

# CS151 Intro to Data Structures

## Heaps & Priority Queues

# Announcements

HW5 autograder broken - sorry :(

Lab today: heap implementation

- autograded

Exam grades coming in an email today

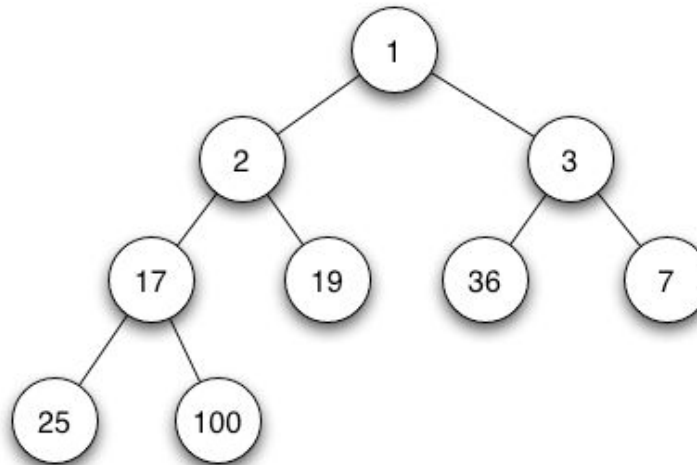
# Outline

- Heap Review
- Selection sort and heap sort review
- Breadth First Traversal
- Priority Queues

# Heap Review

# Binary Heap Properties

1. Each node has **at most 2 children**
1. For every node  $n$  (except for the root):  **$n.\text{key} \geq \text{parent}(n).\text{key}$**
1. **Complete:** all levels of the tree, except possibly the last one are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right



# Heap Operations:

1. Insert
  - a. Upheap
2. Poll
  - a. Downheap
3. Search
4. Remove

# Selection Sort

In place sorting algorithm

1. Separate the array into “sorted” and “unsorted”
  - a. sorted starts empty
2. Find the min element in the unsorted array
3. Swap min with the first element in unsorted
4. repeat

# Selection sort

code

Runtime complexity?

$O(n^2)$

Space complexity?

$O(1)$



# Heap Sort

# Heap Sort

1. Heap construction
  - a. rearrange the array into a heap
2. Heap extraction
  - a. iteratively **poll** and insert into the sorted portion

# Heap Sort - in place

## Heapify Algorithm:

1. **Start from the *first non-leaf node*** and move upwards to the root.
  - a. “first” in a top to bottom left to right order
  - b. There’s a term for this order we will learn it today!
2. **For each node**, downheap if necessary
3. **Recursively heapify** the affected subtree to maintain the min-heap property.

[64, 25, 12, 22, 11]

# Heap Sort

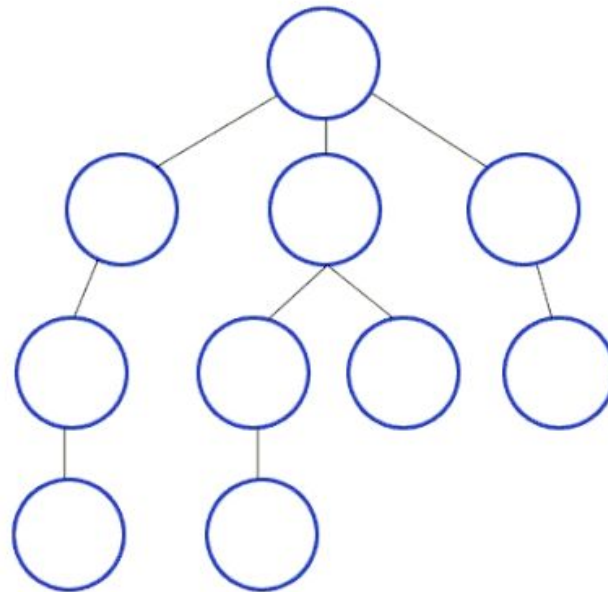
$O(n \log n)$  sort algorithm

Selection sort with the right data structure

# Breadth First Search

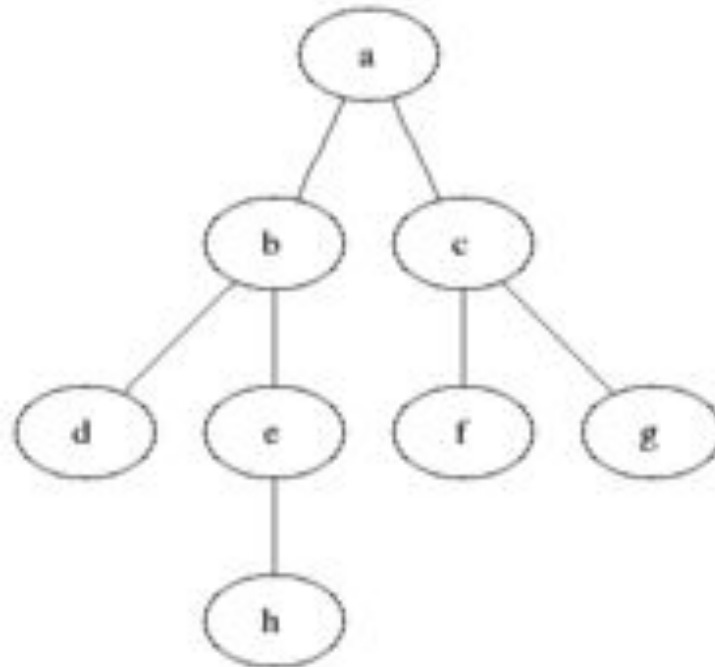
# Depth First Search (DFS)

- Depth First Search (DFS)
  - start at root node and explore as far as possible along each branch
  - applications? when have we used this in class?



# Breadth First Search (BFS)

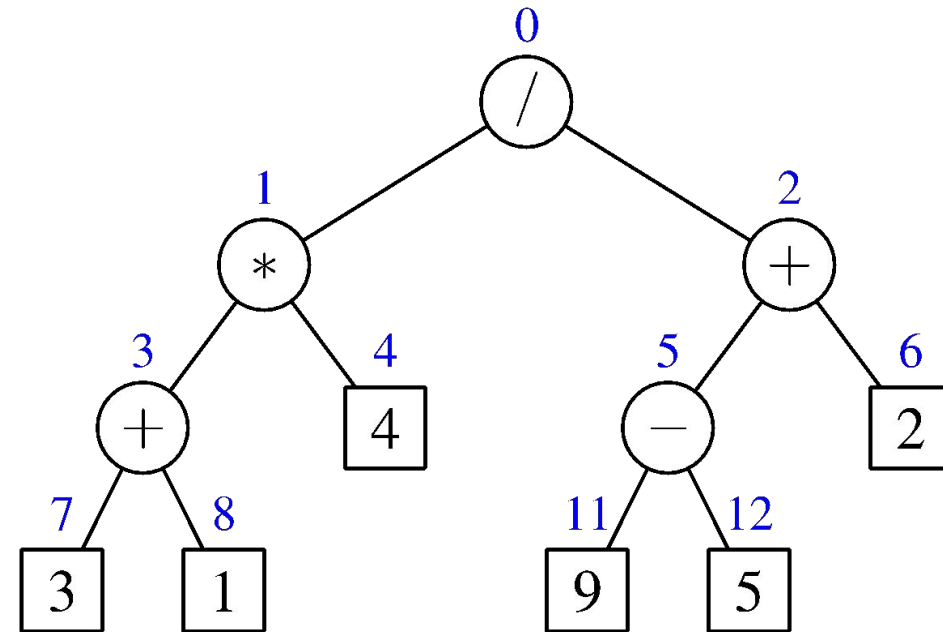
- Breadth First Search (BFS)
  - Starts at the root and explores all nodes at the present “depth” before moving to nodes on the next level
  - Extra memory is usually required to keep track of the nodes that have not yet been explored



# Breadth-First Traversal

Traverse the tree level-by-level

- Within a level go left-to-right

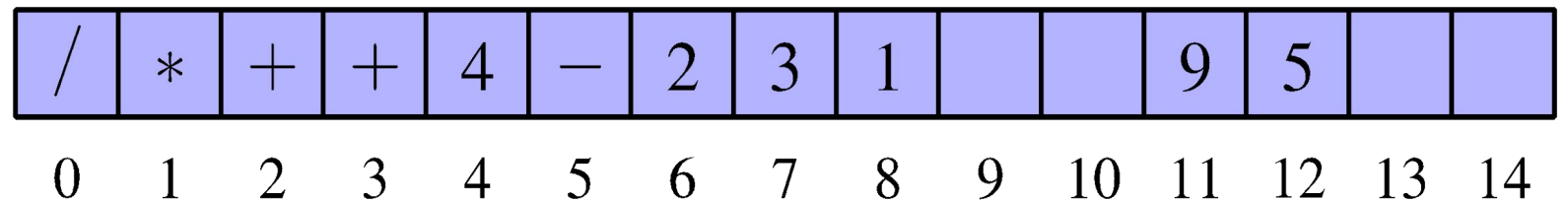
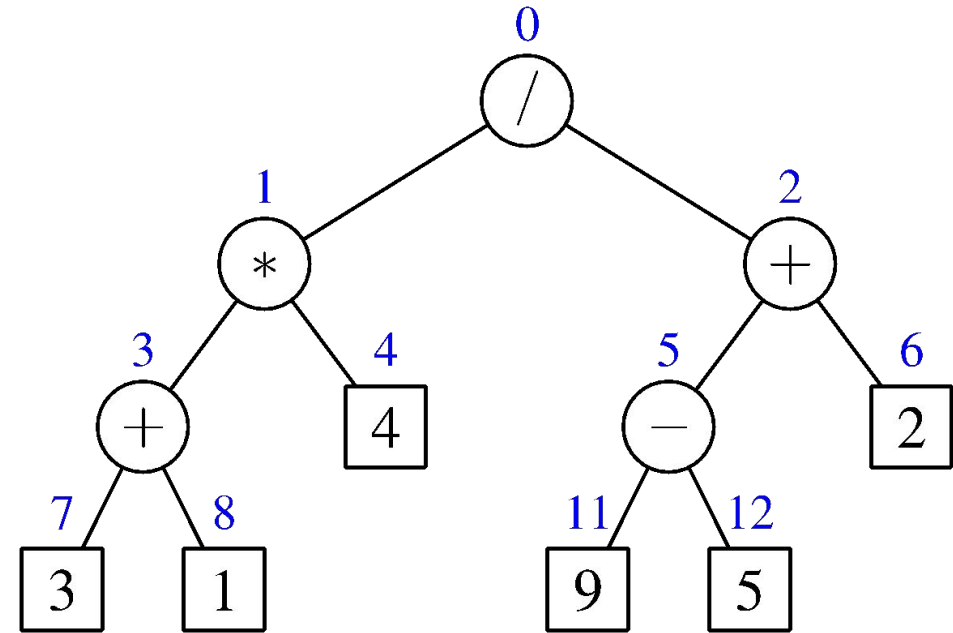




# Breadth-First Traversal

Traverse the tree level-by-level

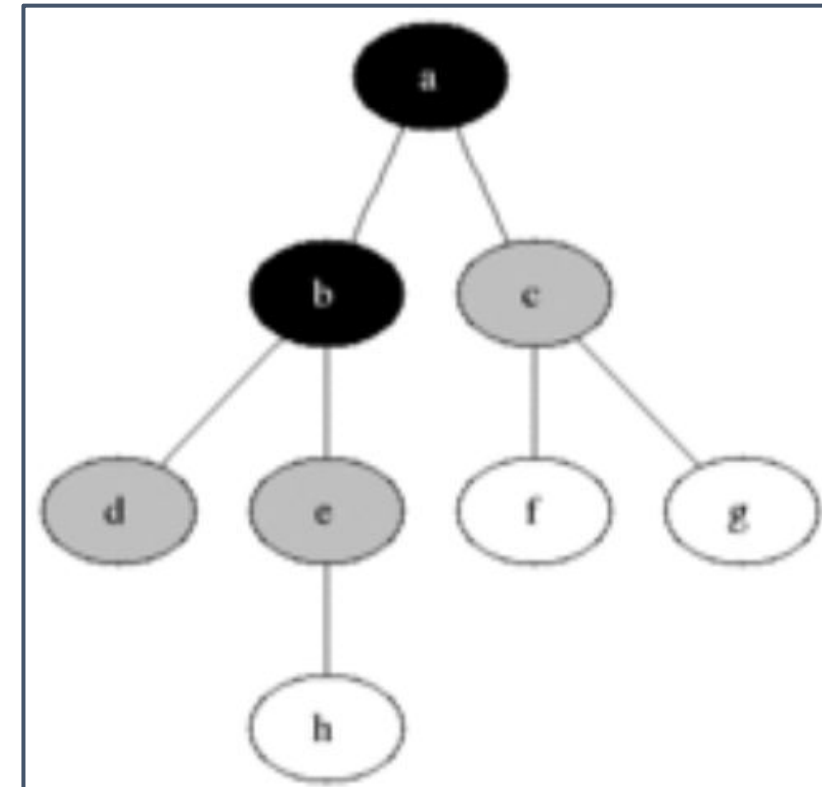
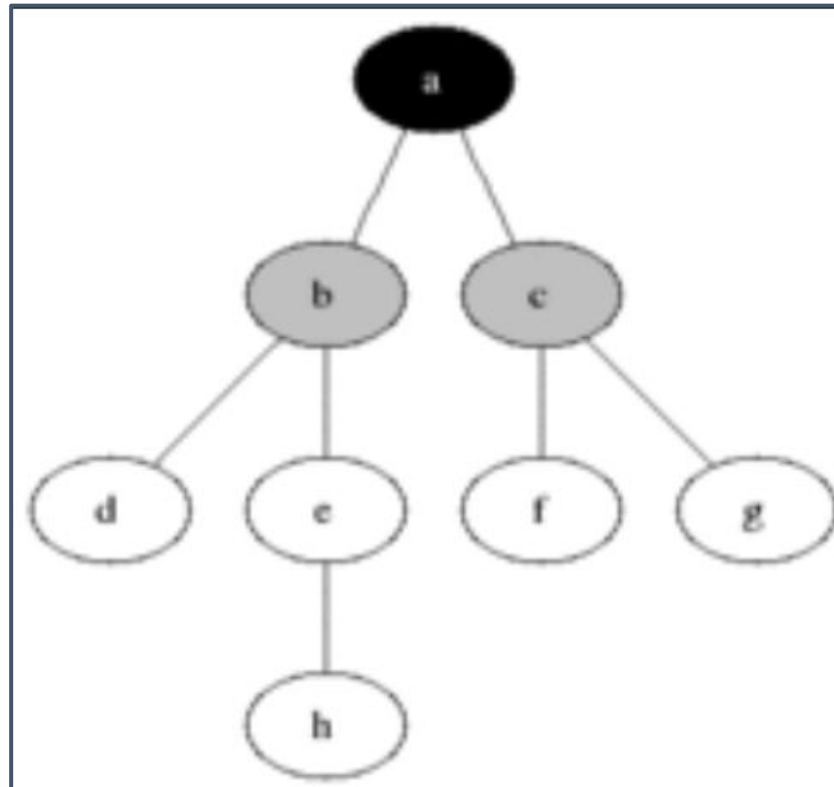
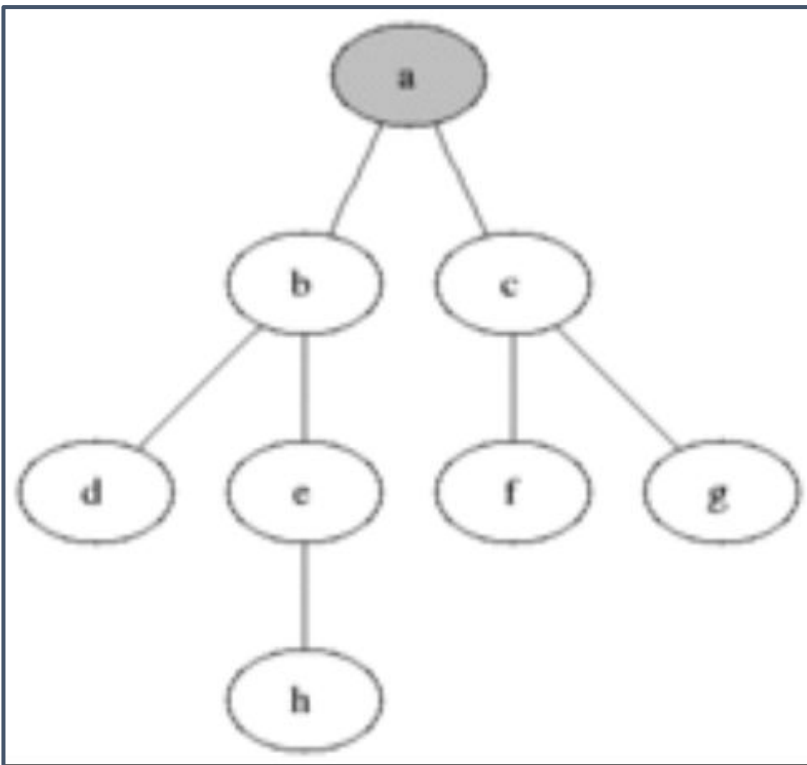
- Within a level go left-to-right



This is the array order of an array-based binary tree

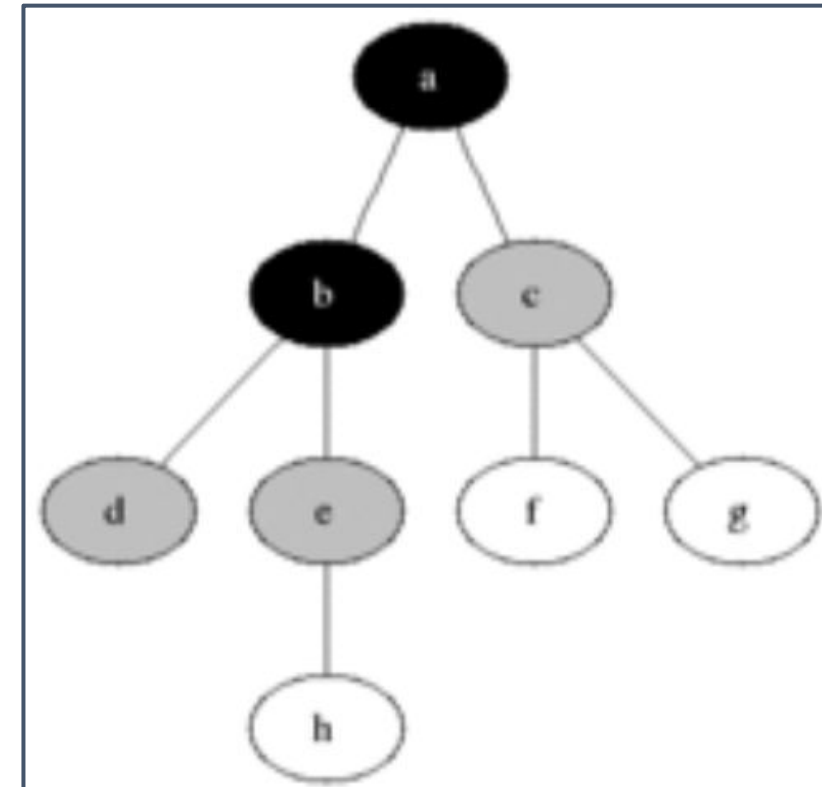
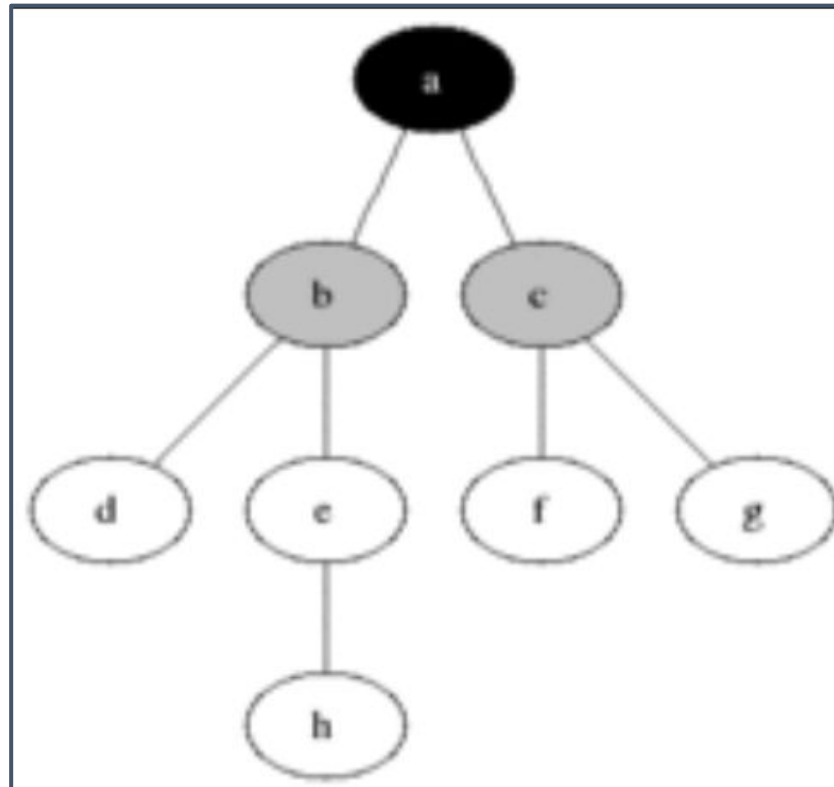
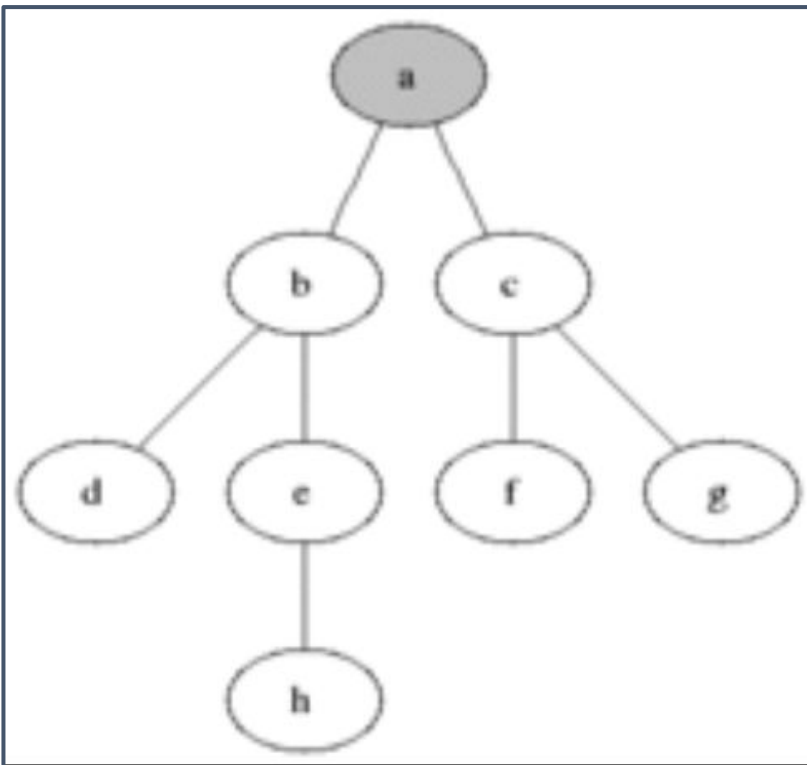
# Breadth-First Traversal

pseudo-code?



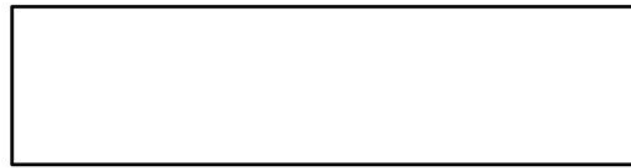
# Breadth-First Search

How would we change the pseudo-code?

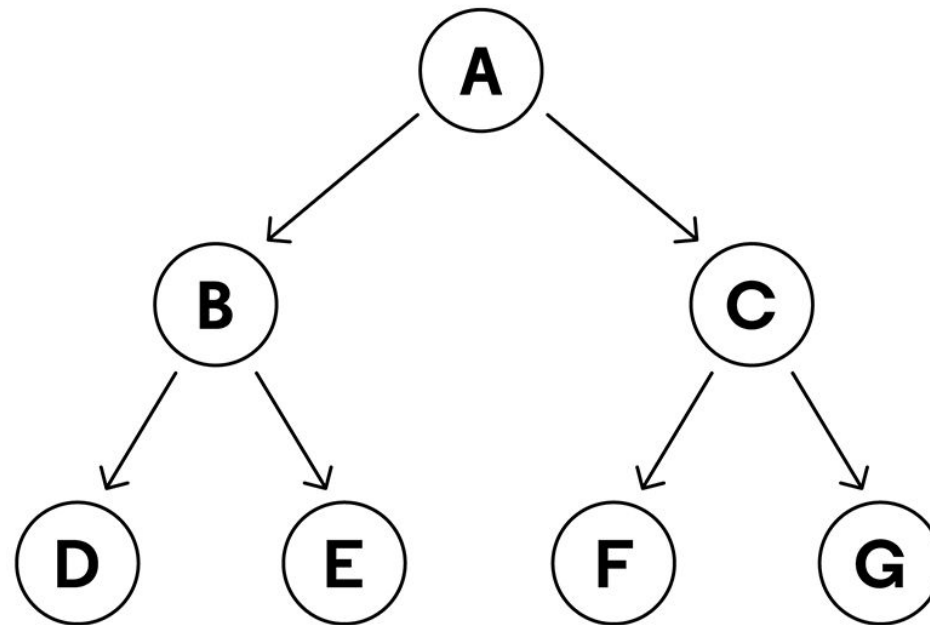


# Breadth First Search (BFS)

**Tree with an Empty Queue**



**Frontier Queue**  
FIFO (First in First Out)



<https://www.codecademy.com/article/tree-traversal>

# Priority Queue

# Priority Queues

- What is a queue?
- What if we want to create a graduation queue of students who pick up their diplomas in alphabetical order?
  - Queues and Stacks removal is based on when it was added to the data structure
  - Instead we want removal order to be based on the student's name

# Priority Queue

A queue that maintains removal order of the elements according to some priority

- generally not related to insertion time (although time of insertion COULD be one criteria)
- each element has an associated priority with it which indicates when it should be removed
- Usually removed based on min priority
- **What data structure can we use to implement a priority queue?**

# Priority Queue ADT

`insert( $k, v$ )`: Creates an entry with key  $k$  and value  $v$  in the priority queue.

`min()`: Returns (but does not remove) a priority queue entry  $(k, v)$  having minimal key; returns null if the priority queue is empty.

`removeMin()`: Removes and returns an entry  $(k, v)$  having minimal key from the priority queue; returns null if the priority queue is empty.

`size()`: Returns the number of entries in the priority queue.

`isEmpty()`: Returns a boolean indicating whether the priority queue is empty.



# Priority Queue

Entries (elements) are Key/Value pairs

The key represents the priority

Key type must be comparable

# Entry Interface

```
public interface Entry<K extends Comparable<K>, V> {  
  
    K getKey();  
    V getValue();  
  
}
```

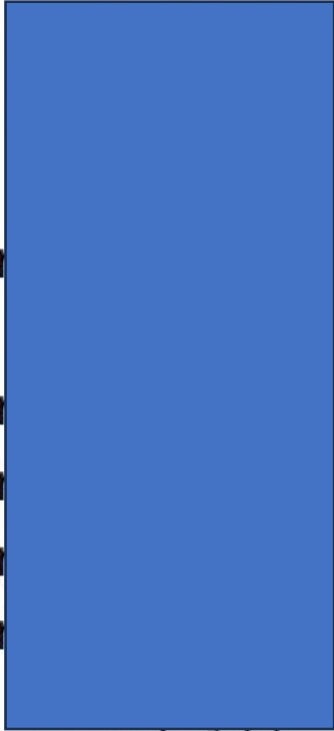
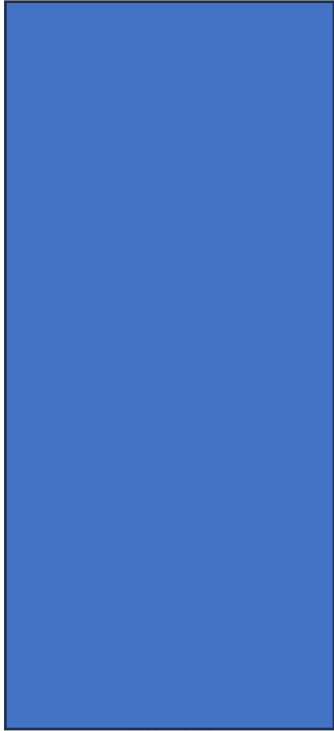
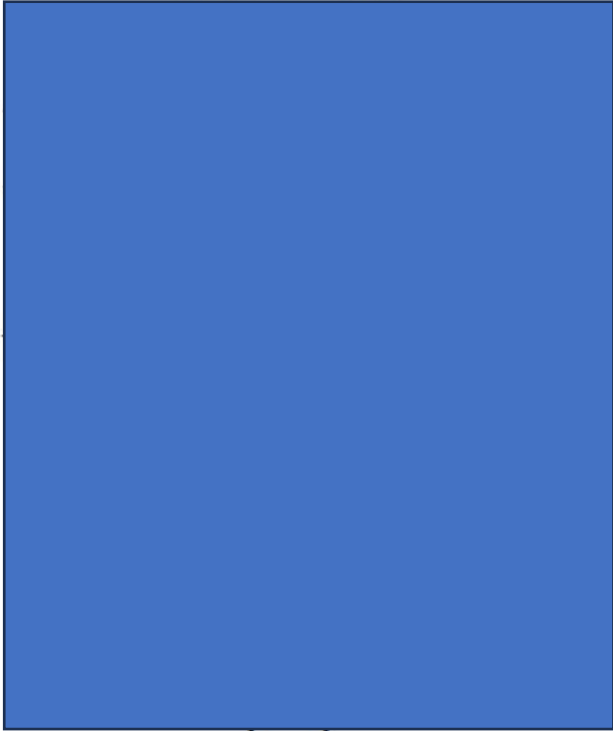
# Example - minPQ

Method	Return Value	Priority Queue Contents

# Example - minPQ

Method	Return Value	Priority Queue Contents
insert(5,A)		

# Example - minPQ

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
		

# Example - minPQ

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min()		{ (3,B), (5,A), (9,C) }
removeMin()		{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin()		{ (7,D), (9,C) }
removeMin()		{ (9,C) }
removeMin()		{ }
isEmpty()		

# Example - minPQ

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin()	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin()	(5,A)	{ (7,D), (9,C) }
removeMin()	(7,D)	{ (9,C) }
removeMin()	(9,C)	{ }
removeMin()	null	{ }
isEmpty()	true	{ }

# Priority Queues can be Min or Max

## Minimum Priority Queue vs Maximum Priority Queue

- Ascending vs Descending Order

`min()`: Returns (but does not remove) a priority queue entry  $(k,v)$  having minimal key; returns null if the priority queue is empty.

`removeMin()`: Removes and returns an entry  $(k,v)$  having minimal key from the priority queue; returns null if the priority queue is empty.

`poll`: `removeMin()` or `removeMax()`

`peek`: `min()` or `max()`



# Updating Key (Priority of an element)

What should happen when you change the key of an existing element in a heap?

What are the cases?


- increaseKey
- decreaseKey

# Priority Queue Implementations

# Ways to implement a priority queue

1. Heap
2. List
3. Sorted List

# Implementing a Priority Queue – Binary Heap

Method	Running Time
size, isEmpty	
min	
insert	
removeMin	

\*amortized, if using dynamic array

# Implementing a Priority Queue – Binary Heap

Method	Running Time
size, isEmpty	$O(1)$
min	$O(1)$
insert	$O(\log n)^*$
removeMin	$O(\log n)^*$

\*amortized, if using dynamic array

# Implementing a Priority Queue – **List**

insert(k, v)

- Add the new item to the end of the list

min():

- Search through all the elements and find the element with the smallest key

# Implementing a Priority Queue - List

insert(k, v)

- Add the new item to the end of the list

min():

- Search through all the elements and find the element with the smallest key

Method	Running Time
size	
isEmpty	
insert	
min	
removeMin	

# Implementing a Priority Queue - List

insert(k, v)

- Add the new item to the end of the list

min():

- Search through all the elements and find the element with the smallest key

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
insert	$O(1)$
min	$O(n)$
removeMin	$O(n)$



# Implementing a Priority Queue - **SortedList**

insert(k, v)

- Find where to put the item based on k, then move other items over

min():

- Find the first element in the list

# Implementing a Priority Queue - SortedList

Method	Unsorted List	Sorted List
size	$O(1)$	
isEmpty	$O(1)$	
insert	$O(1)$	
min	$O(n)$	
removeMin	$O(n)$	

# Implementing a Priority Queue - SortedList

Method	Unsorted List	Sorted List
size	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
removeMin	$O(n)$	$O(1)$

# Implementing a Priority Queue

Method	Unsorted List	Sorted List	Binary Heap
size	$O(1)$	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$	$O(\log n)^*$
min	$O(n)$	$O(1)$	$O(1)$
removeMin	$O(n)$	$O(1)$	$O(\log n)^*$

# Summary

1. Heaps
  - a. DS for efficient removal of the min element
2. Selection Sort
  - a.  $O(n^2)$  sort algorithm
3. Heap Sort
  - a.  $O(n \log n)$  sort algorithm
4. BFS
  - a. top-to-bottom left-to-right traversal
  - b. implemented with a queue
5. Priority Queue
  - a. removal order based on a “priority”
  - b. efficiently implemented with a heap