Lists

Fsoft Academy

Lesson Objectives

- **Understand** the concepts of singly linked lists, doubly linked lists, and circular linked lists.
- **Be able to implement** basic operations like insertion, deletion, and traversal in singly and doubly linked lists.
- **Be able to implement** simple applications using linked lists (e.g., basic stack or queue functionality).
- **Understanding** the `java.util.LinkedList` Class
- **Differentiate** `java.util.LinkedList` from other List implementations like ArrayList.
- **Able to implement** basic operations like `add`, `get`, `remove`, `set`, and `contains` using LinkedList methods.

Agenda

1
- Abstract Data Types

2
- List Interface

3
- Singly/Doubly/Circular LinkedList

4
- The JDK LinkedList Class

Section 1

Abstract Data Types

Abstract Data Types (ADTs) - Introduction

- The **system-defined** data types:
  - ✓ all primitive data types (int, float, etc.)

    Java has 8 primitive data types: byte, short, int, long, float, double, char, boolean.These types support basic operations like addition (+), subtraction (-), multiplication (*), and division (/).
  - ✓ by default, support basic operations such as *addition* and *subtraction* (The system provides the implementations for the primitive data types)
- For **user-defined** data types: using 1 class: Student, Car, Person
  - ✓ we also need to define operations
  - ✓ the implementation for these operations can be done when we want to actually use them
- **ATDs**: combine the data structures with their operations:

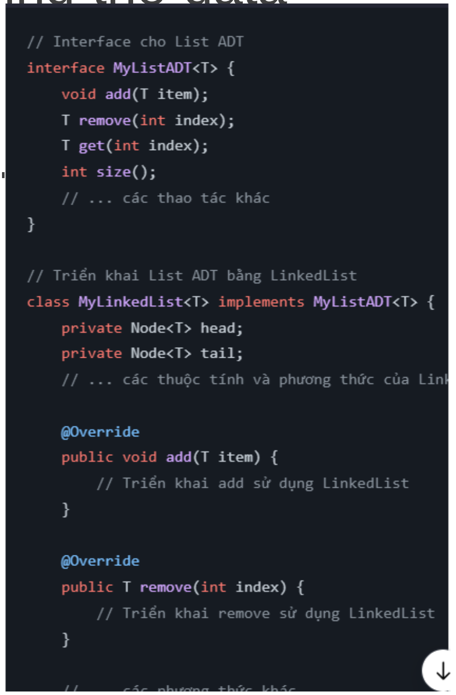
  both Abstract Data Types (ADTs) and User-Defined Types in Java are reference types
  - ✓ Declaration of data

    tóm lại là kiểu ADT thì phải định nghĩa thêm các operation trên data còn user-defined data type thì ko cần, chỉ cần data là đủ
  - ✓ Declaration of operations

Abstract Data Types (ADTs) - Introduction

- **Commonly used ADTs include:**
  - ✓ **Linked Lists**, **Stacks**, **Queues**, **Priority Queues**, **Binary Trees**, **Dictionaries**, **Disjoint Sets** (Union and Find), **Hash Tables**, **Graphs**, and many others.
- **For example,** "**stack**" uses a **LIFO** (Last-In-First-Out) mechanism while storing the data in data structures.
  - ✓ The last element inserted into the stack is the first element that gets deleted.
  - ✓ Common operations are:
    - **creating** the stack,
    - **push** an element onto the stack,
    - **pop** an element from the stack,
    - **finding** the current top of the stack,
    - **finding** the number of elements in the stack, etc.

abstract data type example:
(có thể có generic hoặc ko cần generic)



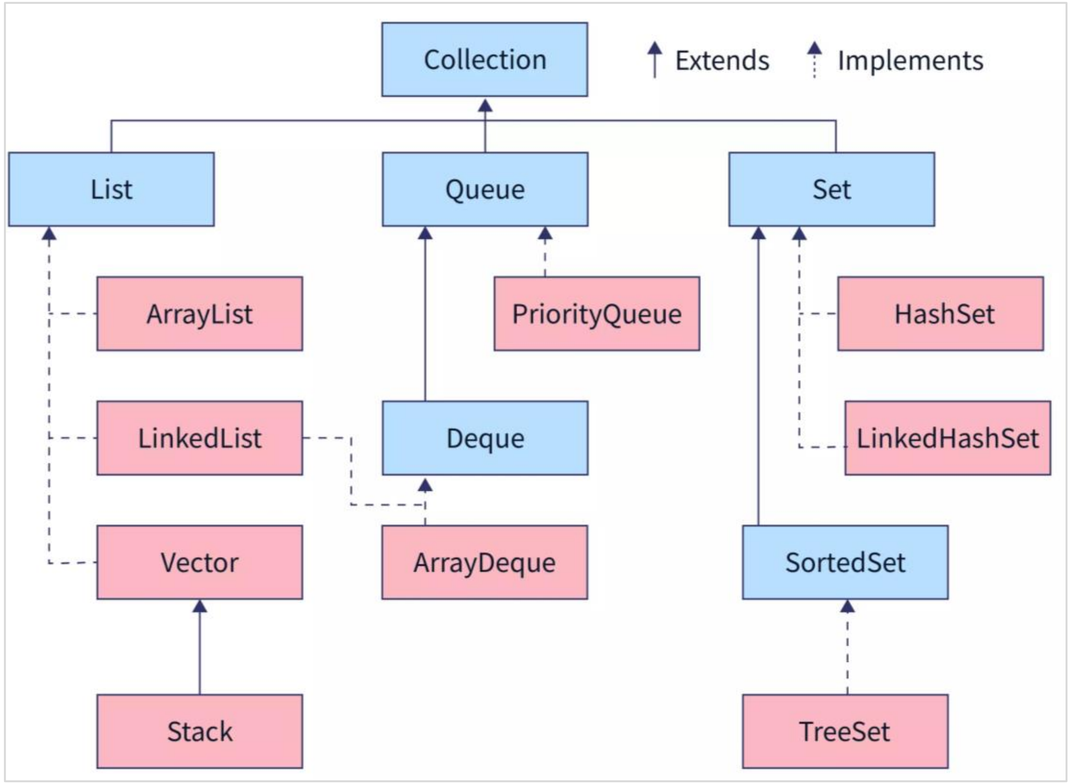
```java
// Interface cho List ADT
interface MyListADT<T> {
    void add(T item);
    T remove(int index);
    T get(int index);
    int size();
    // ... các thao tác khác
}

// Triển khai List ADT bằng LinkedList
class MyLinkedList<T> implements MyListADT<T> {
    private Node<T> head;
    private Node<T> tail;
    // ... các thuộc tính và phương thức của Link

    @Override
    public void add(T item) {
        // Triển khai add sử dụng LinkedList
    }

    @Override
    public T remove(int index) {
        // Triển khai remove sử dụng LinkedList
    }

    // ... các phương thức khác
```

Section 2

List Interface

Introduction List Interface - Recap

- The **List** is an interface in JDK, which is a child interface of **Collection**.
- You can access it using `java.util` package.
- The classes that implement the List interface in Java are **LinkedList**, **ArrayList**, **Vector**, **Stack**, etc

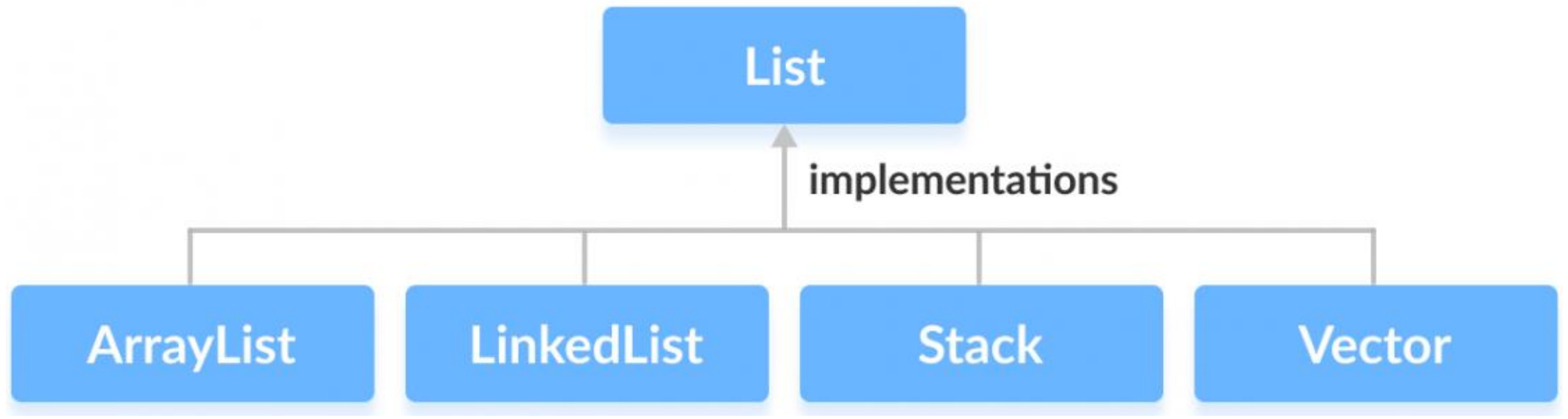


List Interface - Recap

The **List** interface is an ordered collection that allows us to **store** and **access** elements sequentially. It extends the **Collection** interface.

- Since **List is an interface**, we cannot create objects from it.
- We can use these classes:
  - ✓ArrayList
  - ✓LinkedList
  - ✓Vector
  - ✓Stack

List Interface - Recap



- **How to use List?**

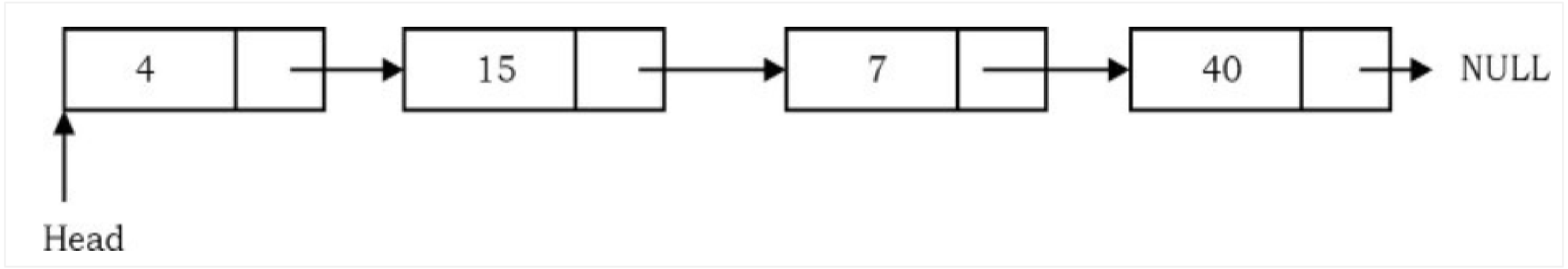
```
// ArrayList implementation of List
List<String> list1 = new ArrayList<>();
// LinkedList implementation of List
List<String> list2 = new LinkedList<>();
```

Section 3

Linked List

What is a Linked List?

- A linked list is a data structure used for storing collections of data. A linked list has the following properties.
    - ✓ Successive elements are connected by **pointers**
    - ✓ The **last element points to NULL**
    - ✓ Can **grow** or **shrink** in size during execution of a program
    - ✓ Can be made just as long as required (until systems memory exhausts)
    - ✓ Does not waste memory space (but takes some extra memory for pointers). It allocates memory as list grows.

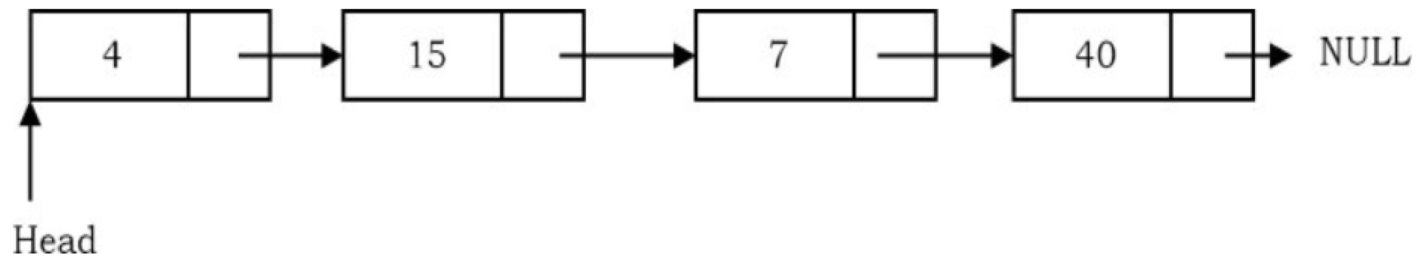


Types of Linked List

- **Following are the various types of linked list.**
  - ✓ **`Simple Linked List`** − Item navigation is forward only.
  - ✓ **`Doubly Linked List`** − Items can be navigated forward and backward.
  - ✓ **`Circular Linked List`** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

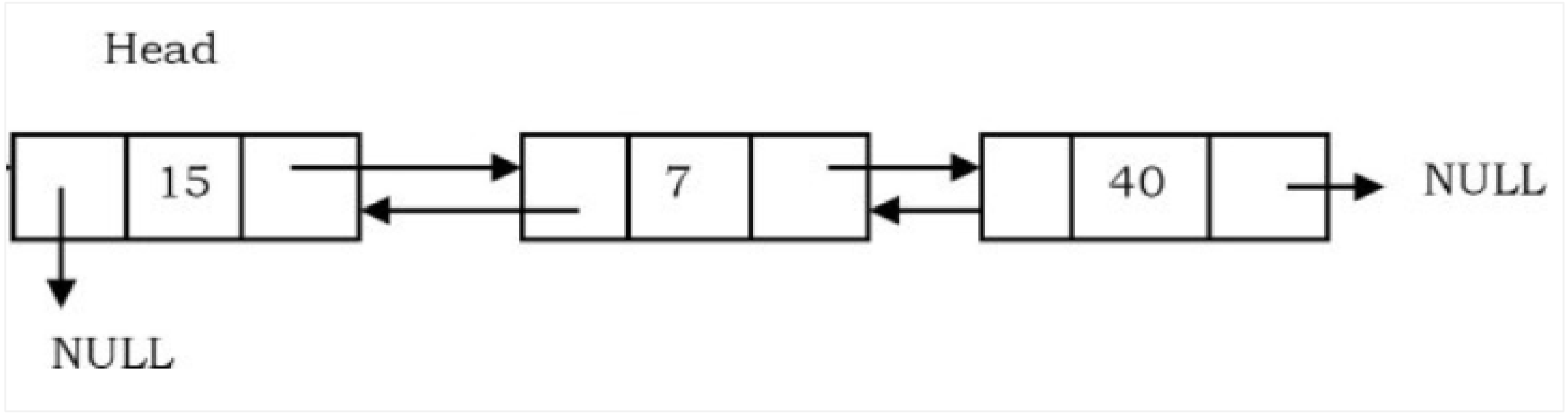
Types of Linked List

- **Singly Linked Lists**:
  - ✓ Generally “linked list” means a singly linked list
  - ✓ This list consists of a number of nodes in which each node has a *next* pointer to the following element
  - ✓ The link of the last node in the list is NULL, which indicates the end of the list



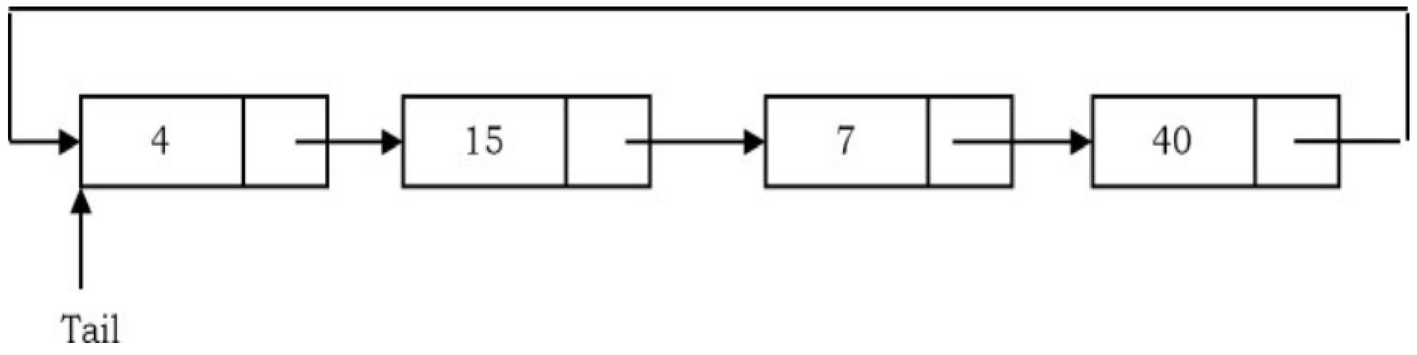
Types of Linked List

- **Doubly Linked Lists**
  - ✓ A node in the doubly linked list, we can navigate **in both directions**.
  - ✓ The primary *disadvantages* of doubly linked lists are:
    - Each node requires an extra pointer, requiring more space.
    - The insertion or deletion of a node takes a bit longer (more pointer operations).



Types of Linked List

- **Circular Linked Lists**:
  - ✓ Do not have ends.
  - ✓ While traversing the circular linked lists we should be careful; otherwise we will be traversing the list **infinitely**.
  - ✓ In circular linked lists, each node has a successor. Note that *unlike singly linked lists*, there is no node with NULL pointer in a circularly linked

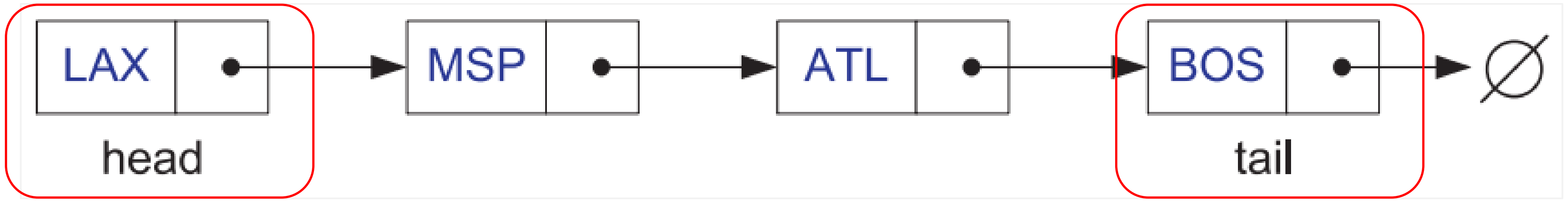


Section 3.1

Singly Linked List

Introduction

- **Arrays** are nice and simple for storing things in a certain order, but they have drawbacks.
  - ✓ makes **resizing** an array difficult.
  - ✓ **insertions** and **deletions** are difficult because elements need to be shifted around to make space for insertion or to fill empty positions after deletion.
- **Linked List:**
  - ✓ A ***linked list***, in its simplest form, is a collection of ***nodes*** that together form a linear ordering.
  - ✓ Each node stores a pointer, called *next*, to the next node of the list and stores its associated element.



Basic Operations

- **Following are the basic operations supported by a list.**
  - ✓**Insertion** − Adds an element at the beginning of the list.
  - ✓**Traversing** − Traverses and Displays the complete list.
  - ✓**Deletion** − Deletes an element at the beginning of the list.
  - ✓**Search** − Searches an element using the given key.
  - ✓**Delete** − Deletes an element using the given key.

Implementing a Singly Linked List

- **Step 1**: Define a class **Node**

```
// A linked list node
class Node {
    String data;
    Node next;

    Node(String value){
       this.data = value;
    }
}
 Node head;
 Node current;
```

✓ *The node stores two values, the member "`data`" stores the element stored in this node, which in this case is a character string.*

✓ *The member "`next`" stores a pointer to the next node of the list. We make the linked list class a friend, so that it can access the node's private members.*

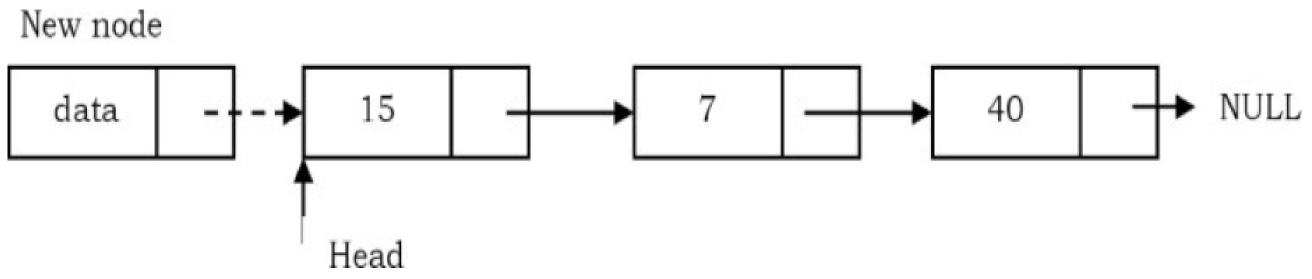
Singly Linked List Insertion

- **Insertion into a singly-linked list has three cases:**
  - ✓Inserting a new node before the head (at the beginning)
  - ✓Inserting a new node after the tail (at the end of the list)
  - ✓Inserting a new node at the middle of the list (random location)
- ***Note:*** To insert an element in the linked list at some position *p*, assume that after inserting the element the position of this new node is *p*

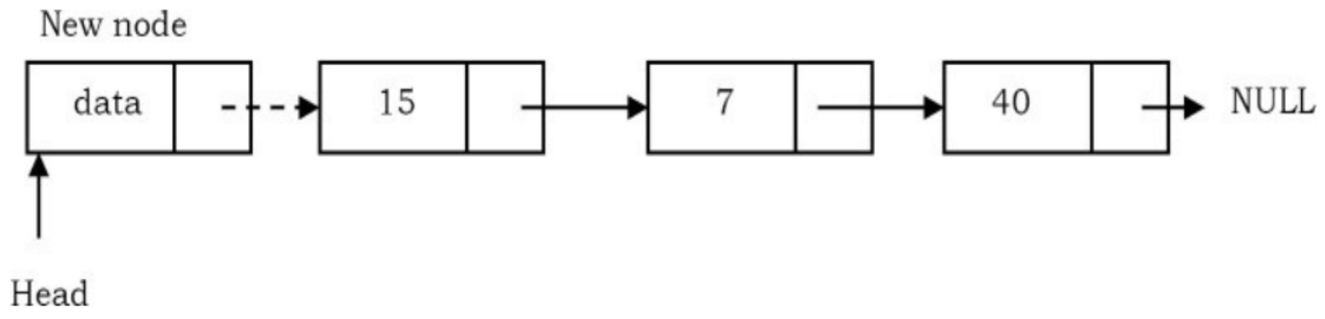
Singly Linked List Insertion

Inserting a Node in Singly Linked List at the Beginning

- **Two steps only:**
  - ✓ ***Update the next pointer of new node, to point to the current head.***



  - ✓ ***Update head pointer to point to the new node.***



Singly Linked List Insertion

Inserting a Node in Singly Linked List at the Beginning

- **Algorithm:**

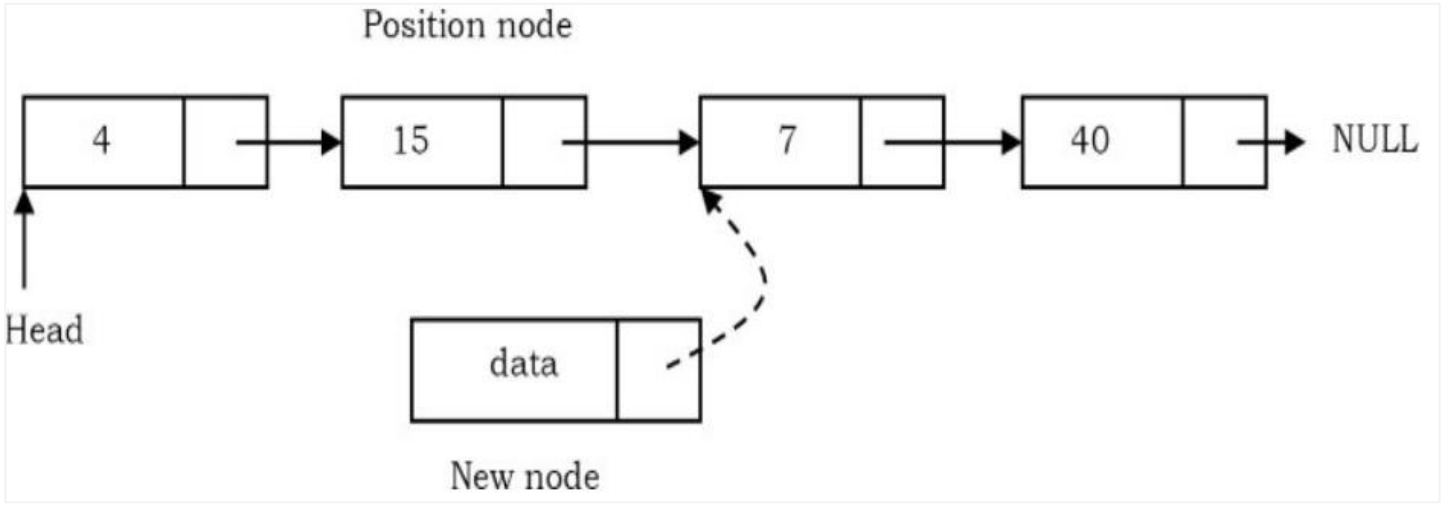
```
//insert link at the first location
public void insertFirst(LinkedList list, int data) {
   //create a link
   Node newNode = new Node(data);

   //point it to old first node
   newNode.next = list.head;

   //point first to new first node
   list.head = newNode;
}
```

Singly Linked List Insertion

Inserting a Node in Singly Linked List at the Middle

❖Two steps:

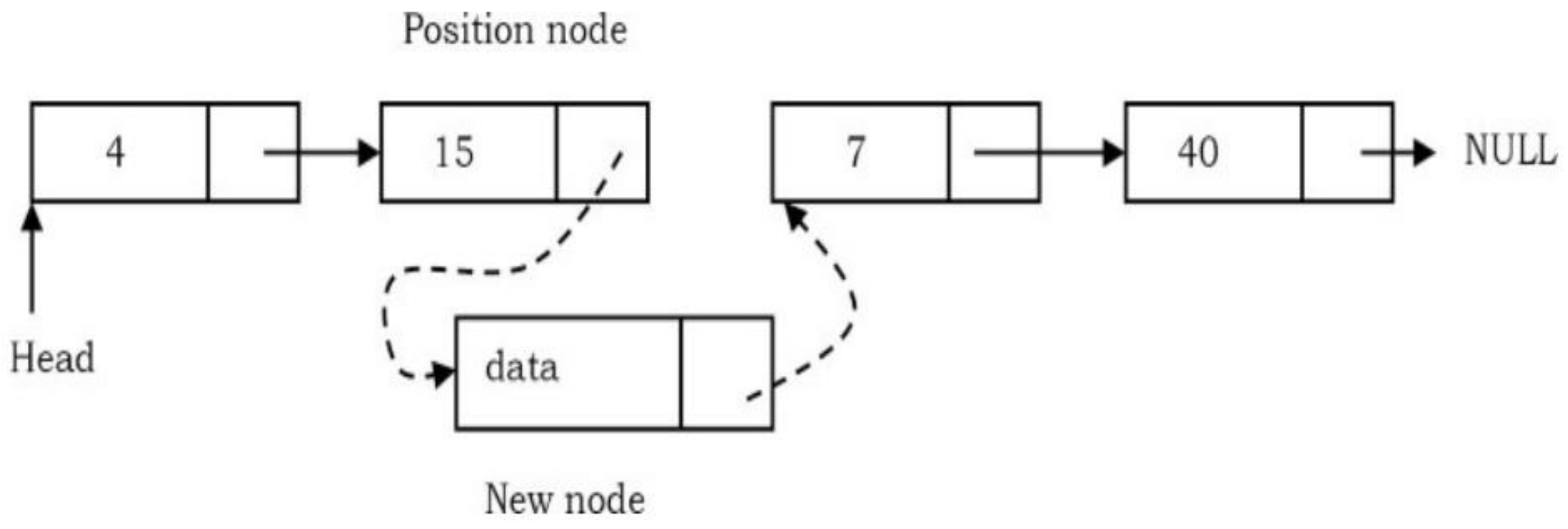
- ✓ ***If we want to add an element at position 3 then we stop at position 2.***
- ✓ That means we traverse 2 nodes and insert the new node. For simplicity let us assume that the second node is called the *positioning* node. The new node points to the next node of the position where we want to add this node.



Singly Linked List Insertion

Inserting a Node in Singly Linked List at the Middle

- **Two steps:**
  - ✓ ***Position node's next pointer now points to the new node.***



Traversing the Linked List

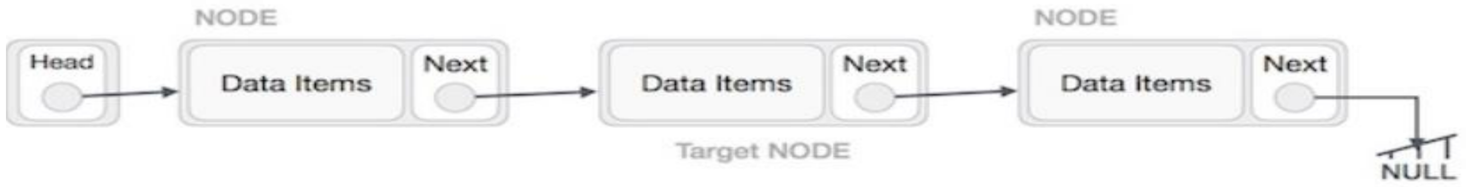
- **Algorithm:**

```
public void printList(LinkedList list) {
        Node ptr = list.head;
        System.out.print("[ ");
        //start from the beginning
        while(ptr!=null){
        System.out.print(ptr.data + " ");
        ptr = ptr.next;
        }
        System.out.println("]");
}
```

- **Complex Algorithm:**
  - ***Time Complexity:*** O(*n*), for scanning the list of size *n*.
  - ***Space Complexity:*** O(1), for creating a temporary variable.

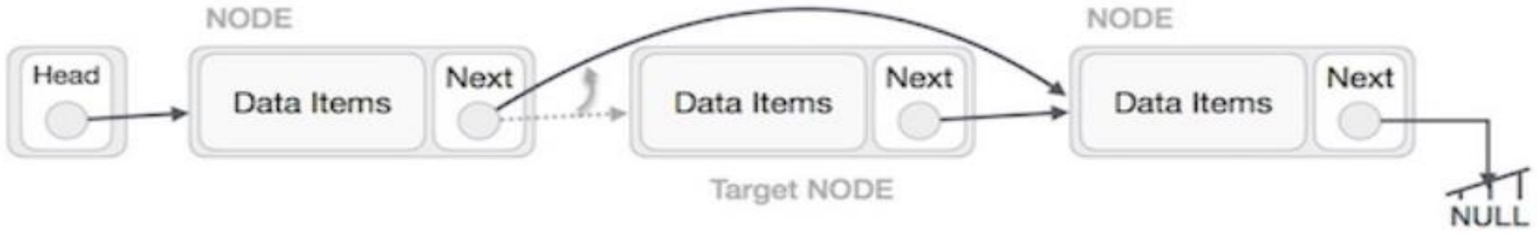
Deletion Operation

❖First, locate the target node to be removed, by using searching algorithms.



❖The left (previous) node of the target node now should point to the next node of the target node

```
LeftNode.next -> TargetNode.next;
```

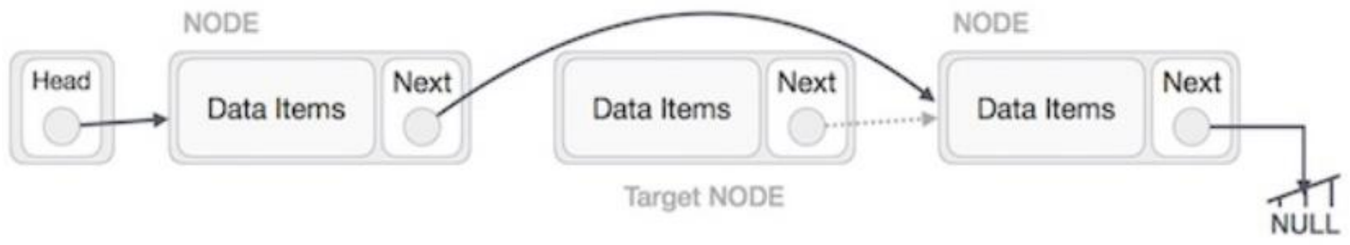


❖This will remove the link that was pointing to the target node.

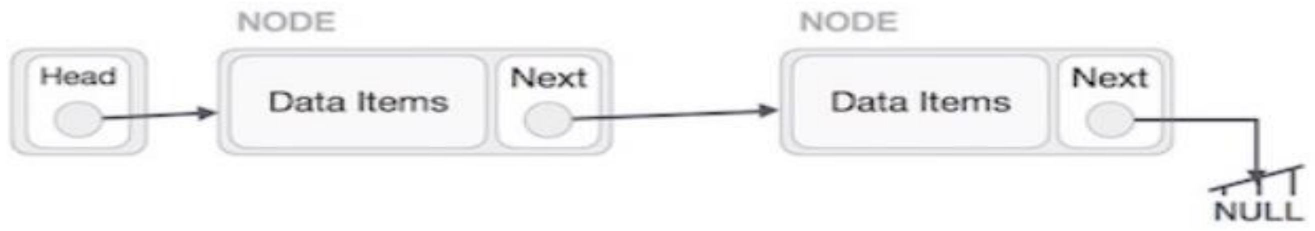
Deletion Operation

❖Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next −> NULL;
```



❖We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.



Singly Linked List

- Code Demo!!

Section 3.2

Doubly Linked List

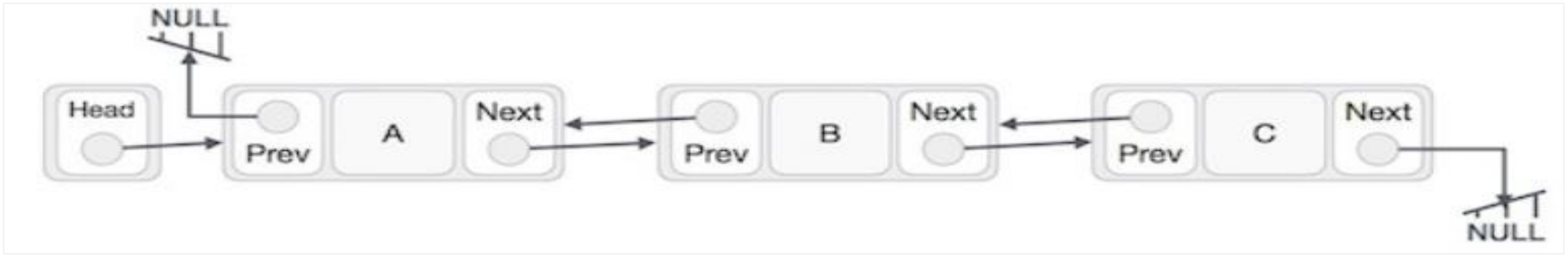
Introduction

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List.

- **The concept of doubly linked list:**
  - ✓ **Link** − Each link of a linked list can store a data called an element.
  - ✓ **Next** − Each link of a linked list contains a link to the next link called Next.
  - ✓ **Prev** − Each link of a linked list contains a link to the previous link called Prev.
  - ✓ **LinkedList** − A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation

- Doubly Linked List contains a link element called *first* and *last*.
- Each link carries a data field(s) and two link fields called *next* and *prev*.
- Each link is linked with its next link *using its next link*.
- Each link is linked with its previous link using its *previous link*.
- The last link carries a link as null to mark the *end of the list*.



Basic Operations

- **Following are the basic operations supported by a list.**
  - ✓ **Insertion** − Adds an element at the beginning of the list.
  - ✓ **Deletion** − Deletes an element at the beginning of the list.
  - ✓ **Insert Last** − Adds an element at the end of the list.
  - ✓ **Delete Last** − Deletes an element from the end of the list.
  - ✓ **Insert After** − Adds an element after an item of the list.
  - ✓ **Delete** − Deletes an element from the list using the key.
  - ✓ **Display forward** − Displays the complete list in a forward manner.
  - ✓ **Display backward** − Displays the complete list in a backward manner.

Basic Operations

- **Pseudo code - Create node:**

```
/* Node of a doubly linked list */
class Node {
    int data;
    Node next; // Pointer to next node in DLL
    Node prev; // Pointer to previous node in DLL

    Node(int value){
       this.data = value;
    }
}
Node head;
```

Insertion Operation

- **Pseudo code:** Following code demonstrates the insertion operation at the beginning of a doubly linked list.

```
//insert link at the first location
public void insertFirst(LinkedList list, int data) {

   //create a link
   Node newNode = new Node(data);

   //point it to old first link
   newNode.next = list.head;

   if(head != null) {
      //update first prev link
      list.head.prev = newNode;
   }
   //point first to new first link
   list.head = newNode;
}
```

Deletion Operation

- Following code demonstrates the deletion operation at the beginning of a doubly linked list.

```
// delete first item
public Node deleteFirst(LinkedList list) {

   //save reference to first link
   Node tempLink = list.head;

   if(list.head.next != null) {
        list.head.next.prev = null;
   }
   list.head = list.head.next;

   //return the deleted link
   return tempLink;
}
```

Insertion at the End of an Operation

- Following code demonstrates the insertion operation at the last position of a doubly linked list.

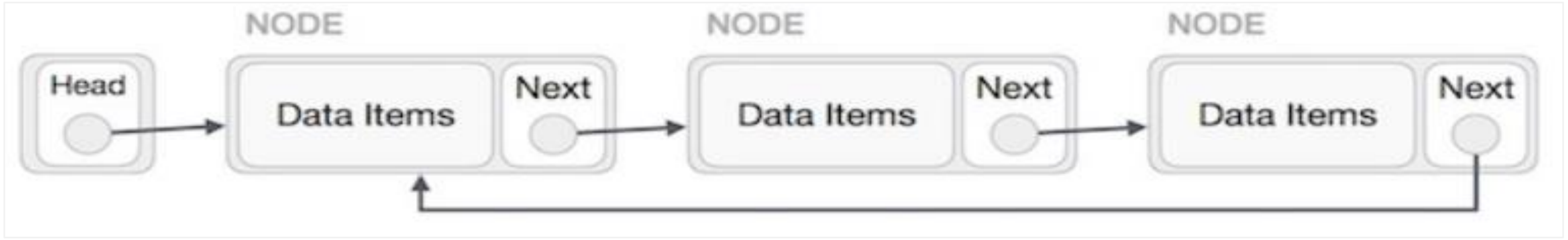
```
//insert link at the last location
public void insertLast(LinkedList list, int data) {

   //create a link
   Node newNode = new Node(data);

   Node last = list.head;
   if(head == null) {
          head = newNode;
          return;
   }

   //find the last link
   while (last.next != null) {
          last = last.next;
   }

    //make link a new last link
    last.next = newNode;

    //mark old last node as prev of new link
    newNode.prev = last;    }
}
```

Section 3.3

Circular Linked List

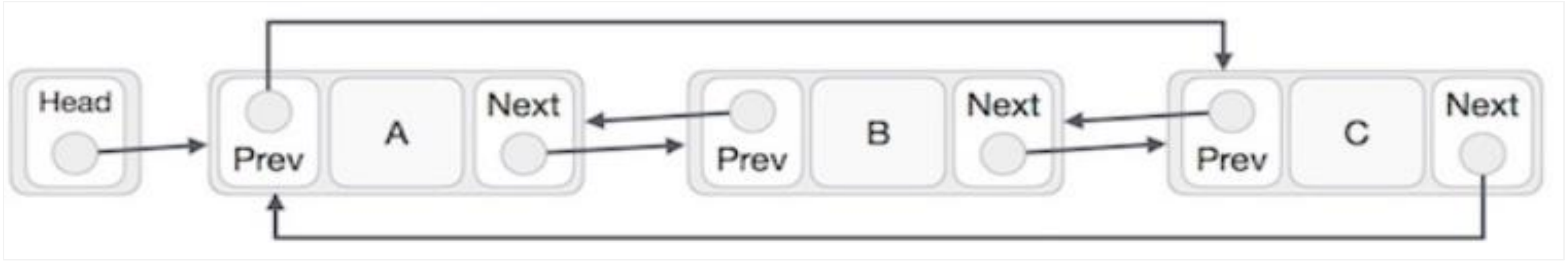
Introduction

- **Circular Linked List** is a variation of Linked list in which the first element points to the last element and the last element points to the first element.
- Both Singly Linked List and Doubly Linked List can be made into a circular linked list.
- **Singly Linked List as Circular**
  - ✓ The next pointer of the last node points to the first node.



Introduction

- **Doubly Linked List as Circular**
  - ✓The next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



Basic Operations

- Following are the important operations supported by a circular list.
  - ✓**insert** − Inserts an element at the start of the list.
  - ✓**delete** − Deletes an element from the start of the list.
  - ✓**display** − Displays the list.

Basic Operations

- **Create node:**

```
/* structure for a node */
class Node {
    int data;
    Node next;

    Node(int value){
       this.data = value;
    }

}
Node head;
```

Insertion Operation

- Following code demonstrates the insertion operation in a circular linked list based on single linked list.

```
public void insertFirst(LinkedList list, int data) {
    Node newNode = new Node(data);

    if(list.head == NULL){
         head = newNode;
         newNode.next = list.head;
    }
    else {
         Node last = list.head;
         while (last.next != list.head) {
            last = last.next;
         }

         newNode.next = list.head;
         last.next = newNode;
         List.head = newNode;

         }
```

Deletion Operation

❖Following code demonstrates the deletion operation in a circular linked list based on single linked list.

```
/* Function to delete a given node from the list */
public void deleteNode(LinkedList list, int key)
{
    if(list.head == null) {
       return;
    }
    // Find the required node
    Node curr = list.head;
    Node prev = new Node();
    while (curr.data != key) {
        if (curr.next == list.head) {
          System.out.println("Given node is not “
                + “found in the list");
          return;
        }
        prev = curr;
        curr = curr.next;
    }
```

```
   // Check if node is only node
    if(curr == list.head && curr.next == list.head)
    {
         list.head = null;
         return;
    }
    // If more than one node, check if
    // it is first node
    if (curr == list.head){
        prev = list.head;
        while (prev.next != list.head) {
          prev = prev.next;
        }

        list.head = curr.next;
        prev.next = list.head;
     }
```

Deletion Operation

❖Following code demonstrates the deletion operation in a circular linked list based on single linked list.

```
          else {
              prev.next = curr.next;
          }
}
```

Display List Operation

❖Following code demonstrates the display list operation in a circular linked list.

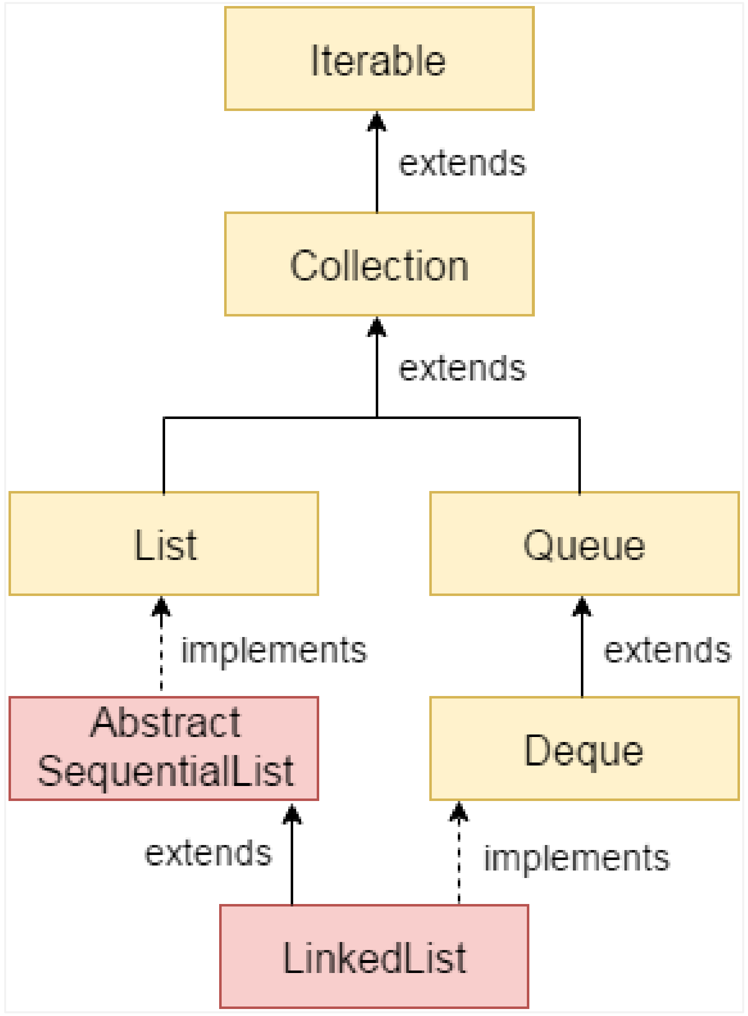
```
/* Function to print nodes in a given
  circular linked list */
public void printList(LinkedList list)
{
    Node temp = list.head;
    if (list.head != NULL) {
        do {
            System.out.print(temp.data + " ");
            temp = temp.next;
        } while (temp != list.head);
    }
}
```

Section 3.4

The JDK LinkedList Class

LinkedList

- Java **LinkedList** class uses a **doubly linked** list to store the elements.
- The important points about Java LinkedList are:
  - ✓ Java LinkedList class can contain duplicate elements.
  - ✓ Java LinkedList class maintains insertion order.
  - ✓ Java LinkedList class is non synchronized.
  - ✓ In Java LinkedList class, manipulation is fast because no shifting needs to occur.
  - ✓ Java LinkedList class can be used as a list, stack or queue.



LinkedList

- **Doubly Linked List**

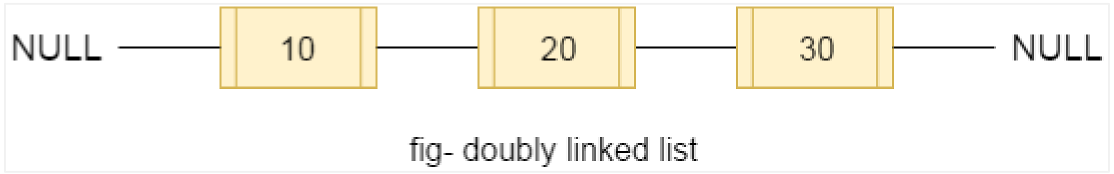


fig- doubly linked list

- **Declaration:**

```
public class LinkedList<E> extends AbstractSequentialList<E>
        implements List<E>, Deque<E>, Cloneable, Serializable
```

- **LinkedList** is implemented as a double linked list.
- Its performance on **add()** and **remove()** *is better than the performance of Arraylist*. The **set()** and **get()** methods have *worse performance than the ArrayList*, as theLinkedList does not provide direct access.

	ArrayList	LinkedList
get()	O(1)	O(n)
add()	O(1)	O(1) amortized
remove()	O(n)	O(n)

Main methods of LinkedList

Method	Description
void 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 get(int index)	It is used to return the element at the specified position in 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.

Main methods of LinkedList

Method	Description
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.
E removeFirst()	It removes and returns the first element from a list.
E removeLast()	It removes and returns the last element from a list.

LinkedList Example

```java
public class TestLinkedList {
   public static void main(String args[]) {

      LinkedList<String> list = new LinkedList<String>();
      System.out.println("Initial list of elements: " + list);
      list.add("Java");
      list.add("Net");
      list.add("Android");
      System.out.println("After invoking add(E e) method: " + list);

      // Adding an element at the specific position
      list.add(1, "iOs");
      System.out.println("After invoking add(int index, E element) method: " + list);
      LinkedList<String> list2 = new LinkedList<String>();
      list2.add("Test");
      list2.add("Automation Test");

      // Adding second list elements to the first list
      list.addAll(list2);
      System.out.println("After invoking addAll(Collection<? extends E> c) method: " + list);
      // Adding an element at the first position
      list.addFirst("C++");
      System.out.println("After invoking addFirst(E e) method: " + list);
```

LinkedList Example

```
        // Adding an element at the last position
        list.addLast("Kotlin");
        System.out.println("After invoking addLast(E e) method: " + list);
        list.removeFirst();
        System.out.println("After invoking removeFirst() method: " + list);
        list.removeLast();
        System.out.println("After invoking removeLast() method: " + list);
    }
}
```

Output:

```
Initial list of elements: []
After invoking add(E e) method: [Java, Net, Android]
After invoking add(int index, E element) method: [Java, iOs, Net, Android]
After invoking addAll(Collection<? extends E> c) method:
                                         [Java, iOs, Net, Android, Test, Automation Test]
After invoking addFirst(E e) method: [C++, Java, iOs, Net, Android, Test, Automation Test]
After invoking addLast(E e) method: [C++, Java, iOs, Net, Android, Test, Automation Test, Kotlin]
After invoking removeFirst() method: [Java, iOs, Net, Android, Test, Automation Test, Kotlin]
After invoking removeLast() method: [Java, iOs, Net, Android, Test, Automation Test]
```

Difference between ArrayList and LinkedList

#	ArrayList	LinkedList
1	ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2	Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3	An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4	ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.

Lesson Summary

- Abstract Data Types
- List Interface
- Singly/Doubly/Circular LinkedList
- The JDK LinkedList Class

THANK YOU!

CODE< YOUR LIFE >