Graphs, Heaps

Exam-Level 08

Announcements

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
			3/13 Mid-semester Survey Due		3/15 Lab 8 Due Project 2B/C Checkpoint and Design Doc Due TRS 3 (11-1pm)	
	3/18 Homework 3 Due			3/21 **Midterm 2**		

Content Review

Trees, Revisited (and Formally Defined)

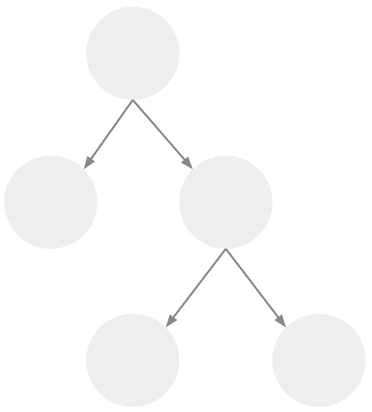
Trees are structures that follow a few basic rules:

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.

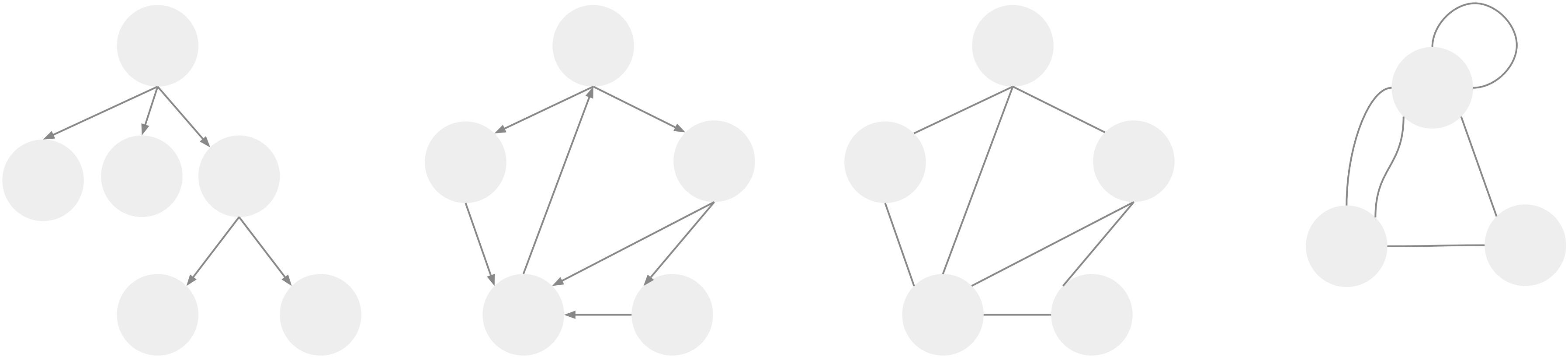
A leaf of a tree is a node with no child nodes.



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)

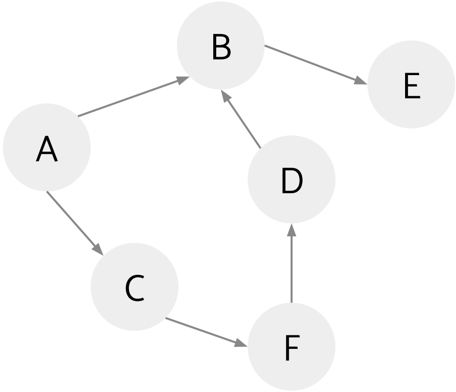


Check! How would you describe each of these graphs (in terms of directedness and cycles)?

Graph Representations

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

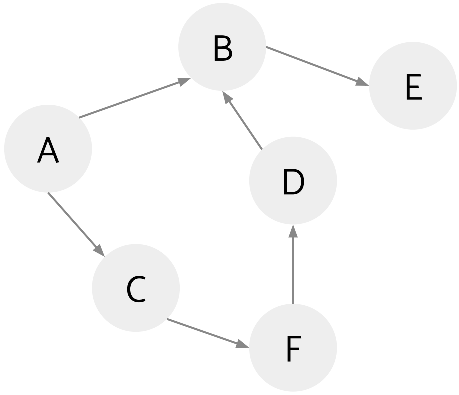
A	B , C
B	E
C	F
D	B
E	
F	D



Graph Representations

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

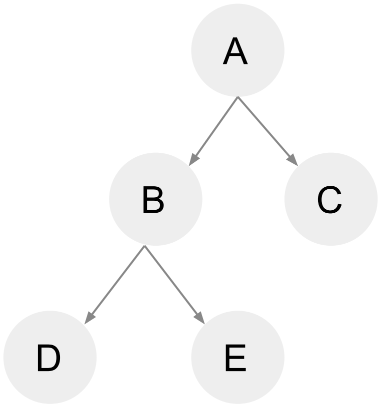
	A	B	C	D	E	F
A	0	1	1	0	0	0
B	0	0	0	0	1	0
C	0	0	0	0	0	1
D	0	1	0	0	0	0
E	0	0	0	0	0	0
F	0	0	0	1	0	0



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.

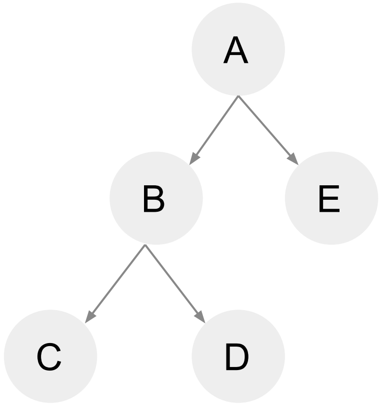
BFS is usually done using a queue.



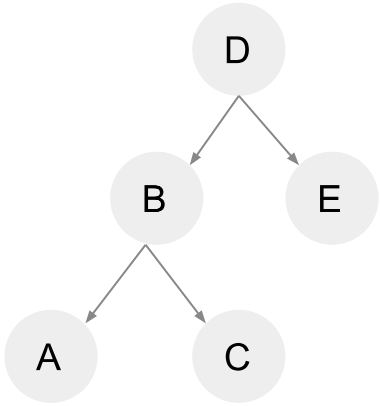

```
BFS(G):
     Add G.root to queue
     While queue not empty:
          Pop node from front of queue and visit
          for each immediate neighbor of node:
               Add neighbor to queue if not
               already visited
```

Depth First Search

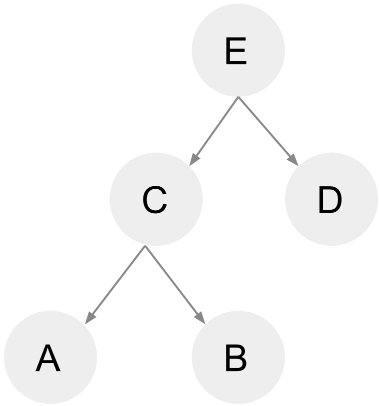
Depth First Search means we visit each subtree (subgraph) in some order recursively. DFS is usually done using a stack. Note that for graphs more generally, it doesn't really make sense to do in-order traversals.



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



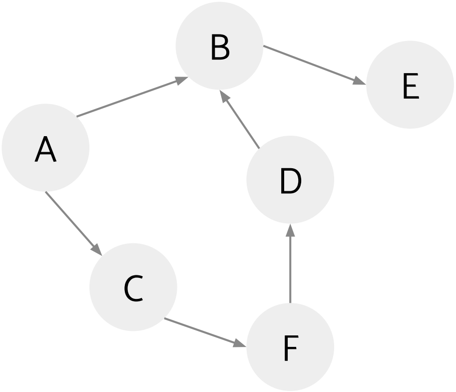
In-order traversals visit the left child, then the parent, then the right child.



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”

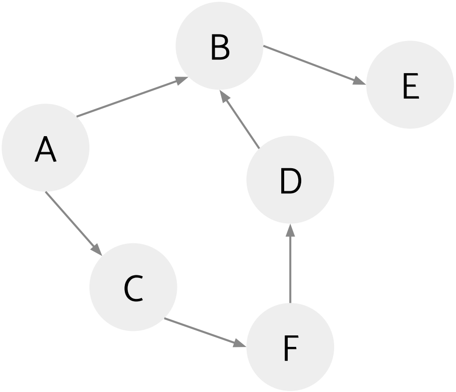
* in-order for binary trees:

```
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
```

Note: technically can add:

```
if start.neighbors is empty
  preorder.add(start)
  visited.add(start)
  postorder.add(start)
```

as base case, but the code on the left will skip the loop if neighbors is empty.

* in-order for binary trees:

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

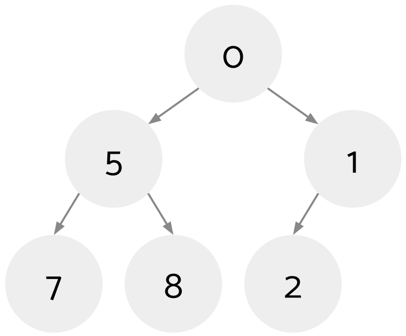
"Visit my left child, then myself, then my right child"*

* 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.

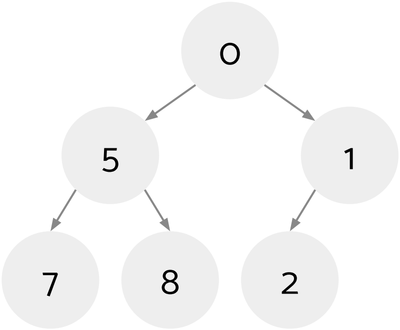


Check! What makes a binary min-heap different from a binary search tree?

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

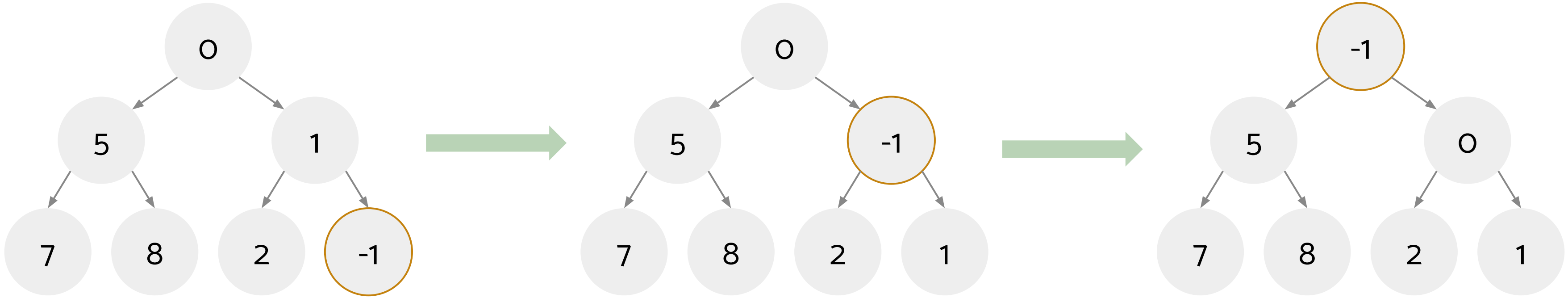


`[-, 0, 5, 1, 7, 8, 2]`

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

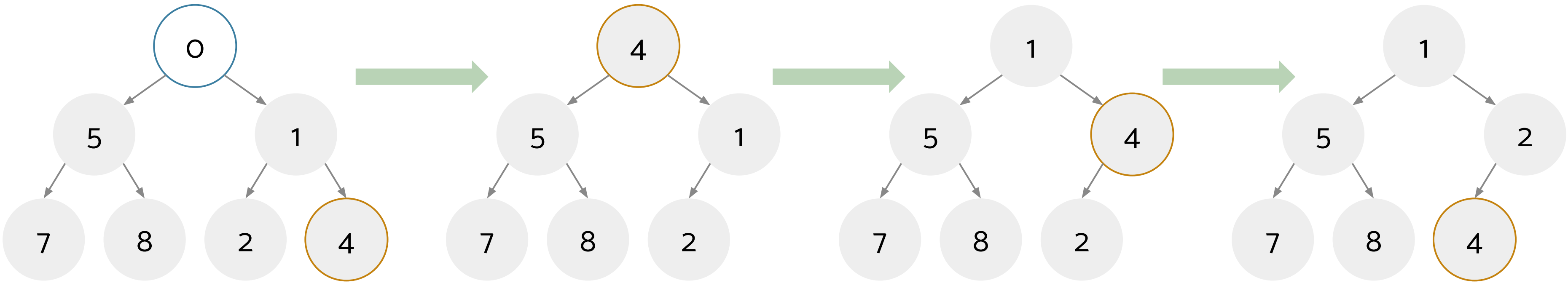
Insertion into (Min-)Heaps

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	$\Theta(\log N)$
findMin	$\Theta(1)$
removeMin	$\Theta(\log N)$

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.

   False - what if it is not connected?

2. Every edge is looked at exactly twice in each full run of DFS on a connected, undirected graph.

   True - every edge has 2 vertices, looked at when both are visited

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 (recall that the fringe in BFS is a queue), $|d(u) - d(v)|$ is **always less than** 2.

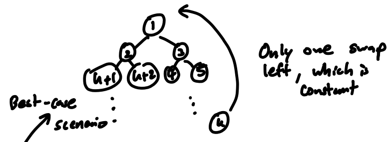
   True - not possible to visit 2 beyond; must go through all of a level in BFS first

(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 through but if a vertex is seen again and has been visited then there must be a cycle

Track parent throughout since the graph is undirected

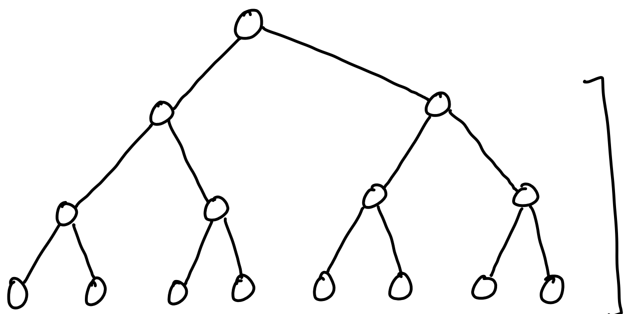
If disconnected, run on each section



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 ___ $\Theta(1)$ ___ and a worst case runtime of ___ $\Theta(\log N)$ ___.
2. `insert` has a best case runtime of ___ $\Theta(1)$ ___ and a worst case runtime of ___ $\Theta(\log N)$ ___. (no swaps, largest item)
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.
4. The fourth smallest element in a min-heap with 1000 **distinct** elements can appear in ___ 14 ___ places in the heap. (Feel free to draw the heap in the space below.)



5. Given a min-heap with $2^N - 1$ **distinct** elements, for an element (full heap, N levels)

   $2^{N-1} - 1 = \frac{2^N - 2}{2} = 2^{N-1} - 1 - 1 = 2^{N-1} - 2$

   - to be on the second level it must be less than ___ $2^{N-1} - 2$ ___ element(s) and greater than ___ 1 ___ element(s). (its subtree, minus itself; root)
   - to be on the bottommost level it must be less than ___ 0 ___ element(s) and greater than ___ N-1 ___ element(s). (could be largest; along its path to top)

   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

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. Our task is to figure out the numeric ordering of the letters. Therefore, 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]

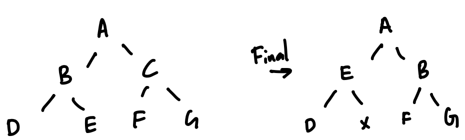
(a) Determine the operations executed and their appropriate order. The first operation has already been filled in for you!

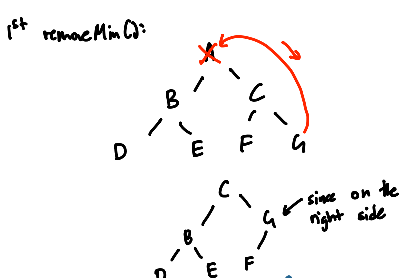
Hint: Which elements are gone? Which elements are newly added? Which elements are removed and then added back?

1. `removeMin()` → removes A, so must remove C, add A, add X : remove Min, insert(A), insert(X)
2. insert(X)
3. removeMin()
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 not compared; only know E < X, E < D to move E down
2. X __?__ C keeps C at top to be removed; C < G < X
3. B __>__ C keeps C at top to be deleted on removeMin
4. G __<__ X necessary to keep X to be swapped in for removeMin





X would go here,
but needs to go left
inserting A means no swaps
removing C then adding X or A
doesn't make X go right

∴ insert (X), removeMin() to delete C and rotate X to left subtree, then insert A