Lab2: Building a Web Crawler

CS251

Lab2: Building a Web Crawler

Please see the FAQ before asking questions to the TAs

You will be using the HashTableTemplate.h that you implemented it in last lab. Ask for help to your lab instructor if you did not successfully finish your HashTableTemplate.h implementation.
You will be using data.cs.purdue.edu to do your implementation. If you have problems using your HashTableTemplate.h then fix it to be used in data.

Goal:

In this lab you will build a web crawler program that will automatically explore the web to retrieve data that will be useful for future searches.

Description:

This project will be individual. A web-crawler is a program that explores the web to build a database that will be used by a search engine. Starting from an initial URL (or a group of initial URL's), the program will follow the hyperlinks of these initial documents to reach other documents and so on. The program retrieves the words in the document and links them to the URL of the document for later search. The web-crawler will scan a URL only once by keeping track of the URL's it visits.

Build the Initial gethttp

Copy the initial files to your cs251 directory

```
  cd
  mkdir -p cs251 _
  tar -xvf /homes/cs251/Summer2017/lab2-webcrawler-shttp/lab2-src.tar
  cd lab2-src
  make
```

Then run gethttp.

```
  gethttp https://www.purdue.edu
```

The gethttp program gets the document from the URL passed as parameter and prints it on the screen. You may try using other URLs.

Look at the code in gethttp.cpp. This program uses the function ***fetchHTML()*** that receives a URL as a parameter and returns a pointer to the document content.

IMPORTANT: Using GIT

We have added to the Makefile some commands to commit to GIT your current changes every time you run "make". This will commit your changes to the local .git/ repository in your lab2-src. It is important that you don't remove these commands. You should let "make" run them since the submission should include a complete history of your changes.

Parsing the HTML Document

You can think of an HTML document as text mixed with tags of the form <...>. The first step for the web crawler when it fetches a document is to strip out the tags and leave the text. Then the HTML text is analyzed for keywords. The second step is to extract the <HREF ...> tags to get what links the document is pointing to.

We are providing in lab2-src a simple HTML parser SimpleHTMLParser.h and SimpleHTMLParser.cpp that implements a simple finite state automaton that parses your document. It provides some basic functionality but you will need to improve it to support other details of HTML.

The HTML parser is implemented as an object of type ***SimpleHTMLParser***. This object has a method ***parse(char * buffer, int n)*** that parses the document passed in ***buffer*** and that has ***n*** bytes. While parsing, parse() calls two call back methods. The method onContentFound(char c) is called when a character in the text (and not in the tags) is parsed. The method onAnchorFound(char * url) is called when an <A ref="url" ...> anchor is found. The string ***url*** contains the url in the anchor. The way you use this parser is by subclassing the ***SimpleHTMLParser*** and then overiding the ***onContentFound***() and ***onAnchorFound***() methods with code to handle the information collected.

For example, when you type gethttp with the -t option:

```
gethttp -t https://www.purdue.edu
```

it will print the text content of that HTML page without the tags. This is done in gethttp.cpp by defining a class HTMLParserPrintContent that inherits from SimpleHTMLParser. The method onContentFound overides that method from the parent and is defined as follows that prints to stdout each character of the document content.

```
void
HTMLParserPrintContent::onContentFound(char c) {
      printf("%c",c);
}
```

The -a option prints the anchor tags that are in the document.

```
gethttp -a https://www.purdue.edu
```

This is done in gethttp by defining a class HTMLParserPrintHrefs that inherits from SimpleHTMLParser. The method
onAnchorFound overides that method from the parent and is defined as follows that prints to stdout the URLs of the links in that document.

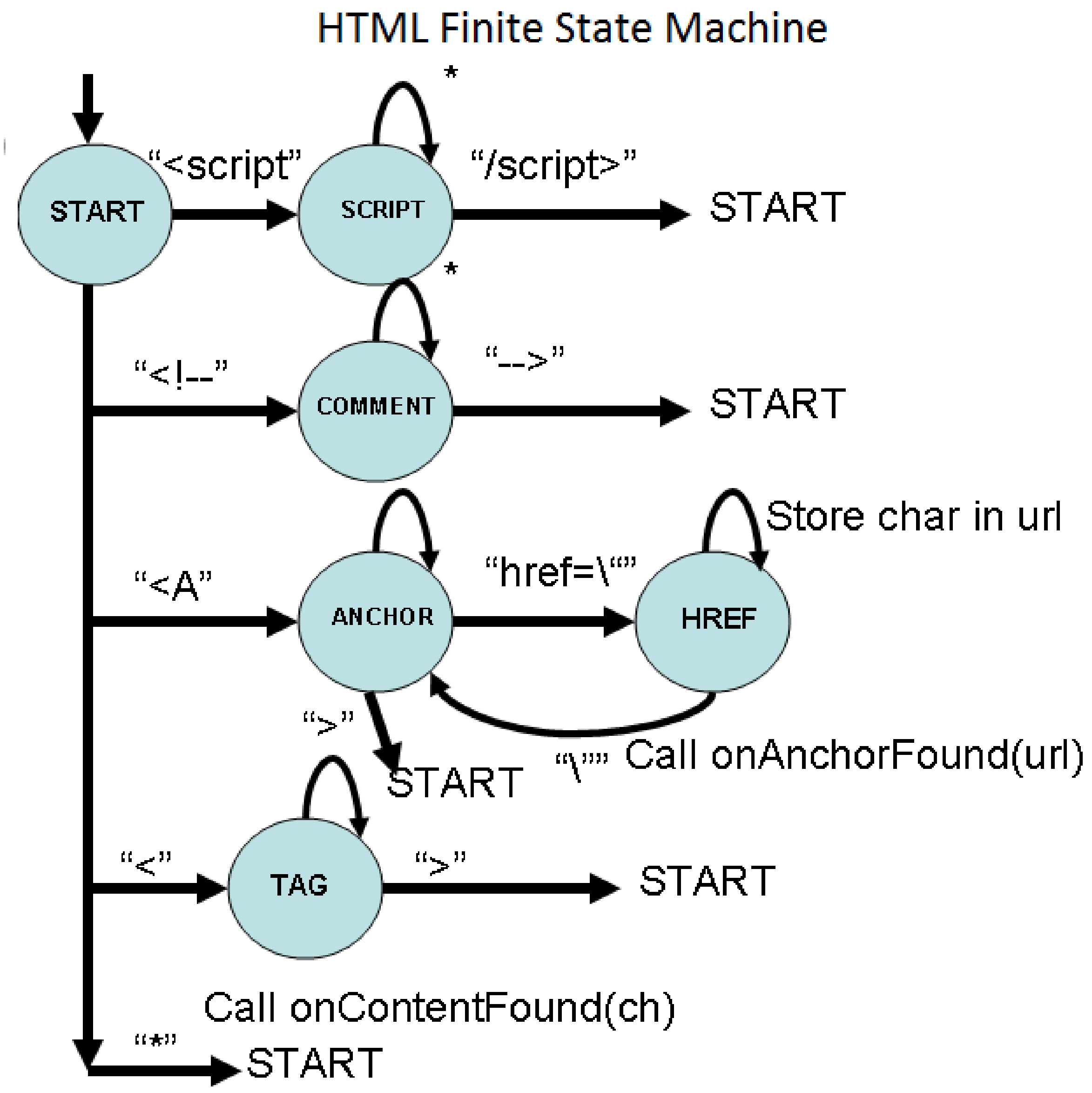
You will also subclass the SimpleHTMLParser class to implement your web crawler. You may also need to add functionality to the SimpleHTMLParser to parse or ignore some of the content.

How the SimpleHTML Parser works

The SimpleHTML parser works by implementing a simple Finite State Machine (FSM). A FSM has "states" represented as circles and "transitions" represented with arrows. At any point in time the FSM is in one of these states.The machine starts at the START state. Each state has transition arrows coming from them. To transition from the current state to another, a string matching the label in the transition has to be read.

The following shows the FSM used to parse HTML. In the SimpleHTMLParser.cpp the FSM is implemented as an infinite loop and each state is implemented as a "case" inside a switch statement. The variable ***state*** stores the current state. When in a state, if the current position in the buffer input matches a string, then it will change the state. For example, in the START state, if the coming string is "<A" then the state will change to ANCHOR. This is represented in the following code:

```
else if (match(&b,"<A ")) {
      state = ANCHOR;
}
```



You will need to add functionality to the FSM as needed.

Step 4. Building a web crawler program.

The webcrawler program will have the syntax:

```
    webcrawl [-u <maxurls>] url-list
```

Where **maxurls** is the maximum number of URLs that will be traversed. By default it is 1000. **url-list** is the list of starting URL's that will be traversed.

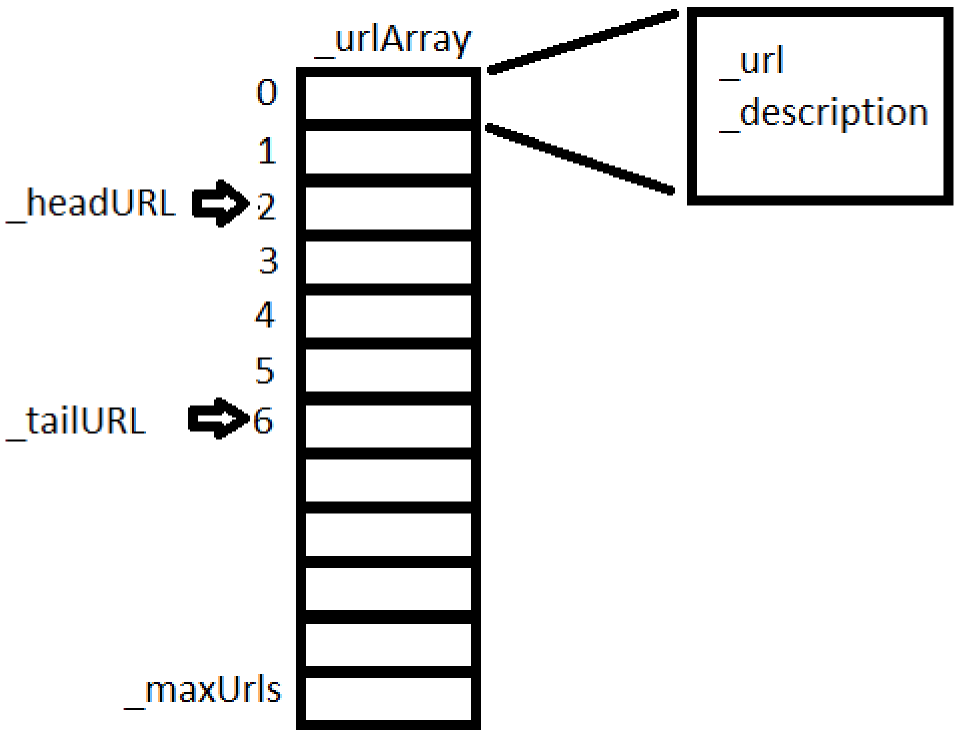
You will use the following data structures that are included in the class WebCrawler.

URL Array (_urlArray)

The URL array consists of an array of entries of type **URLRecord,** where each **URLRecord,** will contain two fields: **_url** is a pointer to the URL and **_description** is a pointer to a description of the URL. The description of the URL will consist of some meaningful description of the document. It can be a) at most 500 characters of the document without tags (<..>), escape characters (&x;), and new lines, b) The meta tag in the document if any, c) Headers H1, H2 etc. Part of the grade of the project will also cover how useful the descriptions are. The size of the **_urlArray** will be **_maxUrls** taken from the command line.

```
struct URLRecord {
   char * _url;         // URL of the document.
   char * _description;  // First 500 characters (at most) of the document without tags
                         // or some other more meaningful description
};

class WebCrawler {
  // The url array stores all the URLs that have been scanned
  int _maxUrls;              // Max size of the URL array
  URLRecord * _urlArray;   // Array of URLs
  int _headURL;              // Points to the next URL to scan by the web crawler
  int _tailURL;             // Next position unused in the urlArray
...
```



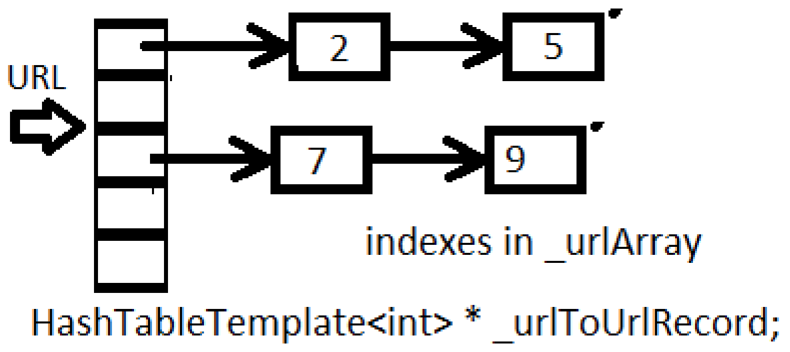
The variable **_headURL** is the index in the array of the next URL to be requested to be scanned for words. **_tailURL** is the next entry in the array that is empty. More will be explained about these variables when the crawler algorithm is covered.

URL to URL Record Table (_urlToUrlRecord)

The URL to URL Record Table maps a URL string to the corresponding index in the _urlArray.

You will use the HashTable template to implement this data structure. If you did not finish your HashTable template ask for extra help to your lab instructor to finish it. You don't need all the functionality it provides. Just implement what you need that is insertion and remove into the hash table. ***_urlToUrlRecord*** also is stored in the WebCrawler class.

```
HashTableTemplate<int> * _urlToUrlRecord;          //  maps a URL to its index in the URLArray
```

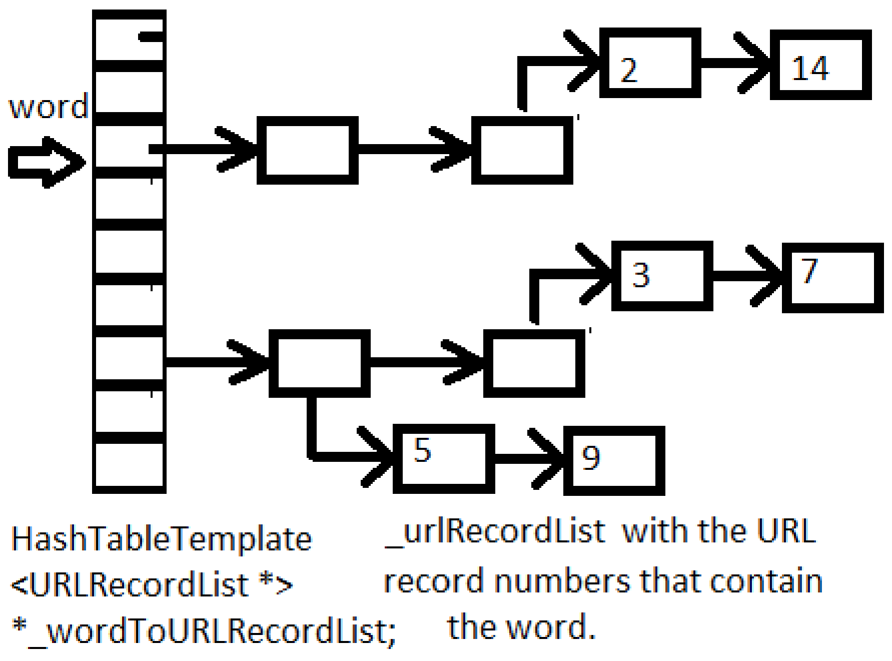


Word to URLRecord List Table (_wordToURLRecordList)

This table maps a word to the list of URLs of the documents that contain this word. You will also use a HashTable template to implement this table.

```
// Used to implement a list of URL indexes of the url array.
// There is one list of URLs for each word
struct URLRecordList {
  int _urlRecordIndex;       // Index into the URL Array
  URLRecordList * _next;   // Pointer to the next URL Record in the list
};

HashTableTemplate<URLRecordList *> *_wordToURLRecordList; // maps a word to a list of URLs
```


word
2
14
3
7
5
9
HashTableTemplate <URLRecordList *> *_wordToURLRecordList;
_urlRecordList with the URL record numbers that contain the word.

Web Crawling Procedure

The web crawler will visit the URLs in a breadth first search manner, that is, it will visit the URLs in the order they were inserted in the URL array. To do this you will use the variables _headURL and _tailURL. _headURL is the index in the URL Array of the next URL to visit. _tailURL is the index in the URL array of the next empty entry in the _urlArray that will be used to store a new URL that will be searched later.

During the web-crawling you follow these steps:

1. First insert the initial URLs from the url-list passed as parameter to "webcrawl" in the URL Array. You will do this in the WebCrawler constructor

```
WebCrawler::WebCrawler(int maxUrls, int nInitialURls,  const char ** initialURLs)
{
  // Allocate space for _urlArray
  // insert the initialURls
  // Update _maxUrls, _headURL and _tailURL
}
```

2. Update the WebCrawler::crawl method with the following

```
Webcrawler::crawl()
{
  while (_headURL <_tailURL) {
      Fetch the next URL in _headURL

      Increment _headURL
      If the document is not text/html
           continue;
      Get the first 500 characters (at most) of the document without tags. Add this
    description to theURL record for this URL.

      Find all the hyperlinks of this document and add them to the
    _urlArray and _urlToUrlRecord if they are not already in the
    _urlToUrlRecord. Only insert up to _maxURL entries.

   For each word in the document without tags, add the index of this URL to
     a URLRecordList in the _wordToURLRecordList table if the URL is not already there.
 }//while
}
```

webcrawl Output

webcrawl will create two files:

url.txt - It will give the list of URLs in the form:
<url-index><space><url><\n>
<description without \n's><\n>
<\n>
. . .
One entry for each URL.

Example:
1 http://www.purdue.edu
Purdue Main Page

2 http://www.purdue.edu/admisions
Admisions Homepage

. . . .
word.txt - It will give the list of words with the list of url's for each word.
<word><space><list of URL indexes separated by spaces><\n>
....

Example:
CS 3 4 6 7 89
Purdue 4 6 3 5 7
....
One entry for each word

Notes

You are allowed to do any modifications to the data structures, classes, and add any methods you need as long as you maintain the format of the output. Also you may modify the file openhttp.cpp if you need to.

Turnin

When you turnin lab2 also include in lab2-src a README file with the following information:

1. The name and login
2. The list of features in the handout that work
3. The list of features in the handout that do not work
4. Any extra features.

There is going to be a late penalty of 5/100 per day after the deadline up to a week.
Follow these instructions to turnin lab2:

1. Make sure that your webcrawl command is built by typing "make". Make sure it builds and runs in data.
2. Type "turnin -c cs251 -p lab2 lab2-src"
3. Type "turnin -c cs251 -p lab2 -v" to make sure you have submitted the right files

The deadline of this project is 11:59pm Monday July 10th, 2017.