Data and Algorithms

FRONT END:

Fortunately, are system doesn’t use many complex data structures. Since we store the data in an SQL database. We can retrieve the data from the database and store it in simple container objects which we then use to generate the display for the user. Also, since the front end is primarily responsible for interacting with the user the complexity of the algorithms is low. Most of the functions involve storing and retrieving text-based data which is a common and well-known type of system.

Back End:

The backend also does not use any complex algorithms, but it uses lists and queues. As discussed earlier, a challenge site may have one index with links to all of its challenges or it may have multiple pages of indexes with a limited number of links on each page. Both of these scenarios may either list the challenges with the most recent first, or in chronological order with the oldest first and the newest last. The data types used ensure that uploads occur in chronological order. This important because because the most recent uploaded link is the comparator for finding duplicates.

|  |
| --- |
| Case 1: Single Index, Newest first. |
| Datatype: Stack |
| The system will parse the links until either an already uploaded link is encountered or until there are no more links. Each encountered link will be added to a stack. The links will then be uploaded to the database with the oldest being added first. |

|  |
| --- |
| Case 2: Single Index, Oldest first. |
| Datatype: Queue |
| The system will parse the links until either an already uploaded link is encountered or until there are no more links. Each encountered link will be added to a stack. If the duplicate is discovered, the previous queue is deleted a new one formed from challenges listed after the duplicate. The links will then be uploaded to the database with the oldest being added first. |

|  |
| --- |
| Case 3: Multi Index, Newest first. |
| Datatype: Stack |
| This is very similar to case 1. However, if the end of the page is reached without finding a duplicate, the next index page must also be checked. This setup maintains chronological order over multiple pages without any special data manipulation. |

|  |
| --- |
| Case 4: Multi Index, Oldest first. |
| Datatype: Stack → Queue |
| This is the trickiest of the cases. An example encounter is:  11 → 12→ 13→ 14→ 15  6 → 7→ 8→ 9→ 10  1 → 2→ 3→ 4→ 5  Our system would encounter link 6 immediately after link 15 and link 1 after link 10. There are several options for solving this. The simplest would be to start with problem 1 every time and iterate through all index pages. However this makes lots of unecessary calls to the challenge site.  A linked list could also be used. Every new page, the current node would be set to the head. New nodes would be added after the current node and the new node would then become the current node. In many languages this would be the ideal solution, but lists in the SPL are implemented as modified arrays. Adding nodes to the front of a list causes all following nodes to shift making it very computationally expensive and losing all benefits to using a list.  A third option is to pull all nodes into a page stack then move the page stack to a master stack. This effectively reverses the order of the links on a page while keeping the pages in order. The final master stack. The master queue will have all non-duplicate links in order from oldest to newest and ready for uploading. |

Front-End Tests

**Display Tests**

Test case 1: Display View

Function Tested: displayView()

Success/fail: an html page containing the details of a clicked challenge

Expected Result: A detailed page contain all relevant challenge information is displayed

Expected Failure: No page is displayed

Test case 2: Display Challenge

Function Tested: displayChallenge()

Success/fail: a html page contains a list of challenges meeting the criteria is generated

Expected Success: a list is correctly displayed of relevant challenges

Expected Failure: No new page is displayed

Test Case 3: Display Error

Function Tested: DisplayError(error)

Success/fail: a detailed error message is displayed on screen

Expected Success: an error message that contains info about the error is displayed

Expected Failure: a generic error message of “something went wrong” is displayed

**Search Tests**

Test case 4: build search

Function Tested: buildSearch()

Success/fail: a successful and correct searchParameters has been built,

Expected Success: An accurate searchParameters is built from user generated parameters

Expected Fail: A default searchParameters is built from default values

Test case 5: send query

Function Tested: sendQuery()

Success/fail: a successful query has been passed to DBconection

Expected Success: a query is sent down through DBconnections to the database

Expected Failure: a query is unable to be sent, and the user is notified that it failed to search

**Format Data Tests**

Test Case 6: Format list

Function Tested: formatList(listData)

Success/fail: correct data is provided, and a list is built

Expected Success: a successful list is built and sent to the lister

Expected Failure: a list was unable to be built and an error is sent to lister

Test Case 7

Function Tested: formatChallenge(challengeData)

Success/fail: correct data is provided and a challenge has been built

Expected Success: an accurate challenge object is sent to viewer

Expected Failure: a error is sent to challenge viewer

Test Case 8

Function Tested: buildPage()

Success/fail: a page has been correctly built that can be displayed

Expected Success: a page has been built and passed into the interface to display to user

Expected Failure: an error message is displayed to user

Back-End Tests

The back end has numerous functions that are important but relatively straight forward when using the appropriate built-in PHP libraries. Retrieving HTML from an external website, parsing out useful information from the HTML, getting information from the database, and uploading to the database. These can be expounded on later, but for now they will be skipped so that we can deal with the heart of the back-end tests: incomplete database states.

Test Case 1: Inititial Search

This case occurs when an external site has never been collected from before. A success means that links to every single challenge is collected and uploaded. A failure occurs if less than all challenges are uploaded.

This will be the status immediately after an upload has occurred and will remain that way until the external site posts new challenges. Success occurs if no uploads are made. Failure occurs if any uploads are made.

Test Case 3: New links on newest index

This will be the most common state that our system encounters. Our system has already uploaded some challenges, but new challenges have been posted. A success occurs when all new challenges are uploaded. Failure occurs if not all new challenges are uploaded or if old challenges are re-uploaded.

Test Case 4: Newest links on older index

This is a modified version of Case 1. If a multi-index site has not been visited for a long time, the number of uncollected challenges may exceed the number of challenges/index on the external site. Our system must collect all new challenges on every index page. A success occurs when all new challenges are uploaded. Failure occurs if not all new challenges are uploaded or if old challenges are re-uploaded.