Nehir (nozden01) and Ariella (amann04)

<u>Restoration Architecture:</u>

1. **Identify what data structures you will need to compute restoration and what each data structure will contain.**
    a. Hanson Table: **SEE PICTURE 3**
        i. This will hold a key, value pair, in which the key represents the infused characters in a line in the corrupted file, and the value represents the digits in a line in the corrupted file
        ii. The purpose of using a hanson table is to be able to search for equivalent keys, which will indicate that the value in the hanson table holds an original digit line (bc the original lines will have the key)
        iii. The key will be represented as an Atom, another Hanson data structure
        iv. The value will be represented as a Sequence, another Hanson data structure
    b. Hanson Atom: Like said above, we will use an Atom to represent the keys in the Hanson table, which are the infused characters in a line
    c. Hanson Sequence: Like said above, we will use a Sequence to represent the value in Hanson table
        i. Another Sequence will store the lines from the original file after it is returned from the table is when there are identical keys
2. **Specify what each void * pointer will point to.**
    a. Seq_a, which is a value of the Hanson table, and holds the original digits in the file
    b. Seq_b, which is a key of the Hanson table, and holds the infused characters in a line in the file
    c. Temp, which is a sequence that holds digits temporarily before inserting them into Seq_a while parsing the line returned by readaline. We include Temp to account for multiple-digit numbers that may be in the original file **SEE PICTURE 2**
    d. Datapp*, which is a sequence that holds the line returned by the readaline function
    e. Seq_b is then stored as an Atom, which, again, are the keys to the Hanson table
    f. The Hanson table, which holds the (key, value) pair of (infused characters, digits)
    g. Seq_orig, which holds the original lines
3. **Which methods will you use to get information in and out of your structures?**
    a. **Table_put:** When you are adding to the table using Table_put, if the key already exists in the table, then it overwrites the current value in the table, replaces it with the value being put in, and then returns the original value in the table. We can then store the returned value into our data structure that holds the original file lines.
    b. **For the Atom**, it has a constant length, so you can't add or take information

   c. ***Seq_addhi(T seq, void *x)** to add new values to the end of sequences when parsing through a line

Implementation/Testing Plan:

1. Create the .c file for your restoration program. Write a main function that spits out the ubiquitous “Hello World!” greeting. Compile and run. Time: 10 minutes
2. Create the .c file that will hold your readaline implementation. Move your “Hello World!” greeting from the main function in restoration to your readaline function and call readaline from main. Compile and run this code. Time: 10 minutes
3. Build your readaline function. Time: 1 hour **SEE PICTURE 1**
4. Test readaline function. Time: 30 minutes
   a. Make a testing main where the only thing it does it run readaline with given test files that we have made as well
   b. Create 3 testing files. One with just digits, one with just non-digits, and one with a mix of both.
   c. Test that memory is allocated successfully. Time: 5 minutes
5. Extend restoration to open and close the intended file, and call readaline with real arguments. Time: 45 minutes
   a. Test that a file given has been opened or closed by trying to read in characters from a given test file. Time: 10 mins
6. Write function which parses through a line and splits the digit and non-digit bytes into sequence A and sequence B respectively. Time: 1.5 hours
   a. Test by calling only this function on lines we write. Some lines with only digits, some with only non-digits, and some with a mix of both in all different orders.
   b. Ensure that sequence A and B contain the correct information.
   c. Test that memory is allocated successfully. Time: 5 minutes
7. Write a function that turns a sequence into an atom. Time: 30 minutes
   a. Call function in testing main. Give it an arbitrary sequence and check that it returns a pointer to an atom with the same information that was stored in the sequence.
   b. Test that memory is allocated successfully. Time: 5 minutes
8. Create the while loop that goes until EOF and calls each function needed (readaline, parsing function, turning seq into atom) which have all been tested on their own. Time: 10 minutes
   a. Test while loop with nothing in it to make sure EOF is working.
9. Write functions to find the length and width of a file just based on a sequence of all the original rows. Time: 10 minutes
   a. Give each function a sequence we create and ensure that it returns the correct rows and columns that were in the sequence.
   b. Test that memory is allocated successfully. Time: 5 minutes
10. Write the restoration function. Time: 30 minutes

    a. Give the function a sequence which we write and test if it gets to the eof when called alone from the testing main.
    b. Add the functions which find length and width into the restoration function, and test that the file returned is a correct P2 file with a correct heading with the sequence that was fed in. Time: 30 minutes
    c. Test that memory is allocated successfully. Time: 5 minutes

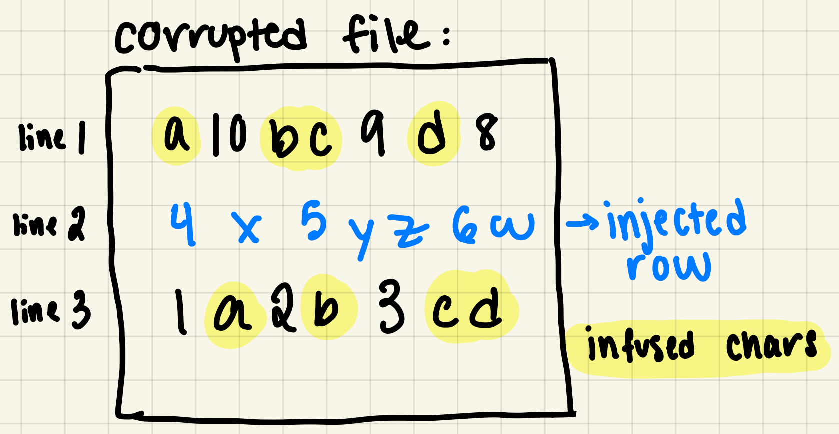
11. Put all functions together and call in main. Time: 1 hour. **SEE PICTURE 4 FOR MORE PROGRAM FLOW INFO**
    a. Give program a corrupted file and test if it returns a correct image.
    b. Test that memory is allocated and freed successfully. Time: 20 minutes

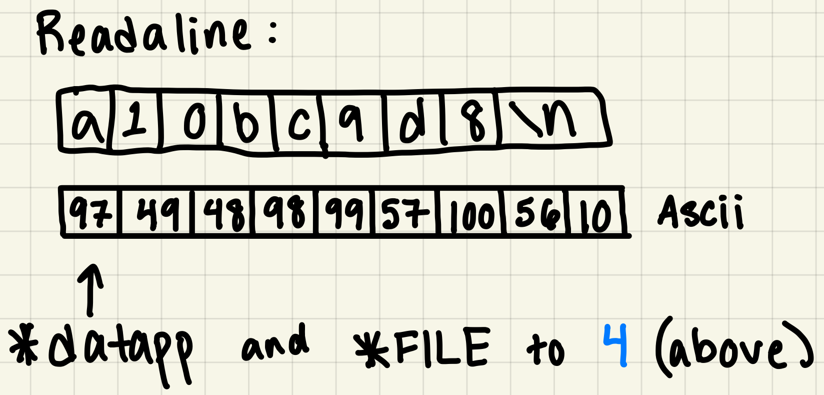
Please note that we incorporated our testing throughout our implementation plan. Below we wanted to add some more edge cases that we would account for.

Edge cases:

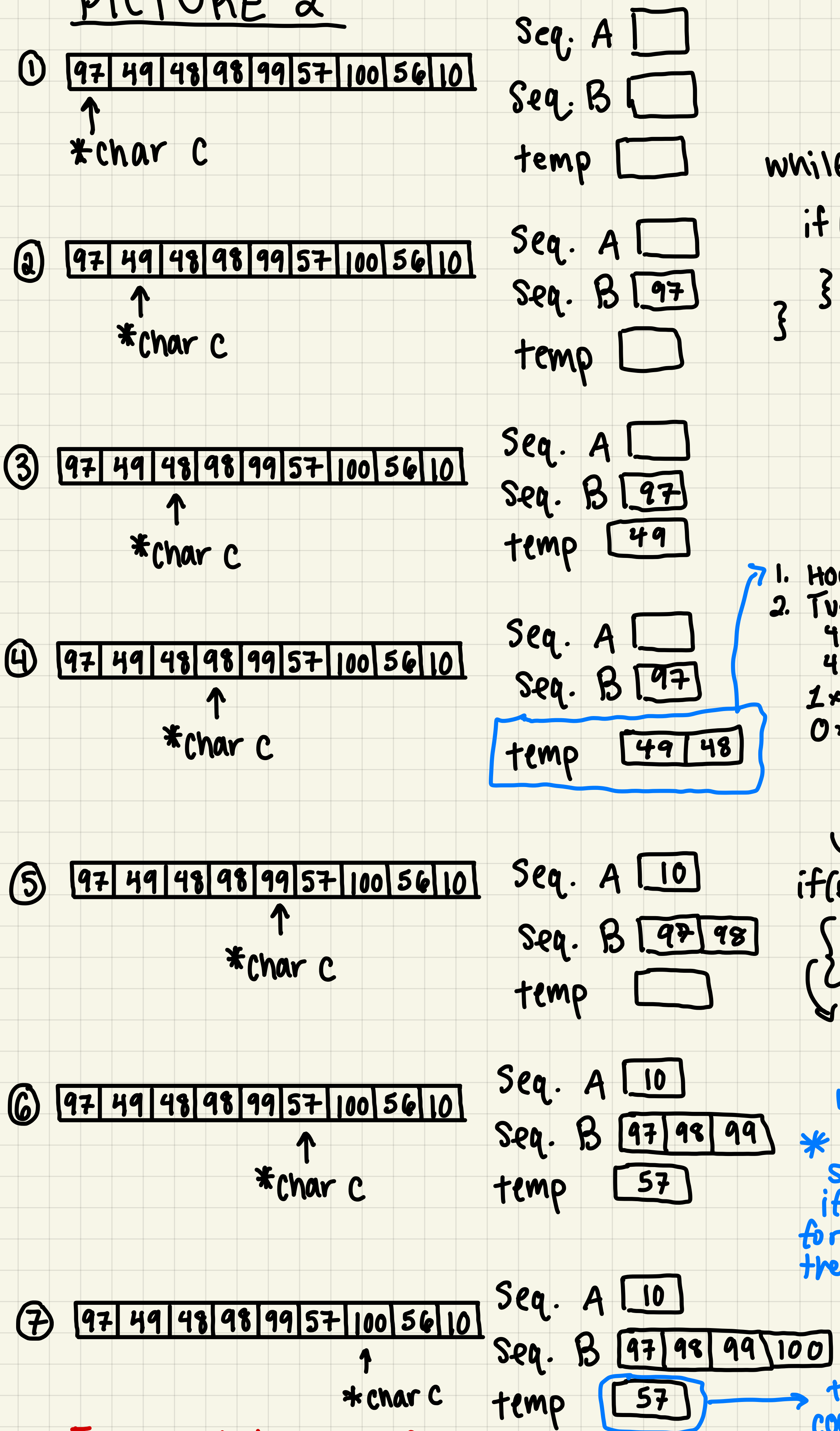
- If the last line of the corrupted file is an original line then it will be stored into the table but never returned (because there are no more lines to Table_put)
    - Store key with infused chars from original rows and search the table for that key after the whole file has been read to ensure that all key,value pairs of original lines are include in the restored file
    - Feed in file with an original line as the last line
- If the program is given a small amount of memory, how will it handle running out of memory
- Test empty files
- Test with lines that have all %%$$%@^* types of characters
- Test files with a ton of lines
- If code works and we have time, test lines with >1000 characters
- Account for file possibly ends in the middle?

PICTURE 1


corrupted file:
line 1
line 2
line 3
a 1 0 b c 9 d 8
4 x 5 y z 6 w
1 a 2 b 3 c d
→ injected row
infused chars


Readaline:
a 1 0 b c 9 d 8 \n
97 49 48 98 99 57 100 56 10
Ascii
*datapp and *FILE to 4 (above)

PICTURE 2



```
while c ≠ '\0' {
    if (48 ≤ c ≤ 57) {
    }
}
```

1. How many digits? 2
2. Turn into ints
   49 → 1
   48 → 0

$1 \times 10 = 10$
$0 \times 1 \overset{+}{=} 0$
$\boxed{10}$ * store into sequence A

$\underbrace{\qquad\qquad}$

```
if(seq_length(temp) =*= 3){
    temp @ index 0 · 10
    + temp @ index 1 · 1
    ↳ store this in sequence A
```

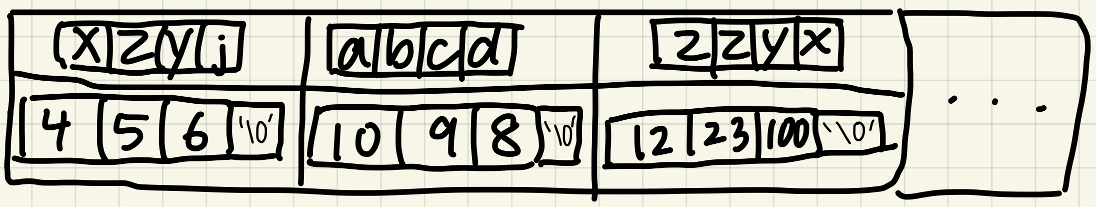
↳ some psuedocode
* have another if statement for if seq_length == 4 for case where there are 3 digits!

turn into corresponding int. (9)

ETC. until end of sequence...

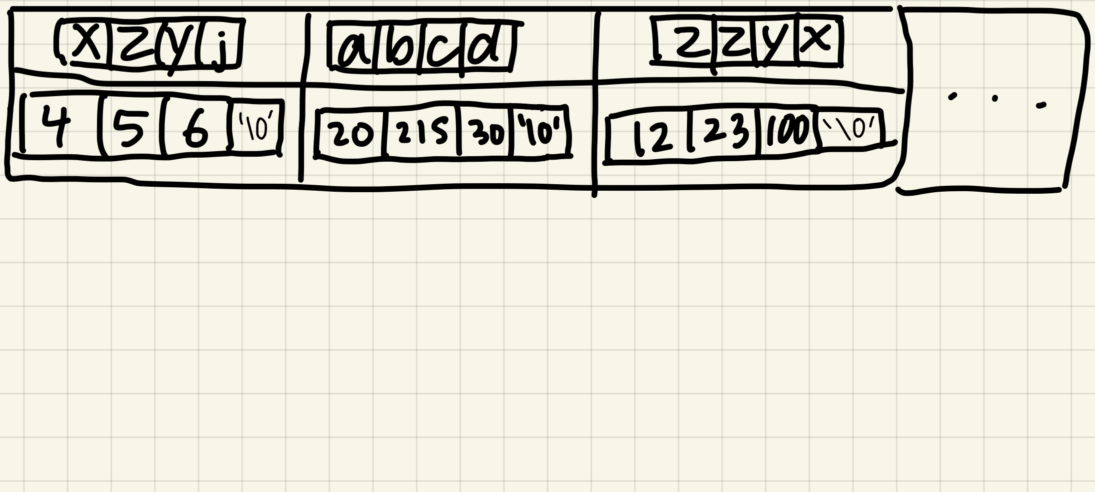
PICTURE 3

Hanson Table:
Pretend when inserted into table, w/ 1 key holding orig. file values...



When the Atom $\boxed{a}\boxed{b}\boxed{c}\boxed{d}\boxed{\text{'\textbackslash 0'}}$ w/ the val $\boxed{20}\boxed{215}\boxed{30}\boxed{\text{'\textbackslash 0'}}$ tries to be inserted into the table, there would be an indentical key already there → we found an original line in o(1) time!

Put $\boxed{10}\boxed{9}\boxed{8}\boxed{\text{'\textbackslash 0'}}$ into seq_orig and Atom w/ val $\boxed{20}\boxed{215}\boxed{30}\boxed{\text{'\textbackslash 0'}}$ into table



PICTURE 4

*represents a function

```
main {
  • openfile
  • prompt for file (if needed)
  • Get_OG(*FILE ) → returns sequence of
                     OG lines
        while (!EOF) {
            readaline
            parse
            Make Atom
            insert in table
        }
        return sequence of original lines
   • Restore (sequence of lines) → return *FILE

        • findwidth, given a sequence it
          returns an int repping the width
        • findLength, given a seq. it returns
          an int repping the length
        • create a file, and write in following:
            - "P2 \n"
            - "#filename.pgm \n"
            - "width length \n"
            - "255"
            - the given sequence
        • return fp to that file
  • call code from lab to convert P2 to
    P5
  • done :)
```