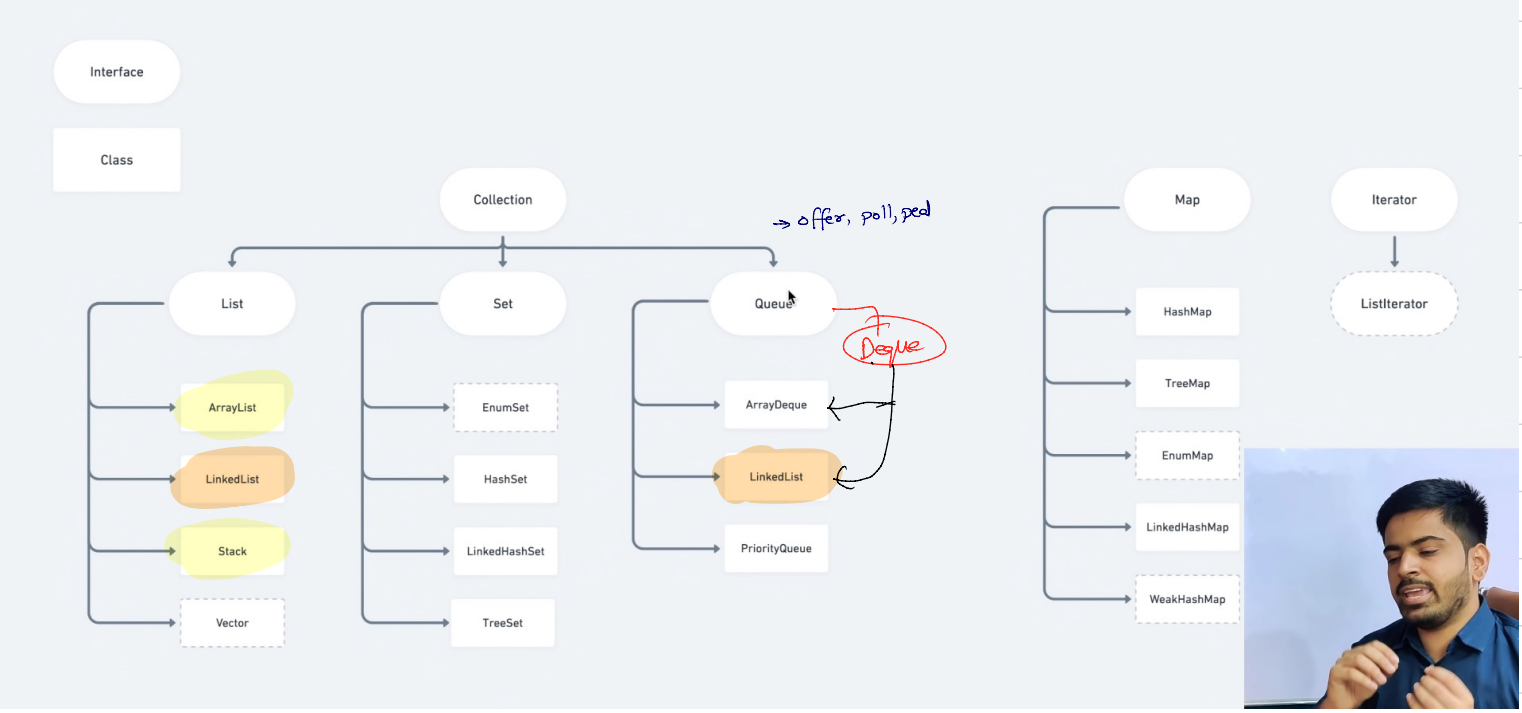
Collections



- isEmpty()
- size()
- contains()
- Iterator()
- toArray()

Common

https://www.linkedin.com/in/kapilyadav22

List

.length() → for strin

→ ArrayList :- dynamic size array

Syntax :- ArrayList<String> names = new ArrayList<>();

→ names.add("Kapil"); names.add(index, value);

In Java, an ArrayList is a dynamic array implementation that allows you to add, remove, and access elements easily. You can use the ArrayList class from the java.util package. Here's a simple example:

→ Default initial size is 10 and it grows double when it get completely filled.

→ names.get(index);

O(N) → names.remove("Kapil"); | → names.remove(Integer.valueOf(value)); ← for integer values in Arraylist

O(N) → names.remove(index); | → names.remove(index);

```
→ ArrayList<String> List2 = new ArrayList<>();
  List2.add("Raj");
  List2.add("Abhishek");

→ names.addAll(List2);
        ↳ to add all elements of List2 in names.
→ names.size(); → No of elements
→ names.clear(); //empty the names list

→ names.set(index, value); // update the value at particular index. O(1)

→ names.contains(value); // checks if value is present or not

→ for(int i=0; i<names.size(); i++)
  {  Sy.out.println(names.get(i));
  }

→ for(int element : names)
  {  s.o.println(element);
  }

Iterator<Integer> it = list.Iterator();
while(it.hasNext()) {
    s.o.println("iterator" + it.next());
}
```

C++ Stl

```
vector<int> data;
data.push_back(value);
data.pop_back();
data.size();
data.clear(); //empty the vector
data.insert(index, value);

vector<int> numbers(size, initial value);
find(numbers.begin(), numbers.end(), valueToFind);

numbers.erase(numbers..begin() + 2);
```

Stack

```
import java.util.Stack;

Stack<String> st = new Stack<>();
```

→ `st.push("kapil");` } push in LIFO order
→ `st.push("Vimal");`

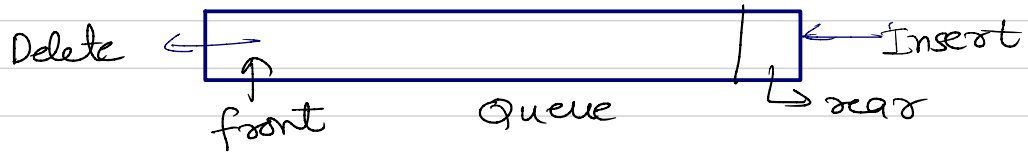
→ `st.peak();` // top element

→ `st.pop();` // pop top element

In C++ Stl

→ `st.push();` // insert in LIFO manner
→ `st.pop();` // pop the top element
→ `st.top();` // get the top element
→ `st.size();` // get the size of stack

Queue



→ Queue is an Interface.
→ import java.util.Queue;

Queue can be create using ArrayDeque, LinkedList and PriorityQueue

1. Create Queue using LinkedList

```
Queue<Integer> que = new LinkedList<>();
```

→ Insert in queue :- There are two ways add() and offer()

```
que.add(10);
que.offer(10);
```

```
// C++ stl
que.push(10);
```

→ Both are same for unbounded queue

Key Difference:

- The primary difference between `add()` and `offer()` is how they handle cases where the queue has a capacity restriction, and adding an element is not possible.
- `add()` throws an exception (`IllegalStateException`) if the element cannot be added due to capacity restrictions.
- `offer()` returns a boolean (`true` if added successfully, `false` if not) without throwing an exception.

When working with unbounded queues or queues with sufficient capacity, the distinction between `add()` and `offer()` might not be critical. However, in scenarios where capacity constraints matter, using `offer()` allows you to handle the failure gracefully without throwing an exception.

c++ stl

Remove elements from Queue :-

que.poll(); //removes front element

que.pop();

que.peak(); returns null if queue is empty

que.element(); front element. If the queue is empty, it throws a **NoSuchElementException**.

que.front();

→ LinkedList using List;

```
List<String> names = new LinkedList<>();
```

→ Rest operations will be same as List (ArrayList which we used above)
→ add, addAll, get, contains, clear, set.

NOTE:- We can also use LinkedList directly from java.util.LinkedList.
It contains all the methods of list and queue.
LinkedList implements both Queue and List Interface.

NOTE:- It's a good idea to store LinkedList in type Queue or List, so that we can get more Code flexibility.

Priority Queue

→ import java.util.PriorityQueue;

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

OR

```
Queue<Integer> pq = new PriorityQueue<>();
//By default it uses minheap.
```

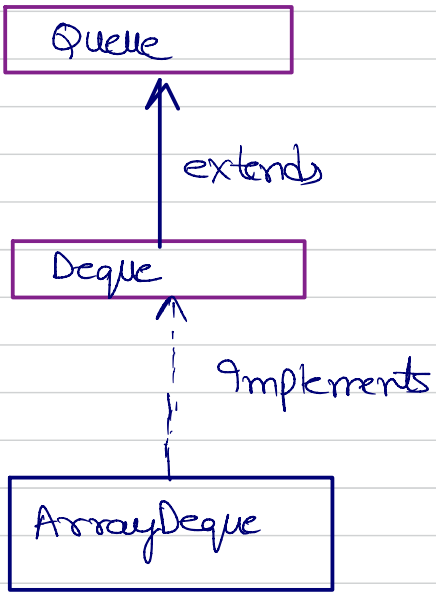
→ uses same methods as Queue – offer, poll, peak, size

→ Using pq as Maxheap

```
Queue<Integer> pq = new PriorityQueue<>(Comparator.reverseOrder());
```

ArrayDeque

→ Implements Deque Interface which extends Queue Interface.
→ java.util.ArrayDeque
→ Double ended Queue



→ `ArrayDeque<Integer> adq = new ArrayDeque<>();`

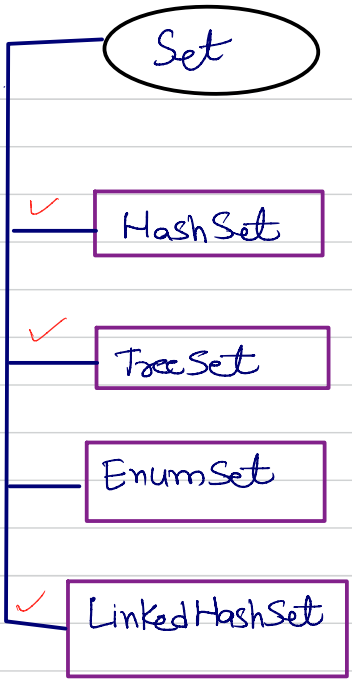
```
adq.offer(value);
adq.offerFirst(value);
adq.offerLast(value); // same as offer
```

```
adq.peek()     // front
adq.peekFirst() // same as peek
adq.peekLast() // last
```

```
adq.poll()     // front
adq.pollFirst() // same as poll
adq.pollLast()  // last
```

NOTE :- We can implement queue & stack using ArrayDeque.

Set



HashSet — unordered set

→ stores unique elements only.

```
Set<Integer> set = new HashSet<>();
```

```
set.add(1);
set.add(2);
set.add(3);
set.add(4);
```
} O(1)

→ stores elements in random order.

- → `set.remove(value);`
- → `set.contains(value);`
- → `set.isEmpty();`
- → `set.size();`
- → `set.clear();`

LinkedHashSet — uses LinkedList

→ Maintains the insertion order of elements

TreeSet → uses BST

→ Maintains the sorted order of elements. — O(logN)

Set of Custom classes (hashCode and equals)

```java
package equalsandHashcode;

import java.util.Objects;

public class Student {

    String name;
    int rollNo;

    public Student(String nameString, int rollNo) {
        this.name = nameString;
        this.rollNo = rollNo;
    }

    @Override
    public String toString() {
        return "Student [name=" + name + ", rollNo=" + rollNo + "]";
    }

    @Override
    public int hashCode() {
        return Objects.hash(rollNo);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Student other = (Student) obj;
        return rollNo == other.rollNo;
    }


}

```

```java
package equalsandHashcode;

import java.util.HashSet;
import java.util.Set;

public class setExample {

    public static void main(String[] args) {
        Set<Student> studentSet = new HashSet<>();

        studentSet.add(new Student("Kapil", 26));
        studentSet.add(new Student("Rahul", 22));
        studentSet.add(new Student("Ajay", 25));
        studentSet.add(new Student("Yogesh", 21));
        studentSet.add(new Student("Vineet", 20));
        studentSet.add(new Student("Ajay", 26));

        System.out.println(studentSet);
    }


}

```

→ using hashcode and equals to, we can differentiate custom class based on any field (Roll no in our case).

MAP

HashMap

```
Map<String, Integer> numbers = new HashMap<>();

numbers.put("One", 1);
numbers.put("Two", 2);
numbers.put("Three", 3);

numbers.putIfAbsent("Two", 22);

for(Map.Entry<String, Integer> e : numbers.entrySet())
{
        Sout(e);  // whole entry
        e.getKey()   // key
        e.getValue(); // value


for(String key : numbers.keySet()) {
        sout(key);
}

for(Integer value: numbers.values()){
        sout(value);
}
```

- numbers.containsKey(key);
- numbers.containsValue(value);
- isEmpty();
- clear();
- numbers.remove(key);

TREEMAP → O(logN)

- Maintains the Sorted Ordering of elements.

ArraysClass

→ Use to perform operations in Arrays (Default Arrays)

1. Binary Search :-
   → Works on sorted array.

```
int index = Arrays.binarySearch(arrayName, valuetoSearch);
```

2. Sorting :- `Arrays.sort(arrayName);`

   parallel sort → (Min 8192 elements)

3. Set values in every Index :-

```
Arrays.fill (arrayName, value);
```

Collections Class

→ import java.util.Collections;
→ useful for collection framework DS;

```
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.add(3);
list.add(4);
```

Min Element :- Collections.min(list);
Max Element :- Collections.max(list);
frequency :- Collections.frequency(list, 2);

Sorting :- Collections.sort(list);
→ Collections.sort(list, Comparator.reverseOrder());

Custom Comparator

1. Using Comparable class :-

Inside it, we have compareTo(Student that);

```
public int compareTo(Student that) {
    return this.rollNo - that.rollNo;
}
```

→ Collections.sort(list);

2. Using Custom Comparator

```
Collections.sort(list, new Comparator<Student>(){

    @Override
    public int compare(Student o1, Student o2) {
     return o1.name.compareTo(o2.name);
     }
});
```

```
Collections.sort(list, (o1,o2) -> o1.name.compareTo(o2.name));
```