For quick reference, you can refer this curated list of ready-to-use data structures and methods.

1) **ArrayList:**

```
ArrayList<Integer> al = new ArrayList<>();
```

1. al.add(5);

The run time of this method is O(1).

2. al.get(index);

The run time of this method is O(1).

3. al.size();

The run time of this method is O(1).

```
al.add(5);                          //adds 5 at the last position
al.add(10);                         //adds 10 at the last position
al.add(20);                         //adds 20 at the last position
System.out.println(al.get(0));      //Prints 5 on new line
System.out.println(al.size());      //Prints the size i.e. 3 on new line
```

2) **HashMap:**

It is Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

```
HashMap<Character, Integer> hm = new HashMap<>();
```

1. hm.put(key, value);

Puts the value of particular key in the hashmap. Overwrite previous value if already present. Operation done in O(1) time.

2. hm.get(key);

Returns the value of particular key in O(1) time.

3. hm.containsKey(key);

Returns true if this map contains a mapping for the specified key. The run time is O(1).

4. hm.isEmpty();

Returns true if this map contains no key-value mappings.

```
hm.put('a', 1);                     //Puts value 1 for character 'a'.
```

```
hm.put('b', 3);                          //Puts value 3 for character 'b'.
System.out.println(hm.get('b'));         //Prints value for 'b' i.e. 3 on new line
System.out.println(hm.containsKey('a')); //Prints true
System.out.println(hm.isEmpty());        //Prints false
```

3) **HashSet:**

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.

HashSet<Integer> hs = new HashSet<>();

1. hs.add(3);

Adds the specified element to this set if it is not already present. Done in O(1) time.

2. hs.contains(1);

Returns true if this set contains the specified element. Run time is O(1).

3. hs.size();

Returns the number of elements in this set (its cardinality). Run time is O(1).

4. hs.isEmpty();

Returns true if this set contains no elements.

```
hs.add(1);                       //Adds 1 to the hashset
hs.add(2);                       //Adds 2 to the hashset
System.out.println(hs.contains(1)); //Prints true on the new line.
System.out.println(hs.size());   //Prints 2 on the new line.
System.out.println(hs.isEmpty()); //Prints false on new line.
```

4) **TreeMap**

A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations.

TreeMap<Integer, Integer> tm = new TreeMap<>();

1. tm.put(key, value);

Puts the value of particular key in the treemap. Overwrite previous value if already present. Operation done in O(log n) time.

2. tm.get(key);

Returns the value of particular key in O(log n) time.

3. tm.containsKey(key);

Returns true if this map contains a mapping for the specified key. The run time is O(log n).

4. tm.isEmpty();
   Returns true if this map contains no key-value mappings.

```
tm.put(3, 1);                          //adds value 1 for key 3 in the map
tm.put(4, 2);                          //adds value 2 for key 4 in the map
tm.put(2, 8);                          //adds value 8 for key 2 in the map
System.out.println(tm.get(3));         //Prints value 1 on the new line.
System.out.println(tm.containsKey(1)); //Prints false on new line
System.out.println(tm.isEmpty());      //Prints false on new line
```

Note: TreeMap always keeps the elements in a sorted(increasing) order, while the elements in a HashMap have no order. TreeMap also provides some cool methods for first, last, floor and ceiling of keys.

5) **PriorityQueue:**

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used. A priority queue does not permit null elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects.

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
```

1. pq.add(1);

Inserts the specified element into this priority queue. Done in O(log n) time.

2. pq.remove();

Retrieves and removes the head of this queue. This method differs from poll only in that it throws an exception if this queue is empty. Run time is O(log n).

3. pq.peek();

Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. Run time is O(1).

4. pq.isEmpty();

Returns true if this collection contains no elements.

```
pq.add(10);                          //Adds 10 to the queue
pq.add(12);                          //Adds 12 to the queue
pq.add(2);                           //Adds 2 to the queue
System.out.println(pq.remove());     //Prints 2 and remove it from the queue
System.out.println(pq.peek());       //Prints 10 on new line
System.out.println(pq.isEmpty());    //Prints false in new line
```