Graphs, Heaps, Midterm 2 Review

Exam Prep 09

Announcements

- Week 8 Survey due 11:59 PM Monday 10/16
- Homework 3 due Monday 10/16 (non-extendable!)
- Midterm 2 is Thursday, 10/19 7-9 pm
- Project 2B Checkpoint due Monday 10/23

Content Review

Trees, Revisited (and Formally Defined)

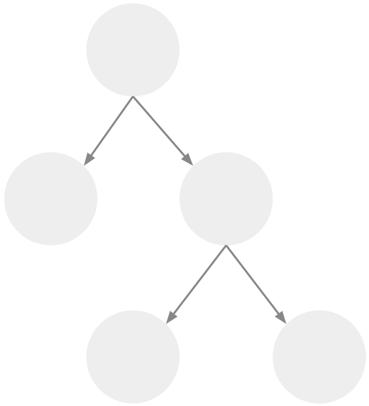
Trees are structures that follow a few basic rules:

1. If there are N nodes, there are N-1 edges
2. There is exactly 1 path from root to every other node
3. The above two rules means that trees are fully connected and contain no cycles

A parent node points towards its child.

The root of a tree is a node with no parent nodes.

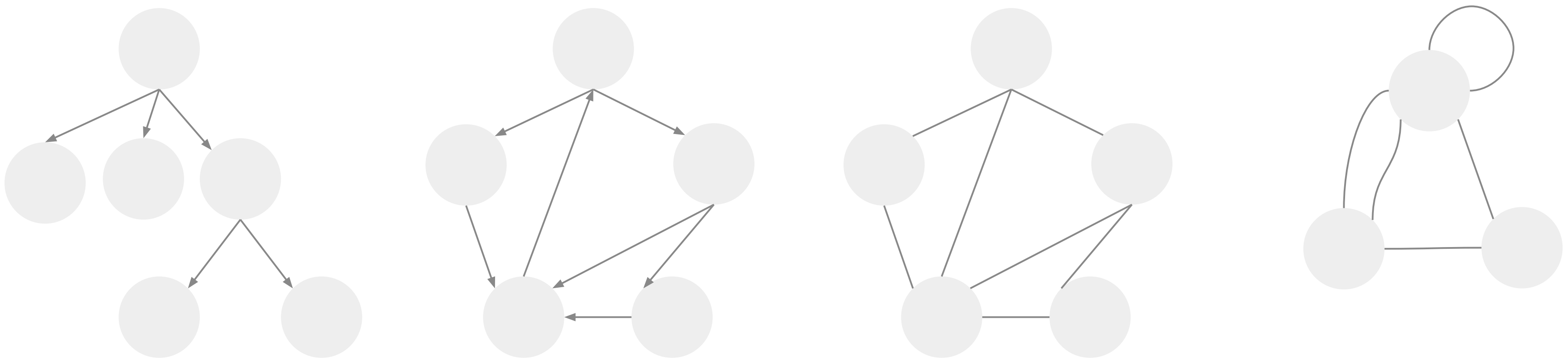
A leaf of a tree is a node with no child nodes.



Graphs

Trees are a specific kind of graph, which is more generally defined as below:

1. Graphs allow cycles
2. Simple graphs don't allow parallel edges (2 or more edges connecting the same two nodes) or self edges (an edge from a vertex to itself)
3. Graphs may be directed or undirected (arrows vs. no arrows on edges)

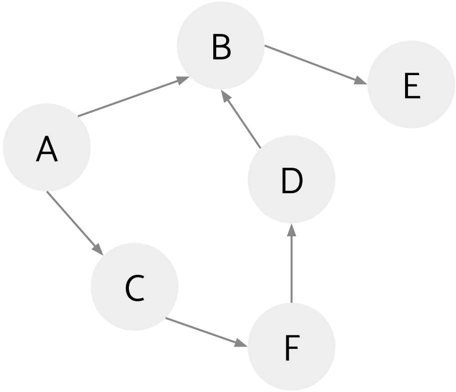


Check! How would you describe each of these graphs (in terms of directedness and cycles)?

Graph Representations

Adjacency lists list out all the nodes connected to each node in our graph:

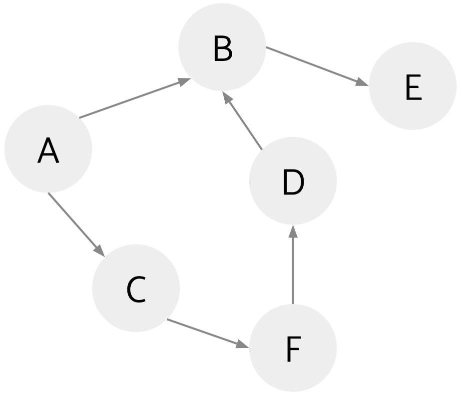
A	B , C
B	E
C	F
D	B
E	
F	D



Graph Representations

Adjacency matrices are true if there is a line going from node A to B and false otherwise.

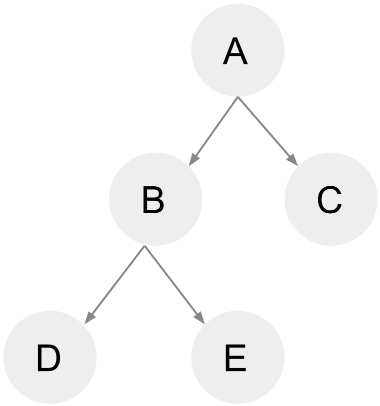
	A	B	C	D	E	F
A	0	1	1	0	0	0
B	0	0	0	0	1	0
C	0	0	0	0	0	1
D	0	1	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	1	0	0



Breadth First Search

Breadth first search means visiting nodes based off of their distance to the source, or starting point. For trees, this means visiting the nodes of a tree level by level. Breadth first search is one way of traversing a graph.

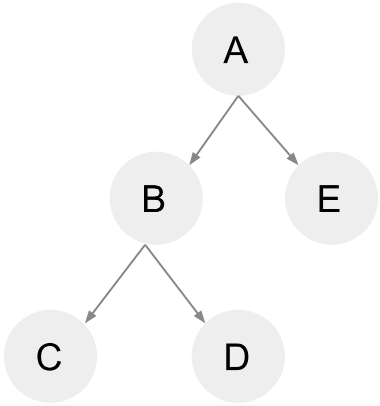
BFS is usually done using a queue.



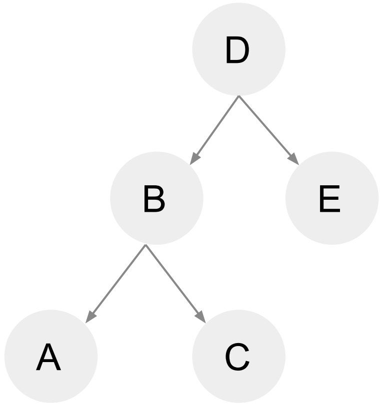

```
BFS(G):
     Add G.root to queue
     While queue not empty:
          Pop node from front of queue and visit
          for each immediate neighbor of node:
               Add neighbor to queue if not
               already visited
```

Depth First Search

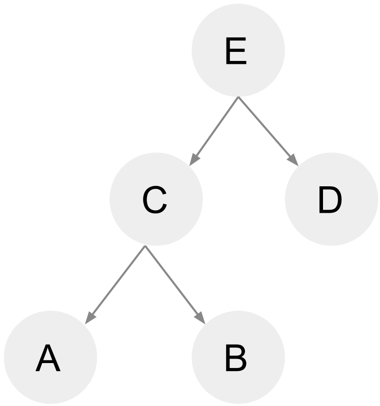
Depth First Search means we visit each subtree (subgraph) in some order recursively. DFS is usually done using a stack. Note that for graphs more generally, it doesn't really make sense to do in-order traversals.



Pre-order traversals visit the parent node before visiting child nodes.*



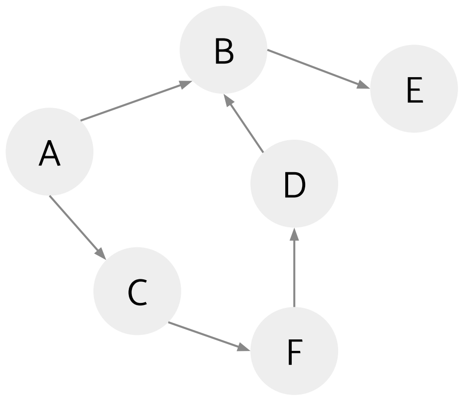
In-order traversals visit the left child, then the parent, then the right child.



Post-order traversals visit the child nodes before visiting the parent nodes.*

* in binary trees, we visit the left child before right child

General Graph DFS Pseudocode (Stack)




```
DFS(start):
    stack = {start}, visited = {}
    while stack not empty:
        n = top node in stack
        visited.add(n), preorder.add(n)
        if n has unvisited neighbors:
            push n’s next unvisited
            neighbor onto stack
        else:
            pop n off top of stack
            postorder.add(n)
    return preorder, postorder
```

Preorder: “Visit the node as soon as it enters the stack: myself, then all my children”

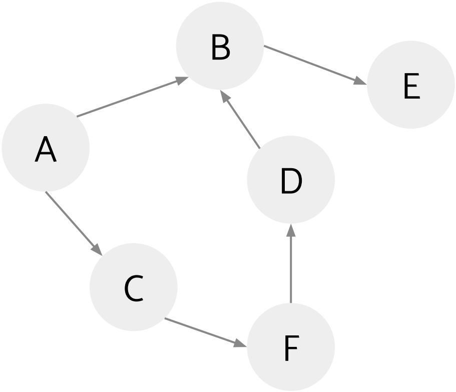
Postorder: “Visit the node as soon as it leaves the stack: all my children, then myself”

* in-order for binary trees:

```
DFSInorder(T):
    DFSInorder(T.left)
    visit T.root
    DFSInorder(T.right)
```

“Visit my left child, then myself, then my right child”*
* can be done with a stack, but usually easier with recursive

General Graph DFS Pseudocode (Recursive)




```
DFS(start):
     preorder.add(start)
     visited.add(start)
     for each neighbor of start:
          if neighbor not visited:
               DFS(neighbor)
     postorder.add(start)
     return preorder, postorder
```

* in-order for binary trees:

```
DFSInorder(T):
     DFSInorder(T.left)
     visit T.root
     DFSInorder(T.right)
```

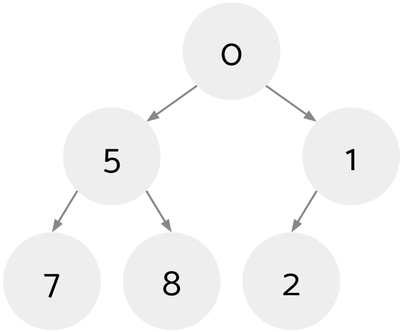
"Visit my left child, then myself, then my right child"*

* can be done with a stack, but usually easier with recursive

Heaps

Heaps are special trees that follow a few invariants:

1. Heaps are complete - the only empty parts of a heap are in the bottom row, to the right
2. In a min-heap, each node must be *smaller* than all of its child nodes. The opposite is true for max-heaps.

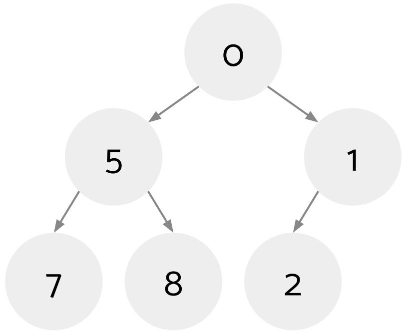


Check! What makes a binary min-heap different from a binary search tree?

Heap Representation

We can represent binary heaps as arrays with the following setup:

1. The root is stored at index 1 (not 0 - see points 2 and 3 for why)
2. The left child of a binary heap node at index i is stored at index 2i
3. The right child of a binary heap node at index i is stored at index 2i + 1

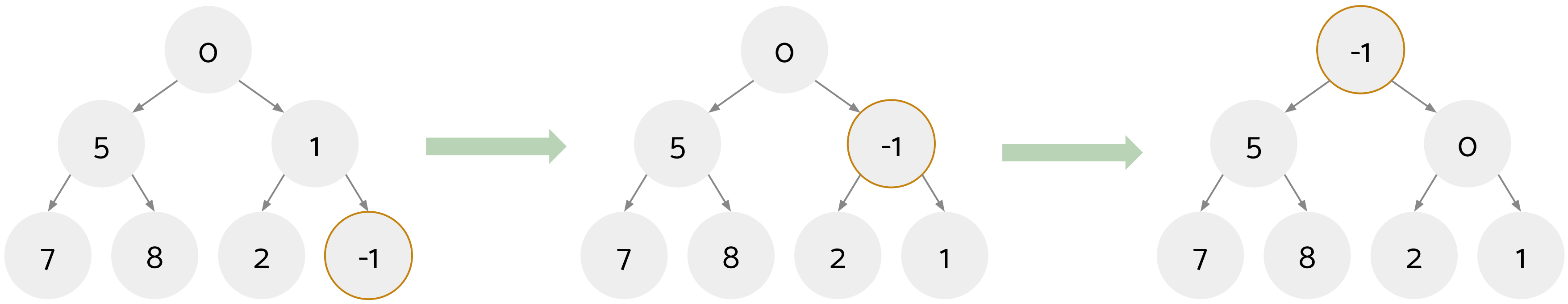



```
[-, 0, 5, 1, 7, 8, 2]
```

Check! What kind of graph traversal does the ordering of the elements in the array look like starting from the root at index 1?

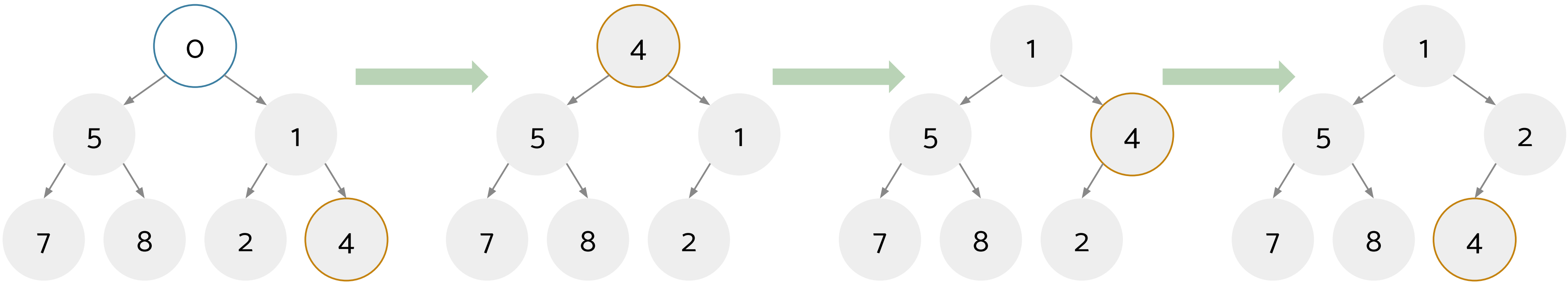
Insertion into (Min-)Heaps

We insert elements into the next available spot in the heap and bubble up as necessary: if a node is smaller than its parent, they will swap. (Check: what changes if this is a max heap?)



Root Deletion from (Min-)Heaps

We swap the last element with the root and bubble down as necessary: if a node is greater than its children, it will swap with the lesser of its children. (Check: what changes if this is a max heap?)



Heap Asymptotics (Worst case)

Operation	Runtime
insert	Θ(logN)
findMin	Θ(1)
removeMin	Θ(logN)

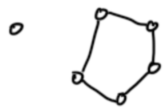
Worksheet

1 Graph Conceptuals

(a) Answer the following questions as either **True** or **False** and provide a brief explanation:

1. If a graph with n vertices has $n-1$ edges, it **must** be a tree.

No,



n-1 edges connecting n-1 vertices

2. Every edge is looked at exactly twice in **every** iteration of DFS on a connected, undirected graph.

Yes - get there and back

3. In BFS, let $d(v)$ be the minimum number of edges between a vertex v and the start vertex. For any two vertices u, v in the fringe, $|d(u) - d(v)|$ is **always less than** 2.

Yes, can't have distance of 2

Formal proof is by contradiction; intuition is BFS goes round by round

(b) Given an undirected graph, provide an algorithm that returns true if a cycle exists in the graph, and false otherwise. Also, provide a Θ bound for the worst case runtime of your algorithm.

DFS but modified

If we see a vertex again in exploration, there are multiple ways to get there

Since this is undirected, that means there is a cycle

One caveat: track parent; since it's undirected, can't have parent ↔ child interpreted as a cycle

2 Fill in the Blanks

Fill in the following blanks related to min-heaps. Let `N` is the number of elements in the min-heap. For the entirety of this question, assume the elements in the min-heap are **distinct**.

1. `removeMin` has a best case runtime of ___θ(1)___ (↙ 1 swap) and a worst case runtime of ___θ(log N)___ (↙ log N height).
2. `insert` has a best case runtime of ___θ(1)___ (↙ No swaps) and a worst case runtime of ___θ(log N)___.
3. A ___pre order___ or ___level order___ traversal on a min-heap *may* output the elements in sorted order. Assume there are at least 3 elements in the min-heap. Can't do in-order, post order
4. The fourth smallest element in a min-heap with 1000 elements can appear in ___14___ places in the heap. (Feel free to draw the heap in the space below.) ← Note: cannot be at top

   Can have on any of the 2-4 levels

   smallest
   2nd
   3rd
   4th at max

   2nd - 2
   3rd - 4
   4th - 8
   14

5. Given a min-heap with $2^N - 1$ distinct elements, for an element
   - to be on the second level it must be less than ___$2^{N-1} - 2$___ element(s) and greater than ___1___ element(s). - the root (half ↑, minus root, itself ↑)
   - to be on the bottommost level it must be less than ___0___ element(s) (← could be the biggest item) and greater than ___N-1___ element(s). - bigger than the path upwards

   Hint: A complete binary tree (with a full last-level) has $2^N - 1$ elements, with N being the number of levels. (Feel free to draw the heap in the space below.)

3 Heap Mystery *Optional*

This question is challenging! It is not expected that the TA would go over this problem, since they may spend time to do midterm Q A. Feel free to refer to the solutions and the linked video.

We are given the following array representing a min-heap where each letter represents a **unique** number. Assume the root of the min-heap is at index zero, i.e. `A` is the root. Note that there is **no** significance of the alphabetical ordering, i.e. just because B precedes C in the alphabet, we do not know if B is less than or greater than C.

Array: [A, B, C, D, E, F, G]

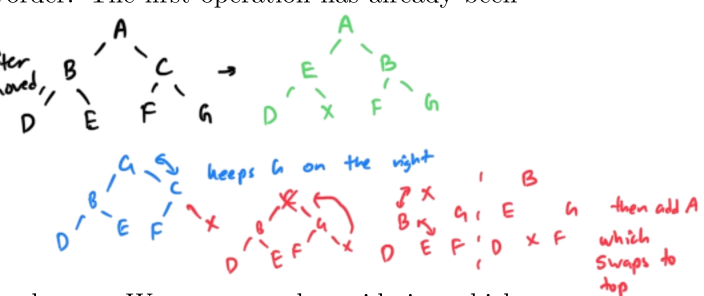
Four unknown operations are then executed on the min-heap. An operation is either a `removeMin` or an `insert`. The resulting state of the min-heap is shown below.

Array: [A, E, B, D, X, F, G]

must have an insert(X) and a removeMin() to delete C

(a) Determine the operations executed and their appropriate order. The first operation has already been filled in for you!

1. `removeMin()` - must have an insert(A) — has to be after, else A gets removed, not C
2. insert(X) - so X goes from right to left subtree
3. remove Min()
4. insert (A)



(b) Fill in the following comparisons with either >, <, or ? if unknown. We recommend considering which elements were compared to reach the final array.

1. X __?__ D only know E < X, E < D, not X in relation to D
2. X __>__ C to have X move to root after remove Min
3. B __>__ C keeps C at root to get deleted and B on second level
4. G __<__ X to have X move to root after remove Min

1a Graph Conceptuals (T/F)

1. If a graph with n vertices has n – 1 edges, it must be a tree.
2. Every edge is looked at exactly twice in every iteration of DFS on a connected, undirected graph.
3. In BFS, let d(v) be the minimum number of edges between a vertex v and the start vertex. For any two vertices u, v in the fringe, |d(u) – d(v)| is always less than 2.

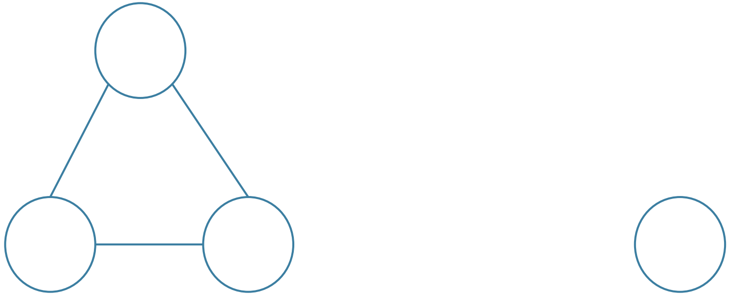
1a Graph Conceptuals (T/F)

1. If a graph with n vertices has n – 1 edges, it must be a tree.

1a Graph Conceptuals (T/F)

1. If a graph with n vertices has n – 1 edges, it must be a tree.

False. Could be disconnected.



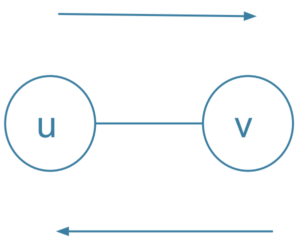
1a Graph Conceptuals (T/F)

2. Every edge is looked at exactly twice in every iteration of DFS on a connected, undirected graph.

1a Graph Conceptuals (T/F)

2. Every edge is looked at exactly twice in every iteration of DFS on a connected, undirected graph.

True. The two vertices the edge is connecting will look at that edge when it's their turn.



1a Graph Conceptuals (T/F)

3. In BFS, let d(v) be the minimum number of edges between a vertex v and the start vertex. For any two vertices u, v in the fringe, $|d(u) - d(v)|$ is always less than 2.

1a Graph Conceptuals (T/F)

3. In BFS, let d(v) be the minimum number of edges between a vertex v and the start vertex. For any two vertices u, v in the fringe, |d(u) – d(v)| is always less than 2.

True.

```
[2, 2, 3, 3, 4]
```

added after dequeuing dist-3 node

1a Graph Conceptuals (T/F)

3. In BFS, let d(v) be the minimum number of edges between a vertex v and the start vertex. For any two vertices u, v in the fringe, |d(u) – d(v)| is always less than 2.

True.

```
[2, 2, 3, 3, 4]
```

but can't deque dist-3 until all dist-2 nodes done!

added after dequeuing dist-3 node

1b Graph Conceptuals

Given an undirected graph, provide an algorithm that returns true if a cycle exists in the graph, and false otherwise. Also, provide a Θ bound for the worst case runtime of your algorithm.

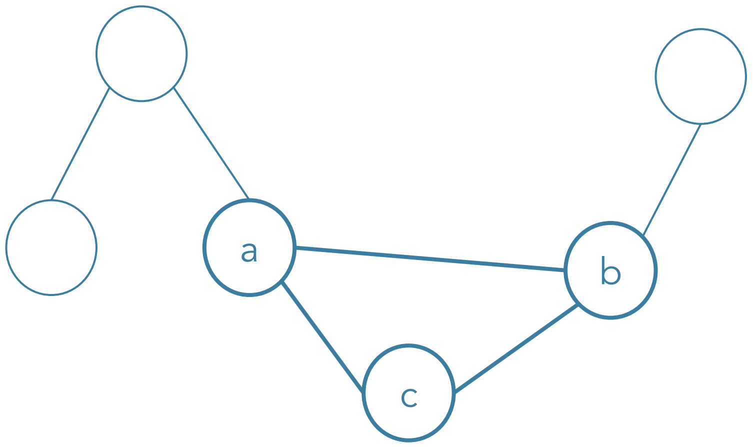
1b Graph Conceptuals

Given an undirected graph, provide an algorithm that returns true if a cycle exists in the graph, and false otherwise. Also, provide a Θ bound for the worst case runtime of your algorithm. You may use either an adjacency list or an adjacency matrix to represent your graph. (We are looking for an answer in plain English, not code).

Basic Idea: Keep track of visited nodes, do a DFS and if we visit any already visited nodes there is a cycle.

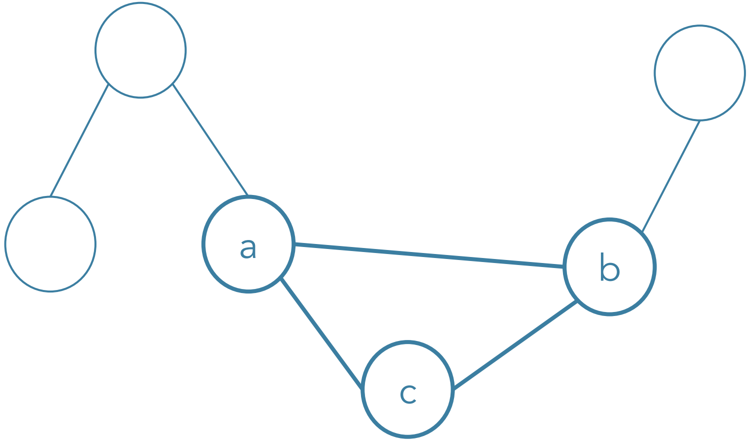
1b Graph Conceptuals

Basic Idea: Keep track of visited nodes, do a DFS and if we visit any already visited nodes there is a cycle.



1b Graph Conceptuals

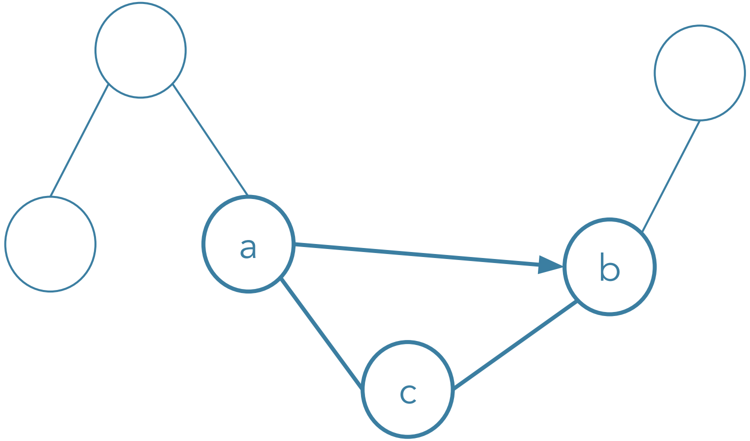
Basic Idea: Keep track of visited nodes, do a DFS and if we visit any already visited nodes there is a cycle.



dfs(a)

1b Graph Conceptuals

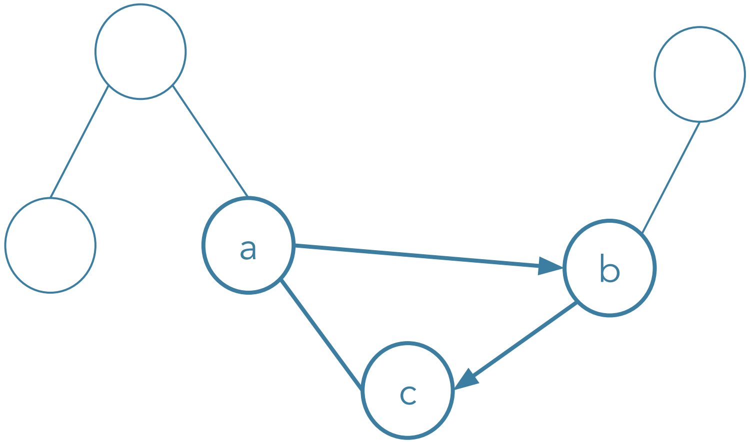
Basic Idea: Keep track of visited nodes, do a DFS and if we visit any already visited nodes there is a cycle.



dfs(a) → dfs(b)

1b Graph Conceptuals

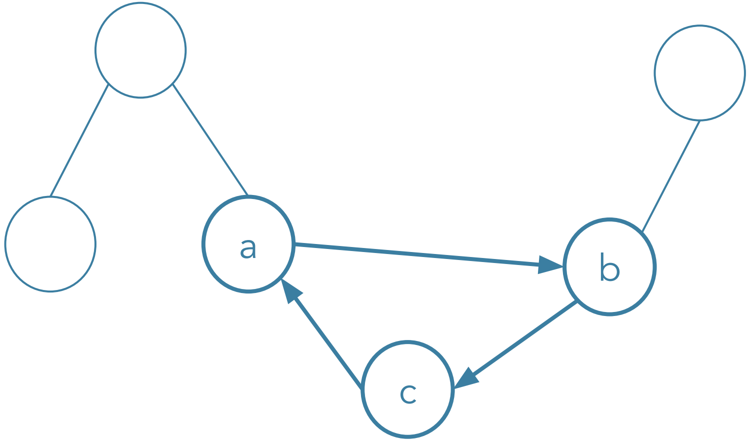
Basic Idea: Keep track of visited nodes, do a DFS and if we visit any already visited nodes there is a cycle.



dfs(a) → dfs(b)
→ dfs(c)

1b Graph Conceptuals

Basic Idea: Keep track of visited nodes, do a DFS and if we visit any already visited nodes there is a cycle.



dfs(a) → dfs(b) → dfs(c) → dfs(a)

repeat = cycle

2 Fill in the Blanks

1. `removeMin` has a best case runtime of ______ and a worst case runtime of ______.

2 Fill in the Blanks

1. `removeMin` has a best case runtime of Θ(1) and a worst case runtime of Θ(logN).

Best case: only one swap down is required, thus finishing in constant time
Worst case: sink down from top to the bottom. Height = Θ(logN)

2 Fill in the Blanks

2. `insert` has a best case runtime of ______ and a worst case runtime of ______.

2 Fill in the Blanks

2. `insert` has a best case runtime of Θ(1) and a worst case runtime of Θ(logN).

Best case: no bubbling up required
Worst case: bubble up from bottom to top. Height = Θ(logN)

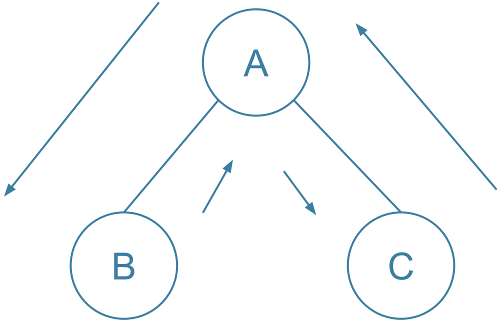
2 Fill in the Blanks

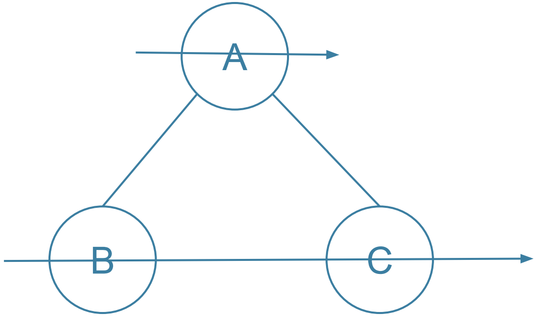
3. A ___________ or ___________ traversal on a min-heap *may* output the elements in sorted order. Assume there are at least 3 elements in the min-heap.

2 Fill in the Blanks

3. A pre-order or level-order traversal on a min-heap *may* output the elements in sorted order. Assume there are at least 3 elements in the min-heap.

Any traversal must output *the top node first*. Only pre-order and level-order obey this constraint.





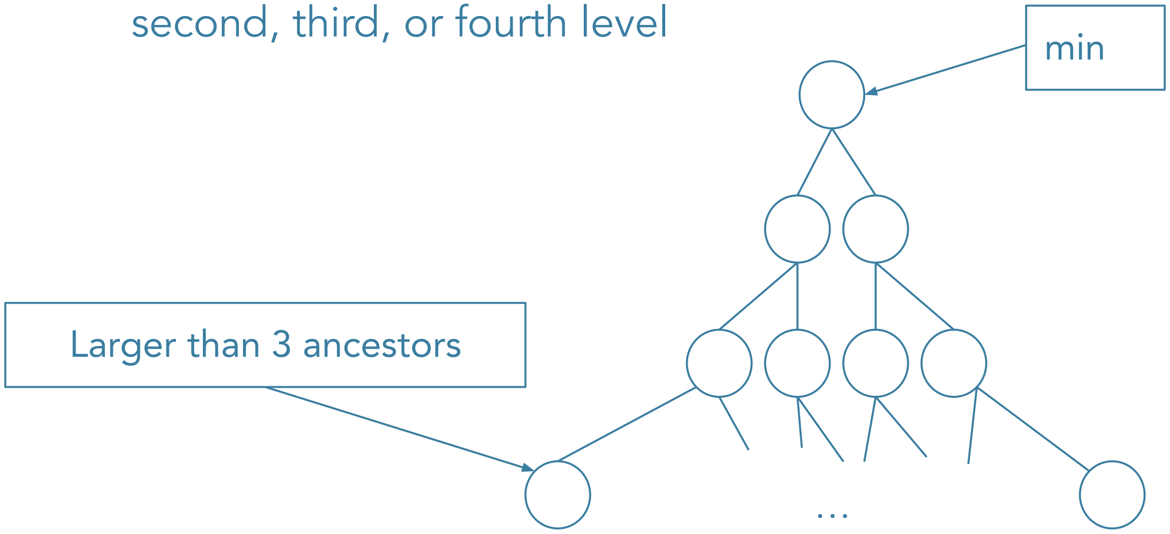
2 Fill in the Blanks

4. The fourth smallest element in a min-heap with 1000 elements can appear in _____ places in the heap.

2 Fill in the Blanks

4. The fourth smallest element in a min-heap with 1000 elements can appear in 14 places in the heap.

second, third, or fourth level



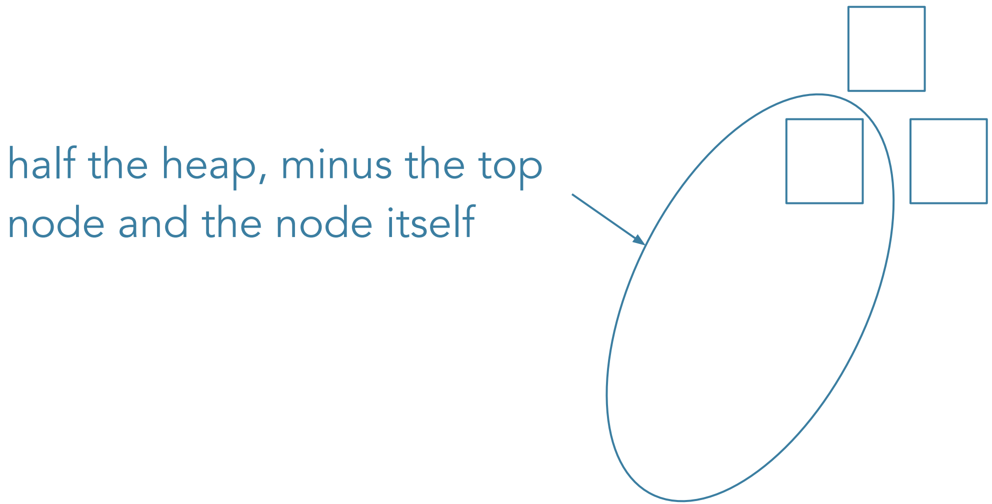
2 Fill in the Blanks

5. Given a min-heap with $2^n - 1$ elements, for an element to be on the second level it must be less than ______ element(s) and greater than ___ element(s).

2 Fill in the Blanks

5. Given a min-heap with $2^n - 1$ elements, for an element to be on the second level it must be less than $2^{(N-1)} - 2$ element(s) and greater than 1 element(s).

must be greater than the topmost and less than the elements in its subtree



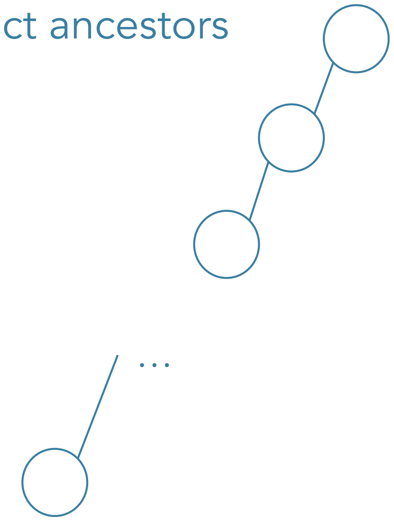
2 Fill in the Blanks

5. Given a min-heap with $2^n - 1$ elements, for an element to be on the bottommost level it must be less than ______ element(s) and greater than ______ element(s).

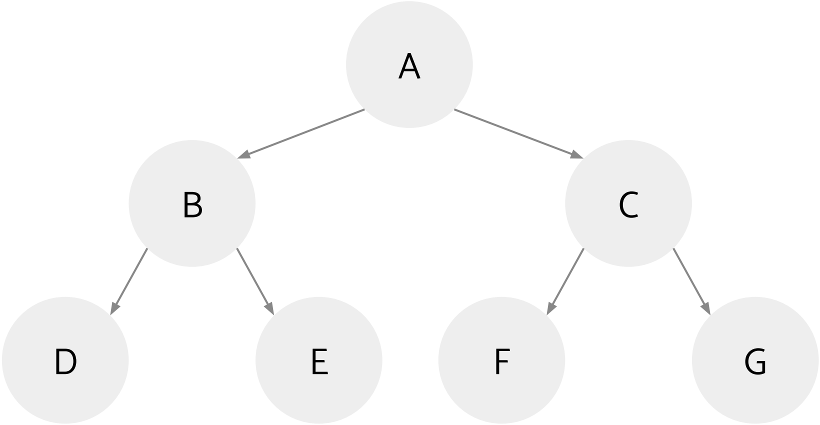
2 Fill in the Blanks

5. Given a min-heap with $2^n - 1$ elements, for an element to be on the bottommost level it must be less than 0 element(s) and greater than N - 1 element(s).

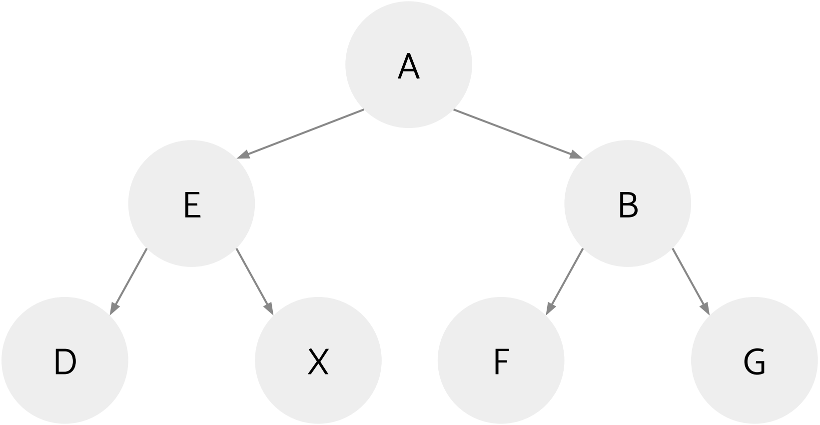
larger than all direct ancestors



3a Heap Mystery (Optional)

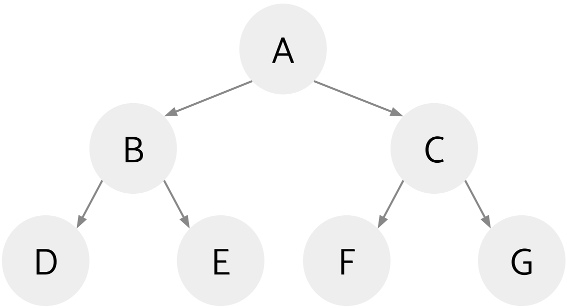


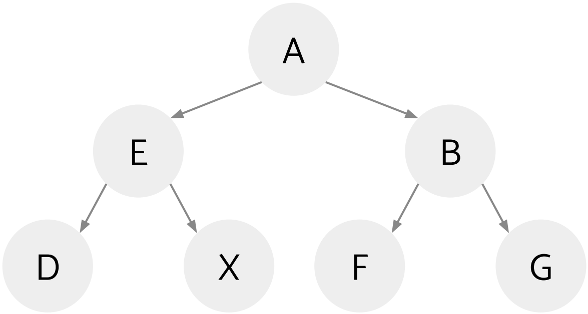
Initial State
[-, A, B, C, D, E, F, G]



Final State:
[-, A, E, B, D, X, F, G]

3a Heap Mystery





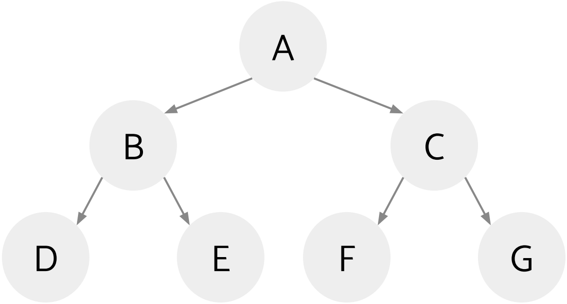
Sequence of calls:

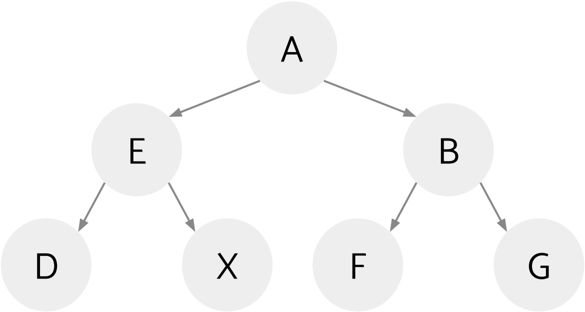
1. `removeMin()`
2. ___________
3. ___________
4. ___________

Differences in state:

- C was removed: `removeMin()`
- X was added: insert(X)
- A was removed by first call to `removeMin()` and added back: `insert(A)`

3a Heap Mystery





`insert(A)` must be after all `removeMin()` – otherwise would remove A again

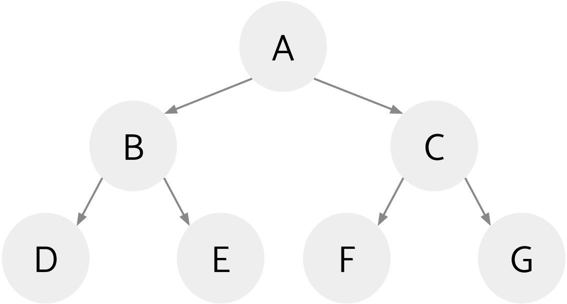
Sequence of calls:

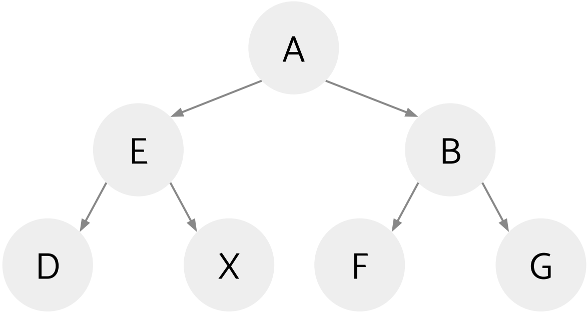
1. `removeMin()`
2. `removeMin()` / `insert(X)`
3. `removeMin()` / `insert(X)`
4. `insert(A)`

Differences in state:

- C was removed: `removeMin()`
- X was added: insert(X)
- A was removed by first call to `removeMin()` and added back: `insert(A)`

3a Heap Mystery





`insert(X)` must be before removeMin, since it bubbles up then down

Sequence of calls:

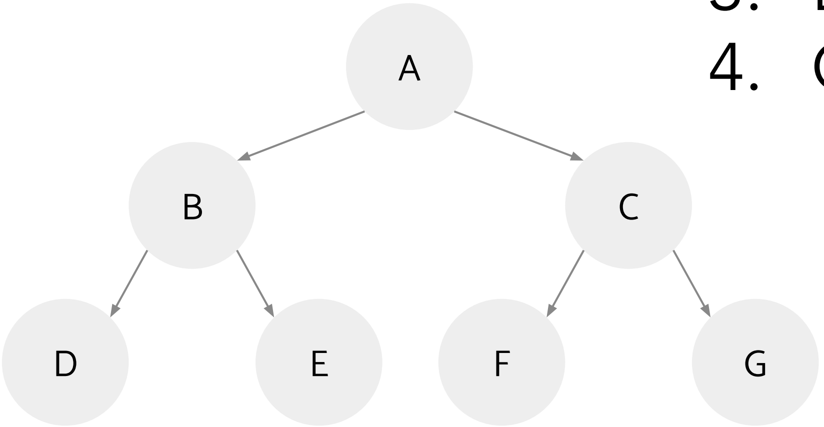
1. `removeMin()`
2. `insert(X)`
3. `removeMin()`
4. `insert(A)`

Differences in state:

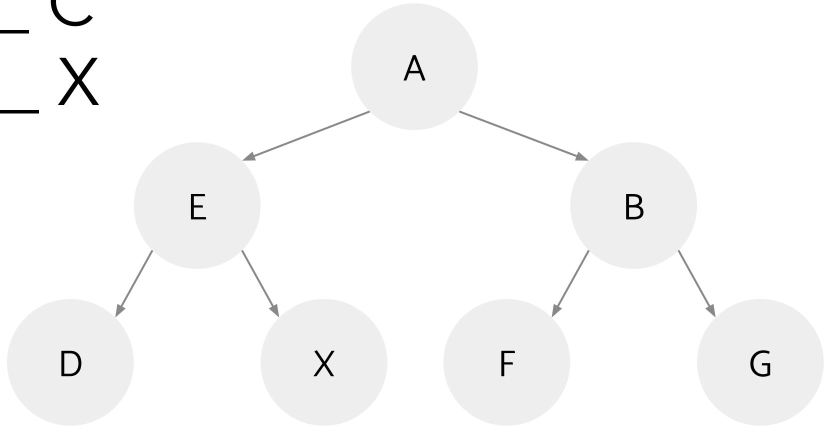
- C was removed: `removeMin()`
- X was added: insert(X)
- A was removed by first call to `removeMin()` and added back: `insert(A)`

3b Heap Mystery

1. X _____ D
2. X _____ C
3. B _____ C
4. G _____ X



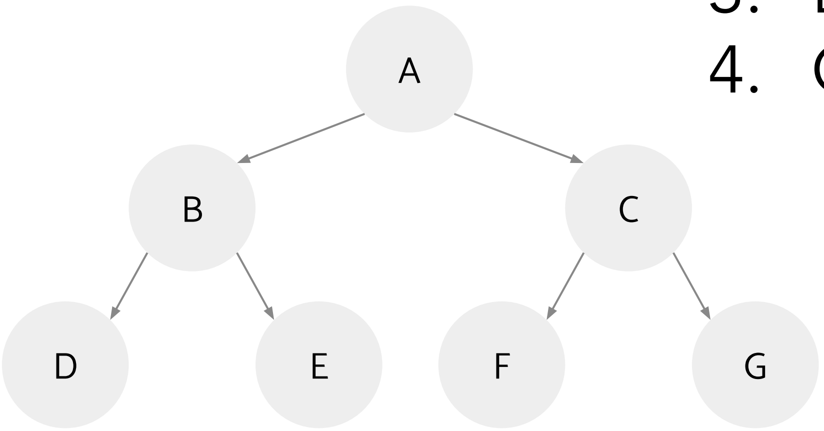
Initial State
[-, A, B, C, D, E, F, G]



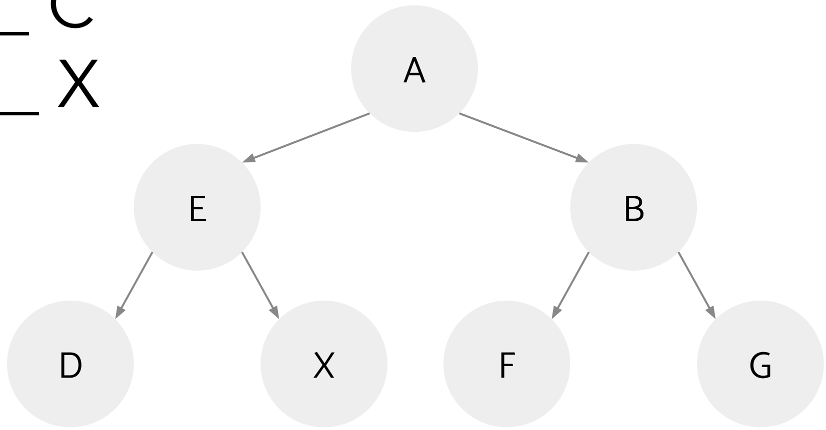
Final State:
[-, A, E, B, D, X, F, G]

3b Heap Mystery

1. X ? D
2. X _____ C
3. B _____ C
4. G _____ X



Initial State
[-, A, B, C, D, E, F, G]

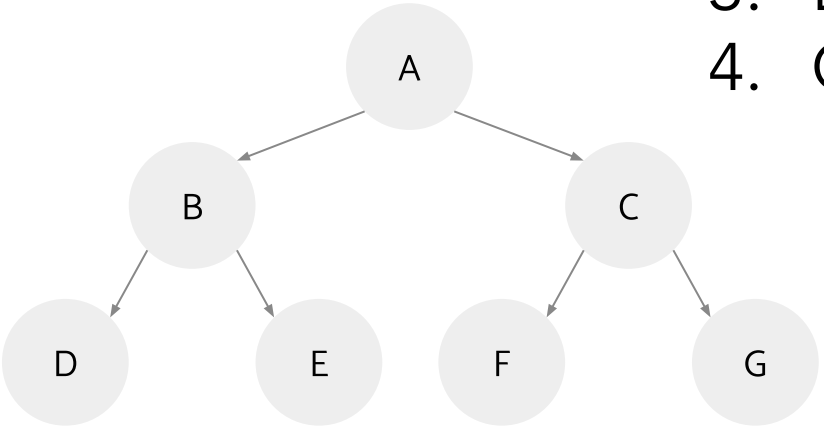


Final State:
[-, A, E, B, D, X, F, G]

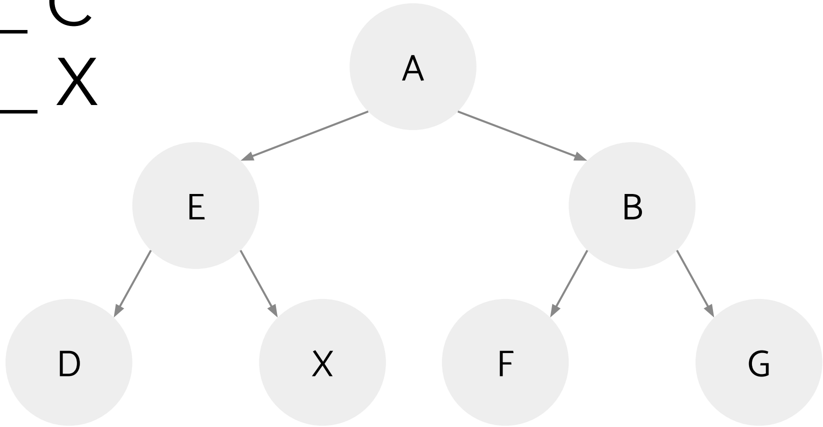
X is never compared to D

3b Heap Mystery

1. X ? D
2. X > C
3. B _____ C
4. G _____ X



Initial State
[-, A, B, C, D, E, F, G]

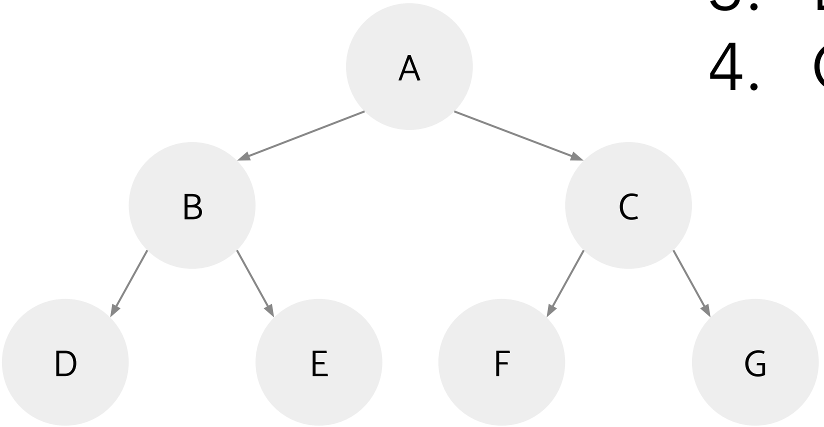


Final State:
[-, A, E, B, D, X, F, G]

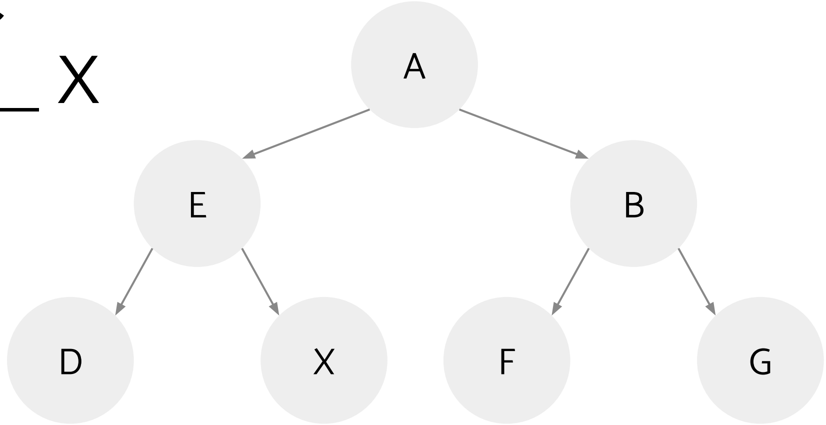
X must be greater than C, since removeMin removes C instead of X

3b Heap Mystery

1. X ? D
2. X > C
3. B > C
4. G ____ X



Initial State
[-, A, B, C, D, E, F, G]

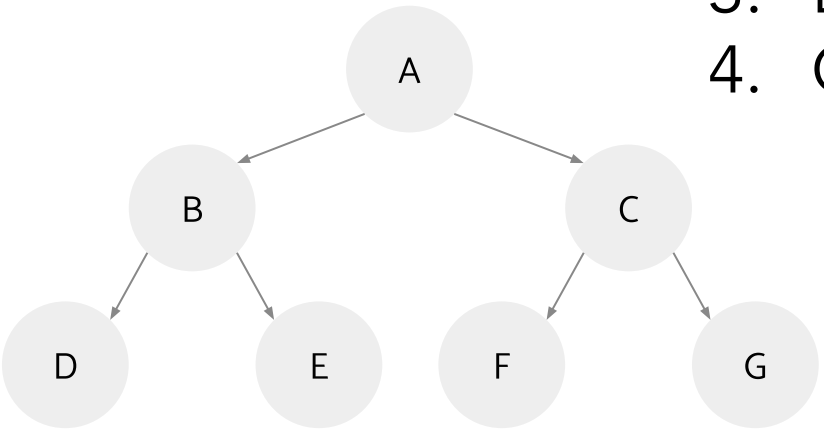


Final State:
[-, A, E, B, D, X, F, G]

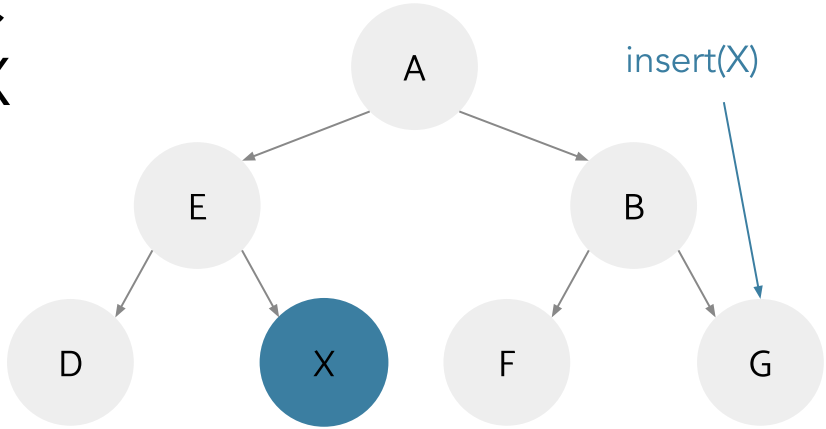
B > C otherwise the second call to removeMin would have removed B

3b Heap Mystery

1. X ? D
2. X > C
3. B > C
4. G < X



Initial State
[-, A, B, C, D, E, F, G]



Final State:
[-, A, E, B, D, X, F, G]

X > G since it bubbles up then down