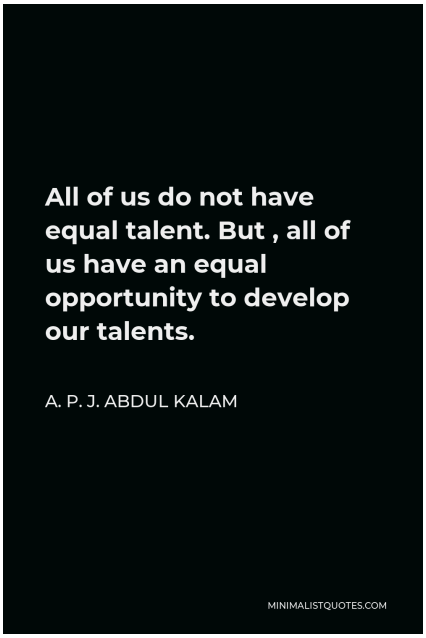
Collection



* Backtracking class

Sad or Sun

Agenda

01. Java collection framework
02. Collection Interface
03. Interfaces that extend Collection interface
04. Map interface
05. Comparable
06. Comparators

Collection → Group of individuals which are being represented as a single unit.

Framework (Library) → Set of **classes** & **interface** which provide a ready made architecture

Interface

→ declare some methods that needs to be implemented by classes.

```
Interface remote {
    turnoff();
    turnon();
    increase vol()
    decrease vol()
}
```

```
class LG implement remote {
    @Override
    turnoff() {
        ≡
    }
    turnon() {
        ≡
    }
}
```

Java collection framework

↳ set of classes & interface that implements commonly used DS

AL, Hashset, Stack, Queues

Need for Collection Framework

→ common set of method

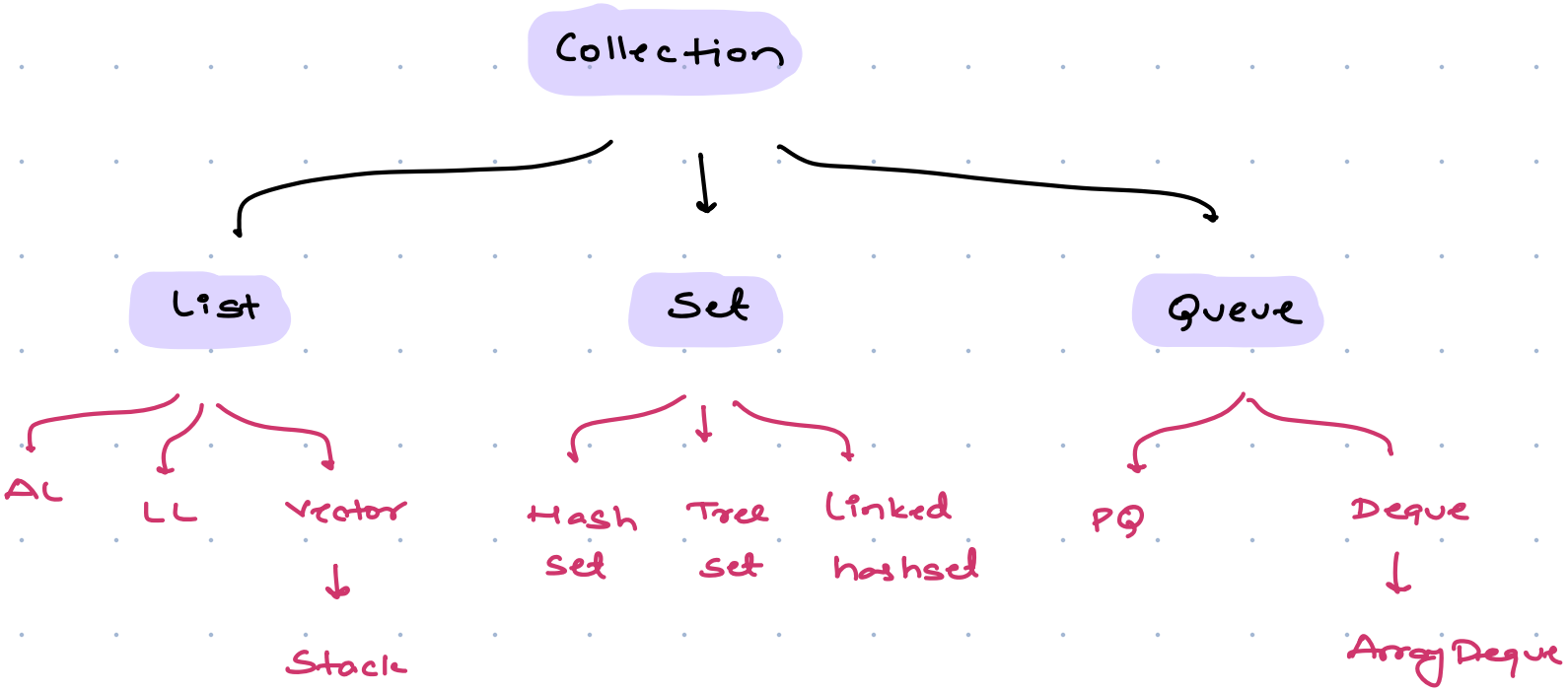
→ Method for a DS is same but implementation may differ & a programmer won't be able to remember all these functions name.

* Advantage of Collection

→ Uniformity will be maintained

→ Increase the speed & efficiency of programmer.

→ Abstraction is also achieved in frame-work



* Collection framework
  - → java util package
  - → Root interface
  - → Collection is implemented indirectly by classes.

import java.util.*;

* Methods of collection interface

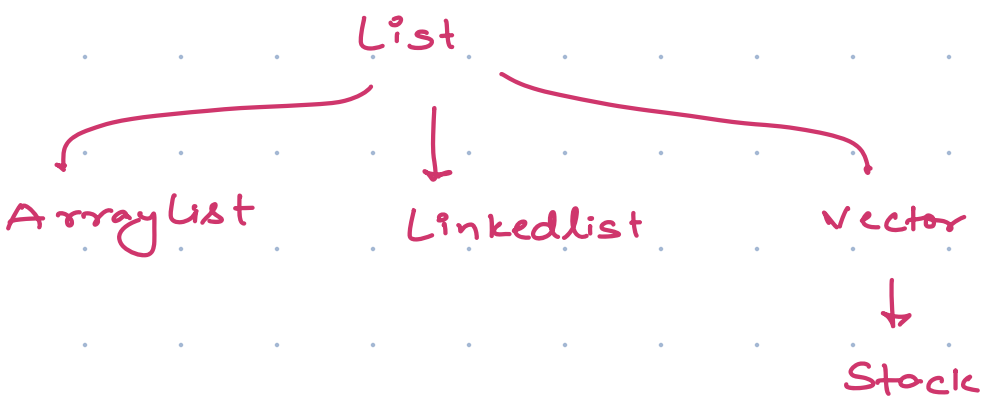
01. add()
02. size()
03. remove()
04. addAll()
05. removeAll()

* Interfaces that extend Collection framework.

1. **List :** This interface is dedicated to the data of the list type in which we can store all the ordered collection of the objects. This also allows duplicate data to be present in it.
2. **Set:** A set is an unordered collection of objects in which duplicate values cannot be stored. This collection is used when we wish to avoid the duplication of the objects and wish to store only the unique objects.
3. **SortedSet:** This interface is very similar to the set interface. The only difference is that this interface has extra methods that maintain the ordering of the elements. The sorted set interface extends the set interface and is used to handle the data which needs to be sorted.
4. **Map** : A map is a data structure that supports the key-value pair for mapping the data. This interface doesn't support duplicate keys because the same key cannot have multiple mappings, however, it allows duplicate values in different keys. A map is useful if there is data and we wish to perform operations on the basis of the key.
5. **Queue:** As the name suggests, a queue interface maintains the FIFO(First In First Out) order similar to a real-world queue line. This interface is dedicated to storing all the elements where the order of the elements matter. For example, whenever we try to book a ticket, the tickets are sold at the first come first serve basis. Therefore, the person whose request arrives first into the queue gets the ticket.

7:40 → 7:50 AM

Will finish this class first



ArrayList

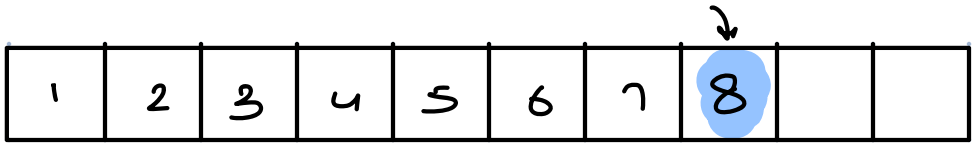
→ Resizeable array implementation

→ Index based accessing

→ Dynamic memory Alocation

→ Insert & delete at specific index

→ AL implementation is not synchronised

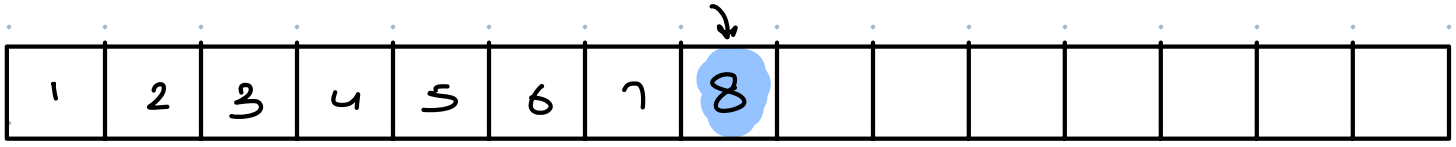


n=10

load factor = $\frac{\text{No. of ele}}{\text{Total size}}$

≒ 0.7

Create a new AL of size = double the prev



* Vector

→ Dynamic Array

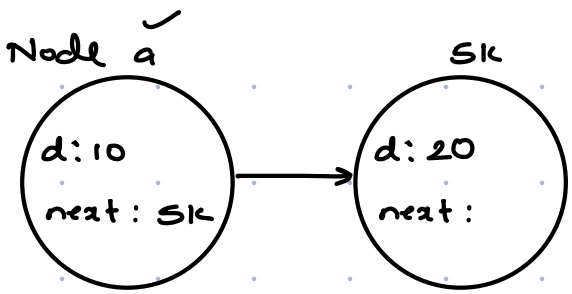
→ Vector is synchronised & AL is not synchronised

Stack → { Last In First Out }

push()
pop()
} vector class

isEmpty()
peek()
search()
} stack class

Linked List




```
class Node {
    int data
    Node next
}
```

Node b = new Node(20)

a.next = b;

→ Random memory Alocation
→ Insertion & deletion are very fast
→ Stores the data in terms of nodes

* Set Interface

→ Ordering of elements depends upon the class implementing it.

→ No duplicates

→ Dynamic memory Alocation

01. Hashset (hashing)

→ Data is not organised in the manner in which it is inserted.

→ Time complexity for all actions → O(1)

02 Linked Hashset

→ Ordered version of Hashset

→ Order of insertion will be maintained. Behind the scene, Linked Hashset uses a doubly linked list

→ TC of operations in Linked Hashset → O(1)

* Tree Set (Red black tree)

→ Ordering of elements is maintained

→ Ascending order of keys

Insertion / Deletion → O(log n)

* Queue Interface

→ Hold the elements in FIFO order

↓

First In First Out

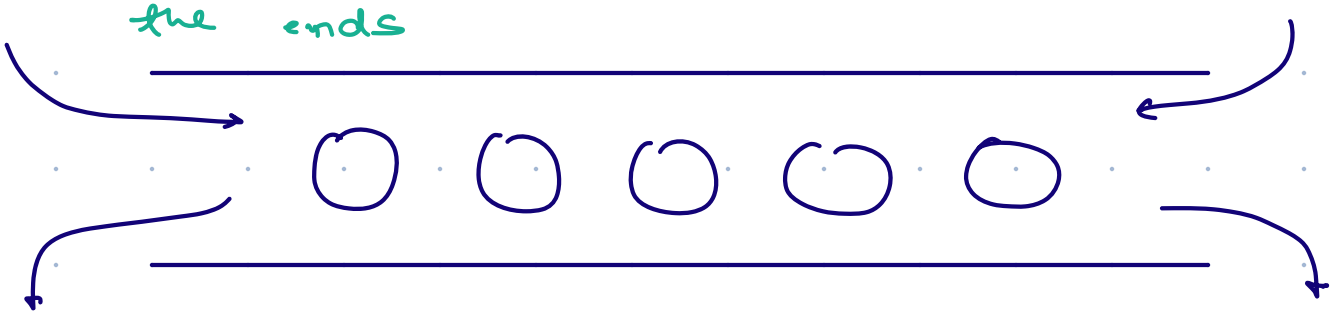
```
Queue<I> q = new ArrayDeque();

Queue<I> q = new PriorityQueue();

Queue<I> q = new LinkedList();
```

PriorityQueue says that element are not going to follow FIFO order, rather than that they will follow the priority order that user will set

Deque → Doubly Ended Queue

(Sliding window maximum)

→ Insertion & Deletions are allowed from both the ends



* Comparable Interface

→ provides natural ordering for a class

→ Compares two object of class with each other

Eg!- AL<I> ans = new AL<>();

ans = {10, 3, 7, 20, 4}

Collections.sort(ans): → {3, 4, 7, 10, 20}

```
AL<I> al = new AL<>();

al = {"abc", "yogi", "joshy"};
```

Collections.sort(al) : ⟶ {"abc", "joshy", "yogi"}

```
class Person
{
    String name;
    int age;
    Person(String n, int a){
        this.name = n;
        this.age = a;
    }
}
```

```
Person P1 = new person("Yogi", 27);
Person P2 = new Person("Indu", 20);
Person P3 = new Person("Abhi", 23);

AL<Person> al = new AL<>();

al.add(P1)
al.add(P2)
al.add(P3)

al = {P1, P2, P3}
```

Collections.sort(al) → Error

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

    Public int compareTo(Person other){
        if (this.age < other.age) return -1;
        else if (this.age > other.age) return 1;
        else return 0;
    }
}
```

Collections.sort(al) → {P2, P3, P1}

Compare two strings = str1.compare(str2)

* Comparator Interface → Custom ordering

```
class Person implements Comparable<Person>{
    string name;
    int age;
    Person(string n, int a){
        this.name = n;
        this.age = a;
    }

    Public int compareTo(Person other){
        if (this.age < other.age) return -1;
        else if (this.age > other.age) return 1;
        else return 0;
    }
}
```

```
Person P1 = new person("Yogi", 20);
Person P2 = new Person("Indu", 20);
Person P3 = new Person("Abhi", 23);
```

```
AL<Person> al = new AL<>();

al.add(P1)

al.add(P2)

al.add(P3)
```

al = { P1, P2, P3 }

Q I want to order these people based on the descending order of Age.

```
Collections.sort(al, new AgeDesc());
```

```
public class AgeDesc implements comparator<>{

    @Override
    public int compare(Person P1, Person P2){

        if (P1.age < P2.age) return 1;

        else (P1.age > P2.age) return -1;

        else {
              P1.name.compare(P2.name)
        }
    }
}
```

al = { P3, P2, P1 };