CSCI 136:
Data Structures
and
Advanced Programming

Lecture 24

Trees, part 4

Instructors: Dan & Bill

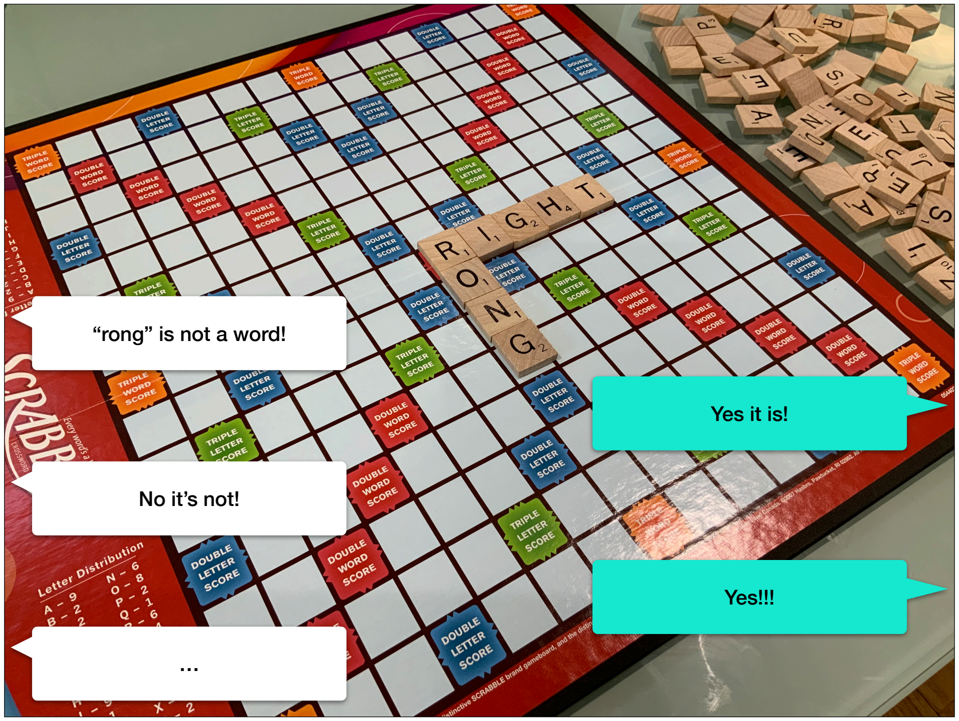
Williams

Outline

Tries

Priority Queue

Deep Thoughts About Ordered Structures



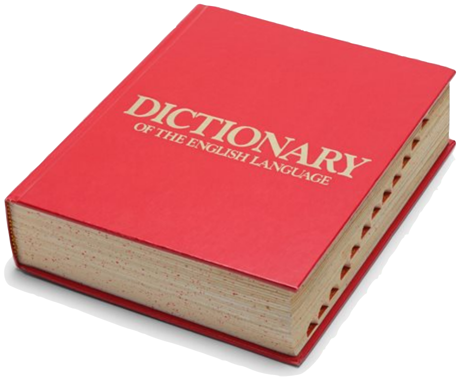
“rong” is not a word!

Yes it is!

No it’s not!

Yes!!!

…



Set

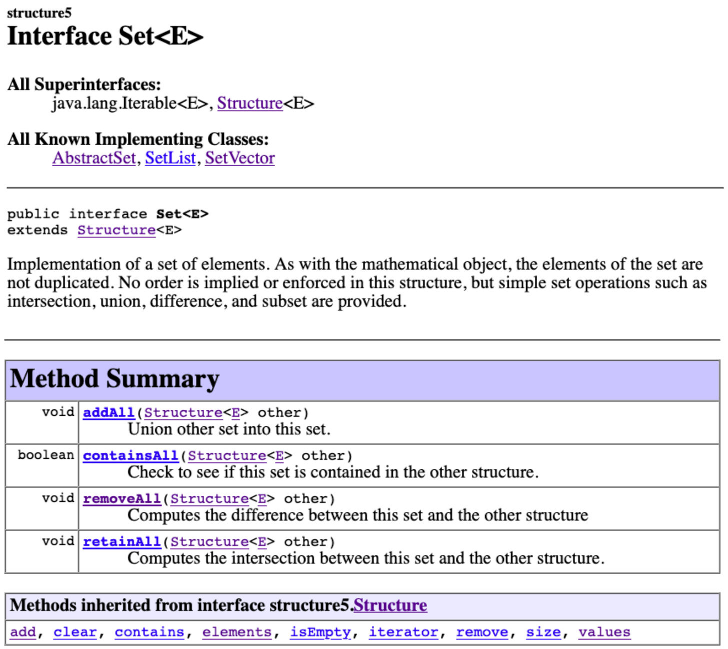
A **set** is an abstract data type that stores **unique values** in **no particular order**.

Important operations are:

- `add`
- `contains`
- `remove`
- `size`

Sounds like any other `Structure`, right? **Less is more.**

`Set` does **not** need to store **duplicate values**.



structure5

Interface Set<E>

All Superinterfaces:
java.lang.Iterable<E>, Structure<E>

All Known Implementing Classes:
AbstractSet, SetList, SetVector

```
public interface Set<E>
extends Structure<E>
```

Implementation of a set of elements. As with the mathematical object, the elements of the set are not duplicated. No order is implied or enforced in this structure, but simple set operations such as intersection, union, difference, and subset are provided.

Method Summary	
void	`addAll(Structure<E> other)` Union other set into this set.
boolean	`containsAll(Structure<E> other)` Check to see if this set is contained in the other structure.
void	`removeAll(Structure<E> other)` Computes the difference between this set and the other structure
void	`retainAll(Structure<E> other)` Computes the intersection between this set and the other structure.

Methods inherited from interface structure5.Structure
`add, clear, contains, elements, isEmpty, iterator, remove, size, values`

Trie

A **trie** is an **ordered tree structure** used to store a set of “strings” in a highly-space efficient manner. A lookup in a trie is also highly efficient, **O(m)**, where **m** is the length of the string, in the worst case.

Tries are often used to represent **set** ADTs.

The word “trie” comes from “retrieval”.

Most people pronounce it “try” to avoid confusion.

Trie

Insight: A path to a node represents a **string prefix**. Every string that shares a prefix with another string **also shares tree ancestors** with that string.

`bit`

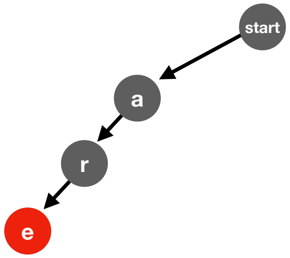
`bitrate`

shared prefix: `bit`

Adding to a trie

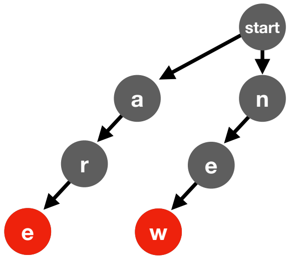
are
new
zen
no
not
as

Adding to a trie



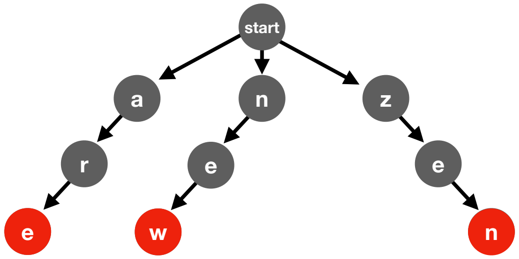
are
new
zen
no
not
as

Adding to a trie



are
new
zen
no
not
as

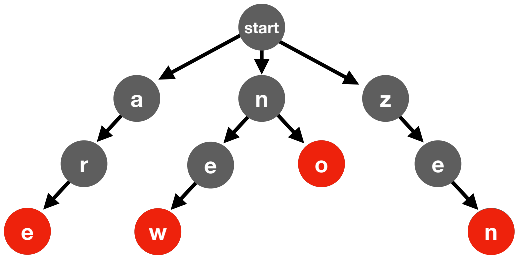
Adding to a trie



are
new
zen
no
not
as

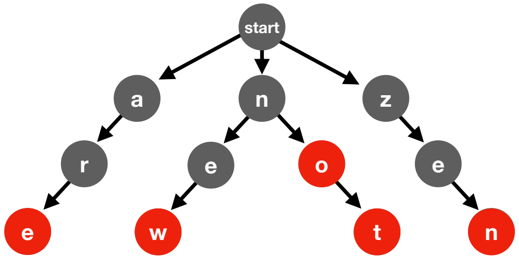
Adding to a trie

are
new
zen
no ←
not
as



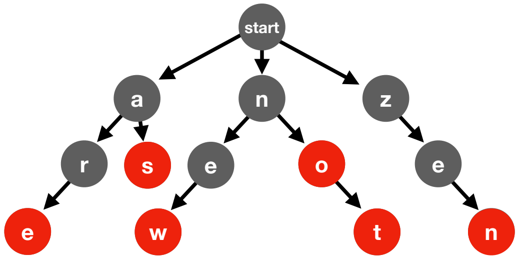
Adding to a trie

are
new
zen
no
not ←
as

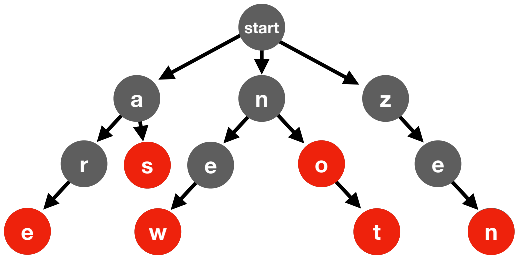


Adding to a trie

are
new
zen
no
not
as ←

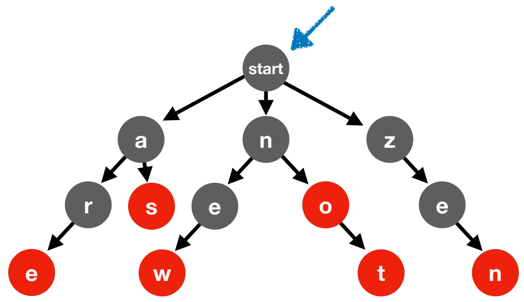


Querying a trie



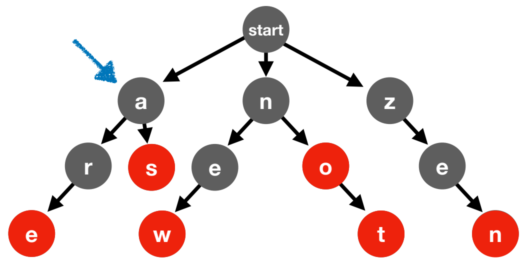
Is the string “`are`” in the set?

Querying a trie



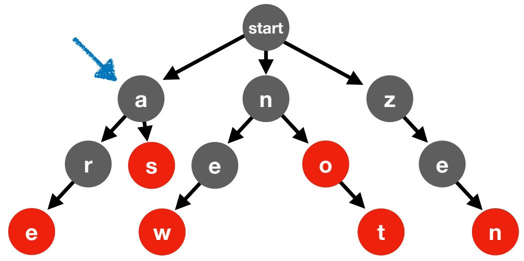
Is the string “are” in the set?

Querying a trie



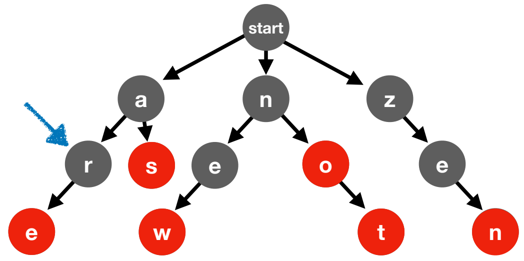
Is the string “are” in the set?

Querying a trie



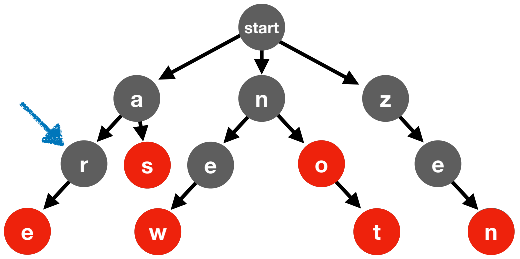
Is the string “are” in the set?

Querying a trie



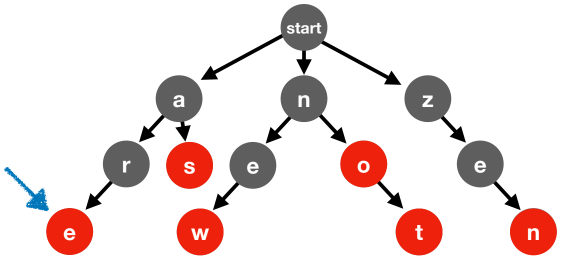
Is the string “are” in the set?

Querying a trie



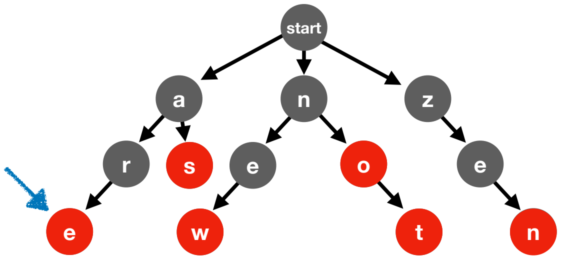
Is the string “`are`” in the set?

Querying a trie



Is the string “`are`” in the set?

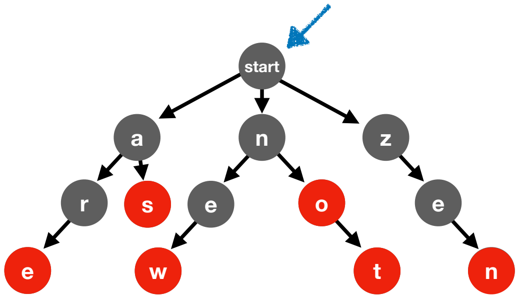
Querying a trie



Is the string “`are`” in the set?

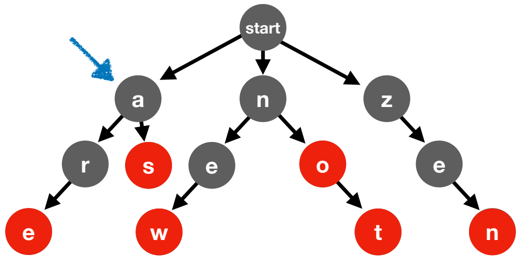
Yes!

Querying a trie



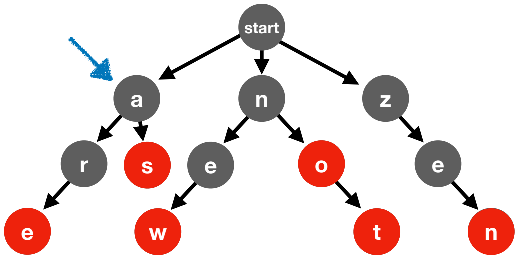
Is the string “`art`” in the set?

Querying a trie



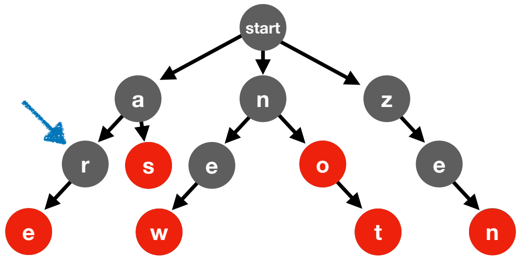
Is the string “`art`” in the set?

Querying a trie



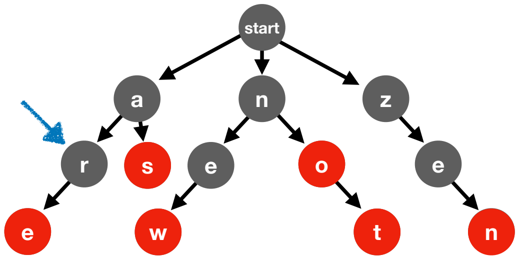
Is the string “`art`” in the set?

Querying a trie



Is the string “`art`” in the set?

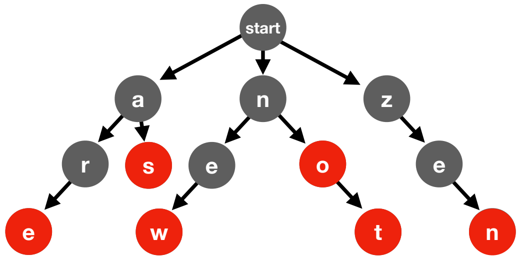
Querying a trie



Is the string “`art`” in the set?

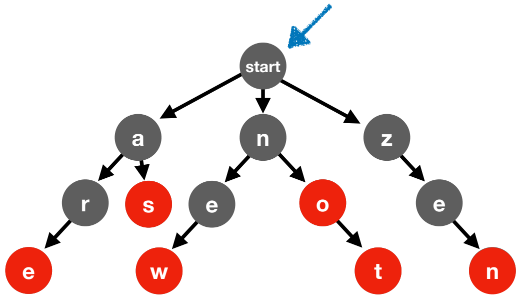
No

Removing from a trie



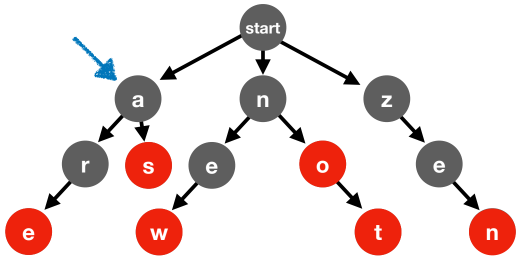
Remove “`as`” from the trie.

Removing from a trie



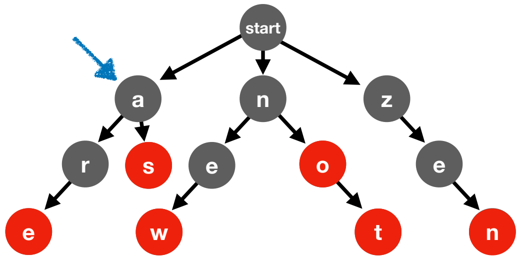
Remove “`as`” from the trie.

Removing from a trie



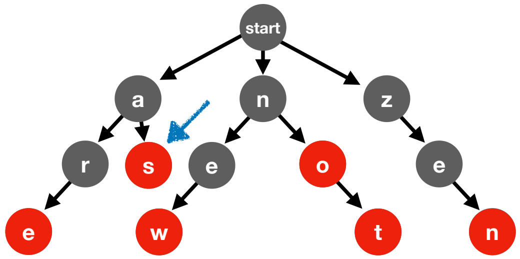
Remove “`as`” from the trie.

Removing from a trie



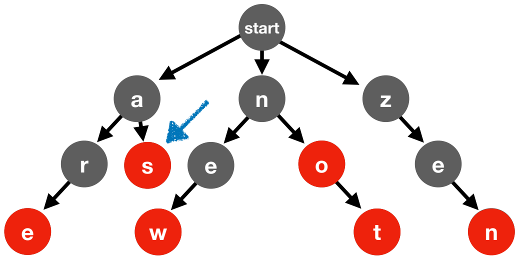
Remove “`as`” from the trie.

Removing from a trie



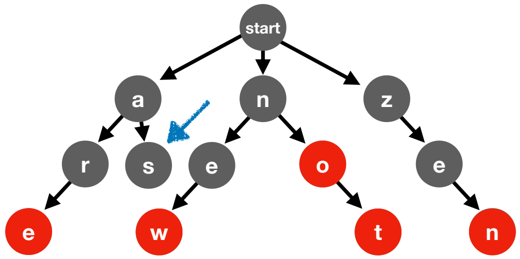
Remove “`as`” from the trie.

Removing from a trie



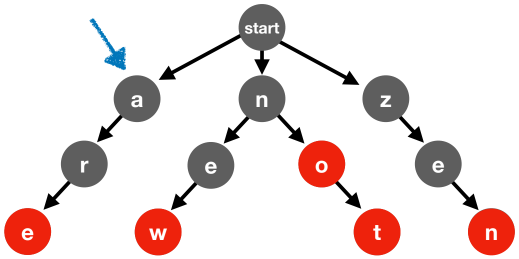
Remove “`as`” from the trie.

Removing from a trie



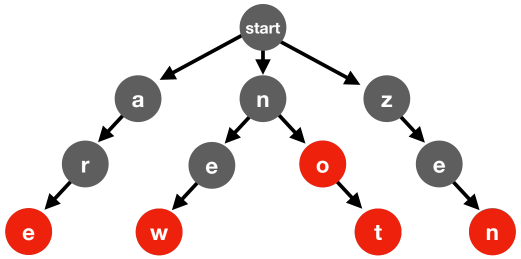
Remove “`as`” from the trie.

Removing from a trie



Remove “`as`” from the trie.

Removing from a trie



Think about how to remove “zen” from the trie.

Lab 7 `Lexicon` interface

```
public class LexiconTrie implements Lexicon {
    public boolean addWord(String word) { … }
    public int addWordsFromFile(String filename) { … }
    public boolean removeWord(String word) { … }
    public int numWords() { … }
    public boolean containsWord(String word) { … }
    public boolean containsPrefix(String prefix) { … }
    public Iterator<String> iterator() { … }
    public Set<String> suggestCorrections(String target, int maxDistance) { … }
    public Set<String> matchRegex(String pattern) { … }
}
```

Priority

MOMS SHOULD GET A FAST PASS TO THE FRONT OF THE LINE AT COFFEE SHOPS. HONEY, YOU'RE 22 & SLEPT 10 HOURS LAST NIGHT? GET TO THE BACK OF THE LINE

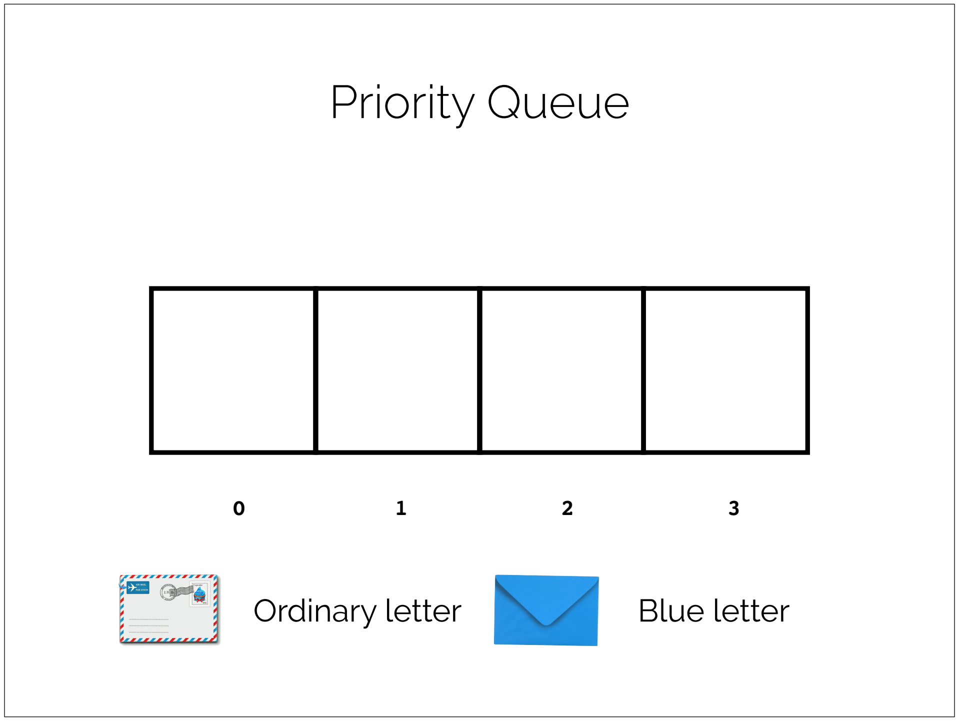
Priority Queue

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**.

Blue letter

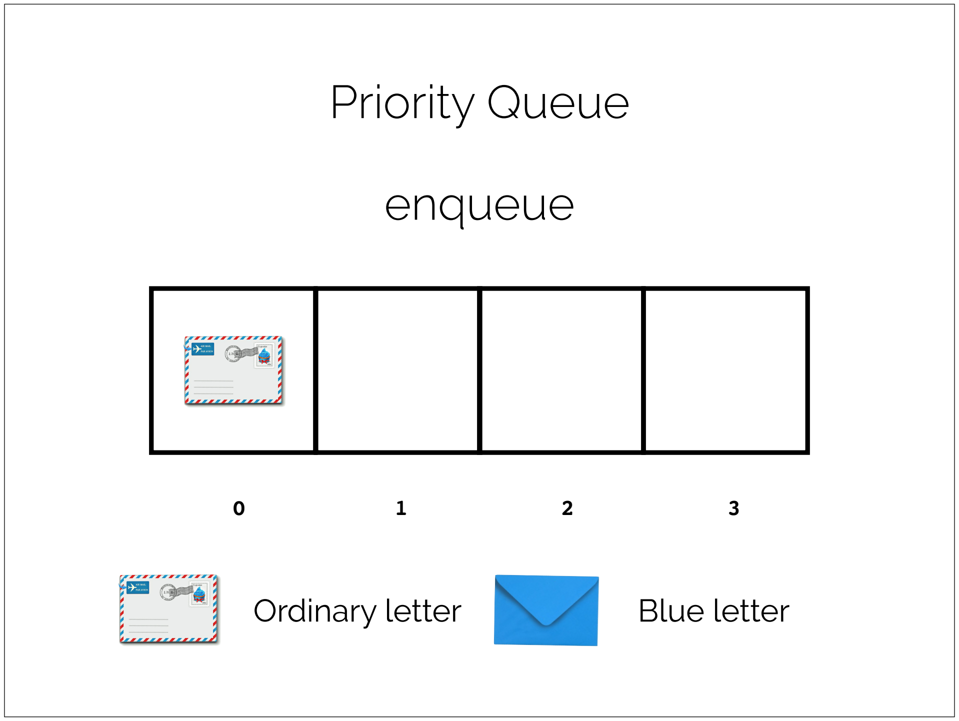


Priority Queue

0	1	2	3

Ordinary letter

Blue letter



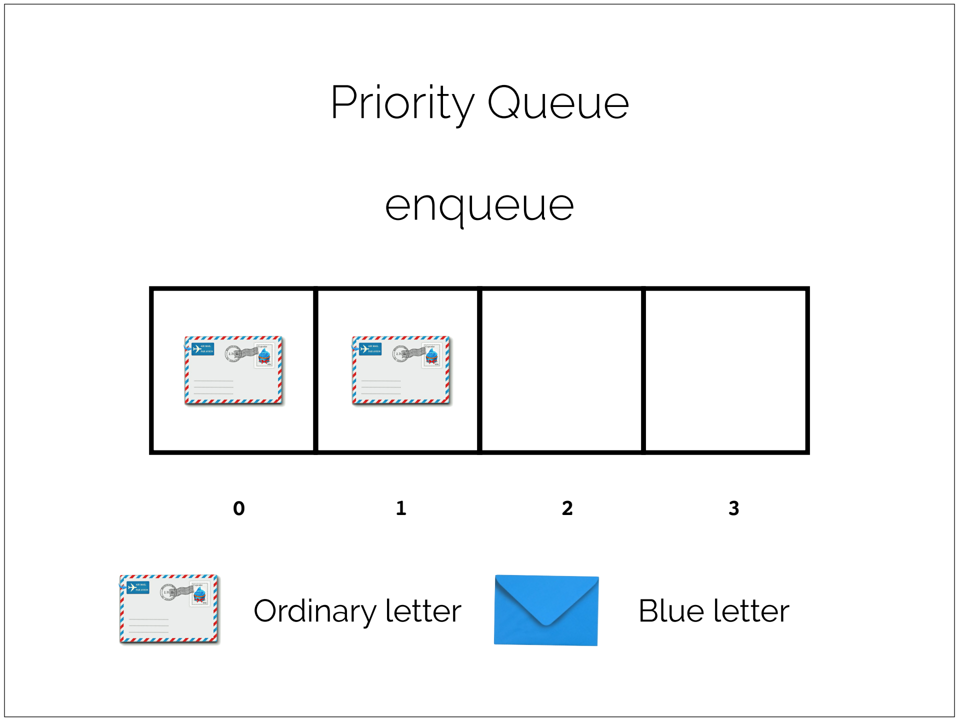
Priority Queue

enqueue

✉			
0	1	2	3

Ordinary letter

Blue letter



Priority Queue

enqueue

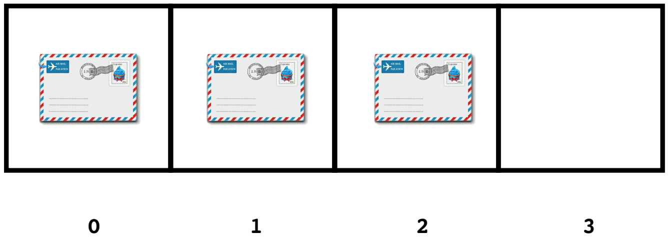
✉	✉		
0	1	2	3

Ordinary letter

Blue letter

Priority Queue

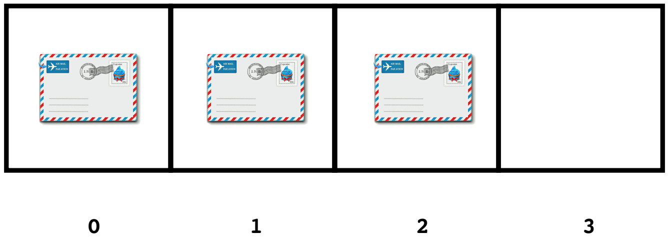
enqueue



✉ Ordinary letter ✉ Blue letter

Priority Queue

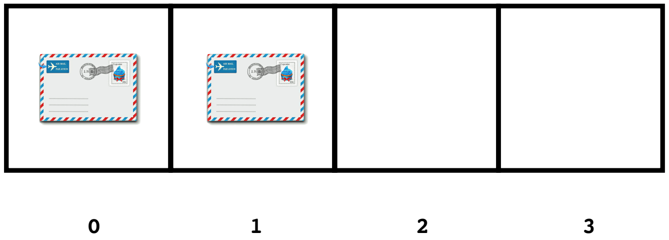
extract



✉ Ordinary letter ✉ Blue letter

Priority Queue

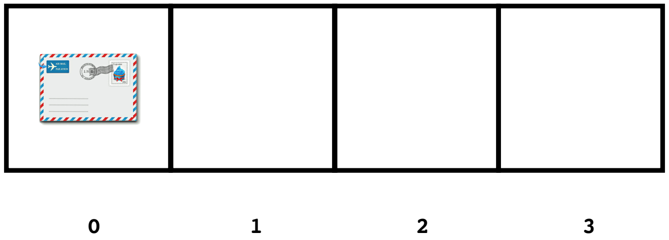
extract



✉ Ordinary letter ✉ Blue letter

Priority Queue

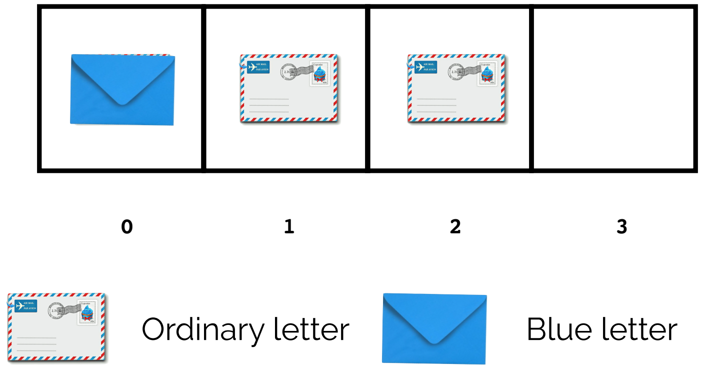
extract



✉ Ordinary letter ✉ Blue letter

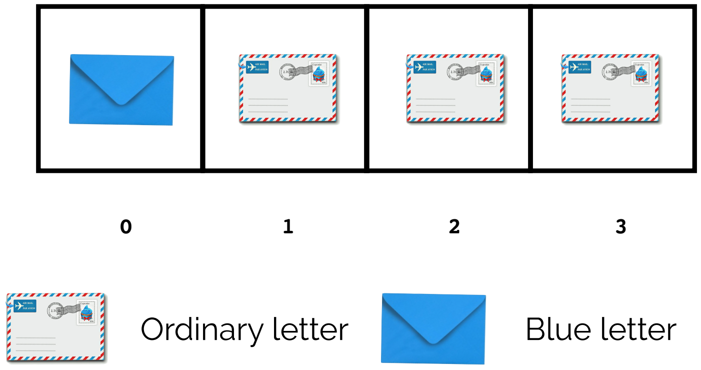
Priority Queue

blue letters: enqueue



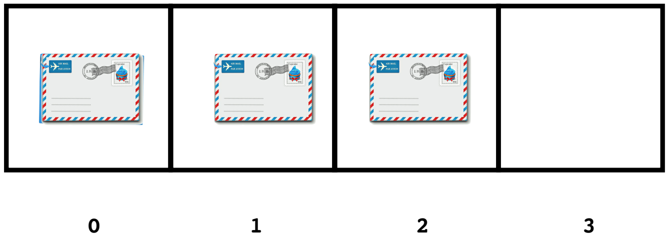
Priority Queue

blue letters: extract



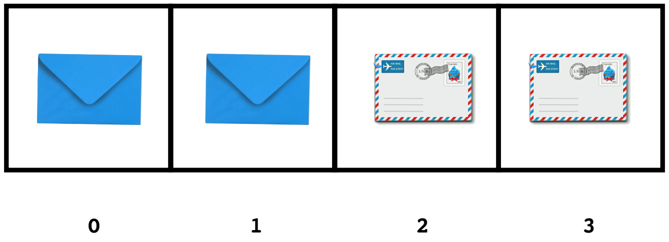
Priority Queue: Operations

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



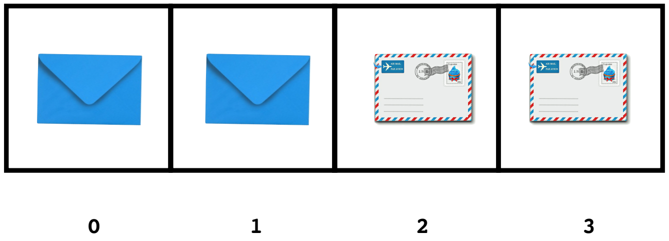
Priority Queue: Operations

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

Recap & Next Class

This lecture:

Tries

Priority Queue ADT

Next lecture:

Heaps