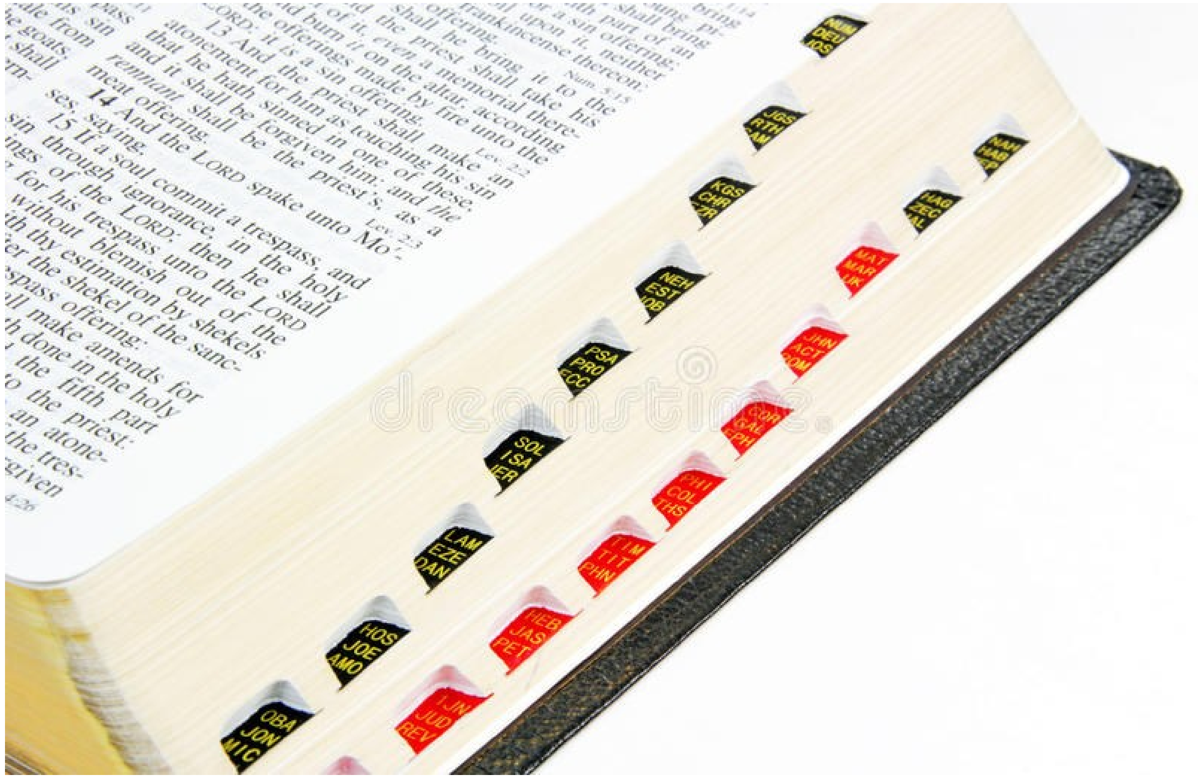
Data Structure

Map

Shin Hong

19 May 2023



DS&A. Chapter 9. Map

Map

- a map allows us to store elements, so they can be quickly located using search keys

- a map is a set of *entries* each of key-value pairs (k, v) such that each key uniquely exists in the set
  - e.g., student number and student record

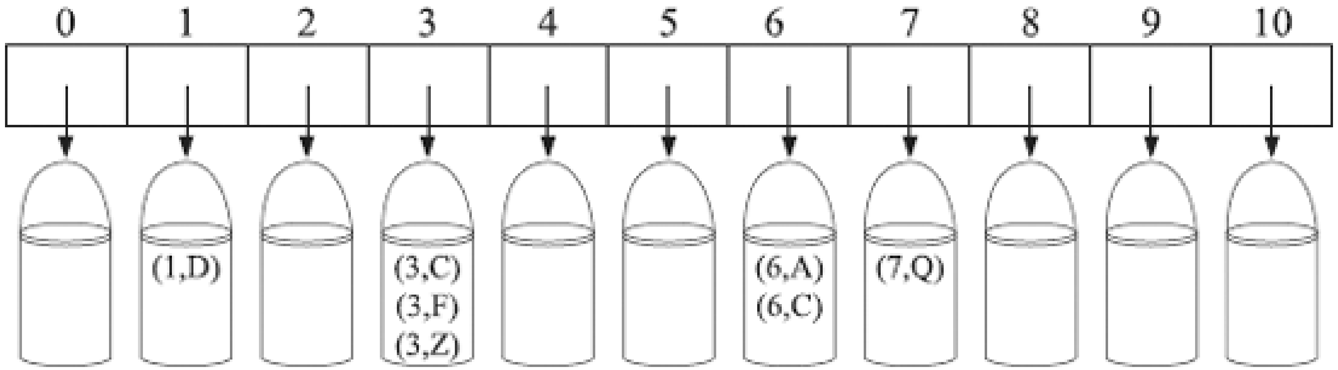
- a key can be used for indicating the address for its value
  - an array is a map where an index is the key of an element

List-based Map

- store entries in a doubly linked list
- operations
  - find(*k*)
  - put(*k*,*v*)
  - get(*k*)
  - erase(*k*)

- every operation takes O(n) times on a map with n entries for linear search of entries

Hash Table



- components
  - a **bucket array** *A* of size *N*, where each cell is a container of entries
  - a **hash function** *h* that converts a key to an integer number between 0 and *N* - 1 (inclusive)

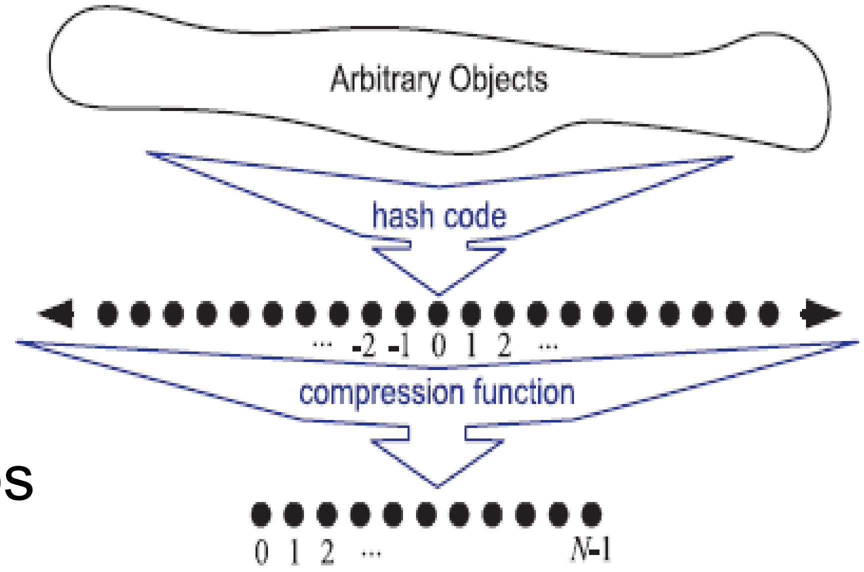
- store an entry (*k*,*v*) to a cell at *A*[*h*(*k*)]
  - multiple entries may exist in a cell at the same time when different key values are mapped to the same number
  - O(1) if the hash function meets the desirable properties

Hash Function

- a hash function is a composition of:
  - mapping a key to an integer called hash code
  - mapping a hash code to a bucket array index

- a hash function is expected to uniformly spread keys over the range of the bucket array
  - a *hash collision* happens when a hash function maps two different keys to the same value
  - hash collisions must be avoided as much as possible

- a hash function must not incur runtime overhead



Converting to Hash Code

- approach 1. summing components
  - use $\sum_{i=0}^{k-1} x_i$ for a value x of a data type with 4*k* bytes
  - limitation
    - keys having the same set of the component collide, e.g., “aaaabbbb” and “bbbbaaaa”

- approach 2. polynomial hash code
  - use $x_{k-1} + a(x_{k-2} + a\left(x_{k-3} + \ \ldots + a\left(x_2 + a(x_1 + ax_0)\right) \ldots \right))$
  - the position of a component is considered
  - empirically, it is found that polynomial hashing is good for hashing strings with a = 33, 37, 39 or 41

Compression

- a hash code cannot be used as a bucket array index mostly since the range of a bucket array is far smaller than the domain of integer

- the compression step maps a hash code to the range of a bucket array

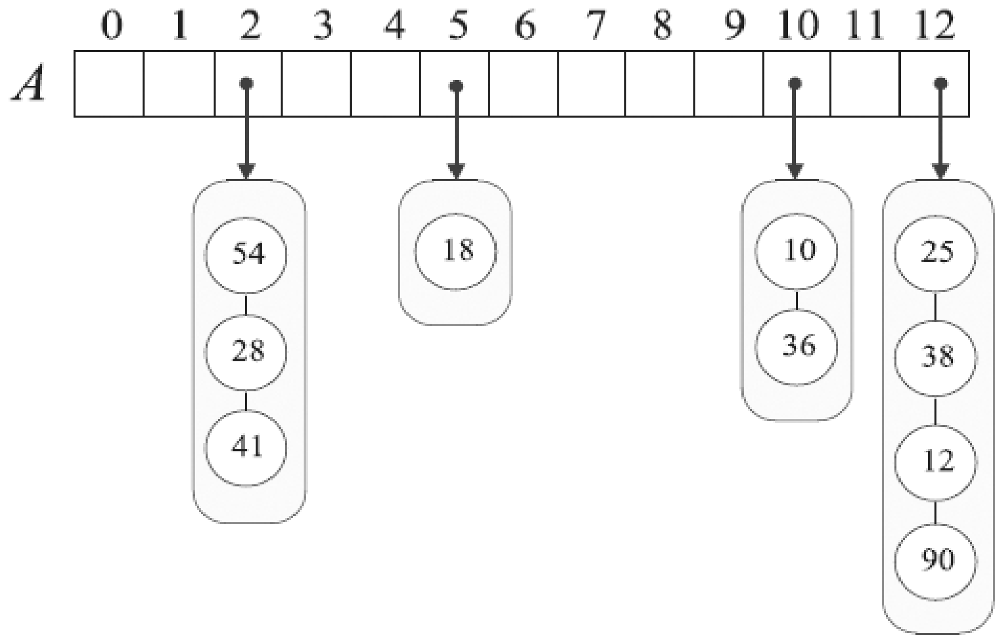
- approach 1. division method
  - $c(x) = |x| \bmod N$ with a prime number N

- approach 2. the MAD method
  - $c(x) = |ax + b| \bmod N$ with a prime number N and non-negatives a and b

Collision Handling Schemes

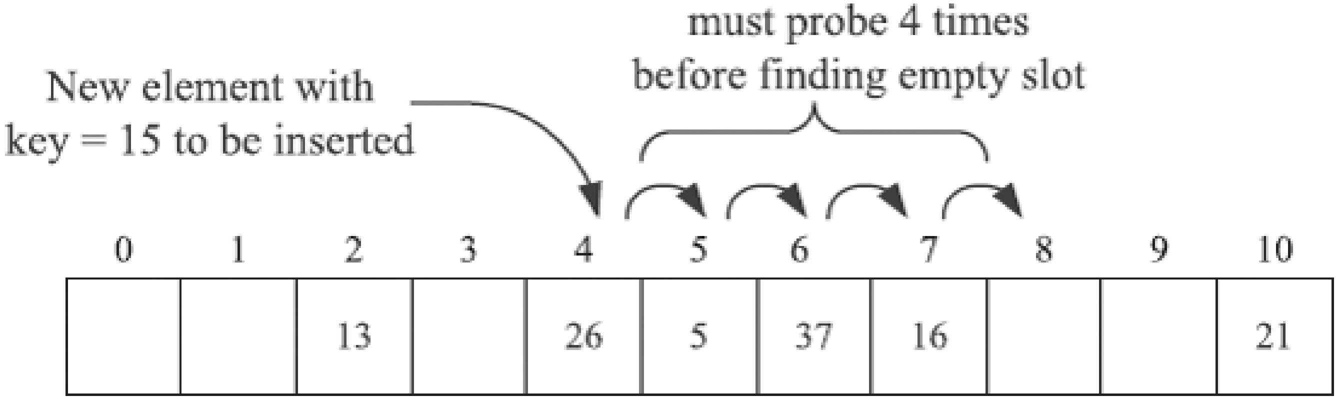
- Two entries can be put to the same bucket if their keys are collided
- A collision handling scheme provides a way to resolve collision cases

- approach 1. separate chaining
  - have a list-based map for each bucket
  - the spreading property of a hash function keeps each list small
  - each map operation takes O($\lceil n/N \rceil$) for n entries and N buckets



Linear Probing

- if $A[h(k)]$ is already occupied, try $A[h(k)+1]$, $A[h(k)+2]$, and so on until we find an empty bucket that can accept a new entry (k,v)
  - called as open-addressing strategy
- search(k) starts from $h(k)$ and check elements until it finds an empty slot
- erase(k) first find the slot holding an element of key k and then marks the slot as “available”



More Open-address Strategies

- Quadratic probing
  - Iteratively try $A[h(k) + j^2]$ for $j = 0, 1, 2, 3, ...$
  - moderate clustering effect

- Double hashing
  - use a secondary hash function h'
  - Iteratively try $A[h(k) + jh'(k)]$ for $j = 0, 1, 2, 3, ...$

- Open-addressing is efficient when the load factor, $\lambda = n/N$, is less than 0.5

Ordered Map

- Keep entries in a map sorted according to a total order defined by a comparator
- Once the entries are stored in an array in ascending order of the keys, searching can be done in O(log *n*) using binary search
  - unlike hash tables, worst-time complexity of searching is bound
  - yet, insertion and removal takes O(*n*)

0	1	2	3	4	5	6	7	8	9	10
4	6	9	12	15	16	18	28	34		

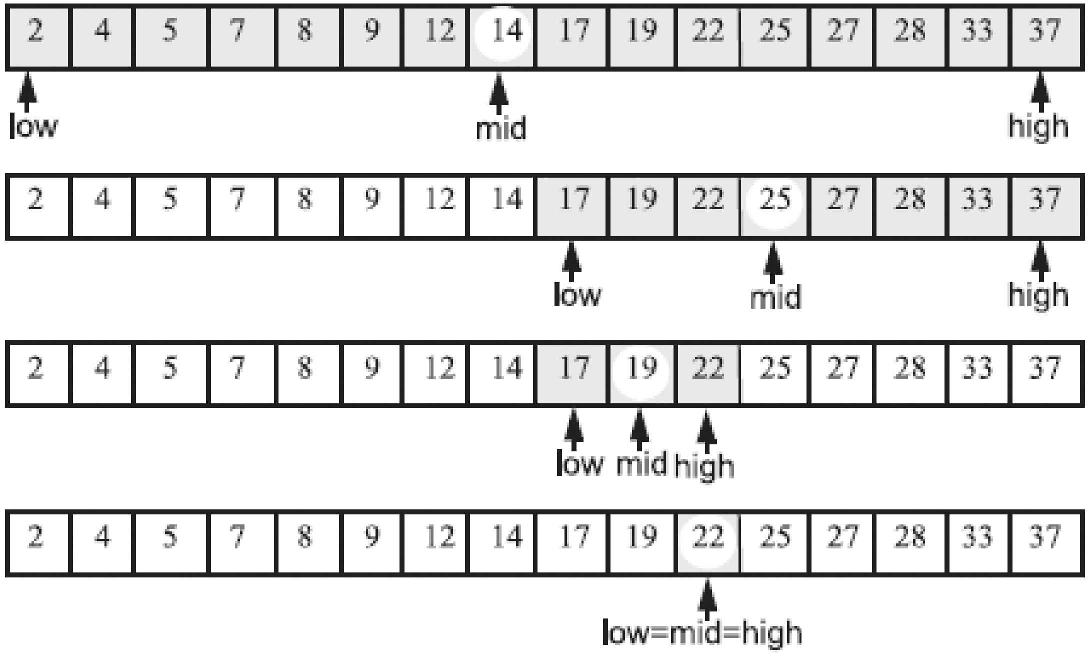
Binary Search

Algorithm BinarySearch (*L*, *k*, *low*, *high*):

Input: An ordered vector *L* storing *n* entries and integers low and high

Output: An entry of *L* with key equal to *k* and index between low and high, if such an entry exists, and otherwise the special sentinel end

if low > high **then**
    return end
else
    mid ← (low + high)/2
    e ← L.at(mid)
    if k == e.key () **then**
        return e
    else if k < e.key() **then**
        return BinarySearch(*L*, *k*, low, mid−1)
    else
        return BinarySearch(*L*, *k*, mid + 1, high)



Time Complexity

Algorithm BinarySearch (*L*, *k*, *low*, *high*):

if low > high **then**
  return end
else
  mid ← (low + high)/2
  e ← L.at(mid)
  if k == e.key () **then**
    return e
  else if k < e.key() **then**
    return BinarySearch(*L*, *k*, low, mid−1)
  else
    return BinarySearch(*L*, *k*, mid + 1, high)

- BinarySearch() is recursively invoked until high – low + 1 < 1
- At each time, the range of candidate is reduced to the half, thus after *m* recursive calls, the size of the range becomes $n/2^m$
- Since $n/2^m$ < 1, *m* is $\lfloor \log\, n \rfloor + 1$