

# Introduction to **Information Retrieval**

# Applications of Crawlers

---

- Search engines
- Copyright violation
- Keyword based searches
- Web malware detection
- Web analytics

# Examples of Web crawlers

---

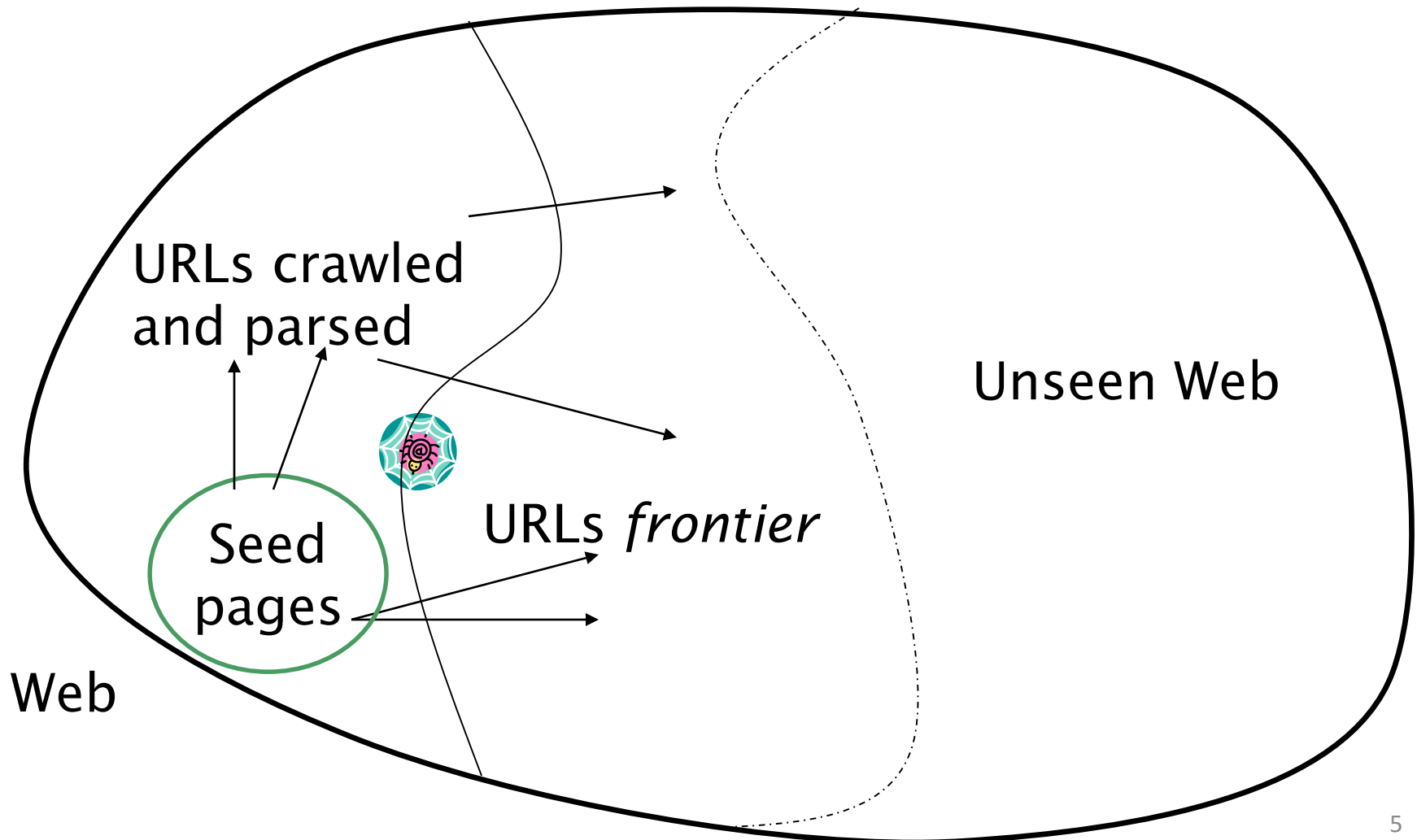
- Amazonbot is the Amazon web crawler.
- Bingbot is Microsoft's search engine crawler for Bing.
- DuckDuckBot is the crawler for the search engine [DuckDuckGo](https://duckduckgo.com/).
- Googlebot is the crawler for Google's search engine.
- Yahoo Slurp is the crawler for Yahoo's search engine.
- Yandex Bot is the crawler for the Yandex search engine

# Basic crawler operation

---

- Begin with known “seed” URLs
- Fetch and parse them
  - Extract URLs they point to
  - Place the extracted URLs on a queue
- Fetch each URL on the queue and repeat

# Crawling picture



# Component I: Crawler/Spider/Robot

- Building a “toy crawler” is easy
  - Start with a set of “seed pages” in a priority queue
  - Fetch pages from the web
  - Parse fetched pages for hyperlinks; add them to the queue
  - Follow the hyperlinks in the queue
- A real crawler is much more complicated...
  - Robustness (server failure, trap, etc.)
  - Crawling courtesy (server load balance, robot exclusion, etc.)
  - Handling file types (images, PDF files, etc.)
  - URL extensions (cgi script, internal references, etc.)
  - Recognize redundant pages (identical and duplicates)
  - Discover “hidden” URLs (e.g., truncating a long URL )

# Major Crawling Strategies

- Breadth-First is common (balance server load)
- Parallel crawling is natural
- Variation: focused crawling
  - Targeting at a subset of pages (e.g., all pages about “automobiles” )
  - Typically given a query
- How to find new pages (they may not linked to an old page!)
- Incremental/repeated crawling
  - Need to minimize resource overhead
  - Can learn from the past experience (updated daily vs. monthly)
  - Target at : 1) frequently updated pages; 2) frequently accessed pages

# Simple picture – complications

---

- Web crawling isn't feasible with one machine
  - All of the above steps distributed
- **Malicious pages**
  - Spam pages
  - Spider traps – incl dynamically generated
- Even non-malicious pages pose challenges
  - Latency/bandwidth to remote servers vary
  - Webmasters' stipulations
    - How “deep” should you crawl a site's URL hierarchy?
  - Site mirrors and duplicate pages
- **Politeness – don't hit a server too often**



# What any crawler *must* do

---

- Be Robust: Be immune to spider traps and other malicious behavior from web servers
- Be Polite: Respect implicit and explicit politeness considerations

# Explicit and implicit politeness

---

- Explicit politeness: specifications from webmasters on what portions of site can be crawled
  - robots.txt
- Implicit politeness: even with no specification, avoid hitting any site too often

# Robots.txt

---

- Protocol for giving spiders (“robots”) limited access to a website, originally from 1994
  - [www.robotstxt.org/robotstxt.html](http://www.robotstxt.org/robotstxt.html)
- Website announces its request on what can(not) be crawled
  - For a server, create a file `/robots.txt`
  - This file specifies access restrictions

# Robots.txt example

---

- No robot should visit any URL starting with `"/yoursite/temp/"`, except the robot called `"searchengine"`:

```
User-agent: *
```

```
Disallow: /yoursite/temp/
```

```
User-agent: searchengine
```

```
Disallow:
```

# What any crawler *should* do

---

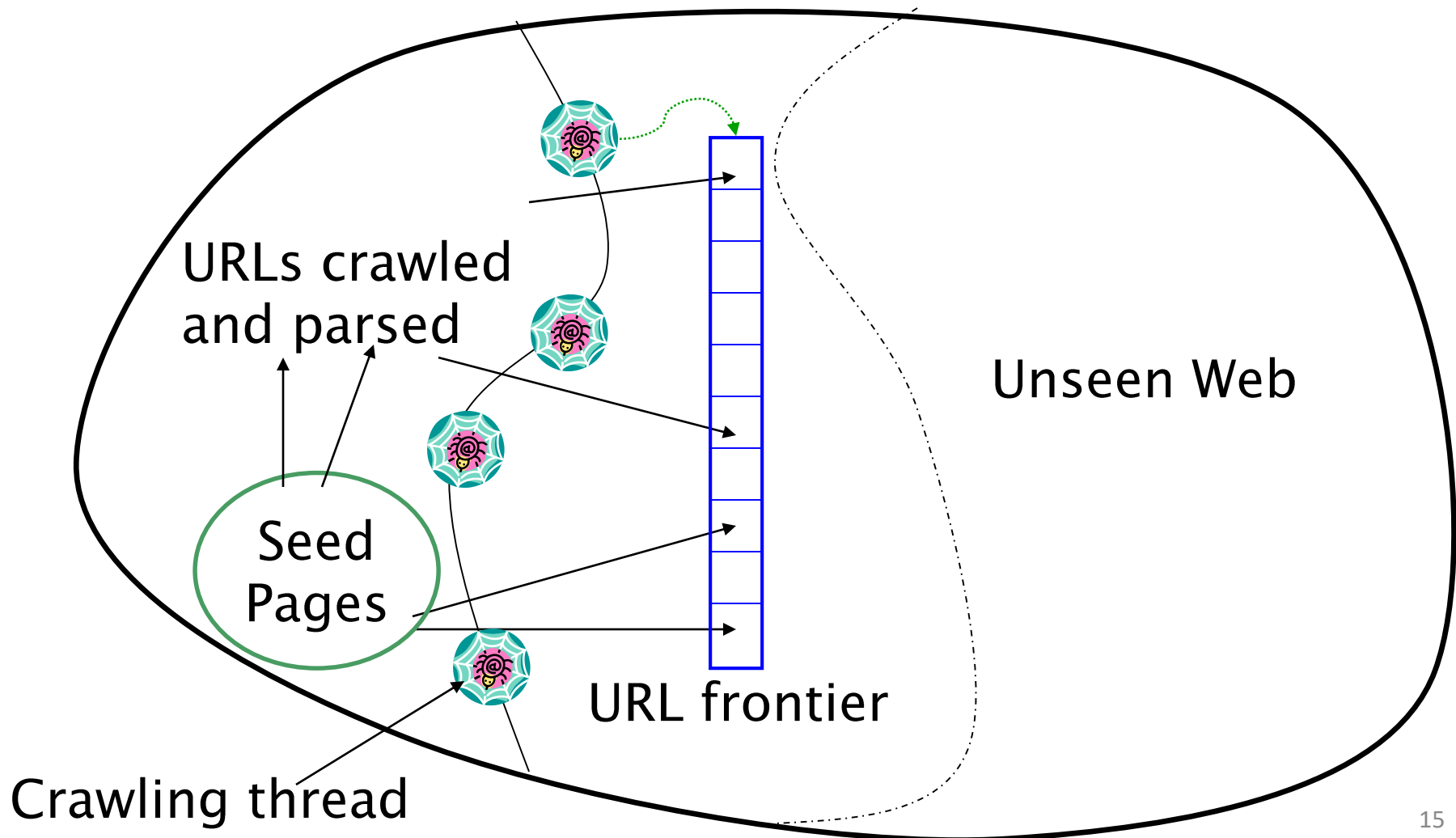
- Be capable of distributed operation: designed to run on multiple distributed machines
- Be scalable: designed to increase the crawl rate by adding more machines
- Performance/efficiency: permit full use of available processing and network resources

# What any crawler *should* do

---

- Fetch pages of “higher quality” first
- Continuous operation: Continue fetching fresh copies of a previously fetched page
- Extensible: Adapt to new data formats, protocols

# Updated crawling picture



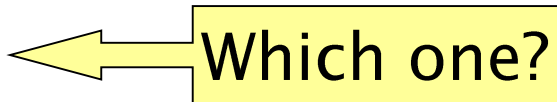
# URL frontier

---

- Can include multiple pages from the same host
- Must avoid trying to fetch them all at the same time
- Must try to keep all crawling threads busy

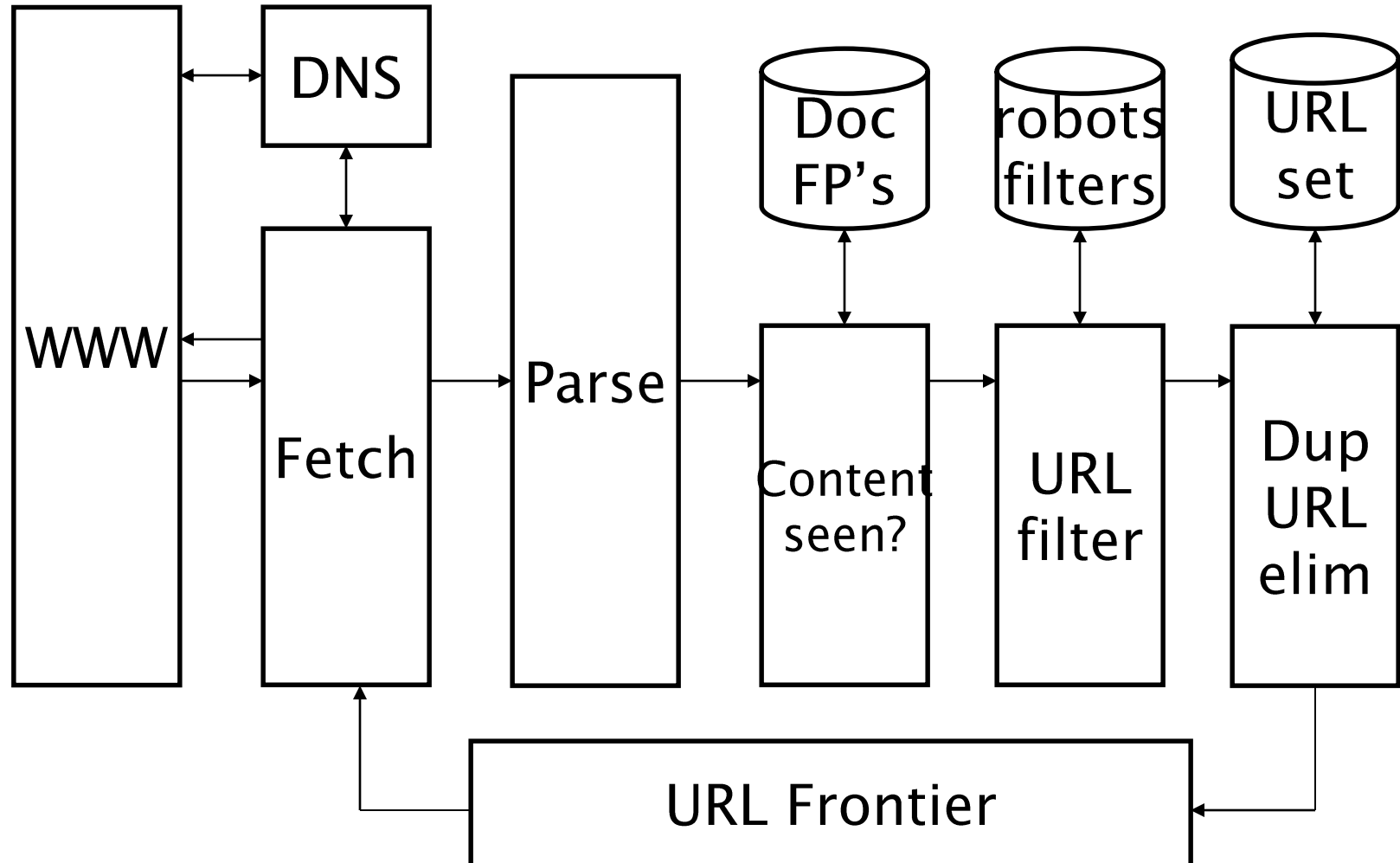


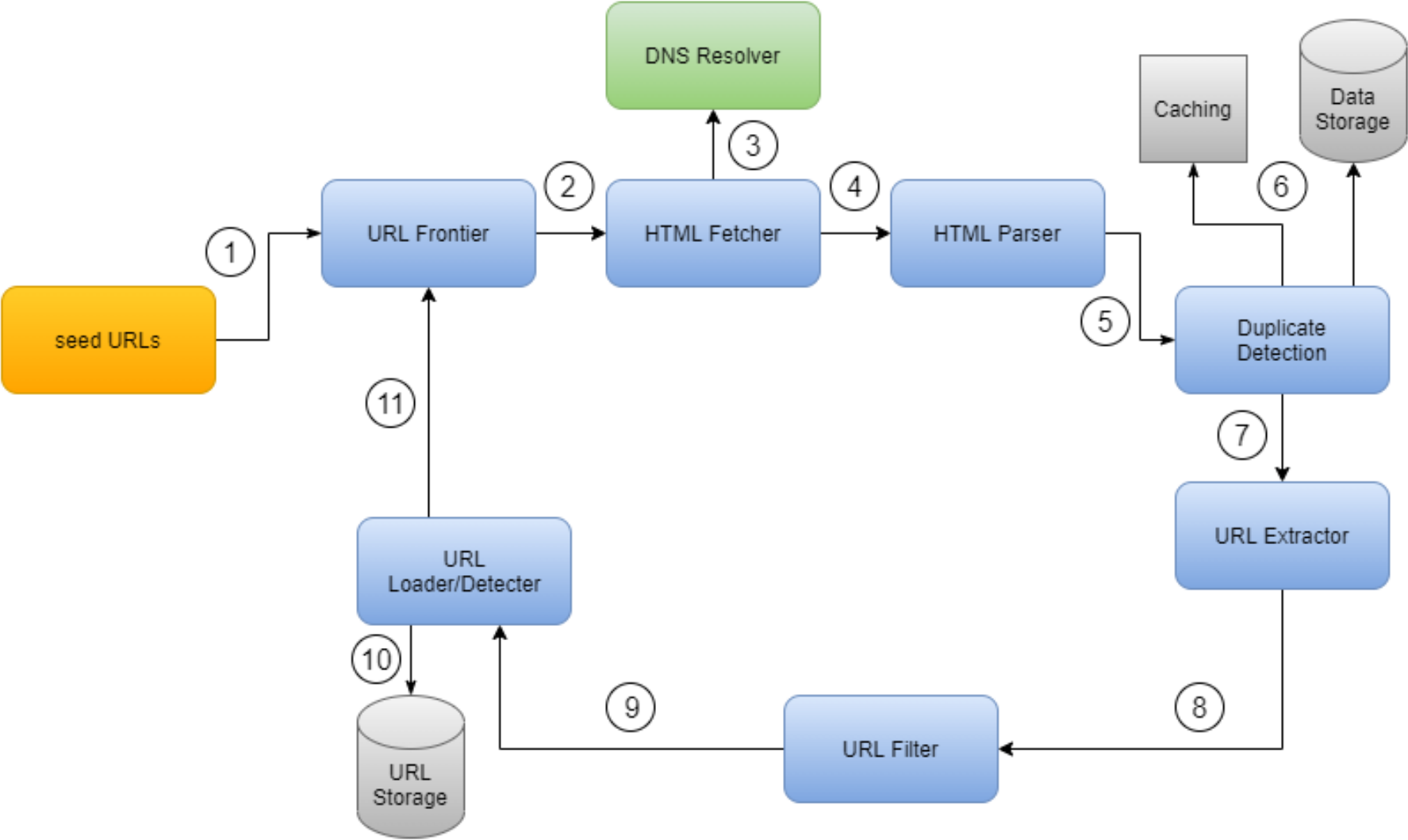
# Processing steps in crawling

- Pick a URL from the frontier  Which one?
- Fetch the document at the URL
- Parse the URL
  - Extract links from it to other docs (URLs)
- Check if URL has content already seen
  - If not, add to indexes
- For each extracted URL
  - Ensure it passes certain URL filter tests
  - Check if it is already in the frontier (duplicate URL elimination)

E.g., only crawl .edu,  
obey robots.txt, etc.

# Basic crawl architecture





# Processing Steps in Crawling

---

1. At the beginning of the cycle, worker thread sends seed URLs to the URL Frontier. URL Frontier then retrieves URLs according to its priorities and politeness policies.
2. HTML Fetcher module is called to fetch URLs from the URL Frontier
3. HTML Fetcher calls DNS resolver to resolve the host address of the web server associated with the URL
4. HTML Parser parses the HTML page and analyzes content of the page
5. The content is validated and then passed to the duplicate detection component to check for duplicity.

# Processing Steps in Crawling

---

6. The duplicate detection component checks the cache, and if the content is not found there, it checks the data storage to see if the content is already stored there
7. If content is not in the storage, web page is sent to the Link Extractor, which extracts any outgoing links from the page.
8. The extracted URLs are passed to the URL filter, which filters out unwanted URLs such as file extensions of no interest or blacklisted sites.
9. After links are filtered, they are passed to the URL Detector component.
10. URL Detector checks if a URL has already been processed and stored. If it has, no further action is needed. If URL has not been processed before, it is added to the URL Frontier to be crawled in a future work cycle.

# DNS (Domain Name Server)

---

- A lookup service on the internet
  - Given a URL, retrieve its IP address
  - Service provided by a distributed set of servers – thus, lookup latencies can be high (even seconds)
- Common OS implementations of DNS lookup are *blocking*: only one outstanding request at a time
- Solutions
  - DNS caching
  - Batch DNS resolver – collects requests and sends them out together

# Parsing: URL normalization

---

- When a fetched document is parsed, some of the extracted links are *relative* URLs
- E.g., [http://en.wikipedia.org/wiki/Main\\_Page](http://en.wikipedia.org/wiki/Main_Page) has a relative link to /wiki/Wikipedia:General\_disclaimer which is the same as the absolute URL [http://en.wikipedia.org/wiki/Wikipedia:General\\_disclaimer](http://en.wikipedia.org/wiki/Wikipedia:General_disclaimer)
- During parsing, must normalize (expand) such relative URLs

# Content seen?

---

- Duplication is widespread on the web
- If the page just fetched is already in the index, do not further process it
- This is verified using document fingerprints or shingles



# Filters and robots.txt

---

- Filters – regular expressions for URLs to be crawled/not
- Once a robots.txt file is fetched from a site, need not fetch it repeatedly
  - Doing so burns bandwidth, hits web server
- Cache robots.txt files

# Duplicate URL elimination

---

- For a non-continuous (one-shot) crawl, test to see if an extracted+filtered URL has already been passed to the frontier
- For a continuous crawl – see details of frontier implementation

## URL frontier: two main considerations

---

- Politeness: do not hit a web server too frequently
- Freshness: crawl some pages more often than others
  - E.g., pages (such as News sites) whose content changes often

These goals may conflict with each other.

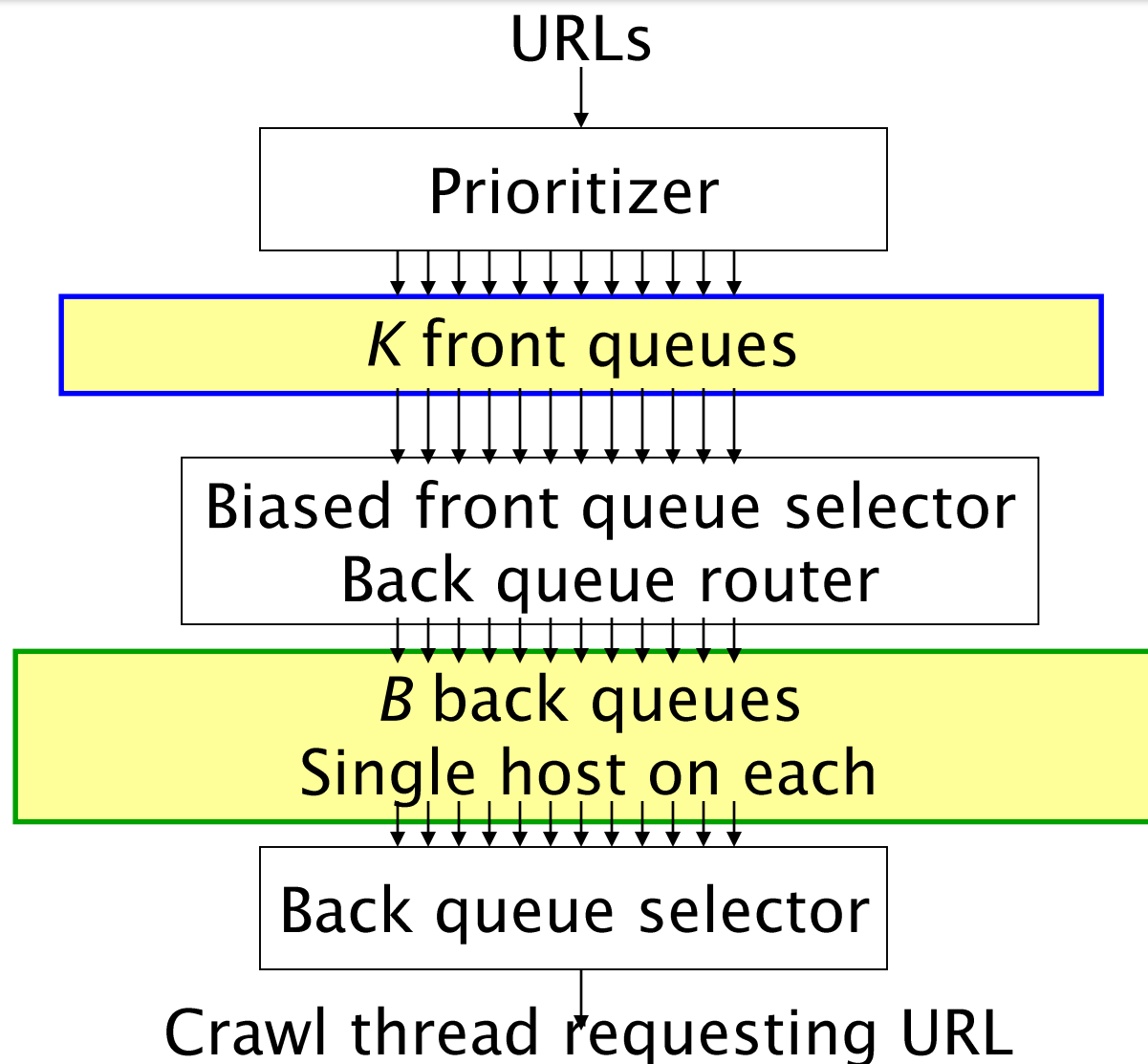
(E.g., simple priority queue fails – many links out of a page go to its own site, creating a burst of accesses to that site.)

# Politeness – challenges

---

- Even if we restrict only one thread to fetch from a host, can hit it repeatedly
- Common heuristic: insert time gap between successive requests to a host that is  $\gg$  time for most recent fetch from that host

# URL frontier: Mercator scheme

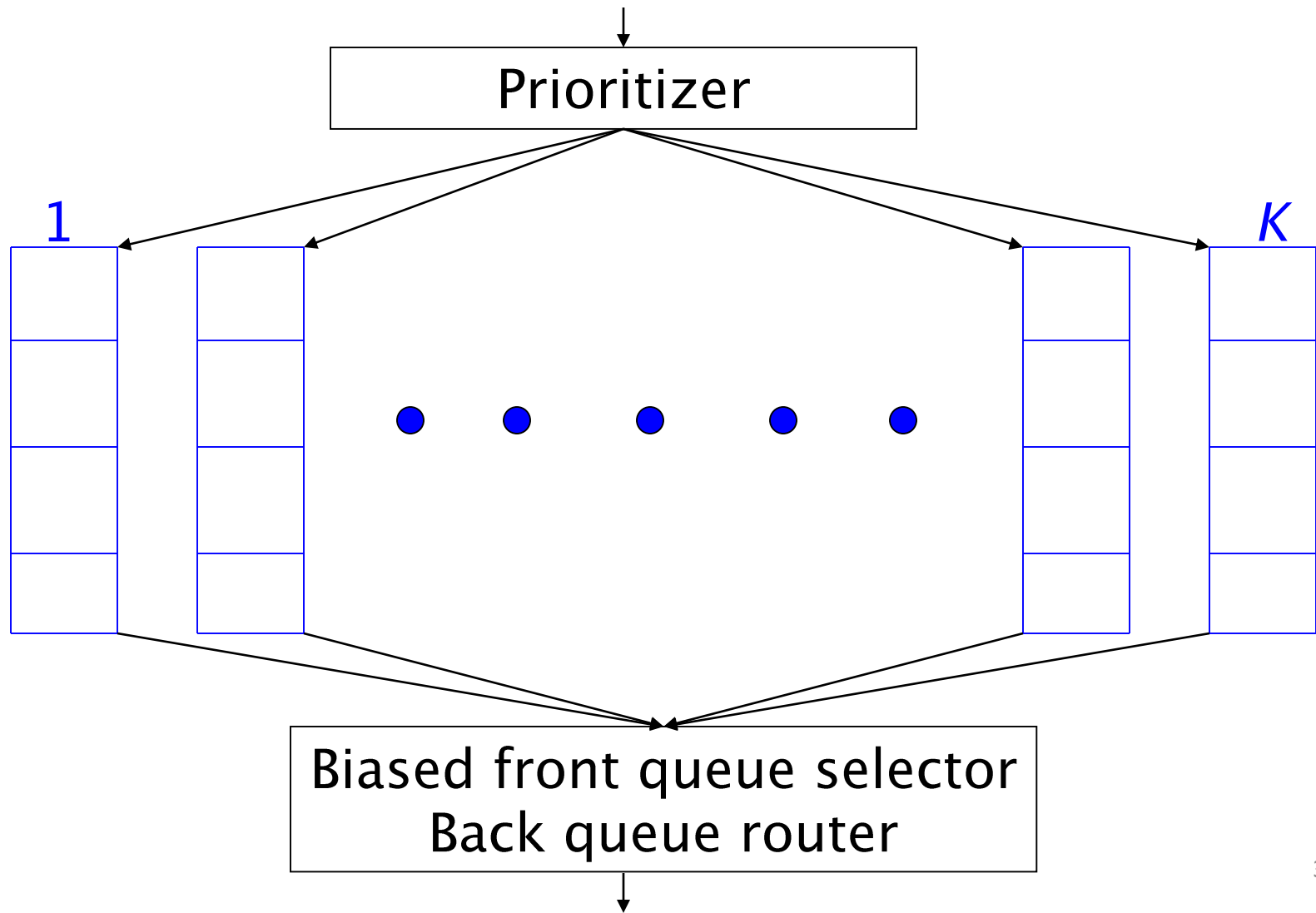


# Mercator URL frontier

---

- URLs flow in from the top into the frontier
- **Front queues** manage prioritization
- **Back queues** enforce politeness
- Each queue is FIFO

# Front queues



# Front queues

---

- Prioritizer assigns to URL an integer priority between 1 and  $K$ 
  - Appends URL to corresponding queue
- Heuristics for assigning priority
  - Refresh rate sampled from previous crawls
  - Application-specific (e.g., “crawl news sites more often”)

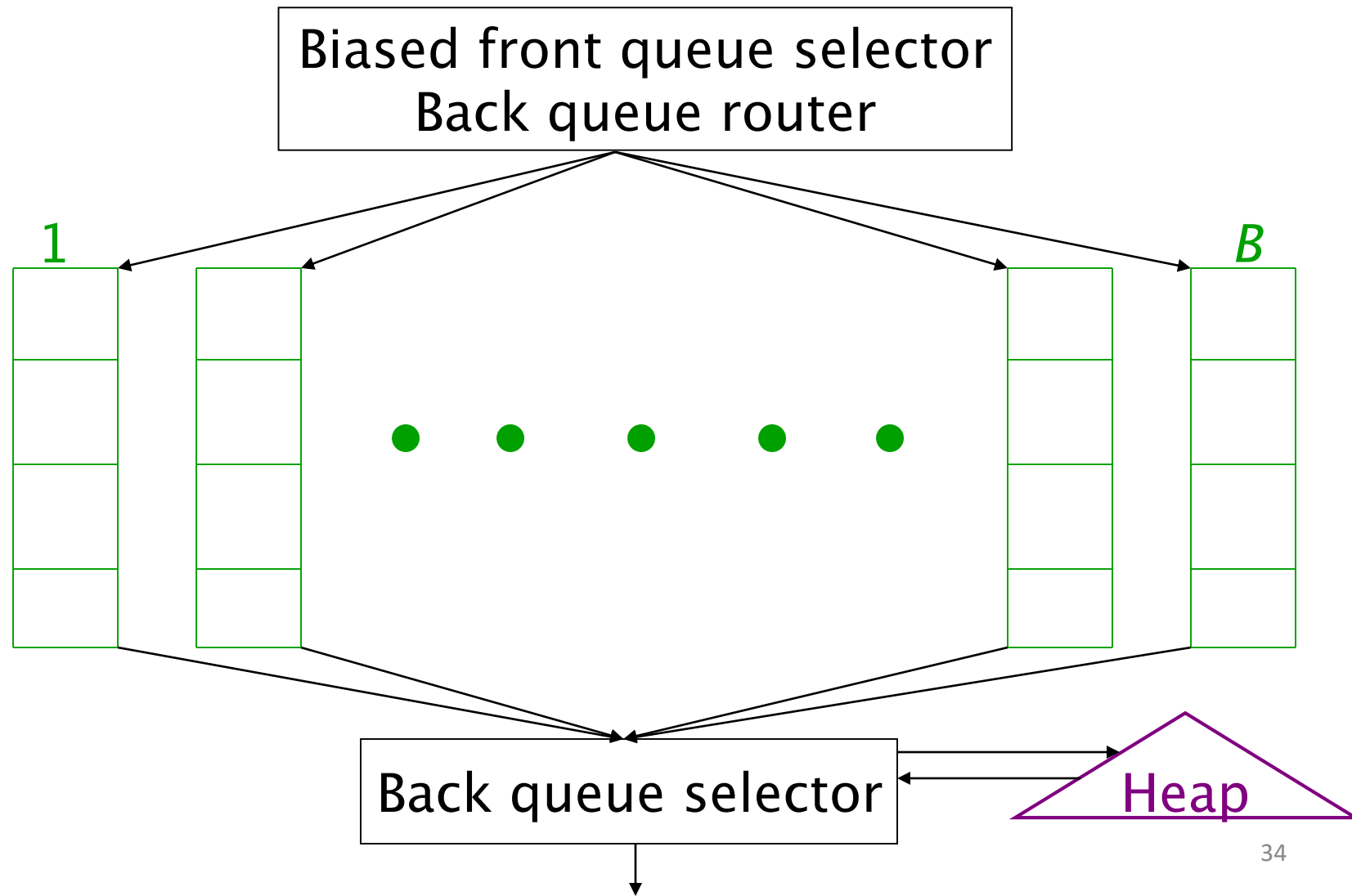


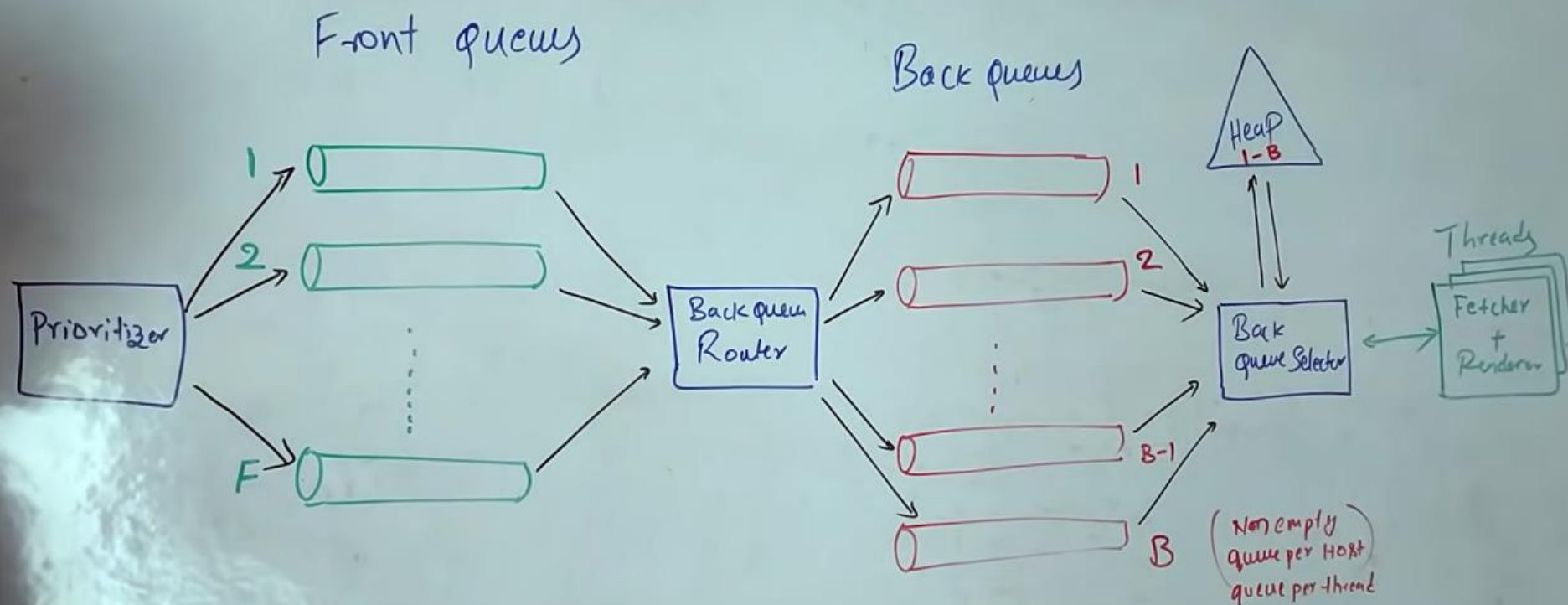
# Biased front queue selector

---

- When a back queue requests a URL (in a sequence to be described): picks a **front queue** from which to pull a URL
- This choice can be round robin biased to queues of higher priority, or some more sophisticated variant
  - Can be randomized

# Back queues





→ Priority / Recrawl / Freshness

→ Politeness

HOST	Back Queue ID
Digit.com	5
Techncrunch.com	1
Youtube.com	17

# Back queue invariants

- Each back queue is kept non-empty while the crawl is in progress
- Each back queue only contains URLs from a single host
  - Maintain a table from hosts to back queues

Host name	Back queue
...	3
	1
	<i>B</i>

# Back queue **heap**

---

- One entry for each back queue
- The entry is the earliest time  $t_e$  at which the host corresponding to the back queue can be hit again
- This earliest time is determined from
  - Last access to that host
  - Any time buffer heuristic we choose

# Back queue processing

---

- A crawler thread seeking a URL to crawl:
- Extracts the root of the heap
- Fetches URL at head of corresponding back queue  $q$  (look up from table)
- Checks if queue  $q$  is now empty – if so, pulls a URL  $v$  from front queues
  - If there's already a back queue for  $v$ 's host, append  $v$  to it and pull another URL from front queues, repeat
  - Else add  $v$  to  $q$
- When  $q$  is non-empty, create heap entry for it

# Number of back queues $B$

---

- Keep all threads busy while respecting politeness
- Mercator recommendation: three times as many back queues as crawler threads