# FilesofPix Design Document

**Jerry Huang (jhuang25) and Yuvit Batra (ybatra01)**

**Restoration Architecture:**

As each row is read, we create two temporary Char* variables: one to store the non-digit sequence and another to store the digit sequence. We then use the Table_get function, which returns NULL if the key is not found in the table, to check whether the non-digit sequence already exists in the table.

- If it does exist, we increase the frequency count of the non-digit sequence and append the corresponding digit sequence to the list. Both values (the frequency and the list) are packaged together using a struct.
- If it doesn't exist, we add the non-digit sequence to the table, set its frequency to 1, and add the digit sequence to the list.

So for the purpose of restoration we use two data structures, namely a list and a Table. The functions used to extract and add data to these data structures are as follows:

**For Table**

- To create table: Table_new
- To get values from table: Table_get(this function returns null if key isn’t in table)
- To free Table: Table_free
- To put values in Table: Table_put

**For List**

- To create list: List_list
- To put values in list: List_append
- To free list: List_free
- To get values from list: List_pop

Further details of how we plan to restore the images are mentioned in the implementation plan. Please refer to the image below for a visual representation of our data structure:

# Table

| Key | Value |
|---|---|
| `non-digit Sequence' → stored as Char* | Struct → has two variable: • frequency: Number of times non-digit sequence occurs (int*) • List: [contains digit rows as sequence]. Eg ['10 2 3', '30 10 5', '2 7 9'] → Stored as Char* |

So given the below Sequence:

y 10 bc 9 d
a 10 b6 cd
a 10bc 7d

| Key | Value |
|---|---|
| `ybcd' | {1, ['10 9']} → struct |
| `abcd' → non digit Sequence | {2, ['10 6', '10 7']} → list [Hanson's Data Structure]; 2 → Frequency → Stored as int |

**Implementation and Testing Plan**

1. **Create the .c file for the restoration program. Time**: 10 minutes
    a. **Task**: Write a main() function that prints the ubiquitous “Hello World!” greeting.
2. **Create the .c file to hold the readaline() implementation. Time**: 7 minutes
    a. **Task**: Move the “Hello World!” greeting from the main() function in restoration to your readaline() function, and call readaline() from main().
3. **Extend the restoration program to open and close the intended file, and call readaline() with real arguments. Time**: 20 minutes
    a. **Tests**:
        i. **Test**: Run with an incorrect number of arguments.
        ii. **Test**: Try opening different file types.
        iii. **Test**: Handle the case when the file can’t be opened.
        iv. **Test**: Use an empty file.
        v. **Test**: Use a file that contains one character.
        vi. **Test**: Use a file that contains multiple characters.
        vii. **Test**: Pass something other than a file as input.
        **viii.** **Test:** Run valgrind to check if the file is being closed properly
4. **Build the readaline() function.**
    a. **Task**: Create a check to see if the supplied arguments are valid (inputfd and datapp are not NULL).
        i. If yes, continue below.
        ii. If no, terminate with a Checked Runtime Error.
        iii. **Tests**:
            1. **Test**: Where inputfd is NULL, but datapp isn’t.
            2. **Test**: Where datapp is NULL, but inputfd isn’t.
            3. **Test**: Where both datapp and inputfd are NULL.
    b. **Create a check to see if there is a line to be read. Time**: 5 minutes
        i. **Task**: Use feof() to check.
        ii. If yes, set *datapp to NULL and return 0.
        iii. If no, continue below.
        iv. **Test**: Use an empty file to verify this.
    c. **Use ALLOC to allocate space for the line. Time**: 10 minutes
        i. If this fails, terminate with a Checked Runtime Error.
    d. **Create a counter variable to store the number of bytes in the line. Time**: 3 minute
    e. **Iterate through the characters in the line until '\n' or EOF, updating the counter variable with each iteration. Time**: 1.5 hour
        i. **Task**: Use the fgetc() function to iterate, ensuring the '\n' character is included in the total number of bytes.

        ii. **Task:** Make sure REALLOC when we need to increase buffersize
        iii. **Tests**:
            1. **Test**: With an empty line.
            2. **Test**: Return value when the line contains one digit character.
            3. **Test**: Return value when the line contains one non-digit character.
            4. **Test**: Return value when the line contains two characters (one digit and one non-digit).
            5. **Test**: Return value when the line contains 500 characters (all digits).
            6. **Test**: Return value when the line contains 500 characters (all non-digits).
            7. **Test**: Return value when the line contains 500 characters (a mix of digits and non-digits).
            8. **Test**: Return value when the line contains 2000 characters (all digits).
            9. **Test**: Return value when the line contains 2000 characters (all non-digits).
            10. **Test**: Return value when the line contains 2000 characters (a mix of digits and non-digits).
    f. **Set *datapp to the address of the first byte. Time**: 3 minute
    g. **Clean up memory. Time**: 15 minute
        i. **Test:** run Valgrind
    h. **Return the counter variable. Time**: 3 minute
5. **Build the restoration program.**
    a. **Build command input checks. Time**: 10 minutes
        i. **Task**: Ensure the restoration program takes at most one argument (the name of the corrupted PGM). If no argument is given, read from standard input.
        ii. **Tests**:
            1. **Test**: Provide multiple arguments.
            2. **Test**: Provide an incorrect input type.
            3. **Test**: Send a file through standard input.
    b. **Initialize the *datapp variable. Time**: 3 minute
    c. **Initialize the Table and LinkedList data structures. Time**: 6 minutes
    d. **Create height and width integer variables. Time**: 3 minute
    e. **Identify fake rows and store real rows. Time**: 1.5 hours
        i. **Task**: Create a loop to allow readaline() to read through all the lines of the plain PGM file.

ii. **Tests**:
    1. **Test**: With an empty file.
    2. **Test**: On a file with no '\n' characters.
    3. **Test**: On a file with 100 lines.

f. **Within the loop, move *datapp across the array to print out each character. Time**: 15 minutes
    i. **Tests**:
        1. **Test**: With an empty array.
        2. **Test**: Using a singleton array.
        3. **Test**: Using a large-sized array (more than 1000 items stored).

g. **Modify the loop to store characters in a string. Time**: 1 hour
    i. **Task**: Instead of printing out each character, store each character into a temporary original string. At the same time, store each non-digit sequence into a temporary string.
    ii. Append the non-digit sequence to the Table if it doesn't already exist (use Table_get()). In addition, create a struct with frequency 1 and the LinkedList containing the original string.
    iii. If the non-digit sequence already exists, increment the frequency of the struct at the non-digit sequence key, and add the original string to the LinkedList.

**Tests:**

1. **Test**: Print out the Table given nothing is stored in
2. **Test**: Print out the table given only a non-digit sequence is stored
3. **Test**: Print out the Table given only one digit and one non-digit sequence is stored
4. **Test**: Print out the table given 10 different sequences
5. **Test**: Print out the table given 10 of the same sequences
6. **Test:** Print out the Table given a mix of 10 of the same and 10 different sequences.
7. **Test:** Print out the Table given sequence size is 2
8. **Test:** Print out the Table given sequence size is 100

h. **Get the number of bytes from the real row and set that value to the width variable. Time**: 1 hour
    i. **Task**: Only store the real rows from now on.
    ii. **Tests**:
        1. **Test**: Use a file that has 2 real rows and one fake row.
        2. **Test**: Use a file that has 50 real rows and 30 fake ones.

3. **Test**: Use a file with only real rows.
4. **Test**: Use a file with only fake rows.

i. **Output to P5 format. Time**: 1.5 hour
   i. **Task**: Get the frequency from the struct of the real non-digit sequence and set it to the height variable.
   ii. Print the P5 header.
   iii. Access the LinkedList in the struct at the real non-digit sequence in the table.
   iv. Access the associated original lines in the LinkedList using the list_pop() function, which gives a pointer to the start of the list.
   v. Print out the original lines from the LinkedList.
   vi. Convert the non-digit byte's of the original lines to whitespace.
   vii. Convert the original characters to P5 format using the code from the rawness lab.
   viii. **Tests**:
      1. **Test:** Use an all-white image.
      2. **Test:** Use an all-black image.
      3. **Test:** Use an image that is totally gray.
      4. **Test:** Use an image of dimensions 2x2.
      5. **Test:** Use an image of dimensions 20x20.
      6. **Test:** Use an image of dimensions 1080x1080