Agenda

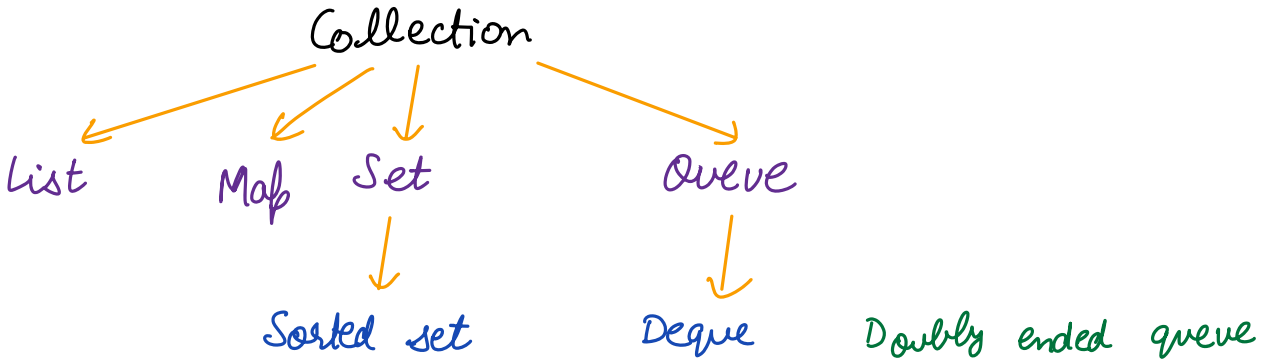
1) Java Collection
2) Collections Interface
3) Interface
4) Map Interface
5) Comparable
6) Comparator

Java Collections Framework JCF

Set of classes & interfaces that implement commonly used data structures like List, Map, Queue etc.

Advantages of JCF

- Consistent API . common set of methods to access, insert, remove etc
- Reduces programmer effort
- Increases programming speed & quality



- Collection Interface
  - Part of java.util
  - Has multiple structures like List, Set etc

add()
size()
remove()
clear()
iterator()

List Interface

- ordered collection where duplicates are allowed.
  Allows positional access (arr[i])
  - → Arraylist
  - → Vector
  - → Stack
  - → LinkedList

Arraylist

Dynamically sized array (dont need to tell size when declaring)

similar to array.

```
List<Integer> arr = new Arraylist<Integer>()
for(i=1 ; i<=5 ; i++) {
|    arr.add(i)
}
print arr.get(2)
```

arr → 1 2 3 4 5 (0 1 2 3 4)

→ 3

Vector → synchronised while arraylist is non-synchronised (will study in multithreading)

Operating Systems

```
List<Integer> arr = new Vector<Integer>()
for(i=1 ; i<=5 ; i++) {
|    arr.add(i)
}
print arr.get(2)
```

arr → 1 2 3 4 5

→ 3

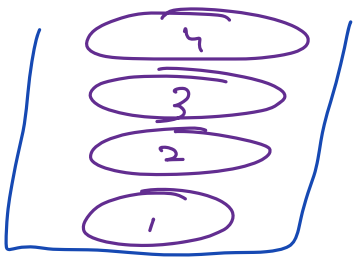
Stack → We will study later as well in Adv DSA

```
List<Integer> st = new Stack<Integer>()
for(i=1 ; i<=5 ; i++){
|     st.add(i)
}
print   st.get(2)
```

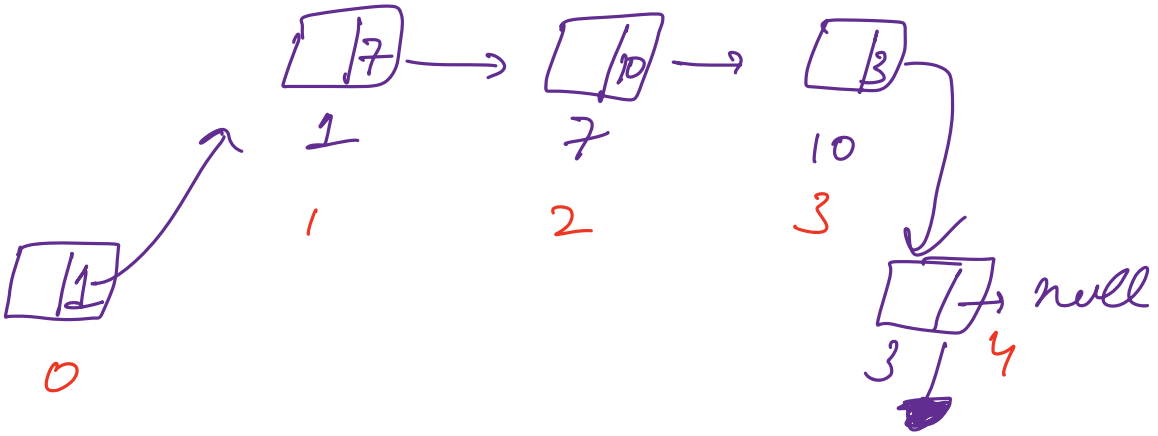
st → 1 2 3 4 5

→ 3

Actual idea ⇒



Linked List → Not stored continuously in memory.
But has address of next guy.



Set Hashset most commonly used
we have seen usages of hashset

How to iterate ?

```
Set<String> hs = new HashSet<String>()
hs.insert("A")
hs.insert("B")
hs.insert("C")
hs.insert("D")

Iterator<String> i = hs.iterator()
    while(i.hasNext()){
    |    print(i.next())
    }
```

Sorted set TreeSet (Stores in sorted order)

```
Set<String> hs = new TreeSet<String>()
hs.insert("A")
hs.insert("B")
hs.insert("C")
hs.insert("D")

Iterator<String> i = hs.iterator()
    while(i.hasNext()){
    |    print(i.next())
    }
```

A, B, C, D

Map → Hashmap
→ Sorted map

Cannot contain duplicate keys

```
Map<String, Integer> = new Hashmap<>()
map.put ("A", 10)
map.put ("B", 20)
```

If hash map → keys are in random order

If sortedmap → keys are in s order

Queue → First in first out, like a bank line

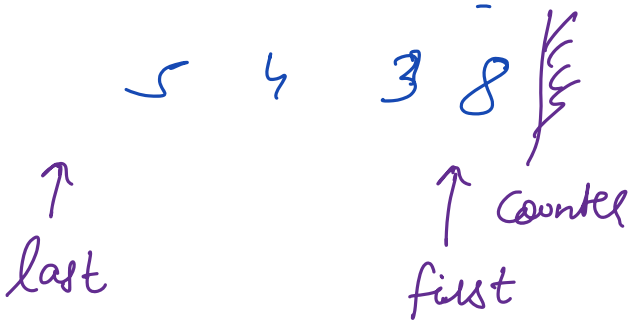
```
Queue <String> q = new PriorityQueue<>()
pq.add (A)
pq.add (B)
pq.add (C)
```

Counter

Deque → Queue but removal & insertion both can happen at front or back

Doubly ended queue

addFirst()
addLast()
removeFirst()
removeLast()



Comparables → Define ordering for a class

```
class Person implements Comparable<Person> {
    String name
    int age
    public Person(String name, int age) {
        this.name = name
        this.age = age
    }

    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age)
    }
}
```

Comparator

```
class Person implements Comparable<Person> {
    String name
    int age
    public Person (String name, int age) {
        this.name = name
        this.age = age
    }

    @Override
    public int compareTo(Person other) {
        return Integer.compare(this.age, other.age)
    }
}
```

} Comparator

Custom ordering logic

```
List<Person> arr = new ArrayList()

arr.add(new Person("Alice", 28))
arr.add(new Person("Bob", 22))
arr.add(new Person("Charlie", 25))

Collections.sort(arr)
```

new order ⟹ Bob, 22 Charlie, 25 Alice, 28

```
class Person implements Comparable<Person> {
    String name
    int age
    public Person ( String name, int age) {
        this. name = name
        this . age = age
    }

    @ Override
    public int compareTo (Person other) {
        if ( this . age ! = other . age)
            return Integer . compare ( this . age, other. age )
        else
            return String . compare ( this . name, other. name)
    }
}
```