LinkedList

eazy bytes

Java LinkedList is a data structure in Java that represents a list of elements, where each element is connected to its neighboring elements using pointers. It's a linear data structure that can dynamically increase or decrease in size during runtime. The LinkedList class in Java is part of the java.util package. Here's an example of how to create a LinkedList in Java:

```
LinkedList<String> list = new LinkedList<>();
```

Few important points about LinkedList:

- We already used arrays and their dynamic cousin, the ArrayList class. However, arrays and array lists suffer from a major drawback. Removing an element from the middle of an array is expensive since all array elements beyond the removed one must be moved toward the beginning of the array. The same is true for inserting elements in the middle.
- Just like ArrayList, LinkedList class can also contain duplicate elements and maintains insertion order.
- LinkedList in Java is a very useful data structure for scenarios where you need to perform frequent insertions or deletions of elements in a list, as it doesn't require the same amount of memory allocation or reallocation as an array-based list.
- To create an LinkedList of primitive types like int, float, and char in Java, we cannot use the primitive types directly. Instead, we need to use the corresponding wrapper class. For instance:

```
LinkedList<int> intNums = LinkedList<>(); // Compilation fails
LinkedList<Integer> intNums = new LinkedList<>(); // works fine
```

LinkedList

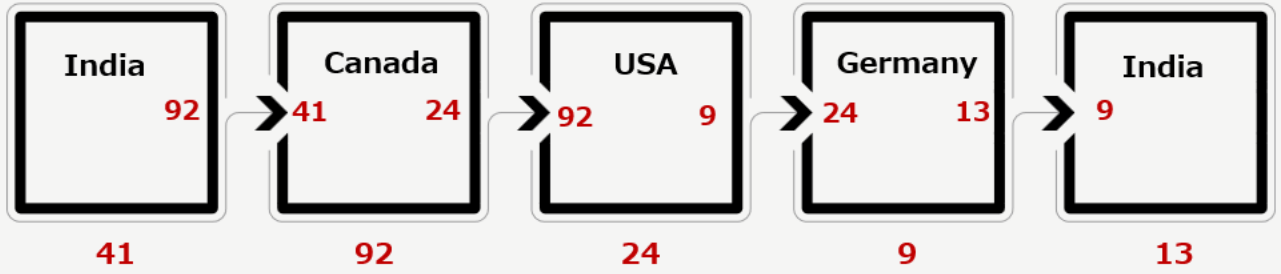
eazy bytes

Creating LinkedList

```
List<Integer> integerNums = new LinkedList<>();
List<Double> doubleNums = new LinkedList<>();
var countryNames = new LinkedList<String>();
List<Character> characters = new LinkedList<>();
LinkedList<Person> persons = new LinkedList<>();
```

Let's see how LinkedList works by using the below code,

```
List<String> countryNames = new LinkedList<>();
countryNames.add("India");
countryNames.add("Canada");
countryNames.add("USA");
countryNames.add("Germany");
countryNames.add("India");
```



LinkedList stores its elements using nodes. Each node in the LinkedList contains two parts: the data part which stores the element itself and a reference part (also called a "pointer") that points to the next & previous nodes in the list.

The first node in the list is called the head of the LinkedList and the last node is called the tail. The head node is the entry point to the LinkedList, and it doesn't have a previous node, while the tail node doesn't have a next node.

Iterating & Sorting of the LinkedList works same as ArrayList. We can use Iterator/ListIterator/for-each for looping and Comparable/Comparator for sorting.

LinkedList

eazy bytes

Important methods while working with LinkedList

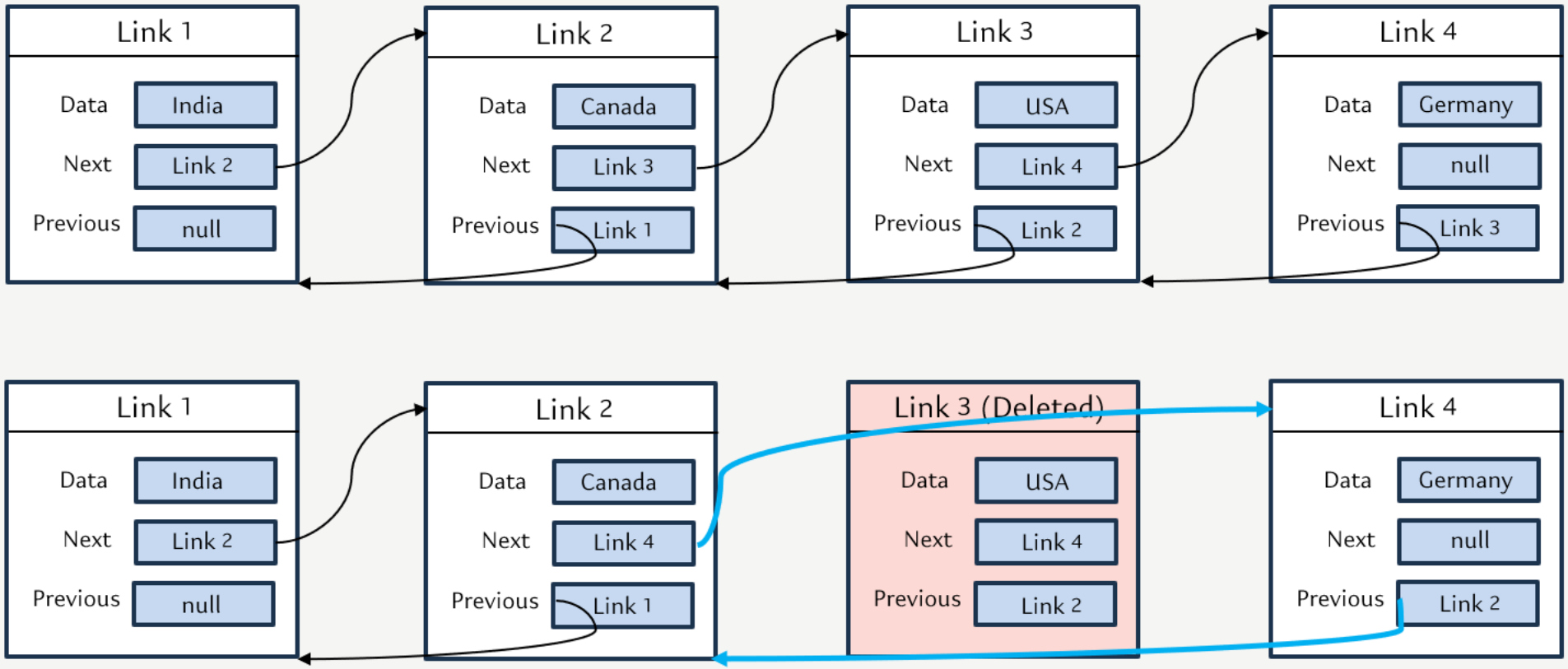
add(element): adds an element to the end of the LinkedList
add(index, element): inserts an element at the specified index in the LinkedList
get(index): returns the element at the specified index in the LinkedList
remove(index): removes the element at the specified index in the LinkedList
size(): returns the number of elements in the LinkedList
addFirst(E e): It is used to insert the given element at the beginning of a list.
void addLast(E e): It is used to append the given element to the end of a list.
E getFirst(): It is used to return the first element in a list.
E getLast(): It is used to return the last element in a list.
E peek(): It retrieves the first element of a list
E peekFirst(): It retrieves the first element of a list or returns null if a list is empty.
E peekLast(): It retrieves the last element of a list or returns null if a list is empty.
E poll(): It retrieves and removes the first element of a list.
E pollFirst(): It retrieves and removes the first element of a list, or returns null if a list is empty.
E pollLast(): It retrieves and removes the last element of a list, or returns null if a list is empty.
E pop(): It pops an element from the stack represented by a list.
void push(E e): It pushes an element onto the stack represented by a list.

LinkedList performance

eazy bytes

An array stores object references in consecutive memory locations, a linked list stores each object in a separate link/node. Each link/node also stores a reference to the next link in the sequence. In the Java programming language, all linked lists are actually doubly linked; that is, each link also stores a reference to its predecessor.

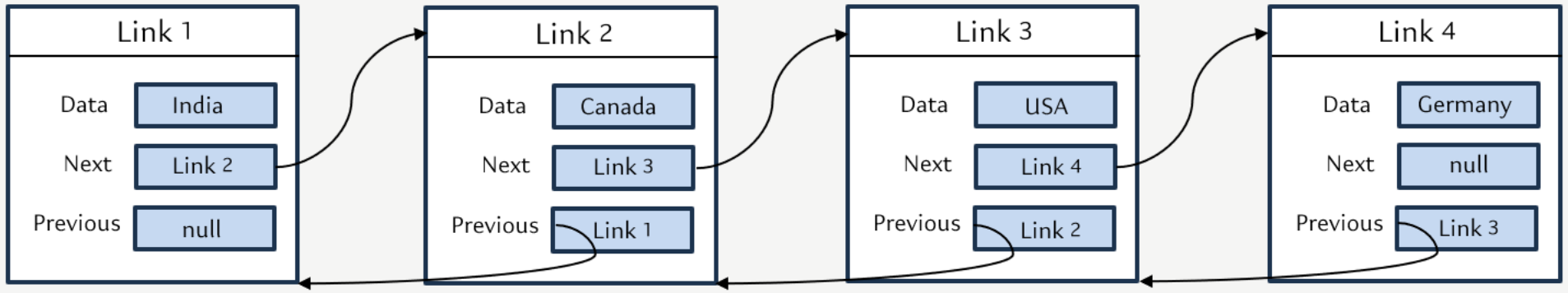
Removing an element or adding an element from the middle of a linked list is an inexpensive operation—only the links around the element to be removed need to be updated. Below is the illustration of the same,



LinkedList performance

eazy bytes

Accessing an random element with LinkedList is going to be expensive as they are not stored sequentially. Let's imagine you want to access the last element inside your LinkedList. Here is the flow of steps followed,



STEP 1: In the first element, details of the next link will be checked

STEP 2: In the second element, details of the next link will be checked

STEP 3: In the third element, details of the next link will be checked

STEP 4: In the fourth element, details of the data will be checked

ArrayList vs LinkedList

eazy bytes

ArrayList is an implementation of the List interface that stores elements in a contiguous block of memory, which makes accessing elements faster than LinkedList in most scenarios.

ArrayList has constant-time O(1) access time to an element given its index, which makes it a good choice when you need to access elements frequently or in a random order.

ArrayList can be slower than LinkedList when it comes to adding or removing elements, especially from the middle of the list. This is because ArrayList has to shift all the elements after the insertion or deletion point to maintain its contiguous memory block structure.

LinkedList stores elements as nodes with a reference to the next node in the list. This allows for faster insertion and deletion of elements in the middle of the list, as the LinkedList only needs to update the references of the affected nodes.

Accessing elements in a LinkedList takes linear time O(n) because the LinkedList has to traverse each node from the head or tail to get to the desired element.

This makes LinkedList a good choice when you need to frequently add or remove elements from the middle of the list, or when the list is relatively small and the access time is not a bottleneck.

To summarize, ArrayList is generally faster for random access to elements, while LinkedList is generally faster for adding or removing elements from the middle of the list. The choice between them depends on the specific use case and the relative importance of different operations. If you need to frequently access elements by index or iterate over the list, use ArrayList. If you need to frequently add or remove elements from the middle of the list, or if the size of the list is small, use LinkedList.