SET ADT

- A collection of UNIQUE elements.

- No particular orders, can’t be accessed by index.

- Support all mathematical set operations,
  - Union, Intersection, Subset, Difference
    (addAll) (retainAll) (containsAll) (removeAll)

- Add, Remove, Search an element efficiently

Typically implemented with tree or hash table.
Two Classes in JCF that implement Set
- TreeSet, HashSet
TreeSet

- Self-balancing Tree – Red-Black tree
- It’s height is not as strict as AVL but still $O(\log n)$
- Same rotation when the tree is unbalanced
- Each node has a member color instead of height
- Node N can’t be red if it’s parent color is red
- Node N, all it’s black height has to be the same
TreeSet

- Implement SortedSet and NavigableSet

- Iterate as in-order traversal
  Method:
  add(), contains(), remove(), isEmpty(), size(), toArray()

  * first(), last()

  * higher(), lower()

  TreeSet<Integer> sset = new TreeSet<>();
  sset.add(1);
  sset.add(2);
  Integer x = sset.first();
  Integer y = sset.higher(x);
MAP ADT

- A collection of key-value pairs
- key are UNIQUE elements.
- Set is like map with no values.
- May see map as an array, but index with any datatype key
- Add, Remove, Update the value and Search the value by key efficiently

Like Set, the key is implemented with tree or hash table.
Two Classes in JCF that implement Set
- TreeMap, HashMap
TreeMap

- Like TreeSet, TreeMap keeps their keys in sorted order

- Each basic operator takes $O(\log n)$

- It may get a key-value pair entry of datatype Map.Entry<K,V>

- Method:
  put(), get(), remove(), size(), isEmpty(),
  keySet(), entrySet(), values()

* firstEntry(), firstKey(), lastEntry(), lastKey()

* higherEntry(), higherKey(), lowerEntry(), lowerKey()

- Map is not Collection, it does NOT implement Iterable
TreeMap

TreeMap<String, Integer> m = new TreeMap<>();
m.put("001", 1000);
m.put("002", 1001);
m.put("003", 1001);

Map.Entry<String, Integer> pair = m.firstEntry();
Integer x = pair.getValue();

Iterator<Map.Entry<String, Integer>> it =
    m.entrySet().iterator();

//or for each loop
for (Map.Entry<String, Integer> p: m.entrySet())
{
    String k = p.getKey();
    Integer v = p.getValue();
}
HashSet and HashMap

- Has same basic operations of Set and Map
- No duplicate value in Set or key-element in Map
- There’s NO order of key
- basic operations take O(1)

HashSet<Integer> s = new HashSet<>();
Map<String, String> m = new HashMap<>();

It’s implement with a hash table
Hash Table

Uses hash function to generate an array index base on the key to store the value. It should take O(1) to map a key to a slot in array.

John
Mary
Ben
John
Adam

(Collision)

Unlike BST, we can’t iterate through the values in order.
Hash Functions

- be consistent
- uniformly distributed to minimize collision
- efficient to compute take $O(1)$

Two steps:
- Convert to an integer (hash code)
- Map the integer to an index in the array

For integers, we may just use the value as its hash code.

Array capacity sometimes set to a prime number to distribute more evenly.
Collision

We’d like to have as few collision as possible, but if it occurs (when two different key hash to the same index), two common method to handle it.

- Separate Chaining
  - Place all data that hash to the same index in a bucket, usually implemented as a list
- Open Addressing
  - At most one data in each slot, check a sequence of slots until we find an empty one.
Separate Chaining

- **Insert/Update**: Search for the key, if found, update the value, otherwise, add it to the bucket.
- **Search**: Compute the index using hash function, then search the bucket.
- **Delete**: Search for the key, if found, delete from the bucket.

```
hash code % capacity
```

- add 2
- add 4
- add 3
- add 9
- delete 4
- search 16

```
2->9
3
4
```

Page 12
Separate Chaining

The average number of keys each bucket hold is called load factor.
Ex: Array capacity is 6, there’re 12 keys in the array, the load factor is 2
If the load factor is large, we expect to have a lot of collision. The worst-case scenario is when all key hash to the same bucket and the buckets are linked list, the worst-case runtime is O(n)
Since Java 8, it avoids this worst-case by implementing the bucket as a self-balancing tree.
If the load factor is too small, it’s very costly to iterate through the keys. If capacity is M and n keys in the array, the runtime is O(M+n)
HashMap uses 0.75 as it’s load factor, with good hash function, it’s unlikely for a bucket to grow too large.
Open Addressing

- **Insert/Update:** Start search at the computed hash, if found – update, if it’s an empty slot, insert. If the slot is occupied by another key, try a sequence of slot (each successive slot - wrapping around if necessary).

- **Search:** Start search at the computed hash, if the slot is occupied by another key, try a sequence of searching until we either found the key or reach an empty slot.

- **Delete:** Search for the key, if found, delete it. It may be in the middle of a cluster, we shouldn’t break a cluster, thus mark “deleted” instead.

- **Slot** mark “deleted” is available to store new entry but it’s part of a cluster. Thus, when we search, we pass over this slot.

**Note:** Since each slot store at most 1 entry, the load factor can’t exceed 1 for open addressing.
Open addressing Probing

The sequence of slot it looks for an empty slot is called probing. There’re different probe sequences, some well-knowns are linear probing, quadratic probing, and double hashing.

Linear probing:
If the computed hash is occupied by another key, do a linear search, skip all occupied slot, until we found the key, or reach “deleted” value/empty slot where we may insert a new key. If necessary, wrapping around from the end of the table to the top of the table.
Con: tend to cluster into contiguous groups, make the search cluster bigger. If there’s i entries hash to the same slot, then i entries occupy a cluster of i entries, if any entry hash to any of the slot in the cluster, it becomes par to the cluster. The probability this next slot will be occupied next is $(i+1)/\text{capacity}$. 
Open Addressing

- Insert/Update: Search for the key, if found, update the value, otherwise, add it to an empty/deleted slot in the same cluster.
- Search: Compute the index using hash function, then search the cluster
- Delete: Search for the key, if found, mark as “deleted”

hash code % capacity

add 2
add 4
add 3
add 9
delete 4
search 16

2
3
*deleted
9
Hash Code

Convert to integer.
For string, common technique is polynomial hash code. Interpret each character a digit (ASCII) of base $a$

Think:
If we add the ASCII value of each digit:
- o: 111
- p: 112
- s: 115
- t: 116

$\text{stop} = 454$, $\text{tops} = 454$, $\text{pots} = 454$, $\text{spot} = 454$

Use polynomial hash code:
- $\text{stop} = 115a^3 + 116a^2 + 111a^1 + 112a^0$
- $\text{tops} = 116a^3 + 111a^2 + 112a^1 + 115a^0$

$a$ should be a number that uniformly distribute all word, Java uses 31 for $a$