Collections: Framework

* Collections are growable nature
* Whenever you add the data its size will automatically increases.
* This collections will in java.util package
* Its stores the data
* These will works with objects
* If you use primitive data types need to use wrapper classes

**List**🡺 Interface

1. Duplicate data is allowed.
2. Null values allowed.
3. Insertion order.

* **ArrayList** 🡺 implementation class

1. Whenever your operations are **for fetching the data,** arraylist is best choice.
2. Internally it uses ***RandomAccess*** Interface.
3. Whenever you have adding and deleting operations in the middle of the list arraylist is worst choice.

1,2,3,4,**5**,6,12,11,7,8,9,10……….**100000**

**A[4] == 5 🡪 1sec**

**A[99999] == 100000 🡪 1sec**

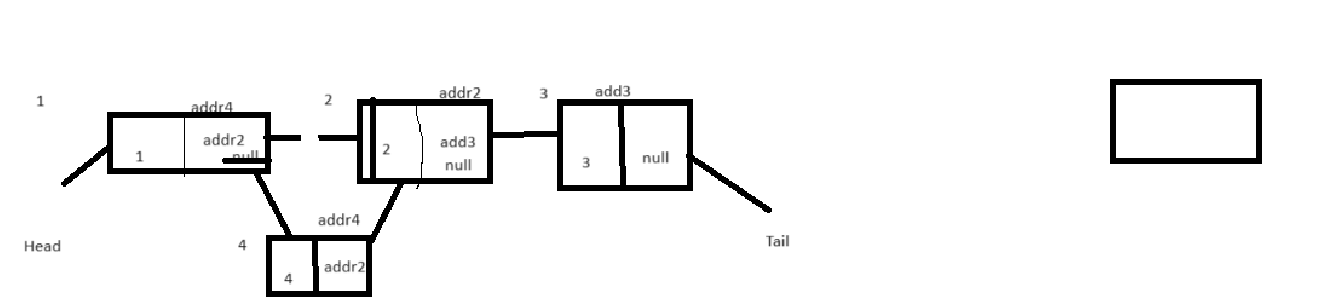
**Int a[] = new int[5];**

1. By default size is 10.
2. Asynchronous.
3. Load factor/Threshold limit 🡺 75% or 0.75

New capacity = Current capacity \*3/2+1 🡺 10\*3/2+1 🡺 16

* **LinkedList** 🡺 implementation class

1. Whenever your operations are adding or deleting the data in the middle of the list it is the best choice.
2. Whenever you want to fetch the data linkedlist worst choice.
3. LInkedlist data is of nodes 🡺 nodes has data and address of next node.
4. 1st node is HEAD and last node TAIL
5. If there only 1 node then address will be null and last node address will be null



1 2 3

6

4 5

* **Vector** 🡺 implantation class

1. Synchronous
2. Whenever your operations are **for fetching the data**.
3. Internally it uses ***RandomAccess*** Interface.
4. Load factor/Threshold limit 🡺 75% or 0.75
5. By default size is 10.

New capacity = 2\* current capacity.

**Set** 🡺 Interface

* Duplicates are not allowed.
* Whenever your tasks is for Search operations.
* Unordered list.
* **HashSet (**implantation class**)**: random order data
* Only 1 null value is allowed.
* **TreeSet (**implantation class**)**: if you need data to be sorted/ascending then treeset is used
* No null value allowed.
* **LinkedHashSet (**implantation class**)**: insertion order data will be printed
* **Default size 10.**
* **Asynchronous.**

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

Set.add(1);

**Map** 🡺 Interface

* Data will be in Key, value pair.

Map<**Key**,**value**> map = new HashMap<>();

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

map.put(1,”ABC”); 🡺 hashcode is calculated for key 1 🡺 2349873 🡺 from this hashcode we will calculate index position 🡺 2349873 % 16(size of hashmap) 🡺 0-15 🡺 suppose output is 1

**hashcode**

**key**

**value**

**address of next node**

map.put(2,”BGC”); 🡺 hashcode calc 🡺2657860 🡺 2657860 % 16 🡺 4

map.put(3,”YUY”) 🡺 63281940 🡺 4 index

map.get(3) 🡺 63281940 hashcode 🡺 index value 4 🡺 YUY

* Keys will be unique (no duplicates are allowed)
* Values can be duplicate.
* Values can be null.
* Its default size 16 (0-15 indexes)
* Each index value will be called as bucket.
* Each index value is internally a linkedlist.
* **HashMap**: random order data will be inserted
* Keys can have only 1 null value.
* **TreeMap**: sorting order
* **Null keys cannot be used**
* **LinkedHashMap**: insertion order
* Keys can have only 1 null value.
* **Asynchronous**

2 Threads

1 thread getdata

1 thread Delete operation

Exception: **concurrentModificationException 🡺 Fail fast iterators**

**HashTable**: 🡺 hashmap 🡺 concurrentHashmap

It is synchronous

It wont allow duplicate keys

It wont allow null keys and null values

Performance will be slow

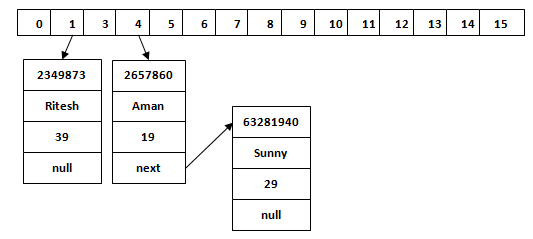
**ConcurrentHashMap**: 🡺 **Fail Safe iterator**

Add/delete/update: Synchronous calls

Get operations: Asynchronous

16 size.

No null keys and values allow



**CopyOnWriteArrayList**: it wont cause any **concurrentModificationException** when we iterate the list data.

**CopyOnWriteArraySet**: it wont cause any **concurrentModificationException** when we iterate the set data.

Fail fast and fail safe iterators:

Fail fast : hashmap

* The Fail Fast iterator throws a ConcurrentModificationException if a collection is modified while iterating over it.
* The Fail Fast iterator uses an original collection to traverse over the collection's elements.
* They are memory savers, don't require extra memory.
* The Fail Fast iterators returned by ArrayList, HashMap, Vector classes.

Fail safe: concurrentHashMap

Stack 🡺 LIFO 🡺 last in first out 🡺

Push 🡺 add

Pop 🡺 remove

Queue 🡺 FIFO 🡺 first in first out