- **Hash Table**

  - We have a universe of \( n \) keys and want to store them to a Hash Table with \( m \) slots.

  - We create a hash value \( h(k_i) \) for each key \( k_i \).

  - **Running Time**: the time to compute hash value + one time step to look up in the hash table.

- **Collision**

  - When two keys map to the same hash value \( \rightarrow \) leading to store to the same slot.

  - For ex.

    ![Hash Table Diagram]

    Now if \( h(k_4) = h(k_1) \), we have to store item 4 in the same slot as item 1.

- **Collision resolution strategy**:

  - **Chaining**: Store collided item in linked list. 

    ![Linked List Diagram]

    If \( h(k_5) = h(k_4) \),

    pros: Can store as many keys as possible.

    cons: If we have a long collision, look up time may not be constant.
* Collision resolution strategy (cont'd)

Linear probing: resolve collision by sequentially searching the
hash table for free location.

Ex. \( h(x,i) = (H(x) + i) \pmod{m} \)

where \( H(x) \) is an ordinary hash function, and
\( i \) is the \( i \)th stepsize (all previous \( i-1 \) slots occupied)

Quadratic probing: similar to linear probing but search for
open slots as a quadratic function

Ex. \( h(x,i) = (H(x) + i^2) \pmod{m} \)

Pros of linear/quadratic probing: fast insert, lookup.

Cons: we are limited in the number of elements can be
stored, and we can not empty the slot when
deleting (we have to put a dummy element in
deleted slots)

* Simple uniform hashing assumption.

  * Each key \( k \) is equally likely to be hashed
to any slot of table \( T \), independent of where
other keys are hashed.

  * Load factor \( \alpha = \frac{n}{m} = \text{average number of keys}
per slot
Example: if $a_1$ and $a_2$ would have $\frac{1}{m}$ chance of collision under simple uniform hashing assumption, so under SUHA, both $a_1$ and $a_2$ don't get hashed to slot 1 is $\left(\frac{m-1}{m}\right)^2$.

- Rolling Hash.

In string matching, consider 2 strings $A$ and $B$, we want to find all matching subsequences of length $k$.

$h(A[i:i+k-1]) = A[i] \cdot 26^{k-1} + A[i+1] \cdot 26^{k-2} + \ldots + A[i+k-1]$ so $h(A[i:i+k]) = A[i] + 26 \cdot h(A[i:i+k-1]) - A[i] \cdot 26$

so for string length $k$, total running time to calculate all hash value is $O(k) + O(n-k) = O(n)$.

When attempting to find the longest common substring, use binary search. Hash all substring length $k$ of $A$ into a hash table and look up all substring length $k$ of $B$.

Another example would be looking up words in a dictionary of DNA bank and trying to find patterns.