Data Structures & Algorithms

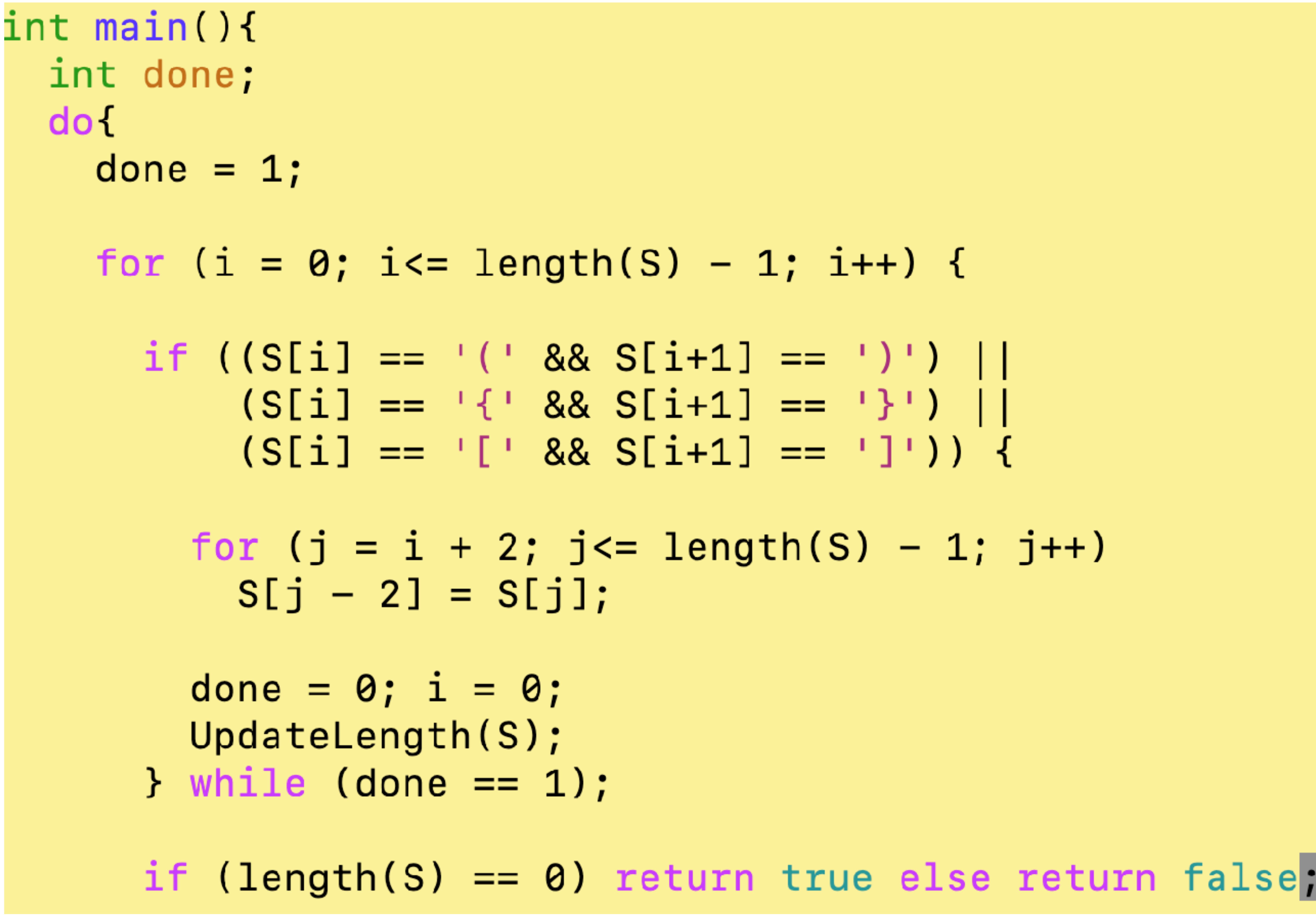
Week 3 - Queues, Hashing (HashTables, HashMaps, Dictionaries)

Subodh Sharma, Rahul Garg

{svs,rahulgarg}@iitd.ac.in

In-Class Discussion: Provide Loop Invariants

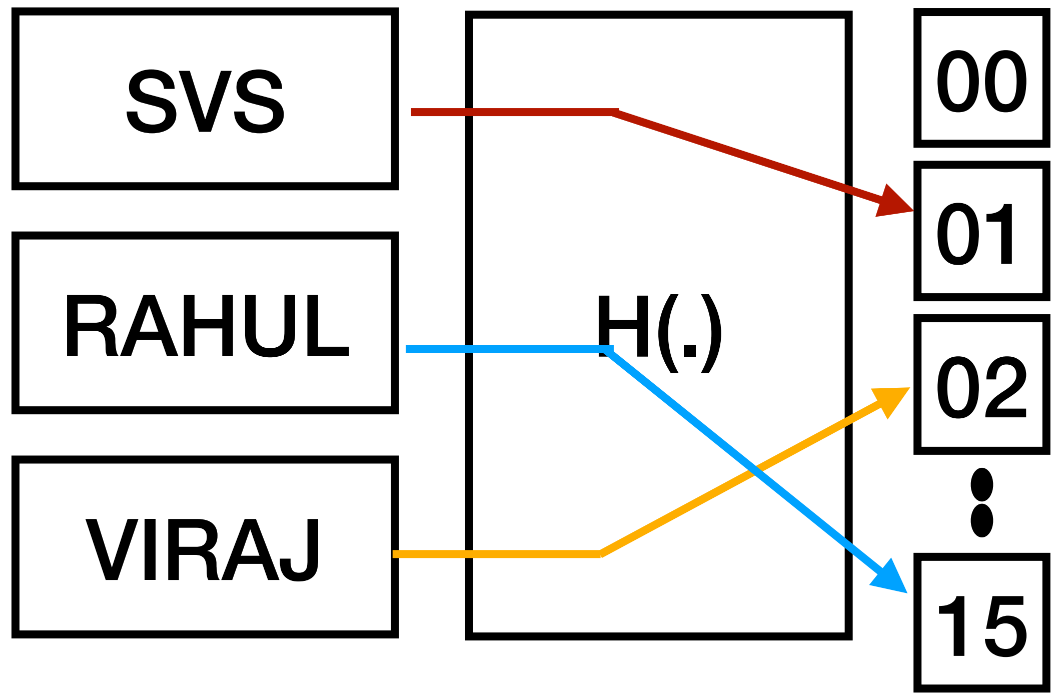
- What are loop invariants?
  - A property that holds:
    - At the start of the loop
    - Is maintained at the end of each iteration of the loop



```
int main(){
  int done;
  do{
    done = 1;

    for (i = 0; i<= length(S) - 1; i++) {

      if ((S[i] == '(' && S[i+1] == ')') ||
          (S[i] == '{' && S[i+1] == '}') ||
          (S[i] == '[' && S[i+1] == ']')) {

        for (j = i + 2; j<= length(S) - 1; j++)
          S[j - 2] = S[j];

        done = 0; i = 0;
        UpdateLength(S);
      } while (done == 1);

      if (length(S) == 0) return true else return false;
```

Hash Functions

Hashing and Hash Tables



- It is a function **H(.)** storing and retrieving data **x** **efficiently**.
    - Maps an arbitrary sized values to a fixed-length output
    - The value returned by H(.) is called a hash, hash code, or digest
- Sometimes the retrieval can be done in **constant time, O(1),** as opposed to **O(log n)** average cost of the Binary Search
- Hash values are stored in a fixed-size table called **Hash Table**
    - The use of hash functions to index in to hash tables is called **Hashing**
- **SUMMARY:** $\forall i \in [0, m-1], H(x) = i \land HT[i] = x$

History of Hashing

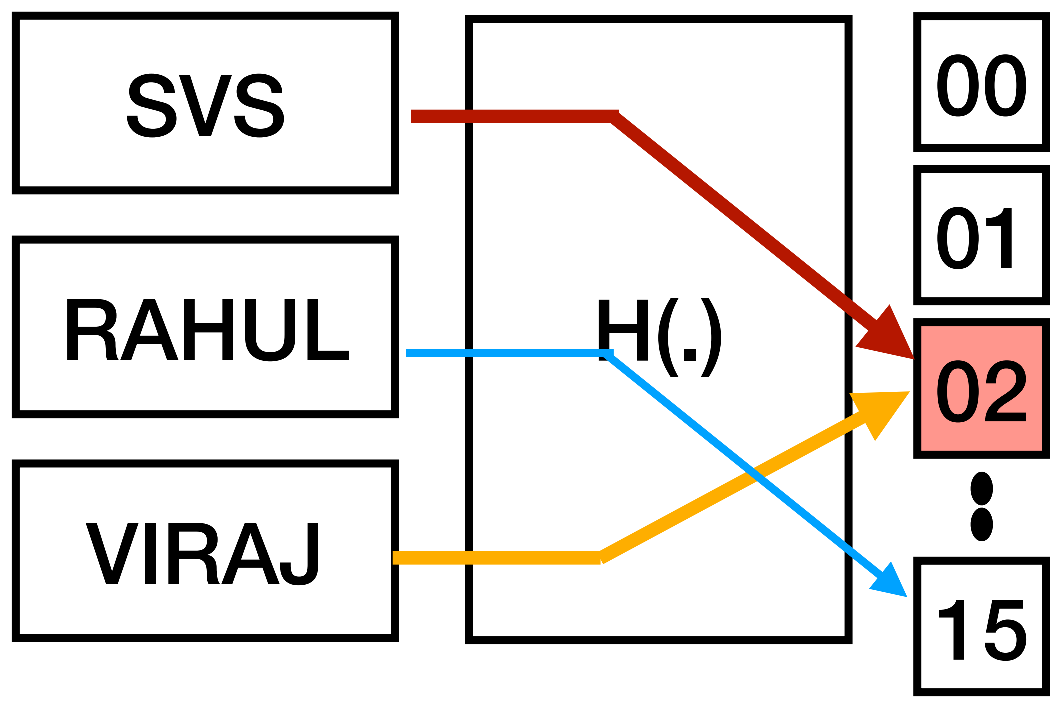
- Hans Peter Luhn: A German researcher from IBM
  - Started as a Textile Engineer; Invented Lunometer: Thread counting Gauge
  - Joined IBM in 1941 in the Information Processing Division
  - Invented Hash Codes, Selective Dissemination of Information etc.

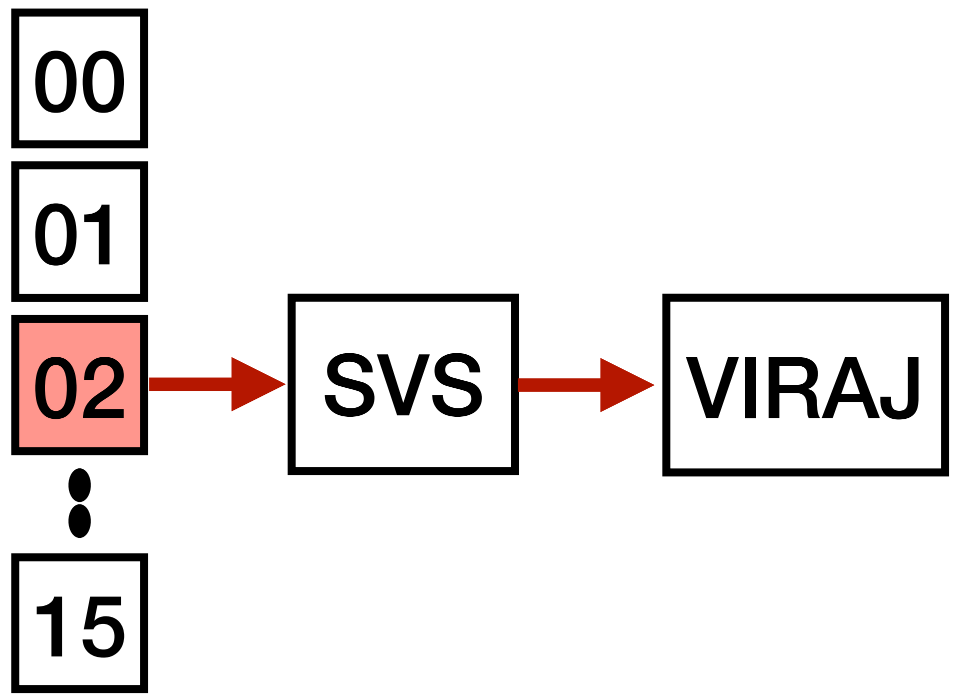
Example Applications of Hashing

- Cybersecurity: Use of crypto hashes
- Blockchain: Hash pointers
- Error detection and correction: in ECCs
- String matching, Data duplication etc.
    - Genomic data processing
- Load Balancing in Distributed Systems
- Machine Learning Model watermarking
- Nearest Neighbour Search in high dimensional spaces: Use of Locality-sensitive hashing

Hashing: Collisions

- Collision: $\exists x, y : x \neq y \wedge H(x) = H(y) = i$
  - This happens when the HT size < Data Domain size
- Collisions adversely affects retrieval, insertion and deletion from HTs.
  - Worst case complexity is **O(n)**
    - One can design the slots/buckets smartly to reduce the worst case complexity to **O(log n)**
  - Thus, hashing is **not good** for applications where multiple records have the same key
- Collision Resolution: Open hashing, Closed hashing





Collision Resolution: Open Hashing

- Suppose **n** is the number of keys and **m** is the number is slots/buckets
    - Then we expect $\alpha = n/m$ elements per bucket. α **is the load factor** of the hash table
- Under **open hashing** — a set of elements can be assigned to a bucket
    - If the set implementation has linear performance then the time complexity of add, insert and search is $O(1 + \alpha)$
    - If $\alpha < \alpha_{max}$ where α_{max} is a constant, then operations have O(1) on average

Collision Resolution: Closed Hashing

- Instead of storing a set of elements at each HT index, only a single element is stored here
- If collision is found, then second possible location is computed.
- Strategies for generating a sequence of hash values:
    - Linear probing: Next free location is linearly searched (ie. searching adjacent slots)
        - Invented by Gene Amdahl, Elaine McGraw, Aurther Samuel from IBM in 1954
    - Quadratic probing: Successive values of arbitrary quadratic polynomial are added to hash index. Eg: $h + 1^2, h + 2^2, \ldots, h + k^2$. **READING ASSIGNMENT**
    - Double hashing: Using a second hash to compute probing step-size. **READING ASSIGNMENT**

A Good Hash Function: Reqs

- **Uniform Distribution**: Distribution of keys uniformly across the HT
    - This minimises collision, improves HT utilisation!
- **Deterministic and Collision Resistant:** Computationally infeasible to find $x, y : H(x) = H(y) \wedge x \neq y$
- **Fast computation**
- **Using all of the input data:** Every part of input affects the output hash
    - $\exists i \in \mathbb{N} : x_i \neq y_i \Rightarrow P(H(x) \neq H(y)) > 0$
- **Dynamic:** Dynamic resizing of HT should be possibles

Cryptographic Hashing vs Hashing

- Cryptographic hashes require additional properties
  - **Preimage Resistance** (One Way property): Let $H(x) = h$. Given h, it is computationally intractable to find x
  - **Second Preimage Resistance**: Given x_1, it should be computationally infeasible to find x_2 s.t. $x_1 \neq x_2 \wedge H(x_1) = H(x_2)$
  - **Avalanche effect:** Small change in the input produces significant change in the output
    - $\forall x, y : d_H(x, y) = 1 \Rightarrow P(H(x)_j \neq H(y)_j) \geq 0.5$

Creating a Simple Hash Function

- Let us use Primes!
  - Choose a prime number P such that is not close to a power of 2. **Why?**
- $\forall k \in \mathbb{I}, H(k) = k \mod P$
- For string inputs: $H(s) = \Sigma \text{ ASCII}(s_i) \mod P$
  - Fast to compute: integer modulo ops are fast to compute
  - Each character of the string affects the output hash value
  - Modulo Prime gives a more uniform distribution, therefore fewer collisions

Some Questions?

- Supposed H(k) = k mod 16. Do you see a problem?
- Suppose you did **binning**. Keys range from 0 to 999, |HT| = 10. Grouped 100 keys to one bucket. Do you see a problem?