Introduction to Java for C++ Programmers

List

Mahboob Ali

The List

- A List stores duplicate elements.
- A list can not only store duplicate elements, but can also allow the user to specify where the element is stored.
- The user can access the element by index.

Syntax

```
public interface List<E> extends Collection<E>{
     }


List<Integer> arrayList = new ArrayList<>();
List<Integer> linkedList = new LinkedList<>();
```

Positional operations

```
public interface List<E> extends Collection<E>{

 E get(int index);

 E set(int index, E element); //optional

 void add(int index, E element); //optional

 boolean add(E element); //optional

 E remove(int index); // optional

 boolean addAll(int index, Collection<? extends E> c; // optional

}
```

Using positional operations

…

```
List <Integer> arrayList = new ArrayList<>();
arrayList.add(0, 1) // adding at 0 index
arrayList.add(1, 5) // adding at 1 index
List <Integer> arrayList1 = new ArrayList<>();
arrayList1.add(1) // adding at 0 index
arrayList1.add(2) // adding at 1 index
arrayList1.add(3) // adding at 2 index
arrayList.addll(0, arrayList1); //adding arrayList1 at 0 index
arrayList.remove(1);
```

Search operations

```
public interface List<E> extends Collection<E>{
        int indexOf(Object o);
        int lastIndexOf(Object o);
}
…
        List<String> arrayList = new ArrayList<>(5); //Size of 5
        arrayList.add(“Hello”);
        arrayList.add(“World”);
        System.out.println(“Hello is at index: ” + arrayList.indexOf(“Hello”));
        System.out.println(“World is at index: ” +
                                                         arrayList.lastindexOf(“World”));
```

Range-view operations

```
public interface List<E> extends Collection<E>{

 List<E> subList(int fromIndex, int toIndex);

}
```

Operations on the List

- Access: manipulates the elements based on the provided index (numerical) position in the list.
- Search: for specified object in the list and returns its index (position).
- Iteration: extends from the Iterator Interface, provide advantage of the List logical order.
- Range: the subList method performs arbitrary range operations on the list.

Two Ways to Implement Lists

There are two ways to implement a list.

Using arrays:

- One is to use an array to store the elements.
- 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.
- Default size is 10, uses ***System.arraycopy*** when increasing size.

Using linked list:

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

List Implementation

- **ArrayList:**
  - A resizable array implementation of a List interface.
  - Default capacity = 10, increased by 50%.
  - *ArrayList*(int initialCapacity) OR

    ensureCapacity(int)
  - Allow duplicates and nulls.

Typical uses

- Simple iteration of elements.
- Fast random access ~ O(1) ~ constant time
  - So size doesn’t matter here.
- Appending elements or deleting elements ~ O(1) ~ constant time

add & remove Methods

- *add*(index, element)
  - Following elements ***shifted right*** by one position.
  - O(n) ~ Linear time

- *remove*(index)
  - Following elements ***shifted left*** by one position.
  - O(n) ~ Linear time

Search methods

- *contains*()

- *indexOf*()
  - O(n) ~ Linear time
  - Uses *equals*()
  - Frequent search operations then consider using **Set** implementation, e.g., HashSet ~ O(1) ~ constant time

```
import java.util.ArrayList;
import java.util.List;

public class CreateArrayListExample {
      public static void main(String[] args) {
      // Creating an ArrayList of String
      List<String> animals = new ArrayList<>();
      // Adding new elements to the
      ArrayList animals.add("Lion");
      animals.add("Tiger");
      animals.add("Cat");
      animals.add("Dog");

      System.out.println(animals);

      //Adding an element at a particular index in an ArrayList
      animals.add(2, "Elephant");
      System.out.println(animals); } }
```

```
[Lion, Tiger, Cat, Dog]
```

```
[Lion, Tiger, Elephant, Cat, Dog]
```

Linked Lists

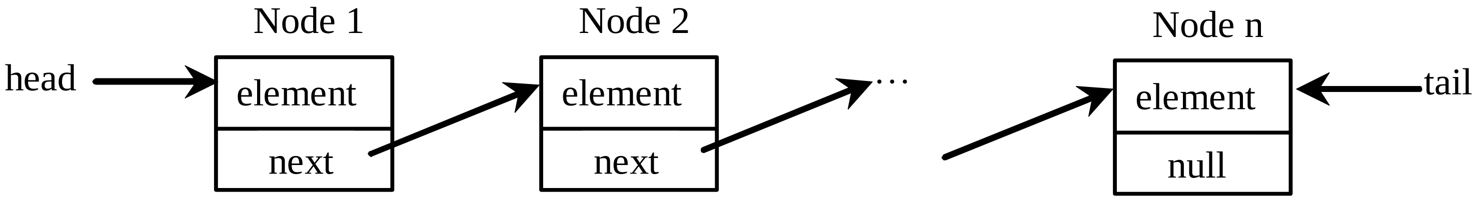
- In ArrayList the methods
  - get(int index)
  - set(int index, Object o)
- for accessing and modifying an element through an index and the add(Object o) for adding an element at the end of the list are efficient.
- However, the methods
  - add(int index, Object o)
  - remove(int index)
- are inefficient because it requires 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.

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:

```
class Node<E> {
  E element;
  Node<E> next;

  public Node(E o) {
    element = o;
  }
}
```



Adding Three Nodes

- The variable
  - head refers to the first node in the list
  - 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:

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

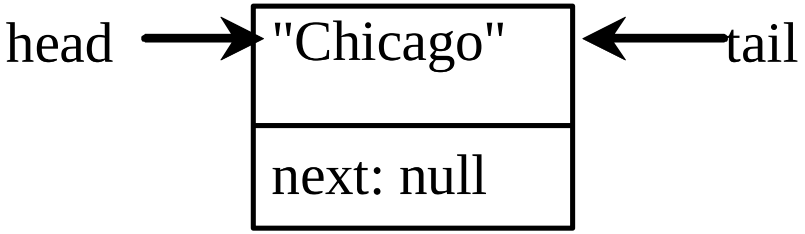
The list is empty now

Adding Three Nodes, cont.

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

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

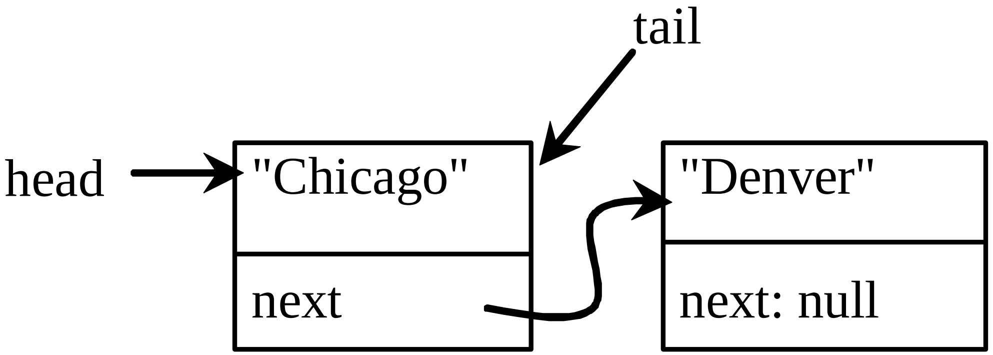
After the first node is inserted



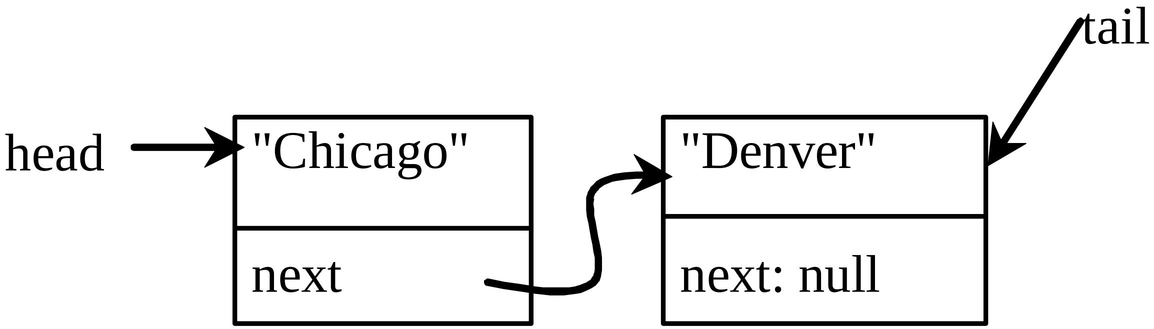
Adding Three Nodes, cont.

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

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



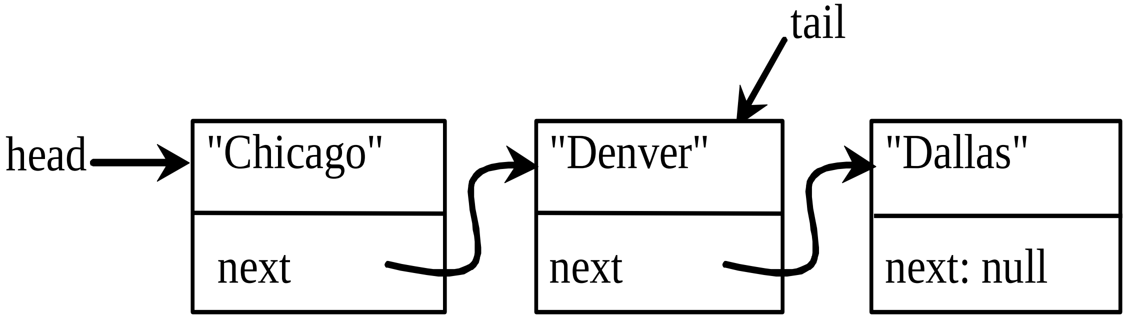
```
tail  = tail.next;
```



Adding Three Nodes, cont.

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

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



tail

head

"Chicago"

next

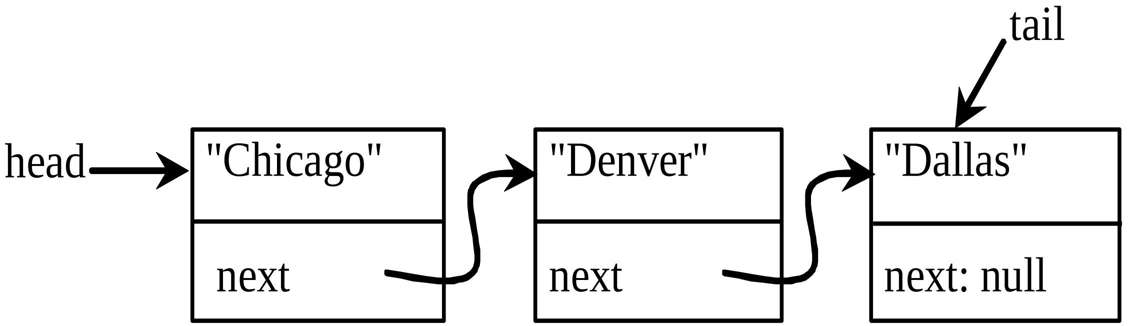
"Denver"

next

"Dallas"

next: null

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



tail

head

"Chicago"

next

"Denver"

next

"Dallas"

next: null