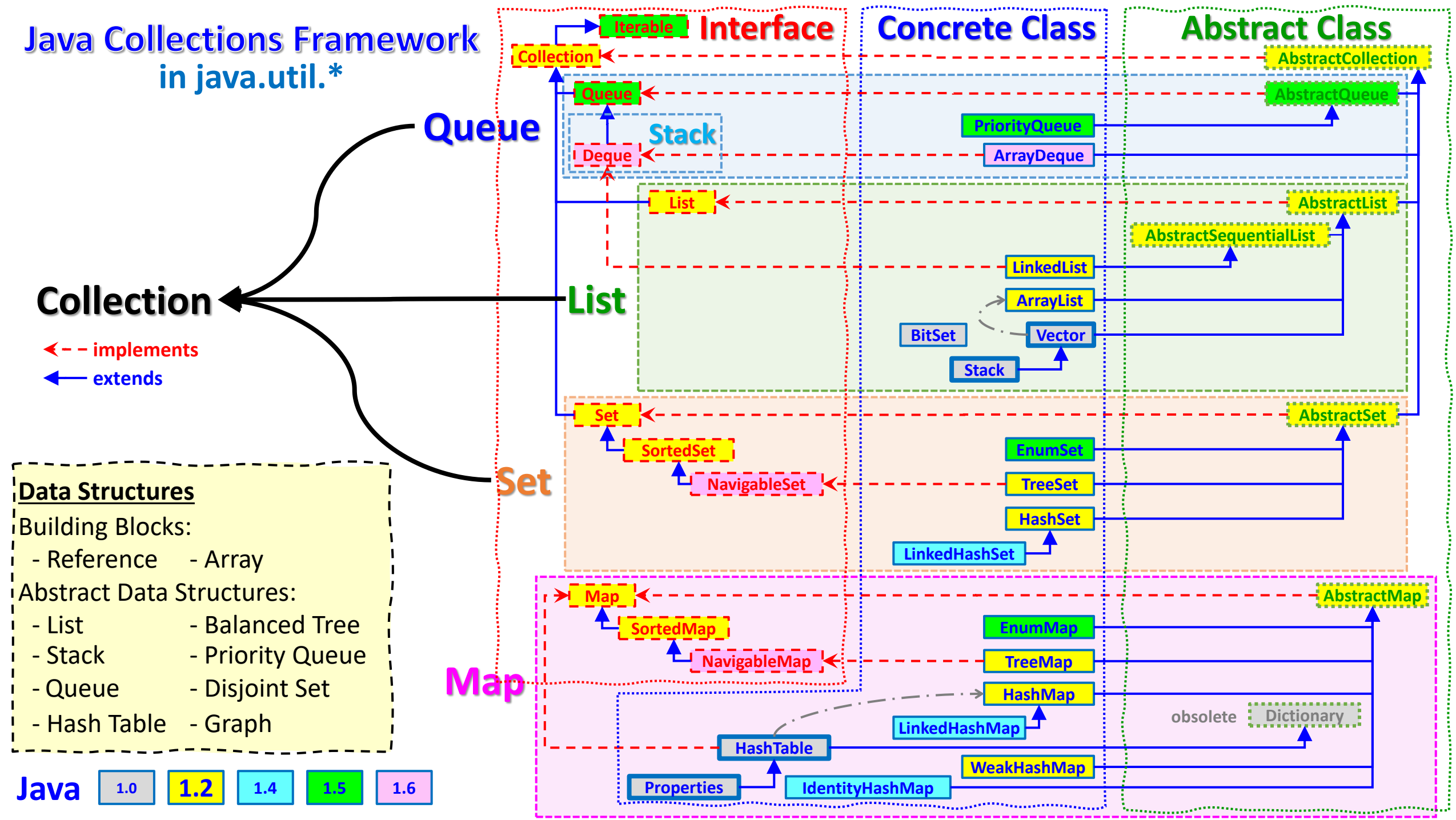
# Java Collections Framework
# in java.util.*




**Data Structures**

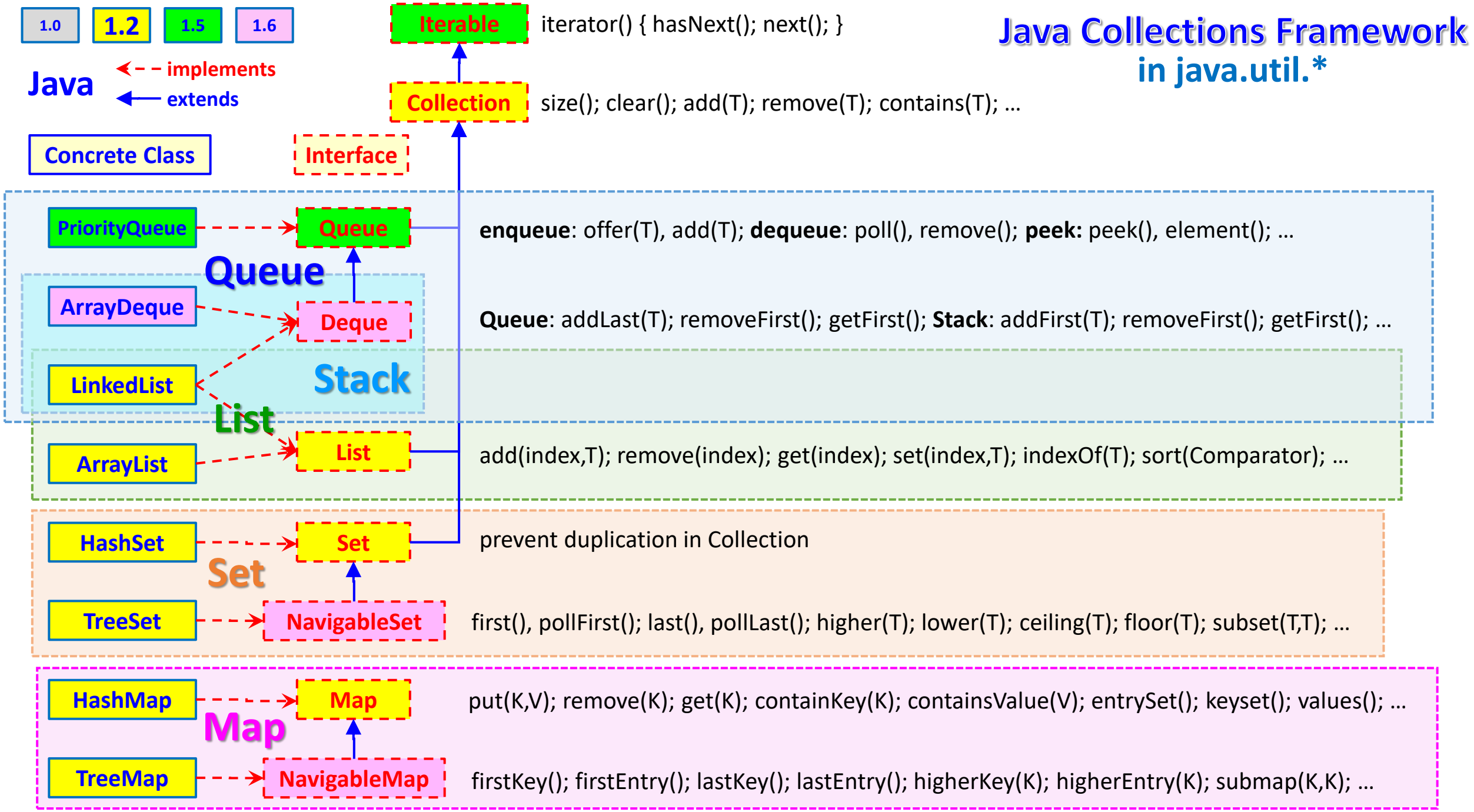
Building Blocks:

- Reference
- Array

Abstract Data Structures:

- List
- Stack
- Queue
- Hash Table
- Balanced Tree
- Priority Queue
- Disjoint Set
- Graph

Java 1.0 1.2 1.4 1.5 1.6


1.0
1.2
1.5
1.6
Java
implements
extends
Concrete Class
Interface
Iterable
iterator() { hasNext(); next(); }
Collection
size(); clear(); add(T); remove(T); contains(T); …
Java Collections Framework
in java.util.*
PriorityQueue
Queue
enqueue: offer(T), add(T); dequeue: poll(), remove(); peek: peek(), element(); …
Queue
ArrayDeque
Deque
Queue: addLast(T); removeFirst(); getFirst(); Stack: addFirst(T); removeFirst(); getFirst(); …
Stack
LinkedList
List
ArrayList
List
add(index,T); remove(index); get(index); set(index,T); indexOf(T); sort(Comparator); …
HashSet
Set
prevent duplication in Collection
Set
TreeSet
NavigableSet
first(), pollFirst(); last(), pollLast(); higher(T); lower(T); ceiling(T); floor(T); subset(T,T); …
HashMap
Map
put(K,V); remove(K); get(K); containKey(K); containsValue(V); entrySet(); keyset(); values(); …
Map
TreeMap
NavigableMap
firstKey(); firstEntry(); lastKey(); lastEntry(); higherKey(K); higherEntry(K); submap(K,K); …

# Java Collections Framework
## in java.util.*

* **Collection**: don't mind and don't care about duplication.
* **Stack**: Collection that is Last-In-First-Out (LIFO).
* **Queue**: Collection that is First-In-First-Out (FIFO).
* **Priority Queue**: Queue that is Higher Priority (based on Comparator) Out First.
* **List**: Collection that is accessible by position; don't allow empty slots.
* **Set**: don't allow duplication. *(In Java, a **Set** is a **Map** where values are a dummy object.)*
* **Map**: Set of *Key-Value Pairs* that don't allow ***key*** duplication.

**Building Blocks:** Pointer, **Reference**, **Array**

**Composite Structures:**

* **List**: **LinkedList**, **ArrayList**, Skip List
* Stack & **Queue > Deque**: **LinkedList**, **ArrayDeque**
* **PriorityQueue**: Binary Heap, Binomial Heap, Fibonacci Heap
* Hash Table: Separate Chaining; Open Addressing
* Disjoint Set (Union-Find Data Structure)
* Tree > Rooted Tree > Binary Tree > Binary Search Tree (BST) > Balanced BST
* Graph > Directed/Undirected
  Single-source/all-pair Shortest Paths
  Minimum-cost Spanning Tree

**Iterable<E>**
- iterator()
+ for-each loop
+ forEach(consumer)

**Collection<E>** extends **Iterable<E>**
# contains null?, duplicate?, ordered?
- contains(object)
- containsAll(collection)
- size(),isEmpty()
- toArray(),toArray(T[])
* add(e),addAll(collection)
* remove(e),removeAll(collection),clear()
* retainAll(collection)
+ stream(),removeIf(predicate)

**Set<E>** extends **Collection<E>**
# no duplicate

**Queue<E>** extends **Collection<E>**
- add(e) [enqueue, IllegalState]
- remove() [dequeue, NoSuchElement]
- element() [peek, NoSuchElement]
- offer(e) [enqueue, false]
- poll() [dequeue, null]
- peek() [peek, null]

**List<E>** extends **Collection<E>**
# ordered, index, no empty position
- get(index),subList(from,to)
- indexOf(object),lastIndexOf(object)
- listIterator(),listIterator(index)
* add(index,e),set(index,e),remove(index)
* addAll(index,collection)
+ sort(comparator)
+ replaceAll(unaryOperator)
[static] of(...e),copyOf(collection)

**Summary of Queue methods**

| | *Throws exception* | *Returns special value* |
|---|---|---|
| Insert | `add(e)` | `offer(e)` |
| Remove | `remove()` | `poll()` |
| Examine | `element()` | `peek()` |

**Comparison of Stack and Deque methods**

| Stack Method | Equivalent Deque Method |
|---|---|
| `push(e)` | `addFirst(e)` |
| `pop()` | `removeFirst()` |
| `peek()` | `getFirst()` |

**Comparison of Queue and Deque methods**

| Queue Method | Equivalent Deque Method |
|---|---|
| `add(e)` | `addLast(e)` |
| `offer(e)` | `offerLast(e)` |
| `remove()` | `removeFirst()` |
| `poll()` | `pollFirst()` |
| `element()` | `getFirst()` |
| `peek()` | `peekFirst()` |

**Summary of Deque methods**

| | First Element (Head) | | Last Element (Tail) | |
|---|---|---|---|---|
| | *Throws exception* | *Special value* | *Throws exception* | *Special value* |
| Insert | `addFirst(e)` | `offerFirst(e)` | `addLast(e)` | `offerLast(e)` |
| Remove | `removeFirst()` | `pollFirst()` | `removeLast()` | `pollLast()` |
| Examine | `getFirst()` | `peekFirst()` | `getLast()` | `peekLast()` |

**Map<K,V>**

# no duplicate key

$ Set<E> is a Map<K,V> with dummy values

- put(key,value),get(key),size(),isEmpty()
- containsKey(key),containsValue(value)
- entrySet(),keySet(),values()

* remove(key),clear(), putAll(map)

+ forEach(BiConsumer)
+ putIfAbsent(key,value),getOrDefault(key,defaultValue)
+ merge(key,value,remappingFunction)
+ remove(key,value)
+ replace(key,value),replace(key,oldV,newV),replaceAll(BiFunction)
+ compute(key,remappingFunction)
+ computeIfAbsent(key,mappingFunction)
+ computeIfPresent(key,remappingFunction)

[static] entry(key,value)

[static] copyOf(map),of(),of(key,value,…),ofEntries(... entries)