HASHTABLES

THE MAP ADT

- Map ADT methods:
  - get(k): if the map M has an entry with key k, return its assoiciated value; else, return null
  - put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
  - remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
  - size(), isEmpty()
  - keys(): return an iterator of the keys in M
  - values(): return an iterator of the values in M

KEYS

- Examples of keys:
  - Phone number
  - REDID
  - Employee ID
  - Item Number

- A[8583354282] = “Jane Doe”
- A[88800011] = “Jane Doe”
- A[2345] = “Dixon No.2 Pencils”

NEED FOR HASHTABLES

"If you know where something is, accessing it is easy"

- Hashtables make lookup O(1) operations.

- Insertion and Deletion can also be O(1) operations.
  - Size of the array and the key value pair that needs to be mapped into has no relation to the number of keys possible. Hence lookup, insert and delete will always be O(1)

HASHTABLES AS ARRAYS

- Problems:
  - If the array in which the dictionary needs to mapped is ordered, O(n) is the time Complexity for insert and delete since the array needs to be shifted after every operation.
  - In this case, only integers can be used as a key, not Strings. String must be converted to an integer value.
  - Using static arrays can also be a problem when the number of possible keys is smaller than the array set aside for the hashtable. This increases Space complexity.
    - Example : Phone number for an area of 5000 people. Indexes are phone numbers.
  - Sometimes while converting strings to integers, multiple key , value pair may get mapped to the same array index.
    - This is called a collision.

HASH FUNCTIONS AND HASH TABLES

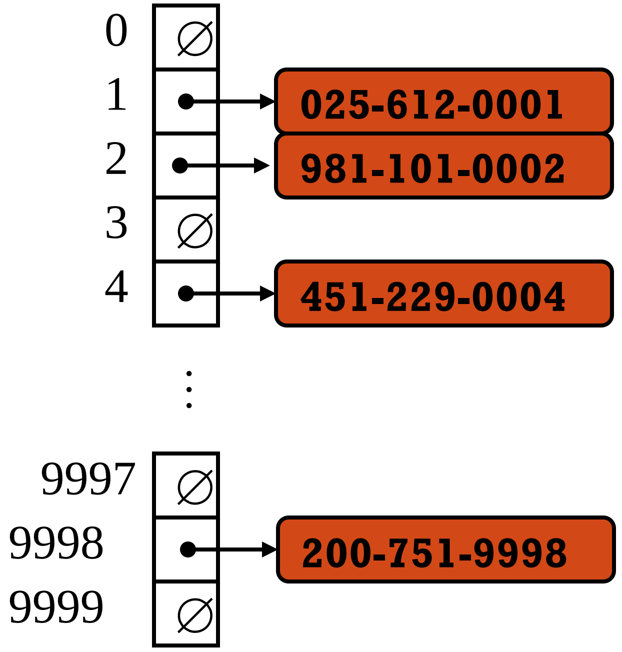
- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:

  $$h(x) = x \bmod N$$

  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, v) at index $i = h(k)$

EXAMPLE

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function $\boldsymbol{h}(\boldsymbol{x}) = \text{last four digits of } \boldsymbol{x}$
- *This is a rare case of perfect hashing where only one key-value pair is mapped to each array index*.



HASH FUNCTION

- Every key value pair is mapped to an array index using a hash function and the hashcode it provides.

Index = HashFunction(Compression(key));

- Hashcode = Compression (key)
- This hash function gives the index in which the key, value pair needs to be mapped.

HASH FUNCTIONS

- A hash function is usually specified as the composition of two functions:

  Compression:
  h_1: keys $\rightarrow$ integers

  Hash:
  h_2: integers $\rightarrow [0, N-1]$

- The compression is applied first, and the hash function is applied next on the result, i.e.,
  $$h(x) = h_2(h_1(x))$$
- The goal of the hash function is to "disperse" the keys in an apparently random way

COMPRESSION

- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
  - Example: take the start address is the array and represent it as an integer

- Integer cast:
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)
  - Directly cast the key as an integer.

STRINGS AS KEYS

- String to integer cast:
  - Consider the ASCII value of each character in the string and concatenate them.
    - "ABC" = 414243
  - Or deduct a constant value from each ASCII value to arrive at the integer value
    - "ABC" = (41-40)+(42-40)+(43-40) = 123

MORE COMPRESSION

- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)
  - Example: phone number can be split and added to get an index.
  - 619+552+2121 = 3292

- Multiplication:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)
  - Example the phone number again:
  - 619*552*2121 = 2483691

HASH CODES (CONT.)

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

    $$a_0\, a_1 \,\ldots\, a_{n-1}$$

  - We evaluate the polynomial

    $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$

    at a fixed value z, ignoring overflows
  - Especially suitable for Strings.

ABCD = $(41-40)*10^3 + (42-40)*10^2 + (43-40)*10^1 + (44-40)*10^0$

ABCD = 1234

HASH FUNCTIONS

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
  - This converts the hashcodes into an index of array size N

- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that
    $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

OTHER HASH FUNCTIONS

- Truncation
  - Extraction or digit selection.
  - Select a hashcode by selecting some digits from the key
  - Easy to compute and Fast
  - Chances of Collision are more

- Midsquare method
  - Square the keys and pick digits from the middle of the key
  - Pick keys based on the size of the hashtable

PROPERTIES OF HASH FUNCTIONS

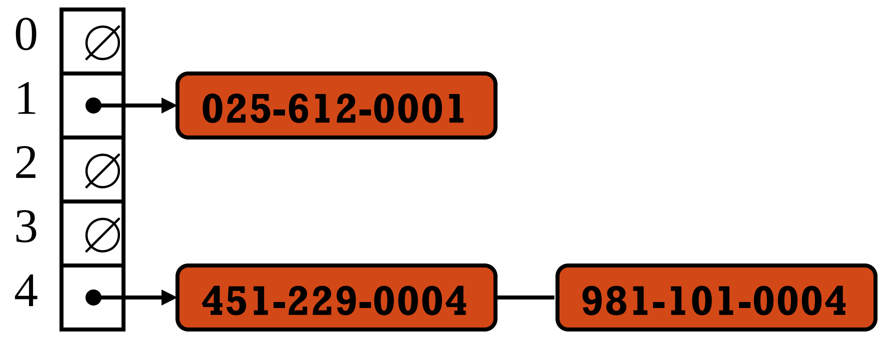
- Any String Object can be used as a key
- The output of the hash function is a 32 bit integer, which will be an array index.
- The probability of all values of integer must be equally likely.
- The method must be repeatable i.e. the same value is returned every time for the same input.
- It must be efficient.
- Equal objects return the same Hashcode.
- Non equal objects return different hashcodes.

COLLISIONS

- Consider the key to be a 10 character string. Total number of possible keys are 26^10.
- This does not fit into the integer range of 2^32 integers that is available in C++
- By using hash functions, we can map all possible keys to a certain integer value, there multiple keys that will be mapped to the same array index.
- This is called a collision.
- There must always be a strategy to combat collisions :
  - Increase the size of the array
  - Change the hash function to evenly distribute the keys.

COLLISION HANDLING

- Collisions occur when different elements are mapped to the same cell
- **Separate Chaining**: let each cell in the table point to a linked list of entries that map there



- Separate chaining is simple, but requires additional memory outside the table

SEPARATE CHAINING - IMPLEMENTATION

```
Algorithm get(k):
Output: The value associated with the key k in the map, or null if there is no
    entry with key equal to k in the map
return A[h(k)].get(k)      {delegate the get to the list-based map at A[h(k)]}

Algorithm put(k,v):
Output: If there is an existing entry in our map with key equal to k, then we
    return its value (replacing it with v); otherwise, we return null
t = A[h(k)].put(k,v)       {delegate the put to the list-based map at A[h(k)]}
if t = null then           {k is a new key}
    n = n + 1
return t
```

```
Algorithm remove(k):
Output: The (removed) value associated with key k in the map, or
    null if there
    is no entry with key equal to k in the map
t = A[h(k)].remove(k)      {delegate the remove to the list-based
    map at A[h(k)]}
if t ≠ null then           {k was found}
    n = n - 1
return t
```

SEPARATE CHAINING PROS & CONS

- **Pros**
  Simple to implement.
  Hash table never fills up, we can always add more elements to the chain.
  Less sensitive to the hash function or load factors.
  It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.
- **Cons**
  Wastage of Space (Some Parts of hash table are never used)
  If the chain becomes long, then search time can become O(n) in the worst case.
  Uses extra space for links.

SEPARATE CHAINING - PERFORMANCE

Suppose we have inserted N items into a table of size hSize.

•In the worst case, all N items will hash to the same list, and we will be reduced to doing a linear search of that list: O(N).

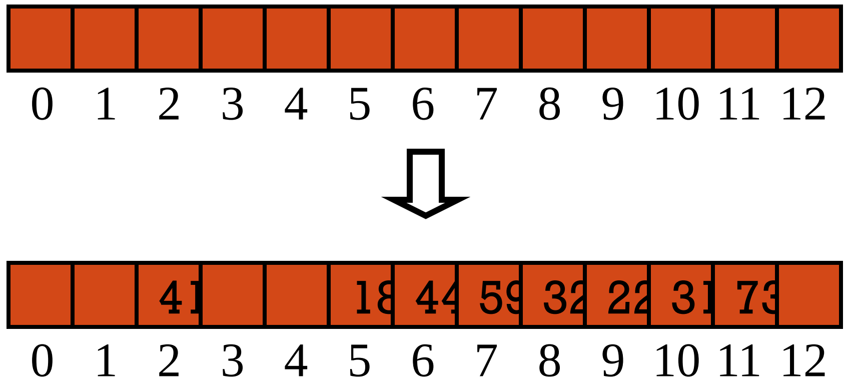
•In the average case, we assume that the N items are distributed evenly among the lists. Since we have N items distributed among hSize lists, we are looking at O(⌈N*hSize⌉).

If hsize is much larger than N, then most lists will have 0 or 1 item, an the average case would be approximately O(1). But if N is much larger than hSize, we are looking at an O(N) linear search sped up by a constant factor (hSize), but still O(N).

Hash tables let us trade space for speed.

LINEAR PROBING

- Open addressing: the colliding item is placed in a different cell of the table
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes



SEARCH WITH LINEAR PROBING

- Consider a hash table *A* that uses linear probing
- get(***k***)
  - We start at cell ***h***(***k***)
  - We probe consecutive locations until one of the following occurs
    - An item with key ***k*** is found, or
    - An empty cell is found, or
    - *N* cells have been unsuccessfully probed

```
Algorithm get(k)
    $i \leftarrow h(k)$
    $p \leftarrow 0$
    repeat
        $c \leftarrow A[i]$
        if $c = \varnothing$
            return null
        else if $c.key() = k$
            return c.element()
        else
            $i \leftarrow (i + 1) \bmod N$
            $p \leftarrow p + 1$
    until $p = N$
    return null
```

UPDATES WITH LINEAR PROBING

- To handle insertions and deletions, we introduce a special object, called *AVAILABLE*, which replaces deleted elements
- remove(***k***)
  - We search for an entry with key ***k***
  - If such an entry (***k***, ***v***) is found, we replace it with the special item *AVAILABLE* and we return element ***o***
  - Else, we return ***null***

- put(***k***, ***v***)
  - We throw an exception if the table is full
  - We start at cell ***h***(***k***)
  - We probe consecutive cells until one of the following occurs
    - A cell ***i*** is found that is either empty or stores *AVAILABLE*, or
    - ***N*** cells have been unsuccessfully probed
  - We store entry (***k***, ***v***) in cell ***i***

QUADRATIC PROBING

- **Clustering:** The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.
- In quadratic probing : we look for i^2space in the array in the ith iteration.

let hash(x) be the array index from a hash function.

If index hash(x) % S is full, then we try (hash(x) + 1*1) % S

If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S and so on…

Quadrating Probing also creates secondary clustering, where chunks of memory spaces with same number of elements are usually clumped together.

DOUBLE HASHING

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series

$$(h(k) + jd(k)) \bmod N$$

for $j = 0, 1, \ldots, N-1$

- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

- Common choice of compression function for the secondary hash function:

$$d_2(k) = k \bmod q$$

where

  - $q < N$
  - q is a prime

- The possible values for $d_2(k)$ are

$$1, 2, \ldots, q$$

LOAD FACTOR

- Another important consideration is the table load factor, λ.
- The load factor is the percentage of the table that is occupied.

N/M

M is the size of the table, and
N is the number of keys that have been inserted in the table

- As the table load factor approaches 1.0 (hash table is almost full), the probe time increases.
- The best case is O(1) and worst case is O(N) for new insert, successful find and unsuccessful find.

- Load factor λ influences the average case.
- Linear Probing, Unsuccessful Search

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

- Linear Probing, Successful Search

$$\frac{1}{2}\left(1+\frac{1}{1+\lambda}\right)$$

PERFORMANCE OF HASHING

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the map collide
- The load factor, λ, affects the performance of a hash table
- The expected running time of all the dictionary ADT operations in a hash table is $O(1)$
- In practice, hashing is very fast provided the load factor is not close to 100%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches