Programming Assignment 1 (Updated, Oct 1)
Points: 500
Due: Oct 11, 11:59PM
Late Submission Due: Oct 12, 11:59PM (25% penalty)

Description of a programming assignment is not a linear narrative and may require multiple readings before things start to make sense. You are encouraged to start working on the assignment as soon as possible. If you start a day or two before the deadline, it is highly unlikely that you will be able to come with correct and efficient programs.

You are encouraged to consult instructor/Teaching Assistants for any questions/clarifications regarding the assignment. Your programs must be in Java, preferably Java 8.1. You should not use any external libraries. Any libraries that you import must start with `java.`

For this PA, you **may work in teams of 2**. It is your responsibility to find a team member. If you can not find a team member, then you must work on your own. **Only one submission per team**.

In this programming assignment you will

- Implement a priority Queue (Max Heap) to store Strings (along with their priorities).
- Implement (variants of) BFS algorithm to crawl wiki pages and construct a partial wiki graph.

You will design following classes:

1. PriorityQ
2. WikiCrawler

All your classes must be in the **default package** (even though it is not a good programming practice).

1 PriorityQ

Read Section 2.5 from the text to refresh your understandings about priority queues. A priority queue stores *data items* along with their *priority* (often called *key* of the item). If the data item is v and its priority is p, then the tuple $\langle v, p\rangle$ is store in the priority queue. For any such tuple $t = \langle v, p\rangle$ we say that $key(t) = p$ and $val(t) = v$. In this PA, you will implement a priority queue using max heap, which is implemented as an array A, where the max heap property is that for every i (array index starts at 1),

$$key(A[i]) \geq key(A[2i]) \text{ and } key(A[i]) \geq key(A[2i+1])$$

In this assignment, the data items are *strings*, where each string has an associated priority.

Your Objective. Implement a class `PriorityQ` with the following methods

1. `PriorityQ()`: constructs an empty priority queue.
2. `add(String s, int p)`: Adds a string `s` with priority p to the priority queue.
3. `returnMax()`: returns a string whose priority is maximum.
4. `extractMax()`: returns a string whose priority is maximum and removes it from the priority queue.
5. `remove(int i)`: removes the element from the priority queue whose array index is i.
6. `decrementPriority(int i, into)`. Decrements the priority of the ith element by k.
7. `priorityArray()`: returns an array `B` with the following property: `B[i] = key(A[i])` for all i in the array A used to implement the priority queue.
8. `getKey(int i).` Returns $key(A[i])$, where A is the array used to represent the priority queue
9. `getValue(int i).` Returns $value(A[i])$, where A is the array used to represent the priority queue
10. `isEmpty().` Return `true` if and only if the queue is empty.

The max heap property must be maintained after each of the above operation. Methods 3–7 have a pre-condition that priority queue is non-empty.

2 BFS and Web Graph

In this problem, you will generate a exploration graph. Recall that given a graph, the BFS algorithm is as follows.

```
1. Input: Directed Graph G = (V, E)
   Root of exploration: root
2. Initilize a FIFO queue Q and a list Discovered
3. Add root to Q and Discovered
4. while Q is not empty do
5.    Extract vertex v from the head of the Q
6.    For every (v, v') in E do
7.       if v' is not in Discovered then
8.          add v' to Q and Discovered
```

If you output the vertices in the `Discovered` list, then that will produce the BFS traversal of the input graph starting from the root.

We can the model web as a directed graph. Every web page is a vertex of the graph. We put a directed edge from a page p to a page q, if page p contains a link to page q. Given a root (we will call it seed URL of the web graph), we can explore this web graph using BFS algorithm as presented above.

2.1 Crawling and Constructing Web Graph

One of the first tasks of a web search engine is to *crawl* the web to collect material from all web pages. The crawl may be performed in a BFS fashion. While doing this, the the search engine will also construct the *web graph*. The structure of this graph is critical in determining the pages that are relevant to a query.

As part of this assignment you will write a program to do *crawl* of the web and construct web graph. However, web graph is too large, and you do not have computational resources to construct the entire web graph (unless you own a super computer). Thus your program will crawl and construct the web graph over 200 pages. Furthermore, Your program will crawl only *Wiki* pages.

Relevance of a page. Before we proceed, let us discuss the notion of *focused crawling*. In certain application, you would like to visit only the web pages that are related to certain topic of interest. For example, if I would like to collect only the web pages that are related to the *Iowa State University*, then any page that is not about *Iowa State University* is not of interest to me. Furthermore, my goal is not collect *every* web page about Iowa State University. My goal would be to collect top, say 200, pages that are *most relevant* to Iowa State University. For this, we need a way to determine the relevancy of a page to the topic of my interest.

Let T be a set of strings that describe a topic. For example, for my topic "Iowa State University", the strings that describe this topic could be "Iowa State University, ISU, Cyclones, Ames, Atanasoff". Given a web address a, let $page(a)$ denote the contents of the page at address a. Given a web address a and a string s, let $f(s, a)$ denote the number of times the string s appears in $page(a)$. *This is case sensitive.* For a T, the relevancy of a web address a to the topic T is defined as":

$$Relevance(T, a) = \sum_{s \in T} f(s, a)$$

Your Objective. Implement a class `WikiCrawler` with the following methods.

1. The constructor

```
WikiCrawler(String seed,
            int max,
            String[] topics,
            String output)}
```

   where

   (a) `seed`: related address of seed URL (within wiki domain)

   (b) `max`: maximum number of pages to consider

   (c) `topics`: array of strings representing keywords in a topic-list

   (d) `output`: string representing the filename where the `web graph` over discovered pages are written.

2. `ArrayList<String> extractLinks(string document)`: the method takes as input a document representing an entire HTML document. It returns a list of strings consisting of links from the `document`. You can assume that the `document` is HTML from some wiki page. The method must

   (a) extract only relative addresses of wiki links, i.e., only links that are of the form `/wiki/XXXX`

   (b) only extract links that appear after the first occurrence of the html tag `<p>` (or `<P>`)

   (c) Must not extract any wiki link that contain characters such as "#" or ":"

   (d) The order in which the links in the returned array list must be exactly the same order in which they appear in the `document`

3. `crawl(boolean focused)`: crawls/explores the web pages starting from the seed URL. Crawl the **first** `max` number of pages (including the `seed` page), that contains every keywords in the `Topics` list (if `Topics` list is empty then this condition is vacuously considered true), and are explored starting from the `seed`.

   (a) if `focused` is false then explore in a BFS fashion

   (b) if `focused` is true then for every page a, compute the $Relevance(\texttt{T}, a)$, and during exploration, instead of adding the pages in the FIFO queue,

   - add the pages and their corresponding relevance (to topic) to priority queue. The priority of a page is its relevance;
   - extract elements from the queue using `extractMax`.

   After the crawl is done, the edges explored in the crawl method should be written to the `output` file.

2.2 Guidelines, Clarifications and Suggestions for Part 2

1. Review the enclosed sample outputs to better understand the result of crawling.

2. Here is an example. Suppose that there following pages. $A, B, C, D, E, F, G, H, I, J$. Page A has links to B, C and D appearing in that order. Page C has links to E, F, B and D. Page D has links to G, H and A. Page B has links to I and J. Page E has a link to page A. None of the other pages have any links. When `focused` is false, the `seed` URL is A, and `max` is 6, the the constructed graph will have the following edges: A to B, A to C, A to D, B to I, B to J, D to A, C to B, and C to D. Thus the output file looks like

```
6
A B
A C
A D
B I
B J
D A
C B
C D
```

First line denotes the number of web pages discovered. Each subsequent line lists an edge.

You may test the your program on the above test case by changing your base url to `http://web.cs.iastate.edu/~pavan`, seed as `/wiki/A.html`, and `max` as 6. If you cut and paste the this link, please retype ~ symbol.

3. The seed url is specified as *relative address*; for example `/wiki/Iowa_State_University` not as `https://en.wikipedia.org/wiki/Iowa_State_University`.

4. Extract only links from "actual text component". A typical wiki page will have a panel on the left hand side that contains some navigational links. Your program should not extract any of such links. Wiki pages have a nice structure that enables us to do this easily. For the purpose of this PA, the "actual text content" of the page starts immediately after the first occurrence of the html tag `<p>`. So, your program should extract links (pages) that appear after the first occurrence of `<p>`. Your program must extract the links **in the order they appear in the page, and when `focused` is false place them in queue in that order**.

   For the purpose of this PA, "actual text component" is defined as the content that appears after the first `<p>` tag

5. While computing the relevance of a page to *topics*, inre the content that appears before the first occurrence of `<p>`.

6. The graph constructed **should not have self loops nor it should have multiple edges between same pair of nodes.** (even though a page might refer to itself or refer to another page multiple times).

7. If you wish you may take a look at the graph our crawler constructed (With `/wiki/Complexity_theory` as root and 100 as maximum number of pages, and empty list as topics) This graph has 847 edges. The graph is attached along with the assignment (named `wikiCC.txt`). Crawl Date: Sep 21, 9:50AM. Please note the format of the file. Before you ask any questions such as "why does this graph contain only 847 edges", "the graph I constructed has way more edges/pages", please understand Item 2.

8. Crawlers place considerable load on the servers (from which they are requesting a page). If your program continuously sends request to a server, you may be denied access. Thus your program must adhere to the following politeness policy: Wait for at least 3 seconds after every 20 requests. **If you do not adhere to this policy you will receive ZERO credit**. Please use `Thread.sleep()` for waiting. **Absolutely no exceptions to this policy.**

9. **Your class `WikiCrawler` must declare a `static, final` global variable named `BASE_URL` with value `https://en.wikipedia.org` and use in conjunction with links of the form `/wiki/XXXX` when sending a request fetch a page. Otherwise, you will receive ZERO credit. No exceptions**.
   For example, your code to fetch page at `https://en.wikipedia.org/wiki/Physics` could be

```
URL url = new URL(BASE_URL+"/wiki/Physics");
InputStream is = url.openStream();
BufferedReader br = new BufferedReader(new InputStreamReader(is));
```

10. Your program should not use any external packages to parse html pages, to extract links from html pages and to extract text from html pages. You can only use the package `java.net` for this.

11. If most of you work in the last hour and start sending requests to wiki pages, it is quite possible that wiki may slow down, or deny service to any request from the domain `iastate.edu`. This can not be an excuse for late submissions or extending the deadline. You are advised to start working on PA as soon as possible.

12. Your program must strictly adhere to the specifications. Class, method names, type and order of parameters to the methods, and the return types of methods must be exactly as specified. Any deviation from the specification will be considered incorrect programming and lead to deduction of significant portion of points.

13. Your grade depends on correctness, efficiency and adherence to specifications.

14. Any questions posted on Piazza within 24 hours prior to the submission due time may not be answered.

15. Your program must not use any of Java's in built classes for priority queues.

3 What to Submit

Your submission must have following files.

- `PriorityQ.java`
- `WikiCrawler.java`
- `YourNetId-PA1-ReadMe.txt` (list the name(s) of team member(s)).

You must include any additional helper classes that you designed. Please include only source .java files, do not include any .class files. Please remember to use default package for all the Java classes. Place all the files that need to be submitted in a folder (without any sub-folders), name it `YourNetId-PA1` and `zip` that folder. Submit the `.zip` file. Name of the zip file must be `YourNetID-PA1.zip`. Please include all team members names as a JavaDoc comment in each of the Java files. **Only one submission per team is necessary.**