CS 1027b - Computer Science Fundamentals II
Assignment 3 – 10%
Due Monday July 16 at 11:55 PM

1 Overview

Your task in this assignment is to build a simple web crawler to find specific keywords on a collection of web pages. A web crawler is a program that starts at a web page (the *seed URL*), scans the contents of the page for links, and then crawls to each of the linked pages, repeating the process all over again. At each page visited, an action might be taken by the crawler, such as examining the text of the page and perhaps storing it in a database for later querying. Search engines like Google or Yahoo! use this sort of approach to index the trillions web pages (see http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html) available on the Internet so that we, as users, can access information in seconds.

You will need additional files than are linked to on the assignment page on WebCT. This assignment makes use of stacks, queues, lists and iterators which we have discussed (or will be discussing) in class. You will need all associated files for these data structures (including interfaces and exception classes where necessary). It is your responsibility to download all necessary files. They can all be found on WebCT in the appropriate Additional Code or Labs folder. Many of these files are missing necessary methods (such as isEqual, toString, etc) that have been left as exercises for your practice as has been mentioned in class. You will need to fill in any empty methods in order to use the classes in this assignment.

2 Learning Outcomes

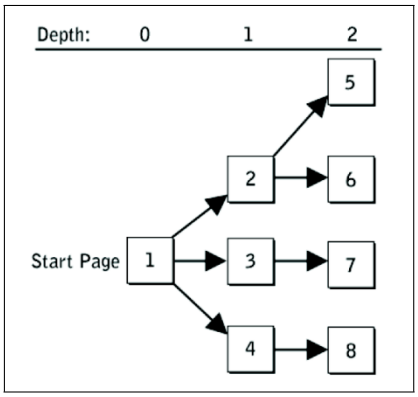
This assignment will provide you with experience in:

- Creating and using stacks and queues
- Inheritance, polymorphism, and overriding
- Using the public API of a set of classes provided to you
- Handling exceptions and testing code thoroughly
- Javadoc commenting and good Java coding style
- Appreciating, at a very high level, the complexity involved in the search engines we use

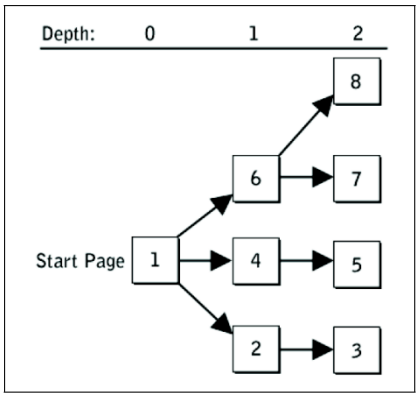
3 Background

In this assignment, you will implement two different web crawler classes: BreadthCrawler and DepthCrawler. Each crawler will have the same goal of finding a keyword in a set of web pages, but will differ in how it performs its search. The BreadthCrawler will visit web pages using a *breadth-first search*. In this type of search, we begin at the starting address or *seed URL*, and we visit each of its linked pages in order. After visiting all of its linked pages, we then move on and visit each of the links found on those linked pages. The diagram below illustrates a breadth-

first search, with the numbers on each page indicating the order in which the page was visited.



Breadth-First Search



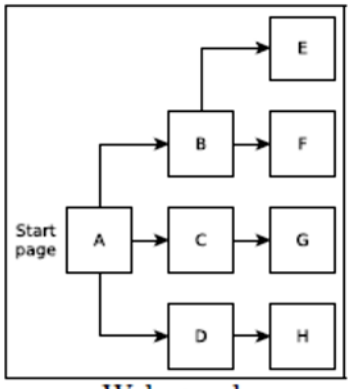
Depth-First Search

The DepthCrawler will visit web pages using a *depth-first search*. Here, the crawler begins at the start page, and visits its first linked page. It continues to explore as far down as it can go until it hits a page with no links or reaches some specified maximum depth. It then backtracks and returns to the most recent page it has not finished exploring, and continues its search. The diagram on the previous page illustrates a depth-first search, again with the numbers indicating the search order. Note that the algorithm first visits the last link on each page. This is because, as we shall see later, a stack is used to store the pages that we still have yet to visit, and therefore the order in which pages are visited is reversed.

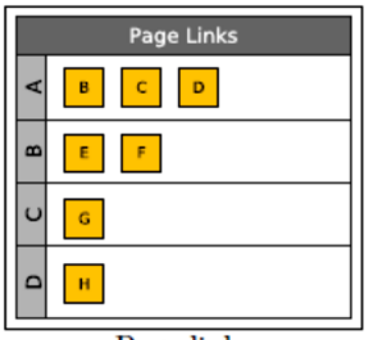
Notice in the diagrams on the previous page that there is a search depth associated with each level of pages. We say that the starting page is at depth 0. If a page is at depth k, then each of its immediately linked pages are said to be at depth $k + 1$. Because the web is so large, we wish to limit the depth to which our crawlers will search so that they do not continue searching for too long. Therefore, as you will see later, we will provide a means to limit the depth to which our crawlers will search.

3.1 Understanding Breadth-First Search

Consider the following web graph. The figure on the right summarizes the links found in the graph. For instance, the first link found on page A is to page B, the second link on page A is to page C, and so on.



Web graph

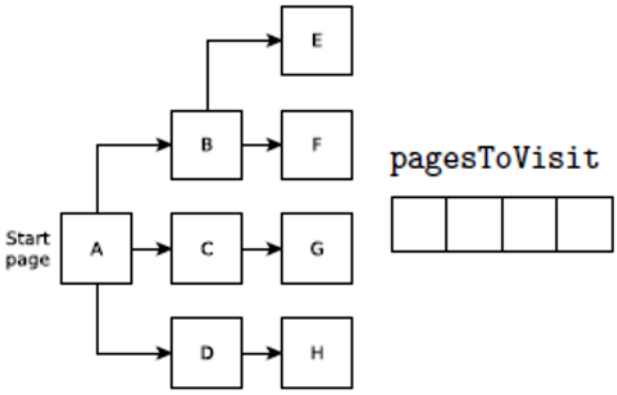


Page links

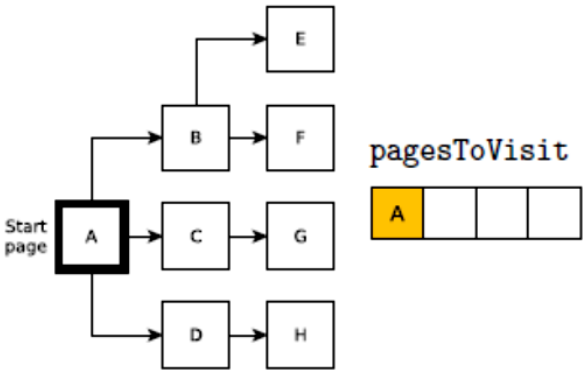
When crawling pages, a good data structure to use might be a queue. We could store Page objects in a *pagesToVisit* queue, corresponding to pages that we have yet to visit. Each time we visited a page, we could then enqueue all the links on the page in *pagesToVisit*. Let's examine the order in which we crawl pages if we use a queue as our data structure. As shown in the diagram above, assume that we begin crawling at page A.

In the graphs below, a shaded node represents a page that has been visited. A heavily outlined node represents a page that is currently enqueued in *pagesToVisit*. Finally, a number inside a node represents the order in which it was visited.

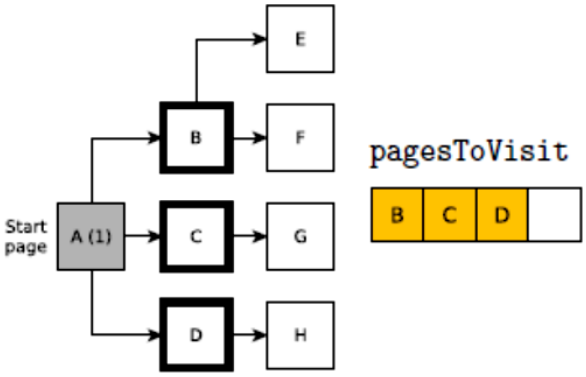
Initial: No pages visited, empty queue



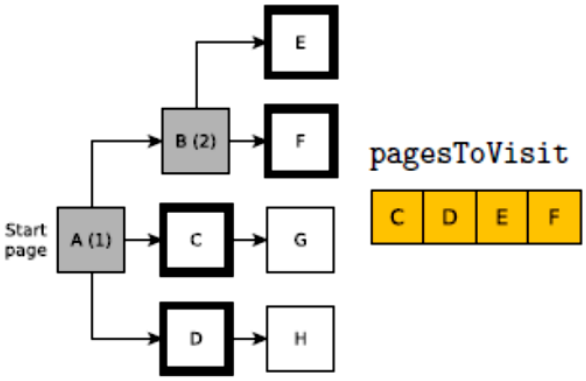
Step 1: Enqueue start page (A). No pages visited



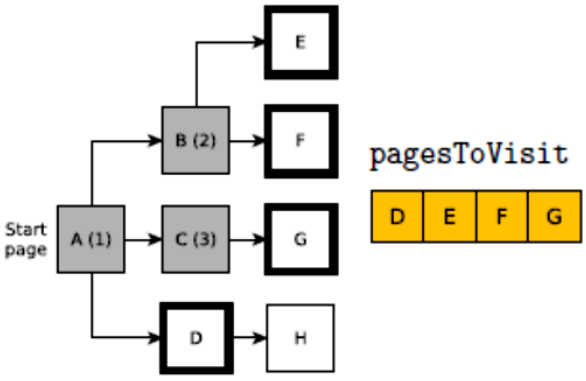
Step 2: Visit A, enqueue B, C, D



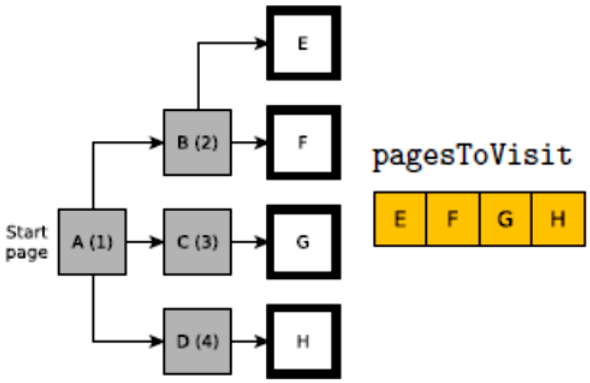
Step 3: Visit B, enqueue E, F



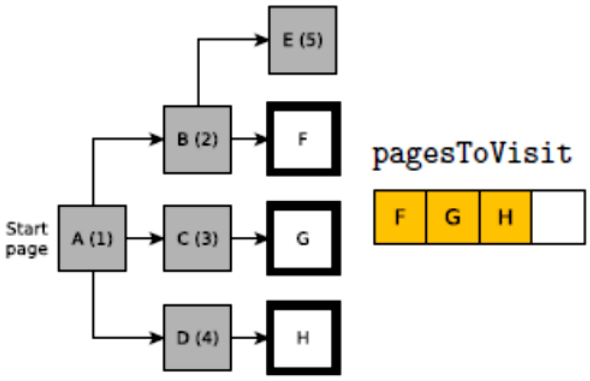
Step 4: Visit C, enqueue G



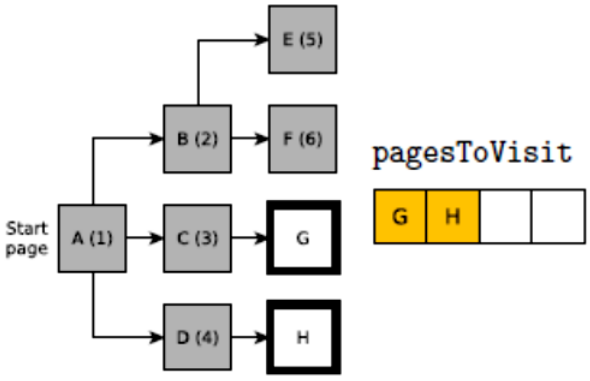
Step 5: Visit D, enqueue H



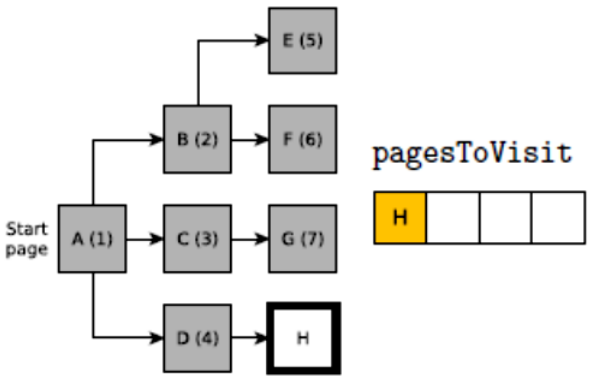
Step 6: Visit E



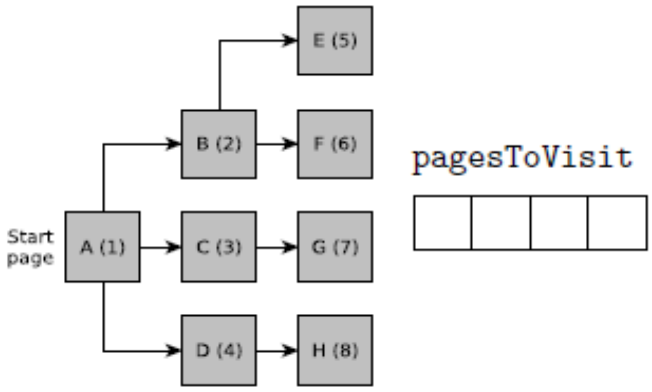
Step 7: Visit F



Step 8: Visit G



Step 9: Visit H



Notice that this algorithm explores each level in the web graph before moving on to the next. This is the idea behind the *breath-first search* (BFS) – we explore the breadth of the graph before exploring its depth.

3.2 Breadth-First Search Pseudocode

The following algorithm can be used in our BreadthCrawlerclass to implement the same breadth-first search that we have just seen:

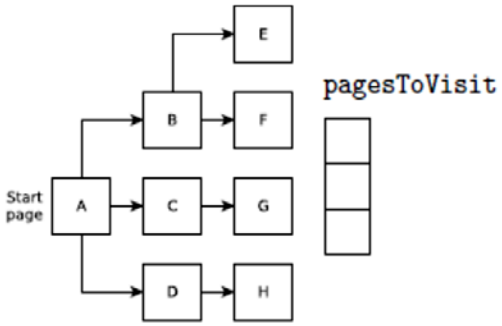
```
startPage ← a new Pageobject, initialized with the starting address
Enqueue startPage

while queue is not empty do
   page ← next Pageobject from the queue
   Download page
   Add the address of page to the list of pages already visited

   if page contains the search keyword then
      Print a message stating so
   end if

   if depth of page < max. page depth then
      while there are more links on page AND we haven't enqueued more links FROM THIS PAGE than the max
      links limit do
         link ← next link from page
         if we haven't already visited link AND it isn't already in the queue then
            Add it to the queue
         end if
      end while
   end if
end while
```

Hint: You should trace this algorithm by hand using the graph above to ensure that you understand what it is doing.

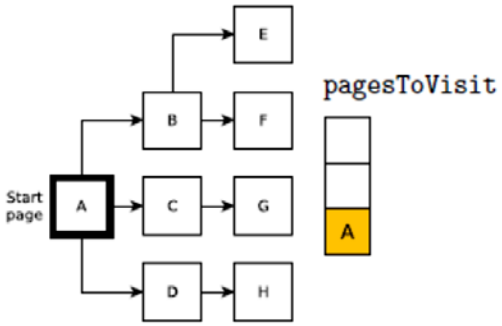
3.3 Understanding Depth-First Search

We have seen how a queue can be used to implement a breadth-first search. What if, instead, we were to use a stack? Let's follow the same algorithm we did earlier, except this time *pagesToVisit* will be a stack. Once again, in the graphs below, a shaded node represents a page that has been visited. A heavily outlined node represents a page that is currently on the *pagesToVisit* stack. Finally, a number inside a node represents the order in which it was visited.

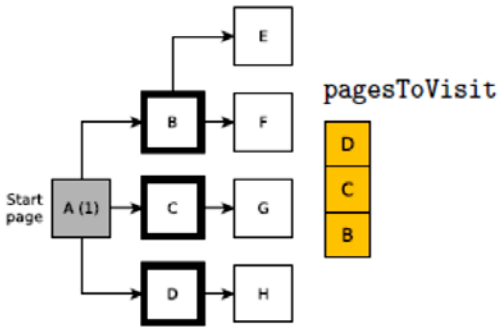
Initial: No pages visited, empty queue



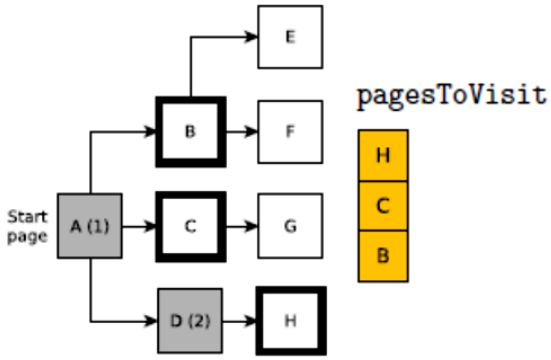
Step 1: Push start page (A). No pages visited



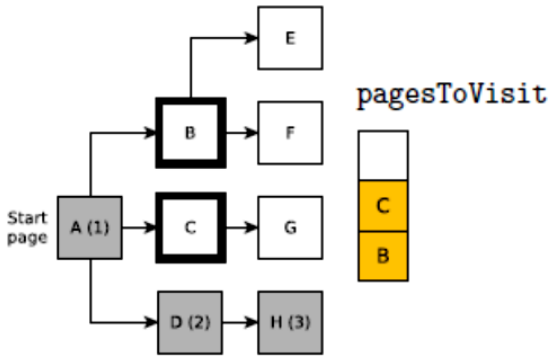
Step 2: Visit A, push B, C, D



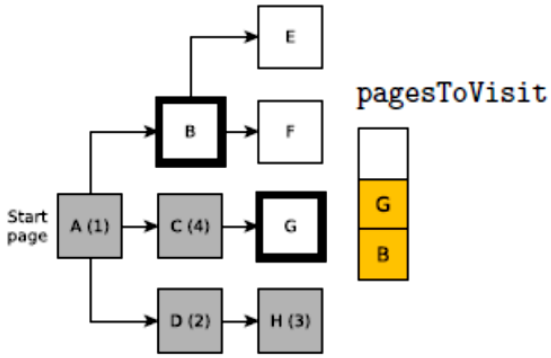
Step 3: Visit D, push H



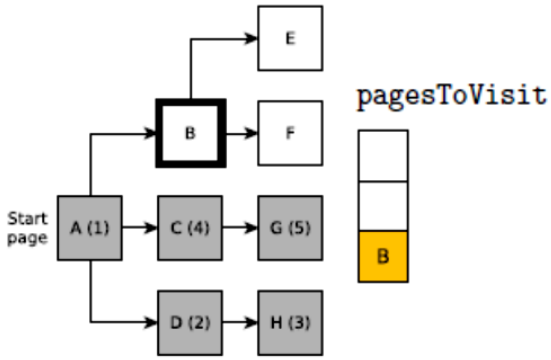
Step 4: Visit H



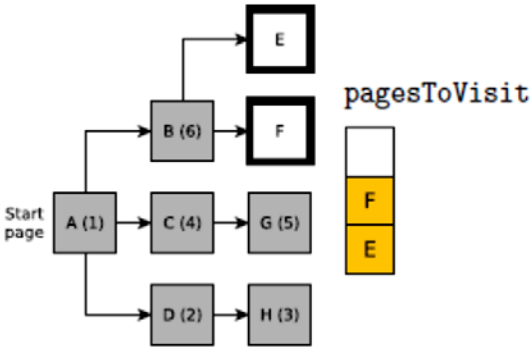
Step 5: Visit C, enqueue G



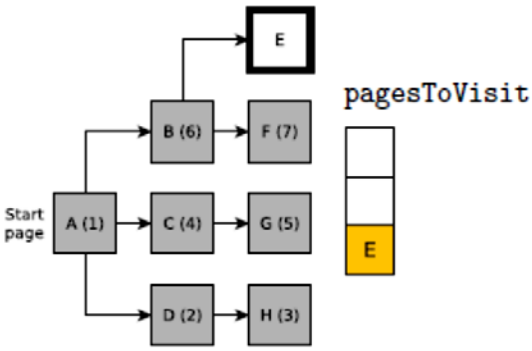
Step 6: Visit G



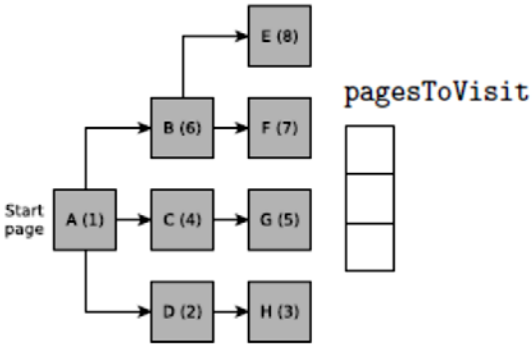
Step 7: Visit B, push E, F



Step 8: Visit F



Step 9: Visit E



Notice now that this algorithm explores as far down each level in the web graph as it can go before backtracking and exploring a new branch. This is the idea behind the depth-first search (DFS) – we explore the depth of the graph before exploring its breadth.

Note: The breadth-first and depth-first search algorithms share an interesting characteristic. If we have written an algorithm to perform a BFS using a queue, we can easily transform this algorithm to a DFS algorithm. After all, in the DFS example above, we performed the same actions as we did in the BFS example, but we used a queue instead of a stack. Indeed, this is all that needs to be changed to transform a BFS algorithm to a DFS algorithm. You simply copy the algorithm, change the queue to a stack, and change all queue methods (e.g. enqueue, dequeue) to stack methods (e.g. push, pop)!

4 Provided Classes

You have been provided with the following classes that you will need to use to build your crawlers.

4.1 Page

This class models a web page that a crawler either will visit, or has already visited. The following methods are provided for your use. While the class has other methods, you are only permitted to use the following methods of the Pageclass:

- `public Page(String address)`
  Constructor that initializes a new Page with a specified address
- `public String getAddress()`
  Returns the page address
- `public int getDepth()`
  Returns the integer search depth of the page
- `public Iterator<Page> linkedPageIterator()`
  Returns an iterator over all the links on a given page. This method is only useful after a page has been visited
- `public boolean containsText(String text)`
  Returns a Boolean value indicating if the page contains the specified text. This method is only useful after a page has been visited

4.2 WebHelper

The WebHelperclass takes care of visiting web pages for you, downloading their contents, and parsing the text and links that they contain. The class contains one method that you will use:

- `public static void downloadPage(Page page) throws Exception`
  Visits the specified page and downloads its contents. The Pageobject passed to this message is populated with the web page text and a list of links that the page contains so that the `linkedPageIterator()` and `containsText(String text)` methods can be used on it. If an error occurs while downloading or parsing the page, an exception will be thrown

Note: This method is static and therefore must be called on the *WebHelper* class itself, rather than creating an instance of the class. For example, to call this method, you would simply use the following code:

```
WebHelper.downloadPage(nextPage);
```

4.3 PageParser

You will not use the PageParserclass, but will need to download it as it is required by WebHelper.

Part I
Implementing the Back-End

5 Crawler

Before we get to our BreadthCrawler and DepthCrawler classes, we first need to think about what methods and attributes might be common to any kind of crawler. After all, we do not wish to duplicate code, so any common code shared between these classes should be *abstracted* into a superclass. The following lists a few items that any type of crawler – no matter how it is implemented – might need to know:

- What pages have I visited? (I don't want to waste time and visit the same page twice)
- What keyword am I searching for?
- At which address will I begin searching?
- What is the max. depth to which I will search?
- What is the max. number of links on a page that I should visit before I move on to another page?

In this section you will complete the Crawler class, which has been provided to you partially completed. Crawler is an abstract class that contains instance variables and methods that help to answer the questions above. It will serve as the superclass for the *BreadthCrawler* and *DepthCrawler* classes that we will implement.

5.1 Functionality Provided to You

The class currently stores a list of page addresses that the crawler has already visited. The following methods have been implemented for you:

- `public Crawler(String keyword)`
  Constructor that initializes an empty list of visited page addresses

- `public boolean hasVisitedLink(String address)`
  Returns a Boolean value indicating whether or not the crawler has visited the specified address

- `public void addVisitedLink(String address)`
  Adds a new page address to the list of visited addresses

5.2 Instance Variables

You must modify Crawler to contain the following instance variables. Do not remove or modify any existing instance variables.

- A String representing the keyword for which the crawler will search
- An integer representing the max. depth to which the crawler will search
- An integer representing the max. number of links to be crawled on each page

5.3 Methods to Implement

Your Crawler class should implement the following methods. Do not remove or modify any existing methods,

except where specified below.

- Modify the constructor to store the keyword passed as a parameter in an instance variable. Do not change any other code in the constructor
- Provide public accessor and mutator methods for all instance variables you added in section 5.2
- Provide a method search:
  - Abstract method – we will implement this method in the BreadthCrawler and DepthCrawler classes (why is this abstract? if its abstract, should you make the whole class abstract?)
  - Takes a Stringrepresenting the address at which to start searching
  - Returns nothing
  - Since errors may occur when we try to search (e.g. when we try to download pages that do not exist), this method may throw an Exception
- Provide a method printMatch that takes a Page as an argument. The method should **print** a message indicating that the search keyword has been found on the page, and should print its address (e.g. *"Keyword found on http://www.csd.uwo.ca"*). printMatch should only be accessible within Crawler and its subclasses
- Provide a method printVisiting that takes a Page as an argument. The method should **print** a message indicating that the crawler is about to visit the page, and should print the page address (e.g. *"About to visit http://www.csd.uwo.ca"*). printVisitingshould only be accessible within Crawlerand its subclasses. This method is intended to be called by a crawler class just before it downloads its next page

6 BreadthCrawler

You will implement the BreadthCrawler class to perform a breadth-first search. BreadthCrawler should extend the Crawlerclass. You must implement the following:

- A public constructor that takes a Stringkeyword for which to search
- An instance variable of type CircularArrayQueue to store the pages that have yet to be visited. This variable should be accessible only within the BreadthCrawler class. Your queue should store Page objects and not Strings
- A public method search that takes a String page address, and performs a breadth-first search, starting at that page. Note that this method overrides the search method in Crawler. You can use the algorithm given in section 3.2 to help you implement this method.

Other requirements:

- You must print a message *before* visiting each page indicating that the page is about to be visited. Be sure to include the page address
- You must print a message when a match has been found. Again, include the page address
- You must handle any exceptions thrown when downloading or parsing the page. You should report that an error occurred, indicate the address of the page on which the error occurred, and then gracefully move on to the next page in the queue
- You must minimize code duplication by making use of superclass methods where possible
- You can add any helper methods that you feel are helpful or reduce duplication of code

Hint: To get access to each link on a page, you should use an Iterator. See the description of the Page class and look for a method that you can use to get an Iteratorover the links in the page. An example will be posted on the FAQ page showing how one can use an iterator.

7 DepthCrawler

You will implement the DepthCrawlerclass to perform a depth-first search. Once again, this class must ex- tend the Crawler class. The DepthCrawler has the same requirements as the BreadthCrawler class, except that it will perform a depth-first search instead of a breadth-first search.

As noted earlier, to perform a depth-first search, you can use the same algorithm as given for the breadth-first search in section 3.2, but the sole difference is that rather than using a queue, you will need to use a stack in the depth-first search version of the algorithm.

This means that your DepthCrawlerclass will need a stack instance variable instead of a queue. You must use the LinkedStackclass for your stack, and once again, your stack should store Pageobjects and not strings.

8 Testing Your Code

In Part 2 of this assignment, you will integrate your back-end code with a graphical user interface that will be provided. Before doing so, you should make sure:

- Your code compiles
- Your algorithms are implemented properly and produce the correct results
- Depth and links/page limits are respected by your program
- Errors are handled gracefully and your program does not crash

One suggestion to help you test your code would be to write a TestCrawlerclass containing a mainmethod. This method would then instantiate a BreadthCrawler or DepthCrawler, and performs a test search on a set of web pages. You are not required to hand in a test class for this assignment, but it would be wise to write one to ensure that your code is correct.

A set of pages are linked from the assignment web page. You can use these pages to test your crawlers. The set of pages provided for testing is rather small, so it would be in your best interest to click the links on these pages and draw a graph (similar to the graphs shown earlier) of the structure of the test pages, along with the text contained on each page. Using such a graph will help you to ensure that your crawlers implement the breadth- and depth-first search algorithms correctly, as you can trace the algorithms by hand.

Part II
Integrating Your Crawlers with a GUI

In this part, you will write several classes and make small changes to your existing classes to integrate your crawlers with the graphical user interface (GUI) that has been provided to you.

9 CrawlResult

A *CrawlResult* object will store the result of crawling one page. It will contain information such as the Page object that was crawled, whether or not the crawl succeeded (e.g. was an Exceptionthrown when attempting to download the page?), and whether or not the page matched the search term. Each time you crawl a page, you will provide the GUI with a *CrawlResult* object describing the result of the crawl, allowing the GUI to provide information on the progress of the crawl.

9.1 Instance Variables

- The Pagecrawled
- A Stringto store an error message if the crawl failed
- A flag indicating if the Pagematched the search term
- An integer indicating the order in which the page was crawled

9.2 Methods to Implement

- A constructor that takes a Page, integer sequence number, and a boolean indicating if the passed Page matched the search term. All instance variables should be initialized by this constructor
- A constructor that takes a Page, integer sequence number, and String error message. All instance variables should be initialized by this constructor
- Accessors:
  - `public Page getPage()`
  - `public String getErrorMessage()`
  - `public int getSequence()`
  - `public boolean isMatch()`
  - `public boolean crawlSuccess()`(Return true if the crawl completed successfully)

10 Crawler

In this section, we will modify Crawler slightly to add instance variables and methods that we require for our integration.

10.1 Instance Variables

- A *CrawlResultList*to store the results of our crawl
- An integer indicating the sequence number of the last page crawled
- A boolean indicating whether or not we should stop crawling (if the user clicks **Stop** in the GUI). This should be accessible in Crawlerand all subclasses
- A boolean indicating if we are currently crawling. This should be accessible in Crawlerand all subclasses

10.2 Methods to Implement

- Modify the constructor to:
  - Take a parameter of type CrawlResultListin addition to its existing parameter
  - Store the value of this new parameter in an instance variable
  - Set the last page sequence number to **0**
  - Set an instance variable to indicate that we are not currently crawling
  - Set an instance variable to indicate that we should not stop crawling
- Remove the throws Exceptiondeclaration from the searchmethod – your searchmethod should now be working and not throwing any exceptions
- **addCrawledPage**
  - Takes a Pageand a boolean indicating if the page matched the search term.
  - Creates a new CrawlResultobject with parameters Page and the boolean from above, as well as an integer sequence number of order, and adds it to the CrawlResultListobject you stored as an instance variable
  - Returns nothing
  - Should be accessible only in Crawlerand all subclasses
- **addFailedPage**
  - Same requirements/restrictions as addCrawledPage but instead takes a Pageand a Stringerror message
  - When creating a CrawlResultobject in this method, the appropriate constructor should be used (e.g. one that accepts an error message)
- **isCrawling**
  Accessor indicating if the crawler is currently crawling. Should be accessible by all classes
- **setCrawling**
  Mutator that sets whether or not the crawler is currently crawling. Should be accessible only in Crawler and all subclasses
- **stop**
  Sets the instance variable that indicates whether or not we should stop crawling. Takes no parameters and returns nothing. Should be accessible by all classes
- **crawlingNextPage**
  Increments the last page sequence number. Takes no parameters and returns nothing. Should be accessible only in Crawlerand all subclasses

11 BreadthCrawler and DepthCrawler

Modify each class as follows:

- Change the constructor to suit the new signature of Crawler
- Before searching begins, inform the superclass that we are now crawling
- Your search loop should now continue while the queue/stack is not empty **AND** it has not been requested that we stop crawling
- Before downloading a page, increment the last page sequence number

- After downloading a page (if it succeeds), use the appropriate superclass method to store the result of the crawl
- If a crawl attempt fails, use the appropriate superclass method to store the error message. You should use the error message from the Exceptionthrown
- After the loop, just before the method returns, inform the superclass that we are done crawling

12 MyCrawlerWindow

Complete the class MyCrawlerWindow. For methods that ask you to interact with GUI components, please note that all components (e.g. text fields, buttons, etc.) are exposed as protected variables in the CrawlerWindow class, so they are inherited by MyCrawlerWindow.

When asked to work with a component, you will need to examine the instance variables of the CrawlerWindow class to find the appropriate variable name that references the component.

12.1 GUI Components

You will need to write a few methods in this class that will work with GUI components, such as text boxes, buttons, radio buttons, and spinners. The following displays the components you will be working with on the GUI and their Java class names:

Reference	Class	Methods You Will Use
A	JLabel	setEnabled
B	JTextField	setEnabled, setText
C	JButton	setEnabled, setText
D	JRadioButton	setEnabled, setSelected, isSelected
E	JSpinner	setEnabled, setValue

12.2 Methods to Implement:

- `private void setComponentsEnabled(boolean enabled)`

Enable or disable all labels, text fields, buttons (except the Search/Stop button), radio buttons, and spinners. Use the .setEnabled(boolean) method for this. For instance, if called with true as a parameter, your GUI would appear as:

If, instead, we called this method with false, your GUI would appear as:

- `private void startCrawler()`
  - Clear the results list (the superclass has a clearResultsmethod to do this)
  - Set the label on the search button to Stopusing .setText(String text)
  - Disable the GUI components
  - If the BFSoption is selected, create a new BreadthCrawlerand store it in the superclass instance variable crawler. You should pass to the constructor the result list obtained from the superclass getResultListmethod, as well as the text of the search term field. To check if a radio button is selected use .isSelected()and to get text out of a text field use .getText()
  - If the DFSoption is selected, do the same as above, except using a DepthCrawler
  - Set the maximum depth to which the crawler will go from the max. depth spinner. To get the value from the spinner use .getValue()which returns an Object
  - Set the maximum links per page that the crawler will visit from the max. links/page spinner
  - Call the superclass method startCrawlerThread, passing it the URL entered by the user
- `public void stopCrawler()`
  - If the superclass instance variable crawler is not null, and the crawler is currently crawling, then stop the crawler (see the Crawlerclass) and invoke the superclass method stopCrawlerThread
  - Set the search button text to Search
  - Renable all GUI components
- `private void searchButtonActionPerformed(java.awt.event.ActionEvent evt)`
  - Called when the search button is clicked
  - If a crawl is currently in progress (use the superclass isCrawlingmethod), then stop the crawler
  - If a crawl is not in progress, check if the user entered both a URL and search term
    - ✯ If so, start the crawler
    - ✯ If not, display an appropriate error message using the superclass showErrorMessagemethod
- `private void resetButtonActionPerformed(java.awt.event.ActionEvent evt)`
  Called when the reset button is clicked. This should:
  - Clear the seed URL and search term fields. You can clear text in a text field using .setText(null)
  - Set the selected crawl type to breadth-first search. You can set a radio button using .setSelected(true)
  - Set the max depth and max. Links per page and fields to 3. You can set the value of a spinner using setValue()which takes an Object

13 Bonus Problems

If you are enjoying this assignment and finish early, feel free to attempt the bonus problems. It is very important that you only attempt these after you have completed the rest of the assignment. You will get no marks for bonus problems if the basic assignment does not work.

Two bonus problems are available, each involving the implementation of more useful/advanced searches.

NOTE: Your assignment must be working to qualify for bonus marks. If your assignment does not work, you will not receive any bonus marks.

13.1 Problem 1 - Levenshtein Distance (up to 10%)

Users often make mistakes when performing searches. For example, one might accidentally search for *fool* when intending to search for *tool*. In cases like these, it might be helpful to return pages that contain words similar to the keywords for which they searched. Your tasks:

- Read the Wikipedia article (http://en.wikipedia.org/wiki/Levenshtein_distance) on computing the Levenshtein distance between two strings
- Create a LevenshteinPage class that extends Page. Add the following method to the class:

  ```
  private int getLevenshteinDistance(String s1, String s2)
  ```

  This should compute the Levenshtein distance between the two strings. To help you implement this method, you can use the pseudocode provided in the Wikipedia article.

- Override the *containsText* method to return true if the page contains the specified text, or contains a word whose Levenshtein distance differs from the specified text by no more than a constant integer value d. You should determine the best value for d by testing the *LevenshteinCrawler* that you will write, and determining the value that returns the most accurate results. For instance, we might want our crawler to return pages containing *tool* if we accidentally searched for *fool*. On the other hand, if we searched for *oranges*, then returning pages containing *televisions* would likely be quite annoying to users
- Create a new crawler class *LevenshteinCrawler* that extends *Crawler*. The class can use a breadth- or depth-first search. Everything will be the same as in your existing crawler classes, with the exception that it will use the *LevenshteinPage* class instead of the *Page* class
- Add a new option to the GUI that allows the user to perform a Levenshtein search

13.2 Problem 2 - Soundex (up to 10%)

Suppose we operate a call center selling widgets over the phone. Customers call in to buy widgets, and customer service representatives start by asking them for their last names. A service representative then searches for a customer's order history using his/her last name to see if the customer has an existing file on record.

Sometimes, our customer service representatives mishear the names that the customers provide and they search for the wrong names. In this case, it would be helpful to return customer records for names that sound like the name for which we are searching. For example, if we search for Robert, we may wish to return pages that also contain a similar-sounding name such as Rupert. For this, we can use a phonetic algorithm such as the Soundex algorithm. Your task:

- Read the Wikipedia article (http://en.wikipedia.org/wiki/Soundex) on Soundex and figure out how to compute a Soundex value for a string
- Create a SoundexPage extending Page. Add the following method to the class:

  ```
  private String getSoundex(String word)
  ```

This method should return the Soundex value for the specified word. Next, override the *containsText* method to return true if the page contains the specified text, or contains a word that has the same Soundex value as the specified text

- Create a *SoundexCrawler* that extends *Crawler*. Again, your crawler can use either a breadth- or depth-first search and should make use of SoundexPage objects instead of Page objects
- Add a new option to the GUI that allows the user to perform a Soundex search

14 FAQs

14.1 How do I iterate over the links in a page?

The following code shows an example of iterating over the links of a Page. A Page objects stores its links in a list data structure (which we will explore in class next week). An iteratore allows us to access each of the links in the list.

```
import java.util.Iterator;

// Get an iterator of the page's links
Iterator<Page> it = page.linkedPageIterator();

while (it.hasNext()) {

        // Get the next Page object from the iterator and store it in linkedPage
        Page linkedPage = it.next();

        // Do something with the new linkedPage that we just obtained
}
```

14.2 In Part 2, I notice that when I try to get the value from one of the spinners, it returns an Object, but I need an integer. How do I handle this?

You can cast its value to an Integer object, as shown here:

```
int val = (Integer)spinner.getValue();
```

Note that you need to cast to Integer, not int, but thanks to Java's auto-unboxing feature, the code above works fine. You can assign an Integer to a primitive int variable and it will convert the Integer object to a primitive for you. Yay :)

15 Submission

You should **only submit .java** files. Please do not submit your .class files as the TAs cannot run these and your assignment will not be able to be marked. Only submit the four **.java** files listed above.

It is your responsibility to ensure you have handed in the correct files. If you fail to hand in the correct files and your assignment cannot be marked you could receive a mark of zero.

Good Luck.