CS151 Intro to Data Structures

Heaps & Priority Queues

Announcements

Quiz next week

Anonymous Course survey (5 bonus points on exam):

https://forms.gle/QRXSyZt9N5rT4iDf9

Outline

- Heap Review
- Breadth First Traversal
- Priority Queues
- Efficient heap construction

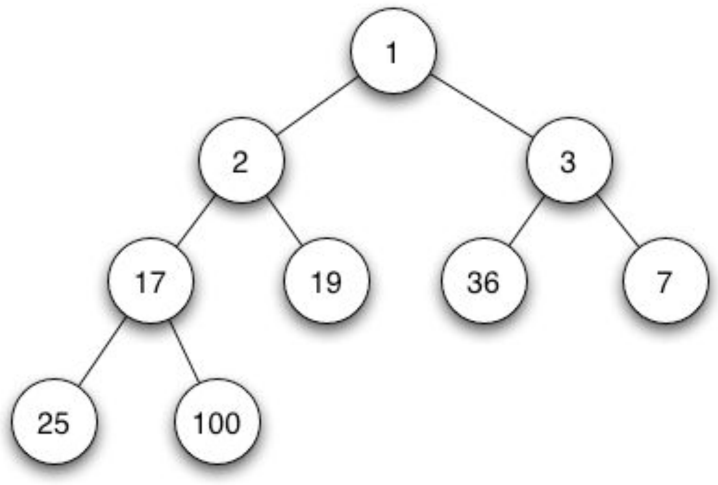
Heap Review

Binary Heap Properties

1. Each node has **at most 2 children**

2. For every node n (except for the root): **n.key >= parent(n).key**

3. **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**

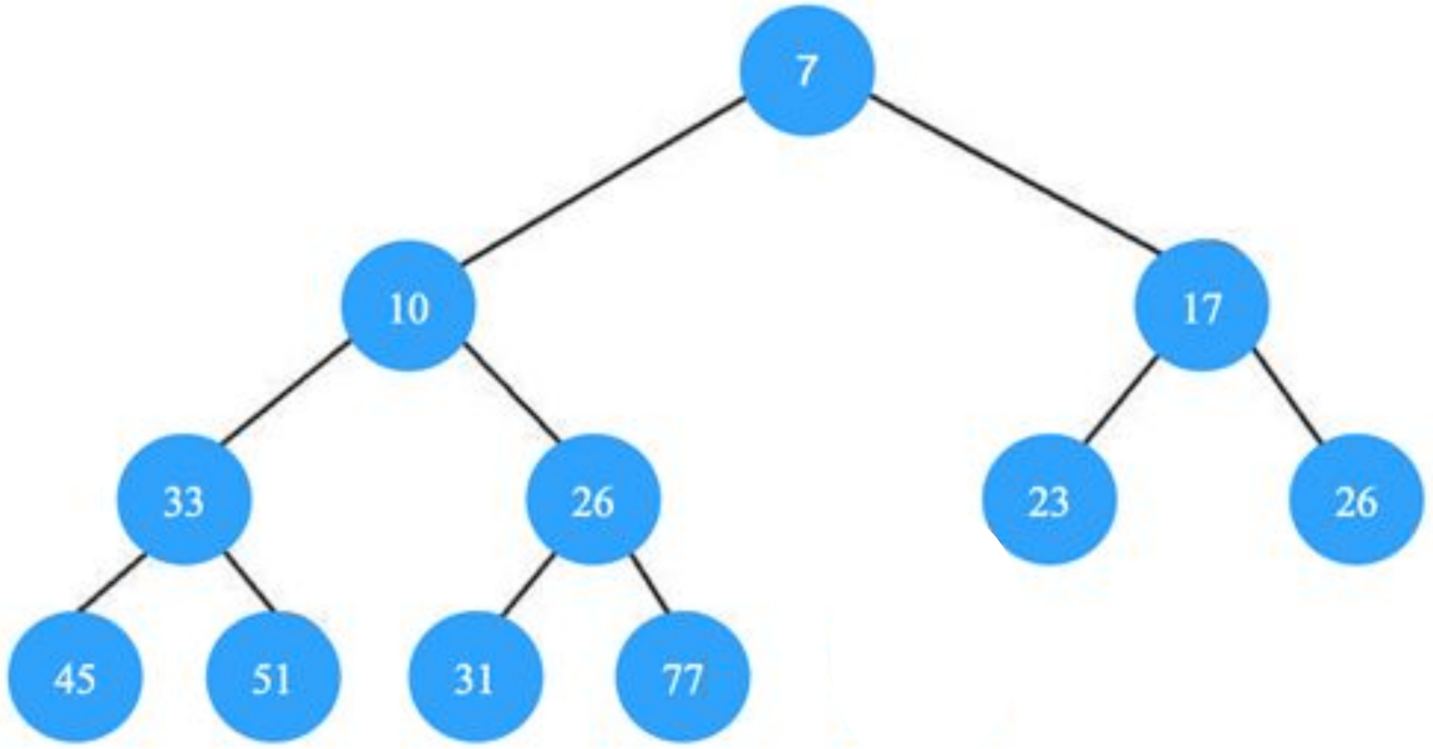
Heap **Remove**

More general case of poll

1. Find the index i of the element we want to delete
2. Swap this element with the last element (rightmost leaf)
3. Downheap to restore the heap property

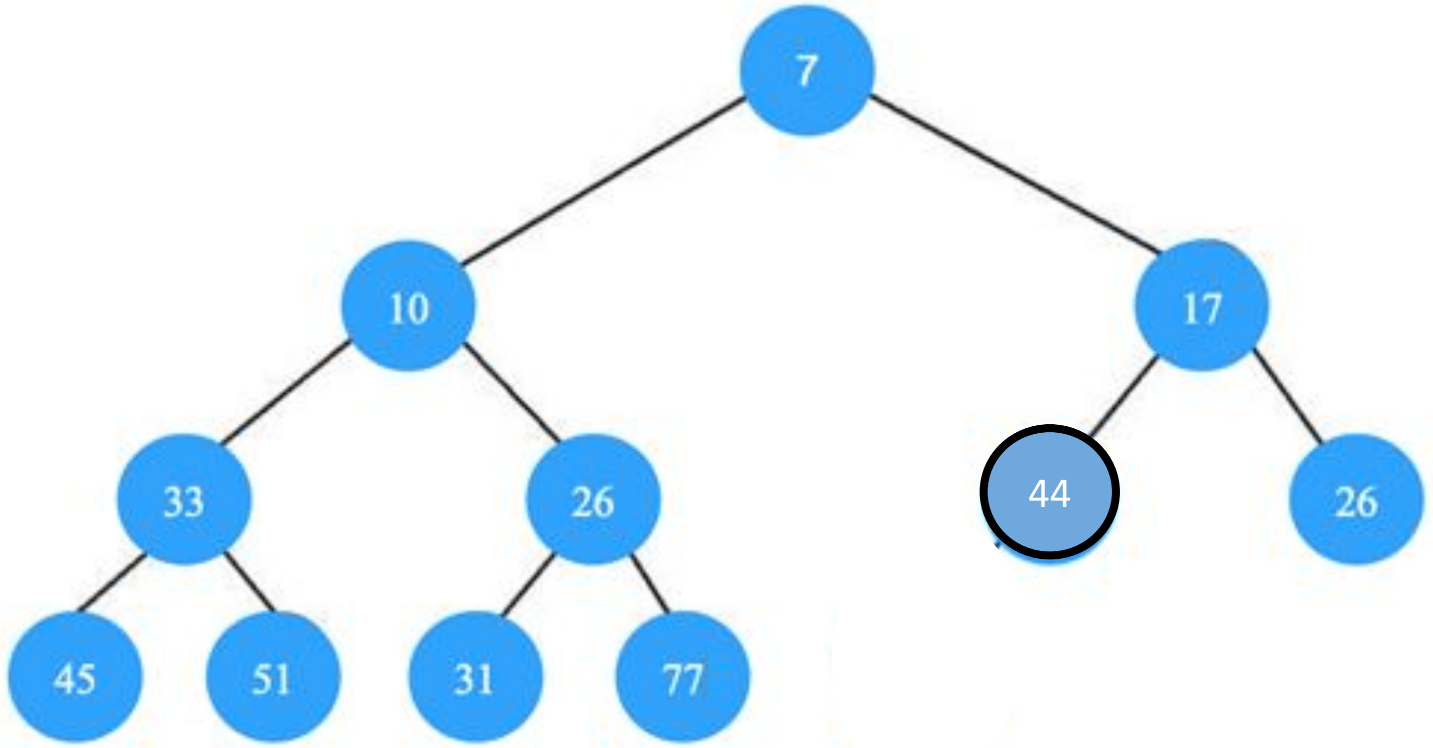
Heap Remove: Example 1

`remove(44)`



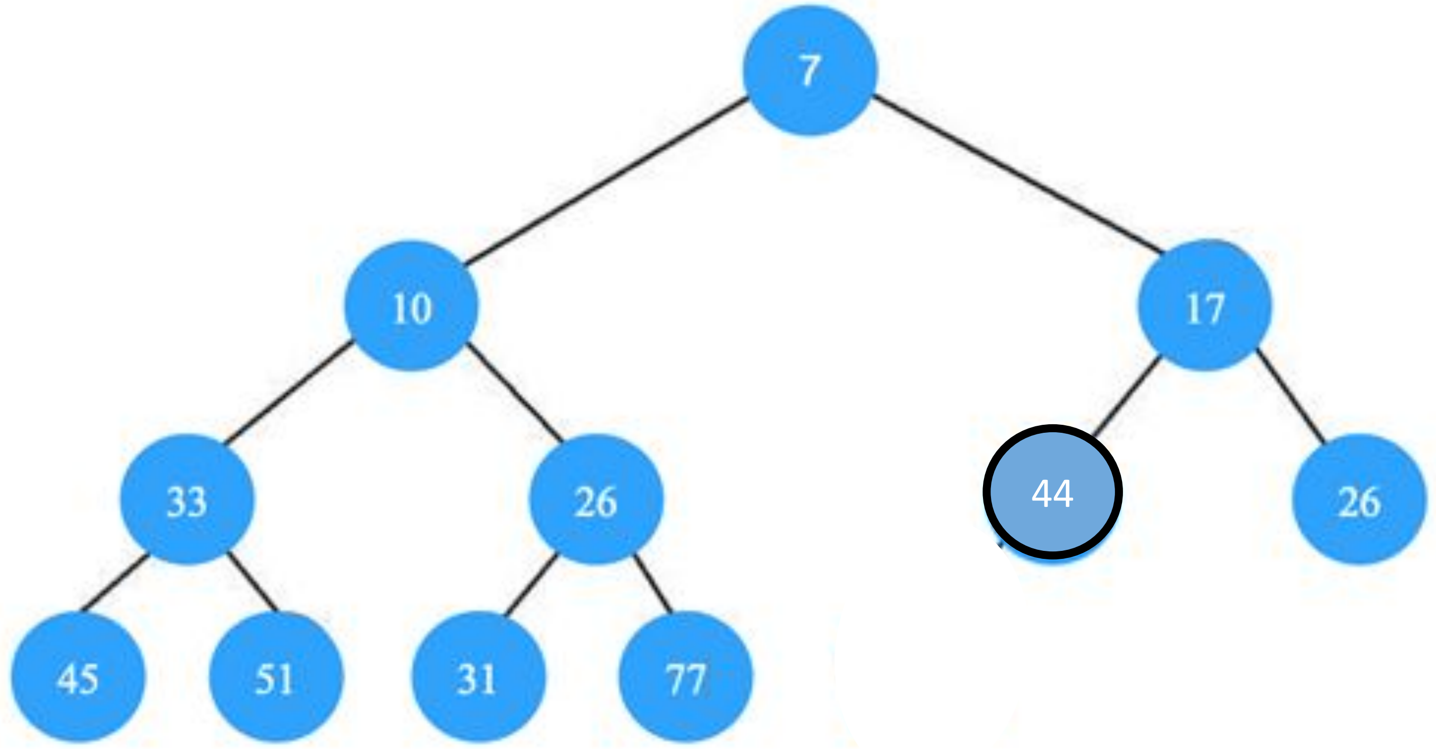
Heap Remove: Example 2

`remove(23)`



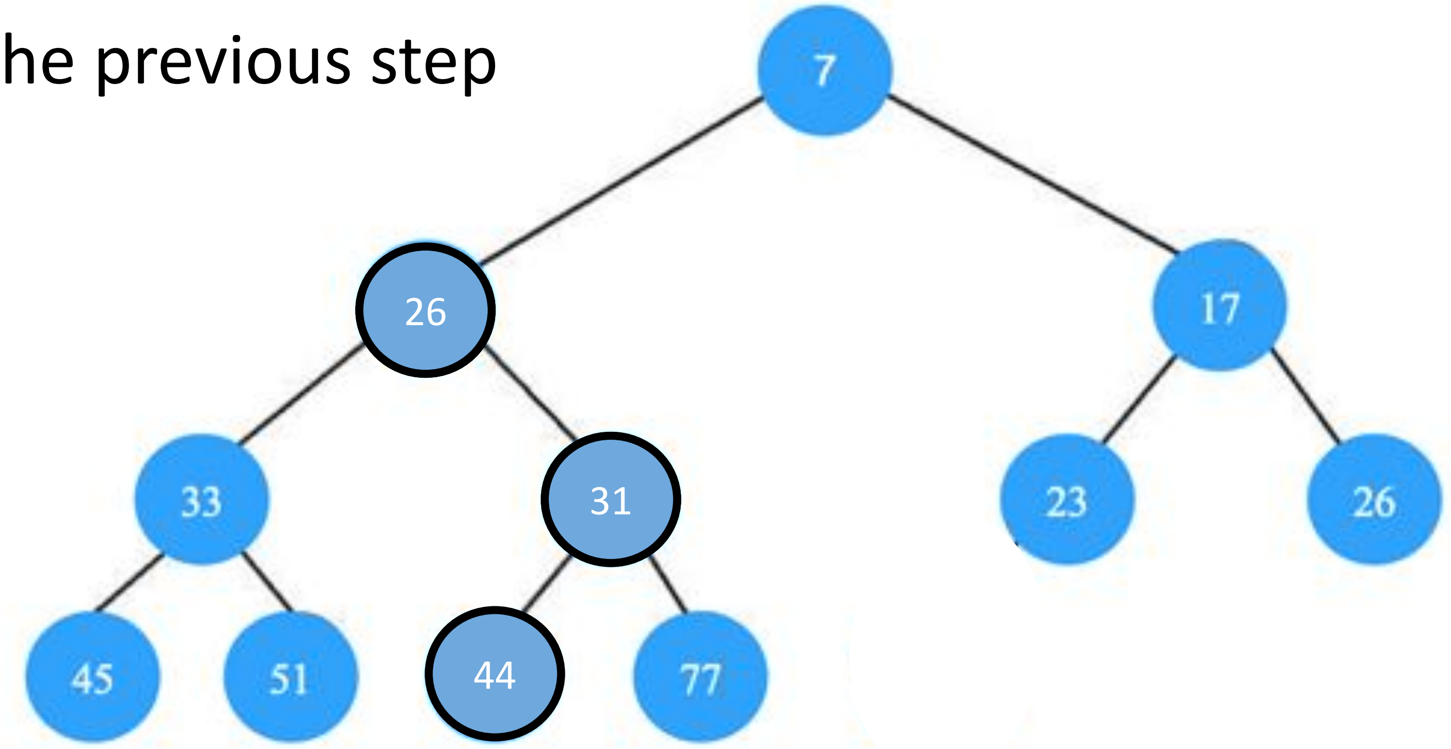
Heap Remove: Example 2

`remove(23)`



Downheap:

1. Compare the new root with its children; if they are in the correct order, stop.
2. If not, swap the element with one of its smallest children and return to the previous step



`remove(10)`

Heap **Remove**

More general case of poll

1. Find the index i of the element we want to delete
2. Swap this element with the last element
3. Downheap to restore the heap property

Runtime complexity?

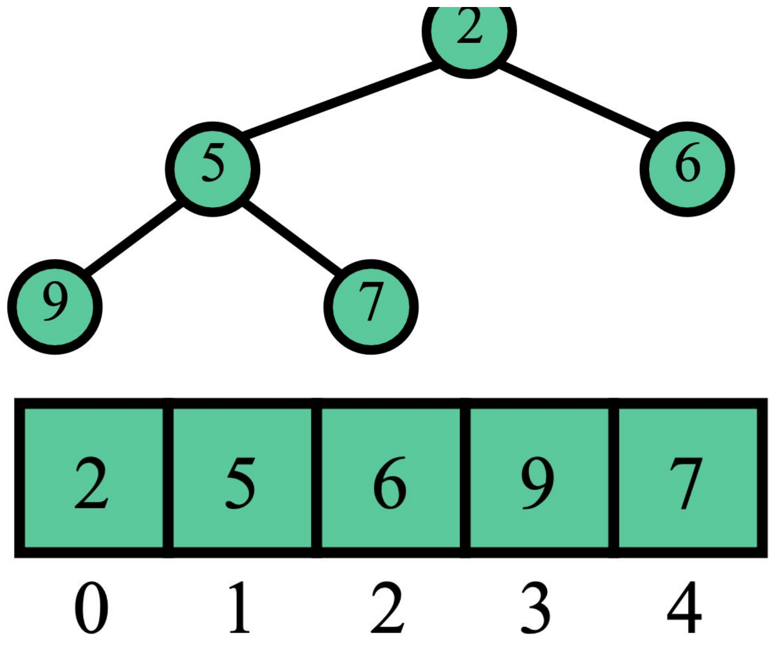
O(n + logn) = O(n)

Array based heap

Array/ArrayList of length n for heap with n keys

Node at index i

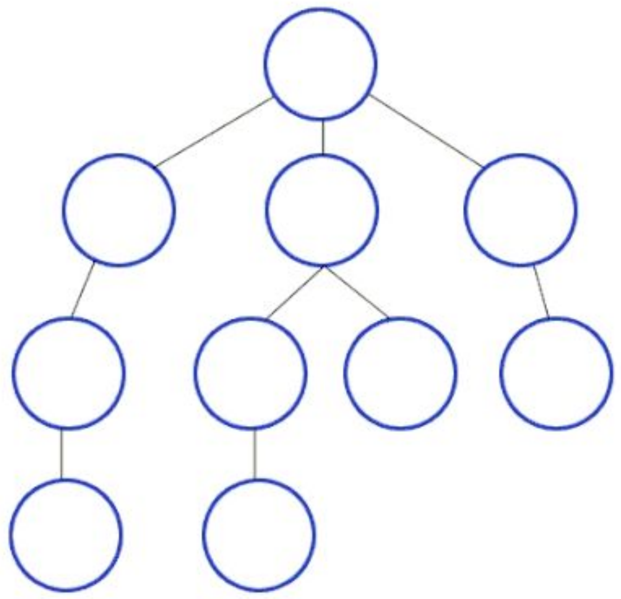
- Left child index:
  - 2i + 1
- Right child index:
  - 2i + 2



Breadth First Search

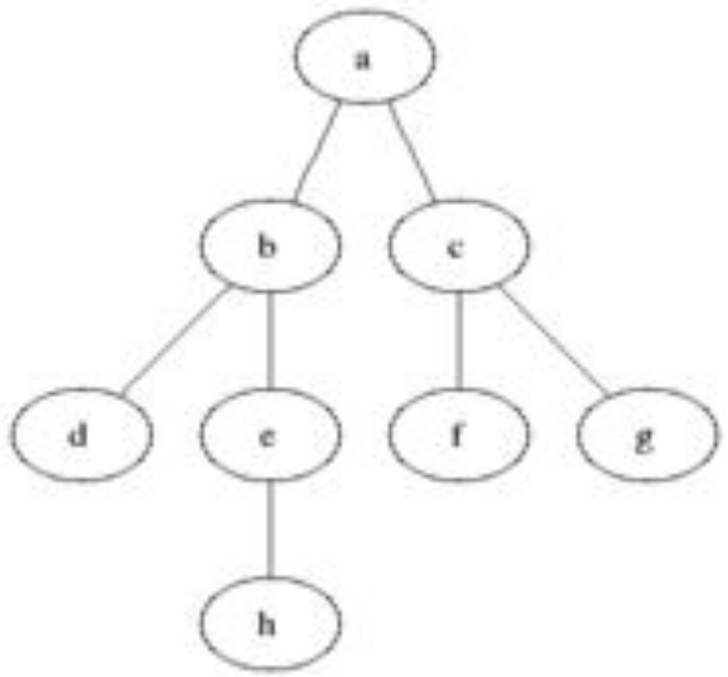
Depth First Search (DFS)

- 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)

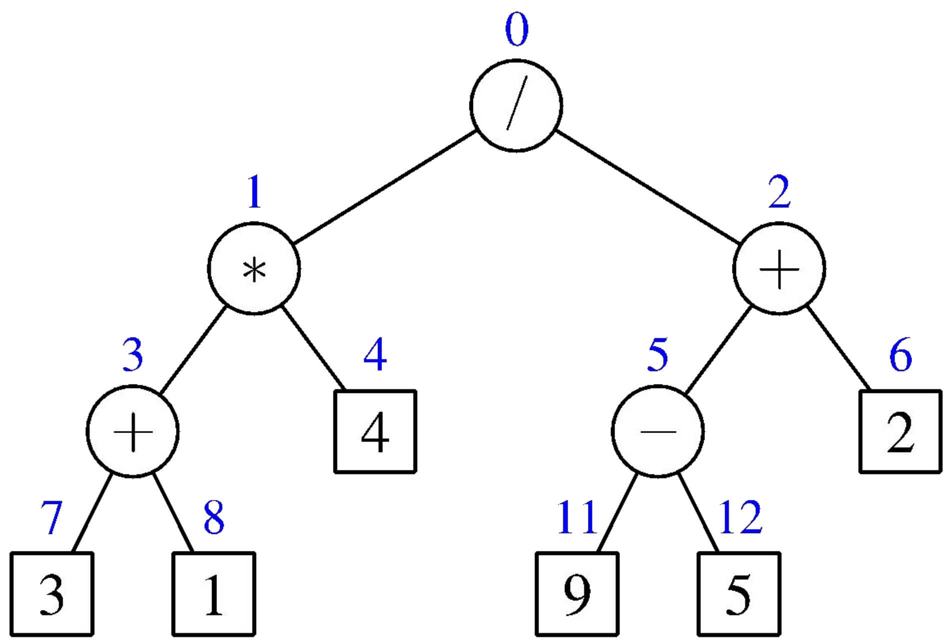
- Breadth First Search (BFS)
  - Starts at the root and explores all nodes at the present “depth” before moving to nodes on the next level
  - Extra memory is usually required to keep track of the nodes that have not yet been explored



Breadth-First Traversal

Traverse the tree level-by-level

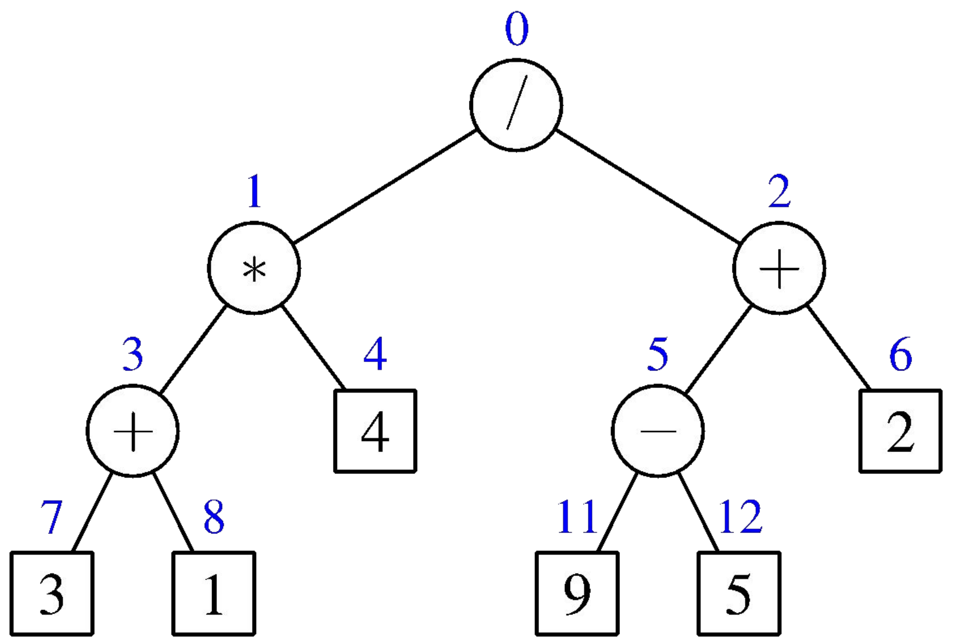
- Within a level go left-to-right

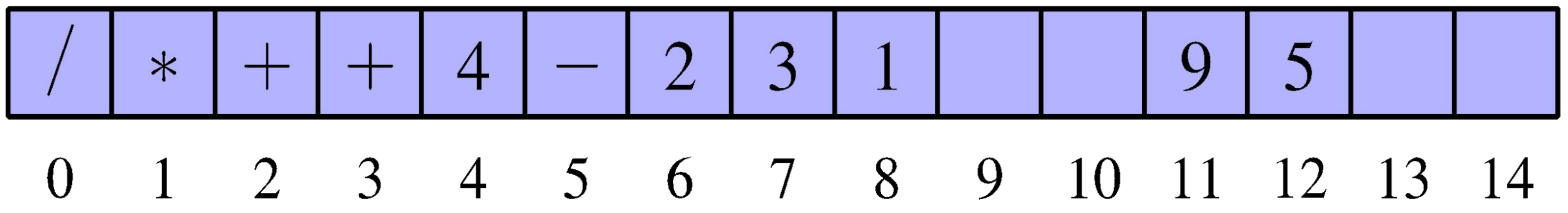


Breadth-First Traversal

Traverse the tree level-by-level

- Within a level go left-to-right

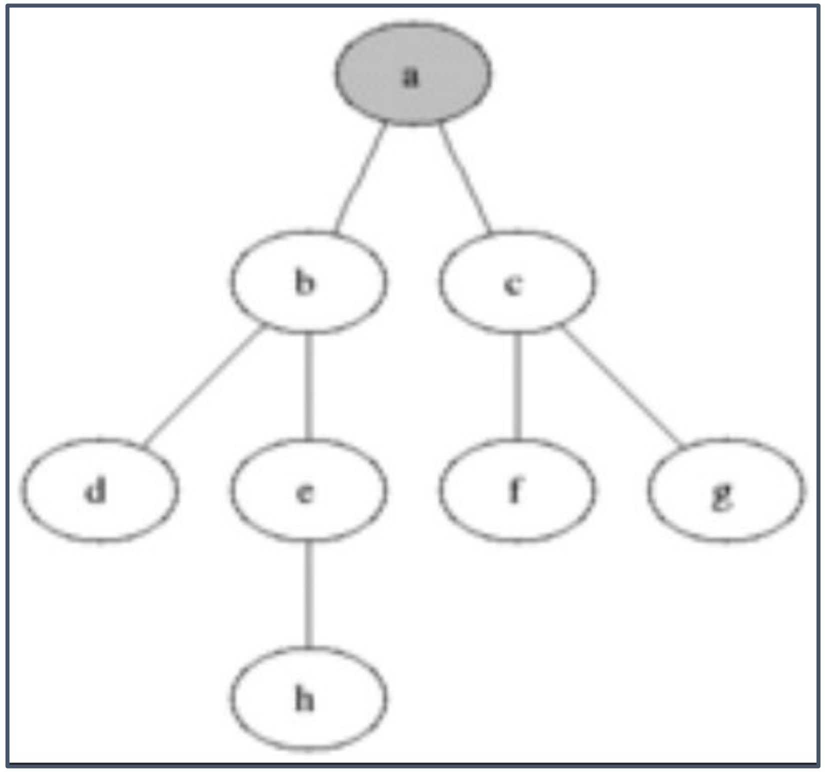


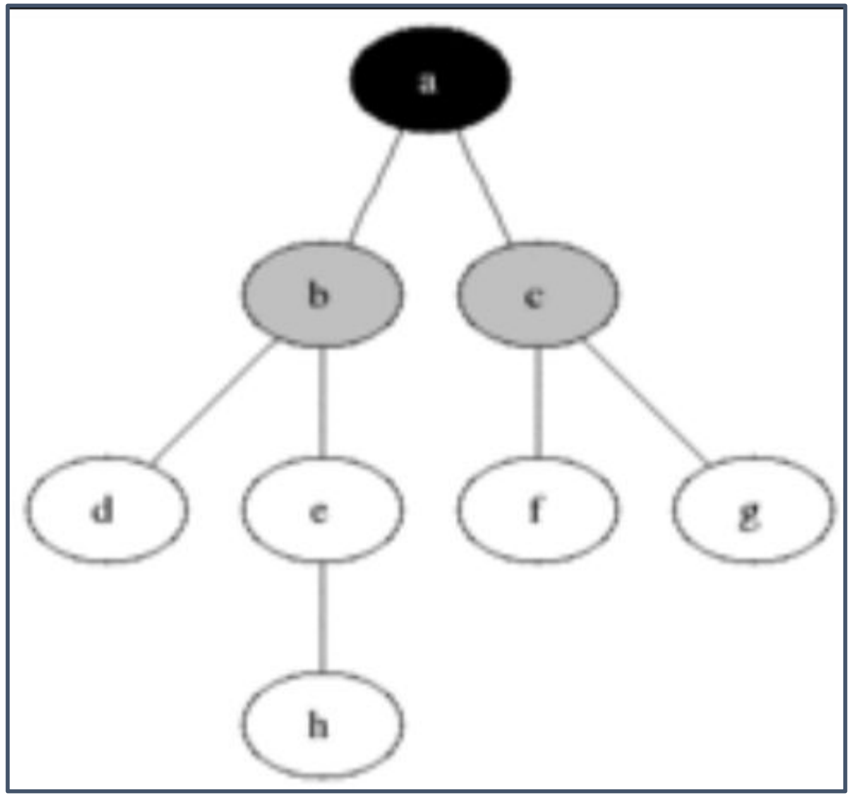


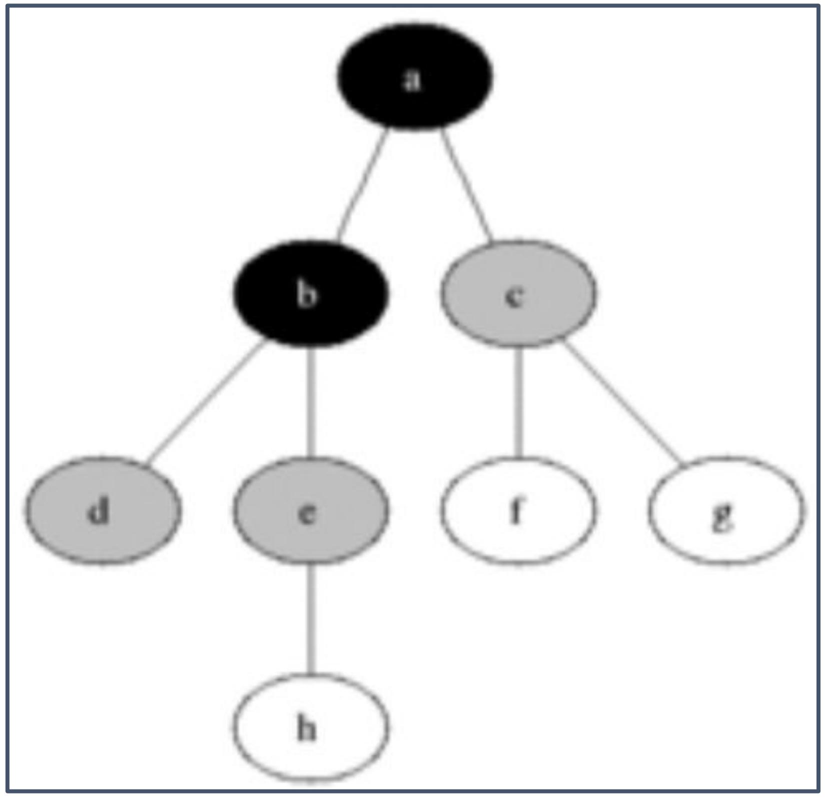
This is the array order of an array-based binary tree

Breadth-First Traversal

pseudo-code?

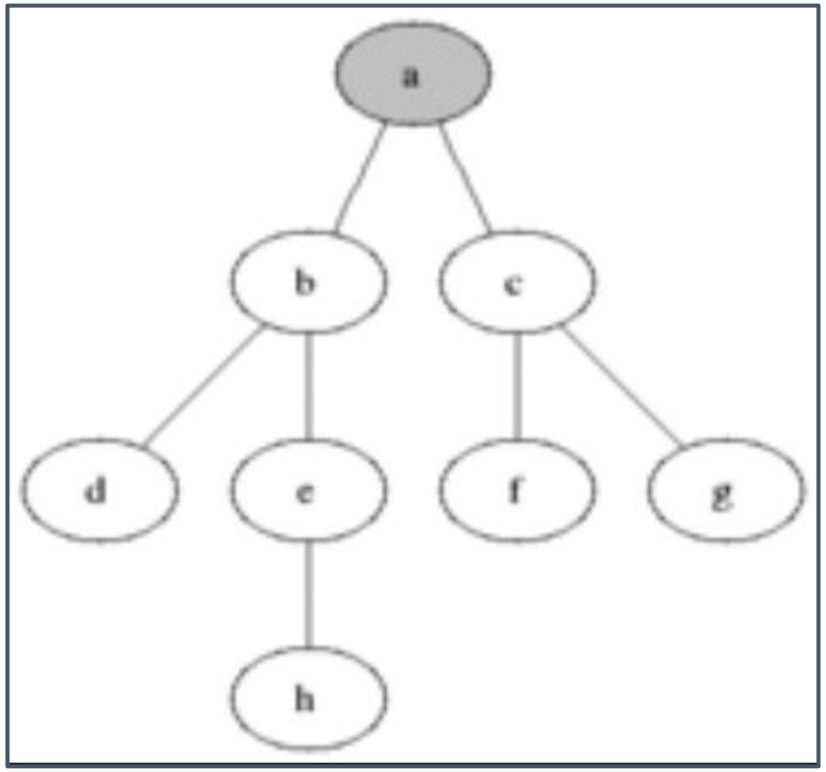


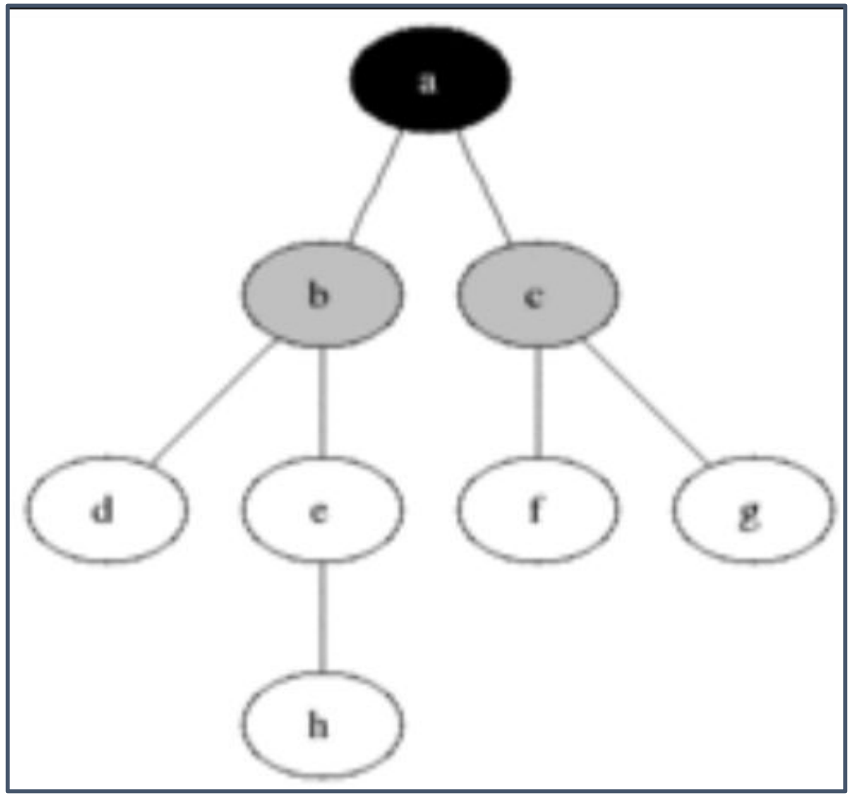


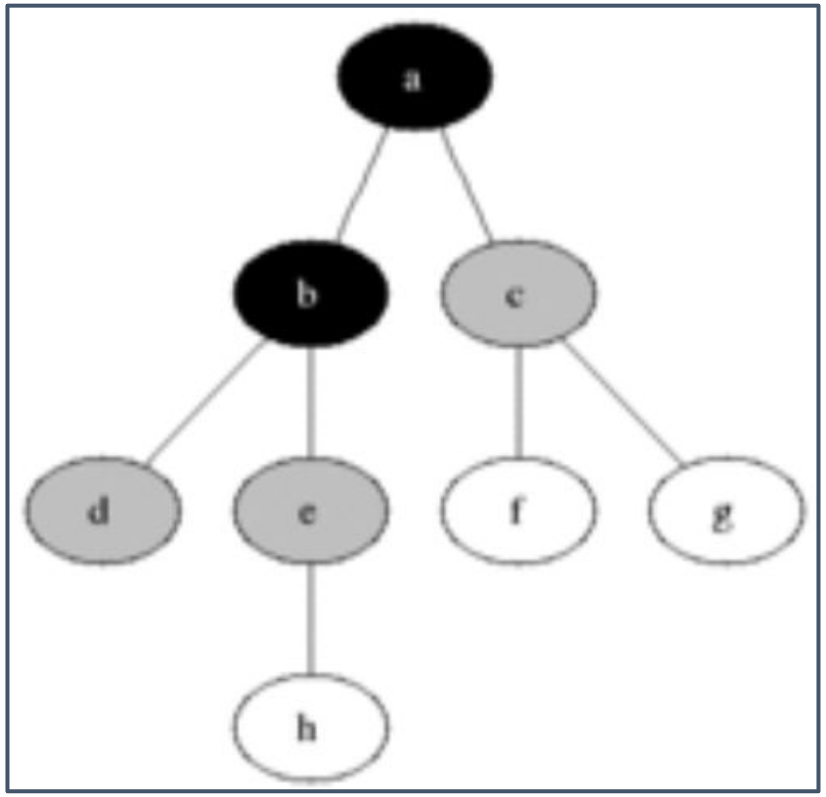


Breadth-First **Search**

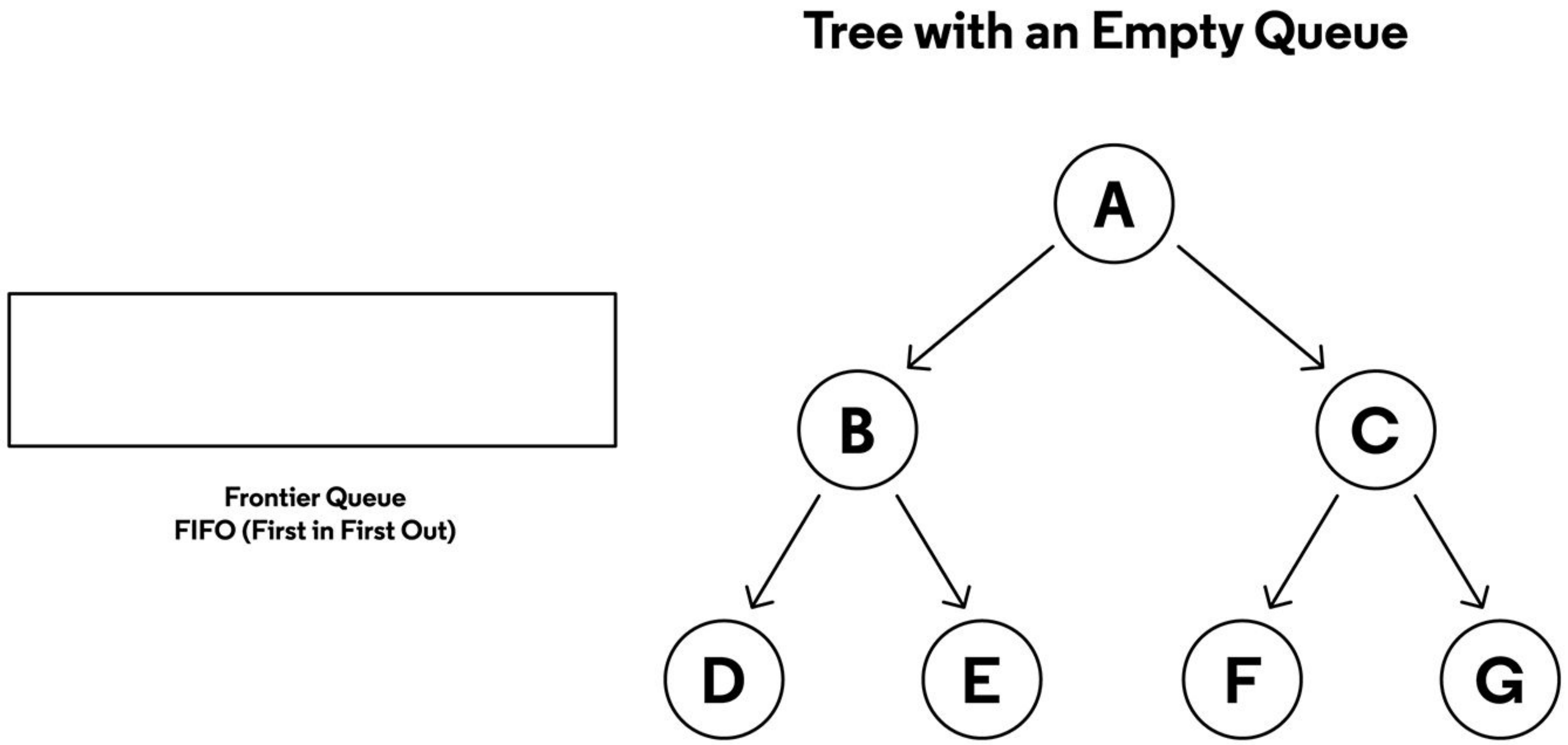
How would we change the pseudo-code?







Breadth First Search (BFS)



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

Priority Queue

Priority Queues

- What is a queue?

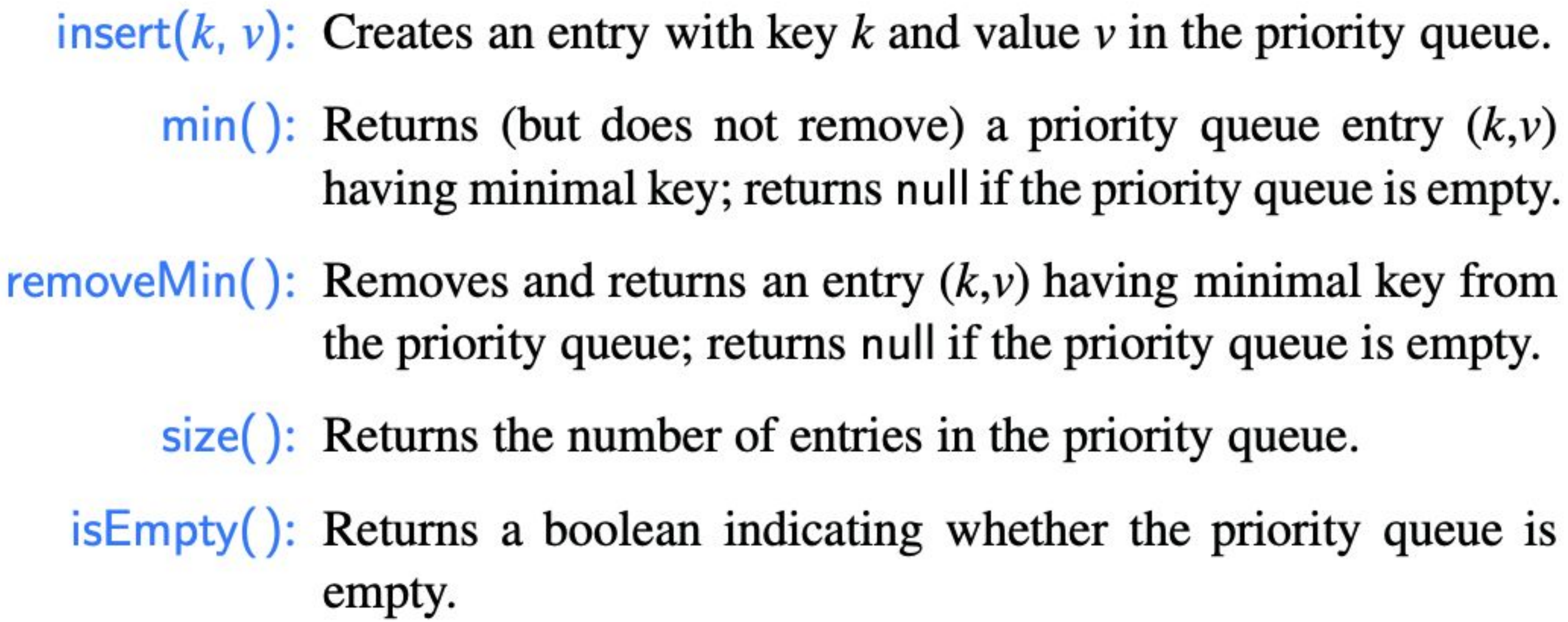
- What if we want to create a graduation queue of students who pick up their diplomas in alphabetical order?
    - Queues and Stacks removal is based on when it was added to the data structure
    - Instead we want removal order to be based on the student’s name

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

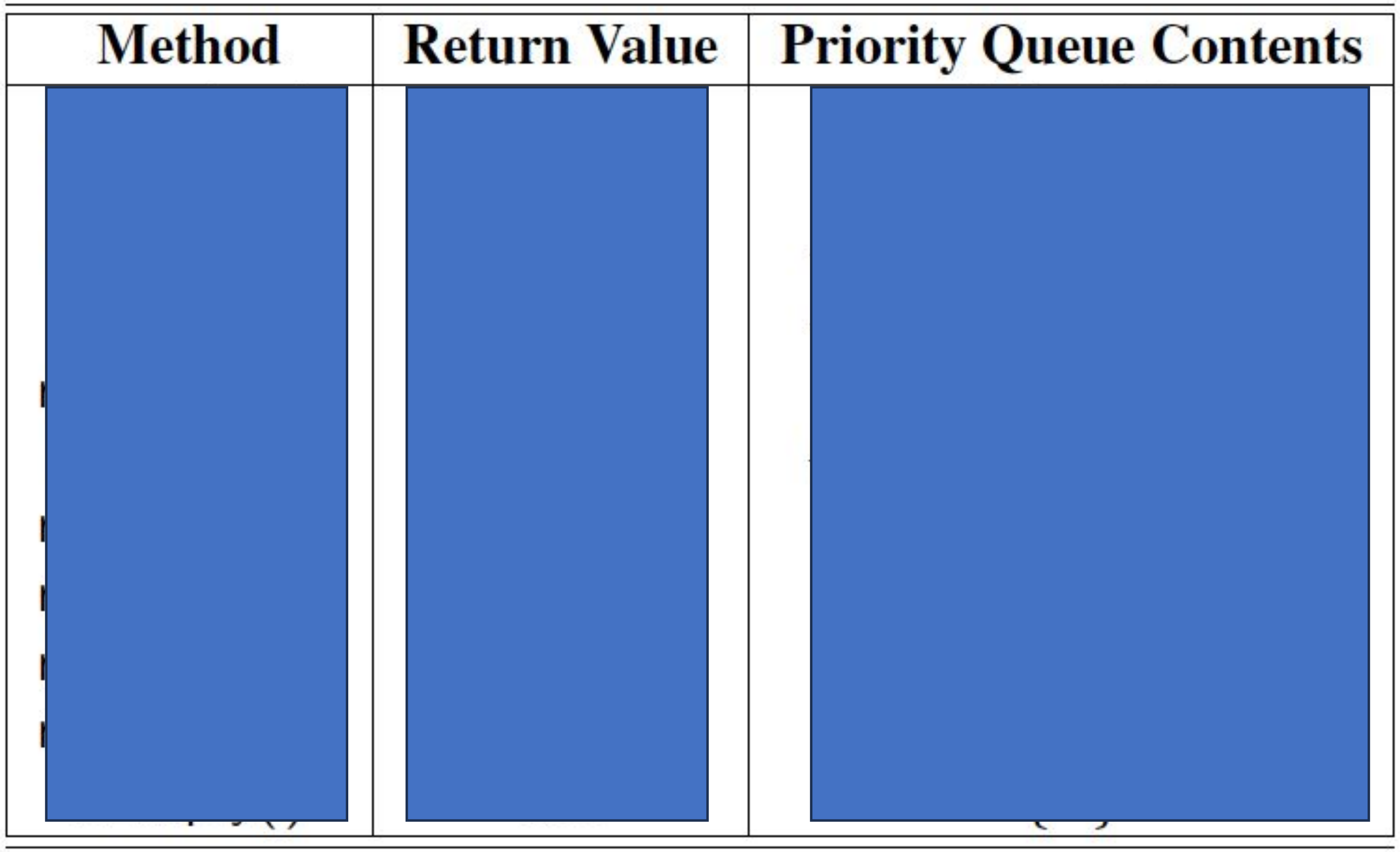
The key represents the priority

Key type must be comparable

Entry Interface

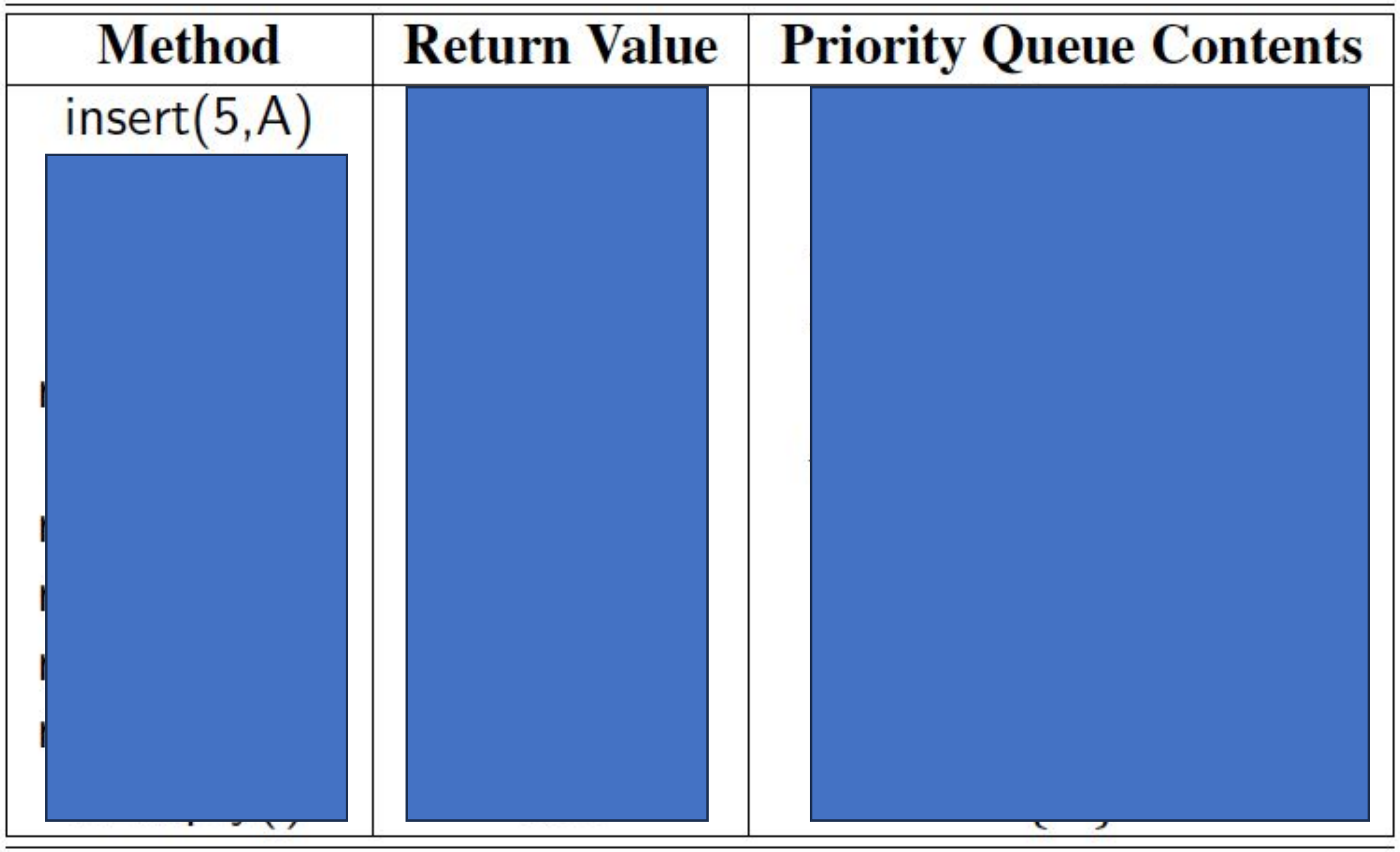
```
public interface Entry<K extends Comparable<K>, V> {

  K getKey();
  V getValue();


}
```

Example - minPQ



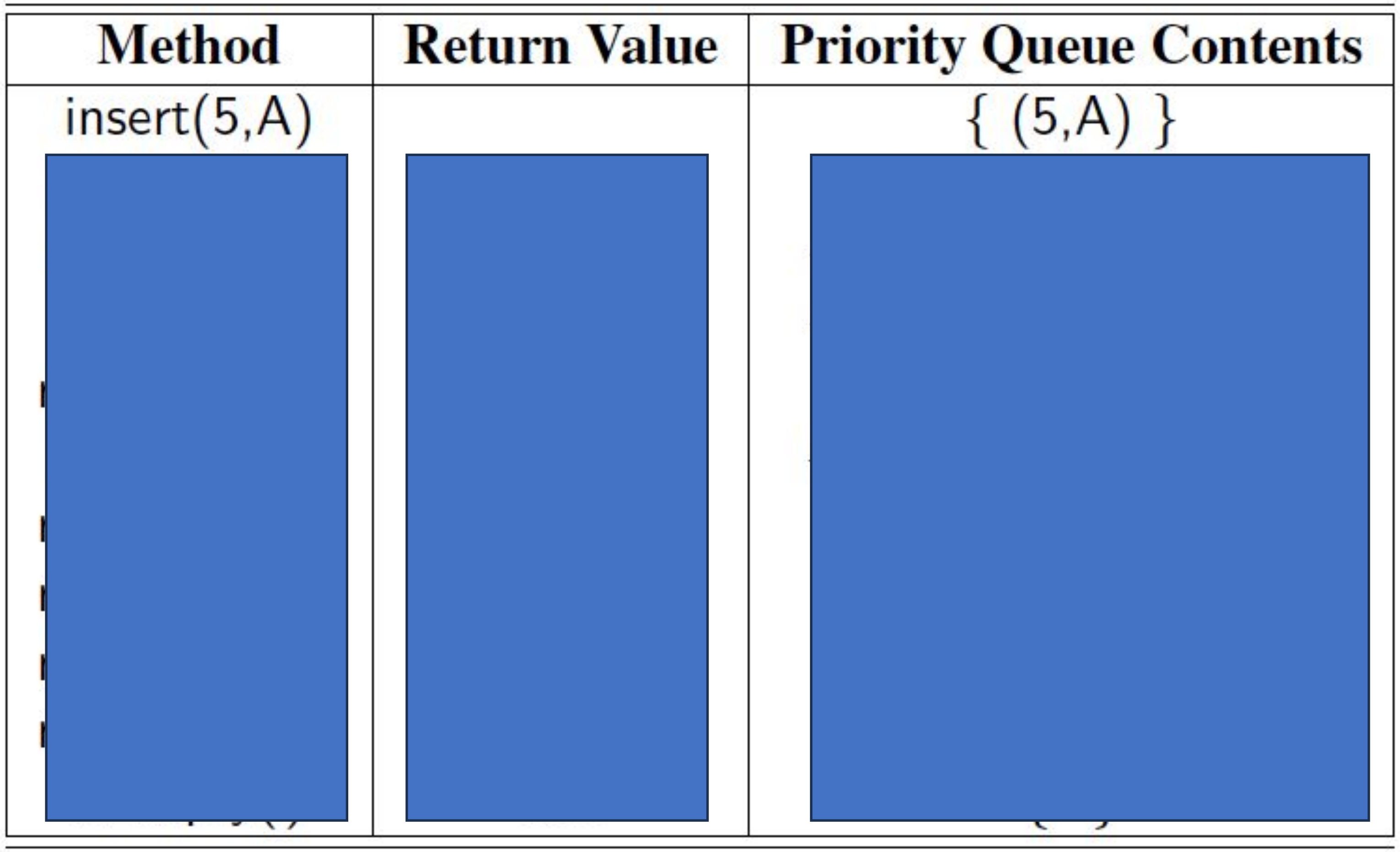
Method	Return Value	Priority Queue Contents

Example - minPQ



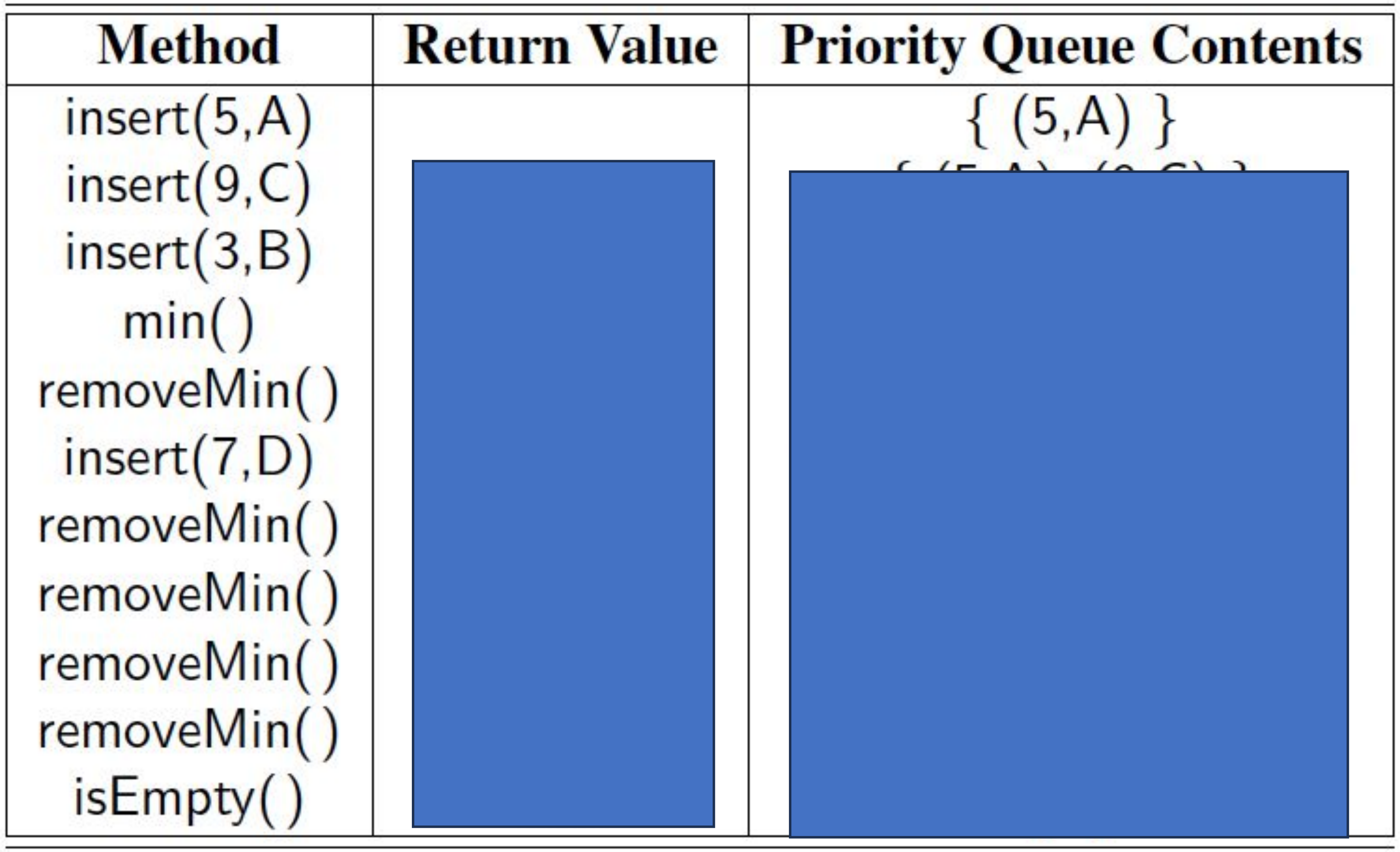
Method	Return Value	Priority Queue Contents
insert(5,A)		

Example - minPQ



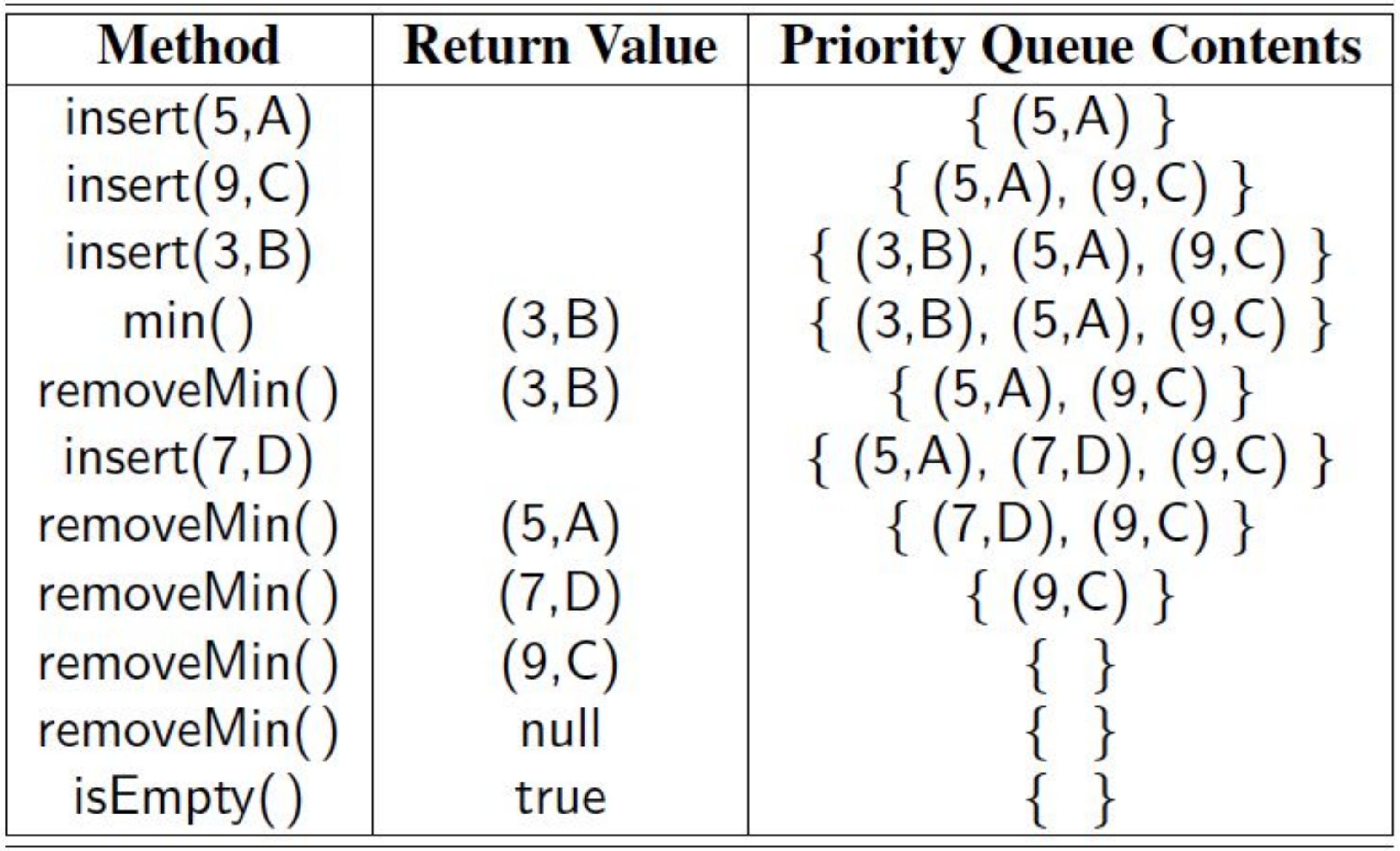
Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
[redacted]	[redacted]	[redacted]

Example - minPQ



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

Example - minPQ

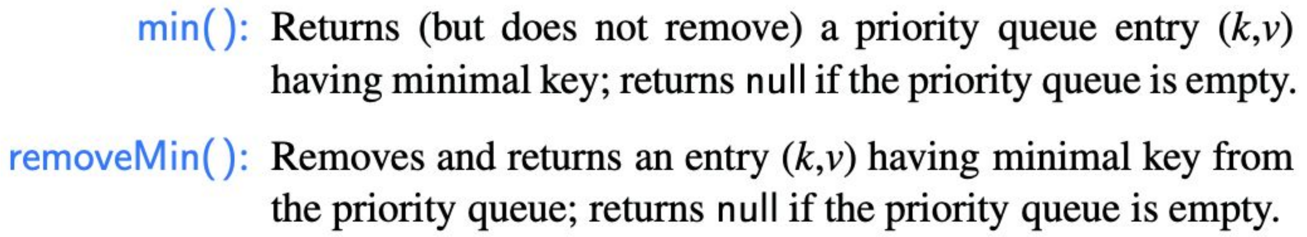


Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{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

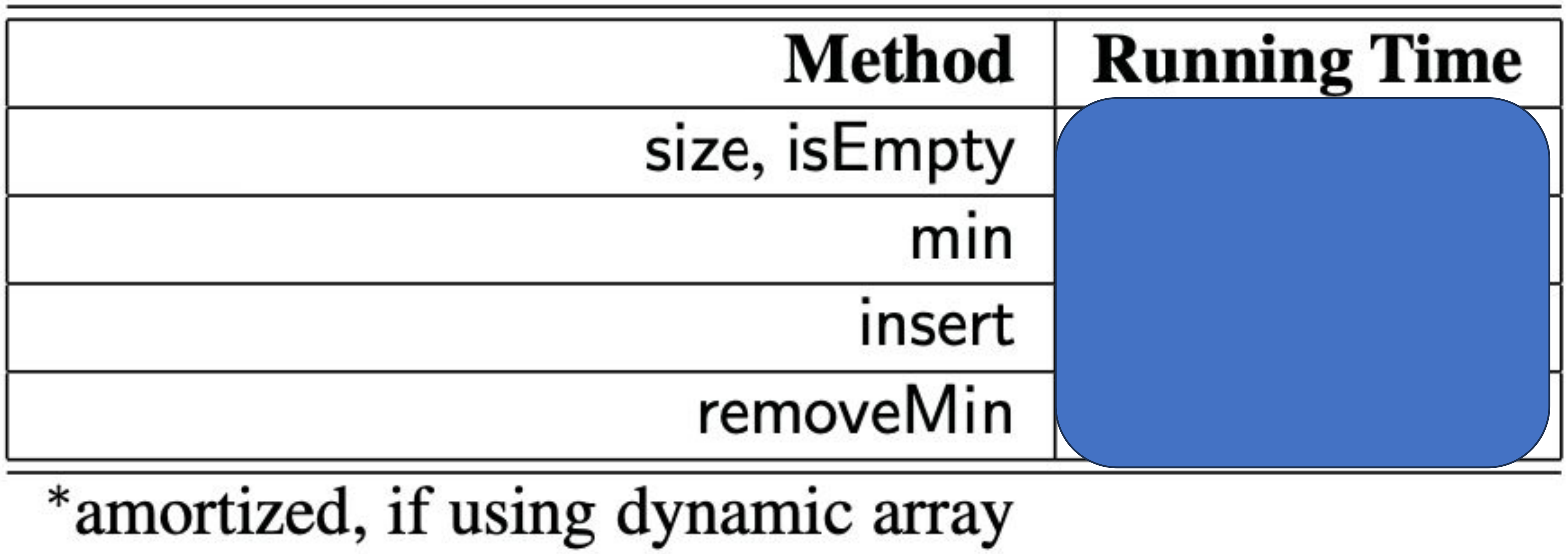
- increaseKey
- decreaseKey

Priority Queue Implementations

Ways to implement a priority queue

1. Heap
2. List
3. Sorted List

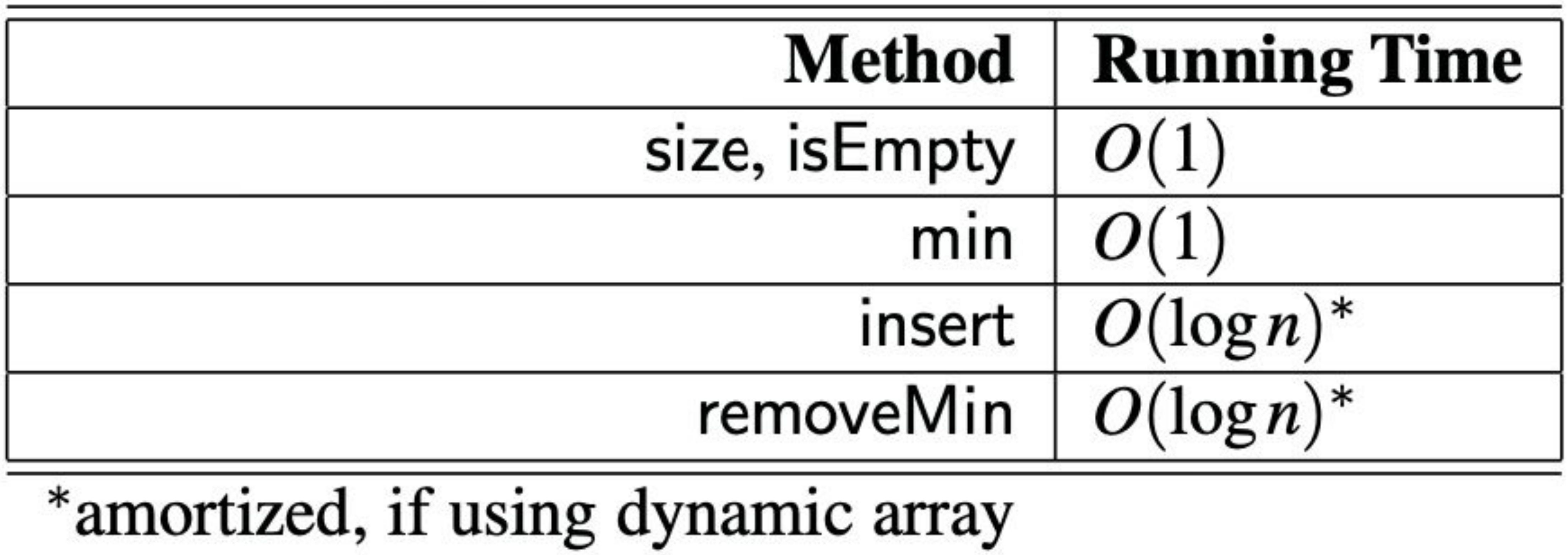
Implementing a Priority Queue – Binary Heap



Method	Running Time
size, isEmpty	[redacted]
min	[redacted]
insert	[redacted]
removeMin	[redacted]

*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

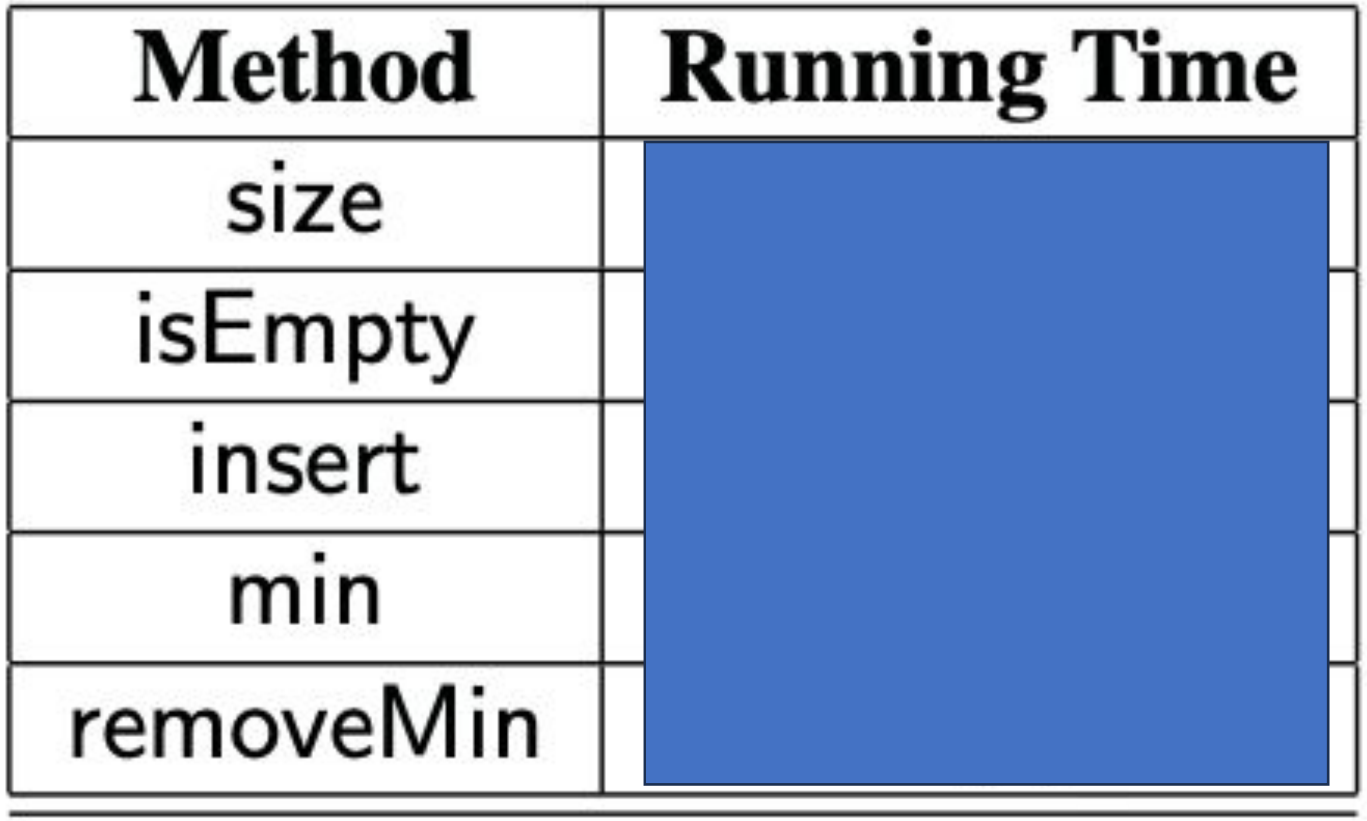
Implementing a Priority Queue - List

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	[redacted]
isEmpty	[redacted]
insert	[redacted]
min	[redacted]
removeMin	[redacted]

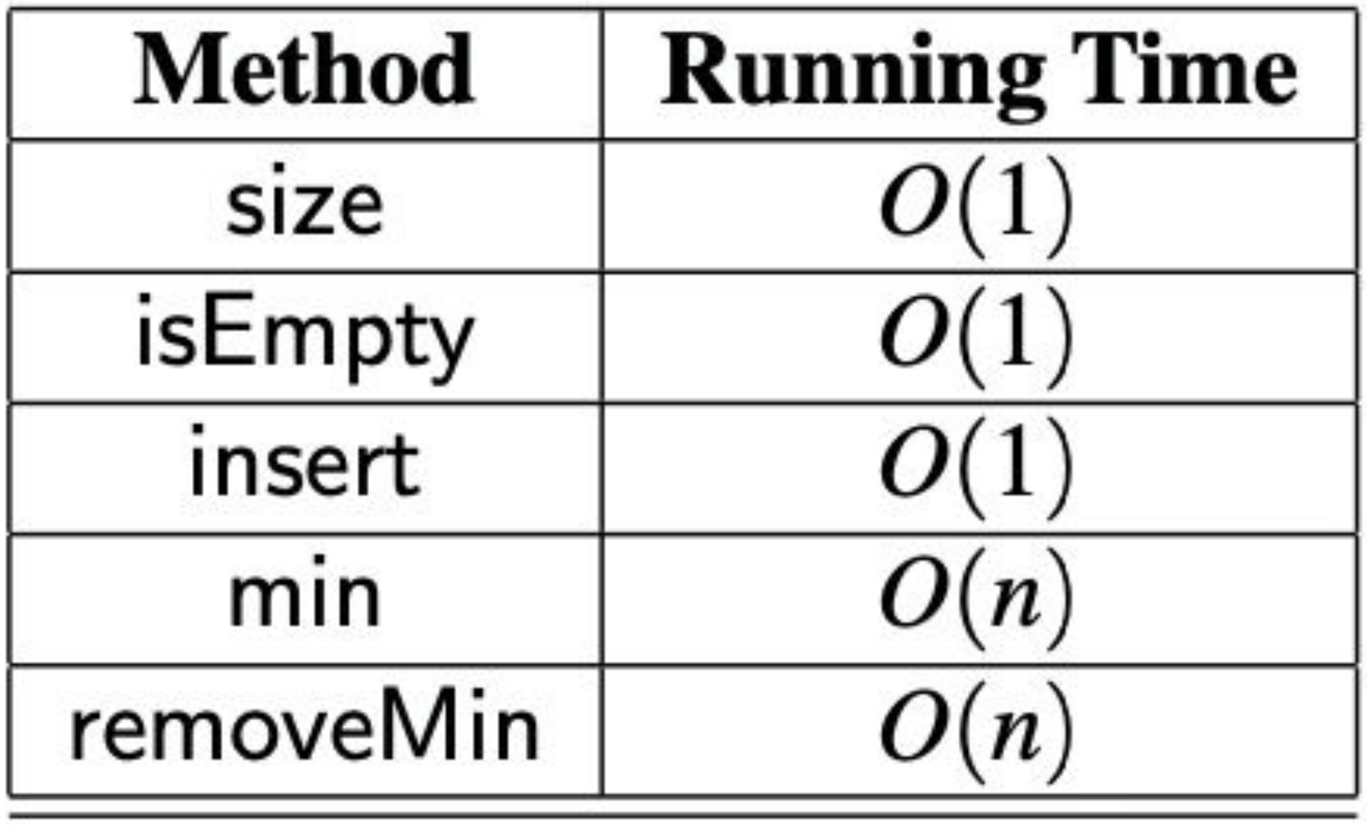
Implementing a Priority Queue - List

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**

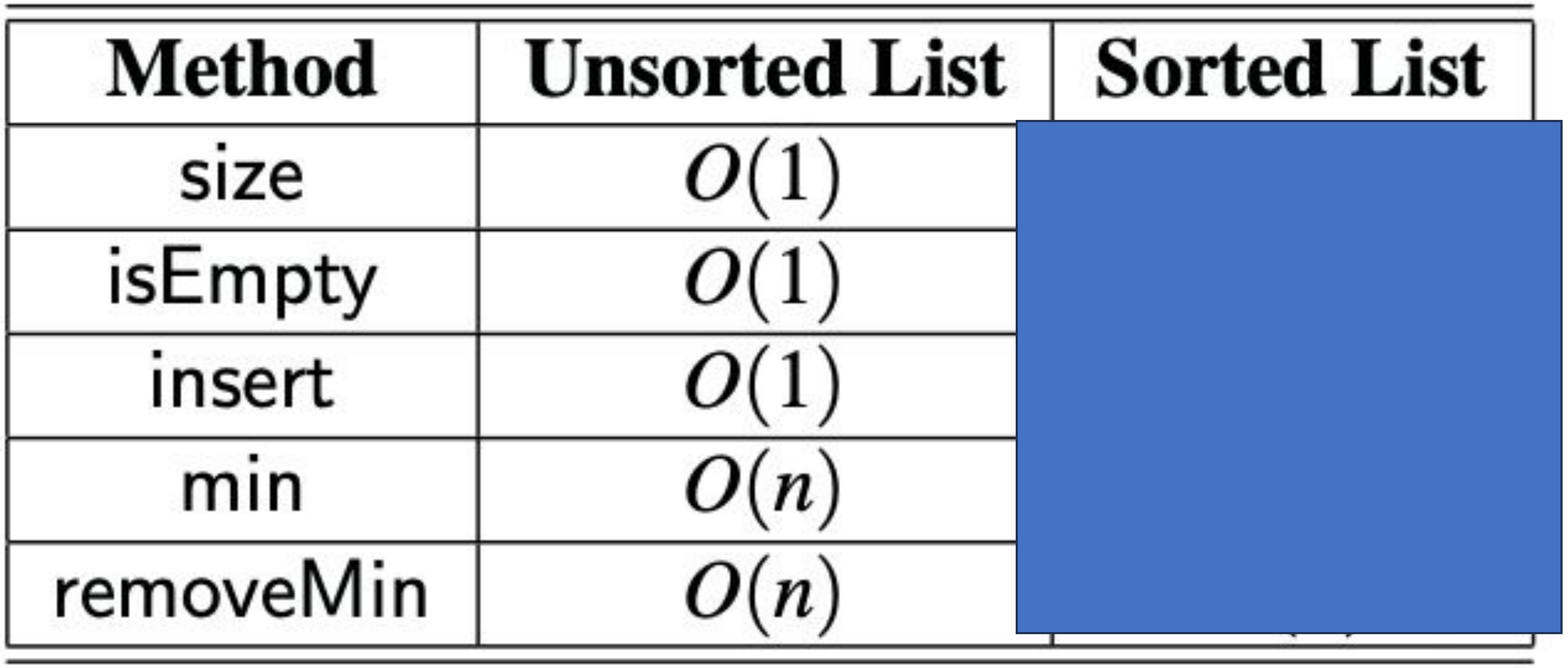
insert(k, v)

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

min():

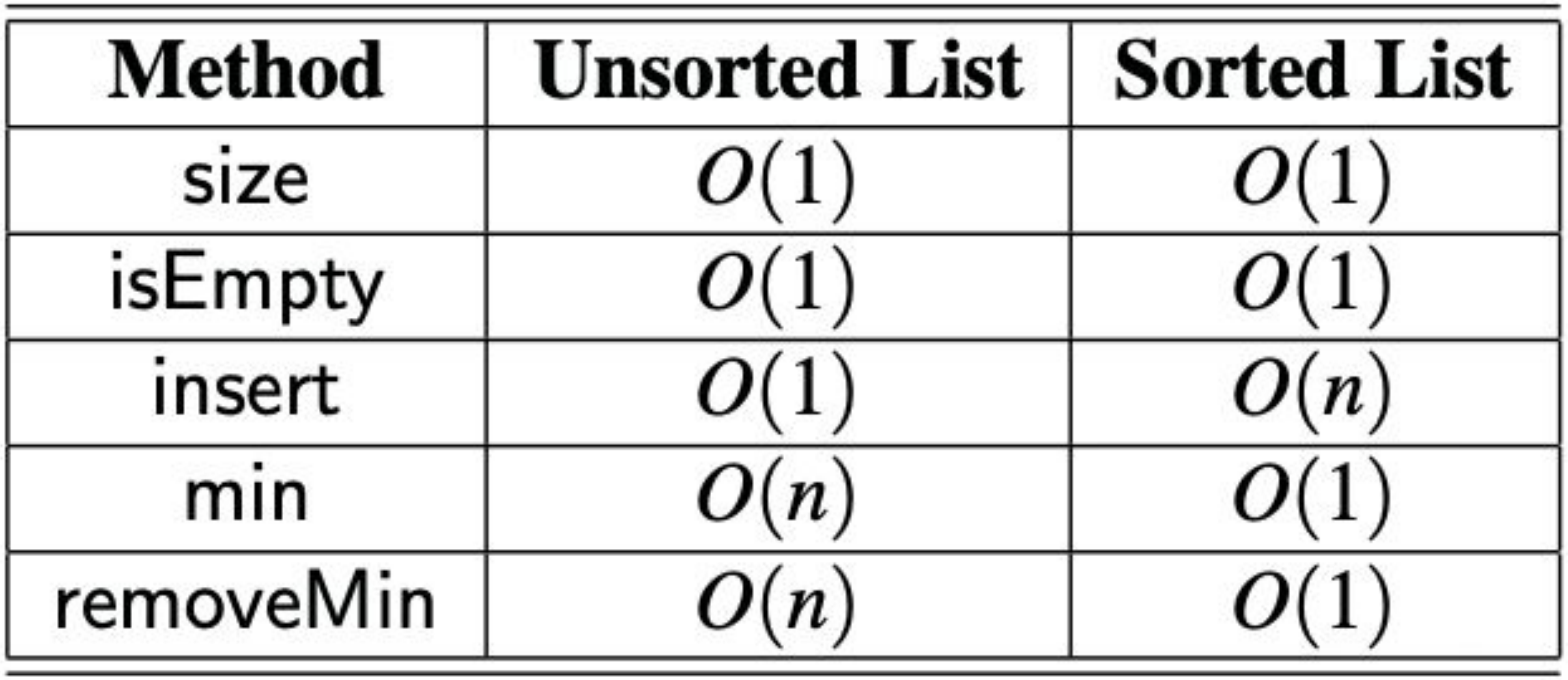
- Find the first element in the list

Implementing a Priority Queue - SortedList



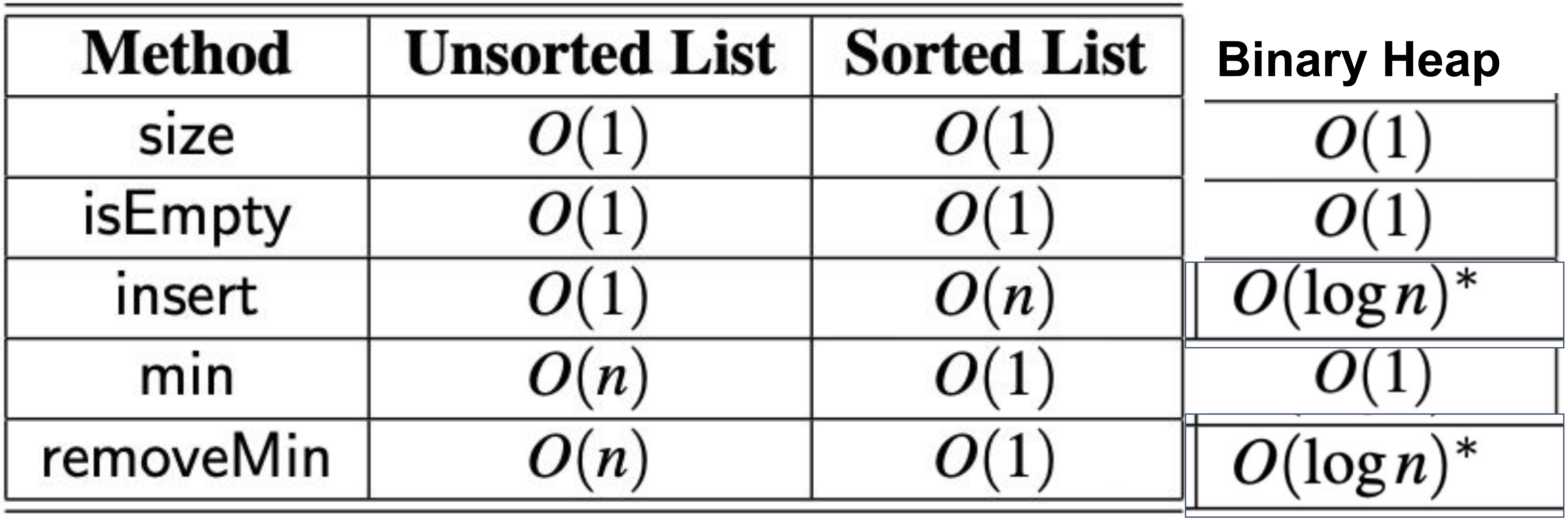
Method	Unsorted List	Sorted List
size	$O(1)$	[redacted]
isEmpty	$O(1)$	[redacted]
insert	$O(1)$	[redacted]
min	$O(n)$	[redacted]
removeMin	$O(n)$	[redacted]

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	$O(n)$	$O(1)$	$O(\log n)^*$

Efficient Heap Construction

Bottom Up Heap Construction

Given all elements to create a heap from:

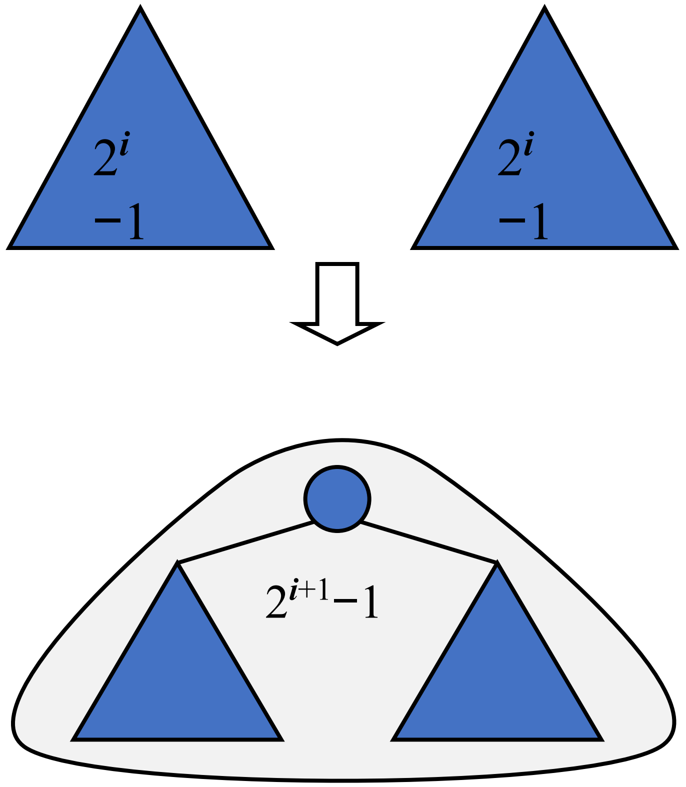
- Step 1: Fill the bottom level
- Step 2: Fill the level above it and fix heap properties
- Step 3: repeat

Example

Construct a heap for the list 2, 9, 7, 6, 5, 8, 3

Bottom-up Heap Construction

- We can construct a heap storing n given keys using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys



Heap Construction

- Direct Approach:
  - Add each element to the left-most leaf and upheap
  - advantages?
  - disadvantages?
- Bottom Up Approach:
  - construct `n` ***elementary heaps*** storing one entry each and **merge heaps pairwise**
  - advantages?
  - disadvantages?