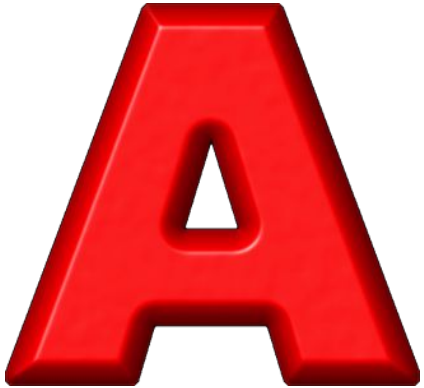
RIZAL TECHNOLOGICAL UNIVERSITY
COLLEGE OF ENGINEERING ARCHITECTURE AND TECHNOLOGY

DATA STRUCTURE AND ALGORITHM

GUESS THE TERM



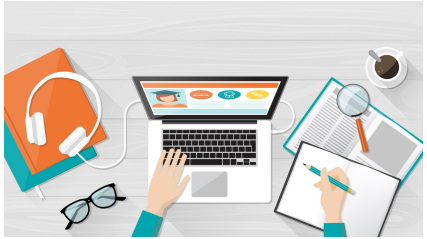
GUESS THE NAME

+

GUESS THE TERM

+ R +

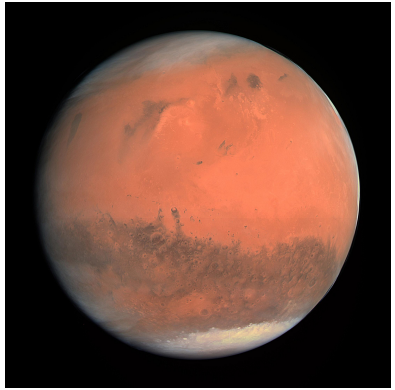
+



GUESS THE NAME

+

+



GUESS THE TERM

+

LINKED LIST

- What is Linked List?
- Linked List ADT
- Why Linked List
- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Unrolled Linked List
- Skip List

Objectives:

- Define linked list
- Differentiate singly-linked list from doubly-linked list.
- List the algorithms and perform operations in singly-and doubly-linked list
- Define circular linked list, unrolled linked list and skip lists.

LINKED LIST

- is a linear data structure, in which the elements are not stored at contiguous memory locations
- it consists of **nodes** where each node contains a data field and a reference(link) to the next node in the list.

The elements in a linked list are linked using **pointers**

Data	Pointer

Note that in Java, primitive types like int and double are stored quite differently than objects.

Why Linked List?

- **Dynamic Size**. Lists that are fixed in length (array) cannot dynamically alter their size once they have defined

- **Ease of insertion or deletion.** Lists that do not require a considerable amount of insertion and deletion operations

Java Linked List

Create linked lists in Java.

```
LinkedList<Type> linkedListname = new LinkedList<>();
```

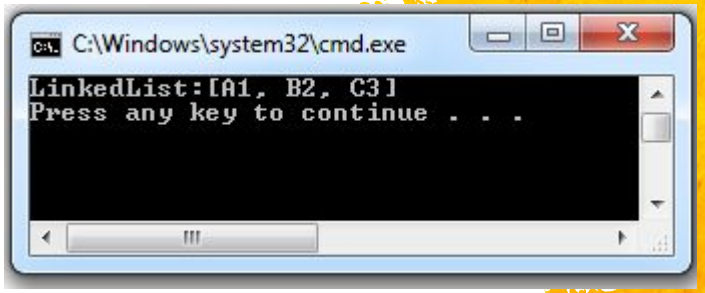
For example:

```
LinkedList<Integer> linkedList = new LinkedList<>();

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

Linked List : Java

Create linked list in Java.

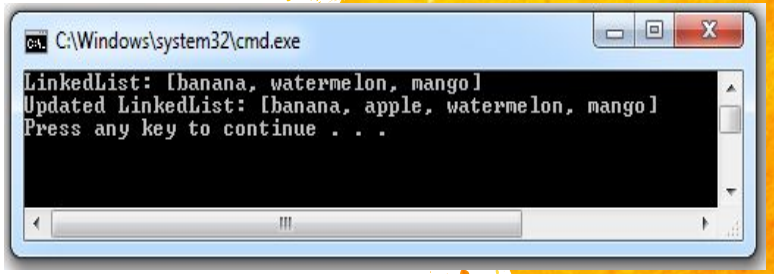
```java
import java.util.LinkedList;

class CreateLinkedList {

    public static void main(String[]args) {

     LinkedList<String> llstring = new LinkedList<>();

        llstring.add("A1");
        llstring.add("B2");
        llstring.add("C3");

        System.out.println("LinkedList:" + llstring);

    }
}
```



Linked List : Methods of Java

- Add element – add( ) method

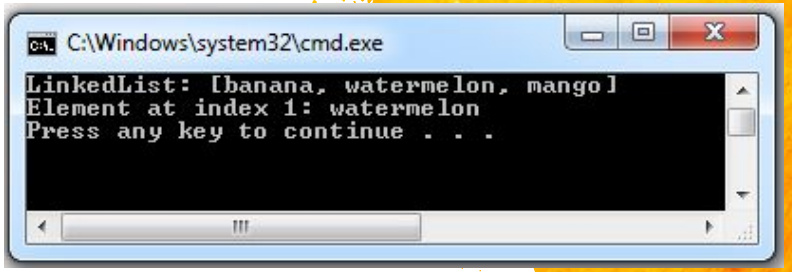
```java
import java.util.LinkedList;

class AddElemLinkedList {
  public static void main(String[] args){
    // create linkedlist
    LinkedList<String> fruits = new LinkedList<>();

    // add() method without the index parameter
    fruits.add("banana");
    fruits.add("watermelon");
    fruits.add("mango");
    System.out.println("LinkedList: " + fruits);

    // add() method with the index parameter
    fruits.add(1, "apple");
    System.out.println("Updated LinkedList: " + fruits);
  }
}
```



Linked List : Methods of Java

- Access element – get( ) method

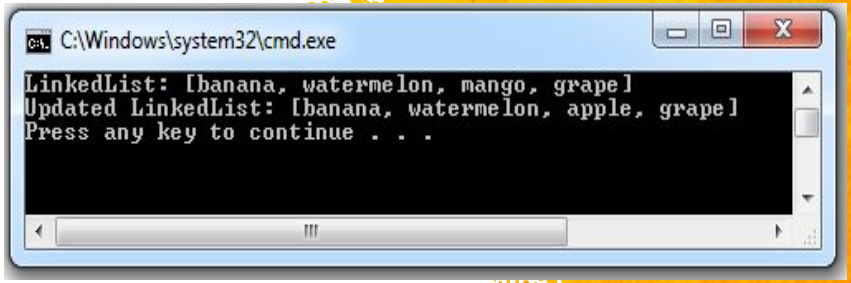
```java
import java.util.LinkedList;

class AccessElemLinkedList {
  public static void main(String[] args){
    // create linkedlist
    LinkedList<String> fruits = new LinkedList<>();

    // add() method without the index parameter
    fruits.add("banana");
    fruits.add("watermelon");
    fruits.add("mango");
    System.out.println("LinkedList: " + fruits);

    // get the element from the linked list
       String str = fruits.get(1);
       System.out.println("Element at index 1: " + str);
  }
}
```



Linked List : Methods of Java

- Change element – set( ) method

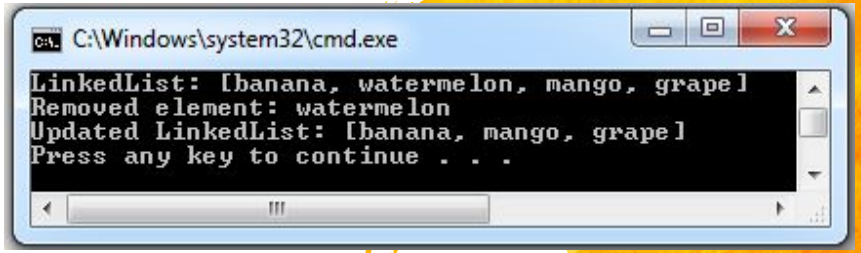
```java
import java.util.LinkedList;

class ChangeElemLinkedList {
  public static void main(String[] args){
    // create linkedlist
    LinkedList<String> fruits = new LinkedList<>();

    // add() method without the index parameter
    fruits.add("banana");
    fruits.add("watermelon");
    fruits.add("mango");
    fruits.add("grape");
    System.out.println("LinkedList: " + fruits);

    // change element at index 2
       fruits.set(2, "apple");
       System.out.println("Updated LinkedList: " + fruits);
  }
}
```



Linked List : Methods of Java

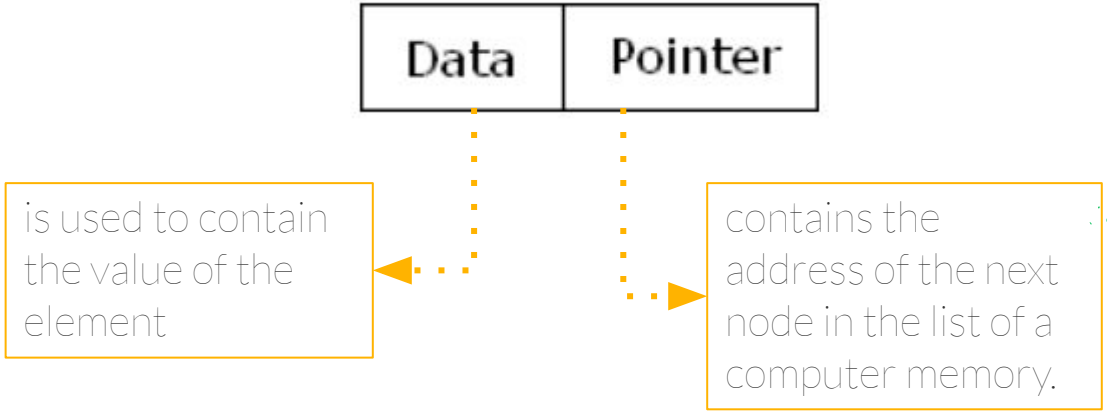
- Remove element – remove( ) method

```java
import java.util.LinkedList;

class RemoveElemLinkedList {
  public static void main(String[] args){
    // create linkedlist
    LinkedList<String> fruits = new LinkedList<>();

    // add() method without the index parameter
    fruits.add("banana");
    fruits.add("watermelon");
    fruits.add("mango");
    fruits.add("grape");
    System.out.println("LinkedList: " + fruits);

    // remove element from index 1
       String str = fruits.remove(1);
       System.out.println("Removed element: " + str);

       System.out.println("Updated LinkedList: " + fruits);
  }
}
```



Singly-Linked List

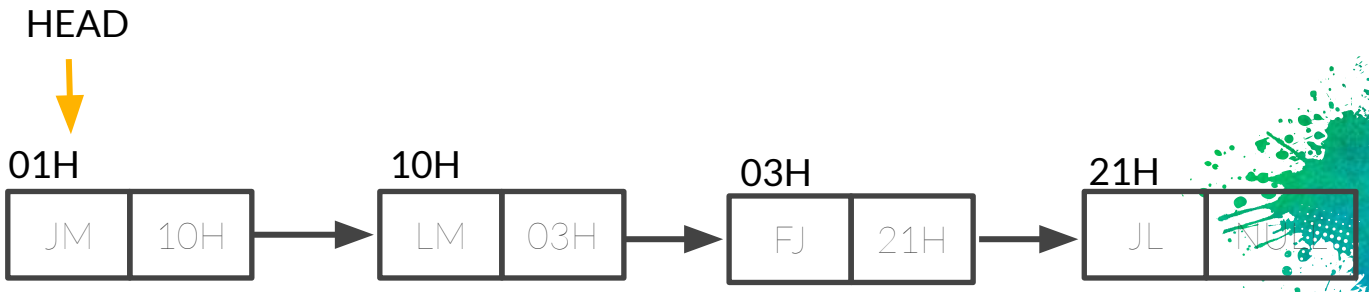
- basic format of a node



- all references are the same size no matter what they refer in a given computer or operating system.

Singly-Linked List

- the "Next Node" in the list is called the SUCCESSOR
- HEAD first node in the list



Note that there is no limit to the size of a singly-linked list. Adding a node to a linked list is simply a matter of:

- creating a new node
- Setting the data field of the new node to the value to be inserted into the list.
- Assigning the address of the new node to the pointer field of the current last node in the list
- Setting the pointer field of the new node to NULL.

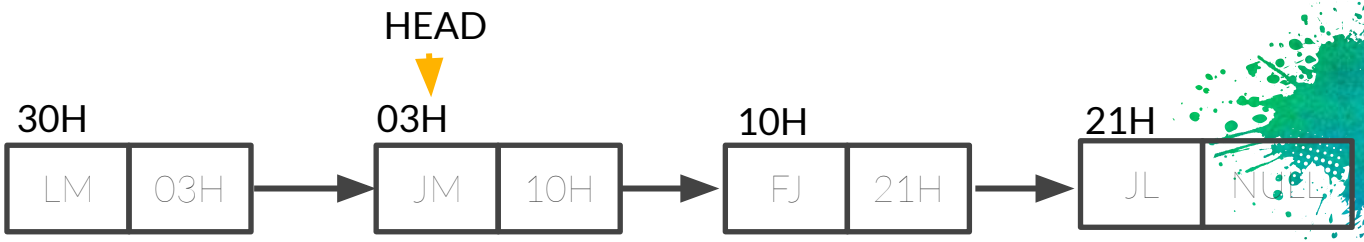
Singly-Linked List : Common Operations

- **Insert.** usually inserted at the beginning of the list
- **Search.** moves along the list searching for the specified value then prompts you the address of the node
- **Delete.** search for the value and delete the data then connects the arrow from the previous link straight across to the following link

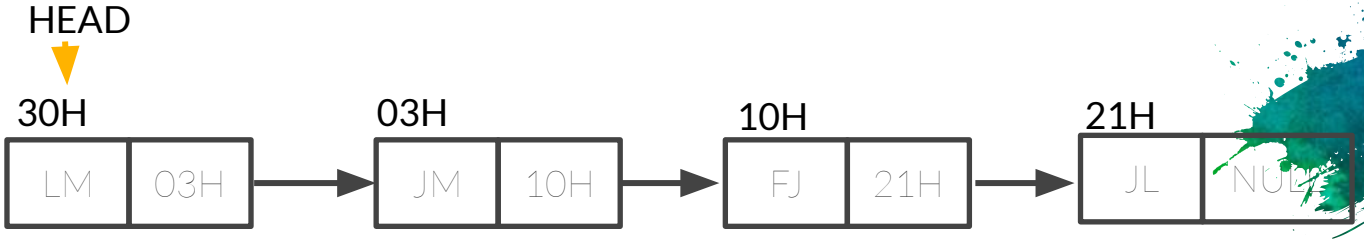
Singly-Linked List

Inserting a Node at the Start of a Singly-Linked List (i=1)

- Set the pointer field of the new node to the value of HEAD when the HEAD points to the current first node in the list
- Set HEAD to the address of the new node



Singly-linked list after inserting “LM”

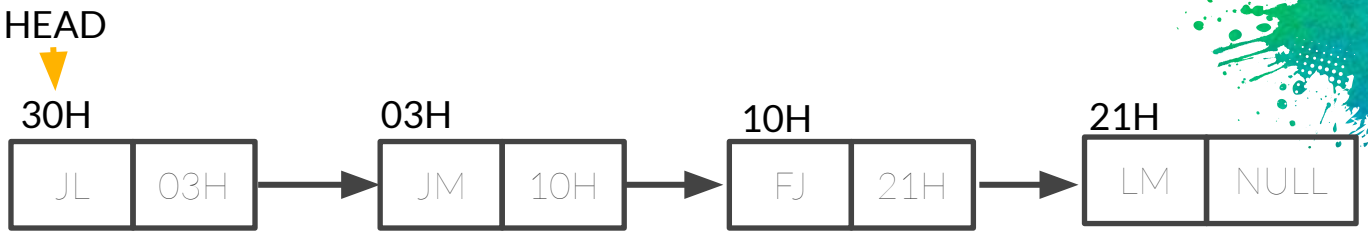


Singly-linked list after reassigning HEAD

Singly-Linked List

Inserting a Node at the End of a Singly-Linked List (i>Length of List)

- Set the pointer field of the current last node to the address of the new node when the pointer field of the current last node contains NULL.
- Set the pointer field of the new node to NULL

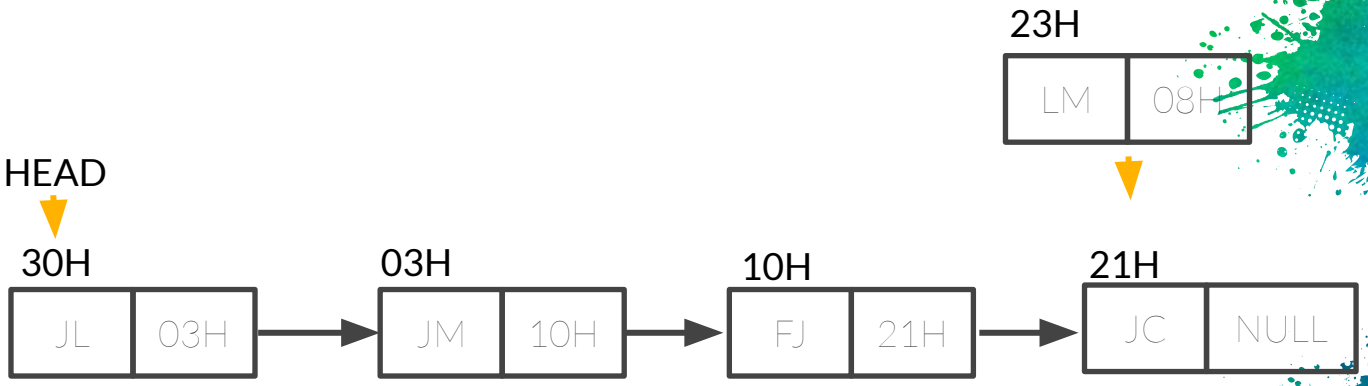


Singly-linked list after inserting “LM” at the end

Singly-Linked List

Inserting a Node at Position i (1<i<Length of List)

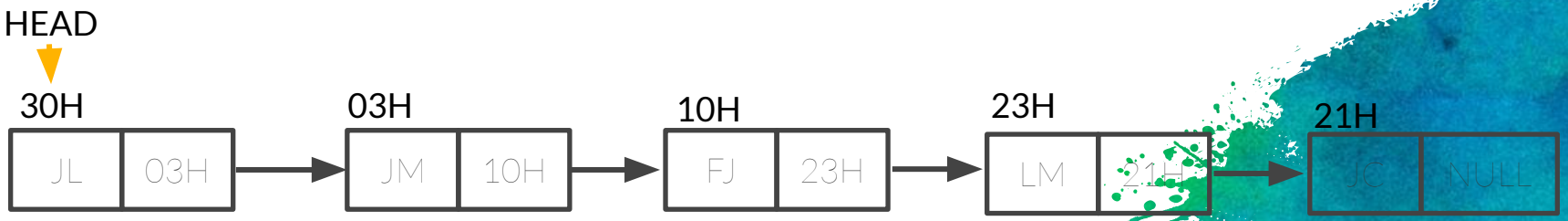
- Locate the node at position i-1.
- Set the pointer field of the new node to the value of the pointer field of node i-1.
- Set the pointer field of node i-1 to the address of the new node.



Singly-linked list after setting the pointer field of “LM”

Singly-Linked List

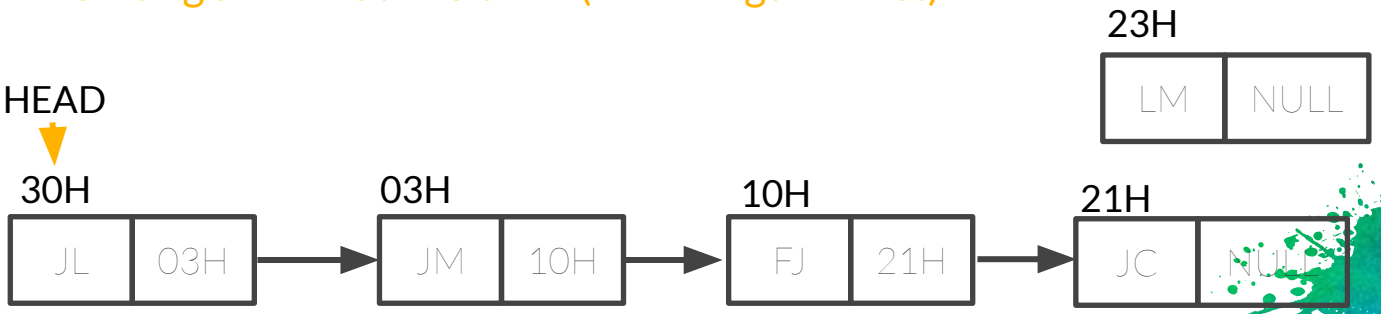
Inserting a Node at Position i (1<i<Length of List)



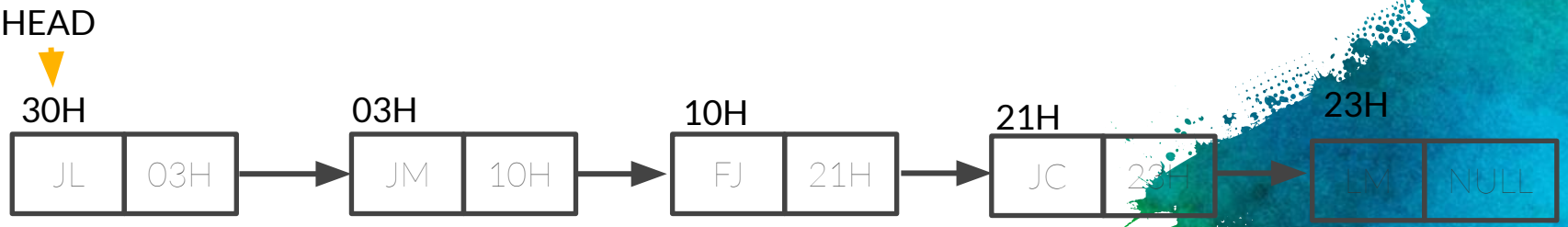
Singly-linked list after inserting “LM” at position 3

Singly-Linked List

Inserting a Node at Position i (1<i<Length of List)



Singly-linked list after setting the pointer field of “LM”

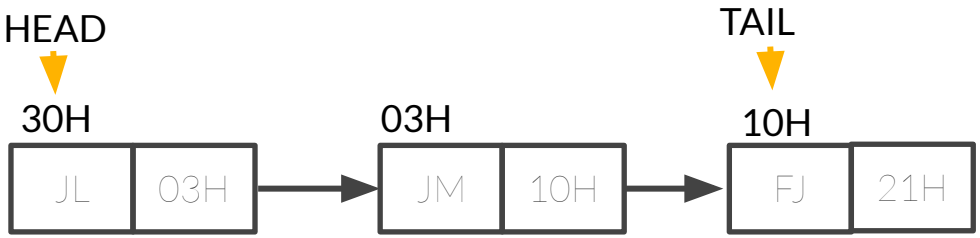


Singly-linked list after setting the pointer field of “JC”

Singly-Linked List

Inserting using the "Tail"

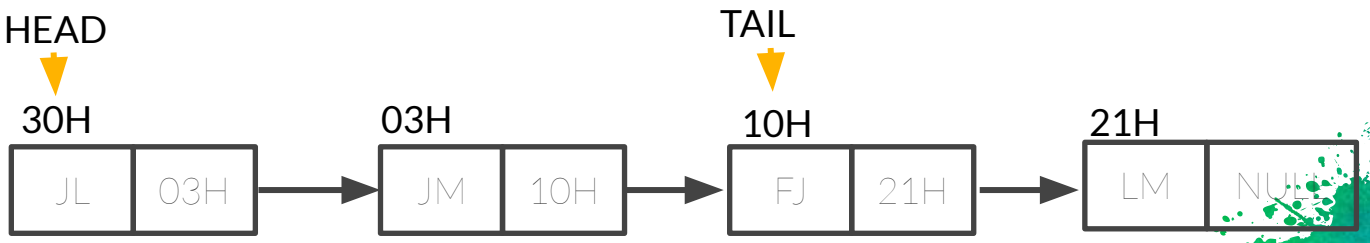
- Create a new node for the element.
- Set the data field of the new node to the value to be inserted.
- Set the pointer field of the new node to the value of NULL.
- Set the pointer field of the node referenced by TAIL to the address of the new node.
- Set TAIL to the address of the new node.



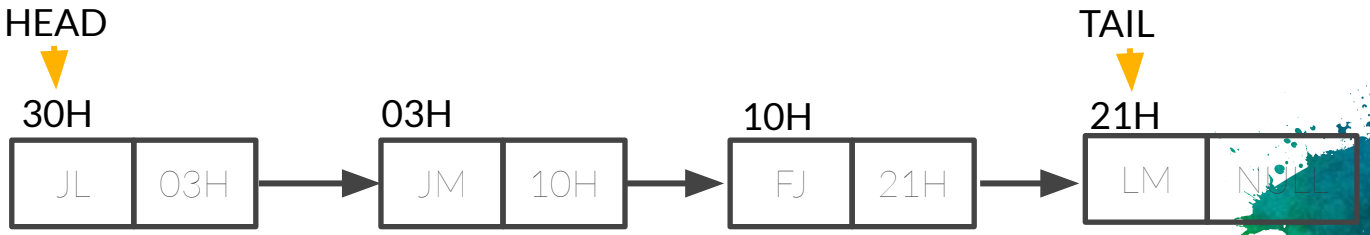
Singly-linked list with HEAD and TAIL pointers

Singly-Linked List

Inserting using the “Tail”



List after setting the pointer field of “LM”

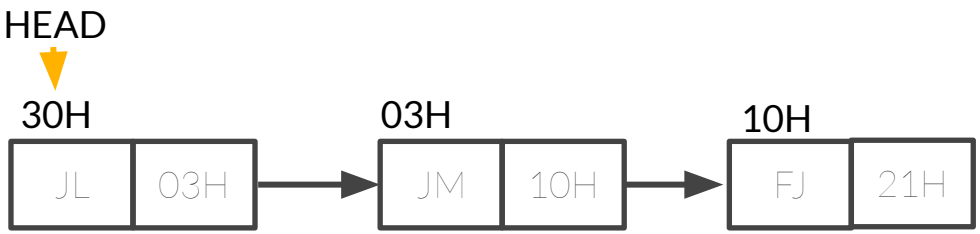


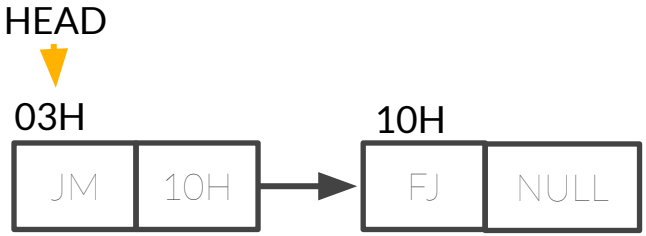
List after setting TAIL

Singly-Linked List

Deleting the Node at the Head of the List (i=1)

- Set the variable HEAD to the address contained in the pointer field of the node to be deleted.



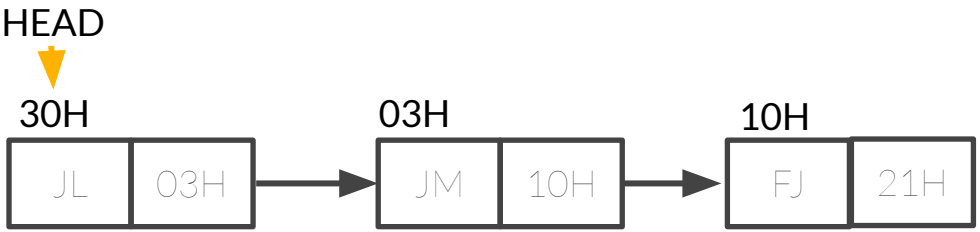


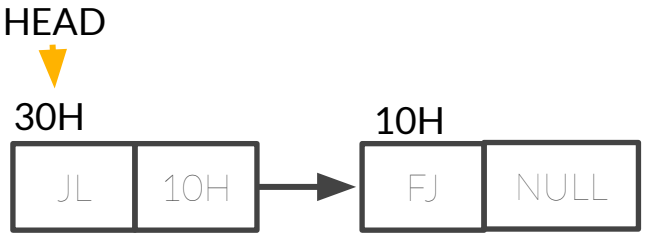
Singly-linked list after deleting "JL"

Singly-Linked List

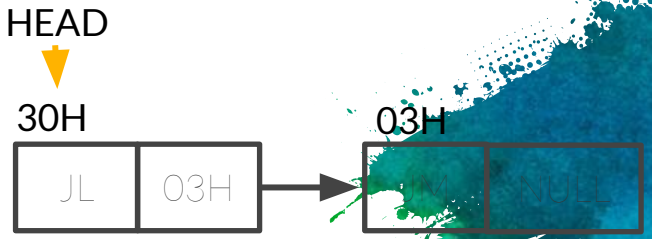
Deleting a Node not at the Head of the List (1<i≤n)

- Locate the preceding node (i-1).
- Set the pointer field of the preceding node (i-1) to thevalue in the pointer field of the node to be deleted.



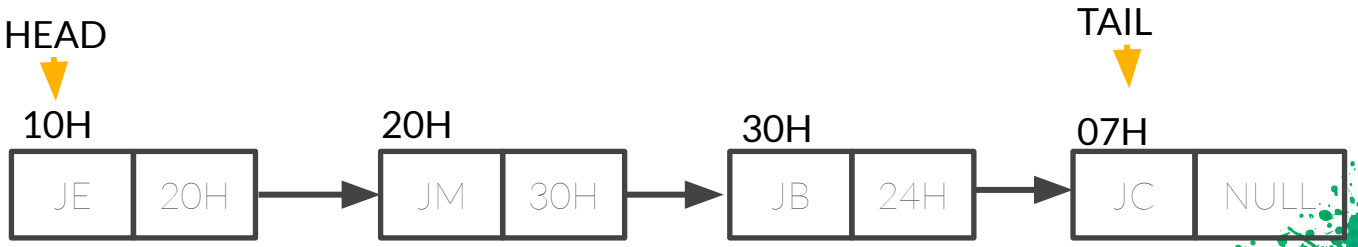


Singly-linked list after deleting "JM"



Singly-linked list after deleting "FJ"

Exercise: Singly-Linked List



- Insert a Node at Position 3

Singly-Linked List : Algorithm Basic Operation

Inserting an Item at the Beginning of the list

- make a new link
- set a new link to the old first
- Set first to new link

```
public void insertFirst(int id, dd)
{
    Link newLink = new Link(id, dd);
    newLink.next = first;
    first = newLink;
}
```

Singly-Linked List : Algorithm Basic Operation

Deleting an Item at the Beginning of the list assuming that the list is not empty.

- save reference to link
- delete it, set first to old next
- return deleted link

```
public deleteFirst()
{
    Link temp = first;
    first = first.next;
    return temp;
}
```

Take note that we used the **return** statement so that whenever an item is deleted, it would be first save to temp before it would be deleted and then return the value of **temp**.

Singly-Linked List : Algorithm Basic Operation

Display the entire list, start at first and follow the chain reference from link to link

- start at the beginning of last until end of list
- print data
- move to next link

```
public displayList()
{
    //optional
    System.out.print("List (first → last): ");
    Link current = first;
    while(current != null)
    {
        current.displayLink();
        current = current.next;
    }
    System.out.println(""); //optional
}
```

Singly-Linked List : Algorithm Basic Operation

Finding Specified Links – algorithm of deleting a link assuming a non-empty list

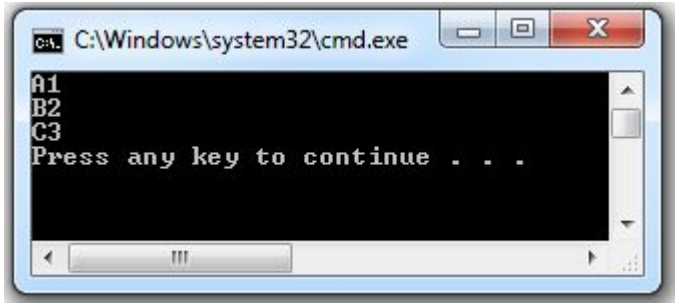
- search for link
- if not found, go to the next link
- found it
- if first link, change first
- otherwise, bypass it

```
public Link delete(int key)
{
    Link current = first;
    Link previous = first;
    while(current.iData != key)
    {
        if(current.next == null)
            return null;
        else
        {
            previous = current;
            current = current.next;
        }
    }
    if(current == first)
        first = first.next;
    else
        previous.next = current.next;
    return current;
}
```

Singly-Linked List : Other Operations

- Finding the length of the list, reading the list from left-to-right
- Retrieving the ith node in the list, where $i \leq n$
- Storing a new value into the ith position, where $i \leq n$
- Inserting a new node at position i, where $i \leq n$
- Deleting the ith element, where $i \leq n$
- Copying a list
- Sorting the nodes in the list
- Merging two or more lists
- Splitting a list into several sublists

Singly-Linked List : Display 3 Sample Nodes




```java
class ThreeNodesLinkedList {
    Node head; // head of list

    static class Node {
        String data;
        Node next;
        Node(String d)
        {
            data = d;
            next = null;
        } // Constructor
    }
        public void printList()
        {
            Node n = head;
            while (n != null) {
                System.out.println(n.data + " ");
                n = n.next;
            }
    }
    public static void main(String[] args)
    {
        ThreeNodesLinkedList llist = new ThreeNodesLinkedList();

        llist.head = new Node("A1");
        Node second = new Node("B2");
        Node third = new Node("C3");

        llist.head.next = second;
        second.next = third;

        llist.printList();
    }
}
```

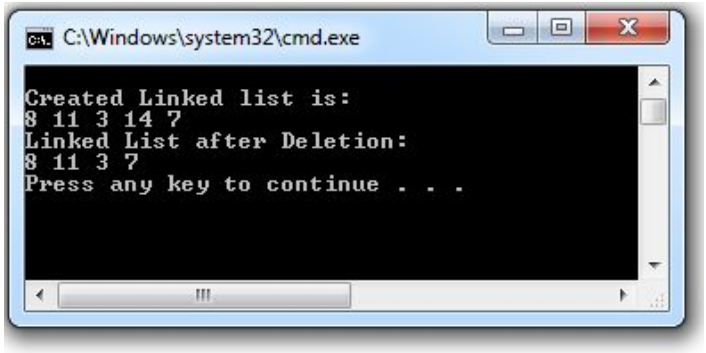
Singly-Linked List : Deleting a Node at Specified Position

```
class DelPositionLinkedList
{
    Node head; // head of list

    class Node
    {
        int data;
        Node next;
        Node(int d)
        {
            data = d;
            next = null;
        }
    }

    public void push(int new_data)
    {
        Node new_node = new Node(new_data);
        new_node.next = head;
        head = new_node;
    }

    public void printList()
        {
            Node tnode = head;
            while (tnode != null)
            {
                System.out.print(tnode.data+" ");
                tnode = tnode.next;
            }
    }
```

Singly-Linked List : Deleting a Node at Specified Position

```
void deleteNode(int position)
{
    // If linked list is empty
    if (head == null)
        return;

    // Store head node
    Node temp = head;

    // If head needs to be removed
    if (position == 0)
    {
        head = temp.next; // Change head
        return;
    }

    // Find previous node of the node to be deleted
    for (int i=0; temp!=null && i<position-1; i++)
        temp = temp.next;

    // If position is more than number of nodes
    if (temp == null || temp.next == null)
        return;

    // Node temp->next is the node to be deleted
    // Store pointer to the next of node to be deleted
    Node next = temp.next.next;

    temp.next = next; // Unlink the deleted node from list
}
```

Singly-Linked List : Deleting a Node at Specified Position




```java
    public static void main(String[] args)
    {
        /* Start with the empty list */
        DelPositionLinkedList llist = new DelPositionLinkedList();

        llist.push(7);
        llist.push(14);
        llist.push(3);
        llist.push(11);
        llist.push(8);

        System.out.println("\nCreated Linked list is: ");
        llist.printList();

        llist.deleteNode(3); // Delete node at position 4

        System.out.println("\nLinked List after Deletion: ");
        llist.printList();

        System.out.println();
    }
}
```

Doubly-Linked List

- format of a doubly-linked list

Left Pointer	Data	Right Pointer

left pointer field is used to contain the address of the preceding node in the list or what is known as **PREDECESSOR**

data field and right pointer field function in the same manner as in singly-linked list

- Doubly-linked list with 3 Nodes

Doubly-Linked List

ADVANTAGES:

- can be traversed in both forward and backward direction.
- The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
- We can quickly insert a new node before a given node.

DISADVANTAGES:

- Every node of DLL Require extra space for an previous pointer. It is possible to implement Doubly-Linked List with single pointer though.
- All operations require an extra pointer previous to be maintained.

Doubly-Linked List

Inserting a Node into a Doubly-Linked List (general procedure)

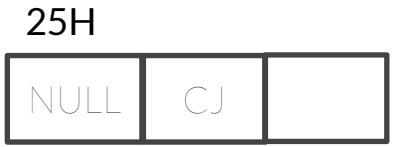
- Create a new node for the element
- Set the data field of the new node to the value to be inserted
- Determine the position of the node in the list based on it's value
- Insert the node

Doubly-linked list with 3 Nodes

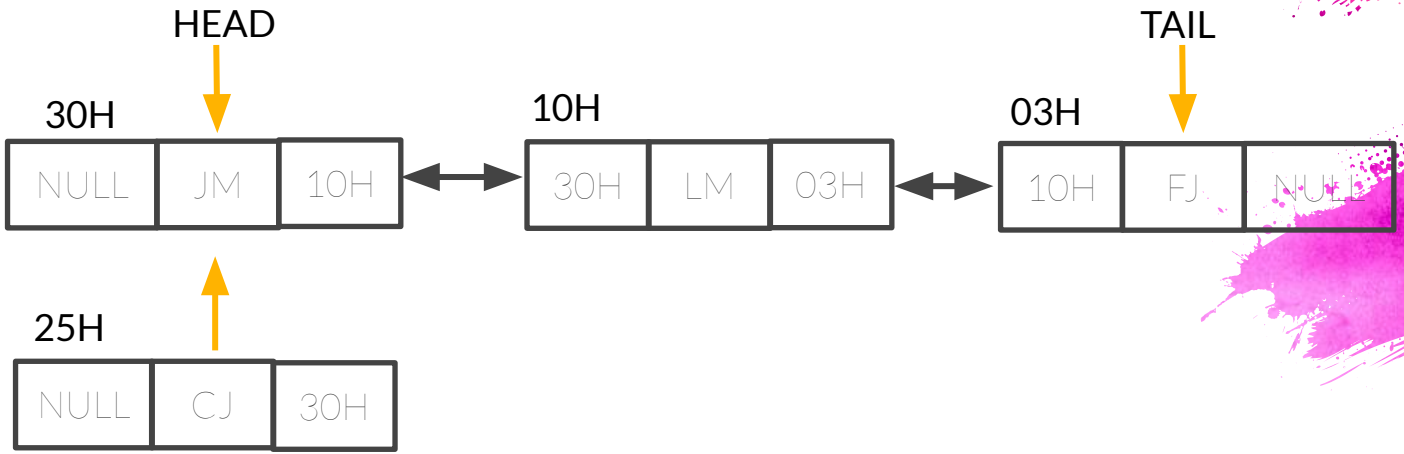
Doubly-Linked List

Inserting a Node at the HEAD of the list

New node after setting the left pointer field



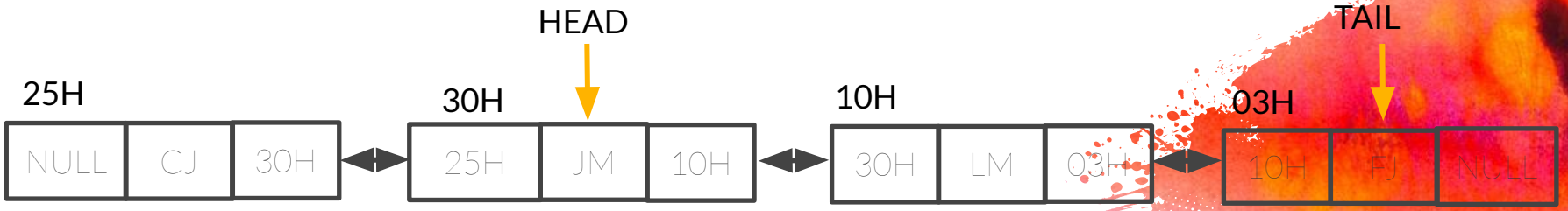
Doubly-linked list after setting the new node's right pointer field



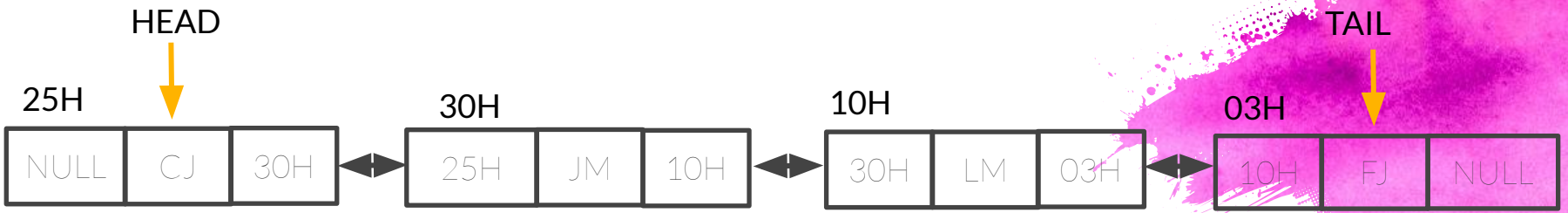
Doubly-Linked List

Inserting a Node at the HEAD of the list

Doubly-linked list after setting left pointer field of HEAD node



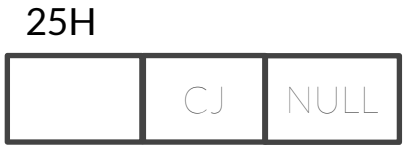
Doubly-linked list after reassigning HEAD



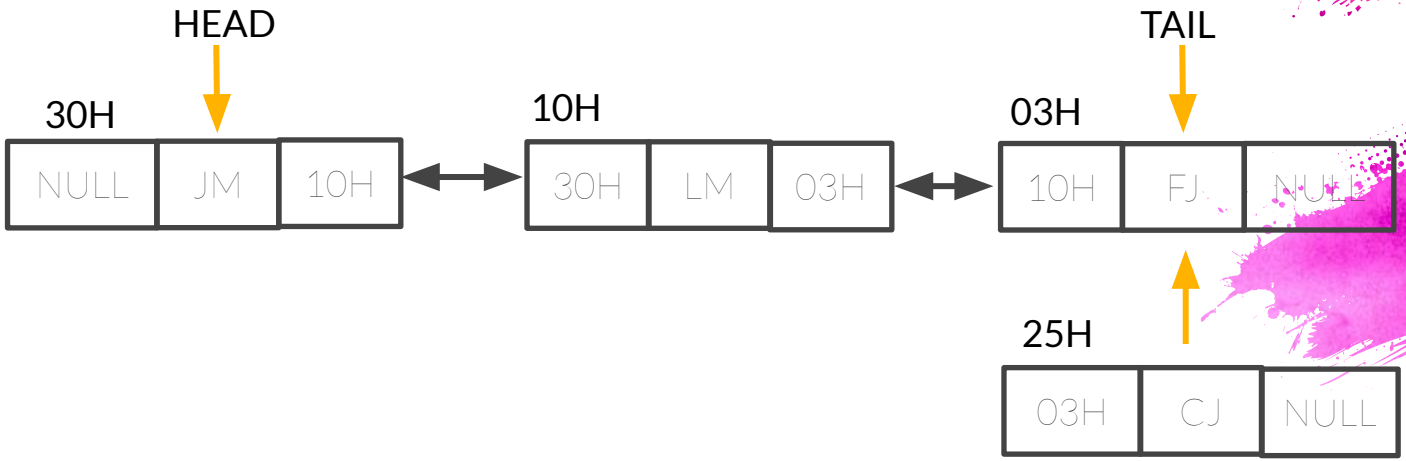
Doubly-Linked List

Inserting a Node at the END of a list

New node after setting the right pointer field



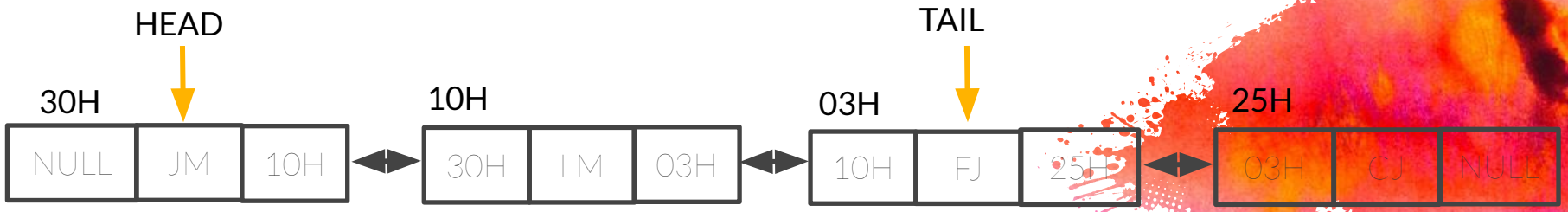
Doubly-linked list after setting the new node's left pointer field



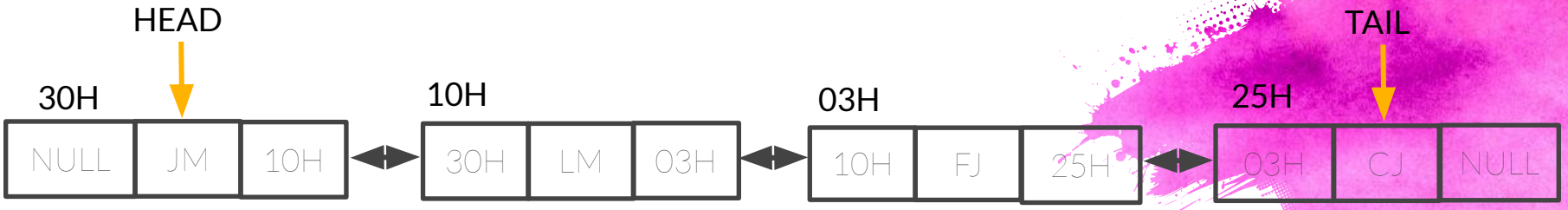
Doubly-Linked List

Inserting a Node at the END of a list

Doubly-linked list after setting right pointer field of TAIL node



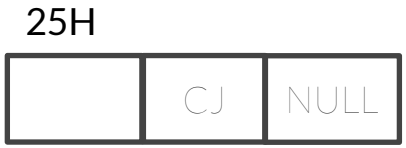
Doubly-linked list after reassigning TAIL



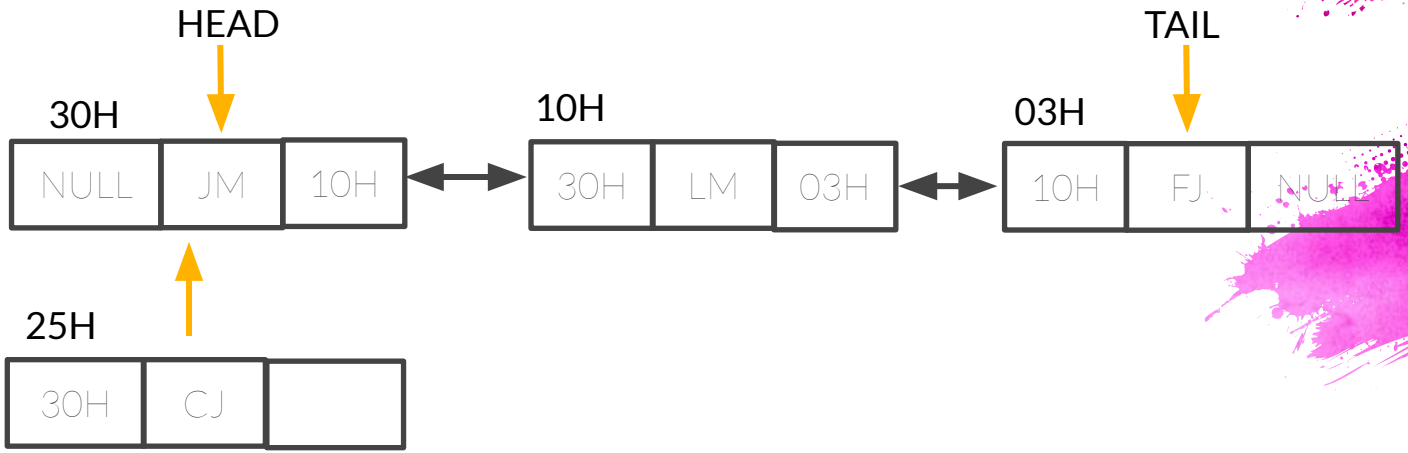
Doubly-Linked List

Inserting a Node WITHIN the List

New node



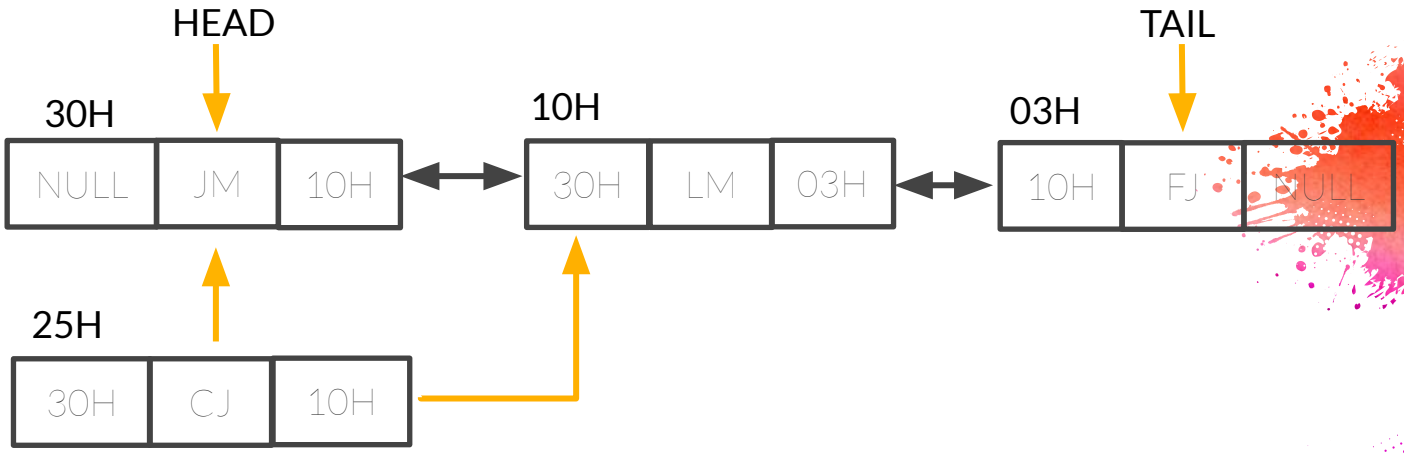
Doubly-linked list after setting the new node's left pointer field



Doubly-Linked List

Inserting a Node WITHIN the List

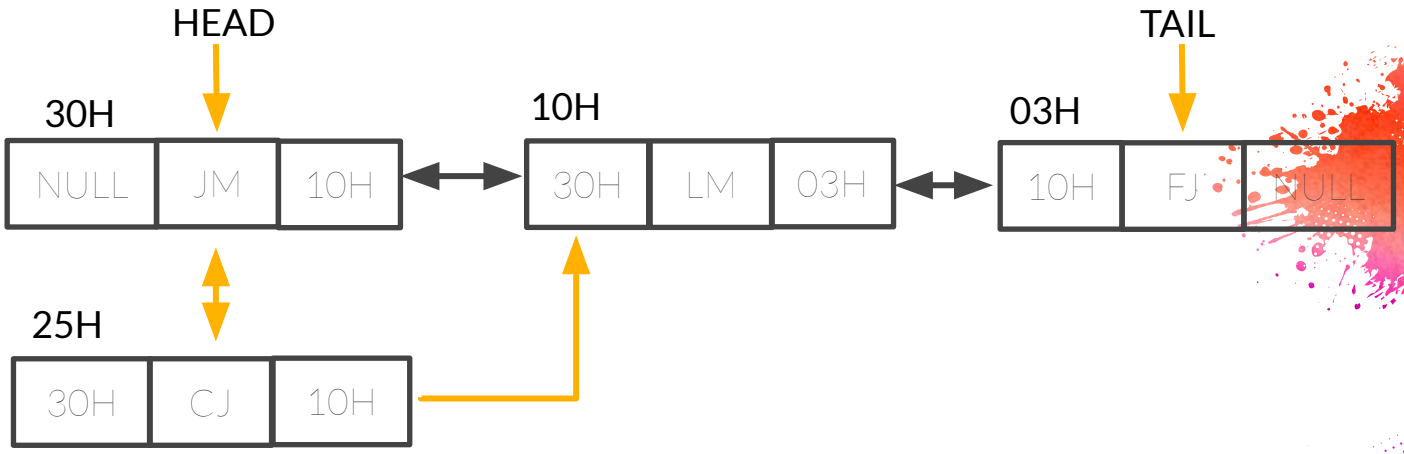
Doubly-linked list after setting the new node's right pointer field



Doubly-Linked List

Inserting a Node WITHIN the List

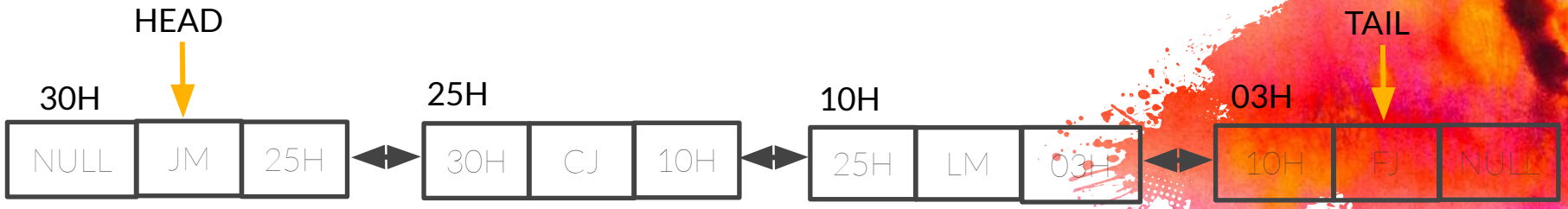
Doubly-linked list after setting the right pointer field



Doubly-Linked List

Inserting a Node WITHIN the List

Doubly-linked list after setting the left pointer field of displaced node



Doubly-Linked List

Deleting a Node from a Doubly-Linked List

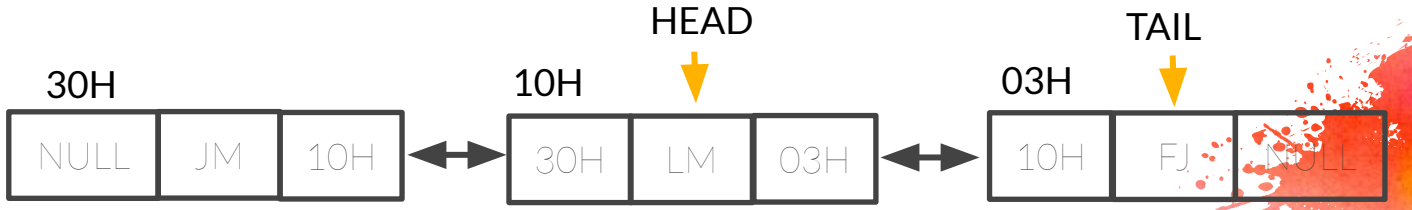
- Exactly the same as the steps in deleting a node from a singly-linked list
- Locate the node
- Delete the node
- Release the node from memory

Doubly-linked list with 3 Nodes

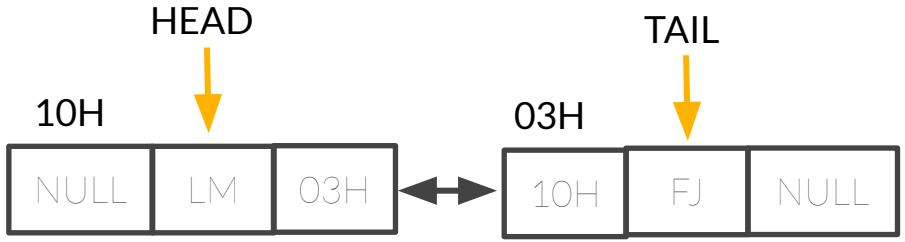
Doubly-Linked List

Deleting the Node at the HEAD of the list

Doubly-linked list after reassigning HEAD



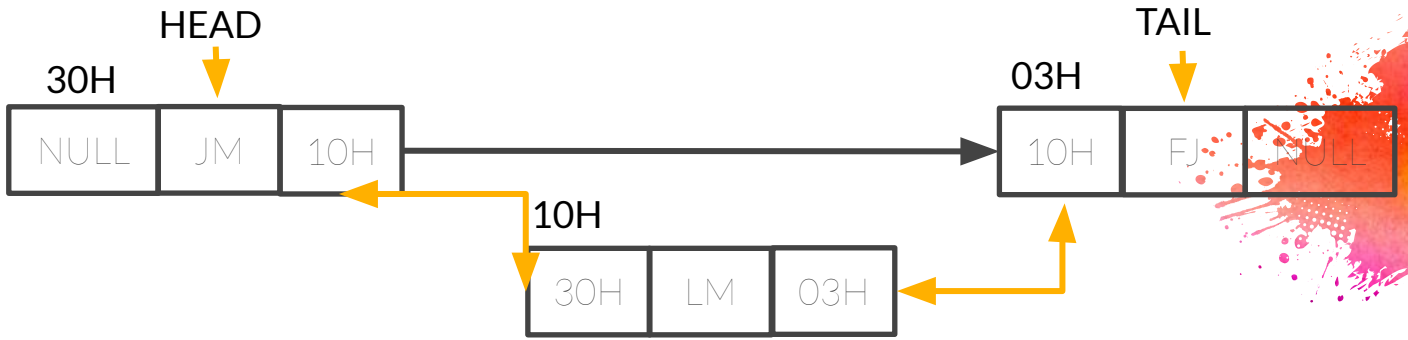
Doubly-linked list setting the left pointer field of node "LM"



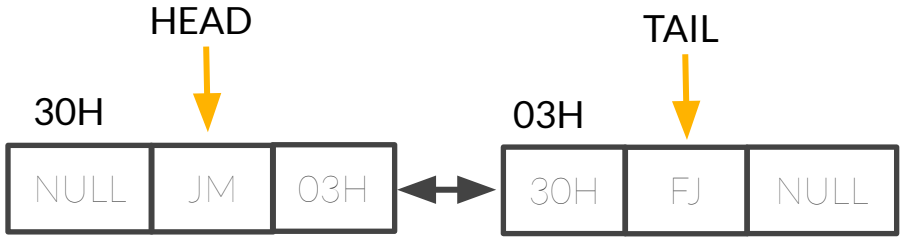
Doubly-Linked List

Deleting a Node WITHIN the list

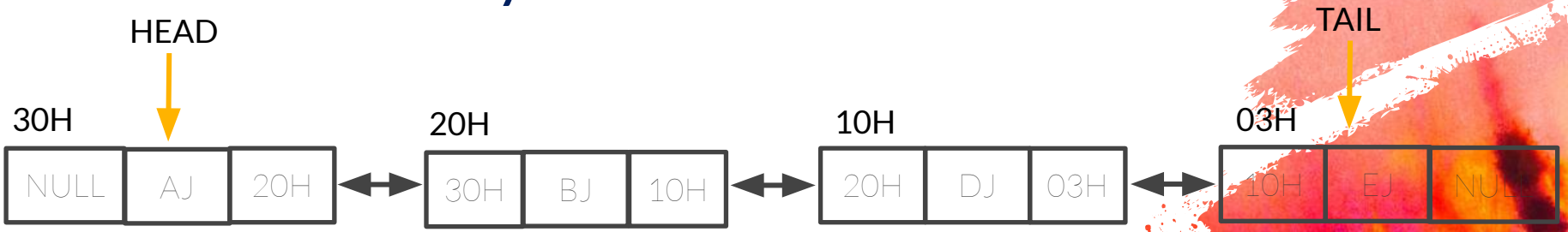
Doubly-linked list setting the right pointer field of node "JM"



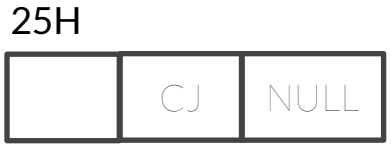
Doubly-linked list setting the left pointer field of node "FJ"



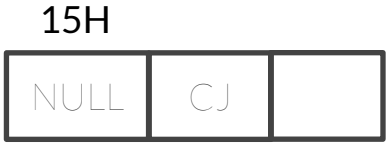
Exercise: Doubly-Linked List



- Insert a Node at Position 2



- Delete TAIL of the list
- Insert the node at the HEAD of the list



Doubly-Linked List : Deleting a Node at the Tail of the List

```java
public class DeleteEndDLL {

    //Represent a node of the doubly linked list

    class Node{
        int data;
        Node previous;
        Node next;

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

     //display() will print out the nodes of the list
        public void display() {
            //Node current will point to head
            Node current = head;
            if(head == null) {
                System.out.println("List is empty");
                return;
            }
            while(current != null) {
                //Prints each node by incrementing the pointer.

                System.out.print(current.data + " ");
                current = current.next;
            }
            System.out.println();
    }
```

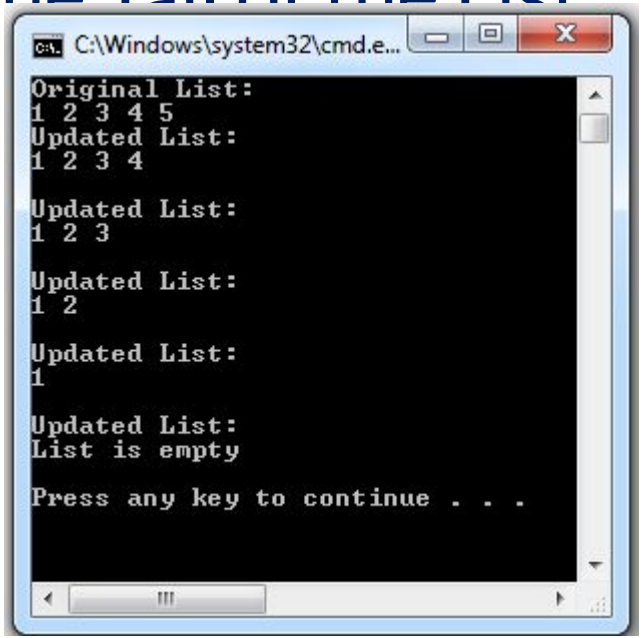
Doubly-Linked List : Deleting a Node at the Tail of the List

```
//Represent the head and tail of the doubly linked list
Node head, tail = null;

//addNode() will add a node to the list
public void addNode(int data) {
    //Create a new node
    Node newNode = new Node(data);

    //If list is empty
    if(head == null) {
        //Both head and tail will point to newNode
        head = tail = newNode;
        //head's previous will point to null
        head.previous = null;
        //tail's next will point to null, as it is the last node of the list
        tail.next = null;
    }
    else {
        //newNode will be added after tail such that tail's next will point to newNode
        tail.next = newNode;
        //newNode's previous will point to tail
        newNode.previous = tail;
        //newNode will become new tail
        tail = newNode;
        //As it is last node, tail's next will point to null
        tail.next = null;
    }
}
```

Doubly-Linked List : Deleting a Node at the Tail of the List

```
//deleteFromEnd() will delete a node from the end of the list
public void deleteFromEnd() {
    //Checks whether list is empty
    if(head == null) {
        return;
    }
    else {
        //Checks whether the list contains only one node
        if(head != tail) {
            //Previous node to the tail will become new tail
            tail = tail.previous;
            //Node next to current tail will be made null
            tail.next = null;
        }
        //If the list contains only one element
        //Then it will remove node and now both head and tail will point to null
        else {
            head = tail = null;
        }
    }
}
```

Doubly-Linked List : Deleting a Node at the Tail of the List



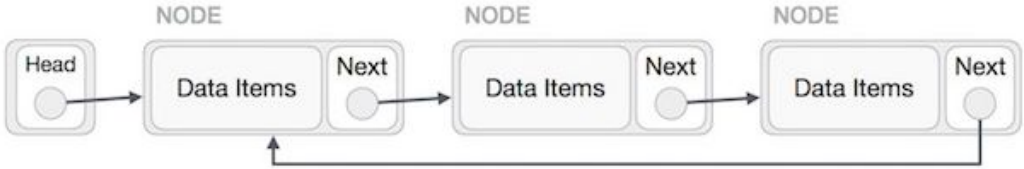

```java
    public static void main(String[] args) {

        DeleteEndDLL dList = new DeleteEndDLL();
        //Add nodes to the list
        dList.addNode(1);
        dList.addNode(2);
        dList.addNode(3);
        dList.addNode(4);
        dList.addNode(5);

        //Printing original list
        System.out.println("Original List: ");
        dList.display();

        while(dList.head != null) {
            dList.deleteFromEnd();
            //Printing updated list
            System.out.println("Updated List: ");
            dList.display();

            System.out.println();
        }
    }
}
```

Circular Linked List

- is a linked list where all nodes are connected to form a circle.
- There is no NULL at the end.

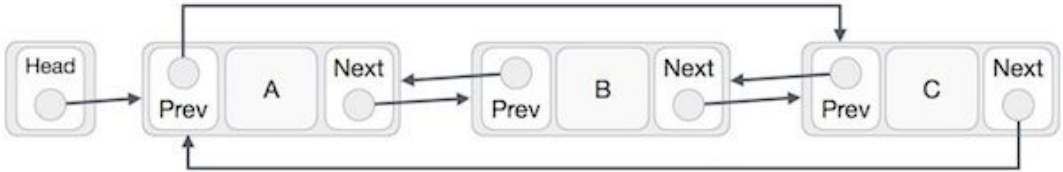
SINGLY-Linked List as Circular

- The next pointer of the last node points to the first node.



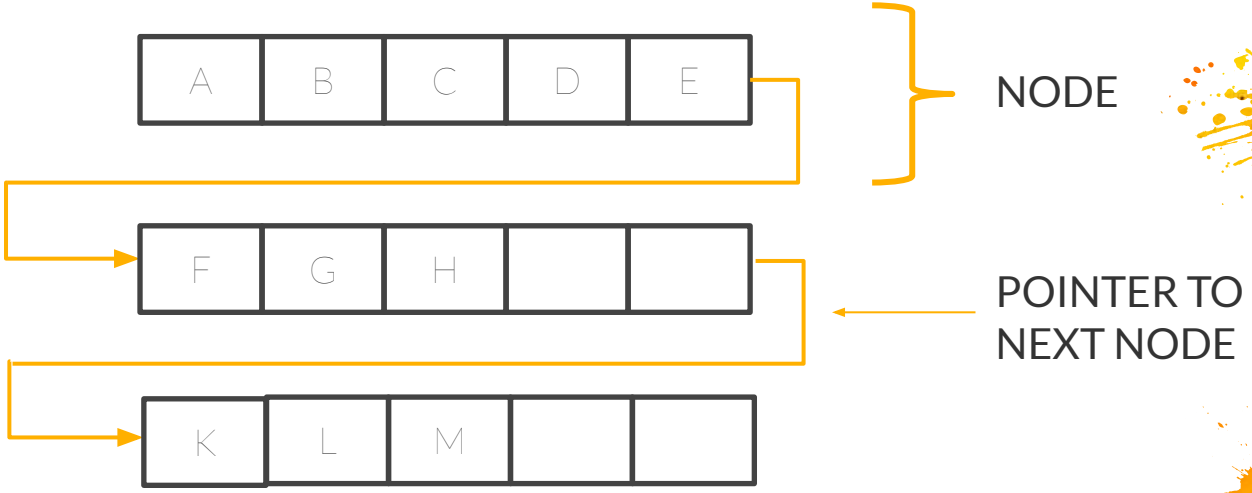
DOUBLY-Linked List as Circular

- 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.



Unrolled Linked List

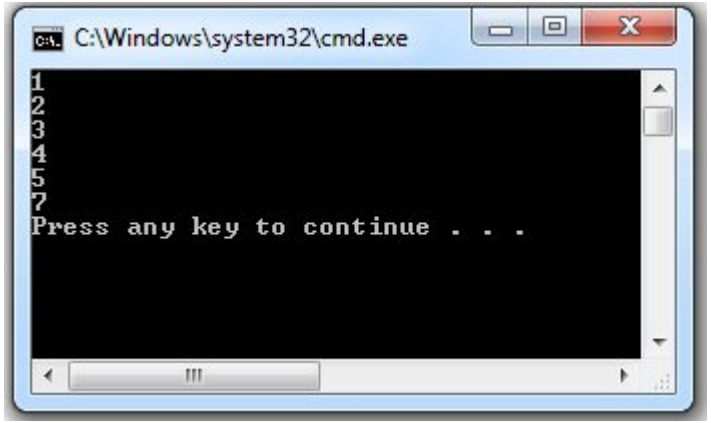
- is a also a linear structure and is a variant of linked list
- it stores multiple elements at each node



Unrolled Linked List : Traverse and display three (3) nodes

```java
import java.util.*;

class UnrolledLL {

static final int maxElements = 4;

// Unrolled Linked List Node
static class Node
{
    int numElements;
    int []array = new int[maxElements];
    Node next;
};

static void printUnrolledList(Node n)
{
    while (n != null)
    {

        // Print elements in current node
        for(int i = 0; i < n.numElements; i++)
            System.out.print(n.array[i] + " ");

        // Move to next node
        n = n.next;
    }
}
```

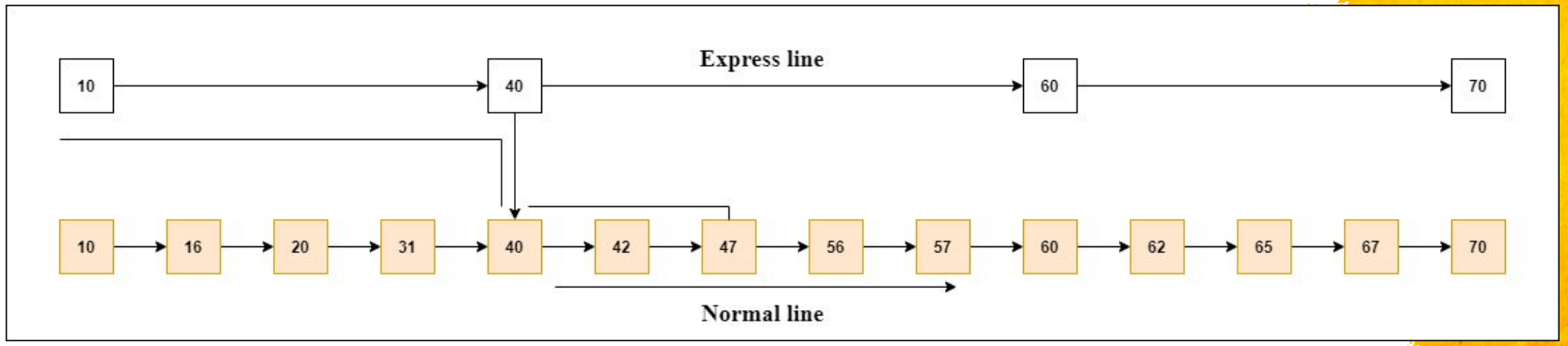
Unrolled Linked List : Traverse and display three (3) nodes




```
public static void main(String[] args)
{
    Node head = null;
    Node second = null;
    Node third = null;

    head = new Node();
    second = new Node();
    third = new Node();

    head.numElements = 3;
    head.array[0] = 1;
    head.array[1] = 2;
    head.array[2] = 3;

    head.next = second;

    second.numElements = 2;
    second.array[0] = 4;
    second.array[1] = 5;

    second.next = third;

    third.numElements = 1;
    third.array[0] = 7;

    third.next = null;

    printUnrolledList(head);
}
}
```

Skip List

- uses probability to build subsequent layers of linked lists upon an original linked list. Each additional layer of links contains fewer elements, but no new elements.



- The **lower layer** is a common line that links all nodes, and the **top layer** is an express line that links only the main nodes

Note: Once you find a node like this on the "express line", you go from this node to a "normal lane" using a pointer, and when you search for the node in the normal line.

Skip List

ADVANTAGES:

- If you want to insert a new node in the skip list, then it will insert the node very fast because there are no rotations in the skip list.
- The skip list is simple to implement as compared to the hash table and the binary search tree.
- It is very simple to find a node in the list because it stores the nodes in sorted form.
- The skip list algorithm can be modified very easily in a more specific structure, such as indexable skip lists, trees, or priority queues.
- The skip list is a robust and reliable list.

DISADVANTAGES:

- It requires more memory than the balanced tree.
- Reverse searching is not allowed.
- The skip list searches the node much slower than the linked list.

RIZAL TECHNOLOGICAL UNIVERSITY
COLLEGE OF ENGINEERING ARCHITECTURE AND TECHNOLOGY

Thank You 😊
Keep safe
and God bless!