

Before the concept of arrays was introduced, programmers used other data structures or manual techniques to store multiple elements.

-Most common approach:

Using Separate Variables: Programmers would declare individual variables for each element they wanted to store.

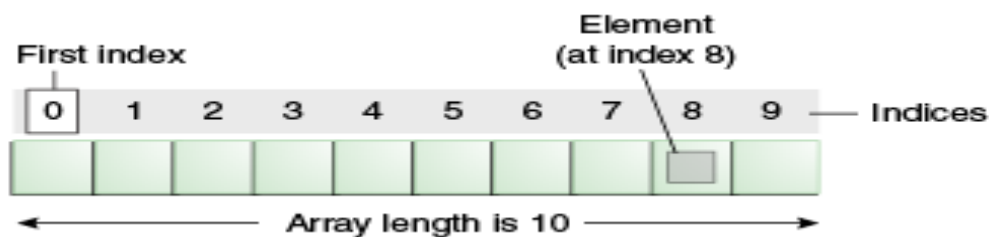
For example, if they wanted to store five integers, they would create ten variables like this:

```
int num1=10;  
int num2=20;  
int num3=30;  
int num4 = 40;  
int num5 = 50;
```

This method becomes cumbersome, impractical and error-prone when dealing with a large number of elements.

Arrays:

- An *array* is a container object that holds a fixed number of values of a single type.
- The length of an array is established when the array is created.
- After creation, its length is fixed.
- Each item in an array is called an *element*.
- and each element is accessed by its numerical *index*.



- About arrays:

1- Declaration and Initialization:

- To create an array, you first declare its type and then initialize it.
- The syntax for declaring an array is:

dataType[] arrayName;

Ex: int [] numbers;

- Initialize the array with a specific size, you use the new keyword:
numbers = new int[9]; // Creates an array of size 9 to store integers.

- You can initialize an array with values at the time of declaration, using curly braces:

Int[] numbers = {10, 20, 30, 40, 50};

2- Indexing:

- Array elements are accessed using zero-based indexing.
- The first element is at index 0, the second at index 1, and so on.
- To access an element, you use square brackets with the index inside:
int element = numbers[2]; // Accesses the third element (index 2) of the array

3- Length:

Every array has a length property, which represents the number of elements it can store. You can retrieve the length using **arrayName.length**:
int size = numbers.length; // Retrieves the size of the 'numbers' array

4-iterating:

Java offers a convenient way to iterate through the elements of an array

```
for (int i = 0; i < arrayName.length; i++) {  
    // Do something with arrayName[i]  
}
```

```
// Enhanced for-loop (for-each loop)  
for (dataType element : arrayName) {  
    // Do something with 'element'  
}
```

5- Operations:

storing a collection of data, searching, deleting, and more.

Arrays - Memory Management:

- The elements in an array are stored in contiguous memory locations, making it efficient for accessing and manipulating elements.

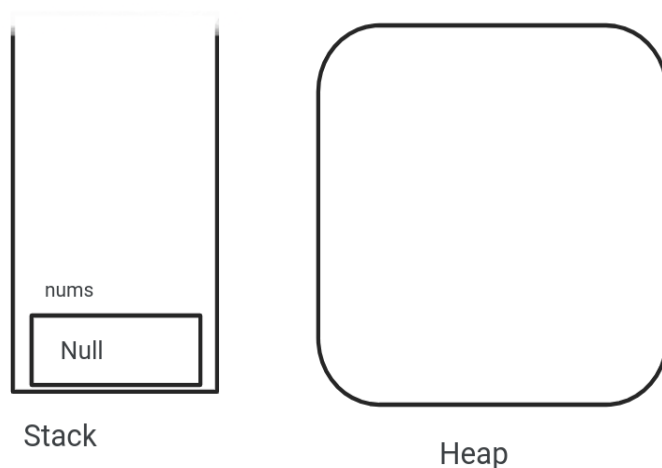
- This means that each element of the array is stored in a contiguous block of memory, with the first element stored at the lowest memory address and the last element stored at the highest memory address.

- This allows for efficient access to the elements of the array, as it is possible to calculate the memory address of a particular element using a simple mathematical formula based on the array's starting address and the index of the element.

-As we said before the memory is managed two areas (Stack , Heap)

```
Int [] nums;
```

This will create a variable in the stack and has value null by default.

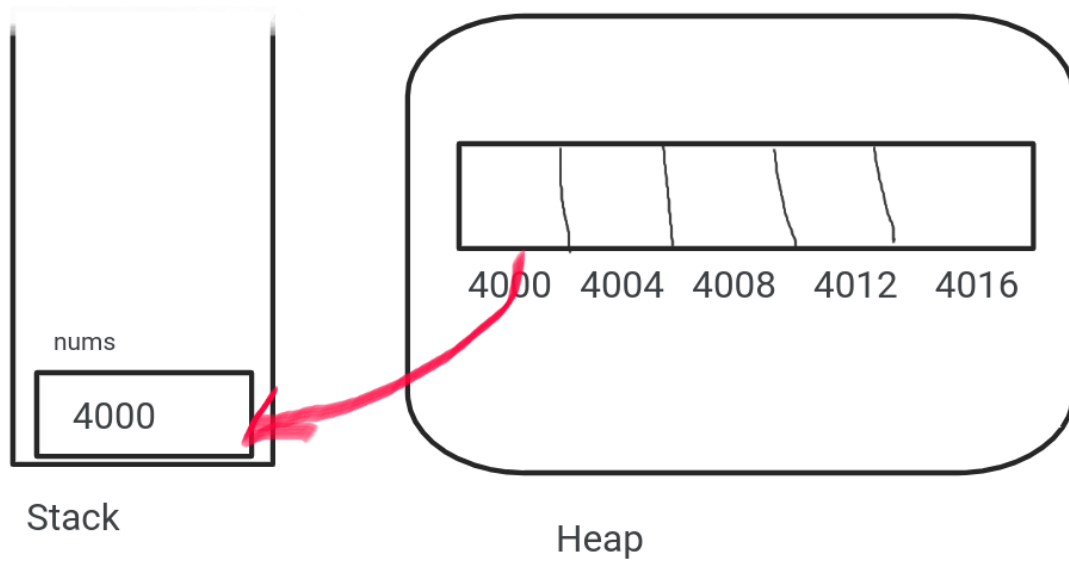


Ok, But Now what happen when we allocate memory:

```
nums = new int[5] // allocates memory for 5 integers
```

This will allocate memory for 5 integers, Each one has its own address.

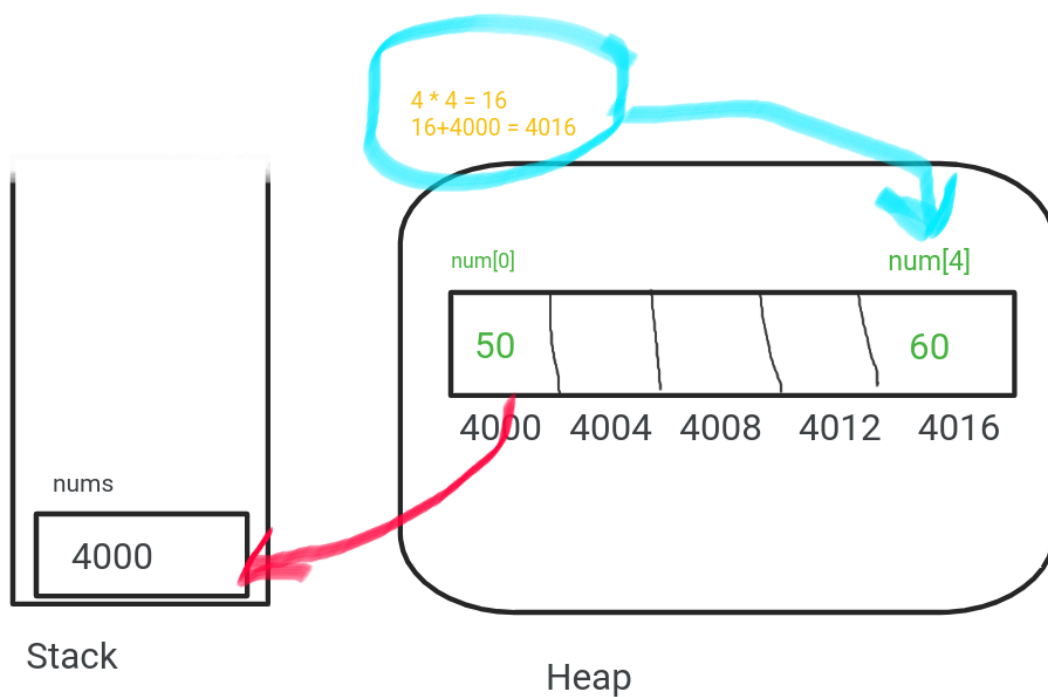
As we said above these allocated memory units are contiguous, hence they will be spaced out by the same distance.



Now, what will happen if we store values by index.

`nums[0] = 50; // $0 * 4 = 0 \Rightarrow 0 + 4000 = 4000$`

`num[4] = 60; // $4 * 4 = 16 \Rightarrow 16 + 4000 = 4016$`



- **Efficiency:** Arrays provide efficient element access because the elements are stored contiguously in memory.

-accessing an element at a specific index is a constant-time operation ($O(1)$).

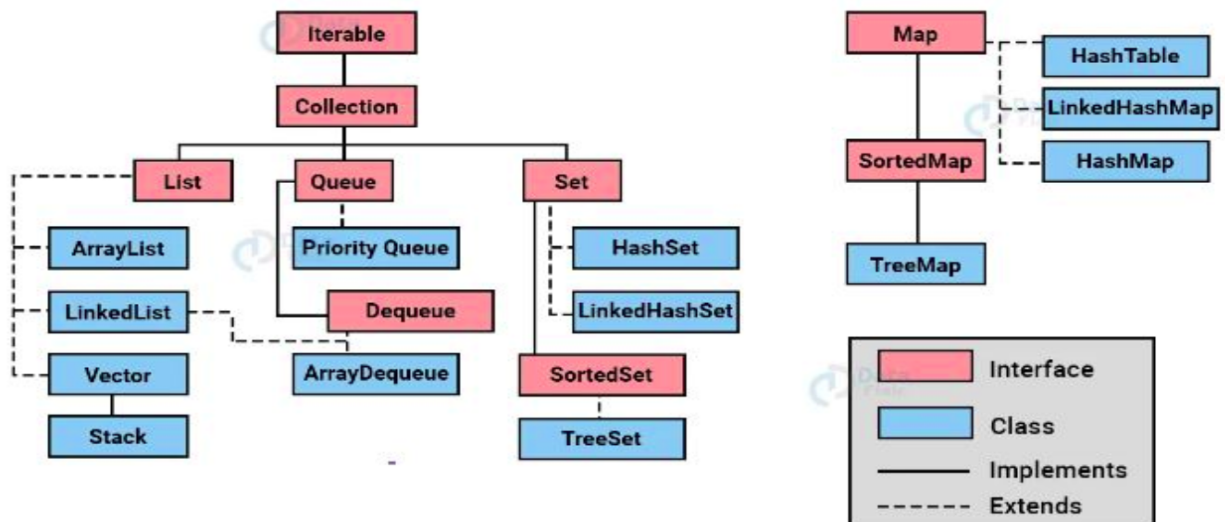
However, insertion and deletion of elements might be less efficient, especially if done frequently, as it requires shifting the elements in the array.

Note*: keep in mind that their fixed size might limit their flexibility in certain situations.

- Is there something that offers more flexibility?!

Collections:

collections refer to a framework that provides a set of classes and interfaces to store, manage, and manipulate groups of objects. The Java Collections Framework includes various data structures, such as lists, sets, maps, queues, and more. These collections efficiently store, retrieve, and process data, making it easier to work with aggregate data and perform common operations like searching, sorting, and iteration.



List Interface:

- The `List` interface represents an ordered collection of elements that allows duplicates.
- The `List` interface extends the `Collection` interface and adds methods to manipulate and access elements based on their index.
- Key features include ordered collection, allowing duplicates, and dynamic size changes.



ArrayList:

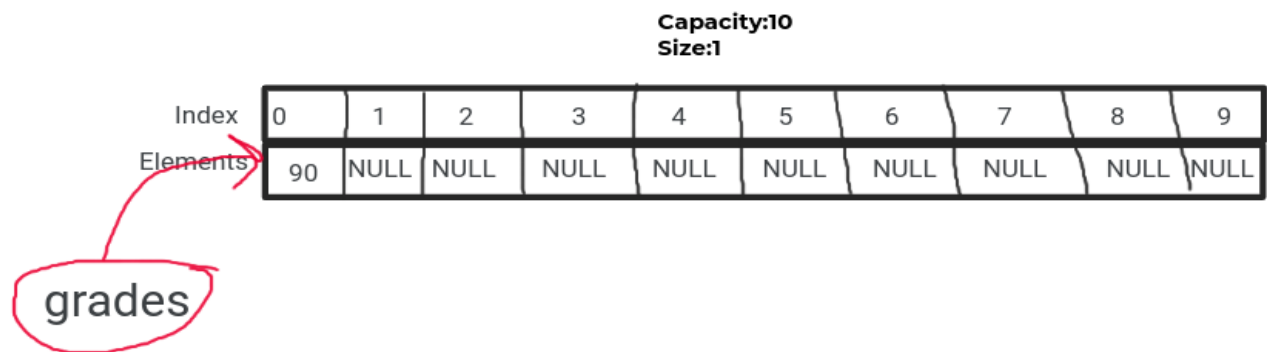
- The `ArrayList` class implements the `List` interface.
- Can contain duplicate elements.
- maintains the insertion order.
- Non-synchronized. (not-thread safe).
- allow random access because the array works at index basis.
- Unlike arrays, arraylists can automatically adjust their capacity when we add or remove elements from them. (dynamic arrays).

in Java 8: to save memory consumption and avoid immediate memory allocation make The default capacity of an empty `ArrayList` is 0 and not 10. Once the first item is added, the `DEFAULT_CAPACITY` which is 10 is then used.

Let's See what Happen when create Array List and add first element:

```
ArrayList<Integers> grades = new ArrayList<>();
```

```
grades.add(90);
```

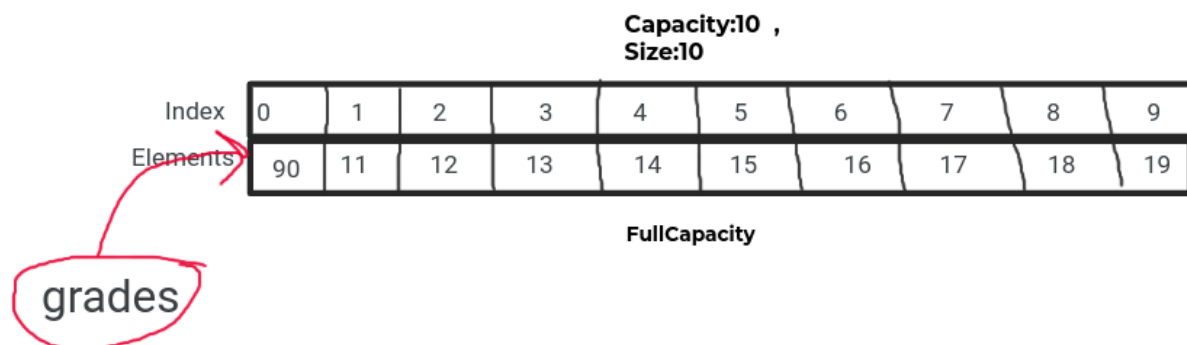


-now we will add another 9 elements :

```
for (int i=1; i<10; i++) {
```

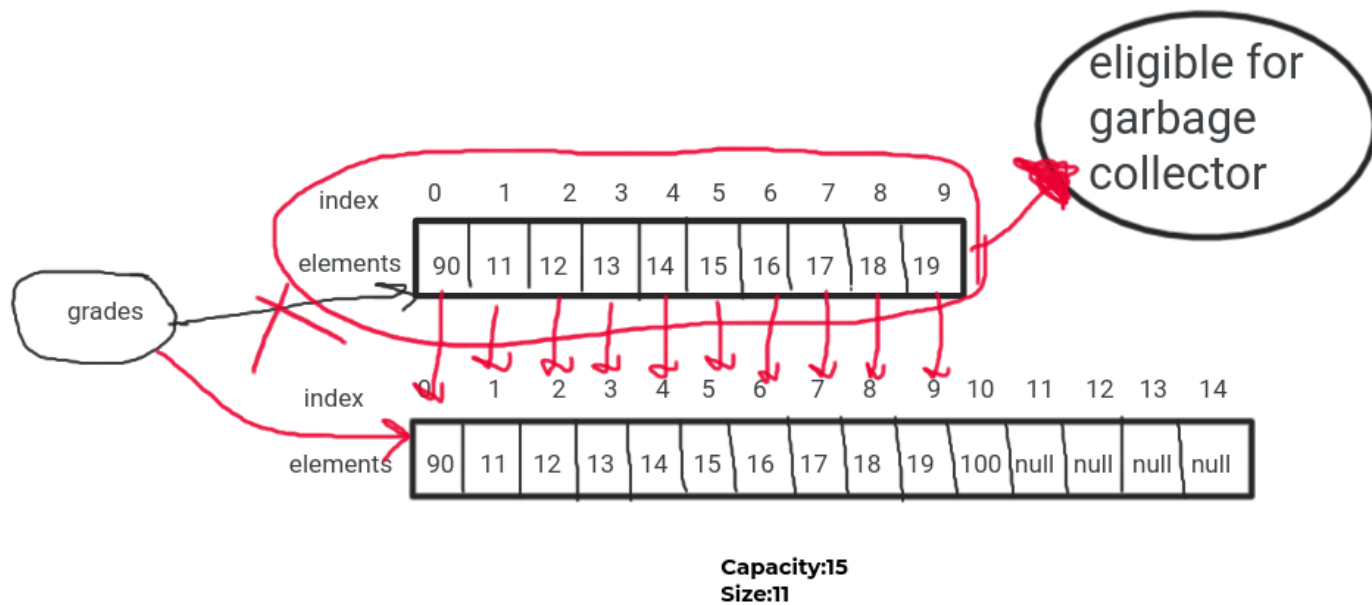
```
    grades.add(i+10);
```

```
}
```



What happens when you want to add the 11th element? How does ArrayList become resizable?

```
grades.add(10, 100);
```



- The efficiency of common ArrayList operations:

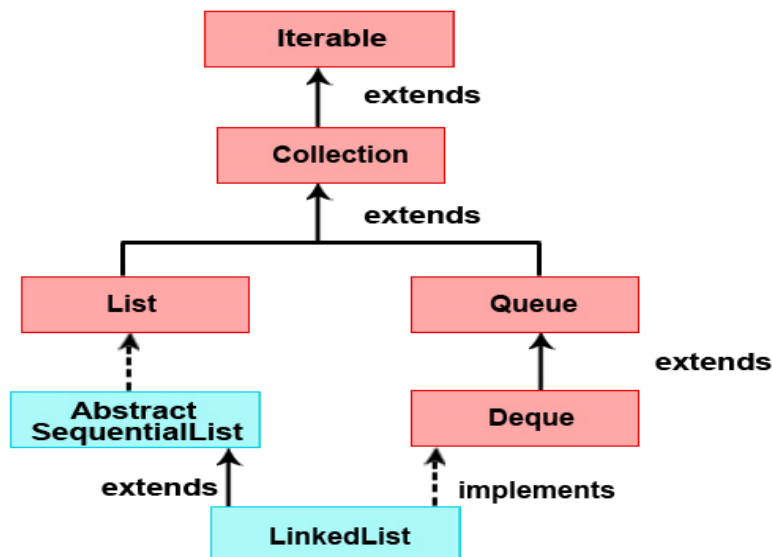
Retrieval: Getting a specific element in the ArrayList is extremely fast with the `ArrayList.get(i)` method, where *i* is the element's index.

Addition: Adding a new element to the ArrayList is usually extremely fast, as long as you are adding to the end and there is still room in the array. If the array is full, adding a new element requires more time because the array's size first needs to be expanded.

Deletion: Deleting an element from the ArrayList is relatively slow, even though the actual deletion can be done quickly. This is because all elements after the deleted element need to be shifted one place to the left to fill the hole in the array left by the deletion.

LinkedList:

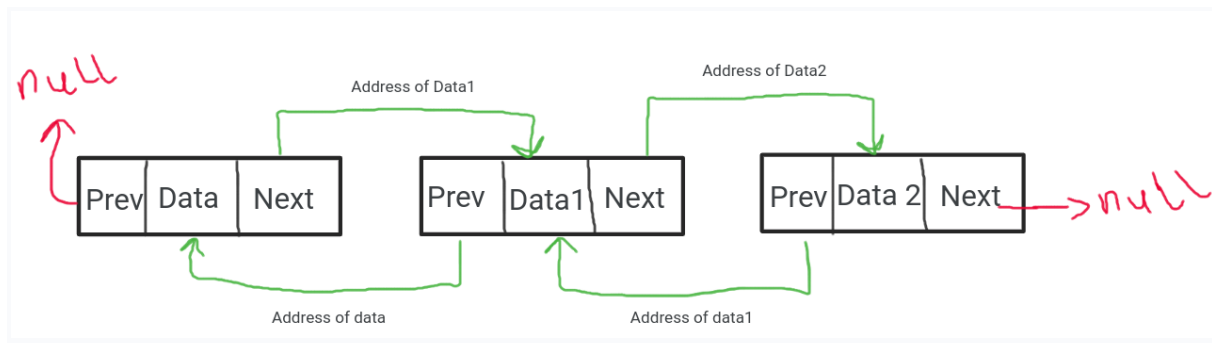
The `LinkedList` class provides the functionality of the linked list data structure (use a doubly linked list to store the elements).



- contain duplicate elements.
- maintains insertion order.
- non synchronized.
- manipulation is fast because no shifting needs to occur.
- can be used as a list, stack or queue.

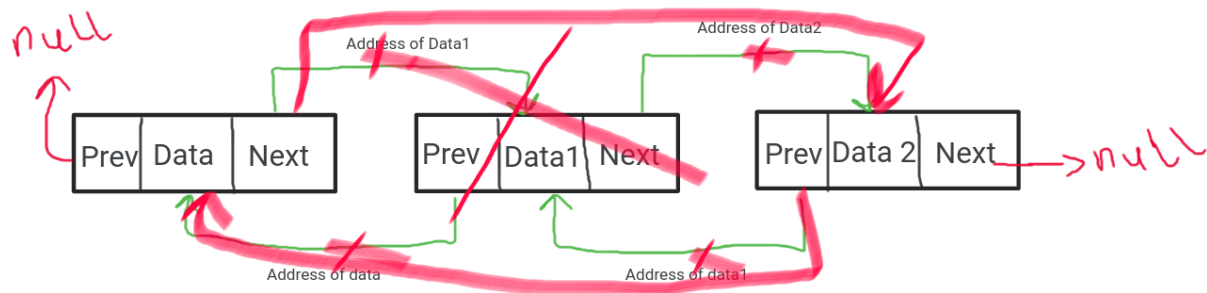
Each element in a linked list is known as a node. It consists of 3 fields:

- Prev - stores an address of the previous element in the list. It is `null` for the first element
- Next - stores an address of the next element in the list. It is `null` for the last element
- Data - stores the actual data



When we remove Data1:

The Next pointer of data will be pointed to data2 Address and the prev pointer of data 2 will point address to Data.



Queue(FIFO) First Input First output:

Enqueue -> add element to the end of the list.

Dequeue -> retrieve and remove the first element of the list

Stack (LIFO) Last input first output:

Push() Append the specified element to the end of the list

pop() retrieve and remove the last element from this List.

-efficiency:

Retrieval: LinkedLists do not have an easy way to retrieve arbitrary elements. In the worst case, getting a specific element in the LinkedList therefore requires you to traverse the entire list to find the element you're looking for.

Addition: Adding a new element to the LinkedList can be done in fast constant time if you are appending to the front or end of the list. If you want to insert a new element in an arbitrary position, you'll need to spend more time traversing the list.

Deletion: Deleting an element from the LinkedList, like adding one, is fastest when deleting from the list's front or end. However, deletion is slower when removing an arbitrary element.

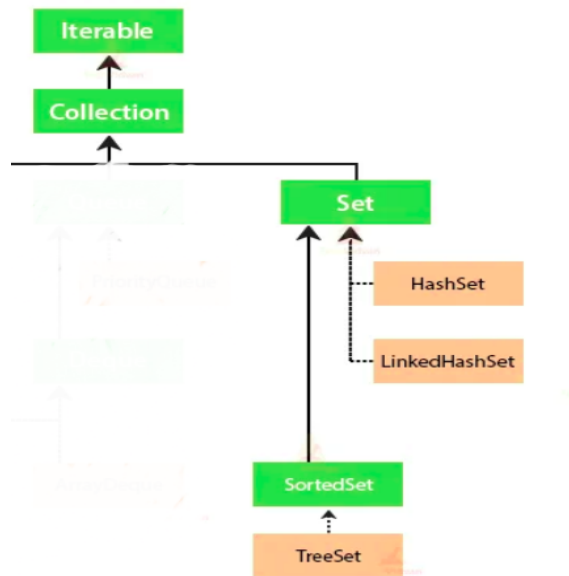
use cases for ArrayLists and LinkedLists:

ArrayLists are best for cases where you will be retrieving elements (that is, reading from the array) more frequently than modifying the array.

LinkedLists are best for cases where you will be modifying the list often, especially at the front or end of the list.

Set:

Set contains unique elements only.



HashSet: (Use HashTable for Storage)

- Store Elements by using a mechanism called hashing.
- Doesn't maintain Insertion Order. Elements are inserted on the basis of their hashCode.

The hashCode of an element is a unique identity that helps to identify the element in a hash table.

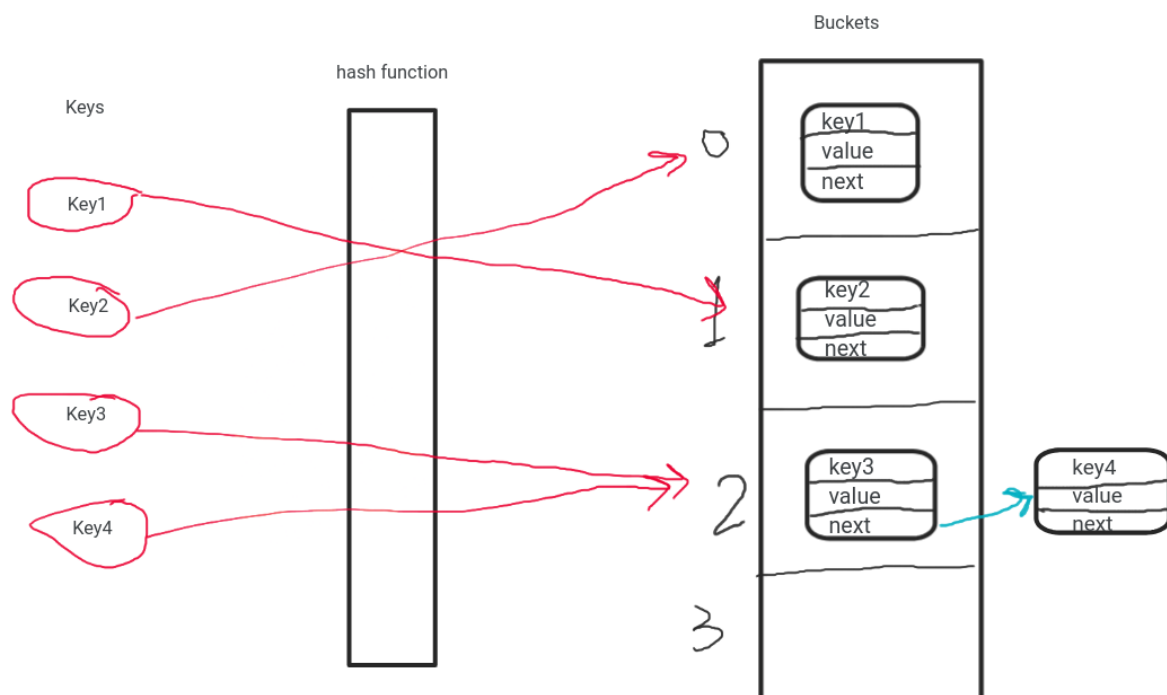
- Non Synchronized
- Contains unique elements only.
- HashSet is the best approach for search operation.

How Does HashSet Work Internally?

- HashSet uses HashMap internally to store data in the form of Key-Value pair.
- Elements are stored in the form of key-value pair.
- Whenever you create a HashSet object, one HashMap object associated with it is also created.
- where key will be the actual element value and value will be the ***PRESENT*** Constant.

-HashSet has default initial capacity of 16.

-HashSet has default load factor of 0.75 or 75%.



LinkedHashSet:

- provides functionalities of both the hashtable and the linked list data structure.
- Elements are stored in hash tables like HashSet.
- When you create a LinkedHashSet in Java, it internally uses a combination of a hash table and a doubly-linked list to store its elements

Hash Table: LinkedHashSet uses a hash table to quickly access elements based on their hash codes, enabling constant-time ($O(1)$) operations for adding, removing, and existence checks. Each element is placed in a specific bucket within the hash table, determined by its hash code.

Doubly-Linked List: Each element in the set is represented as a node in this linked list. The doubly-linked list allows the set to maintain the insertion order of elements.

When adding an element:

- It is first hashed to find its bucket location in the hash table.
- If the bucket is empty, the element is added to the bucket, and a new node is created in the doubly-linked list to maintain insertion order.
- If the bucket is not empty, the set checks if the element already exists using its hash code.
- If the element is not found, it is added to the bucket, and a new node is created in the doubly-linked list to maintain insertion order.

This approach ensures that `LinkedHashSet` maintains the order in which elements were inserted while preventing duplicates based on their hash codes.

The use of the doubly-linked list provides predictable iteration order, meaning elements are returned in the same order as they were inserted.

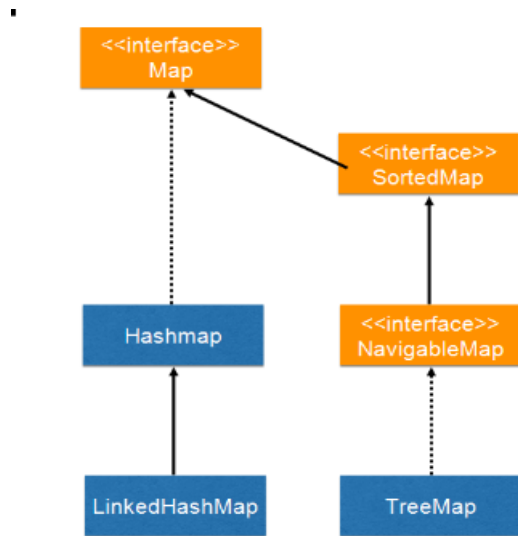
However, due to the additional overhead of maintaining the linked list, `LinkedHashSet` may be slightly slower in terms of memory and insertion/deletion performance compared to a regular `HashSet`.

Nevertheless, if you require a set with preserved insertion order and efficient access to elements, `LinkedHashSet` is a good choice.


TreeSet:

- The object of the `treeSet` Class are sorted in ascending order.
- Contains Unique Elements Only.
- We can customize the sorting of elements by using the `Comparator` interface or `comparable`.

Map:



UseCases:

- When working with a large collection where frequent insertion and deletion operations are expected, and random access is not a primary concern, what is the best choice ?
 - Suppose you are building an inventory management system for an online store. You need to keep track of the available stock of products. Each product has a unique identifier (product ID) and a corresponding quantity in stock. Which Collection can we use to solve this?
- 
- Suppose you are developing a user activity tracking system where you need to maintain the order of user actions. You want to store the actions in a collection while ensuring uniqueness?
 - In a social media application, there are millions of user posts generated every day. How can Java Collections help you handle and display the latest posts efficiently? Which collection would you use to maintain a sorted order of posts based on their timestamps?
 - You are developing a game, and you want to keep track of the top 10 players' high scores. Which Java Collection would be most appropriate for maintaining this leaderboard?
 - You are building a movie recommendation system. How would you store the movie data and user preferences using Java Collections? How would you efficiently retrieve and suggest movie recommendations based on a user's previous choices?