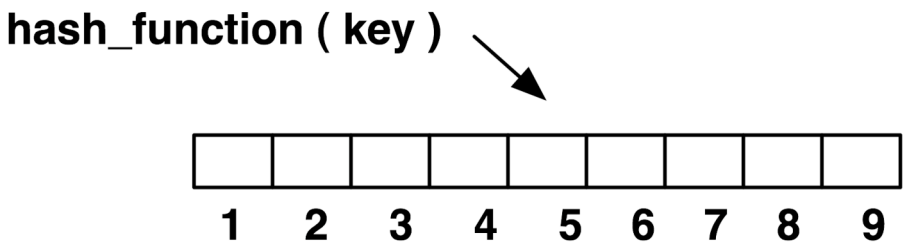
Hash Tables

Definition

- A hash table has the following components
  - **An array – called a table – of size N**
  - **A mathematical function – called a hash function – that maps keys to valid array indices**

    hash_function: key → 0 .. N – 1

- Table entries are stored and retrieved by applying the hash function to the entry key to get the index used to probe the table.



Hash Function Basic Properties

- A hash function consists of two parts
  - » **Hash code function**
    - > **Maps the key into an interval that at least includes the interval [0 , N – 1]**

      $$\text{hash\_code}(\text{key}) \rightarrow \text{integer}$$
  - » **Compression function**
    - > **Maps the integer from the hash code function to the integer interval [0 , N – 1]**

      $$\text{compress}(\text{integer}) \rightarrow 0 .. N - 1$$
    - > **Backward function composition**

      $$\text{compress} \circ \text{hash\_code} : \text{key} \rightarrow 0 .. N - 1$$
    - > **Program function composition**

      $$\text{compress}(\text{hash\_code}(\text{key})) \rightarrow 0 .. N - 1$$

Hash Function Basic Properties – 2

- A good hash function will distribute the expected keys uniformly over the array
  - » **The index probability distribution should follow a uniform distribution**
  - » **Any index is equally likely as any other index**
- In Java the Object class has a default hashCode() method that returns a 32 bit integer.
  - » **In general this is not a good method to use as frequently it is just the address of the object in memory.**
    - > **It is implementation dependent and cannot be relied on**
    - > **It does not do a good job for strings, which are most frequently used as keys**

Polynomial Hash Function

- Good to use for strings
  - » **Have a sequence of items (characters for strings) that have a hash code (ASCII representation for characters)**

$$string =< c_1, c_2, ..., c_k >$$

  - » **Combine the sequence of hash codes using Horner's rule for evaluating polynomials, where m is often a prime number**

$$hash\_code = c_k + m(c_{k-1} + m(c_{k-2} + ... + m(c_2 + mc_1)...)))$$

Compression Functions

- The simplest compression function is to use the modulus function

  Hash_Code mod Table_Size

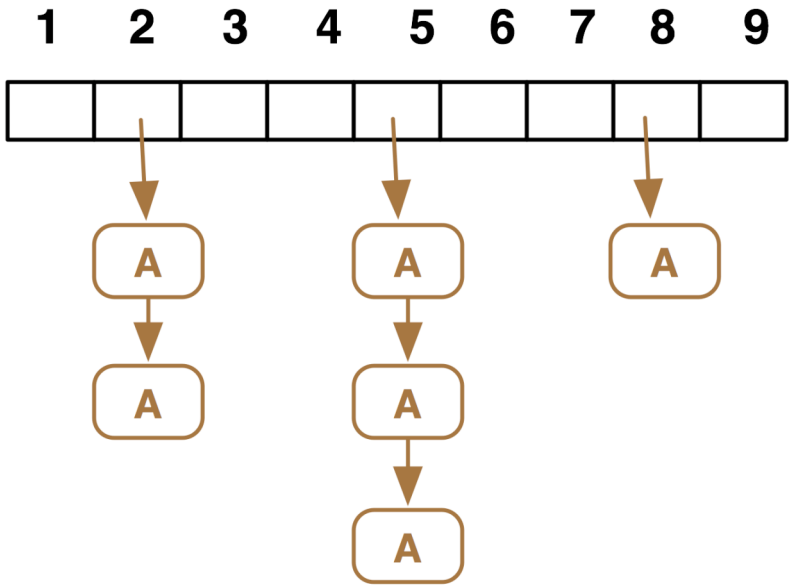
- Sometimes the **MAD** (**M**ultiple **A**dd **D**ivide) is used
  - **(A*hash_code + B) mod Table_Size**
    - **Where**
      A mod Table_Size ≠ 0
      B ≥ 0
      Table_Size is a prime number

Collisions

- The key space is very, very, very much larger than the table size
  - » **Therefore many keys map to the same table index**
  - » **These keys are said to collide**
- As a consequence, we need a collision resolution method

Separate Chaining

- Each entry in the array contains a list of the keys that hash to that bucket.
  - » **O (⌈number_of_entries / Table_size⌉)**
  - » **It is O(1) provided number_of_entries is O(Table_Size)**
  - » **Keep load factor below 90%, 75% is used in Java API**



Open Addressing

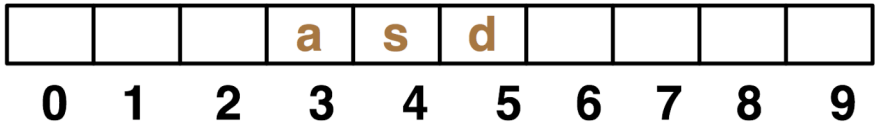
- When collisions occur select another location in the array to store the item.
- The following are common variations
  - » **Linear probing**
  - » **Quadratic probing**
  - » **Double hashing**

Linear Probing

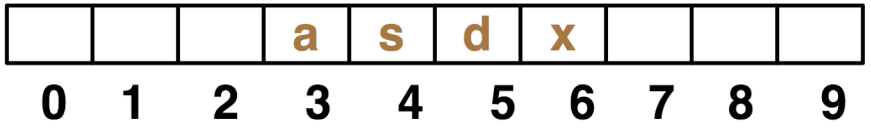
- Initial location is

  i ← hash_code(key)

- For collision resolution iterate over k = 0, 1, 2 … until an empty bucket is found
  - **(i + k) mod Table_Size**
    - **Given the following table**



    - **Suppose hash_code(X) = 3, 4 or 5, then X is stored in location 6**



Linear Probing – Problem

- If the table is relatively empty and keys hash with equal probability for any bucket then insert and search is O(1)
- As the table fills, collision chains build up degrading performance
- As the table becomes over 80% separate chains start combining creating every long chains at an ever increasing rate, degrading performance even more.
  - » **For many keys performance can approach O(N).**
- Heuristic
  - » **have tables about 50% to 80% full to make good use of space and keep performance close to O(1).**

Quadratic Probing

- Initial location is

  i ← hash_code(key)

- For collision resolution iterate over k = 0, 1, 2 … until an empty bucket is found
  - » **(i + k^2) mod Table_Size**

Quadratic Probing Problem

- Secondary clustering
  - » **Set of filled buckets “bounces” around in a fixed pattern**
  - » **May not find an empty bucket if the table is more than 50% full.**

Double Hashing

- Initial location is

  i ← hash_code(key)

- Collision resolution iterate over k = 0, 1, 2 … until an empty bucket is found
  - **(i + k*hash2(key)) mod Table_Size**
- Hash2 cannot evaluate to zero
  - **If key is an integer, a common choice is**
    - **hash2(key) = prime – (key mod prime)**
      - **prime < Table_Size**

Double Hashing Problem

- Have to carefully analyze hash function to minimize clustering for the expected key distribution

Open Addressing vs Chaining

- Open addressing saves space
  - » **no need for pointers**
- Open addressing could be faster
  - » **no need to create list nodes and link them**
- But chaining is found to be competative with open addressing depending upon the load factor in the table array
- Chaining tends to be used more unless
  - » **space is at a premium**
  - » **and clustering can be minimized with open addressing**

Open addressing entry removal

- Cannot just remove a key and set the cell to empty
  - » **Could be part of a chain for a different key**
- Only reasonable algorithm is have cells in three states
  - » **Empty**
  - » **Deleted**
  - » **Full**
    - > **Cells marked as deleted are considered to part of collision chains**
    - > **Cells marked as deleted are eligible to be filled with new entries**

Load Factor Too Large

- As tables become too full they are resized
  - » **Typically double the size of the array**
  - » **Rehash all the entries in the old table into the new table**
    - > **O(Number_of_table_entries)**
- When cost is amortized over all insert operations we can maintain O(1) performance