**Frontier**

**MAX\_URL\_SIZE** = 256 *for robustness, avoids too long URLs (symptom of spider traps)*

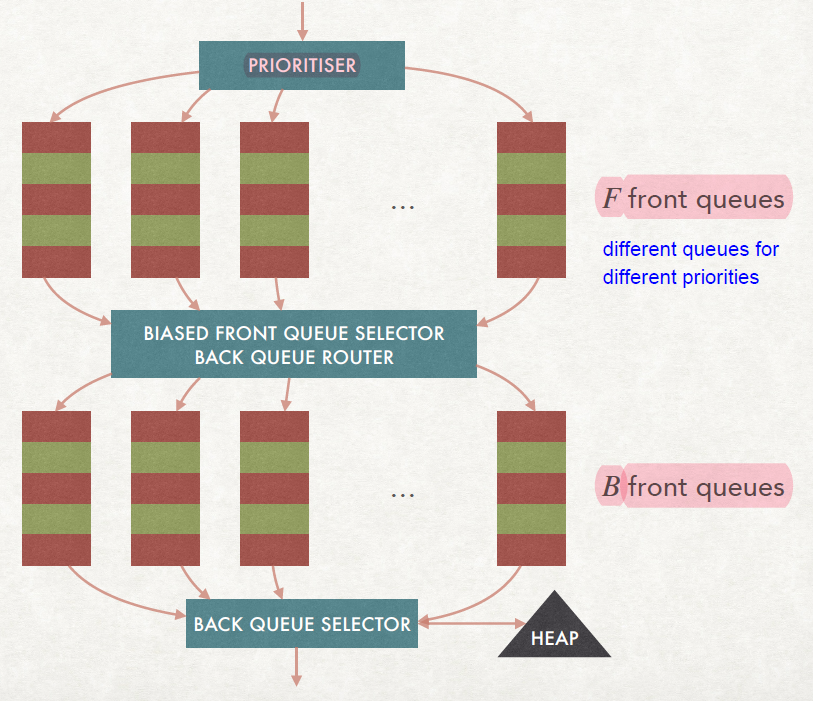
**MIN\_WAIT\_TIME\_MILLIS** = 200 *for fairness, avoids bombarding the host*

*Only used if the wait time (calculated as response time \* 10) is too short*

**MAX\_URLS\_PER\_HOST** = 100 *for robustness, not more than this number of URLs can be visited per host*

**WAIT\_BEFORE\_RETRY\_MILLIS** = 20000 *if all front queues are empty, we wait this before retrying*

**MAX\_WAIT\_ATTEMPTS** = 2 *if all front queues are empty, we retry at most this many times*



**Prioritiser prioritiser**

*Hand defined object -> see the yellow section*

**ArrayList<ConcurrentLinkedQueue<URI>> frontQueues**

*Arraylist (or simply an ordered list) of queues. Each queue is a concurrentLinkedQueue which is a thread safe Java implementation of a queue. Each queue contains URIs.*

**ConcurrentHashMap<String, ConcurrentLinkedQueue<URI>> backQueues**

*The queues are organised in a hashmap, since each queue only contains URIs of a single host, its address is the hash of the hostname.*

**PriorityBlockingQueue<HeapEntry> heap**

*Data structure that acts like a heap, keeping elements ordered according to the criterion specified in the HeapEntry class definition. Objects can only be popped from the top of the heap. The heap is internally kept ordered upon insertion.*

**Set<URI> pendingUrls**

*Set (no duplicates, no order) of all the URIs which are currently in any queue OR are being visited, but whose crawl has not been completed yet. This structure is necessary to prevent duplication due to multithreading: two examples of risky situations without this structure follow.*

*Example 1: thread 1 wants to insert page A in the frontier, thread 2 wants to do the same, thread 1 checks whether page A is in the front or back queues, page A is not there yet; thread 2 checks the same: page A is still not there. Thread 1 adds page A to a queue, same does thread 2 so now page 1 is duplicated. With the pendingUrls data structure, instead, thread 1 inserts page A and gets true, since it’s not present anywhere, so it can confidently add it to the desired queue; when thread 2 tries to do the same, however, gets false from the attempted insertion into pendingUrls and aborts further insertions, avoiding duplication.*

*Example 2: thread 1 finds url A and tries to add it to the frontier. However, thread 2 has just removed it from the frontier and is about to visit this page. With pendingUrls, thread 1 can check and notice that the page is in the frontier and does not care whether it’s in the queues or is being fetched.*

**ConcurrentHashMap<String, Integer> urlsPerHost**

*Structure which keeps track of the amount of URIs for each host are in the frontier now.*

METHODS

**Constructor Frontier** (int numberOfFrontQueues, String... seedPages)

*Creates all the needed structures with suitable constructors and inserts seedPages into the front Queues using the* insertURL method*.*

public void insertSeenURLToRefresh(URI uri) {

//only for absolute links!

if (!uri.isAbsolute()) {

return;

}

//check if the protocol is http or https

if (uri.getScheme().equals("http") || uri.getScheme().equals("https")) {

if (pendingUrls.add(uri)) {

// here I should give very high priority

prioritiser.addToQueue(uri, frontQueues);

}

}

}

**public void insertURLS(Set<String> urls)**

*Inserts each URL into the front Queues using the* insertURL method*.*

**public void insertURL(String url)**

*Checks if the url is not longer than MAX\_URL\_SIZE and makes sure it is a well-constructed absolute URI, with HTTP(S) protocol. Then, proceeds with a synchronised part (no other threads can execute this piece of code simultaneously) where the number of URIs per host is updated (only if below the threshold of MAX\_URLS\_PER\_HOST), the URI is added pendingUrls and is inserted into a frontQueue through the prioritiser.addToQueue(uri, frontQueues) method.*

**public URI getNextURL()**

//if heap is not empty

//extract the one with minimum date (remove it from heap)

HeapEntry heapEntry = heap.poll();

// if the heap is empty, we need to move links from the front to the back

// which implies adding an host to visit in the heap as well

// then we will be able to draw from the heap again (if another thread

// has emptied the thread before this no problems occur, we just restart)

if (heapEntry == null) {

try {

moveFromFrontQueueToBackQueue();

} catch (EmptyFrontQueuesException e) {

e.printStackTrace();

return null;

}

return getNextURL();

}

String host = heapEntry.getHost();

// check if wait time is respected, if it's not then wait

// could probably be improved +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

try {

Thread.sleep(Math.max(0, heapEntry.getNextVisitTime().getTime() - (new Date()).getTime()));

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

throw new AssertionError(e);

}

//check that host's queue

ConcurrentLinkedQueue<URI> hostQueue = backQueues.get(host);

//if not empty return first

if (hostQueue != null && !hostQueue.isEmpty()) {

//before returning the URL, update the heap for that host

URI uri = hostQueue.poll();

if (uri != null) {

return uri;

}

}

try {

moveFromFrontQueueToBackQueue();

} catch (EmptyFrontQueuesException e) {

//e.printStackTrace();

}

return getNextURL();

}

private void moveFromFrontQueueToBackQueue() throws EmptyFrontQueuesException {

//draw URL from one of the non empty frontQueues

URI uri = drawURIFromFrontQueue();

String host = uri.getHost();

if (host == null) {

return;

}

// if there's no value for the host in the back try to create a queue

if (!backQueues.containsKey(host)) {

ConcurrentLinkedQueue<URI> backQueue = new ConcurrentLinkedQueue<>();

backQueue.add(uri);

// extra check to make sure no other thread has created the B queue in

// the meanwhile -> if it happened act as if the queue was already there before

if (backQueues.putIfAbsent(host, backQueue) == null) {

// add the host to the heap so that the B queue is accessible

addToHeap(host);

return;

}

}

// get a reference to the queue and add the new uri to it

ConcurrentLinkedQueue<URI> backQueue = backQueues.get(host);

backQueue.add(uri);

}

// could suffer from cold start problem as at the beginning

// many front queues will be empty causing long search

private URI drawURIFromFrontQueue() throws EmptyFrontQueuesException {

URI uri = prioritiser.selectQueueToDrawFrom(frontQueues).poll();

// if that queue was empty draw another one

if (uri == null) {

int waitAttempts = 0;

ConcurrentLinkedQueue<URI> frontQueue;

while (true) {

// we use this draw to avoid bouncing between queues before finding a full one

// and also because if they are all empty this returns null

frontQueue = prioritiser.selectFirstNonEmptyQueueToDrawFrom(frontQueues);

// we found a non empty queue and we managed to draw a non null url from it

if (frontQueue != null && (uri = frontQueue.poll()) != null) {

break;

} else {

// if we reach the max allowed attempts we throw an exception and we stop

if (waitAttempts == MAX\_WAIT\_ATTEMPTS) {

throw new EmptyFrontQueuesException("All front queues are empty, impossible to draw url");

} else {

//we wait and we redo this procedure

try {

Thread.sleep(WAIT\_BEFORE\_RETRY\_MILLIS);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

throw new AssertionError(e);

}

}

}

}

}

return uri;

}

// this method adds a new host to heap whenever such host is discovered

// to make sure we respect delays, before adding the host to the heap

// with 0 delay time, a check on whether such host is already in the heap

// is performed (if it's already in the heap nothing is done)

private synchronized void addToHeap(String host) {

HeapEntry entry = new HeapEntry(host);

if (!heap.contains(entry)) {

heap.add(entry);

}

}

// given an host sets the next time we will be able to visit it

// this method is to be used whenever the host is contacted and

// we set a delay for visiting it next time, is the host is already

// in the heap (which means it has a shorter delay set), we delete it

// and insert the new version with longer waiting time

public synchronized void updateHeap(String host, long delayMillis) {

if (delayMillis < MIN\_WAIT\_TIME\_MILLIS) {

delayMillis = MIN\_WAIT\_TIME\_MILLIS;

}

HeapEntry entry = new HeapEntry(host, delayMillis);

heap.remove(entry);

heap.add(entry);

}

public void updateHeap(String host) {

updateHeap(host, MIN\_WAIT\_TIME\_MILLIS);

}

public void removeFromPending(URI uri) {

pendingUrls.remove(uri);

}

}