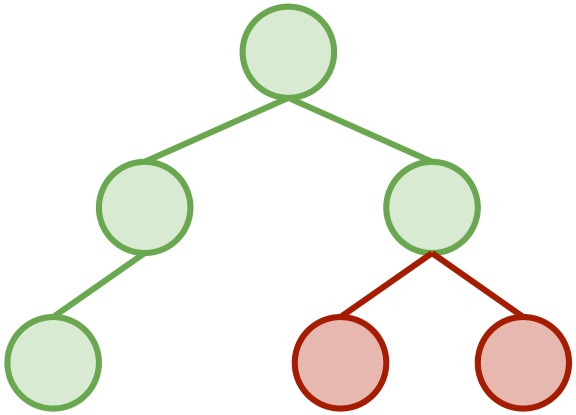
Formulas expressing column attributes in terms of row attributes

	ℓ	h	v
ℓ	ℓ	$log_2(N+1)-1-\ell$	2^{ℓ}
h	$log_2(N+1)-1-h$	h	$(N+1)/(2^{1+h})$
v	$log_2 v$	$log_2((N+1)/v)-1$	v

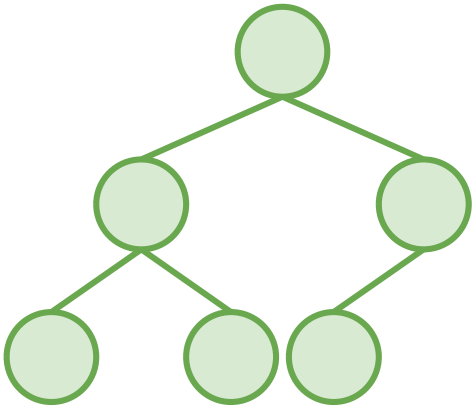
Complete binary tree

A *complete binary tree* **is a binary tree**, where for every level, except possibly the last, is completely filled, and all nodes in the last level are **as far left as possible**.

Note that the height of a complete binary tree with *N* vertices is *O(log N)*



Not a complete binary tree

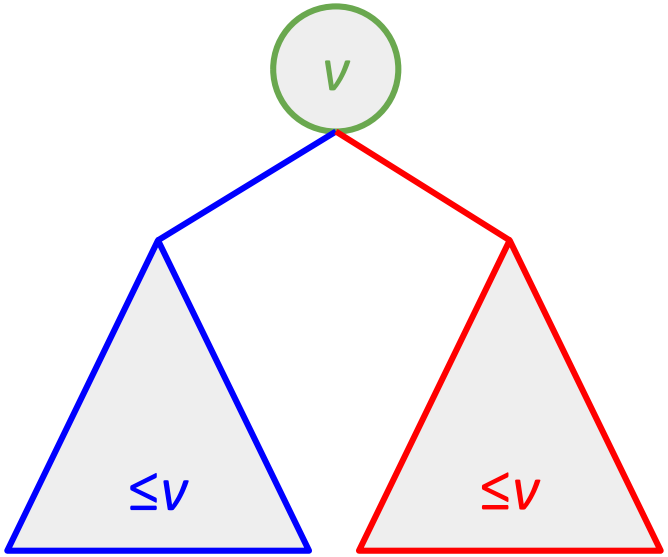


A complete binary tree

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

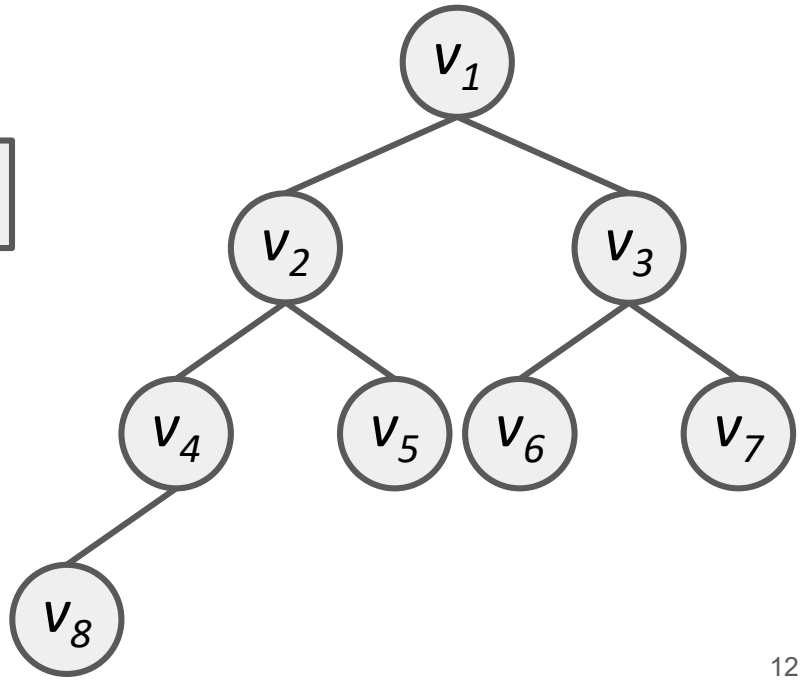
- v_i to represent value at index i
- L_i to represent left child index of index i
- R_i to represent right child index of index i
- $P_{i,j}$ to represent parent index of index i and j

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 $i \times 2$
R_i : index $i \times 2 + 1$
P_i : index $\lfloor i \div 2 \rfloor$

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

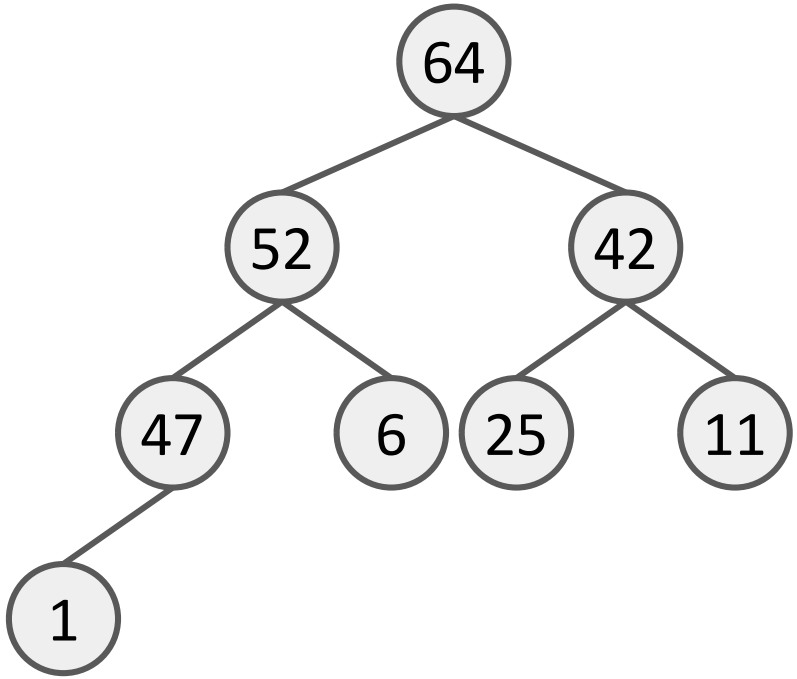
0-based indexing

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

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

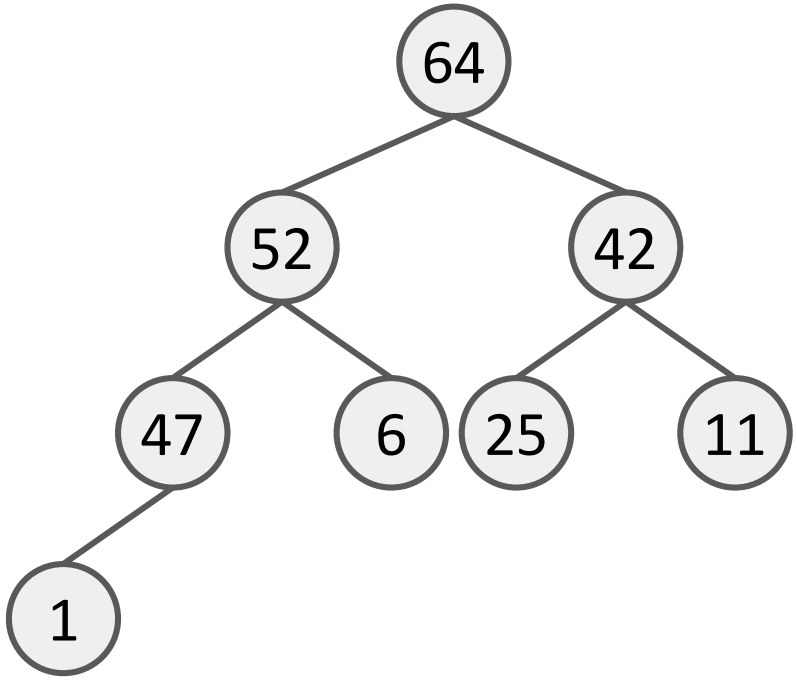
Is the value at the top of the max heap is ≥ all other values in the heap?



Test yourself

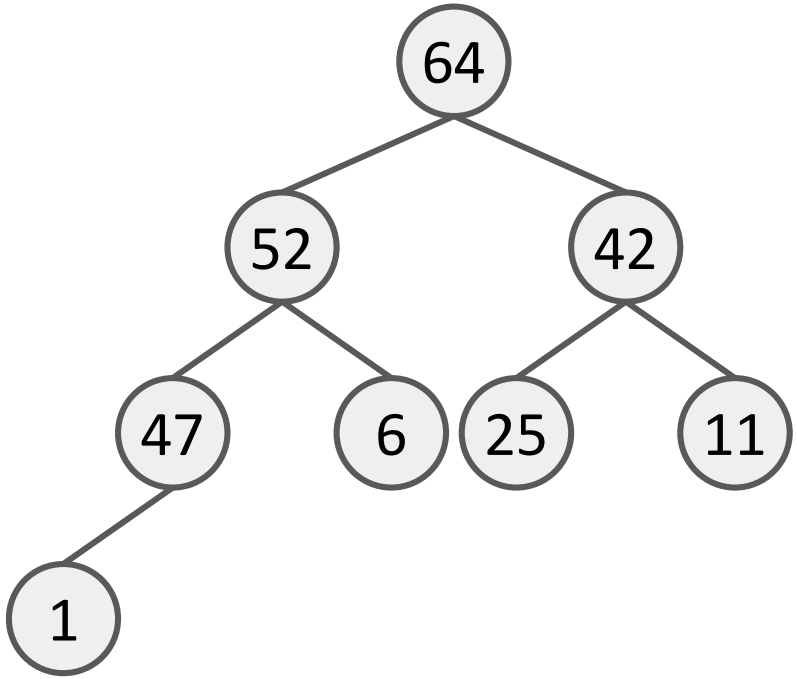
Is the value at the top of the max heap is ≥ all other values in the heap?

Yes! By definition!



Test yourself

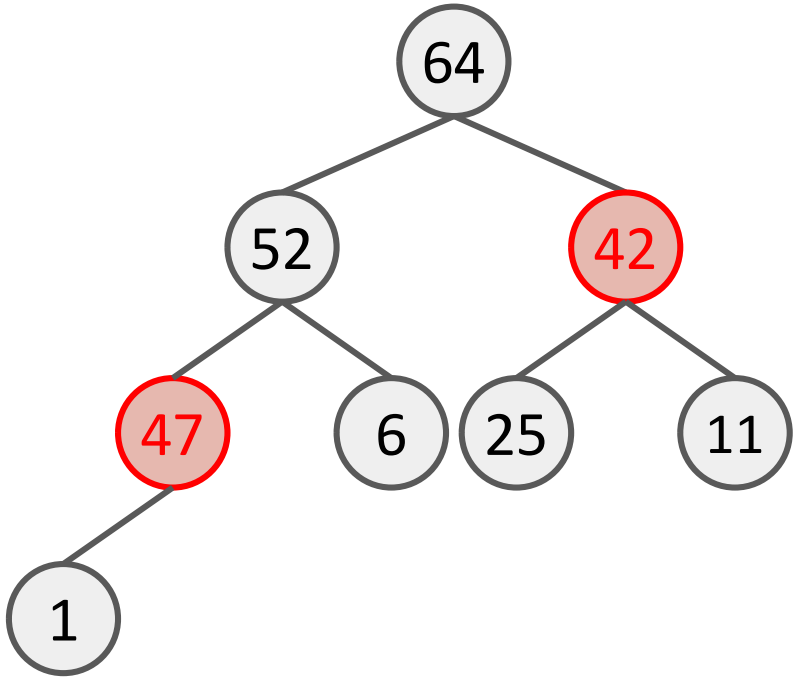
Must the value at a higher level be ≥ all values from lower levels?



Test yourself

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

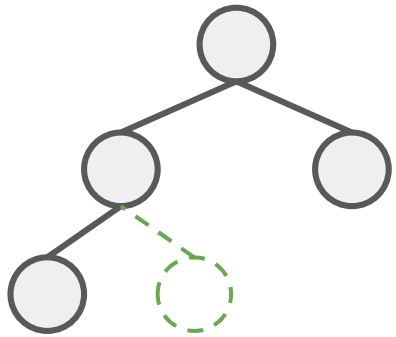
Not necessarily!



`Insert(v)` (VisuAlgo slide)

Steps:

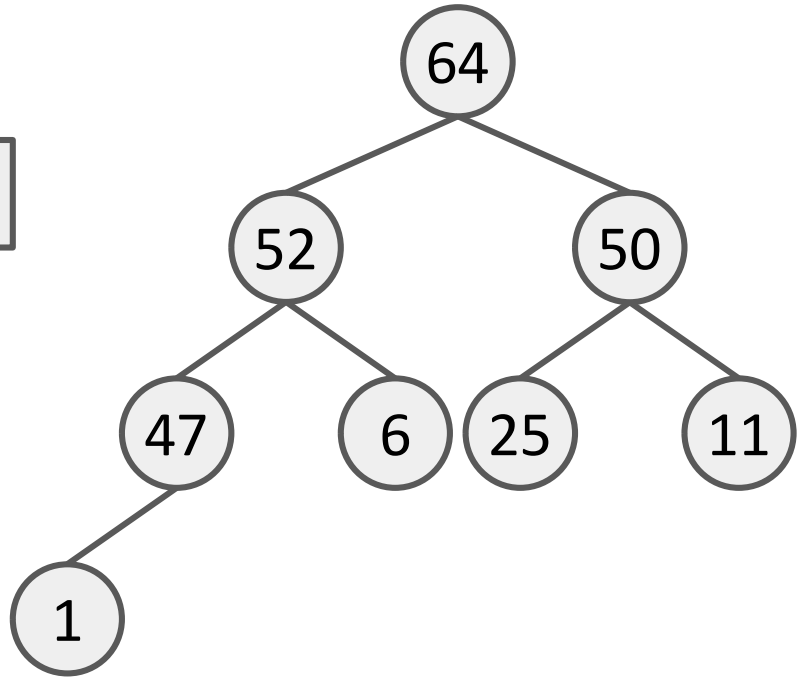
1. Attach a new vertex with value `v` and insert it into the **leftmost position at the bottommost level**. In the 1-based indexed array, this value will be inserted at index $N + 1$
2. *Bubble up* (A.K.A *swap up*, *shift up*, *increase key*) that value up until a valid spot in the heap is reached



Complexity: $O(\log N)$

`Insert(v)` (VisuAlgo slide)

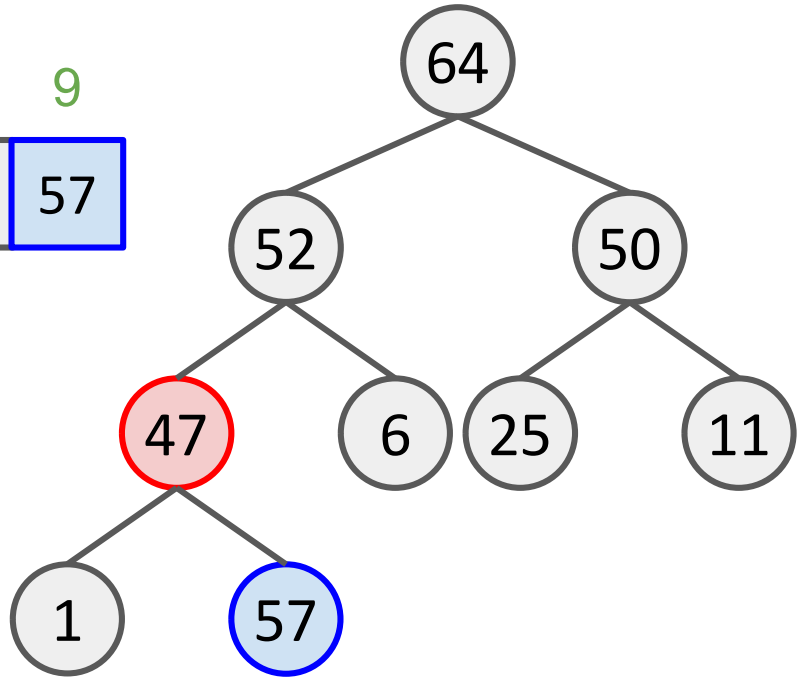
index	1	2	3	4	5	6	7	8
value	64	52	50	47	6	25	11	1



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

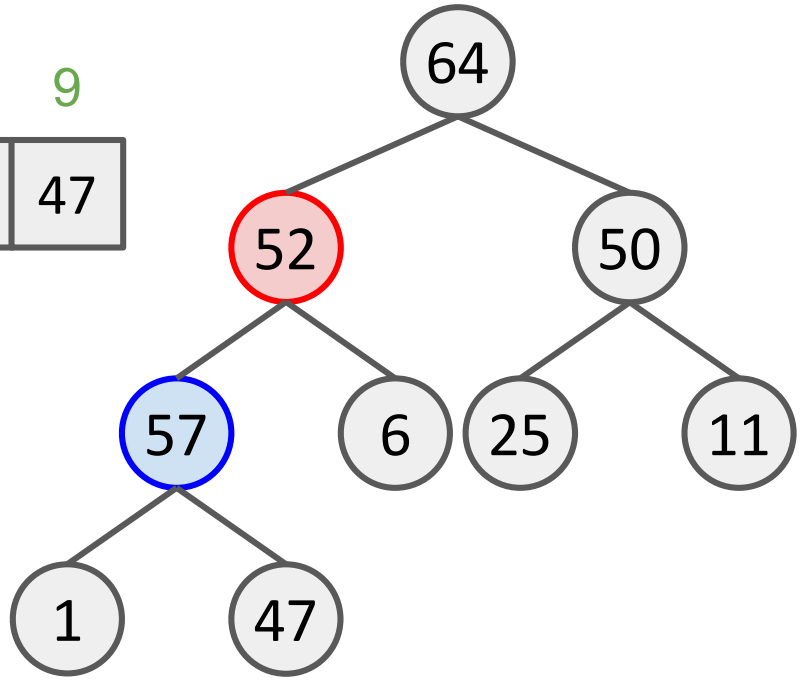


Insert 57 at last position in max-heap

57 is greater than its parent 47, so we will bubble 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

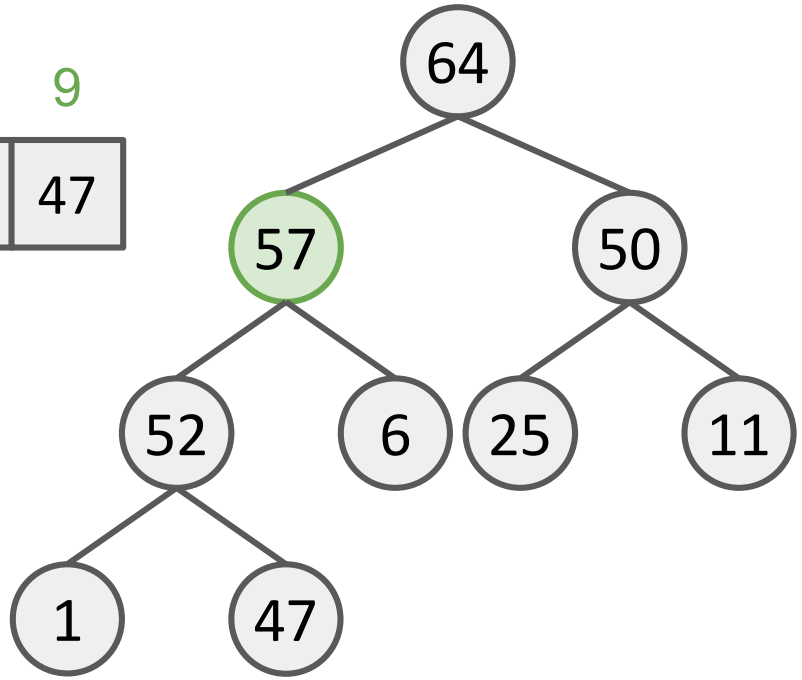


47 is now in its rightful spot.

57 is still greater than its new parent 52, so we will bubble 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



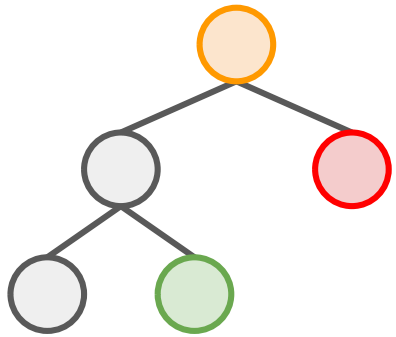
52 is now in its rightful spot.

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

`ExtractMax()` (VisuAlgo slide)

Steps

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

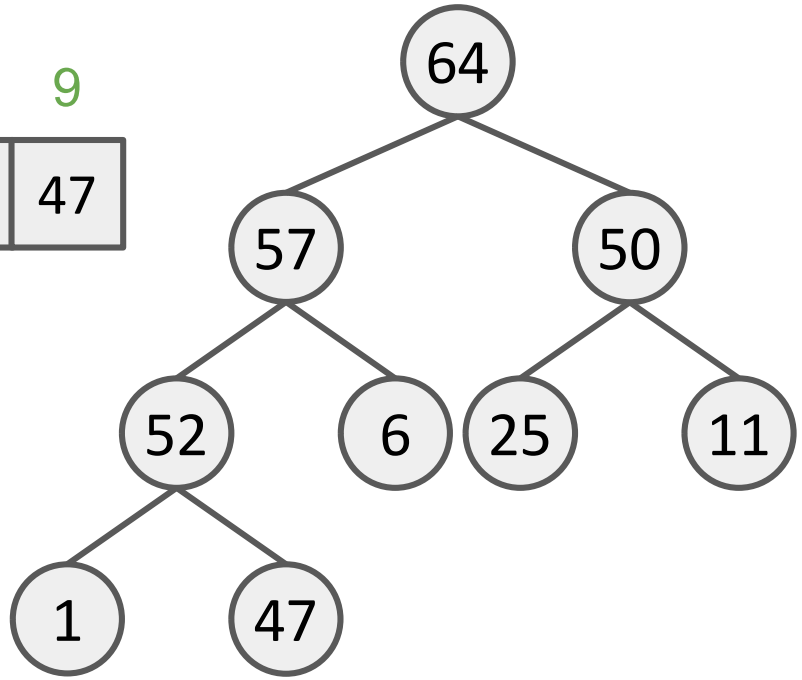


*: Note that this is **not necessarily** the **rightmost leaf of the tree**!

Complexity: *O(log N)*

`ExtractMax()` (VisuAlgo slide)

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

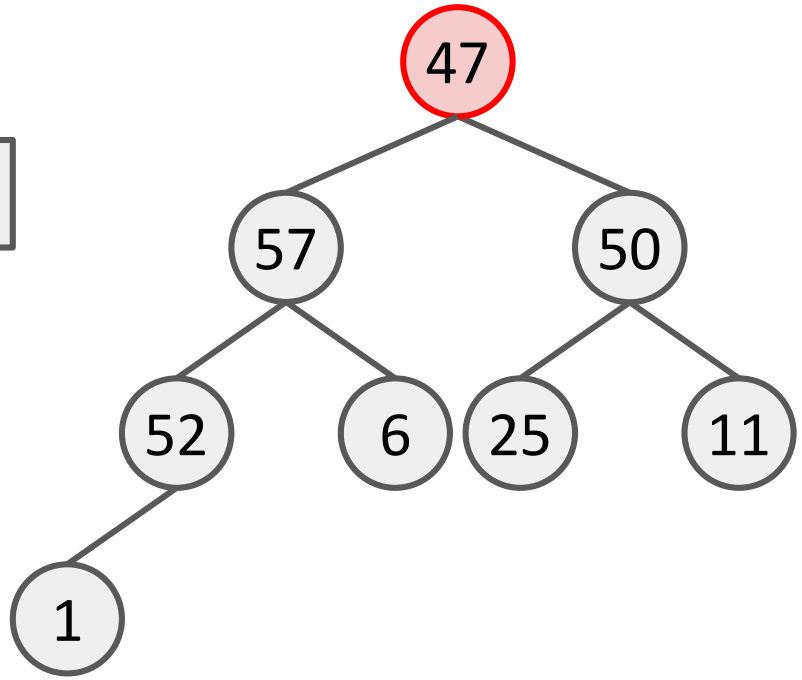


Say for instance, we call `ExtractMax()` on this max-heap.

We'll remove 64 and replace it with 47

`ExtractMax()` (VisuAlgo slide)

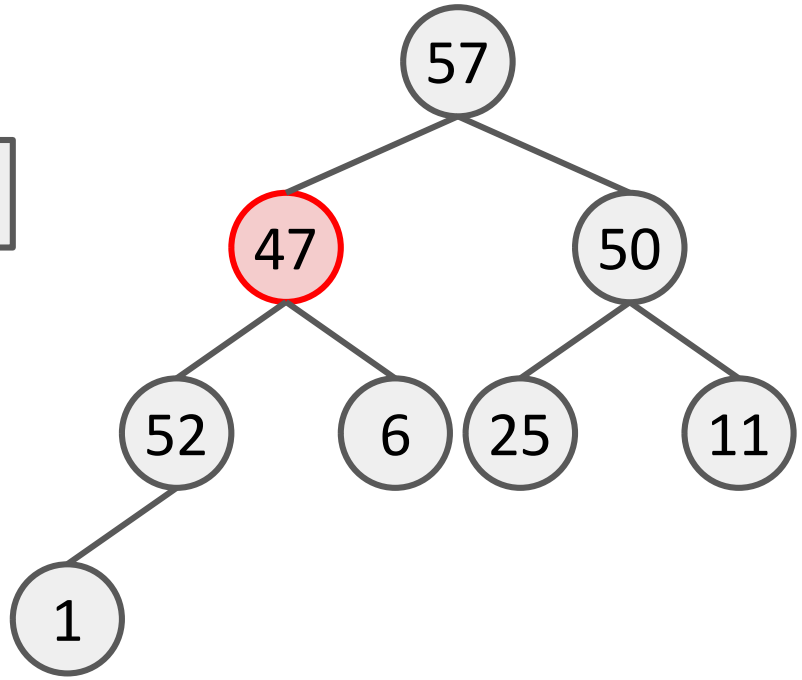
index	1	2	3	4	5	6	7	8
value	47	57	50	52	6	25	11	1



47 is lower than both its children, so we will bubble 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

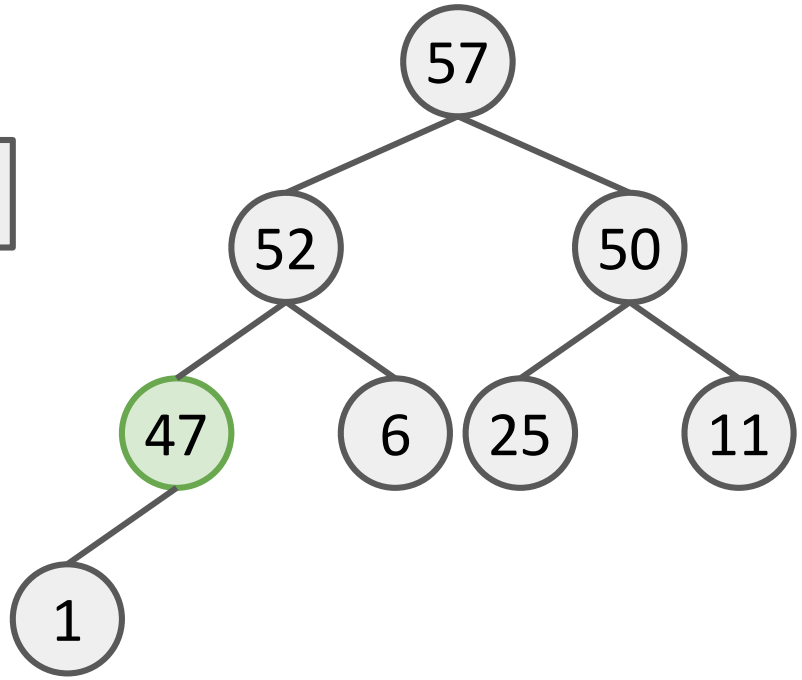


57 is now in its rightful spot.

47 is lower than one of its new children, so we will bubble 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 vertex v_i from v_N down to v_1:
    heapify(v_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

For a heap which is a full binary tree (worst case) with 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}.

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

Number of vertices at height h

See slide 29

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

`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

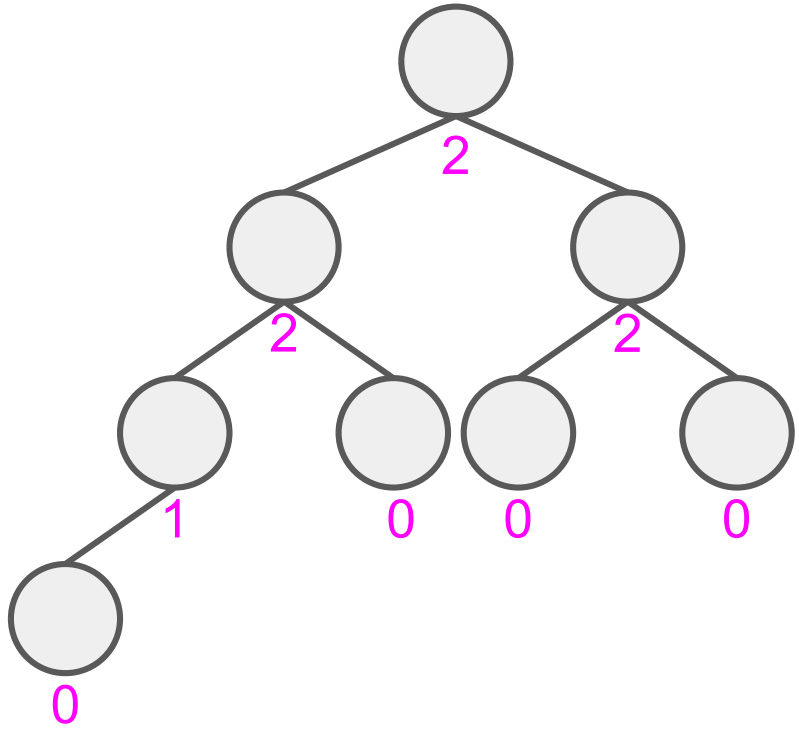
What is the **minimum** and **maximum** number of comparisons between Binary Heap elements required to construct a Binary (Max) Heap of arbitrary n elements using the *O(N)* `Create(A)`?

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

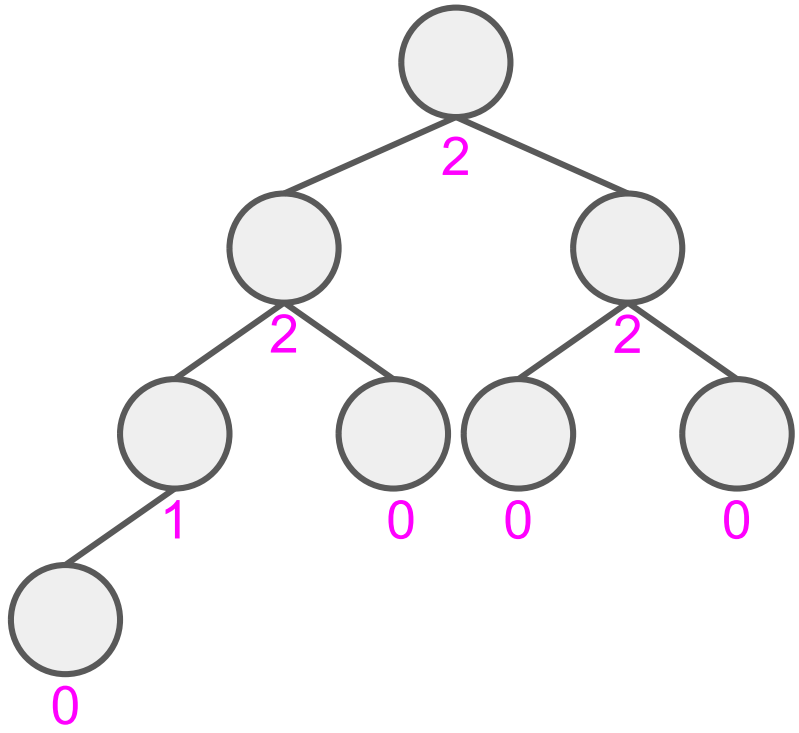
Take for instance a heap with 8 vertices.

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



Question 2

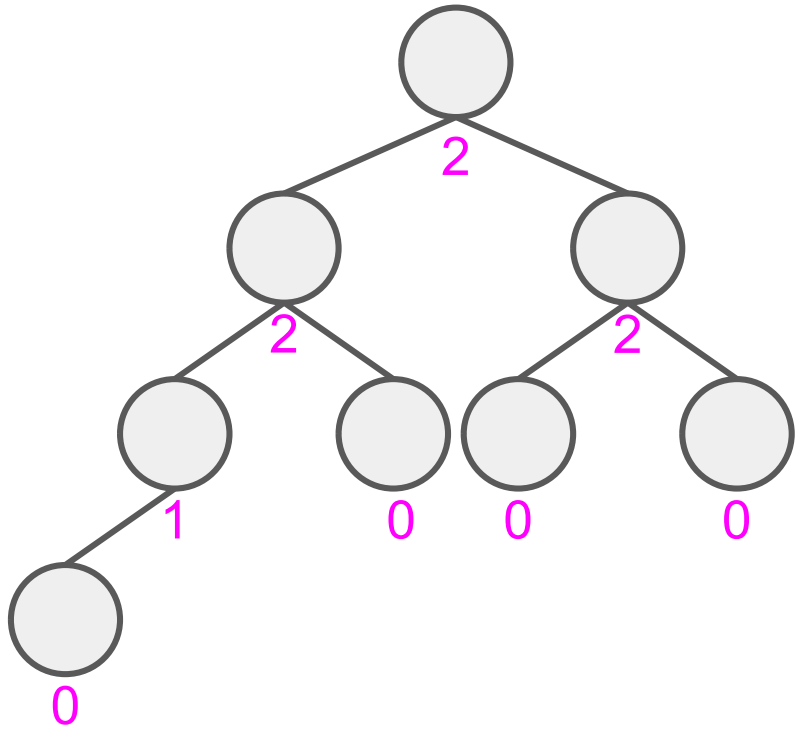
We incur minimum number of total comparisons if the array is already in heap order. So each vertex just need to conduct its own comparisons once and determine that no further heapify is required.



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 $1 + (2 + 1) + 2 + (2 + (2 + 1)) = 11$

`HeapSort(A)`

Steps to sort an array of values in descending order

1. Create max-heap using *O(N)* `Create(A)`
2. Repeatedly call `ExtractMax()` until max-heap is empty (*N* times). *O(N log N)* time
3. The sequence of extracted values will be in descending order

Time and space complexity: *O(N log N)*

Note: For sorting ascending order, repeat the same steps as above but use a min-heap

Question 3

Problem statement

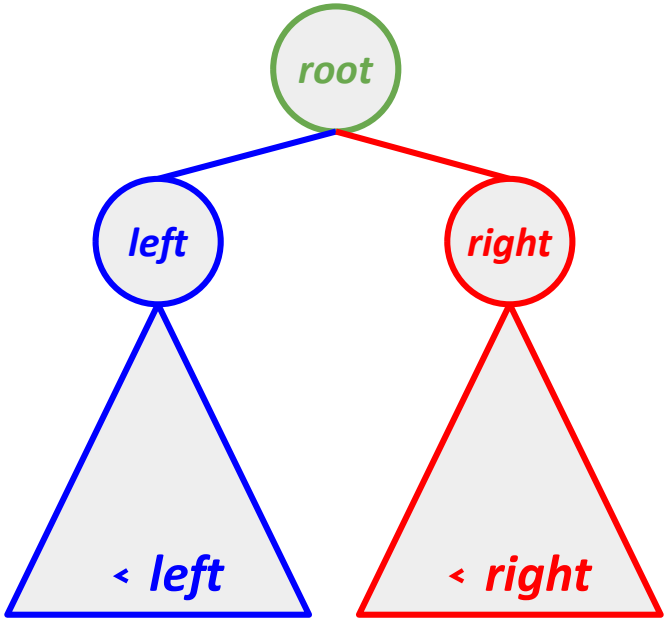
Claim: The second largest element in a max heap with more than two elements is always one of the children of the root. Is this true? If yes, show a simple proof. Otherwise, show a counter example.

For simplicity please assume that all values are unique!

Note that this kind of (simple) proof may appear in future CS2040/C written tests, so please refresh your CS1231 (if you have taken that module) or just concentrate on how the tutor will answer this kind of question.

Proof by contradiction

1. Assume that 2nd largest is neither ***left*** nor ***right***
2. Realize both ***left*** and ***right*** must be lesser than 2nd largest
3. That means 2nd 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 2nd largest **must be** either ***left*** or ***right***!



Variations

- Think about variations of the problem
- 3rd largest element
- Smallest element
- 2nd smallest element
- Etc.

Question 4

Problem statement

Give an algorithm to count all vertices that have value `>x` in a max-heap of size n.

Your algorithm must run in $O(k)$ time where 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 n but rather the output size k :O...

Note that this question has also been integrated in VisuAlgo Online Quiz, so it may appear in future Online Quizzes :)

Observation

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

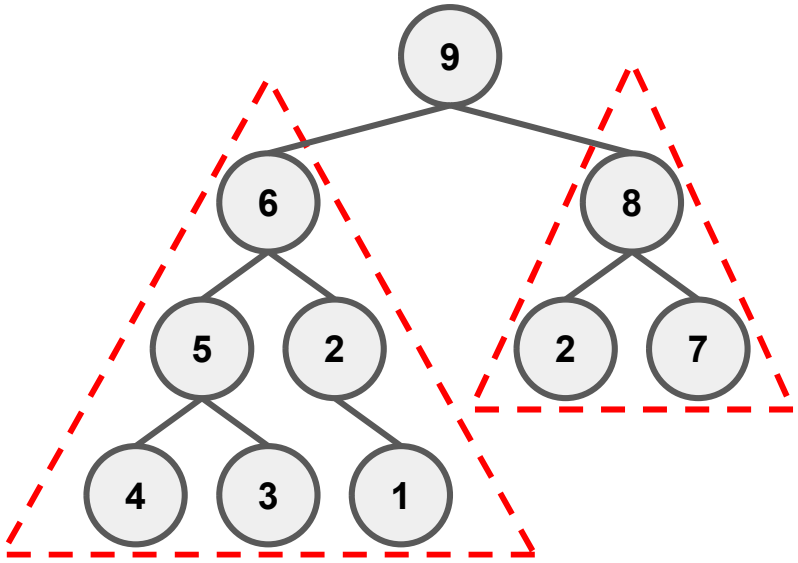
More specifically, a binary max-heap is a complete binary tree where the root vertex has value greater than both its children and each of its children is also a binary heap.

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

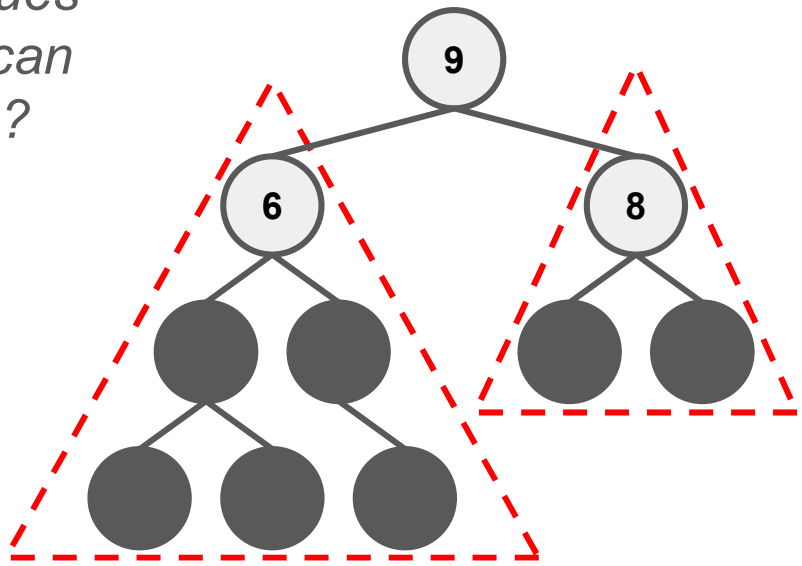
Recall for a binary heap, the root vertex has the greatest value in the entire heap.

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`.

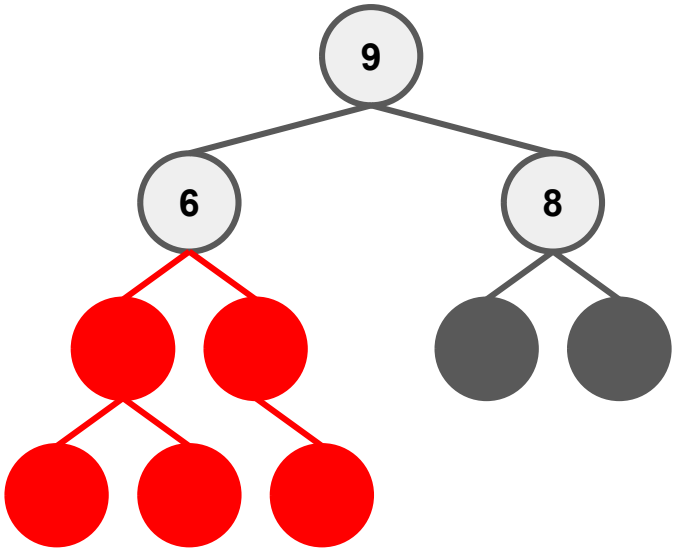
If I hide all the values except the roots, can you infer anything?



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) {
        cout << v.value << endl;              // Output
        findVerticesBiggerThanX(v.left, x);  // Recurse left
        findVerticesBiggerThanX(v.right, x); // Recurse right
    }
}
```

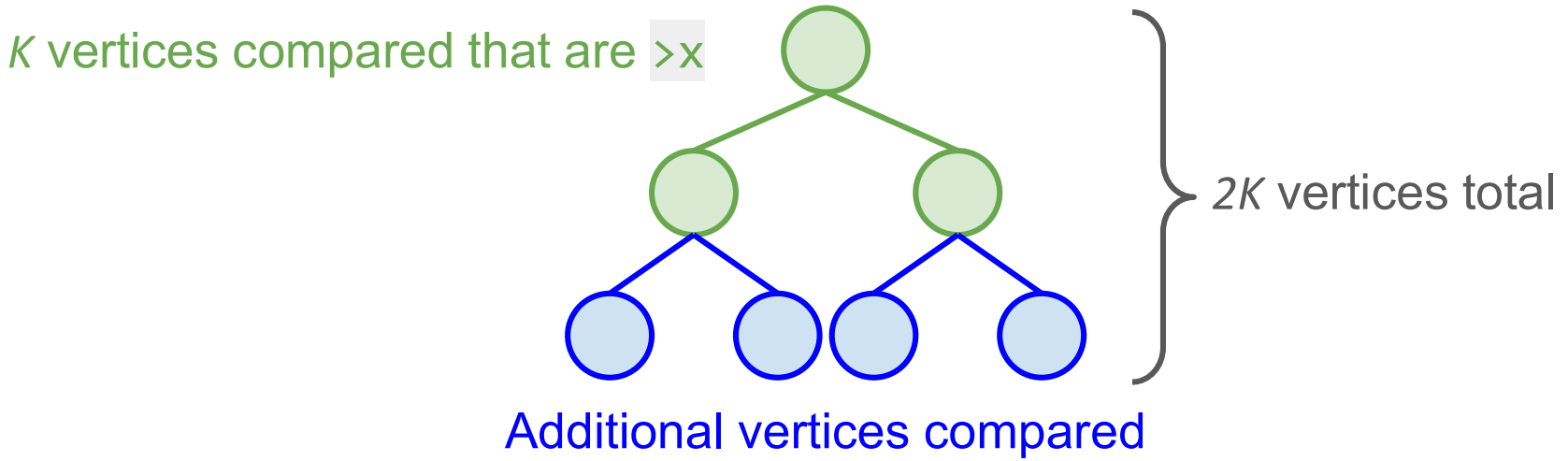
Test yourself!

If the answer has *K* outputs, what is the total number comparisons done?

Test yourself!

If the answer has *K* outputs, what is the total number comparisons done?

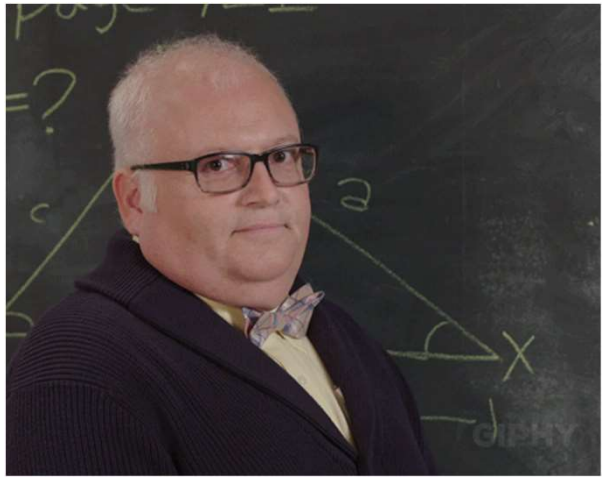
Answer: *2K* (i.e. 2 comparisons for every vertex with value >x)



Time complexity

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

$$O(K + 2K) = O(3K) = O(K)$$



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:

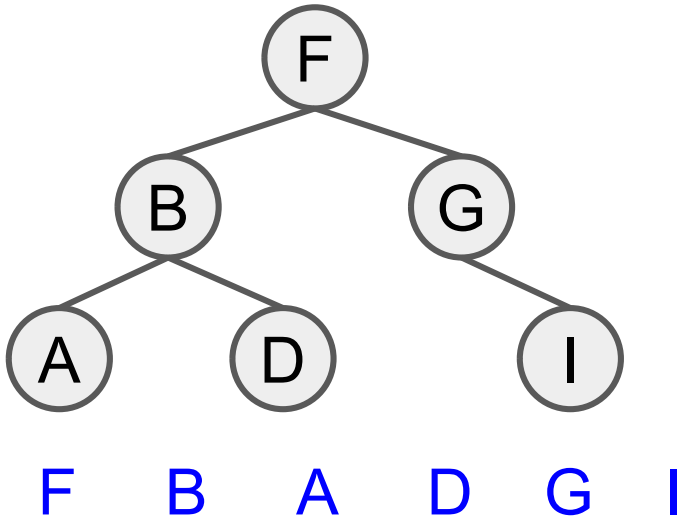
1. Pre-order traversal
2. In-order traversal
3. Post-order traversal

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) {
        cout << v.value << " ";
        pre_order(v.left);
        pre_order(v.right);
    }
}
```

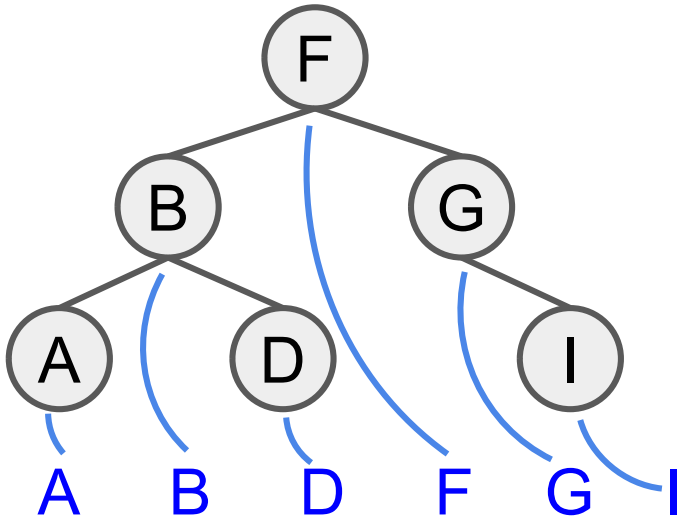


Print order in blue

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);
        cout << v.value << " ";
        in_order(v.right);
    }
}
```



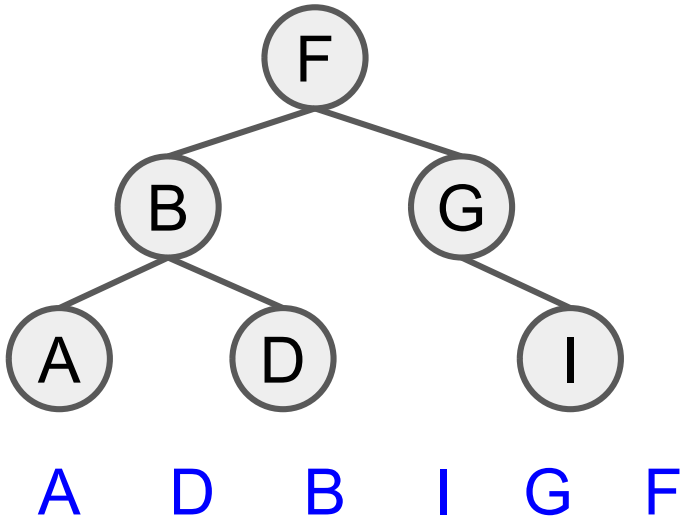
Print order in blue

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

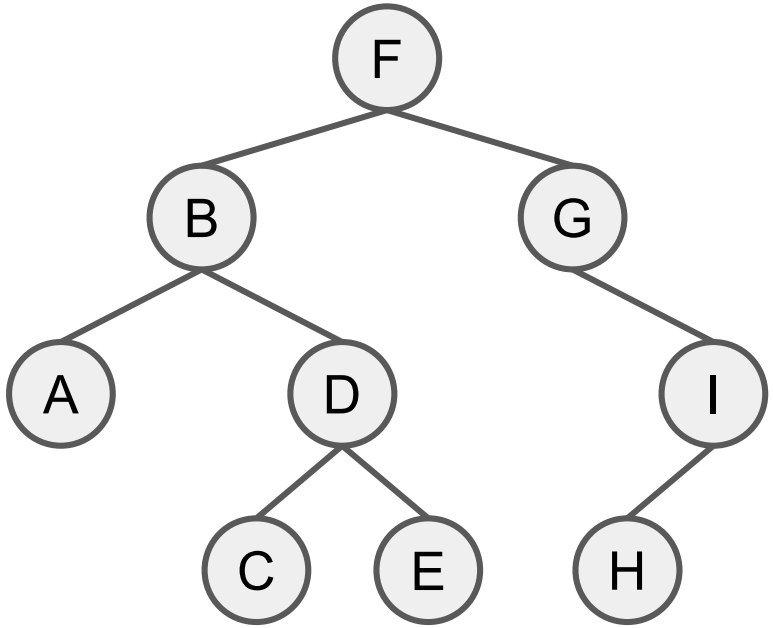
```
void post_order(vertex v) {
    if (v) {
        post_order(v.left);
        post_order(v.right);
        cout << v.value << " ";
    }
}
```



Print order in blue

Test yourself!

For your own practice, try out the 3 types of traversals on this tree!



Question 5

Did you know?

There are two interesting features of Binary Heap data structure that are not available in C++ STL `priority_queue` and Java PriorityQueue yet:

1. `Increase/Decrease/UpdateKey(old_v, new_v)`
2. `DeleteKey(v)`

These two operations are not yet included in VisuAlgo (the hidden slide https://visualgo.net/en/heap?slide=3-1).

Question 5 a)

Given Steven's Binary Heap Demo code (that is a Binary Max Heap), what should we modify/add so that we can implement `DecreaseKey(old_v, new_lower_v)`?

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?
- Will it affect Complete Binary Tree (compact array) property? If so how do we handle?
- Will it affect heap property? If so how do we handle?
- What's the time complexity?

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, “bubble down” *O(log N)*
- What’s the time complexity?
  - *O(log N)*

Question 5 b)

Given Steven's Binary Heap Demo code (that is a Binary Max Heap), what should we modify/add so that we can implement `DeleteKey(v)` where `v` is not necessarily the max element?

Solution 1: Building upon `UpdateKey`

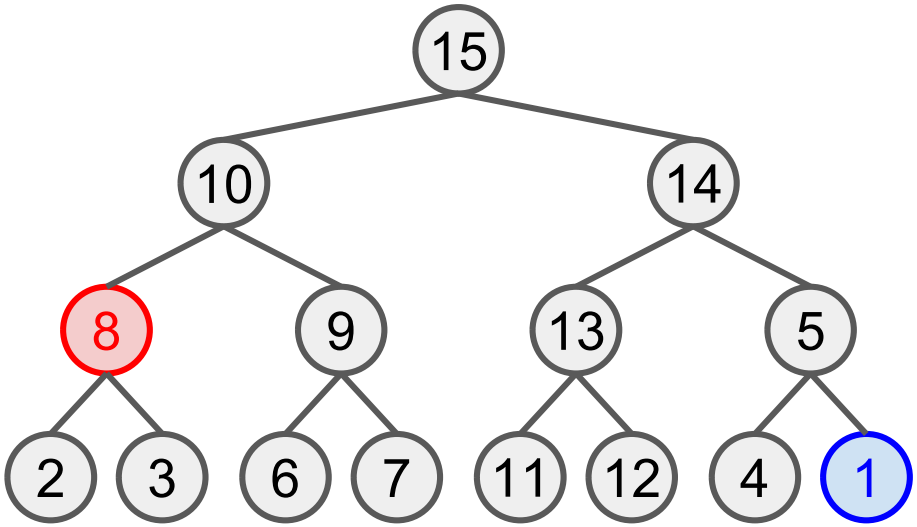
1. `UpdateKey(v, +∞)`
2. `ExtractMax()`

Solution 2: Generalising `ExtractMax()`

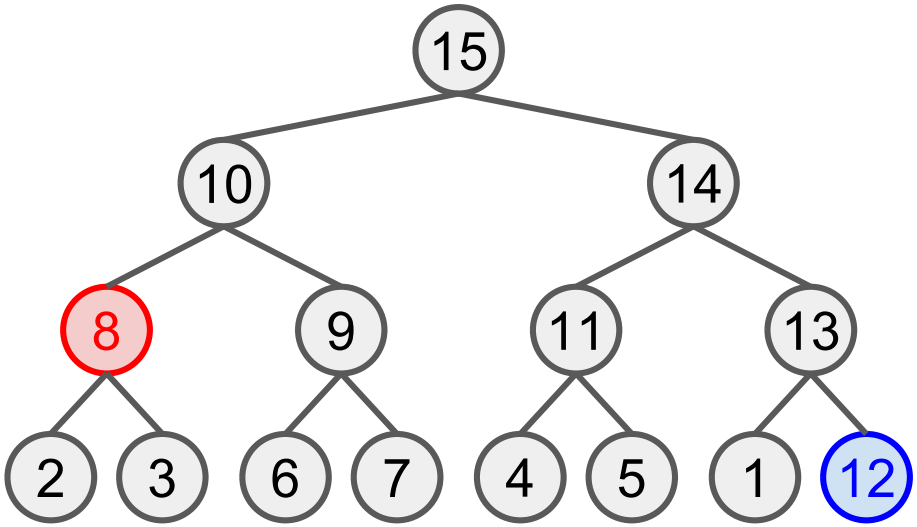
1. Find index of `i` vertex with value `v` in compact array
2. Move value at index `N` into value at index `i`
3. Then `shift_down` or `shift_up`?

Solution 2: Generalising `ExtractMax()`

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



This will need `shift_down`



This will need `shift_up`

Extra stuff—A primer for lazy algorithms

We can actually use a “lazy approach” for `deleteKey(v)` `increaseKey(old_v, new_v)` and `decreaseKey(old_v, new_v)`!

Lazy `DeleteKey(v)`

1. Find vertex. Can be done in *O(1)* using a Hash Table
2. Flag vertex as “invalid”
3. Done! :O

Complexity: *O(1)*

We will discard “invalid” vertices as we encounter them during during `ExtractMax()` calls! If we’re lucky to not end up encountering invalidated vertices in our overall routine, we circumvent the *O(log N)* deletion time.

Lazy `UpdateKey(old_v, new_v)`

1. Find vertex. Can be done in *O(1)* using a Hash Table
2. Flag vertex as “invalid”
3. `insert(new_v)`
4. Done!

Complexity: *O(log N)*

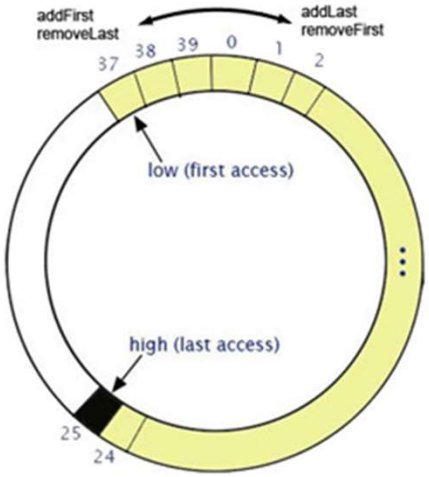
Same as lazy `DeleteKey`, we will discard “invalid” vertices as we encounter them during during `ExtractMax()` calls

Questions?

/janeeyre

What really is a deque?

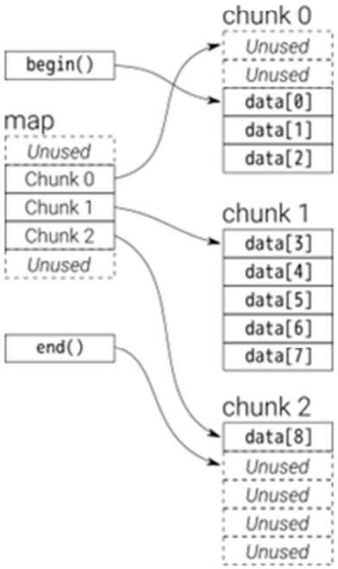
Implementations can vary.



Circular buffer

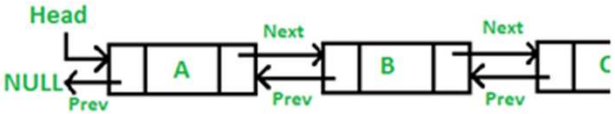
https://www.cs.wcupa.edu/rkline/ds/deque-stack-algorithms.html

C++ implementation generally uses this to guarantee O(1) random access, O(n) insertion



Vector of fixed-sized vectors

https://stackoverflow.com/a/6292437



Doubly linked list

https://stackoverflow.com/a/6292437