1 Abstract Data Type: Maps

A **map** models a collection of **key-value** entries that is searchable **by the key**. The main operations of a map are for **searching, inserting, and deleting** items. Multiple entries with the same key are **not allowed**. (may be allowed in a multi-map)

1.1 The Map ADT over K,V

Map ADT methods

- V get(K k): if the map M has an entry with key k, return its associated value; else, return null
- V put(K k, V 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
- V remove(K k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- int size()
- boolean isEmpty()
- K keys(): return an iterable collection of the keys in M
- V values(): return an iterable collection of the values in M
- ¡K,V¿ entries(): return an iterable collection of the entries in M

1.2 Performance of a List-Based Map

- `put` would have taken $O(1)$ time if we could just insert the new item at the beginning (or end) of the sequence, but we have to check if the key occurs in the map, so it is $O(n)$.
- get and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key

2 Hash tables

Hash tables are a **concrete data structure** which is suitable for implementing maps. **Basic idea**: convert each key into an index into a (big) array. Look-up of keys and insertion and deletion in a hash table usually runs in $O(1)$ time.

2.1 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: $h(k) = k \ mod \ N$ is a hash function for integer keys. The integer $h(k)$ is called the hash value of key k. 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)

2.2 Compression Functions

- Division
  - $h2(y) \ = \ y \ mod \ N$
  - The size N of the hash table is usually chosen to be a **prime** (hash codes will be spread better)
- Multiply, Add and Divide (MAD):
  - $h2(y) \ = \ (ay + b) \ mod \ N$
  - a and b are nonnegative integers such that $a \ mod \ N \ \neq 0$
  - Otherwise, every integer would map to the same value b

2.3 Collision Handling

Collisions occur when **different** elements are mapped to the **same** cell. A lot of the theory and practice of hashing consists of devising better ways to avoid or handle collisions

- *Separate Chaining*: let each cell in the table point to (e.g.) a linked list of entries that map there

2.3.1 Map Methods with Separate Chaining used for Collisions

Delegate operations to a list-based map at each cell:

- **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)]
- **Algorithm** `remove(k)`:
  - Output:
  - The (removed) value associated with key k in the map
  - `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)]

Separate chaining is **simple** and **fast**, but requires **additional memory** outside the table. When memory is critical then try harder to stick with using the existing memory:

2.4 Open addressing

- 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
- (variant: cell + c where c is a constant)
- Each table cell inspected is referred to as a *probe*
- **Disadvantage**: Colliding items **lump together**, causing **future collisions** to cause a longer sequence of probes

Lazy deletion: dont mark the entry as a blank, but as a deleted and fix the entries later. E.g. see

2.4.1 Search with Linear Probing

- **Algorithm** `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 ← h(k)
    p ← 0
    repeat
        c ← A[i]
        if c = ∅
            return null
        else if c.key () = k
            return c.element()
        else
            i ← (i + 1) mod N
            p ← p + 1
    until p = N
    return null
```

2.5 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

$$(h(k) + jd(k)) \; mod \; N \; for \; j = 0, 1, \ldots, N-1$$

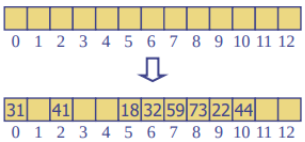
The secondary hash function $d(k)$ **cannot** have zero values. The table size N must be a **prime** to allow probing of all the cells (With a prime N, then eventually all table positions will be reached)

Consider a hash table storing integer keys that handles collision with double hashing

- $N = 13$
- $h(k) = k \bmod 13$
- $d(k) = 7 - (k \bmod 7)$

Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes
18	5	3	5
41	2	1	2
22	9	6	9
44	5	5	5 10
59	7	4	7
32	6	3	6
31	5	4	5 9 0
73	8	4	8



2.5.1 Performance

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time (when **all** the keys inserted into the map **collide**)
- The *load factor* $a = n/N$ affects the performance of a hash table
- In Java, maximal load factor is 0.75 (75%) after that, *rehashed*
- as for Vector, it is good to **double** the table size each *rehash*
- The expected running time of all the map ADT operations in a hash table is $O(1)$
- In practice, hashing is very **fast** provided the load factor is not close to **100%**

2.6 Rehashing

When the table gets too full then *re-hash*: Create a new **larger** table and **new hash function**. Need to (eventually) **transfer** all the entries from the old table to the new one. If do so immediately, then

- can amortise the cost over many entries (as for Vector) and so get an average cost of $O(1)$ again
- but the worst case might be $O(n)$ when the table is rehashed and this might be bad for a real time system
- **Option**: do not transfer all entries **in one go** but do **a few at a time**
- Keep **both** tables until the transfer is complete; but only do **insertions** into the **new table**.

2.7 Comparison of HashMap and PQ

- HashMap does not use the ordering of keys
- E.g. does not implement `min()`
- In a hash table it would need a scan of all the keys in the table, so $O(n)$ (or worse)

- PQ does not allow direct access to a key
- E.g. there is no easy way to do `get(k)`
- In a (standard) heap we would have to walk through all the entries

Reference section

load factor
: a measure of how full the hash table is allowed to get before its capacity is automatically increased.