CSCI 136: Data Structures and Advanced Programming

Lecture 31

Heaps

Instructor: Dan Barowy

Williams

Topics

Connectedness

Priority Queues

Heaps

Your to-dos

1. Read **before Fri**: Bailey 13.4. Last reading!
2. Lab 10 (partner lab), **due Tuesday 12/6 by 10pm**.
3. **Last quiz**, this **Fri/Sat**.

Announcements

1. **Student course surveys**,
   in lab, **Wednesday & Thursday this week**.
2. **Final exam**: Saturday, Dec 17, 1:30pm.
   Room TBD.
3. **Final exam review session**,
   in class, last day of class, **Friday 12/9**.

Announcements

Sean Barker '09, Bowdoin College

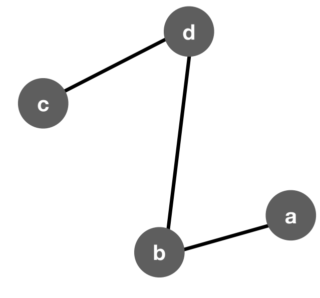
Friday, Dec 2 @ 2:35pm
Computer Science Colloquium – Wege TCL 123
Smart Meters for Smart Cities: Data Analytics in Energy-Aware Buildings

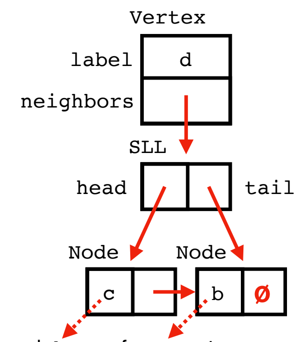
The proliferation of smart energy meters has resulted in many opportunities for next-generation buildings. Energy-aware "smart buildings" may optimize their energy consumption and provide convenience and economic benefits through analysis of their meter data. However, storing and analyzing this data presents computational challenges, especially when conducted at scale. In this talk, I discuss our work on several problems in this space, focusing particularly on efficient compression of smart meter data and the disaggregation of building-wide consumption into individual device consumption. Our work in these areas aims to support the development of sustainable, energy-efficient smart cities and smart grids.

But first, a clarification

Object-oriented adjacency list:

```
public class Vertex<T> {
    T label;
    List<Vertex<T>> neighbors = new SinglyLinkedList<>();
    …
}
```





(strictly speaking, `c` and `d` are references to `Vertex` objects)

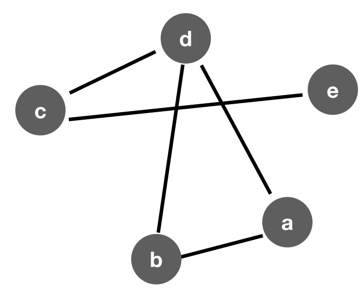
Connectedness

Activity: connectedness

`boolean isConnected()`

How might I compute this using fundamental ops?

(`adjacent`, `vertices`, `incident`, `degree`, `neighbors`)



(note that graph is undirected)

Idea: breadth-first counting

Idea:

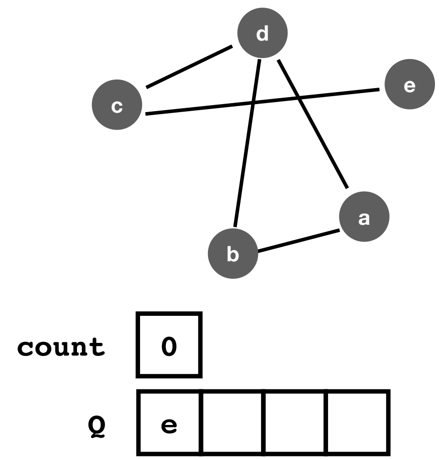
(suppose we know |G|)

`boolean isConnected(Vertex start)`

1. let `count = 0`
2. let `Q` be an empty queue
3. enqueue `start`
4. while `Q` not empty
   a. dequeue `v`
   b. count `v`
   c. mark `v` as visited
   d. put `v`'s unmarked neighbors in `Q`
5. if `count` = # of vertices in graph, return `true` else `false`

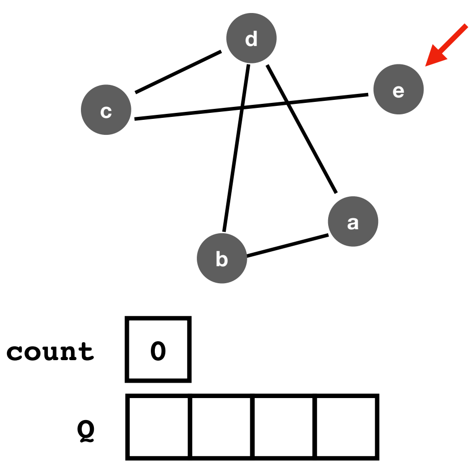
Algorithm: connectedness

initialize algorithm



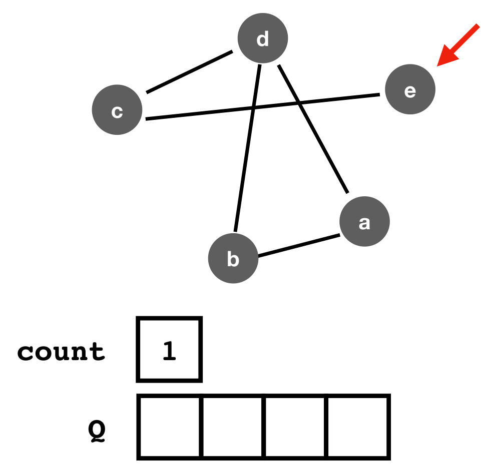
Algorithm: connectedness

dequeue v



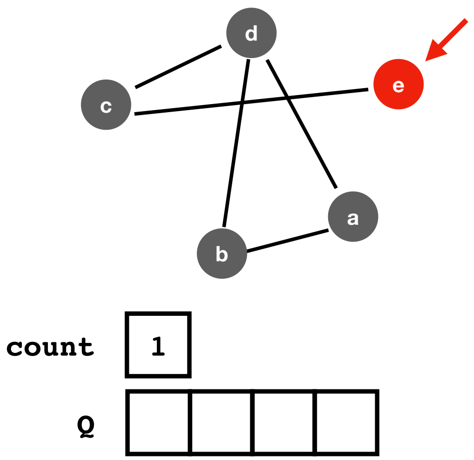
Algorithm: connectedness

count v



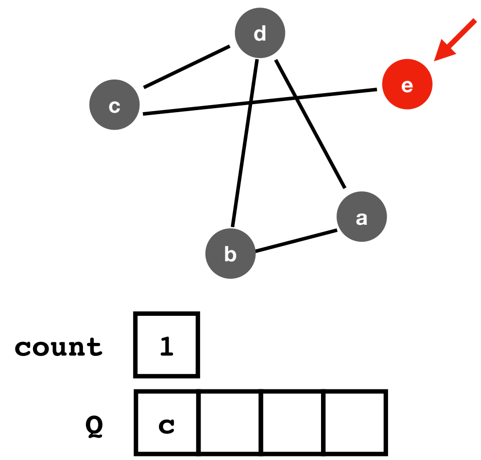
Algorithm: connectedness

mark v



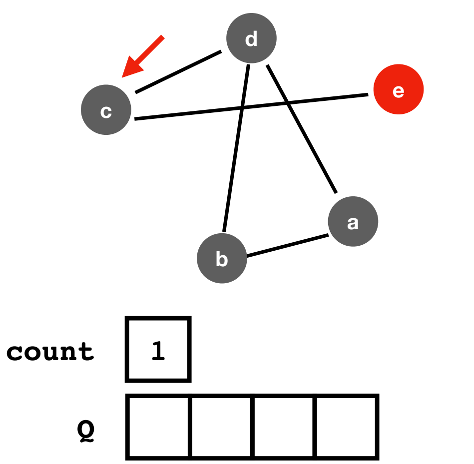
Algorithm: connectedness

enqueue unmarked neighbors



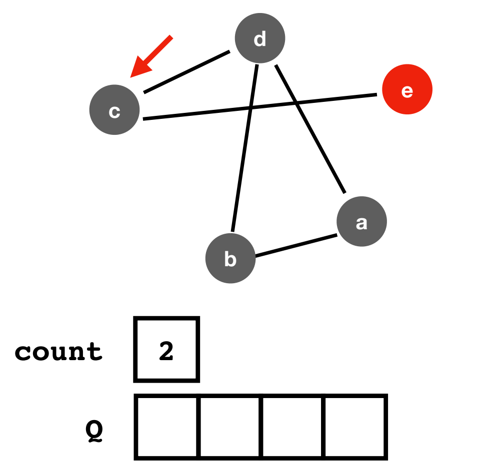
Algorithm: connectedness

dequeue v



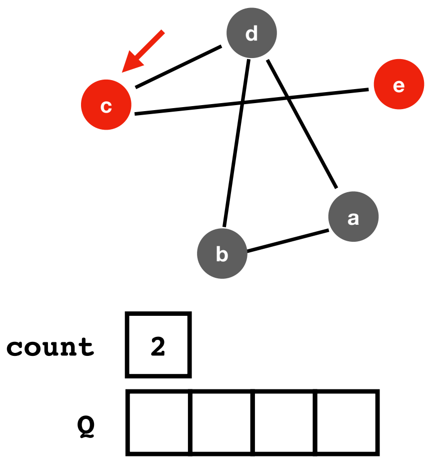
Algorithm: connectedness

count v



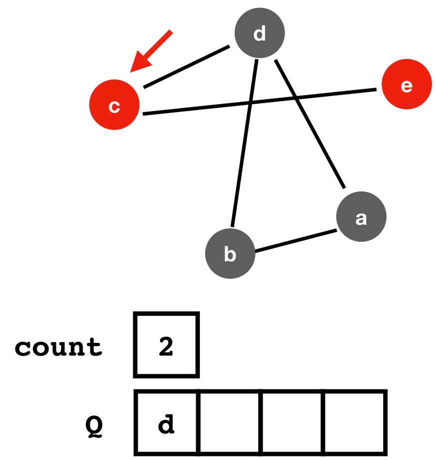
Algorithm: connectedness

mark v



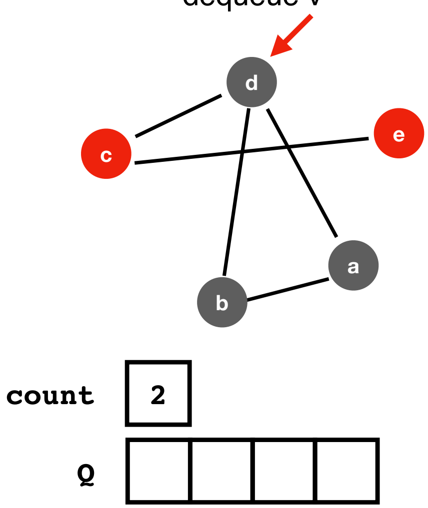
Algorithm: connectedness

enqueue unmarked neighbors



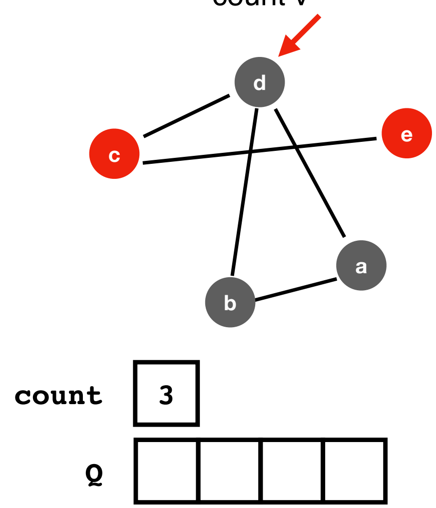
Algorithm: connectedness

dequeue v



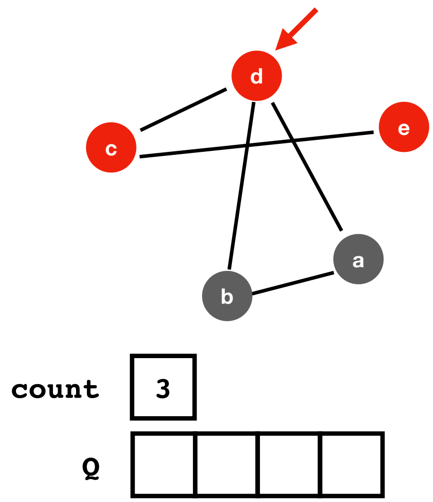
Algorithm: connectedness

count v



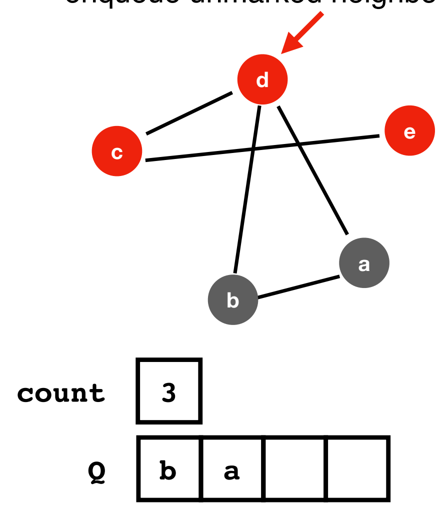
Algorithm: connectedness

mark v



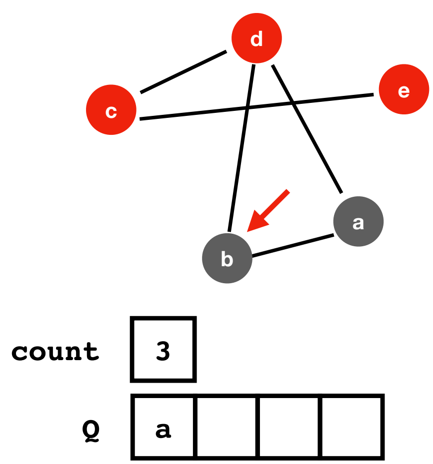
Algorithm: connectedness

enqueue unmarked neighbors



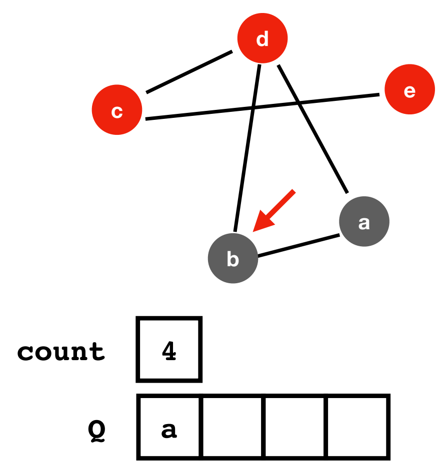
Algorithm: connectedness

dequeue v



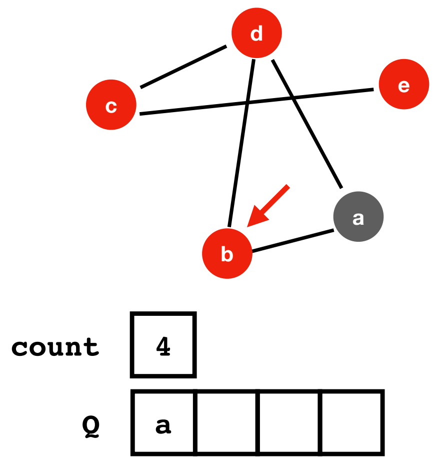
Algorithm: connectedness

count v



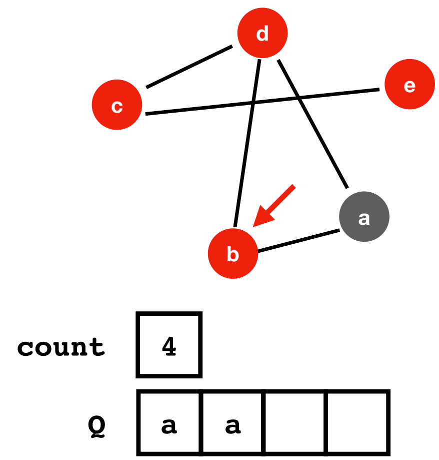
Algorithm: connectedness

mark v



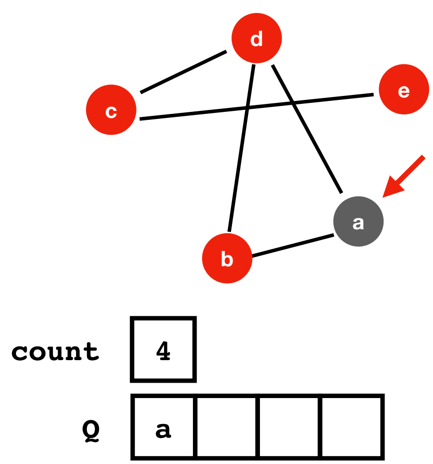
Algorithm: connectedness

enqueue unmarked neighbors



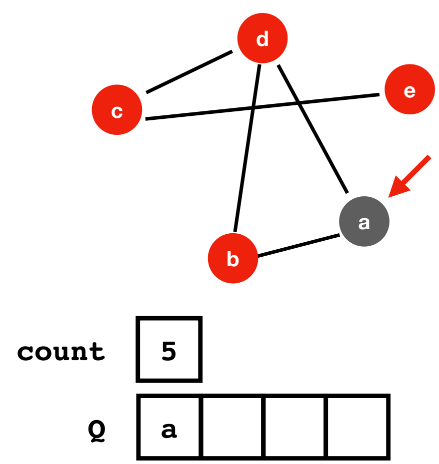
Algorithm: connectedness

dequeue v



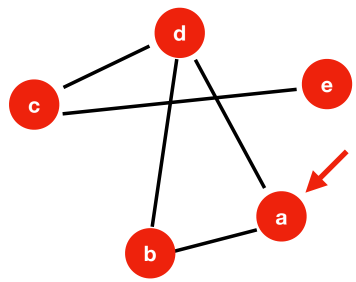
Algorithm: connectedness

count v



Algorithm: connectedness

mark v

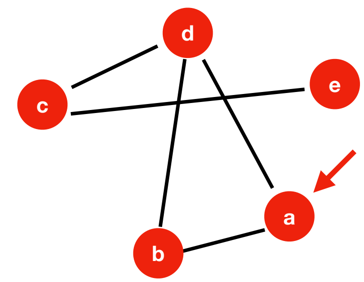


count 5

Q a

Algorithm: connectedness

dequeue v (but don't visit)

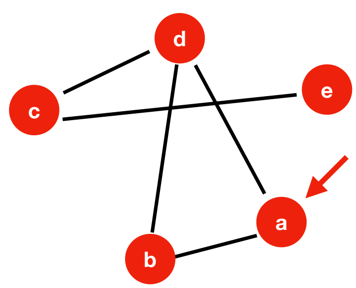


count 5

Q

Algorithm: connectedness

compute |G| == count



count 5

Q

`return 5 == 5`
`(true)`

Priority Queues

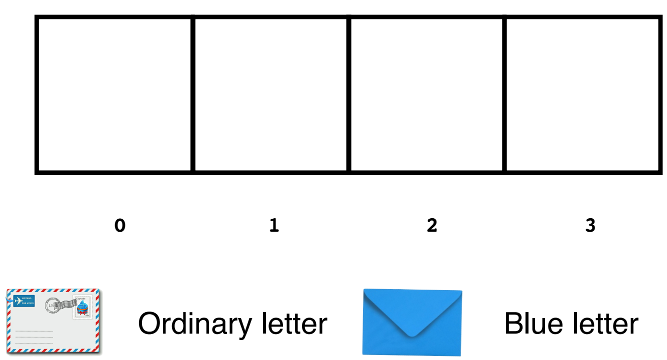
Priority Queue

A **priority queue** is an abstract data type that returns the elements in **priority order**. Under priority ordering, an element **e** with a higher priority (an integer) is returned before all elements **L** having lower priority, even if that **e** was enqueued after all **L**. When any two elements have **equal priority**, they are returned in **first-in, first-out order** (i.e., in the order in which they were enqueued).

Note

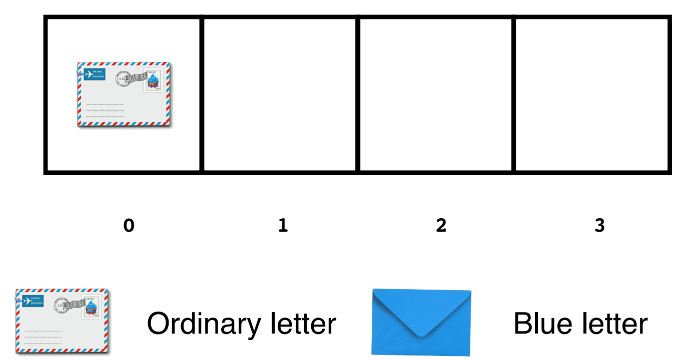
I will refer here to the **maximum** priority. But you could also refer to **minimum** priority. All that matters is that you order your data with respect to some **extremum**.

Priority Queue



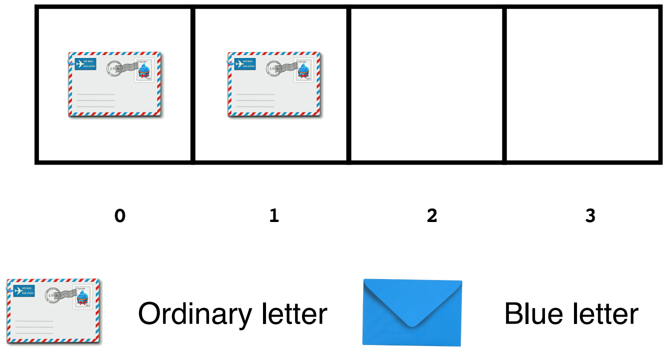
Priority Queue

enqueue



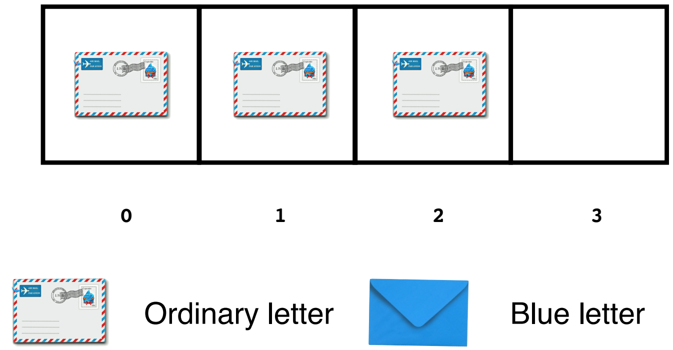
Priority Queue

enqueue



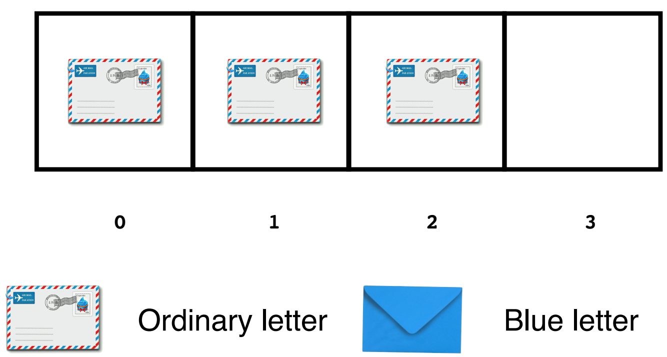
Priority Queue

enqueue



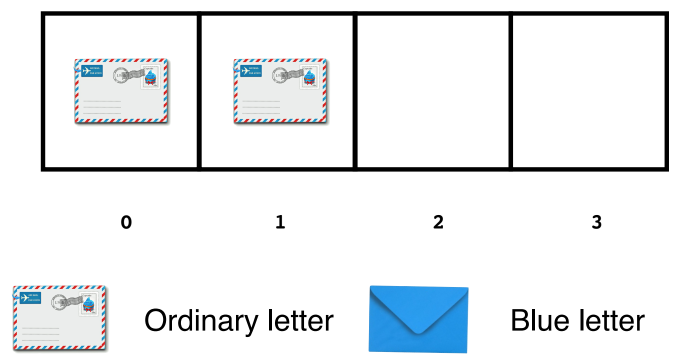
Priority Queue

extract



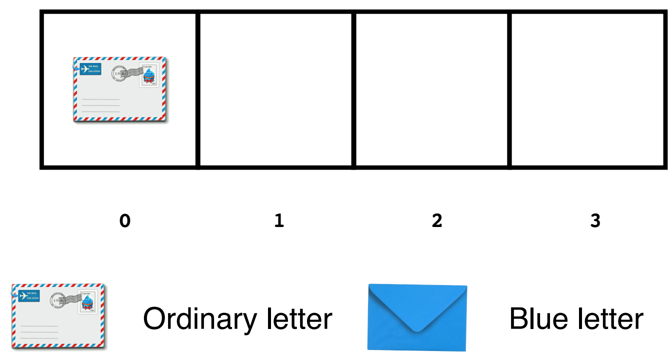
Priority Queue

extract



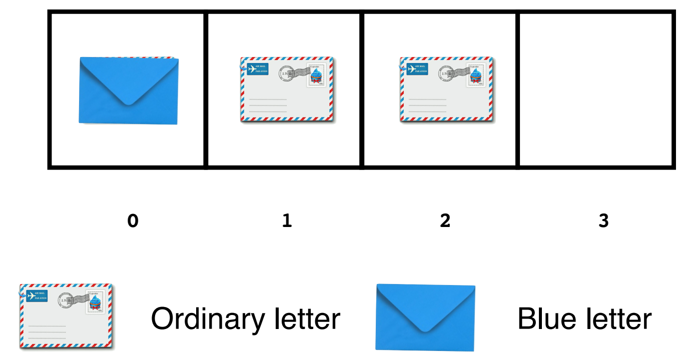
Priority Queue

extract



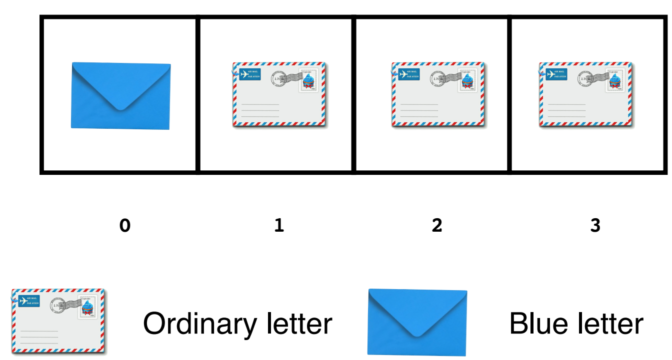
Priority Queue

blue letters: enqueue



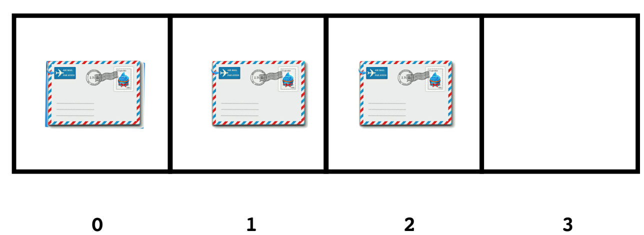
Priority Queue

blue letters: extract



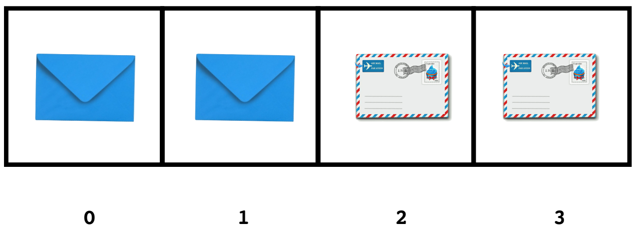
Priority Queue: Operations

insert: inserts an element with a given priority value. Ensures that the next element of the queue is in priority order. Like **enqueue**.



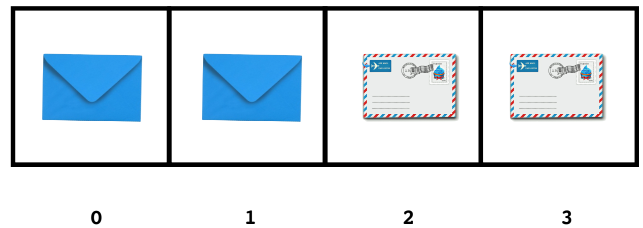
Priority Queue: Operations

find-max: returns the next element with a highest priority value. Like **peek**, does not modify the queue.



Priority Queue: Operations

extract: removes and returns the next element with a maximum priority value. Like **dequeue**.



Priority Queue

How to implement?

Vector:
- **find-max**: O(1)
- **insert**: O(n)
- **extract**: O(n)

BinarySearchTree:
- **find-max**: O(n)
- **insert**: O(n)
- **extract**: O(n)

Heap:
- **find-max**: O(1)
- **insert**: O(log n)
- **extract**: O(log n)

Priority Queue

Is it **necessary** to keep the **entire queue** in sorted order?

Operations:

find-max
insert
extract

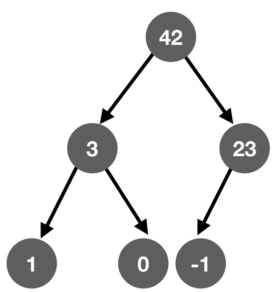
Heaps

Max Heap

A **max heap** is a tree-based data structure that returns its elements in **priority order**. A heap maintains the **max heap property**: for any given node **n**, if **p** is a parent node of **n**, then the **key** of **p** is ≥ to the **key** of **n**.

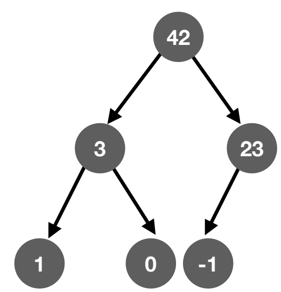
A max heap is a tree whose root is the maximum element and whose subtrees are, themselves, heaps.

Is this a binary search tree?



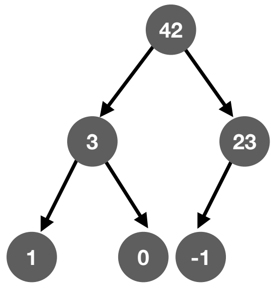
No. Values do not obey **binary search property**.

(Binary) max heap



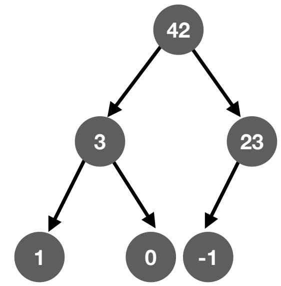
Max heap property: for any given node **n**, if **p** is a parent node of **n**, then the **key** of **p** is ≥ the **key** of **n**.

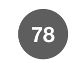
Insertion



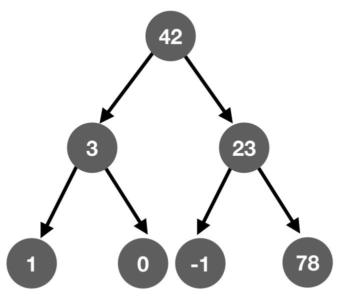
A **binary heap** is usually implemented as an **always-complete binary tree**.

Insertion



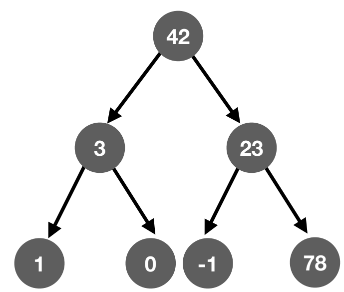
Suppose we want to insert a new node, 

Insertion



First, **insert** the new node at the first available position in the tree that **maintains completeness**.

Insertion

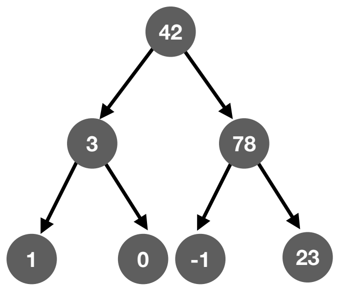


23 ≥ 78 **?**

No.

Next, **compare** the new node with its parent.

Insertion

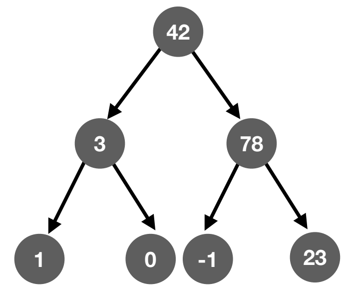


23 ≥ 78 ?

No.

If the **max heap property** is violated, **swap**.

Insertion

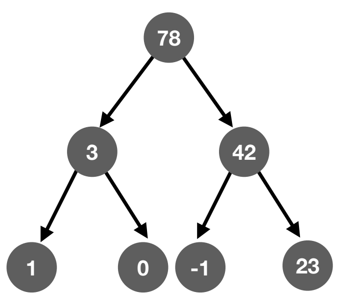


42 ≥ 78 ?

No.

Continue swapping the new node with parents until the **max heap property is satisfied**.

Insertion

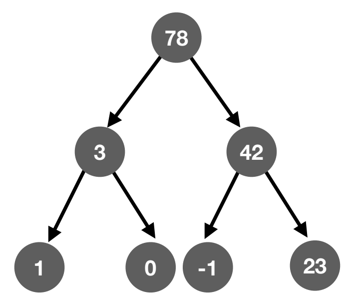


42 ≥ 78 ?

No.

Continue swapping the new node with parents until the **max heap property is satisfied** (parent ≥ node or no parents remain).

Insertion

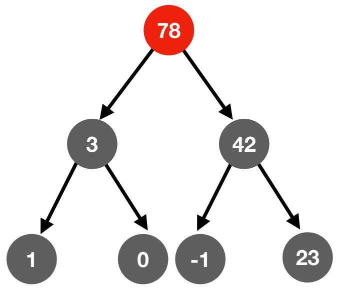


42 ≥ 78 ?

No.

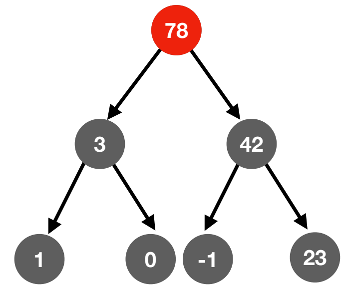
The **swapping procedure** performed on **insert** is often referred to as **heap-up** or **percolate-up**.

Find-max



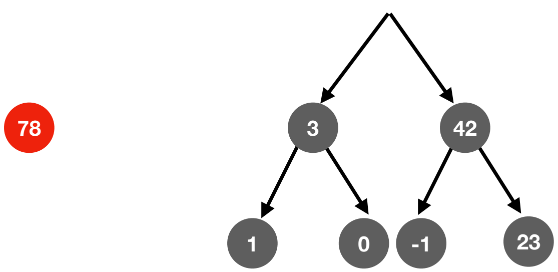
To find the **maximum element** in a max heap, simply **return** the **root**.

Extract



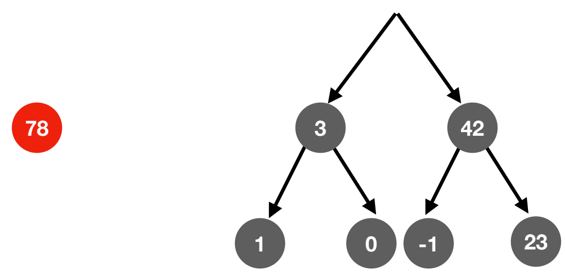
To **remove and return** the **maximum element** in a max heap, first perform **find-max**.

Extract



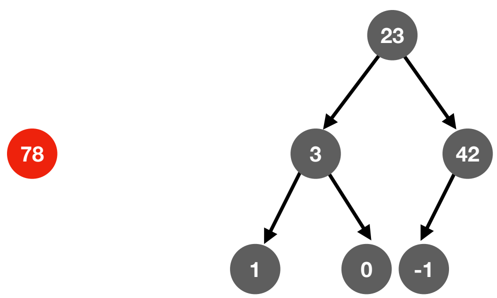
Temporarily store the max element.

Extract



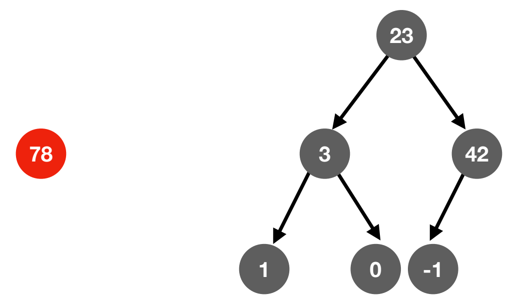
Replace the **root** with the **last element** in the complete tree.

Extract



Replace the **root** with the **last element** in the complete tree.

Extract

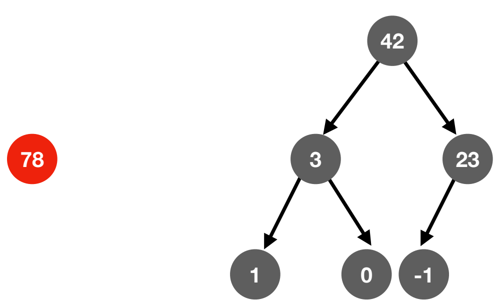


23 ≥ 42 ?

No.

Compare the root with its children. **Swap** the **root** with **the largest element**.

Extract

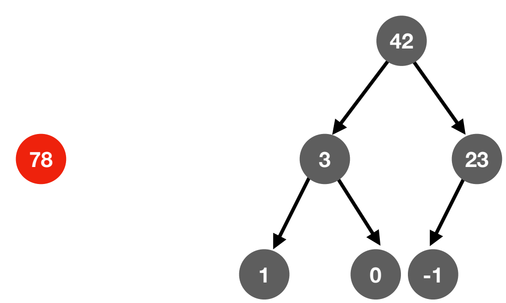


23 ≥ 42 ?

No.

Compare the root with its children. **Swap** the **root** with **the largest element**.

Extract

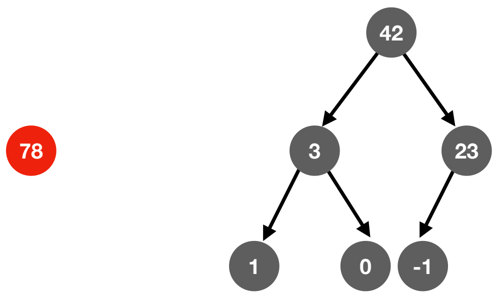


23 ≥ -1 ?

Yes.

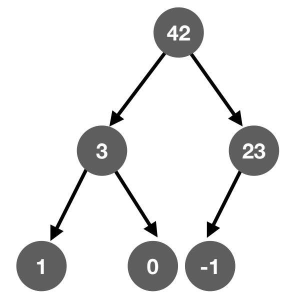
Continue swapping until the **max heap property is satisfied** (parent ≥ node or no parents remain).

Extract



Return the saved maximum element.

Extract



The **swapping procedure** performed on **extract** is often referred to as **heap-down** or **percolate-down**.

Recap & Next Class

Today:

Priority queues

Heaps

Next class:

More heaps!