Lecture 12

List Iterators

Recap: Abstract Data Type *List*

- *List* is an abstraction for a sequence of values where order matters
- Nothing is specified about how the underlying data should be stored:
  - The data could be stored in an array
  - The data could be stored in a linked list
  - The data could be stored in some other way
- For some problems the users of a List need to iterate over all the data in a sequential way

List: supported operations

ADT *List* supports the following main operations

- Get element by position: `get(int index)`
- Search element: `indexOf(E element)`
- Add new element: `add(int index, E element)`
- Remove element by position: `remove(i)`

All these operations might be insufficient and we might **need access to the underlying implementation**

Example: count occurrences

- Write a method *count* that counts the number of times a particular element *o* appears in a List:

```
public static int count (List list, E o) {
    int counter = 0;
    for (int i=0; i<data.size(); i++) {
        E obj = data.get(i);
        if (obj.equals(o)) counter++;
    }
    return counter;
}
```

get(i) is provided by any implementation of the List ADT

- **Question**: would this work well no matter if the List is an *Array List* or a *Linked List*?

Example: count occurrences

- Write a method *count* that counts the number of times a particular element *o* appears in a List:

```
public static int count(List list, E o) {
    int counter = 0;
    for (int i=0; i<data.size(); i++) {
        E obj = data.get(i);
        if (obj.equals(o)) counter++;
    }
    return counter;
}
```

1+2+3+…+n

- **Answer:** No, this method is very inefficient for Linked Lists: *get(i)* always starts from the *head* and the loop above is O(n^2)

Efficient solutions are fundamentally different for:

- **Array List**

```
int count (E element){
  int counter = 0;
  for(int i=0; i<size; i++){
      if(data[i].equals(element)
          counter ++;
  }
  return counter;
}
```

Using *for* loop and indexes

- **Linked List**

```
int count (E element){
  int counter = 0;
  Node finger = head;
  while(finger != null){
   if(finger.data.equals(element)
      counter ++;
   finger = finger.next;
  }
  return counter;
}
```

Using *while* loop and *next*

- But the principle of ADT **forbids the use of underlying data structures directly**!
- We need a uniform interface to iterate over List elements efficiently

Efficient uniform iteration over List

- Problem: Efficient and uniform dispensing of values from the underlying data structures
- Solution: We create and use the common **interface for iteration**

Extending operations for List ADT

- get()
- indexOf()
- add()
- remove()
- size()
- isEmpty()
- clear()

But also a method for efficient looping through the data

- iterator()

Iterator interface

- Iterators provide support for efficiently visiting all elements of an underlying data structure
- We customize the implementation of the iterator depending on the underlying data structure
- We abstract away the details of how to access elements

`public interface Iterator<E> :`

`boolean` ***`hasNext()`*** – are there more elements for iteration?

`E` ***`next()`*** – return next element

What do we need for efficient iteration

- We maintain the state of the iteration in a variable and progress one location at a time
- We don't start at the beginning each time – rather we resume where we left off

Example: Iterator for *Array List*

This is a part of the ArrayList class: it has access to list.data array

```
private class ArrayListIterator implements Iterator{
     ArrayList list;
     int currentIndex;
     public ArrayListIterator (ArrayList list){
          this.list = list;
          this.currentIndex = 0;
     }

     public boolean hasNext (){
          return (this.currentIndex < list.size());
     }

     public Object next(){
          Object result = list.data[currentIndex];
          currentIndex++;
          return result;
     }
}
```

Iterator for *Array List*

```
private class ArrayListIterator implements Iterator{
      ArrayList list;
      int currentIndex;
      public ArrayListIterator (ArrayList list){
            this.list = list;
            this.currentIndex = 0;
      }

      public boolean hasNext (){
            return (this.currentIndex < list.size());
      }

      public Object next(){
            Object result = list.data[currentIndex];
            currentIndex++;
            return result;
      }
}
```

Reference to the actual Array List

We set it in the constructor

Iterator for *Array List*

```
private class ArrayListIterator implements Iterator{
      ArrayList list;
      int currentIndex;
      public ArrayListIterator (ArrayList list){
            this.list = list;
            this.currentIndex = 0;
      }

      public boolean hasNext (){
            return (this.currentIndex < list.size());
      }

      public Object next(){
            Object result = list.data[currentIndex];
            currentIndex++;
            return result;
      }
}
```

Stores the current state of the iteration: the position in the array to be returned *next()*

Iterator for *Array List*

```
private class ArrayListIterator implements Iterator{
     ArrayList list;
     int currentIndex;
     public ArrayListIterator (ArrayList list){
          this.list = list;
          this.currentIndex = 0;
     }

     public boolean hasNext (){
          return (this.currentIndex < list.size());
     }

     public Object next(){
          Object result = list.data[currentIndex];
          currentIndex++;
          return result;
     }
}
```

As long as *currentIndex* is within valid bounds

Iterator for *Array List*

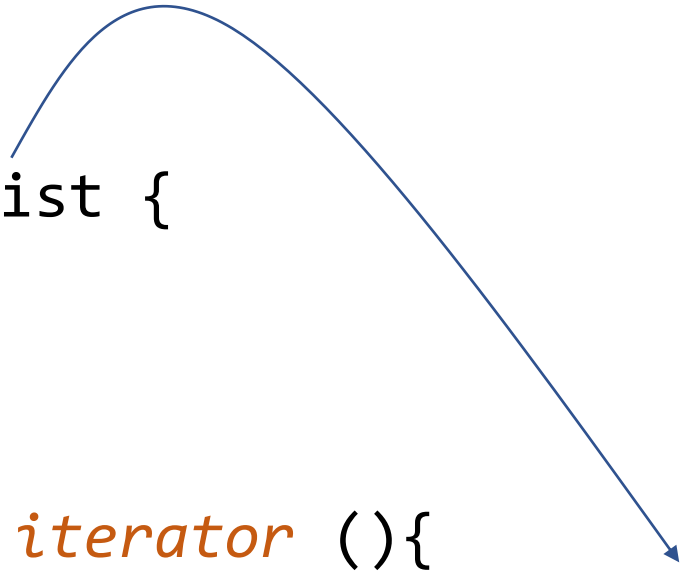
```
private class ArrayListIterator implements Iterator{
     ArrayList list;
     int currentIndex;
     public ArrayListIterator (ArrayList list){
          this.list = list;
          this.currentIndex = 0;
     }

     public boolean hasNext (){
          return (this.currentIndex < list.size());
     }

     public Object next(){
          Object result = list.data[currentIndex];
          currentIndex++;
          return result;
     }
}
```

Return the element at position *currentIndex*, and advance *currentIndex* to the next position

ArrayList *iterator*() returns an array-specific Iterator:



```
public class ArrayList {
    Object[] data;
    int size;

    public Iterator iterator (){
        return new ArrayListIterator(this);
    }

}
```

Iterator for *Linked List*

```
private class LinkedListIterator implements Iterator{
      LinkedList list;
      Node currentNode;

      public LinkedListIterator (LinkedList list){
            this.list = list;
            this.currentNode = list.head;
      }

      public boolean hasNext (){
      }

      public Object next(){
      }
}
```

Same as before: reference to the actual Linked List

Iterator for *Linked List*

```
private class LinkedListIterator implements Iterator{
      LinkedList list;
      Node currentNode;

      public LinkedListIterator (LinkedList list){
            this.list = list;
            this.currentNode = list.head;
      }

      public boolean hasNext (){
      }

      public Object next(){
      }
}
```

Stores the current state of the iteration: node to be read next

Initializes current node to the head node

Linked List Iterator: *hasNext*()

Which of the following is the correct implementation of *hasNext*()?

A.
```
boolean hasNext(){
    return (this.list.size()>0)
}
```

B.
```
boolean hasNext(){
    return (currentNode.next != null)
}
```

C.
```
boolean hasNext(){
    return (currentNode!= null)
}
```

D. None of the above

```
public class LinkedListIterator
                implements Iterator{
        LinkedList list;
        Node currentNode;
        public boolean hasNext (){
        }

        public Object next(){
        }
}
```

Linked List Iterator: *next*()

Which of the following is the correct implementation of *next*()?

A.
```
Object next(){
    return this.list.get(currentNode)
}
```

B.
```
Object next(){
    currentNode = currentNode.next;
    return currentNode.data;
}
```

C.
```
Object next(){
    Object result = currentNode.data;
    currentNode = currentNode.next;
    return result;
}
```

D. None of the above

```
public class LinkedListIterator
                implements Iterator{
    LinkedList list;
    Node currentNode;
    public boolean hasNext (){
    }

    public Object next(){
    }
}
```

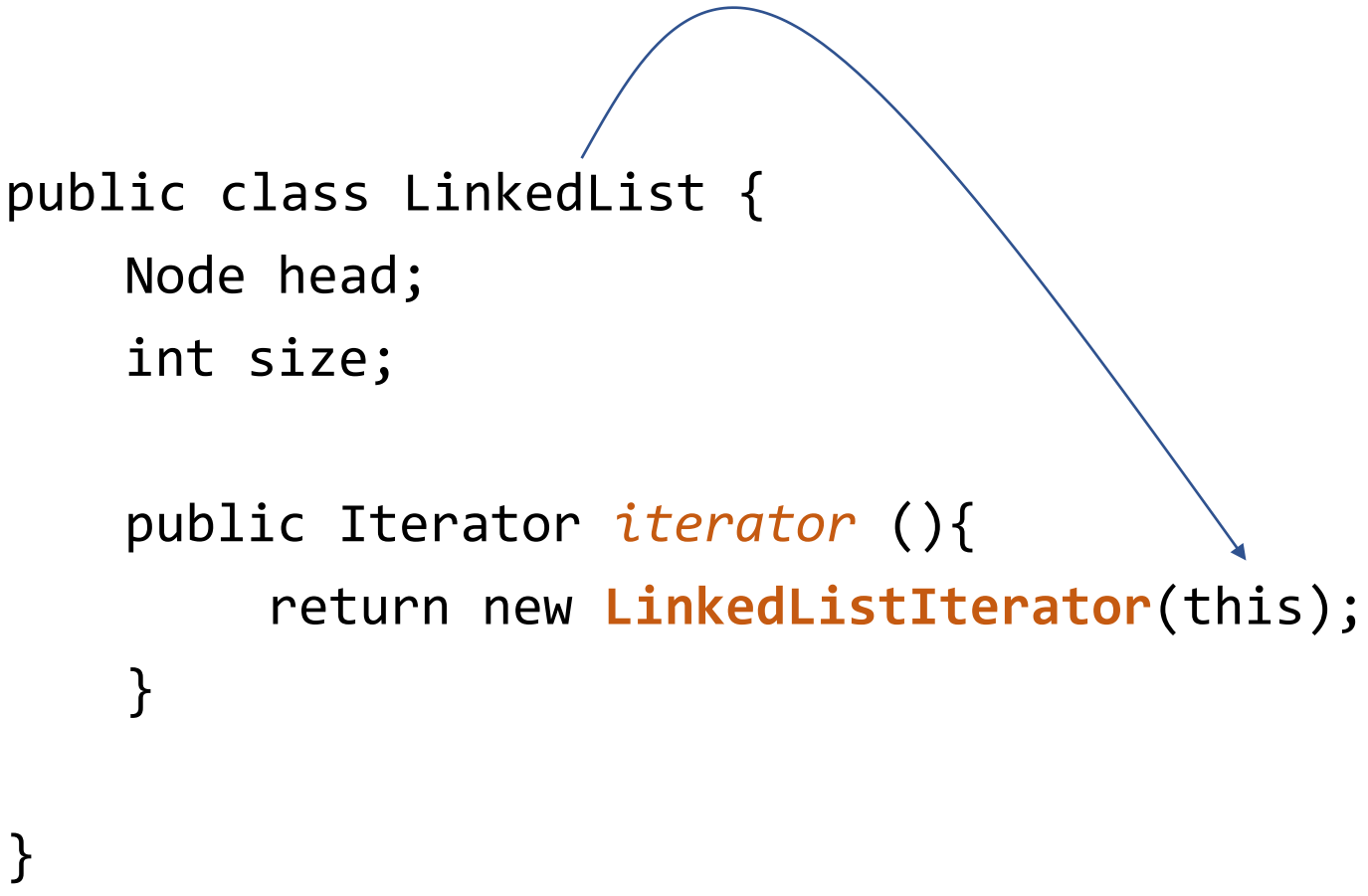
Iterator for *Linked List*

```
private class LinkedListIterator implements Iterator{
      LinkedList list;
      Node currentNode;
      …
      public boolean hasNext (){
            return (currentNode != null);
      }

      public Object next(){
            Object result = currentNode.data;
            currentNode = currentNode.next;
            return result;
      }
}
```

hasNext() basically answers: can we call next()?

Linked List with its own iterator



```
public class LinkedList {
    Node head;
    int size;

    public Iterator iterator (){
        return new LinkedListIterator(this);
    }

}
```

Uniform Counting with iterator()
Works for both Array List and Linked List

```
public int count (List list, Object o) {
    int counter = 0;
    Iterator iter = list.iterator();
    while (iter.hasNext())
        if(o.equals(iter.next())) counter++;
    return counter;
}
```

Data-structure specific operations inside

Iterators: notes

- Iterator objects provide a unified interface for traversing List ADT
- They have access to internal data representations
- They also store the state of traversal
- To implement an efficient iterator you need to **understand the mechanics of the underlying data structure and the complexity of its operations**

DEMO

- External definition of a Node:
  Node.java
- Full example of implementing Iterator for a LinkedList
  LinkedListWithIterator.java

- Why iterators are useful:
  ModeIterators.java
  Computes the mode [most common value] in a list of Integers using a nested loop. Shows why the iterators are useful: using get(i) for LinkedList degrades the performance.