Restoration Architecture

Set:

- Purpose: used to identify original rows in the pgm from rows inserted by the corruption.
- The set will contain character pointers (c-style strings) that contain non-numeric characters inserted into whitespace for each row in the pgm.
- Logic of the algorithm:
    - For every new line read by readaline(), restoration will extract the string of non-numeric characters from it.
    - Next, check if the set contains this string
    - If yes, we have found the sequence of inserted non-numeric characters in original rows. Going forward, we only need to check against this string to distinguish original rows.
    - If no (AND we have yet to find the recurring string), add string to the set.
- Functions to use: Set_new, Set_put, Set_free, Set_member; void* pointing to non-numeric strings

Sequence:

- Stores the lines of the processed data. It contains char* components, which are the numerical data that were added to the set.
- In case of duplicates, where there are multiple lines with the same non-numerical data, only the first occurrence of the numerical digit is stored.
- Functions to use: Seq_addhi, Seq_get, Seq_length, Seq_free

Table:

- Purpose: Stores and manages the non-numerical data and their associated numerical values.
- Each unique non-numerical char* acts as the key, while the corresponding numerical row acts as the value.
- Functions to use: Table_new, Table_put, Table_get, Table_map

Implementation Plan

1. Create the .c file for the restoration program. Write a main function that spits out the ubiquitous “Hello World!” greeting. Compile and run. (10 min)
2. Create the .c file that will hold our readaline implementation. Move the “Hello World!” greeting from the main function in restoration to the `readaline()` function. (3 min)
3. Call `readaline()` from `main()`. Compile and run restoration. (4 min)
4. Add parameters to `main()` in restoration to read in arguments from standard input; also add error-checking if-statements. Compile & run. (4 min)
5. In restoration, write and call the function `openOrDie()` that takes in a filename and attempts to open it. (6 min total)
    a. In the function, add if-statement that exits program with error if file cannot be opened (3 min)
6. Add a line that closes the opened file in restoration. (1 min)
7. Call `readaline()` and provide its parameters to it. (2 min)
8. Test that this open - read - close cycle is working and reporting errors correctly. (3 min)
9. Write `readaline()` (30 min total)
    a. Parameters: FILE *inputfd, char **datapp
    b. Check that the file pointer and **datapp are not NULL.
    c. If first character in file is EOF, then set *datapp to NULL and return 0.
    d. Create a char that stores the current character in the file and int for the number of bits in a line.
    e. Loop through file as long as the current character is not a new line or EOF.
        i. Read a single character from the file using fgetc and store that in the current character char.
        ii. If the current char is a new line, increment the number of bits.
    f. Using *datapp, allocate memory for each line based on the number of bits.
    g. Check that memory allocation succeeded.
    h. Move the file pointer to the start of the line.
    i. Loop through the file.
        i. Read 1 char at a time and store it in datapp.
    j. Return number of bits.
10. Test `readaline()` (30 min total)
    a. Test that seekg() and malloc is working correctly with a .txt file of numbers. (10 min)
    b. Test that the function is correctly reporting errors when the file is invalid. (10 min)
    c. Test that the function is correctly modifying char** by printing the char* to screen (10 min)

11. In restoration, write out the skeleton of the restoration function, including all helper functions and when to call them. (15 min)
12. Write `getNonNum()` (20 min total)
    a. This helper function will take the current line and extract all non-numeric characters.
13. Write `AddtoSet()` (20 min total)
    a. Parameters: a non-numeric string
    b. Returns: void
    c. Create a char* `ogPattern` in main that will first be set to NULL; when the recurring pattern of non-numeric is found, set to the sequence (1 min)
    d. Create a set in main (2 min)
    e. If-statement will be checking if this char* is NULL (1 min)
        i. If yes: call `AddtoSet()`
    f. Checks if the non-numeric string passed is contained in the set
        i. If yes: We have found the pattern of non-numeric chars for original rows! Update `ogPattern`, add row to the sequence.
        ii. If no: Add current pattern to the set, add row to the table.
14. Write the `findLine()` function (15 min total)
    a. Create in `main()` a Table (3 min)
    b. Parameters: char** of the non-numeric pattern
    c. Returns a char** of the original row stored in the table
    d. Uses the pattern received (`ogPattern`) as the key, retrieve the single original row in the table
    e. Back in main, write code that adds this single row to the start of the sequence. (5 min)
15. Write the `isOriginal()` function (15 min total)
    a. We know what the non-numeric pattern for original rows are. For every new row read in, check if it is an original row.
    b. Parameters: char** to `ogPattern`; char** to the new line read
    c. Returns a boolean value; true if the new line is an original row, false otherwise.
16. Put a while loop around the entire thing above. We will continue reading lines from the file provided until we have reached the end of line. (2 min)
17. Write error-checking function before printing out the restored pgm (10 min)
    a. Check things such as width and height
18. Write the `printPnm()` function (15 min total)
    a. We have reached EOF in the file, print the raw pnm to screen
    b. Parameters: int width, int height, sequence
    c. Returns void
    d. Prints the header and iterates through sequence data to print pixel data.

19. Test the implementation (5 hours)

Testing Plan

Readaline: Test readaline with .txt files with inputs containing short text (a singular letter or multiple letters and numberss) and long text (100s of letters and numbers new lines).

- Example files:
  - 1
  - 1a2b3c4d5f6g
  - 1001 ASCII characters, should print an error message saying readaline: input line too long
  - 345x
    x123
    678x
  - 123xyz
- Edge cases: empty file, file with now new line at the end, file with no new lines at all, file with more than 1000 characters, file with only newlines and white space, file with null characters.

Restoration:

- The overall logic of our testing strategy for the restoration program is: we will first test each individual function independently by calling the function in main and provide unique inputs to each function (intended inputs and invalid inputs), checking that the printed output is correct.
  - For `getNonNum()`:
    - This function will always be receiving a valid line of input, assuming readaline() works as intended. Hence, we only need to test it on expected inputs. (e.g., 19A68%207^vb3o should produce A%^vbo),
  - For `AddtoSet()`:
    - Check that the function indeed adds the provided string into the set.
    - Test by verifying back in main after the function call that if we now try to add the same string, the set says it is already contained.
  - For `findLine()` :
    - Manually add into the table unique key-value pairs of non-numeric strings and rows read in from the file.
    - Call `findLine()` with one of the non-numeric strings and verify that it returns the correct corresponding row.
  - For `isOriginal()`:
    - Check that the function returns the correct boolean value.

        - Test by using assertion and comparing the new line with the ogPattern and making sure that it returns true if the new line matches the pattern or vice versa.
    - For `printPnm()`:
        - Manually create sequences that contain a half-restored example of a raw pgm, provide this sequence to the function.
        - Verify that the function correctly adds headers to the file when printing.
        - Also verify by calling pnmrdr on the output, check if the interface can read the output as a raw pgm.
- After testing each function independently, now we will debug the entire restoration program. Since functions' inner logic and code is working as intended, our focus in this second round of debugging is on logic between functions (i.e., connecting between functions, providing parameters to functions, receiving function returns.) We will mainly do this by inserting in print statements that reports where in the code we are and narrowing where errors may be.