GitHub Classroom Link:

PART A [50 pts]

Difficulty: Tier-1

Learning Outcome: The student will gain experience implementing previously seen data structures in C using manual memory allocation techniques (such as Malloc).

Description:

One useful thing about data structures is that, conceptually, they are the same across all programming languages. Unfortunately, the implementation details vary somewhat from language to language.

Your first mission in this assignment is to use what you have learned about stacks and queues in previous semesters to adapt the linked list code presented in class to implement stacks and queues. You must implement the following functions:

- Stacks
  - push – Add new item to the top of the stack
  - pop – Remove an item from the top of the stack and print it
- Queues
  - enqueue – Add a new item to the back of the queue
  - dequeue – Remove an item from the front of the queue and print it

The screenshots requested in the deliverables should demonstrate the following operations being performed. After each operation, you should print the entire stack or queue.

Stacks
push(10)
push(20)
push(30)
pop()
push(40)
pop()
pop()
push(50)

Queues
enqueue(10)
enqueue (20)
enqueue (30)
dequeue()
enqueue (40)
dequeue()
dequeue()
enqueue(50)

PART B [50 pts]

Difficulty: Tier-2

Learning Outcome: The student will gain experience using bit-shifting in a real-world application.

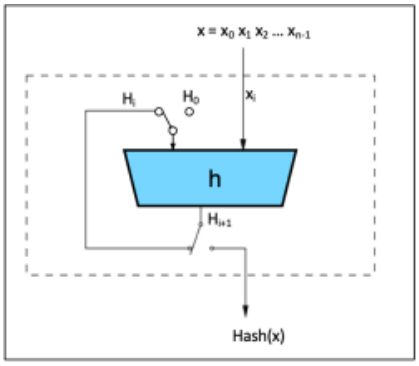
Description:

Note – Lab 2, Part B will be the basis for some elements of Lab 3.

Your second task in lab 2 is to construct a 40-bit cryptographic hash function named SHA_40 and investigate its security properties. A hash function is any function that can be used to map data of arbitrary size to fixed-size values. Hash functions play an important role in cryptography. They have been used for establishing data integrity and authentication. The value returned by a hash function is called a **digest** which can be considered as a fingerprint of the input message. A secure hash function possesses the following properties:

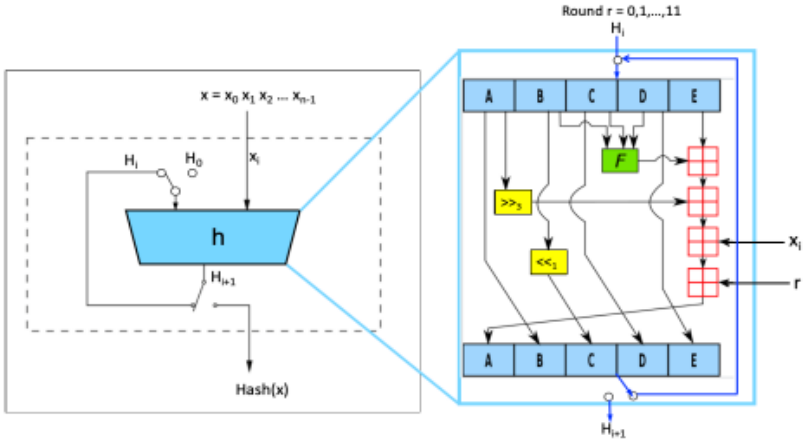
- Randomness – Two similar inputs must produce two seemingly unrelated random looking outputs.
- Onewayness – It should be easy to compute hash digest for a given input, whereas it should be hard to compute an input message that produces a given output.
- Collision resistance – It should be hard to find two different messages that result in the same hash digest.

How can hash functions process an arbitrary-length message and produce a fixed length output? In practice, this is achieved by segmenting the input into a series of blocks of equal size. These blocks are processed iteratively by a function called compression function, denoted as **h** in the diagram below. The current output of the compression function is used as input in the next iteration. This iterative design is known as Merkle–Damgard construction which is in fact used by many cryptographic hash functions including the SHA family and the one you are going to implement here.



<Merkle–Damgard construction>

In SHA_40, each block x_i of message x is a byte and hash output is 40 bits, a sequence of 5 bytes. The compression function **h** takes a byte $\mathbf{x}_i$ and the previous output $\mathbf{H}_i$ as input and produces 40 bits output $\mathbf{H}_{i+1}$, for each i = 0, 1, …,n-1,



<Internal Structure of SHA_40>

• SHA_40 can process an arbitrary-length message and produces a 40-bit output.

• **x** denotes a sequence of bytes (unsigned characters) $x_0x_1…x_{n-1}$. These bytes are processed sequentially by the **h** function. The function consists of 12 rounds, each of which performs identical operations. More precisely, i-th round takes x_i and H_i as input and generates a sequence of 5 bytes H_{i+1}. Each H_i comprises five bytes, denoted by A, B, C, D, and E.

• The hash digest, SHA_40(**x**), is then defined as the output of the last iteration of the **h** functions.
  - H_0 is the initial seed value. Let H_0 = {11, 22, 33, 44, 55}.
  - $H_{i+1} = h(H_i, x_i)$, for i = 0, 1, …, n-1
  - H_n = SHA_40(x)

• The **h** function uses bit shifting and Boolean operators to scramble the bits efficiently.

• **<<** and **>>** are bitwise shift left and shit right operators respectively.
• **F(**B, C, D) = (B & C) ^ D where **&** is a bitwise AND operator and **^** is a bit wise XOR operator.

First, open **hash.h** file and read it. Create a file named hash.c and write the following functions:

• **SHA_40**, which takes a message of arbitrary length and generated a 40-bit hash value using the sha_40 algorithm
• **digest_equal**, which indicates whether or not two digests are equal
• **main**, which tests the functions above

The screenshots requested in the deliverables should contain the following information.

- The SHA-40 hash of the string “Rob”
- The SHA-40 hash of the string “James”
- The SHA-40 hash of the string “Ahmed”
- The SHA-40 hash of the string “CSEC”
- The SHA-40 hash of your first name
- The result of digest_equal when given the SHA-40 hashes of “Rob” and “Rob”
- The result of digest_equal when given the SHA-40 hashes of “James” and “Ahmed”

GitHub Folder Structure:

- Your GitHub repository should contain the following folders:
  - PartA
    - All code for your stacks and queues should be put here
  - PartB
    - All code for your SHA-40 hash should be put here

Deliverables:

a. By 8:00 PM on 09/22/2022, you must submit a **PDF** containing screenshots of the output of your program executing under the following scenarios:
   a. The stated test cases for part A
   b. The stated test cases for part B

b. By 8:00 PM on 09/22/2022**,** you must submit a GitHub classroom link to the drop box on MyCourses
   a. Your GitHub repo must include your source code and the cipher files used in the above examples.