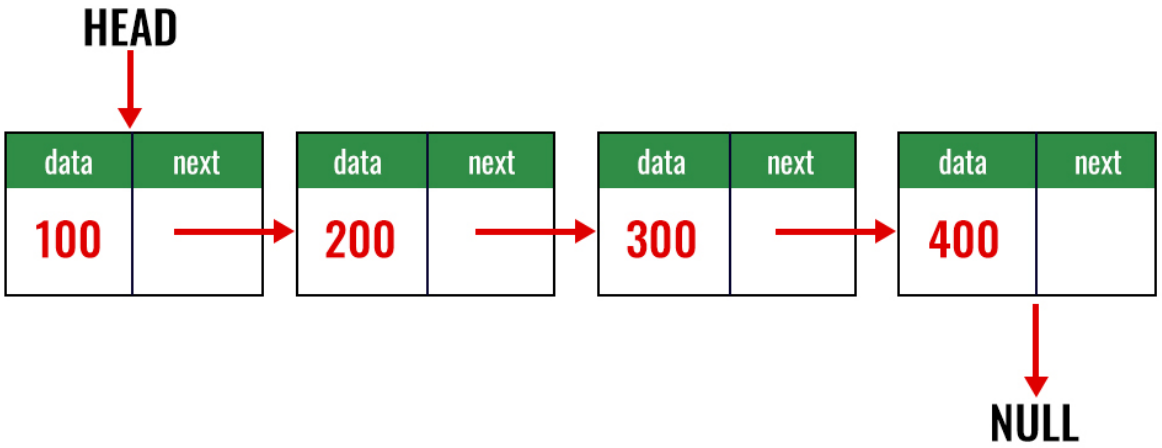
IT 179
9

Single-Linked Lists

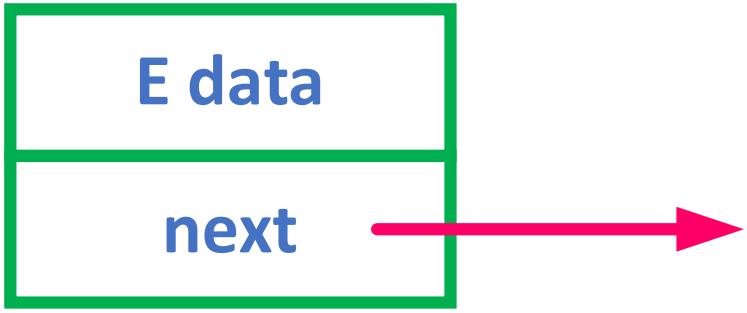
Single-Linked Lists

- A linked list is useful for inserting and removing at arbitrary locations
- The `ArrayList` is limited because its `add` and `remove` methods operate in linear (O(*n*)) time—requiring a loop to shift elements
- A linked list can add and remove elements at a known location in O(1) time
- In a linked list, instead of an index, each element is **linked to the following element**

Example of Single Linked Lists



A List Node



- A node can contain:
  - a data item
  - one or more links
- A link is a **reference** to a list node
- In our structure, the node contains:
  - a data field named `data` of type `E`
  - a reference to the next node, named `next`

Note

To simplify the explanation:

- I will **not** use nested classes.
- Also, the fields `data` and `next` will not be declared as private.

A List Node


```
public class Node<E>
{
    public E data;
    public Node<E> next;

    /**
     * Creates a new node with a null next field
     *
     * @param dataItem The data stored
     */
    Node(E dataItem)
    {
        this.data = dataItem;
        next = null;
    }

    /**
     * Creates a new node that references another node
     *
     * @param dataItem The data stored
     * @param nodeRef  The node referenced by new node
     */
    Node(E dataItem, Node<E> nodeRef)
    {
        data = dataItem;
        next = nodeRef;
    }
```

A Single-Linked List Class

- Generally, we do not have individual references to each node.
- A **`ISUSingleLinkedList`** object has a data field **`head`**, the *list head*, which references the first list node

```
public class ISUSingleLinkedList<E> {
  private Node<E> head = null;
  private int size = 0;
  ...
}
```

The ISUSingleLinkedList Class

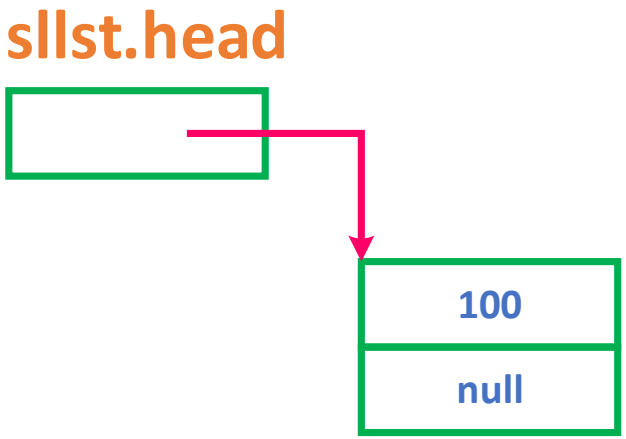

```
public class ISUSingleLinkedList<E>
{

    private Node<E> head = null;
    private int size = 0;


    public void addFirst(E item)
    {
        Node<E> temp = new Node<E>(item, head);
        head = temp;
        size++;
    }
```

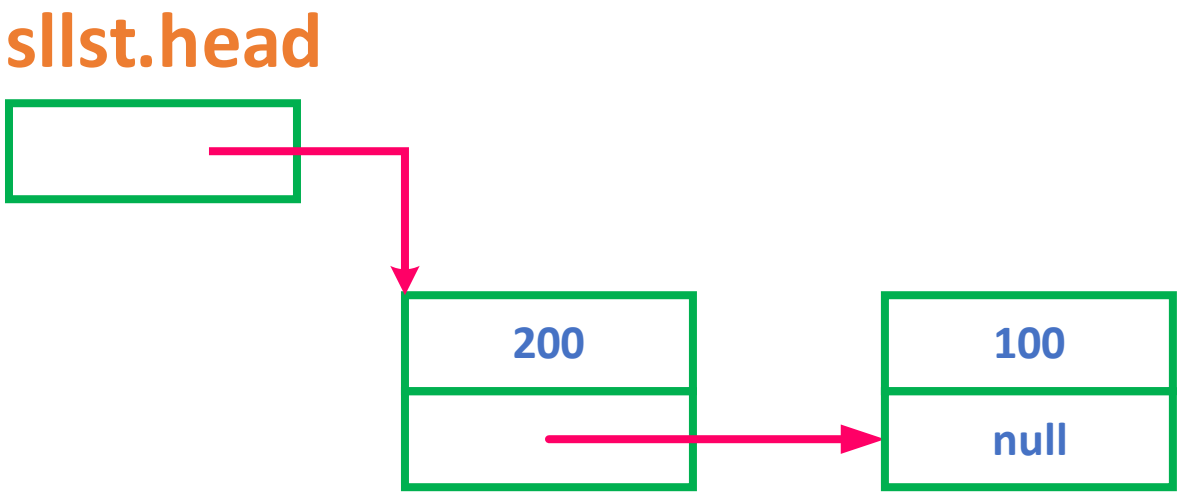
Adding one element


```
public static void main(String[] args)
    {
        ISUSingleLinkedList<Integer> sllst = new ISUSingleLinkedList<>();
        sllst.addFirst(100);
    }
```



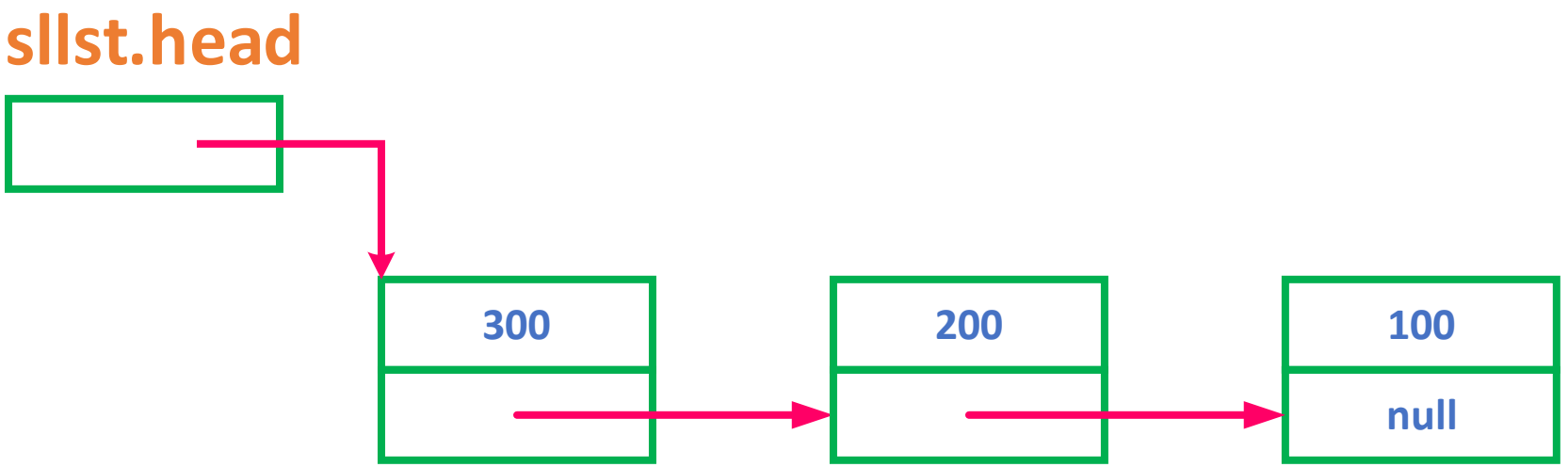
Adding a second element


```
public static void main(String[] args){
        ISUSingleLinkedList<Integer> sllst = new ISUSingleLinkedList<>();
        sllst.addFirst(100);
        sllst.addFirst(200);
    }
```



Adding a third element


```
public static void main(String[] args)    {
        ISUSingleLinkedList<Integer> sllst = new ISUSingleLinkedList<>();
        sllst.addFirst(100);
        sllst.addFirst(200);
        sllst.addFirst(300);
    }
```



Traversing a Single-Linked List

```
public class ISUSingleLinkedList<E>
{

    private Node<E> head = null;
    private int size = 0;

    public void displayList()
    {

        Node<E> currNode = head;
        while (currNode != null)
        {
            System.out.print(currNode.data + " ");
            currNode = currNode.next;
        }

    }

    public void addFirst(E item)
    {
        Node<E> temp = new Node<E>(item, head);
        head = temp;
        size++;
    }
```

Implementing `removeFirst()`

```
public E removeFirst () {
  Node<E> temp = head;
  if (head != null) {
    head = head.next;
  }
  if (temp != null) {
    size--;
    return temp.data
  } else {
    return temp;
  }
}
```

Implementing `removeLast()`

```
public E removeLast()
   {
       Node<E> temp;
       Node<E> prev = head;
       System.out.println("\nRemoving LAST element ...");

       if (prev == null)
           return null;
       if (prev.next == null)
       {
           head = null;
           return prev.data;
       }
       while (prev.next.next != null)
           prev = prev.next;

       temp = prev.next;
       prev.next = null;
       return temp.data;
   }
```

More Methods of `List<E>` Interface in `ISUSingleLinkedList<E>`

Method	Behavior
`public E get(int index)`	Returns the data in the element at position `index`
`public E set(int index, E anEntry)`	Stores a reference to `anEntry` in the element at position `index`. Returns the data formerly at position `index`
`public int size()`	Gets the current size of the `List`
`public boolean add(E anEntry)`	Adds a reference to `anEntry` at the end of the `List`. Always returns `true`
`public void add(int index, E anEntry)`	Adds a reference to `anEntry`, inserting it before the item at position `index`
`int indexOf(E target)`	Searches for `target` and returns the position of the first occurrence, or –1 if it is not in the `List`
`E remove(int index)`	Removes the entry formerly at position `index` and returns it

public E get(int index)

```
public E get (int index) {
 if (index < 0 || index >= size) {
    throw new

IndexOutOfBoundsException(Integer.toString(index));
  }
  Node<E> node = getNode(index);
  return node.data;
}
```

SLList.getNode(int)

- In order to implement methods required by the List interface, we need an additional helper method:

```
private Node<E> getNode(int index) {
  Node<E> node = head;
  for (int i=0; i<index && node != null; i++) {
    node = node.next;
  }
  return node;
}
```

public E set(int index, E newValue)

```
public E set (int index, E newValue) {
  if (index < 0 || index >= size) {
    throw new  I
       IndexOutOfBoundsException(Integer.toString(index));
  }
  Node<E> node = getNode(index);
  E result = node.data;
  node.data = newValue;
  return result;
}
```

public void add(int index, E item)

```
public void add (int index, E item) {
 if (index < 0 || index > size) {
   throw new
     IndexOutOfBoundsException(Integer.toString(index));
 }
 if (index == 0) {
   addFirst(item);
 } else {
   Node<E> node = getNode(index-1);
   addAfter(node, item);
 }
}
```

Implementing `addAfter(Node<E>, E)` (cont.)

```
private void addAfter (Node<E> node, E item) {
  Node<E> temp = new Node<E>(item, node.next);
  node.next = temp;
  size++;
}
```

or, more simply …

```
private void addAfter (Node<E> node, E item) {
  node.next = new Node<E>(item, node.next);
  size++;
}
```

We declare this method `private` since it should not be called from outside the class. Later we will see how this method is used to implement the public add methods

public boolean add(E item)

- To add an item to the end of the list

```
public boolean add(E item) {
   add(size, item);
   return true;
  }
```