CSCM712: Problem efficiency & software solving
Stacks and queues

Cécilia Pradic

March 9th 2026

In the previous episode

- We've seen a couple of implementations for ordered collections:
  - linked lists
  - dynamic arrays

You've played with them in the lab.

Now

Two more restrictive interfaces with applications.

stacks

queues

And then some teasing about **priority queues**.

Stacks

- **LIFO**: last-in, first-out
- Minimal interface:

```
class stack:
  def __init__(self):
    # generates an empty stack
  def pop(self):
    # remove and returns the top element
  def push(self, x):
    # puts x on top
  def isEmpty(self):
    # returns a boolean
```

- Common "extensions": `__len__`, `peek`

Why stacks? Example 1: parsing

What is parsing?

Turning strings into a more structured presentation.

- Example: the python interpreter first parses your text files before turning them into bytecode and running your programs.
- Similar task: check for syntax errors.

Baby example

Are the following expressions valid?

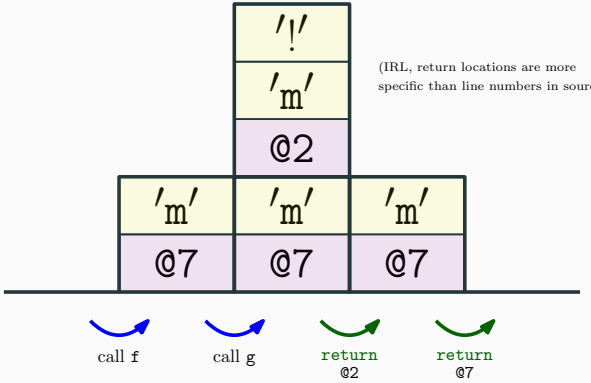
```
'[(1 + 5) * 2] + [77 + 8]'
'((1)+(2-1) * [w + ([5]))] / (((10)] + 5)'
```

Implementation: your first exercise for next lab.

Why stacks? Example 2: interpreting function calls (1/3)

- You may have heard of the notion of a **call stack**
- When function calls are made in programs, your computer remembers
  - where the functions are called from (to `return`)
  - what are the arguments of the functions
- They're put in a dedicated memory chunk treated as a stack.
- It has a fixed size, typically $\sim$ 4Kb on modern computers (smoll :()

```
1  def f(s):
2    g(s, '!')
3
4  def g(s, r):
5    print('boo' + s + r)
6
7  f('m')
```

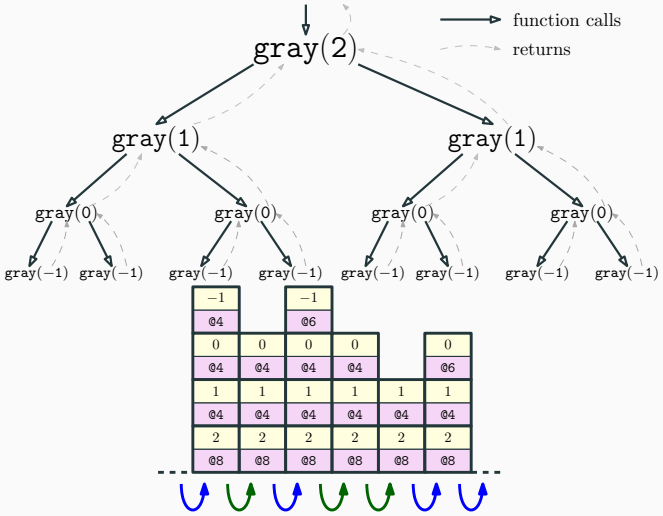


(IRL, return locations are more specific than line numbers in source code)

Why stacks? Example 2: interpreting function calls (2/3)

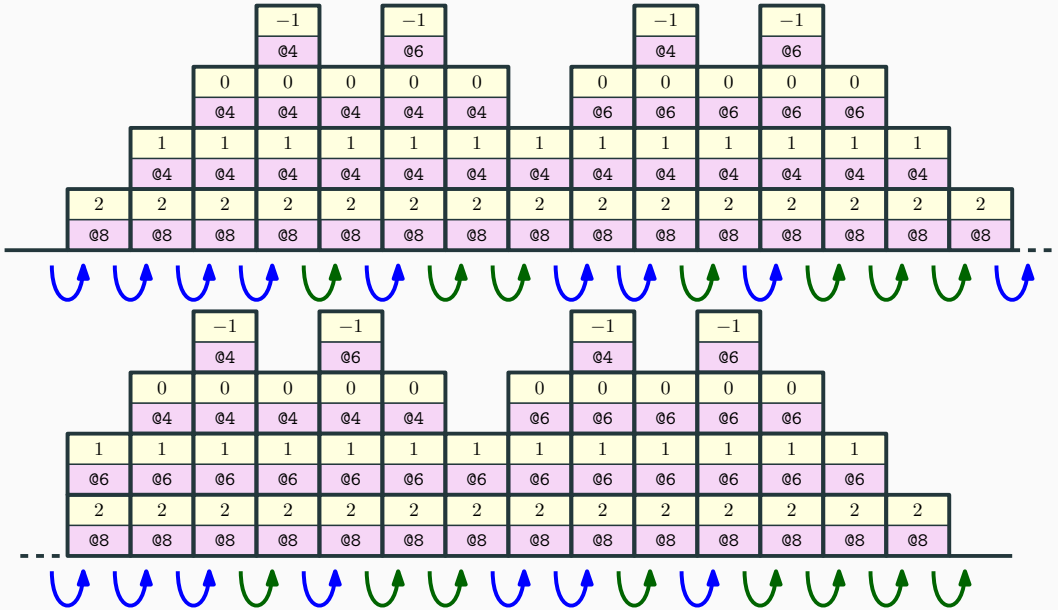
Of course this gets more fun with recursive functions.

```
1  def gray(n):
2      if n < 0:
3          return
4      gray(n-1)
5      print(n, end='')
6      gray(n-1)
7
8  gray(2)
```



Why stacks? Example 2: interpreting function calls (3/3)

The full thing.



Queues

FIFO: first-in first-out

```
class queue:
  def __init__(self):
    # initialize an empty queue
  def enqueue(self, x):
    # add x at the end of the queue
  def dequeue(self):
    # remove & return the first element
  def isEmpty(self):
    # returns a boolean
```

Why queues? Example 1: buffers

Typical usage of queues: **buffering**

- Sometimes a system receives much data
- Can't process everything at once
- → store that in a queue

Buffers you know and love

`stdin` and `stdout`

- Buffers of inputs and outputs for the terminal
- `fflush(stdout)` = "hey please process everything in there, don't worry I'll be patient and wait for that"

Why queues? Example 2: breadth-first traversals

To be discussed **soon** :)

How does implement

Typically linked lists and arrays

- Arrays are memory-efficient, but fixed-size
- Memory-efficiency aside, linked lists are cool and $\mathcal{O}(1)$

How does implement

Typically linked lists and arrays

- Arrays are memory-efficient, but fixed-size
- Memory-efficiency aside, linked lists are cool and $\mathcal{O}(1)$

Dynamic arrays (python list) don't make too much sense in my view, but probably not too horrible if you are lazy anyway `¯\("/)/¯`

Priority queues

Not just a bourgeois invention.

Idea:

- each element is inserted with a priority
- remove the element with **minimal priority** first

```
class priorityQueue:
  def __init__(self):
    # empty queue
  def enqueue(self, x, k):
    # enqueue x with priority k
  def dequeue(self):
    # remove and return an element with minimal priority
  def min(self):
    # return the pair (x, k) of an element x with priority k
    # that would be dequeued next, or None if the priority queue is empty
```

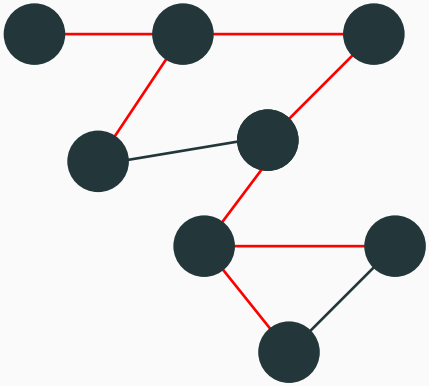
Why priority queues? Example 1: sorting

```
def pQueueSort(xs):
  p = priorityQueue()
  for i in xs:
    p.enqueue(i,i)
  result = []
  while p.min != None:
    result.append(p.dequeue())
  return result
```

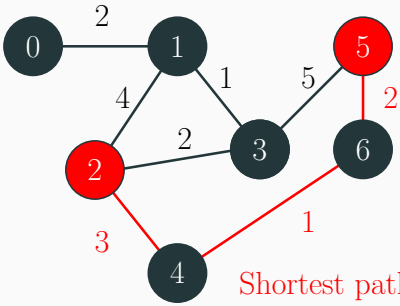
If `enqueue` and `dequeue` run in $\mathcal{O}(f(n))$

$\Longrightarrow$ `pQueueSort` in $\mathcal{O}(n \times f(n))$

Why priority queues? Examples 2 & 3: graph algorithms

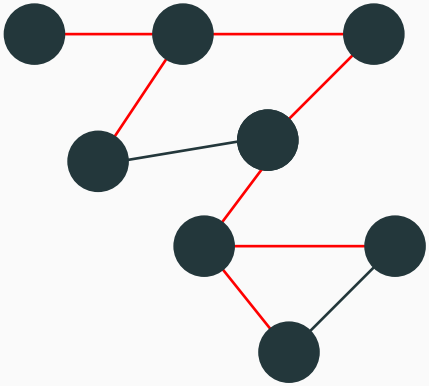


Minimum Spanning Trees

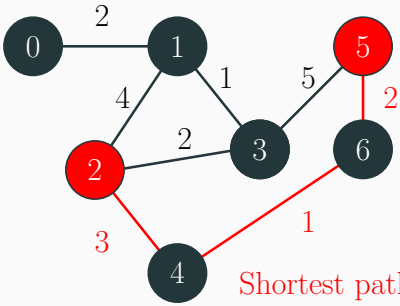


Shortest paths

Why priority queues? Examples 2 & 3: graph algorithms



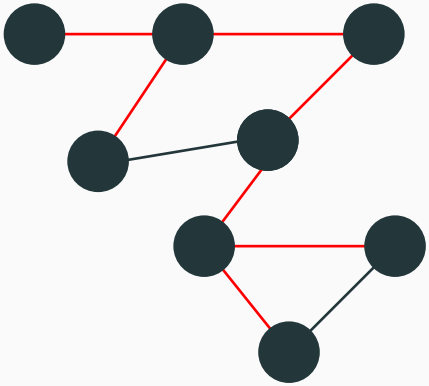
Minimum Spanning Trees



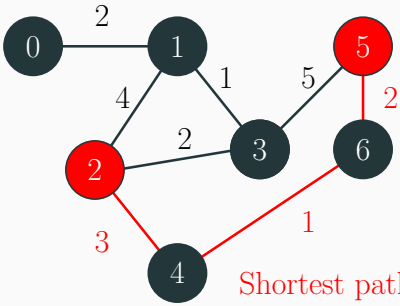
Shortest paths

(see e.g. §8.20 and §8.22 of *Problem Solving with Algorithms and Data Structures using Python* by B. Miller & D. Ranum for details; **maybe** also later lectures!)

Why priority queues? Examples 2 & 3: graph algorithms



Minimum Spanning Trees



Shortest paths

(see e.g. §8.20 and §8.22 of *Problem Solving with Algorithms and Data Structures using Python* by B. Miller & D. Ranum for details; **maybe** also later lectures!)

N.B.: graphs $\notin$ proforma $\supseteq$ *learning outcomes* $\supseteq$ exam $\subseteq$ *assessment & feedback*

How does implement

Using arrays and lists, time complexity?

`enqueue`, `dequeue` in...

How does implement

Using arrays and lists, time complexity?

`enqueue`, `dequeue` in... **linear time** $\mathcal{O}(n)$ (whether you keep things ordered or not)

Sad! :(

How does implement

Using arrays and lists, time complexity?

`enqueue`, `dequeue` in... **linear time** $\mathcal{O}(n)$ (whether you keep things ordered or not)

Sad! :(

How to be unsad?

Let us now investigate the power of trees ($\neq$ the spirit of the forest)

How does implement

Using arrays and lists, time complexity?

`enqueue`, `dequeue` in... **linear time** $\mathcal{O}(n)$ (whether you keep things ordered or not)

Sad! :(

How to be unsad?

Let us now investigate the power of trees ($\neq$ the spirit of the forest)

(material in §4 of the textbook now covered.)