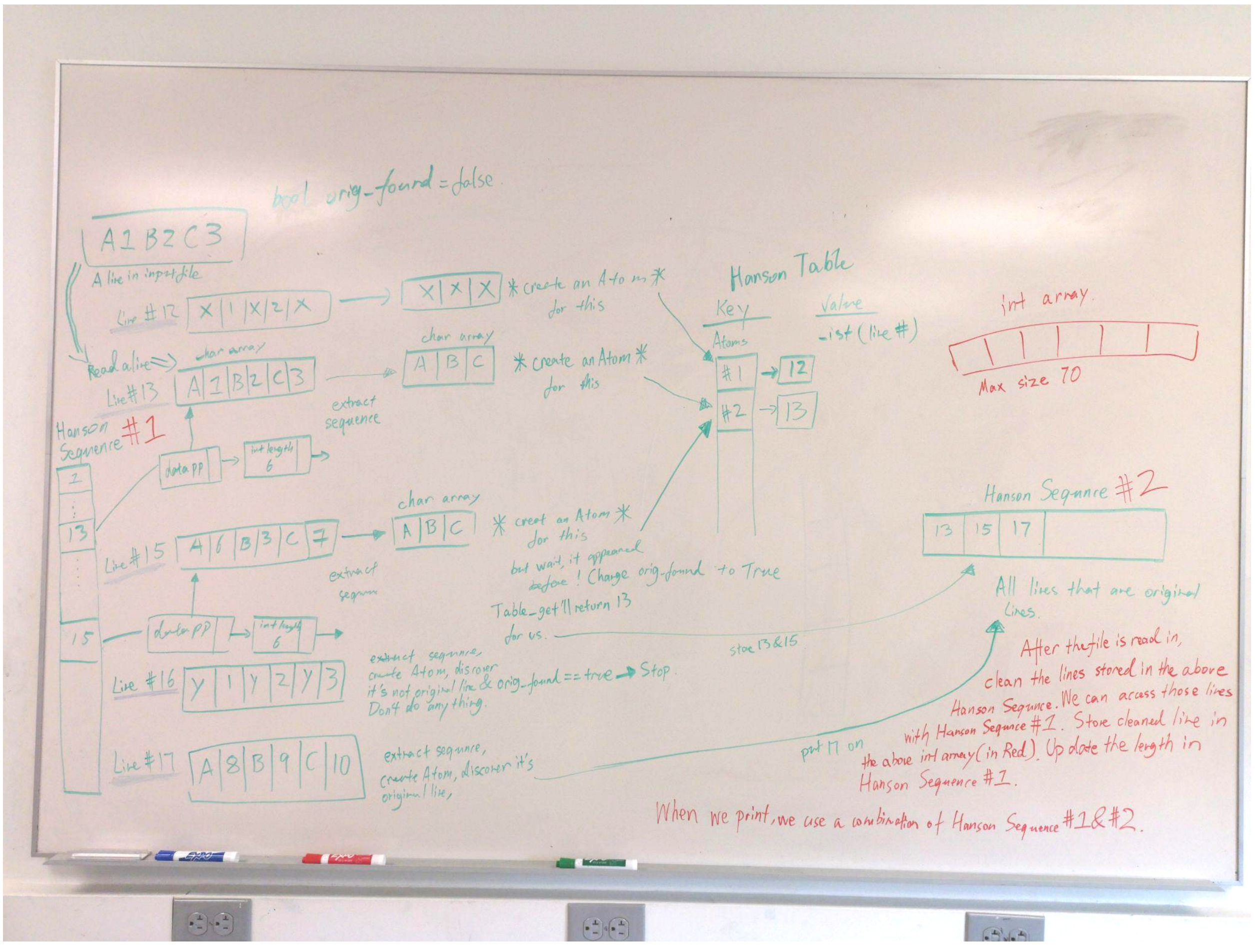
Katie Yang (zyang11) and Eli Intriligator (eintri01)

Restoration Architecture

This section must convey your high-level plan for using Hanson's data structures to both store and retrieve information in your restoration implementation.


bool orig_found = false
A1B2C3
A line in inputfile
Line #12 X|1|X|2|X
X|X|X
* create an Atom * for this
Read a line
char array
Line #13 A|1|B|2|C|3
char array A|B|C
* create an Atom * for this
extract sequence
Hanson Sequence #1
data pp
int length 6
Hanson Table
Key Atoms
Value -ist (line #)
#1 → 12
#2 → 13
int array
Max size 70
Line #15 A|6|B|3|C|7
char array A|B|C
* creat an Atom * for this
but wait, it appeared before! Change orig_found to True
Table_get'll return 13 for us.
extract sequence
data pp
int length 6
Hanson Sequence #2
13|15|17
All lines that are original lines.
store 13&15
Line #16 Y|1|Y|2|Y|3
extract sequence, create Atom, discover it's not original line & orig_found == true → Stop. Don't do any thing.
Line #17 A|8|B|9|C|10
extract sequence, create Atom, discover it's original line,
put 17 on
After the file is read in, clean the lines stored in the above Hanson Sequence. We can access those lines with Hanson Sequence #1. Store cleaned line in the above int array (in Red). Update the length in Hanson Sequence #1.
When we print, we use a combination of Hanson Sequence #1 & #2.

1. Interface
    a. struct Line which stores:
        i. void pointer which initially points to the first char in each line, then later points to the first integer in each line
        ii. unsigned int to represent length of line
    b. Hanson Sequence of type Line that stores the above data for each line of the file
    c. Hanson Table
        i. Key->Hanson Atoms
        ii. Value->int line number
    d. A Hanson Sequence which stores the line numbers of original lines

2. Implementation
    a. Initialize Hanson Table using `Table_new`
    b. Initialize an int `curr_height` for the height of this pic to zero. `curr_height`++ for each iteration of the while loop.
    c. Initialize bool `orig_found` to track whether we've found repeated/original infusion sequence yet.
    d. Until we reach end of file, loop the following:
        i. Readaline reads a line and stores it in a dynamic char array (initial size is 1000 elements).
        ii. Extract the line's sequence of non-digit characters. Store it in dynamic char arrays (initial size is 1000 elements).
        iii. Use Atom_new method to create an atom for the current sequence of non-digit characters.
        iv. Use the new atom as a key for the table and look at the corresponding value.
        v. If the original infusion sequence has *not* been found yet (`orig_found` is false)
            1. Use `Table_get` to see if the table already contains the new atom as a key. If the table does *not* contain the key, this is the first time we have seen this sequence, so add a new key-value pair to the table with `Table_put`. The key is the atom and the value is the current line number (`curr_height`) stored as an int. Also, make a Line struct for the current line and append it to the Sequence with `Seq_put`.
            2. If the table already contains a value at the key, we have found the original infusion sequence! Change `orig_found` to true, then take the int line number that was already stored in the value. Then append the current line number to the Sequence of original line numbers. After this, we won't add further lines with different non-digit sequences to the table.
        vi. If the original infusion sequence *has* been found (`orig_found` is true)
            1. If passing the infusion sequence to `Table_get` returns null, i.e. this sequence is unique, don't do anything. We know this is an infused line, so we can discard it.
            2. If passing the infusion sequence to `Table_get` returns a value (i.e. the line number of the first occurring original line), append the current line number to the Sequence of original line numbers.
    e. Function clean_line() will store all digit characters in a new int array (max size 70) and free the memory of the original array, then return the array's length and a pointer to the new array (by reference).
    f. Function encode_binary() will convert the char representations of the digits to pure binary digits, which are encoded as chars.
    g. Function print_raw() will print cleaned lines + header to standard output. The

height of the document would be the length of the Hanson Sequence. We will find the width by getting the length of an original line that is stored in the Hanson Sequence.

h. Function free_memory() will free all memory we used. It accesses each line's location in memory via the struct Line.

Implementation Plan

A detailed implementation plan should be part of every COMP40 assignment, regardless of whether or not you are required to submit it. It lays out a stepwise progression of the programming aspect of the assignment, allowing you to focus on one thing at a time instead of being overwhelmed by the assignment as a whole.

1. Create the .c file for your restoration program. Write a main function that prints out "Hello World". 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: 5 minutes**
3. Extend restoration to open and close the intended file, and call readaline. **Time: 10 minutes**
4. Add Checked Runtime Errors if readaline's either argument is NULL. **Time: 30 minutes**
5. Make the program throw an error if reading from file fails. **Time: 20 minutes**
6. Write the readaline function to read the first line. **Time: 15 minutes**
7. Allow user to input file via stdin and ensure readaline continues to accept input from command line until the user types '\n'. **Time: 15 minutes**
8. Route the output from readaline into the Hanson data structures that you have selected for your restoration implementation. **Time: 30 minutes**
9. Build a function for getting the sequences out from each line. **Time: 2 hours**
10. Build function for creating the final binary lines. **Time: 1 hour**
11. Build a function for adding the header and printing out the cleaned lines. **Time: 1 hour**

Testing Plan

We need to know the specific inputs you'll use and their expected outputs. A recommended strategy is to interleave your testing plan with your implementation plan (these do not need to be separate sections). For each step in your implementation plan, what tests will you run to confidently move forward with your implementation? Some implementation steps may require only one or two tests, while other steps will be more involved.

Step	Test	Expected output
1	Make the main function print "Hello World"	"Hello World" printed
2	Make readaline print "Hello World"	Another "Hello World" would be printed after the previous one.

3	Check if *inputfd is NULL. If no, print “ready”. If not, print “needs to check.”	“ready” is printed.
4	Supply readaline with a NULL FILE *inputfd.	Terminates with CRE
	Supply readaline with a NULL char **datapp.	Terminates with CRE
	Supply readaline with a NULL FILE *inputfd and char **datapp.	Terminates with CRE
	Call the executable with two arguments: “./restoration turing-corrupt.pgm alum-corrupt.pgm”	Terminates with CRE
5	Call the executable with an incorrect filename: “./restoration does_not_exist.pgm”	Terminates with CRE
	Test with files that you don’t have permission to read. Change permissions on a testfile with chmod so the file is unreadable, then call executable: “./restoration unreadable.pgm”	Terminates with CRE
	Call the executable correctly: “./restoration turing-corrupt.pgm”	First line of input file printed: “z0q0Fj0bCvo0^0ws0Kx0HC0M1v0.0kj0i C0^d0~0h1<j1f#h,t0Y0v"0GC0r1{e0;0q^1 $0,0#2On0ne0#1]D0g\0W1N0HiR0[u2q1` 0B2t2R0[<0yA1M0k0K1{0L0x0K0;<0'>0o v0Q0wNu0jb1rF0$e0Q0F%0zL0AI0./0+Y +0X0R0V0>e0)S0 1zw1x1Eb0\0/0T0`1I0PQ0Nu0se?0VI0K0 [TN0X0Ub1>^1,0i0)1[1a0#Lo0_[0D1j0O1j 0<s1aZ3$1v0'M1I0b0w;0<0w0tzWU”
	Call the executable with “./restoration”, i.e. no filename supplied	No output, waiting for PGM from stdin
	Call the executable with “./restoration turing-corrupt.pgm <doesn’t-exist.pgm”	Terminates with CRE
6	Create an empty file called empty.pgm and call “./restoration empty.pgm”	No output.
	Feed alum-corrupt.pgm in via standard input. Check if readaline reads in the first line correctly. Print what readaline stored.	“z0q0Fj0bCvo0^0ws0Kx0HC0M1v0.0kj0i C0^d0~0h1<j1f#h,t0Y0v"0GC0r1{e0;0q^1 $0,0#2On0ne0#1]D0g\0W1N0HiR0[u2q1` 0B2t2R0[<0yA1M0k0K1{0L0x0K0;<0'>0o v0Q0wNu0jb1rF0$e0Q0F%0zL0AI0./0+Y +0X0R0V0>e0)S0 1zw1x1Eb0\0/0T0`1I0PQ0Nu0se?0VI0K0 [TN0X0Ub1>^1,0i0)1[1a0#Lo0_[0D1j0O1j

		0<s1aZ3$1v0'M1l0b0w;0<0w0tzWU”
	Feed turing-corrupt.pgm in via standard input. Check if readaline reads in the first line. Print the read line.	“ !61 32?219?236t?+$107?156?115)??F150hT?235?? ?43? ”
	Build our own pgm with a single line of “0a@1sa$c2dr3v4h5ua!NUL6.7(8(9” (where NUL is that NUL character). Name that file NUL.pgm. Feed NUL.pgm in via standard input. Check if the size of the read line is 11 with an if statement. The if statement should print “Yes!” if true and “NOOOOOOO” if false.	“Yes!”
7	Make the program read through the entire file. Feed alum-corrupt.pgm in via standard input. Print the read things.	The contents of alum-corrupt.pgm. Diff with the original.
8	Make the program only read the first line of the input file. Feed alum-corrupt.pgm in via standard input. Print the sequence.	“zqFjbCvo^wsKxHCMv.kjiC^d~h<jf#h,tYv" GCr{e;q^$,#Onne#]Dg\WNHiR[uq`BtR[<y AMkK{LxK;<'>ovQwNujbrF$eQF%zLAI./+ Y+XRV>e)S zwxEb\/T`IPQNuse?VIK[TNXUb>^,i)[a#L o_[DjOj<saZ$v'Mlbw;<wtzWU”
	Make the program only read the first line of the input file. Feed NUL.pgm in via standard input. Print the extracted non-digit sequence.	“a@sa$cdrvhua!.((“
	Build our own pgm with the following 3 lines: “A1B2C3” <-original line “1AB2C3” <-original line “1BB2B3” <-fake line Name that file simple.pgm. Feed simple.pgm in via standard input. In restoration program, print the line numbers of the original lines.	1 2
	Change the order of the original lines so that one original line is first and the other is last. Print the line numbers of the original lines.	1 3
	Insert many different random infusions, including two that match. Print the line numbers of the original lines.	(original line numbers)