CS2040C Tut 5

Introduction to Trees

Binary Heaps

PS3

How to proof?

Proof/Disproof Approaches

Trees

Approaches to Proofs

- **Direct** proof
  - State facts (premises)
  - Show that they logically lead to a conclusion

- Proof by **contradiction**
  - Assume the *reverse* is true
  - Show that *with the assumption*, it follows logically to contradict some known facts

Approaches to Proofs

- Proof by **construction**
  - State an example
    - "There exists a testcase that … "
    - Just give the testcase
  - Describe an approach such that the statement will true
    - "For all X, there exist a testcase that …"
    - Just give a generic way of constructing the testcase based on X

Approaches to Proofs

- Proof by **division** of cases
  - List the cases and proof each case separately
    - Eg: When N is even: … , when N is odd: …
- Proof the **contrapositive** is true
  - Proof that <u>if P is true, then Q is true</u>
  - We can proof that <u>if Q is false, then P is false</u>
  - Tutor *thinks* CS2040C qns will be *direct*
- Proof by mathematical **induction**
  - Tutor *thinks* CS2040C won't require this

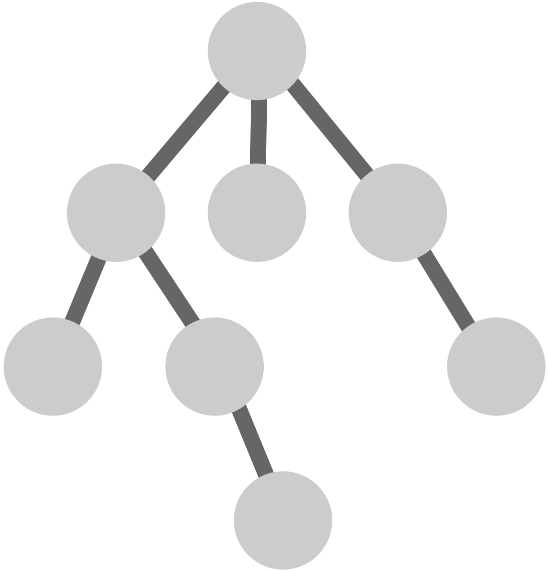
Approaches to Disprove

- Proof the **inverse** is true
  - “Direct disprove” is to proof the converse via contradiction

- Construct a **counterexample**
  - “All even numbers are not prime.” $\rightarrow$ 2

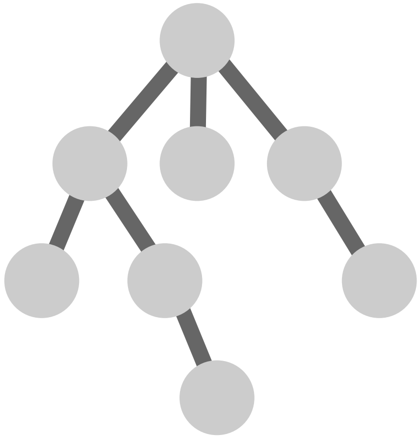
Trees

- A specific type of special graph
  - Acyclic
  - Undirected
  - Connected



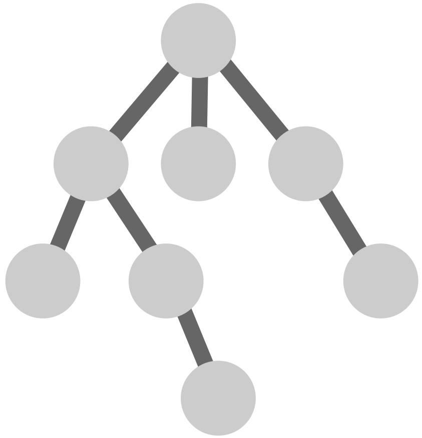
Tree Properties

An undirected tree of **N** vertices will **always** have **N-1** edges.



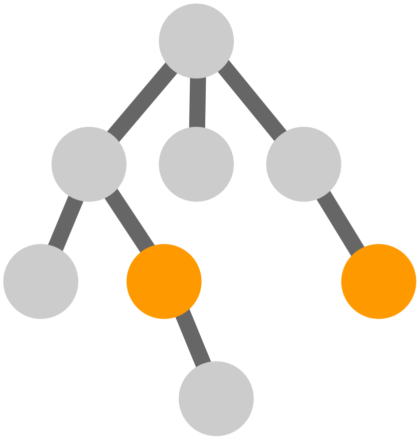
Tree Properties

Conversely, a connected undirected graph of **N** vertices and **N-1** edges is *always* a tree.



Q1

For *all* pairs of two distinct nodes in a tree (**u**, **v**), there is only **one unique path** to between **u** to **v**.



Why?

Proof by Contradiction

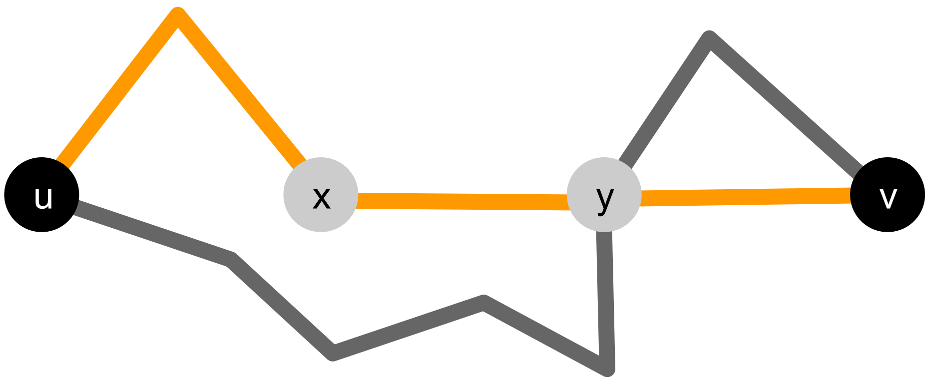
For *all* pairs of two nodes (**u**, **v**) in a tree , there is only **one unique path** to get from **u** to **v**.

*Assume that it is **false**.*

(We can find a pair of two nodes (**a**, **b**) in a tree such that there are *more than one* unique path to get from **a** to **b**)

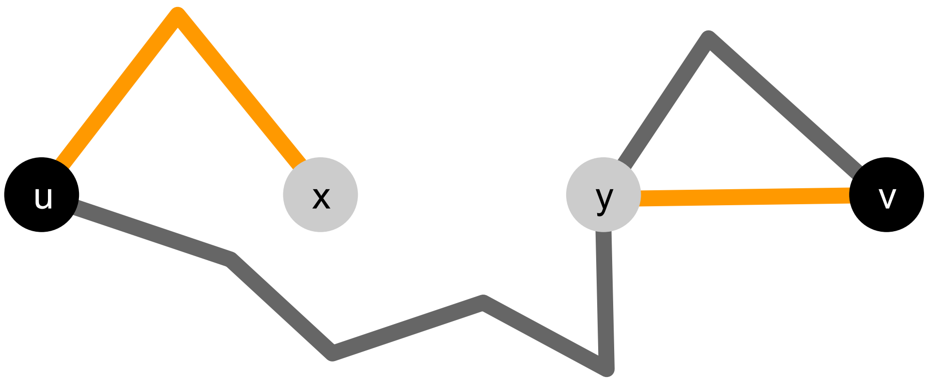
Proof by Contradiction

- We let the two paths be $\mathbf{p_1}$ and $\mathbf{p_2}$.
- As $\mathbf{p_1}$ and $\mathbf{p_2}$ are distinct, there is at least one edge $\mathbf{e}$: ($\mathbf{x}$, $\mathbf{y}$) that is in $\mathbf{p_1}$ but not $\mathbf{p_2}$



Proof by Contradiction

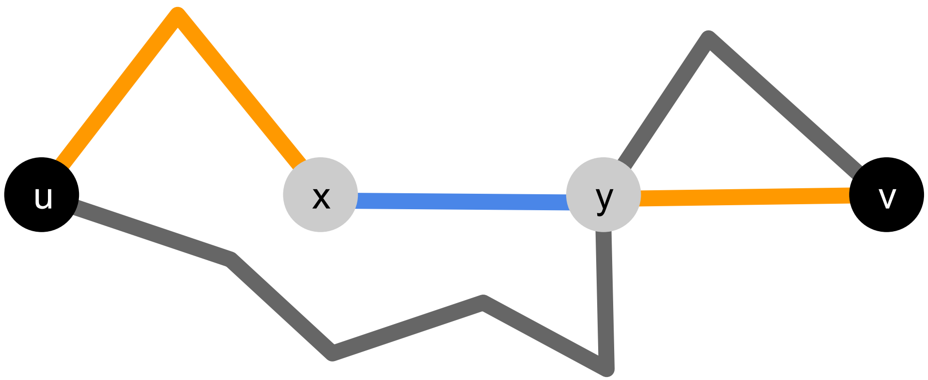
- If we remove edge **e**,
  - We can still get from $\mathbf{x} \rightarrow \mathbf{y}$ via $\mathbf{x} \rightarrow \mathbf{u} \rightarrow \mathbf{v} \rightarrow \mathbf{y}$
- We denote this path as $\mathbf{p_3}$.



Proof by Contradiction

- $\mathbf{p_3}$ with the **e** will hence form a cycle between **x** and **y**

  i.e. $\mathbf{x} \rightarrow \mathbf{u} \rightarrow \mathbf{v} \rightarrow \mathbf{y} \rightarrow \mathbf{x}$



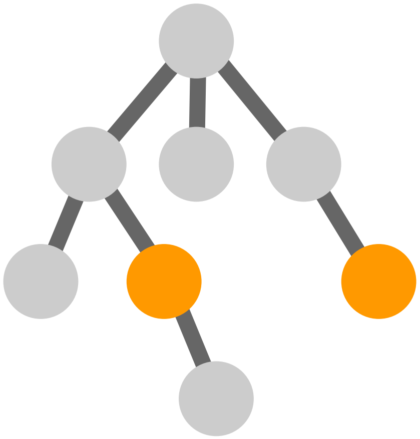
Proof by Contradiction

- Thus, the graph will contain cycles.
- However, a tree is acyclic by definition.
  - Contradiction

- **Conclusion**: there is only one unique path between two distinct nodes of a tree

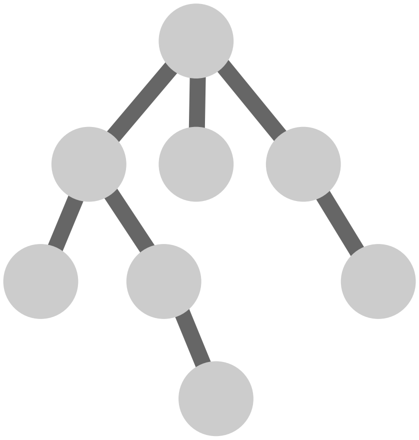
Tree Properties

For *all* pairs of two vertices in a tree (**u**, **v**), there is only **one unique path** to between **u** to **v**.



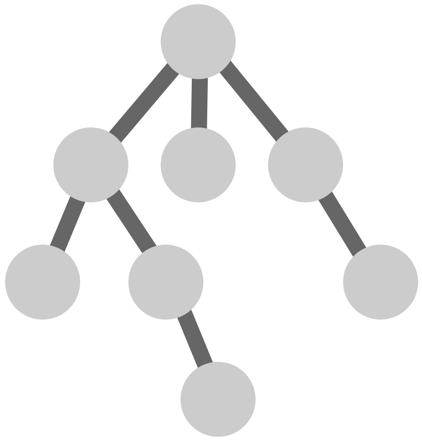
Tree Properties

A undirected graph with **exactly one** unique path between every pair of 2 vertices is a **tree**.



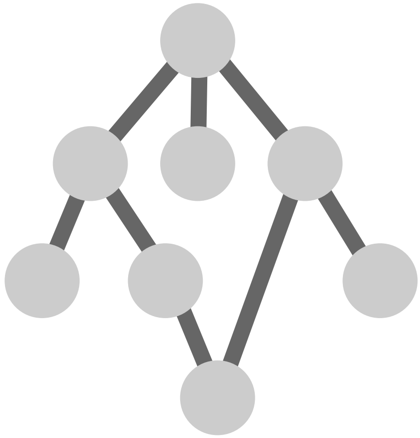
Phrasing of trees in questions (undirected)

1. “Connected acyclic graph”
2. “Connected graph with **N** vertices and **N-1** edges”
3. “There is exactly one path between every two vertices in the graph.”



Tree?

Is this a tree?

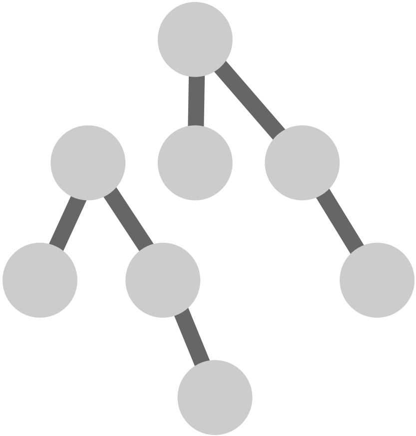


Tree?

Is this a tree?

Tree?

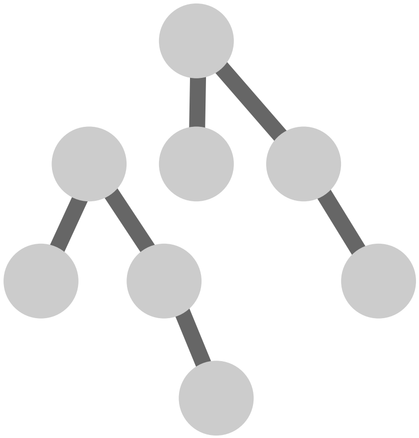
Is this a tree?



Forests

A graph composed of **only** trees is called a forest.

Usually, we will then consider each tree separately in our algorithms.



Q2

“In a complete binary tree with **N** vertices, the number of leaf nodes is *strictly* less than **N/2**.

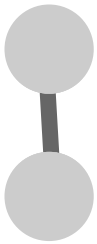
True or False?

Q2

Consider the complete binary tree of **N** = 2:

- How many leaf nodes?

- What is N/2?



Disprove by Counterexample

Statement claims it is true *for **all** cases*.

Just state/draw/construct a case that violates the statement.

Binary Heaps

Basic Properties

Advanced Stuff

Binary Heap

1. Insert(element **v**)
2. ExtractMax()
3. Create(array **A**)
   a. O(N log N) version
   b. O(N) version

Binary Heap

Main Idea

- Each vertex has up to 2 children
  - Binary means 2
- Each vertex has a value **larger*** than its children
  - All our operations are to maintain this property
  - Also known as the **heap property**

Binary Heap

Height of the tree is O(log **N**)

- Top = Max. Element
- Insert
  - “Bubble sort” upwards -- O(log **N**) height
- Extract Max
  - “Bubble sort” downwards -- O(log **N**) height

Q3

Give an **O(K)** algorithm to find all vertices bigger than some value **x** in a binary heap of size **N** and **K** is the number of vertices in the output.

??

O(K)? ... i not OK with this

Q3

Give an **O(K)** algorithm to find all vertices bigger than some value **x** in a binary heap of size **N** and **K** is the number of vertices in the output.

O(**K**) : linearly proportional to output size

Q3

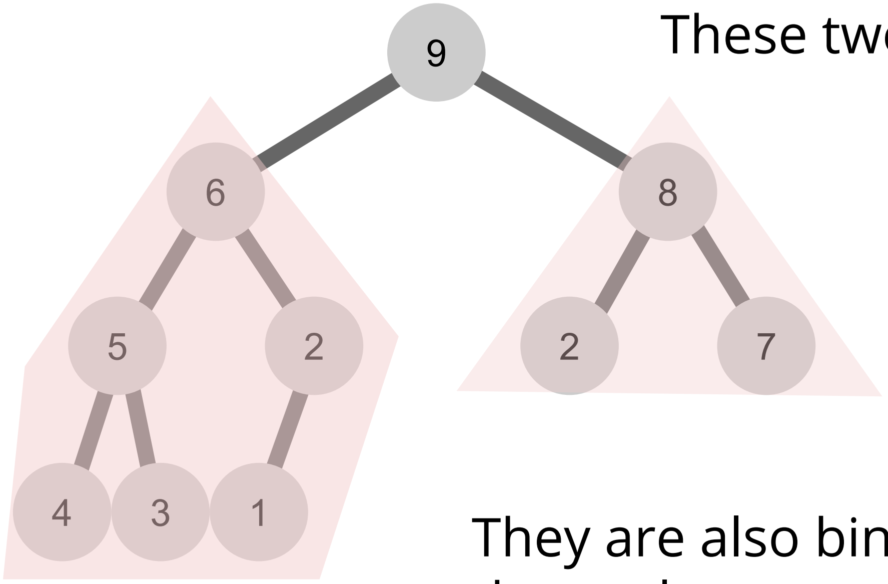
Observation

A binary heap “contains more binary heaps”.

Formally,

Any subtree of a rooted binary heap is also a binary heap.

Q3



These two are subtrees

They are also binary heaps by themselves

Q3

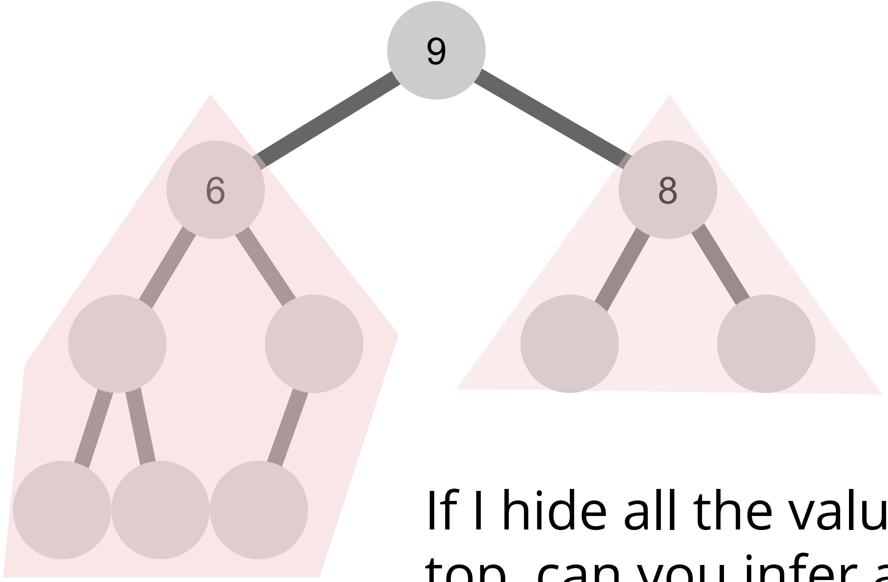
Approach

Remember for a binary heap, the largest element is always the root.

All elements in the binary heap, have **values ≤ the root**.

Q3

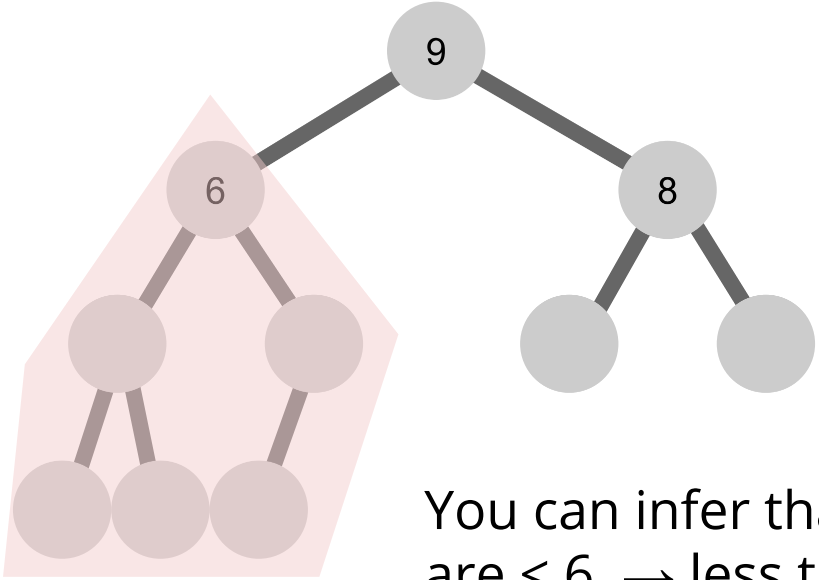
Let's say I want to find all vertices with value ≥ 7.



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

Q3

Lets say I want to find all vertices with value ≥ 7.



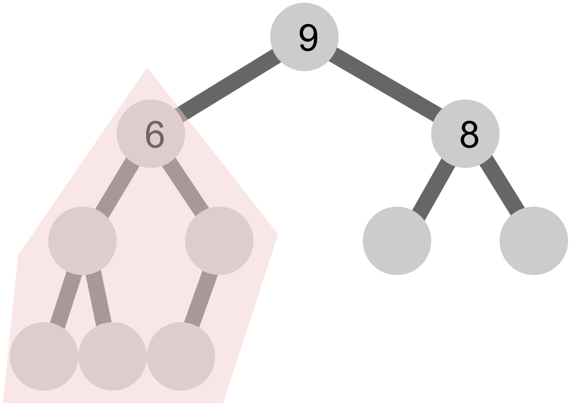
You can infer that all these nodes are ≤ 6 → less than what we want!

Q3

We can design our algorithm such that:

If the value of the vertex **≤ X**,

then we need not continue to search its children.



Q3

We can implement this recursively:

Algorithm 1 findVerticesBiggerThanX(vertex, x)

```
if (vertex.key > x) then
    output(vertex.key)
    findVerticesBiggerThanX(vertex.left, x)
    findVerticesBiggerThanX(vertex.right, x)
end if
```

Q3

Time Complexity

If the answer has **K** vertices,

What is the maximum number of vertices that the algorithm checked that are < **X**?

2K (i.e. the 2 childrens of vertices with ≥ **X**)

Q3

Time Complexity

In total: $O(\mathbf{2K} + \mathbf{K}) = O(3\mathbf{K}) = O(\mathbf{K})$

Q3

DFS and Tree Traversals

Actually, this recursive algorithm is what we call a depth first search (DFS).

It will iterate to the *deepest* vertex it can find, before backtracking.

Q3

DFS and Tree Traversals

This is also an example of (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!)

(Optional, self-read)

Tree Traversals [Credits: SG IOI Training 2017]

Pre-order traversal

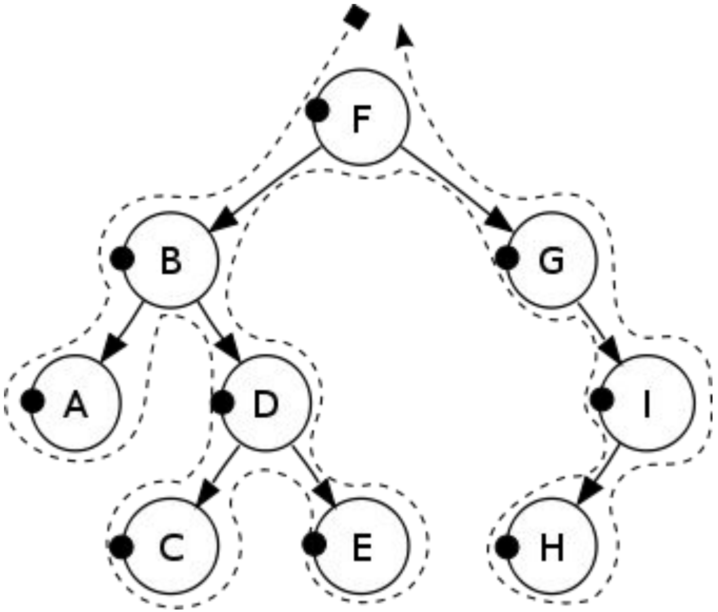
- Perform operations on the vertex only when first encountered.

Post-order traversal

- Perform operations on the vertex only when on the last encounter.

(Optional, self-read)

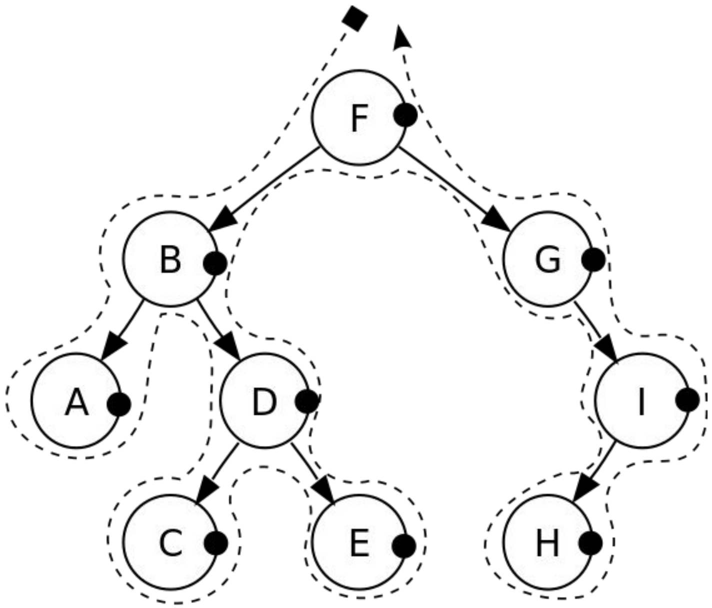
Pre-order Traversal [Credits: SG IOI Training 2017, Wikipedia]



F B A D C E G I H

(Optional, self-read)

Post-order Traversal [Credits: SG IOI Training 2017, Wikipedia]



A C E D B H I G F

(Optional, self-read)

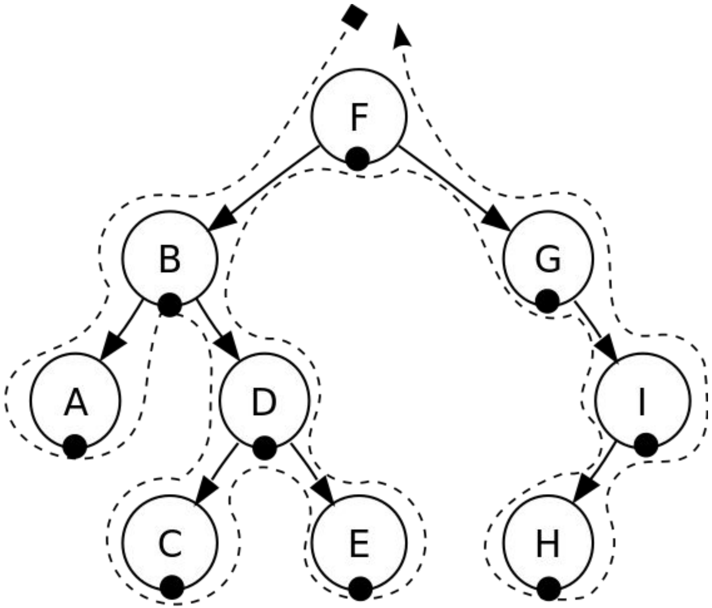
Tree Traversals [Credits: SG IOI Training 2017]

In-order traversal

- Perform operations on the vertex after completing the left subtree, but before commencing the right subtree.

(Optional, self-read)

In-order Traversal [Credits: SG IOI Training 2017, Wikipedia]



A B C D E F G H I

Q4

The **second** *largest* element in the binary *max* heap is always one of the children of the root.

(You may assume that all elements are distinct and the heap has more than 2 elements)

Is this true? If true, proof.

If not, show a counter-example.

Q4

Proof by Contradiction

Let **X** denote the second largest element.

Assume <u>**X** is *not* one of the childrens of the root</u>:

Then **X** must either be

1. The root
2. Have a parent that is not the root

Q4

1. **X** cannot be the root as the root is the largest element. [By definition of binary max heap]

2. **X** cannot have a parent that is not the root. Otherwise, **X** will be larger than its parent and that is a contradiction. [By definition of binary max heap]

By contradiction, the statement is true.

Variants of Q4

The **third** *largest* element in the binary *max* heap is always one of the children of the root.

(You may assume that all elements are distinct and the heap has more than 3 elements)

Proof/Disprove.

Variants of Q4

The **third** *largest* element in the binary *max* heap is always one of the children of the root.

(You may assume that all elements are distinct and the heap has more than 3 elements)

Disprove (by counterexample)

Variants of Q4

The **second** *smallest* element in the binary *max heap* will <u>*always* have no children</u>.

(You may assume that all elements are distinct and the heap has more than 2 elements)

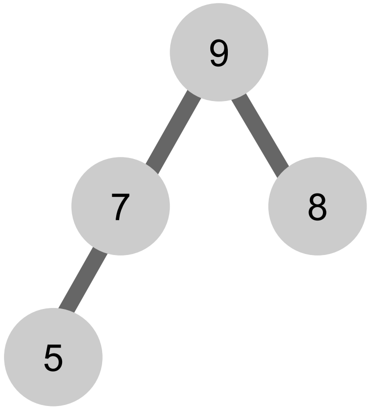
Proof/Disprove.

Variants of Q4

The **second** *smallest* element in the binary *max heap* will *always* have no children.

(You may assume that all elements are distinct and the heap has more than 2 elements)

Disprove (by counterexample)



PS3

Scheduling Deliveries

PS3A

Approach

N is *very small*: Brute force.

If you get *Wrong Answer* even if you write a very naive brute force...

High chance you *misinterpreted* the question.

PS3C

Approach

Let’s say I modify the task as such:

1. Instead of woman names, we label them with an **integer ID**
2. The **integer ID** is their order of arrival (1, 2, 3... etc)

PS3C

We will need a data structure that can:

- Get/Extract Max Dilation
- Update (actually just increase) Dilation
- Delete *any* item

PS3C

- Get/Extract Max Dilation
  - Easily handled by priority queue
- Update (actually just increase) Dilation
  - Not in a standard priority queue
- Delete *any* item
  - Not in a standard priority queue

PS3C

Q5

How do I remove *any* element from a binary heap?

Swap the *last* element to the index **i** to be removed.

Execute **both** *shiftUp* and *shiftDown*. Why?

PS3C

Q5 + Q6

How do I remove *any* element from a binary heap?

Swap the *last* element to the index **i** to be removed.

Execute **both** *shiftUp* and *shiftDown*. Why?

How to find that element's index **i** in binary heap in O(1)?

- *Use Hash Table?? (but we haven't learn it -> Week07 :O)*

PS3C

Q6

So... how do I remove *any* element from a STL priority queue?

You don't.

Be lazy. Do it later.

Lazy Deletion

What if we don’t remove it instead?

It will only affect the *pq.top()*, when *an element that is supposed to be removed* is the top element.

Otherwise, *pq.top()* will be the correct value.

PS3C

Lazy Deletion

If *pq.top()* is *supposed* to be removed previously, it will affect our result.

But it is now the top of our PQ! → we can use ExtractMax() to remove it.

Do so iteratively as the next element might also be *supposed* to be removed.

PS3C

Lazy Deletion

We need a way to **flag**/check if an element *should* be removed in lazy deletion.

For our case, it just suffices to check whether the woman (identified by integer ID) at *pq.top()* has already given birth.

Why? How would you code this?

PS3C

Lazy Deletion

Does this work for all use cases of priority queue?

How can we check if an item is *supposed* to be deleted, in the general case?

PS3C/D

Approach

If you understand how to solve with the modified task (with integer IDs):

Can you create these *modifications* by yourself?

:)

#	#	#
#	#	#
#	#	#

PS3C/D

Some (major) hints to get people on the right track:

- (Maybe) more than one type of data structure
  - Each handling different parts of the question
- You can use STL data structures up to PS3C
  - Use them first, then substitute with your own for PS3D

Questions?