Chapter 20 - Part 2

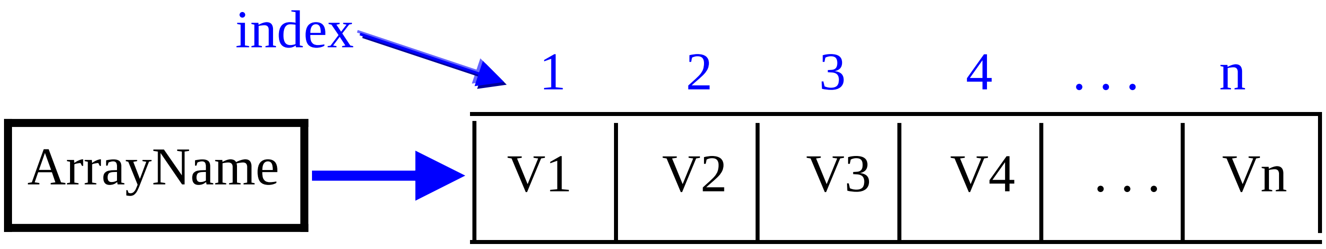
Lists

Array Concept

- For example, an ***array*** is a data structure that holds a sequence of values of the same type.
- Each value is referenced (accessed) by its location (index value). For example,

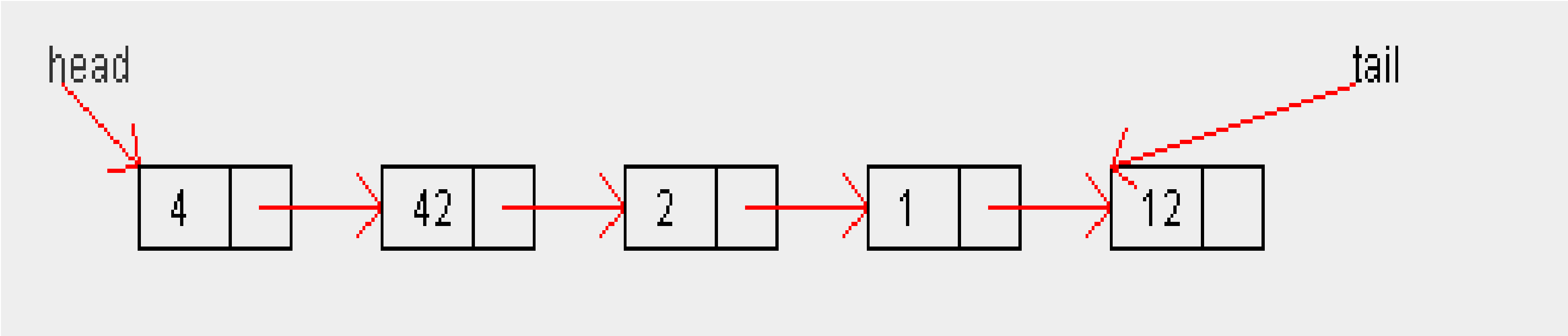
```
print ArrayName[2];
ArrayName[3] = 100;
```

- Array operations allows us to manipulate the array, such as find the size, copy the array, store values, retrieve values, and modify values.



Linked-List Concept - 1

- A *linked-List* in a chain of nodes such that each node has a data field of a given type and a link field of type pointer.
- The data filed store user data; while the link filed stores the memory address of the next node in the chain.
- The last node's link field hold a NULL pointer.



Linked-List Concept - 2

A node is a record structure (pseudo code):

```
Node_Type is Record
   data : data_type
   next : pointer_type; //pointer type (mem. address)
End Record


//Java syntax
Class Node<E> //E is valid data type in Java
{
  E    element;
  Node next;   //self-referencing

  Public Node(E e)
  {
     element = e;
  }
}
```

Linked-List Concept - 3

Logical steps for working with pointers:

1. define the node type (record structure)
2. define pointer variables (initially, null pointer, no memory space)
3. allocate space to pointer variables
4. initialize data fields (memory space) of each node
5. link the nodes to form the linked-list
6. manipulate the linked-list nodes using linked-list operations.

List Data Structure

A list is a popular data structure to store data in sequential order. For example, a list of students, a list of available rooms, a list of cities, and a list of books, etc. can be stored using lists.

The common operations on a list are usually the following:

- Retrieve an element from the list
- Insert a new element to the list
- Delete an element from the list
- Find how many elements are in the list
- Find if an element is in the list
- Check if the list is empty
- Others...

Two Ways to Implement Lists

There are two ways to implement a list.

1. Array: The array is dynamically created. If the capacity of the array is exceeded, create a new larger array and copy all the elements from the current array to the new array.

2. Linked-list: A linked structure consists of nodes. Each node is dynamically created to hold a data element. All the nodes are linked together to form the list.

Array Implementation

Initially, an array, say myList of Object[] type, is created with a default size. When inserting a new element into the array, first ensure there is enough room in the array. If not, create a new array with the size as twice as the current one. Copy the elements from the current array to the new array. The new array now becomes the current array.

Array List Illustration

Insert Operations: (insert to end of the list)

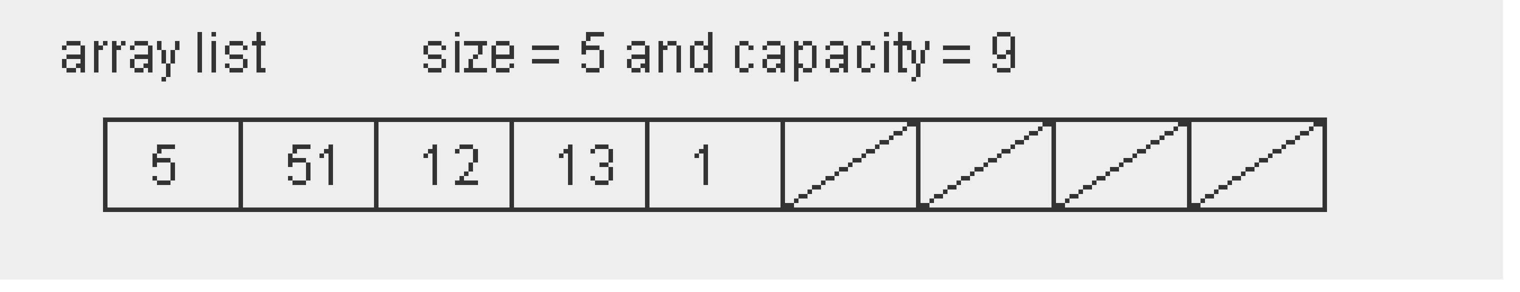
Insert 5;

Insert 51;

Insert 12;

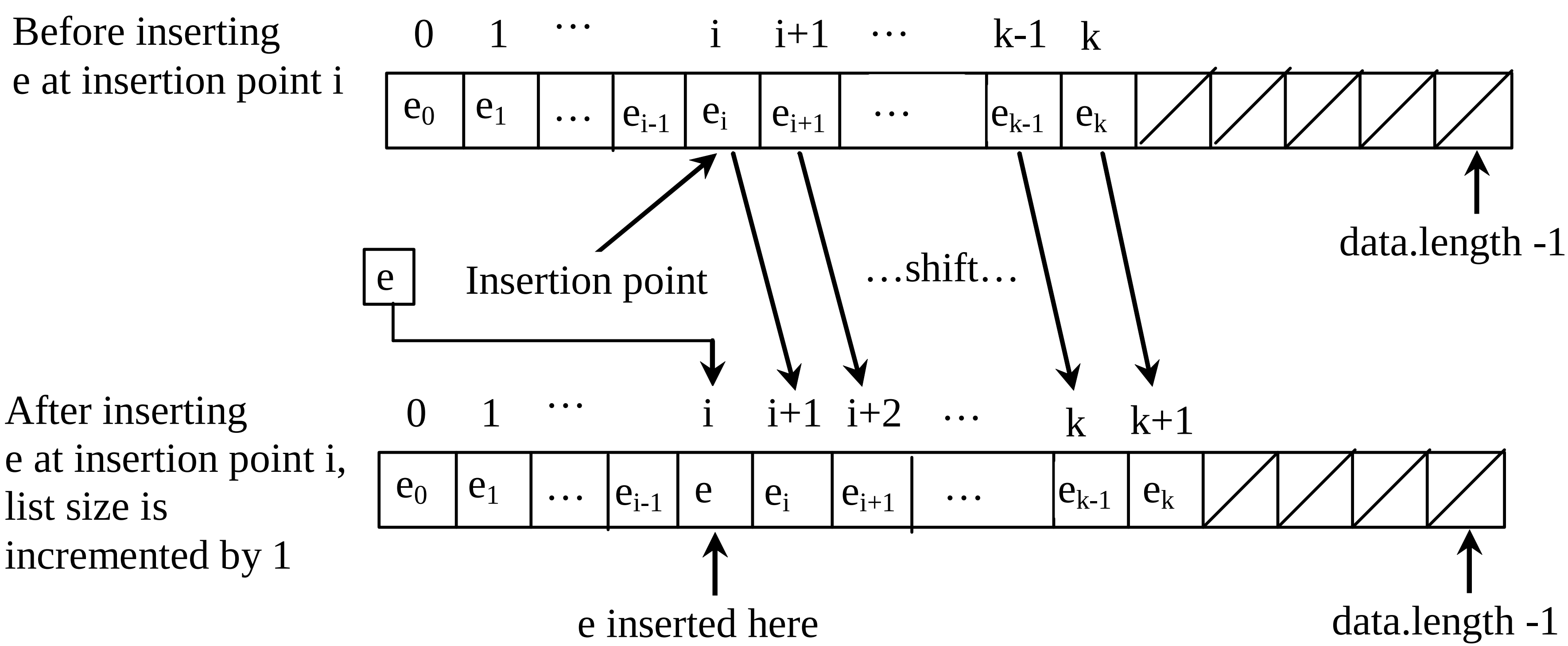
Insert 13;

Insert 1;



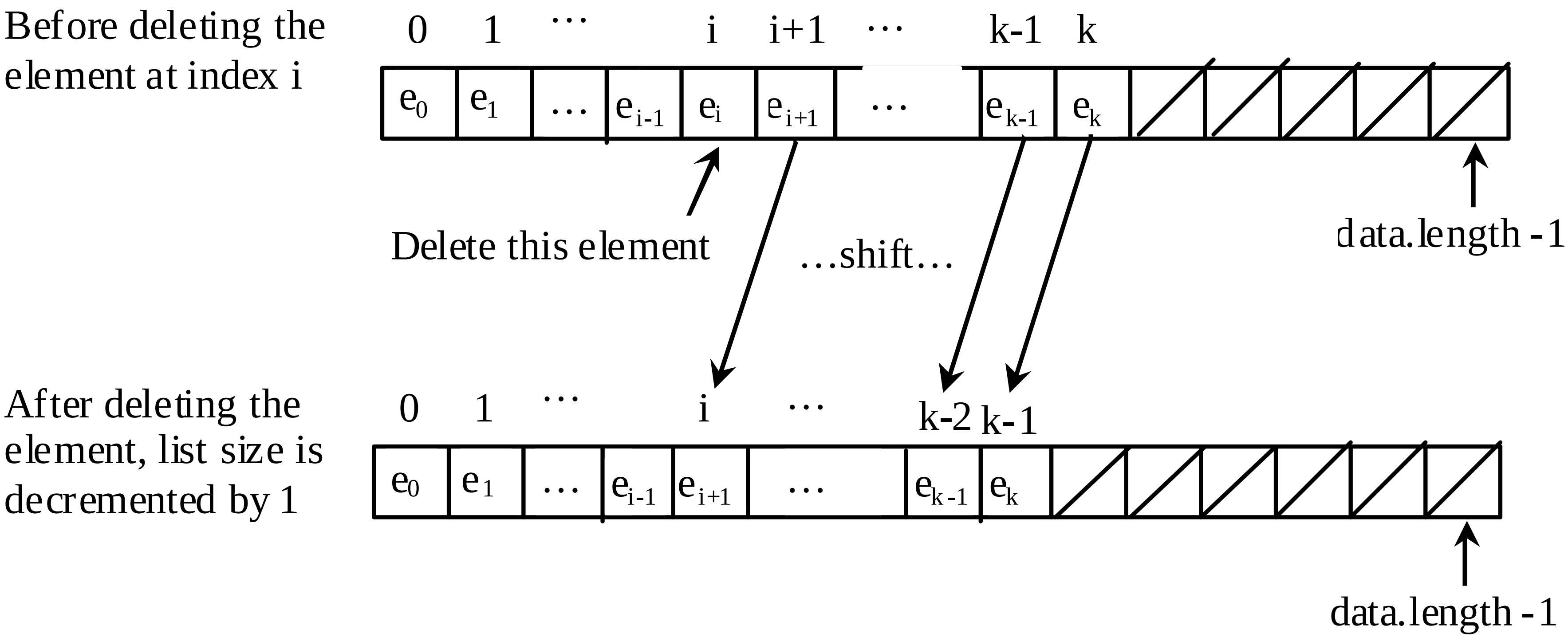
Insertion at Specific Index

Before inserting a new element at a specified index, shift all the elements after the index to the right and increase the list size by 1.



Deletion from Specific Index

To remove an element at a specified index, shift all the elements after the index to the left by one position and decrease the list size by 1.



Other Array List Operations

Think of how would you implement other operations such as:

- Retrieve an element at specific index
- Insert a new element as first element
- Delete first element from the list
- Delete last element from the list
- Determine the list size (number of elements)
- Find if an element is in the list or not
- Check if the list is empty

Exercise: Assume we have class ArrayList with an array data field and the above methods. Implement selected methods for practice.

Linked-List Implementation

In the array implementation, operations add(int index, Object o) and remove(int index) are inefficient because they require shifting potentially a large number of elements.

You can use a linked structure to implement a list to improve efficiency for adding and removing an element anywhere in a list.

Linked-List Operations

Linked-List Operations:

- Creates a default linked list (empty list)
- Add a node to the list (end of list by default)
- Add a node to the beginning of the linked list (new head node)
- Add a node at a specific position in the list
- Return the data in the first node in the list
- Return the data in the last node in the list
- Return the data in a node at a specific position in the list
- Remove the first node from the list
- Remove the last node from list
- Remove the node at a specific position in the list
- Determine the size of the list (number of nodes)
- Search the list for a data element
- Others…

Linked-List Illustration

Insert Operations: (insert to end of the list)

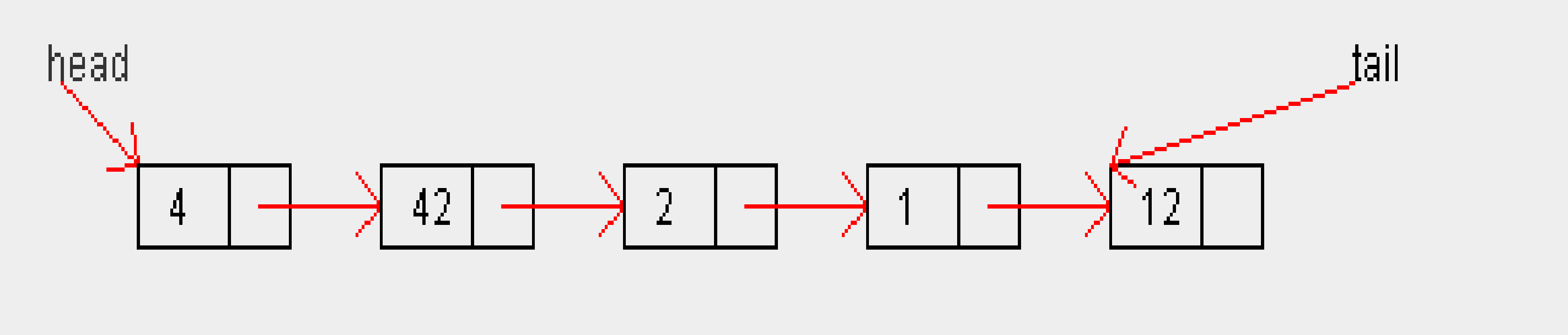
Insert 4;

Insert 42;

Insert 2;

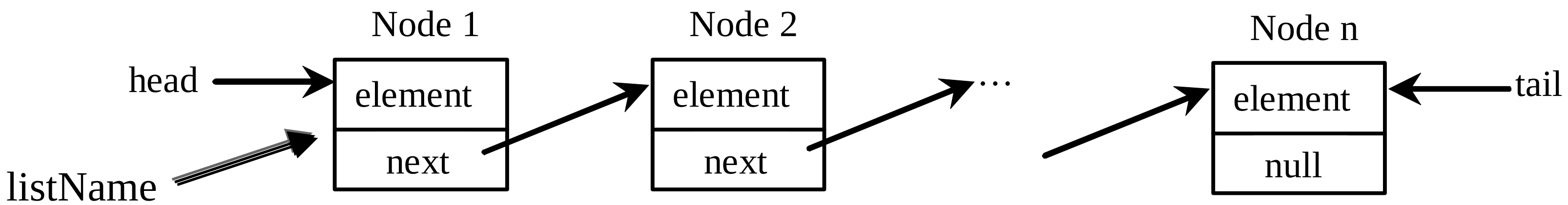
Insert 1;

Insert 12;



Nodes in Linked-Lists

A linked list consists of nodes. Each node contains an element, and each node is linked to its next neighbor. Thus a node can be defined as a class, as follows:




```
//Java syntax
Class Node<E> //E is valid data type in Java
{
  E    element;
  Node next;   //self-referencing

Public Node(E e) { element = e;} //constructor
}
```

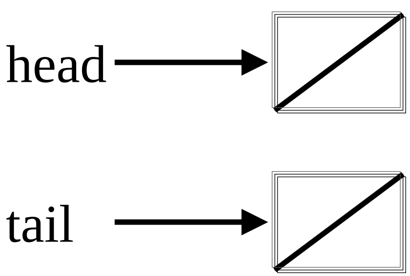
Adding Three Nodes

The variable head refers to the first node in the list, and the variable tail refers to the last node in the list. If the list is empty, both are null. For example, you can create three nodes to store three strings in a list, as follows:

Step 1: Declare head and tail: (create empty list)

```
Node<String> head = null;
Node<String> tail = null;
```

The list is empty



Pointer head holds null value.

Pointer tail holds null values.

Adding Three Nodes, cont.

Step 2: Create the first node and insert it to the list:

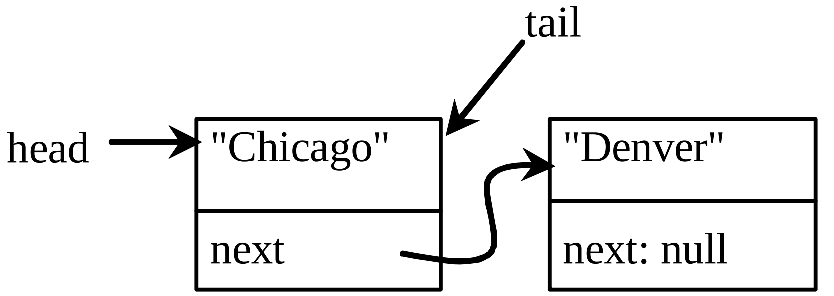
```
head = new Node<String>("Chicago");
tail = head;
```

After the first node is inserted
inserted

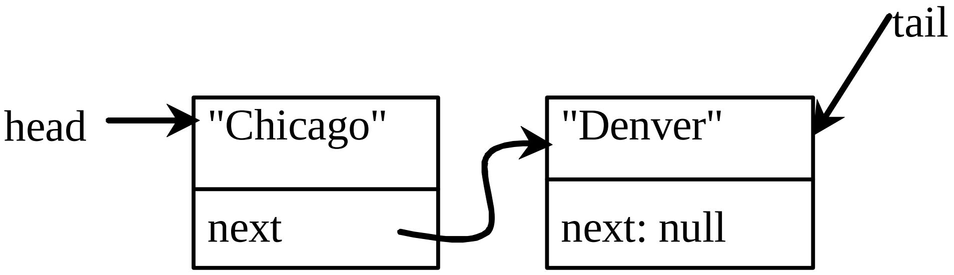
Adding Three Nodes, cont.

Step 3: Create the second node and insert it to the list:

`tail.next = new Node<String>("Denver");`



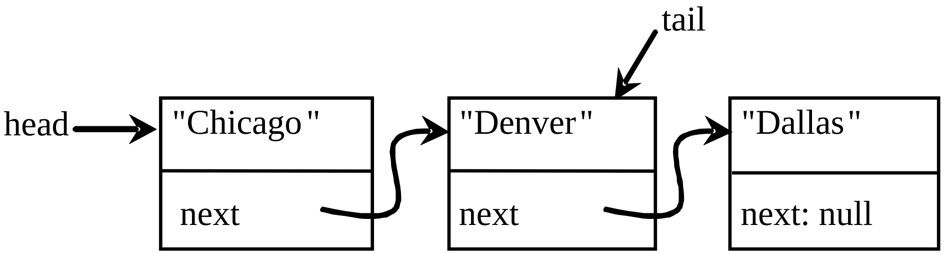
`tail = tail.next;`



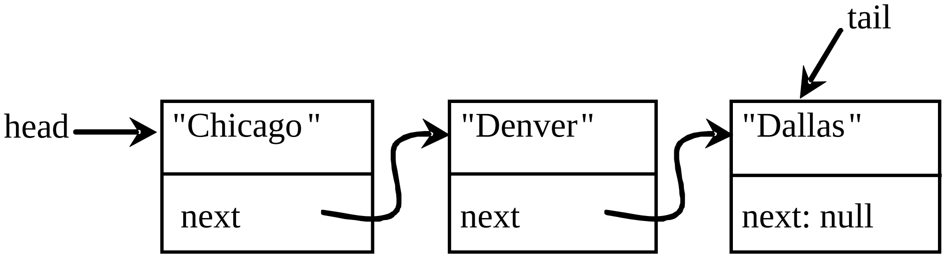
Adding Three Nodes, cont.

Step 4: Create the third node and insert it to the list:

```
tail.next =
  new Node<String>("Dallas");
```




```
tail = tail.next;
```



Method InsertLastNode(E e)

The method inserts a new node to the end of the list.

```
//Pseudo code - Method InsertLastNode(e)
Method InsertLastNode(E e)
{
  // 1. create and initialize a new bode
  Node<E> newNode = new Node<E>(e);

  // 2. check if empty list
  if (Head == NULL)
      Head = newNode;
      Tail = newNode;

  // 3. Generic case, add to end of list
  Tail.next = newNode;
  Tail = Tail.next;
}
```

Method InsertFirstNode(E e)

The method inserts a new node to the beginning of the list.

```
//Pseudo code - Method InsertFirstNode(e)
Method InsertFirstNode(E e)
{
  // 1. create and initialize a new bode
  Node<E> newNode = new Node<E>(e);

  // 2. check if empty list
  if (Head == NULL)
      Head = newNode;
      Tail = newNode;

  //3. Generic case, add as head node
  newNode.next = Head;
  Head = newNode;
}
```

Traversing all Elements in the List

Each node contains field *next* that points to the next element. If the node is the last in the list, field next contains value null.

We can use this property to detect the last node. For example, one may write the following loop to traverse (visit and process) all the nodes in the list.

```
// printout the list
Node<E> Current; // declare a pointer variable, named Current
Current = Head;  // set current to Head
while (Current != NULL)
{
  System.out.println(Current.element); //process current node
  Current = Current.next; //Move pointer Current to next node
}
```

Homework Problem

Write a method (called *CopyList()*) to duplicate a list.

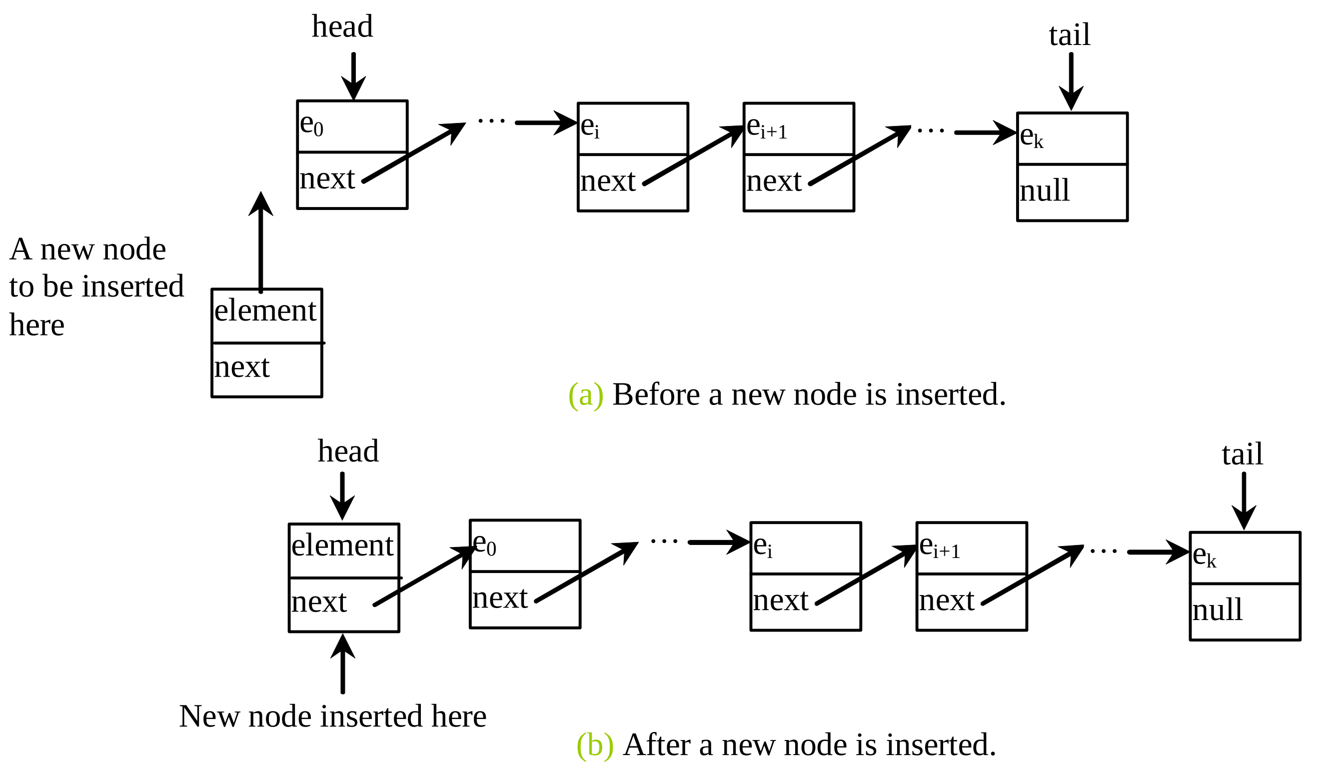
```
//Pseudo code.  Assume the new list has Head2 and Tail2 pointers.
Method InsertFirstNode(E e)
{
   Create and set Head2 and Tail2 to Null.
//special case 1:  If list is empty
   Do nothing.

//special case 2: list size = 1 (one node), No loop needed
   Create a new node
   Populate it
   Set Head2 and Tail2 to the new node
   Increase new list size by 1

//general case: list has 2 or more nodes. Need a loop
   Create first node, populate it
   Set Head2 and Tail2 to the new node
   Need to traverse (loop through) original list using temp pointer
   Create a new node for each mode temp points at
   Populate the new node and link to the end of new list
   Adjust pointer Tail2 to point to the newly added node
   Increase new list size by 1
```

MyLinkedList Class - Textbook

MyLinkedList<E>	
-head: Node<E>	
-tail: Node<E>	
+MyLinkedList()	Creates a default linked list.
+MyLinkedList(objects: E[])	Creates a linked list from an array of objects.
+addFirst(e: E): void	Adds the object to the head of the list.
+addLast(e: E): void	Adds the object to the tail of the list.
+getFirst(): E	Returns the first object in the list.
+getLast(): E	Returns the last object in the list.
+removeFirst(): E	Removes the first object from the list.
+removeLast(): E	Removes the last object from the list.

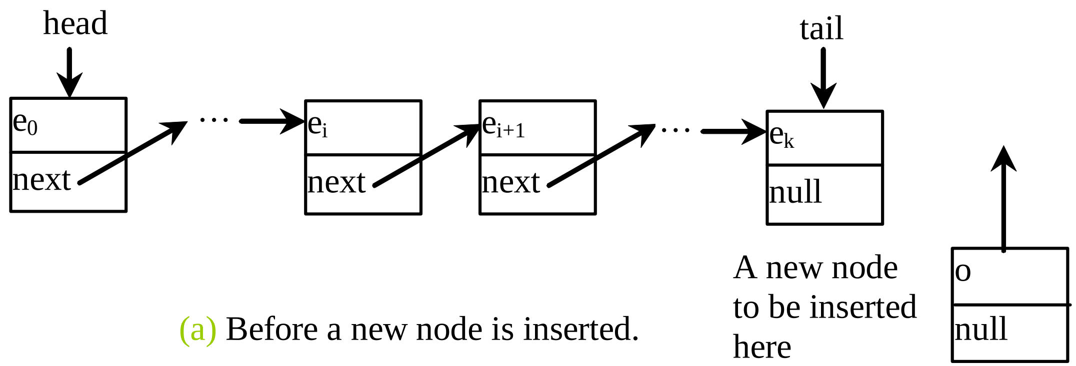
Implementing addFirst(E o)

```
public void addFirst(E o) {
  Node<E> newNode = new Node<E>(o);//create new node
  newNode.next = head; //link new node as first node
  head = newNode;  //head points to new node
  size++;  //increase list size by one element
  if (tail == null) tail = head; //if empty list
}
```

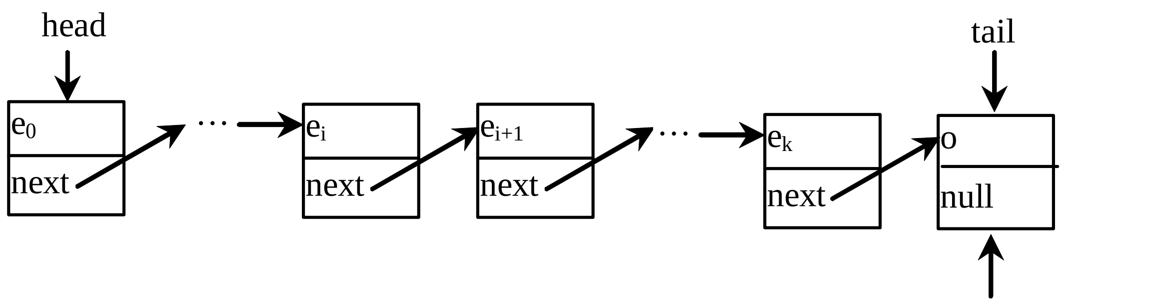


Implementing addLast(E o)

```
public void addLast(E o) {
  if (tail == null) { head = tail = new Node<E>(o);}//empty list
  else {
        tail.next = new Node(o); //link new node as last node
        tail = tail.next; //make tail pointer points to last node
       }
  size++; //increase list size by one element
}
```

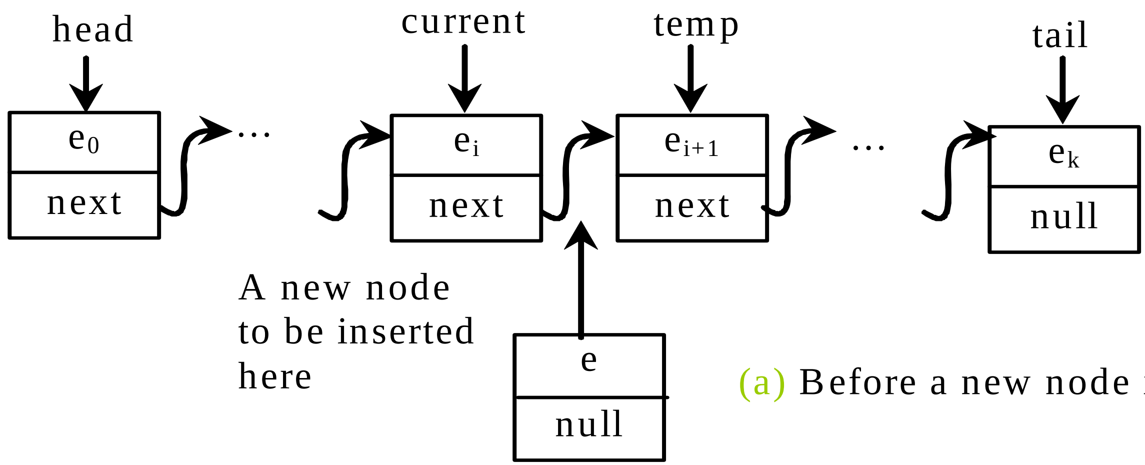


(a) Before a new node is inserted.

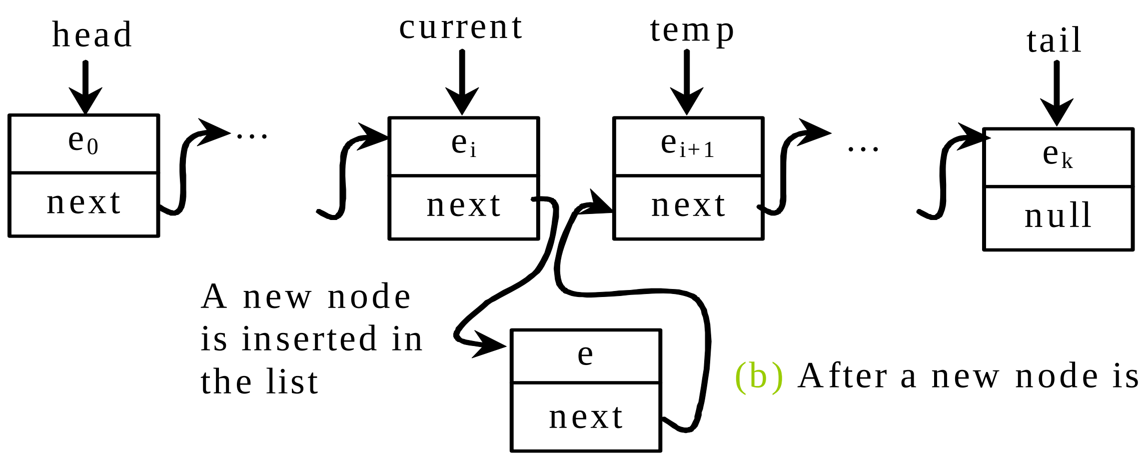


(b) After a new node is inserted.

Implementing add(int index, E o)



(a) Before a new node is inserted.



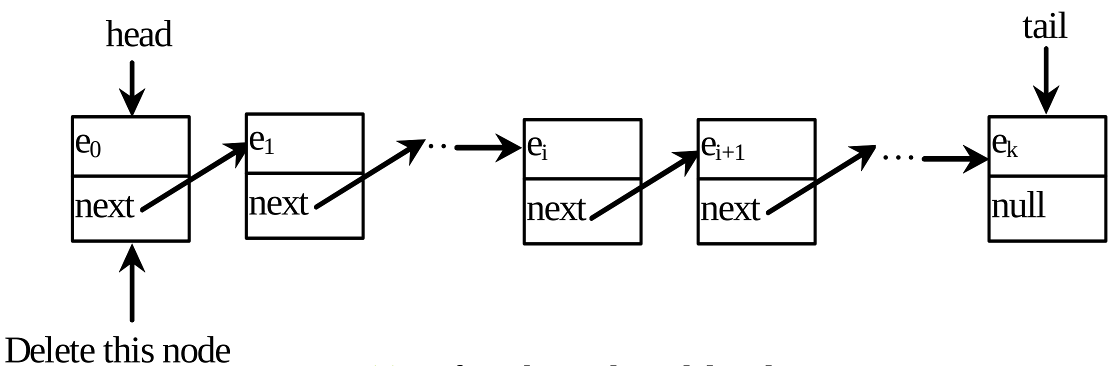
(b) After a new node is inserted.

Implementing add(int index, E o)

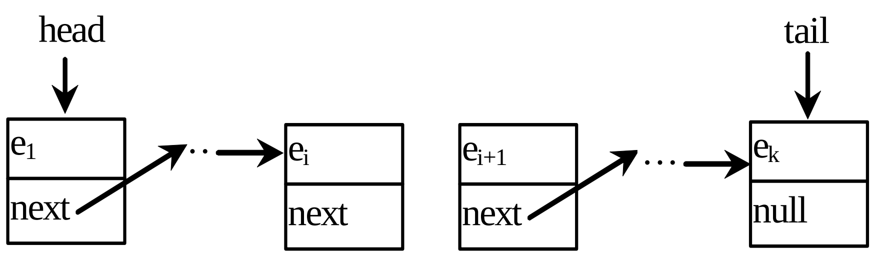
```
public void add(int index, E o)
{
  if (index == 0) //add as first node
     addFirst(o);
  else if (index >= size) //add as last node
          addLast(o);
  else
    { //move pointer current to the correct position
      Node<E> current = head;
      Node<E> temp = head.next; //temp is one step ahead of current
      for (int i = 1; i < index; i++){
          current = current.next; //advance pointer current one node
          temp = temp.next; //advance pointer temp one node
      }
      //link the new node to the list
      Node<E> temp = current.next;
      current.next = new Node<E>(o); //create new node with o object
      (current.next).next = temp;
      size++; //increase list size by one element
    }
}
```

Implementing removeFirst()

```
public E removeFirst() {
  if (size == 0) return null; //empty list
  else {
    Node<E> temp = head; //temp points to head node
    head = head.next; //head points to second node
    size--;  //decrease list size by one element
    if (head == null) tail = null;
    return temp.element;
  }
}
```

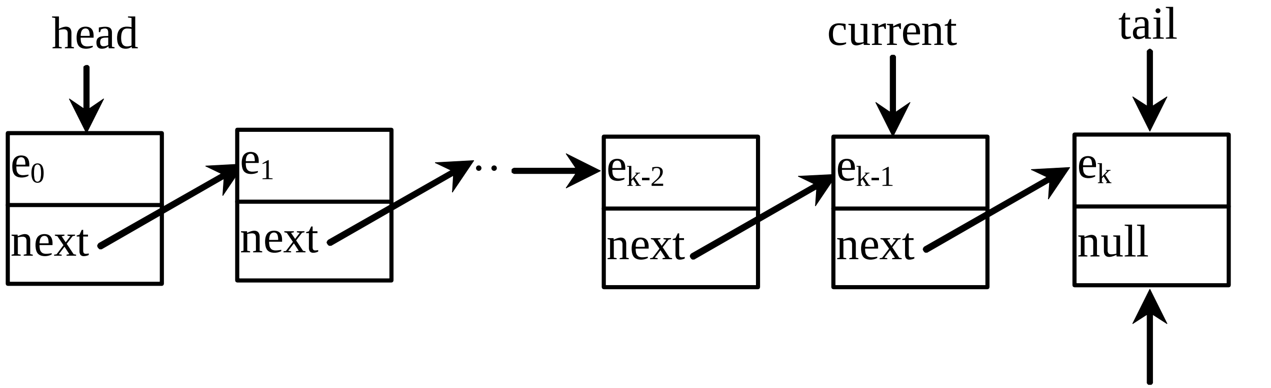


(a) Before the node is deleted.

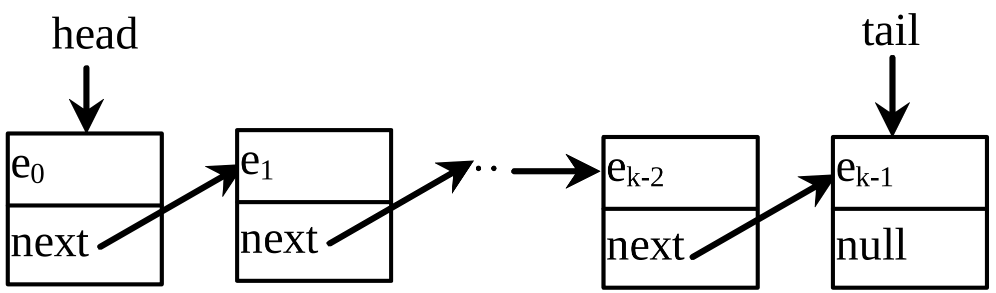


(b) After the first node is deleted

Implementing removeLast()



(a) Before the node is deleted.

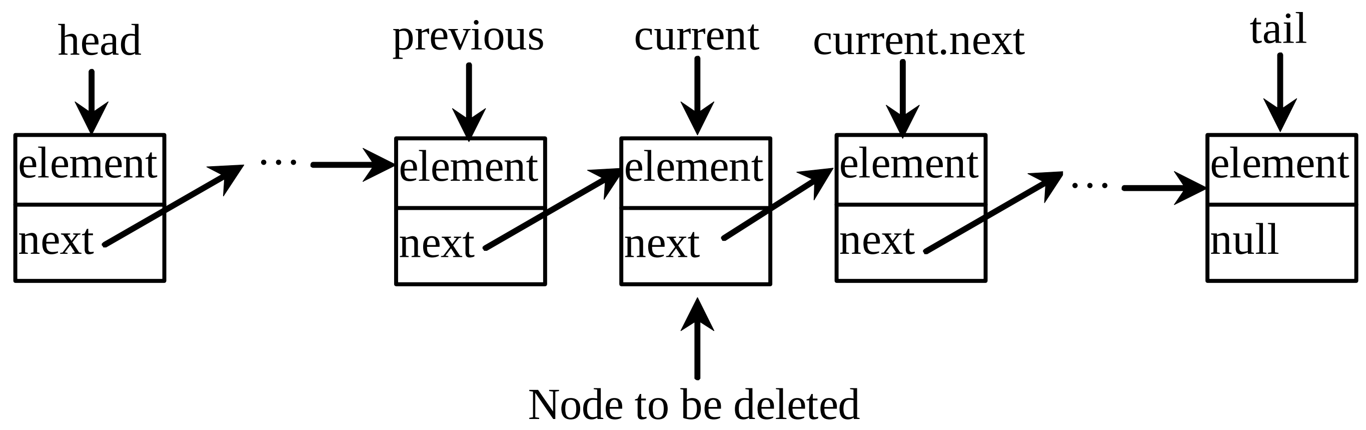


(b) After the last node is deleted

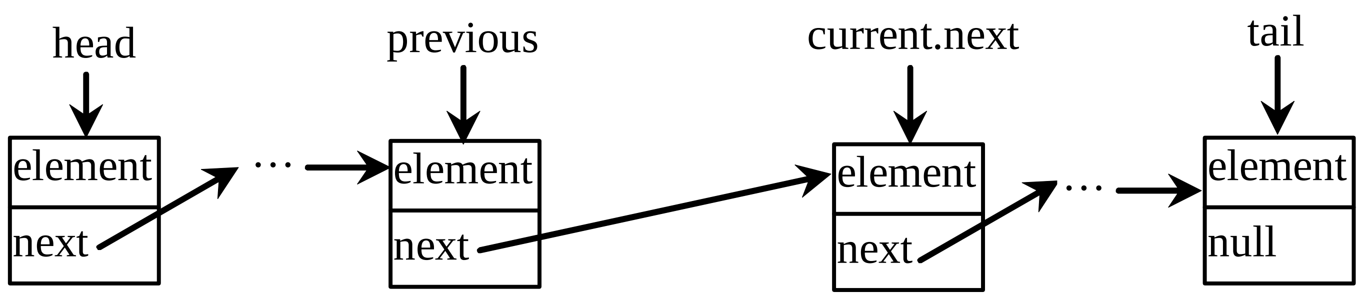
Implementing removeLast()

```
public E removeLast() {
  if (size == 0) return null; //empty list
  else if (size == 1) //one node list
  {
    Node<E> temp = head;
    head = tail = null;
    size = 0;
    return temp.element; //return data in the node
  }
  else
  {
    Node<E> current = head; //current points to head
    for (int i = 0; i < size - 2; i++)
      current = current.next; //move current to second last node
    Node<E> temp = tail; //temp point to tail
    tail = current; //tail point to current node
    tail.next = null; //set tail.next to null
    size--; //decrease list size by one element
    return temp.element; //return data element
  }
}
```

Implementing remove(int index)



(a) Before the node is deleted.



(b) After the node is deleted.

Implementing remove(int index)

```
public E remove(int index)
{
  if (index < 0 || index >= size) return null; //invalid index
  else if (index == 0) return removeFirst(); //first node
  else if (index == size - 1) return removeLast(); //last node
  else
  {
    Node<E> previous = head;  //create and set pointer previous
    for (int i = 1; i < index; i++) {
      previous = previous.next; //move previous to node index-1
    }
    Node<E> current = previous.next;//create and set pointer
                                    //current
    previous.next = current.next; //update the links
    size--; //decrease list size by one element
    return current.element; //return data element
  }
}
```

Recursive Linked List Operations

Think about writing pseudo code for recursive linked list operations.

Example: Print_List (List ListName). List may be empty.

```
//Pseudo code - Recursive method Print_List
//Print list content one value per line
Method Print_List (List ListName)
{
  if (ListName != null)
  {   print(ListName.data);
      Print NewLine;
      Print_List(ListName.next); //recursive call
  }
}
```

Try to modify to print the list in reverse order!

Recursive Linked List Operations

Example: Add node to end of list. List may be empty.

```
//Pseudo code - Recursive method Add_End_Node
//Add a new node to the end of the list
Method Add_End_Node(List ListName; DataType item)
{
  if (ListName == null)
  {  // 1. create and initialize a new bode
     Node temp = new Node(); //create temp node
     temp.data = item; // add data to temp node
     temp.next = null; // set next to null
     ListName = temp;  // link as first node
  }
  else Add_End_Node(ListName.next, item);
}
```

Recursive Linked List Operations

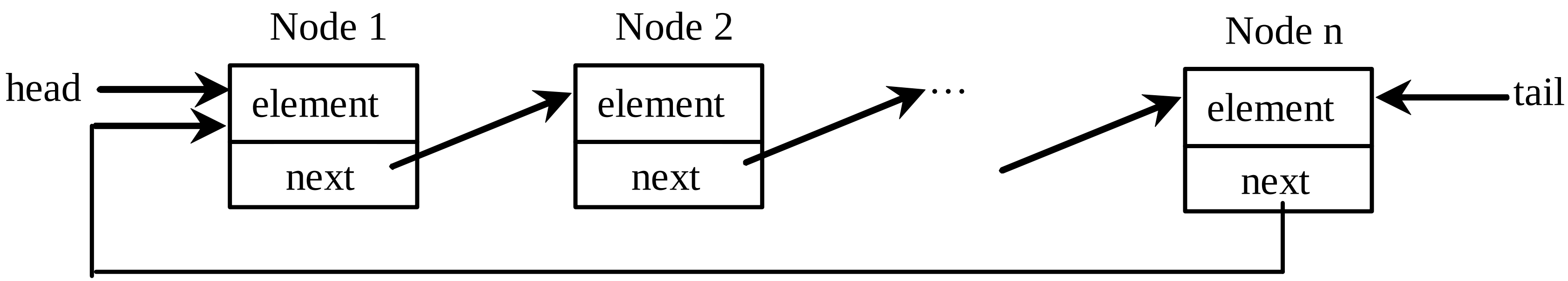
Other possible recursive linked list operations include

```
Delete_List (Node L);       // delete list one node a time.
Copy_List (Node L);         // make a copy of the list.
Count_Nodes (Node L);       // return number of nodes (list length)
Sum_Nodes (Node L);         // Sum all nodes value (assuming a list of numbers)
ReversPrint_List (Node L);  // make a copy of the list.
```

Try to write pseudocode or Java code for these operations as practice for writing recursive methods.

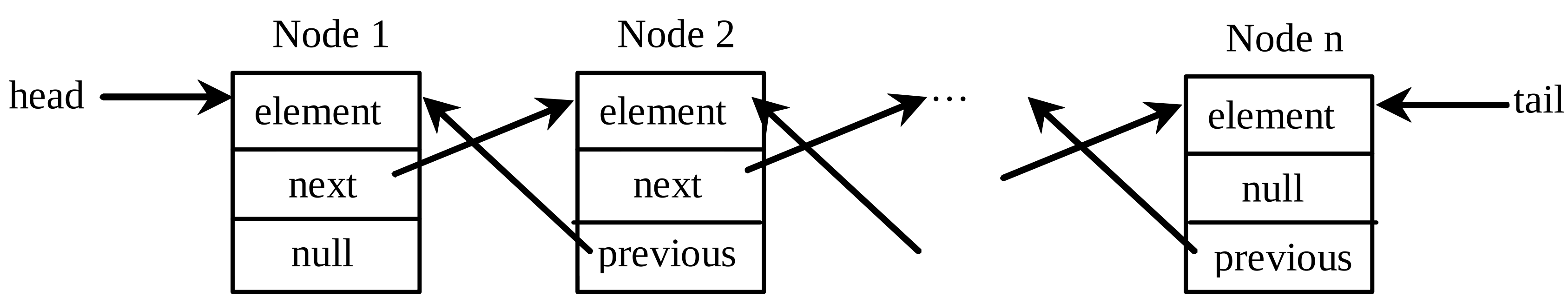
Circular Linked Lists

A *circular, singly linked list* is like a singly linked list, except that the pointer of the last node points back to the first node.



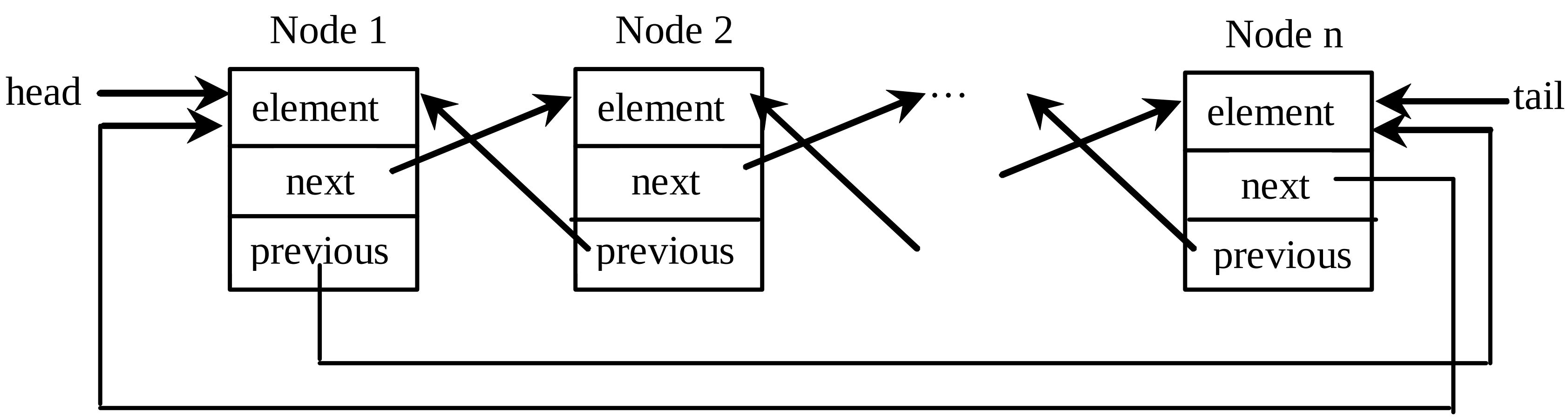
Doubly Linked Lists

A *doubly linked list* contains the nodes with two pointers. One points to the next node and the other points to the previous node. These two pointers are conveniently called *a forward pointer* and *a backward pointer*. So, a doubly linked list can be traversed forward and backward.



Circular Doubly Linked Lists

A *circular, doubly linked list* is doubly linked list, except that the forward pointer of the last node points to the first node and the backward pointer of the first pointer points to the last node.



End of Slides