java-ds-collections-mostly used

Basic Idea to understand :Map -> V get(k1)

- get : method,
- k1 : accept key as an value,
- V : return datatype

---

1. Map

1. Definition ->Map<K, V> map = new HashMap<>();
2. insert / update -> V put(k1, v1); // TC: O(1)
3. delete -> V remove(k1); // TC: O(1)
4. get -> V get(k1); // TC: O(1)
5. size -> int size(); // TC: O(1)
6. check for Empty -> boolean isEmpty(); // TC: O(1)
7. value present -> boolean containsKey(k1); // TC: O(1)
8. remove all map values -> clear(); // TC: O(2n + 1) -> O(n) *(n-key, n-value, 1 for map itself)*
9. To increment map value if its exist

```
for(char current : input.toCharArray()){
            count.put(current, count.getOrDefault(current, 0) + 1);
    }
```

---

2. ArrayList // Collection

1. Definition -> ArrayList list = new ArrayList<>();
2. insert -> boolean add(t) **[TC: O(1)]** / add(int index, T) **[TC: O(n)]**
3. delete -> T remove(int index); // TC: O(n) as you have to shuffle the elements above that point
4. set/update index value -> T set(int index, T); // TC: O(1)

5. get index-> T get(int index); // TC: O(1)
6. size -> int list.size(); // TC: O(1)
7. clear elements -> void clear(); // TC: O(n) & removeAll : O(n^2).
8. check for Empty -> boolean isEmpty(); // TC: O(1)
9. value contain check -> boolean contains(t); // TC: O(n)
10. get Index of value -> int indexOf(t); // TC: O(n), checking each element one by one
11. non premitive to premitive list -> toArray(); // TC: O(n)
12. Sorting for List ->
    - Collections.sort(list, (a, b) -> a - b); // ascending , TC: O(nlogn)
    - Collections.sort(list, (a, b) -> b - a); // descending , TC: O(nlogn)

---

3. Array

1. Definition ->T arr [ ]= new T[N]; // N: static size , T : datatype
2. insert -> arr[index] = v1; // TC: O(1)
3. update -> arr[index] = v2; // TC: O(1)
4. get -> T arr[index] // TC: O(1)
5. size -> int arr.length // TC: O(1)
6. Arrays.fill(arr, 0); // filled array with value=0, TC: O(n)
7. Sorting -> TC: O(nlogn)
    - premitive (int[] ..)
        - Arrays.sort(arr); // default ascending,
    - non-premetive (Integer[] ..)
        - Arrays.sort(arr); // default ascending
        - Arrays.sort(arr, (a,b) -> b-a); // descening

---

4. Stack

1. Definition ->Stack st = new Stack<>();
2. insert -> T push(t); // TC: O(1)
3. size -> int size(); // TC: O(1)

4. look up for head element -> T peek(); // TC: O(1)
5. remove head element -> T pop(); // TC: O(1)
6. check for Empty -> boolean isEmpty(); // TC: O(1)

---

5. Queue

1. Definition -> Queue queue = new LinkedList<>();
2. insert -> boolean add(t); // TC: O(1)
3. size -> int size(); // TC: O(1)
4. look up for head element -> T peek(); // TC: O(1)
5. remove head element -> T poll(); // TC: O(1)
6. check for Empty -> boolean isEmpty(); // TC: O(1)
7. points to remember :

- queue poll vs stack pop
- queue add vs stack push
- we can define queue via LinkedList, PriorityQueue based on use case

---

6. String / StringBuilder

1. Definition -> String str = new String();
2. size -> int length();// TC: O(1)
3. convert to char Array -> toCharArray(); // TC: O(n)
4. value for specific index -> charAt(int index); // TC: O(1)
5. substring from string -> substring[a,b) // a : inclusive, b: Exclusive, TC: O(n)
6. transform to Lowercase -> toLowerCase(); // TC: O(n)
7. transform to UpperCase -> toUpperCase(); // TC: O(n)
8. replace all characters in string -> replaceAll(from, to) // TC: O(n)
9. Some useful Character properties

- Character.isLetter();
- Character.isAlphabetic();
- Character.isUpperCase();
- Character.isLowerCase();

- Character.isDigit();

Concatenation

- T str1 + str2
- StringBuilder ->
  - new StringBuilder() / new StringBuilder(int)
  - append("adding string") // better way to do
  - toString() // converting back to string

10. Change alphabet case without in-built functions:

```
a) Uppercase to Lowercase : (OR with space)
    char currentChar = 'A';
    char result = (char)(currentChar | ' ');

b) Lowercase to Uppercase : (AND with underscore)
    char currentChar = 'A';
    char result = (char)(currentChar & '_');
```

---

7. HashSet

1. Definition ->Set set = new HashSet<>();
2. insert / update -> boolean add(t); // TC: O(1)
3. delete -> boolean remove(t); // TC: O(1)
4. get -> boolean contains(t); // TC: O(1)
5. size -> int size(); // TC: O(1)
6. check for Empty -> boolean isEmpty(); // TC: O(1)
7. remove all set values -> clear(); // TC: O(n)

Helpful link for Time Complexity(TC).

- https://stackoverflow.com/questions/7294634/what-are-the-time-complexities-of-various-data-structures
- https://www.baeldung.com/java-collections-complexity

---

8. Mostly in graph problems we need to add edges between nodes. If node is not created, we first create list for the children and then add edges

```
Map<Integer, List<Integer>> graph = new HashMap<>();
    for(int[] edge : edgeList){
        if(graph.containsKey(edge[0]) == false){
            graph.put(edge[0], new ArrayList<>());
        }
        graph.get(edge[0]).add(edge[1]);
    }

Above code can be abstracted as follows :

for(int[] edge : edgeList){
        graph.computeIfAbsent(edge[0], val -> new ArrayList<>()).add(edge[1]);
    }
```

---

9. We often need to return empty Set, List or Map when input is invalid. To make code more readable:

```
    return Collections.EMPTY_SET;
    return Collections.EMPTY_LIST;
    return Collections.EMPTY_MAP;
```