NC STATE UNIVERSITY

Iterators

Readings:
Zybooks 11.8 & 11.9

Slides are modified from those provided by Marty Stepp
www.buildingjavaprograms.com
Additional content from Dr. Jessica Schmidt

NC STATE UNIVERSITY

Why Use ADTs?

- Why would we want more than one kind of list, queue, etc.?
- Answer: Each implementation is more efficient at certain tasks.
  - `ArrayList` is faster for adding/removing at the end;
    `LinkedList` is faster for adding/removing at the front/middle.
  - You choose the optimal implementation for your task, and if the rest of your code is written to use the ADT interfaces, it will work.
  - And you can change out the underlying type later if you find something more efficient!

NC STATE UNIVERSITY

Complexity Classes

- **Complexity Class**: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

Class	Big-Oh	If you double N, …	Example
constant	O(1)	unchanged	10ms
logarithmic	O($\log_2$ N)	increases slightly	175ms
linear	O(N)	doubles	3.2 sec
log-linear	O(N $\log_2$ N)	slightly more than doubles	6 sec
quadratic	O(N^2)	quadruples	1 min 42 sec
cubic	O(N^3)	multiplies by 8	55 min
…	…	…	…
exponential	O(2^N)	multiplies drastically	5 * 10^{61} years

NC STATE UNIVERSITY

Linked list iterator

- The following *client* code is particularly slow on linked lists:

```
List<Integer> list = new LinkedList<Integer>();
...
for (int i = 0; i < list.size(); i++) {
    int value = list.get(i);
    if (value % 2 == 1) {
        list.remove(i);
    }
}
```

- Why is this client code slow?
- What is the runtime efficiency?
- What can we do to improve the runtime?

for-each loop and `Iterable`

- Java's collections can be iterated using a "for-each" loop:

```
List<String> list = new LinkedList<String>();
...
for (String s : list) {
    System.out.println(s);
}
```

  – Our collections do not work in this way

- To fix this, your list must implement the `Iterable` interface.

```
public interface Iterable<E> {
    public Iterator<E> iterator();
}
```


Interface Iterable<T>

Type Parameters:

T - the type of elements returned by the iterator

All Known Subinterfaces:

BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Collection<E>, Deque<E>, DirectoryStream<T>, List<E>, NavigableSet<E>, Path, Queue<E>, SecureDirectoryStream<T>, Set<E>, SortedSet<E>, TransferQueue<E>

All Known Implementing Classes:

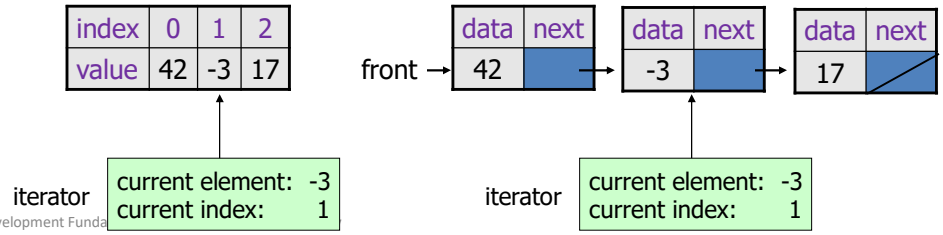
AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BatchUpdateException, BeanContextServicesSupport, BeanContextSupport, ConcurrentHashMap.KeySetView, ConcurrentLinkedDeque, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DataTruncation, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, RowSetWarning, SerialException, ServiceLoader, SQLClientInfoException, SQLDataException, SQLException, SQLFeatureNotSupportedException, SQLIntegrityConstraintViolationException, SQLInvalidAuthorizationSpecException, SQLNonTransientConnectionException, SQLNonTransientException, SQLRecoverableException, SQLSyntaxErrorException, SQLTimeoutException, SQLTransactionRollbackException, SQLTransientConnectionException, SQLTransientException, SQLWarning, Stack, SyncFactoryException, SynchronousQueue, SyncProviderException, TreeSet, Vector

`List` Interface

```
// Represents a list of values.
public interface List<E> extends Iterable<E> {
    public void add(E value);
    public void add(int index, E value);
    public E get(int index);
    public int indexOf(E value);
    public boolean isEmpty();
    public Iterator<E> iterator();
    public void remove(int index);
    public void set(int index, E value);
    public int size();
}
```


Iterators

- **Iterator**: An *object* that allows a client to traverse the elements of a collection, regardless of its implementation.
  - Remembers a position within a collection, and allows you to:
    - get the element at that position
    - advance to the next position
    - (possibly) remove or change the element at that position
  - Benefit: A common way to examine *any* collection's elements.



NC STATE UNIVERSITY

`Iterator` methods

`hasNext()`	returns `true` if there are more elements to examine
`next()`	returns the next element from the collection (throws a `NoSuchElementException` if there are none left to examine)
`remove()`	removes from the collection the last value returned by `next()` (throws `IllegalStateException` if you have not called `next()` yet)

- Every Java Collection Framework class has an `iterator` method

```
LinkedList<String> list =
        new LinkedList<String>();
…
Iterator<String> itr = list.iterator();
…
```


NC STATE UNIVERSITY

Iterator Expectations

```
100
101         @Override
102         public E next() {
103             E rtn = current.data;
104             current = current.next;
105             return rtn;
106         }
```

Questions to Ask

- State
  - What will the state (field/s) of the iterator be?
- Constructor
  - What will the state be initialized to?
  - What will be passed to the constructor? (What parameters will the constructor take?)
- hasNext() method
  - How will we know if there is another element in the collection? (What condition can we test?)
- next() method
  - Does another element exist?
  - What is the next element to return?
  - How must we update the state in order to prepare for following call to next?

Array List Iterator

```
public class ArrayList<E> extends AbstractList<E> {

    public Iterator<E> iterator() {
        return new ArrayIterator();
    }

    // not perfect; doesn't forbid multiple removes in a row
    private class ArrayIterator implements Iterator<E> {
        public ArrayIterator() {
        }

        public boolean hasNext() {
        }

        public E next() {
        }

        public void remove() {
        }
    }
}
```

NC STATE UNIVERSITY

Array List Iterator

```
public class ArrayList<E> extends AbstractList<E> {
    // not perfect; doesn't forbid multiple removes in a row
    private class ArrayIterator implements Iterator<E> {

        public ArrayIterator() {

        }
        public boolean hasNext() {

        }
        public E next() {



        }

        public void remove() {
            ArrayList.this.remove(index - 1);
            index--;
        }
    }
}
```

What will the state (field/s) of the iterator be?

NC STATE UNIVERSITY

Array List Iterator

```
public class ArrayList<E> extends AbstractList<E> {
    // not perfect; doesn't forbid multiple removes in a row
    private class ArrayIterator implements Iterator<E> {
        private int index;   // current position in list
        public ArrayIterator() {

        }
        public boolean hasNext() {

        }
        public E next() {



        }

        public void remove() {
            ArrayList.this.remove(index - 1);
            index--;
        }
    }
}
```

What will the state be initialized to?

Array List Iterator

```
public class ArrayList<E> extends AbstractList<E> {
    // not perfect; doesn't forbid multiple removes in a row
    private class ArrayIterator implements Iterator<E> {
        private int index;   // current position in list

        public ArrayIterator() {
            index = 0;
        }

        public boolean hasNext() {

        }

        public E next() {



        }

        public void remove() {
            ArrayList.this.remove(index - 1);
            index--;
        }
    }
}
```

How will we know if there is another element in the collection?

Array List Iterator

```
public class ArrayList<E> extends AbstractList<E> {
    // not perfect; doesn't forbid multiple removes in a row
    private class ArrayIterator implements Iterator<E> {
        private int index;   // current position in list

        public ArrayIterator() {
            index = 0;
        }

        public boolean hasNext() {
            return index < size();
        }

        public E next() {



        }

        public void remove() {
            ArrayList.this.remove(index - 1);
            index--;
        }
    }
}
```

Does another element exist?
What is the next element to return?
How must we update the state in order to prepare for following call to next?

NC STATE UNIVERSITY

Array List Iterator

```
public class ArrayList<E> extends AbstractList<E> {
    // not perfect; doesn't forbid multiple removes in a row
    private class ArrayIterator implements Iterator<E> {
        private int index;   // current position in list

        public ArrayIterator() {
            index = 0;
        }

        public boolean hasNext() {
            return index < size();
        }

        public E next() {
            if(!hasNext()) throw new NoSuchElementException();
            index++;
            return get(index - 1);
        }

        public void remove() {
            ArrayList.this.remove(index - 1);
            index--;
        }
    }
}
```


NC STATE UNIVERSITY

Linked List Iterator

```
public class LinkedList<E> extends AbstractList<E> {
    ...

    public Iterator<E> iterator() {
        return new LinkedIterator();
    }

    // not perfect; doesn't support remove
    private class LinkedIterator implements Iterator<E> {
        public LinkedIterator() {
        }

        public boolean hasNext() {
        }

        public E next() {
        }
        public void remove() {      // not implemented for now
            throw new UnsupportedOperationException();
        }
    }
}
```

NC STATE UNIVERSITY

Linked List Iterator

```
public class LinkedList<E> extends AbstractList<E> {
    ...
    // not perfect; doesn't support remove
    private class LinkedIterator implements Iterator<E> {

        public LinkedIterator() {

        }
        public boolean hasNext() {

        }
        public E next() {



        }

        public void remove() {      // not implemented for now
            throw new UnsupportedOperationException();
        }
    }
}
```

What will the state (field/s) of the iterator be?

NC STATE UNIVERSITY

Linked List Iterator

```
public class LinkedList<E> extends AbstractList<E> {
    ...
    // not perfect; doesn't support remove
    private class LinkedIterator implements Iterator<E> {
        private ListNode current;   // current position in list
        public LinkedIterator() {

        }
        public boolean hasNext() {

        }
        public E next() {



        }

        public void remove() {      // not implemented for now
            throw new UnsupportedOperationException();
        }
    }
}
```

What will the state be initialized to?

NC STATE UNIVERSITY

Linked List Iterator

```
public class LinkedList<E> extends AbstractList<E> {
    ...
    // not perfect; doesn't support remove
    private class LinkedIterator implements Iterator<E> {
        private ListNode current;   // current position in list

        public LinkedIterator() {
            current = front;
        }

        public boolean hasNext() {

        }
        public E next() {




        }

        public void remove() {      // not implemented for now
            throw new UnsupportedOperationException();
        }
    }
}
```

How will we know if there is another element in the collection?

NC STATE UNIVERSITY

Linked List Iterator

```
public class LinkedList<E> extends AbstractList<E> {
    ...
    // not perfect; doesn't support remove
    private class LinkedIterator implements Iterator<E> {
        private ListNode current;   // current position in list

        public LinkedIterator() {
            current = front;
        }

        public boolean hasNext() {
            return current != null;
        }
        public E next() {




        }

        public void remove() {      // not implemented for now
            throw new UnsupportedOperationException();
        }
    }
}
```

Does another element exist?
What is the next element to return?
How must we update the state in order to prepare for following call to next?

NC STATE UNIVERSITY

Linked List Iterator

```
public class LinkedList<E> extends AbstractList<E> {
    ...
    // not perfect; doesn't support remove
    private class LinkedIterator implements Iterator<E> {
        private ListNode current;   // current position in list

        public LinkedIterator() {
            current = front;
        }

        public boolean hasNext() {
            return current != null;
        }

        public E next() {
            if(!hasNext()) throw new NoSuchElementException();
            E result = current.data;
            current = current.next;
            return result;
        }

        public void remove() {      // not implemented for now
            throw new UnsupportedOperationException();
        }
    }
}
```


NC STATE UNIVERSITY

Linked list iterator

```
List<Integer> list = new LinkedList<Integer>();
Iterator<Integer> it = list.iterator();
while(  it.hasNext()  )
    int value =  it.next();
    if (value % 2 == 1) {
        it.remove();
    }
}
```

Note our current LinkedList doesn't implement remove, yet…

`Iterator` Limitations

- **`Iterators` are unidirectional**
  - iterate only in one direction
- **With `Iterators`, iteration can be done only once**
  - If you need to iterate again, you have to use a new Iterator
- **`ConcurrentModificationException`** thrown when list is modified during iteration
  - If you add elements to a list while you are iterating over the list

```
LinkedList<Integer> list = new LinkedList<Integer>();
Iterator<Integer> it = list.iterator();
while(it.hasNext()) {
   int e = it.next();
   if(e % 2 == 0) {
      list.add( -1 * e ); //exception!
   }
}
```


`ListIterator`

- An iterator for lists
- Allows you to:
  - traverse the list in *either* direction (forwards and backwards)
  - modify the list during iteration
  - obtain the iterator's current position in the list.
- A **`ListIterator`** has no "current element"
  - its *position* always lies *between* the element that would be returned by a call to **`previous()`** and the element that would be returned by a call to **`next()`**
  - the **`remove()`** and **`set(Object)`** methods operate on the last element returned by a call to **`next()`** or **`previous()`**.

NC STATE UNIVERSITY

ListIterator

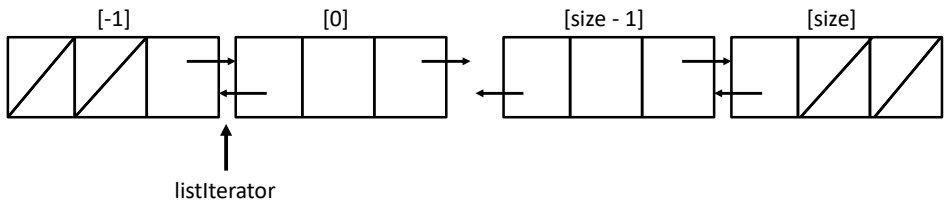
`hasNext()`	returns `true` if this list iterator has more elements when traversing the list in the forward direction
`next()`	returns the next element from the list (or throws a `NoSuchElementException`) and advances the cursor position
`nextIndex()`	returns the index of the element that would be returned by a subsequent call to `next()`
`hasPrevious()`	returns `true` if this list iterator has more elements when traversing the list in the reverse direction
`previous()`	returns the previous element in the list (or throws a `NoSuchElementException`) and moves the cursor position backwards.
`previousIndex()`	returns the index of the element that would be returned by a subsequent call to `previous()`
`add(E e)`	inserts the specified element into the list
`remove()`	removes from the list the last element that was returned by `next()` or `previous()`
`set(E e)`	replaces the last element returned by `next()` or `previous()` with the specified element (optional operation).

28

NC STATE UNIVERSITY

AbstractSequentialList

- Part of Java API
- Used for sequential access – linked list
- All standard **`List`** methods are implemented for you (using a **`ListIterator`**)
  - Except for **`listIterator()`** and **`size()`**
  - All methods use a **`ListIterator`** to iterate to the appropriate location for the operation and then delegate to the **`ListIterator`** to complete the operation.



29

NC STATE UNIVERSITY

LinkedList using ListIterator

- Extend **`AbstractSequentialList`**
- Create the **`LinkedList`** state
  - Reference to the **`front`** of the list
  - Reference to the **`back`** of the list
  - **`Size`**
- Create the **`ListNode`** inner class with references forward and back (doubly linked)
  - **`ListNode`** has a reference to **`data`**, the **`next ListNode`**, and the **`prev(ious) ListNode`**
- Create the default (parameterless) **`LinkedList`** constructor
  - Creates the two nodes with null data at index -1 and size (which initially is 0)
  - Sets **`front`** to point to the node with null data at index -1
  - Sets **`back`** to point to the node with null data at index 0
  - The nodes null data should initially point to each other
- Create the skeleton for the concrete **`ListIterator`** as an inner class of **`LinkedList`**
- Provide an implementation for the **`LinkedList.listIterator()`** method
  - Constructs a new **`ListIterator`** at the provided position
  - ***But you have to create a `ListIterator`!***
- Provide an implementation for the **`LinkedList.size()`** method

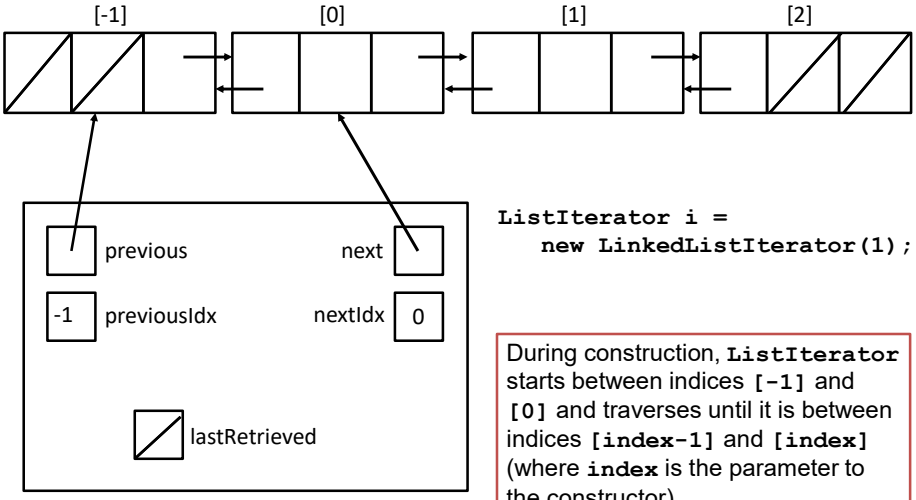
NC STATE UNIVERSITY

ListIterator

- Create an inner class that implements **`ListIterator`**
  - Unmodifiable list: Implement **`hasNext()`**, **`next()`**, **`hasPrevious()`**, **`previous()`**
  - Modifiable list: Implement **`set()`**
  - Variable-size list: **`remove()`** and **`add()`**

NC STATE UNIVERSITY

`ListIterator` Operations – Traversal



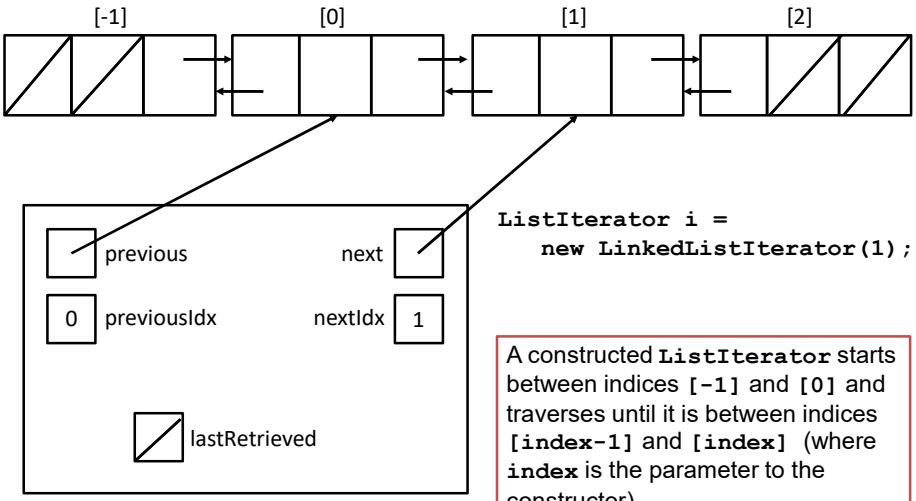

```
ListIterator i =
    new LinkedListIterator(1);
```

During construction, **`ListIterator`** starts between indices **`[-1]`** and **`[0]`** and traverses until it is between indices **`[index-1]`** and **`[index]`** (where **`index`** is the parameter to the constructor).

32

NC STATE UNIVERSITY

`ListIterator` Operations – Traversal



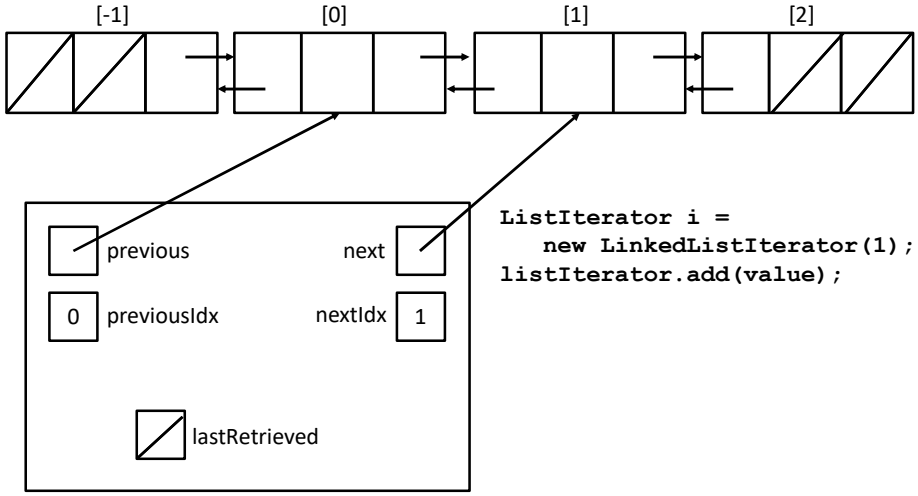

```
ListIterator i =
    new LinkedListIterator(1);
```

A constructed **`ListIterator`** starts between indices **`[-1]`** and **`[0]`** and traverses until it is between indices **`[index-1]`** and **`[index]`** (where **`index`** is the parameter to the constructor).

33

NC STATE UNIVERSITY

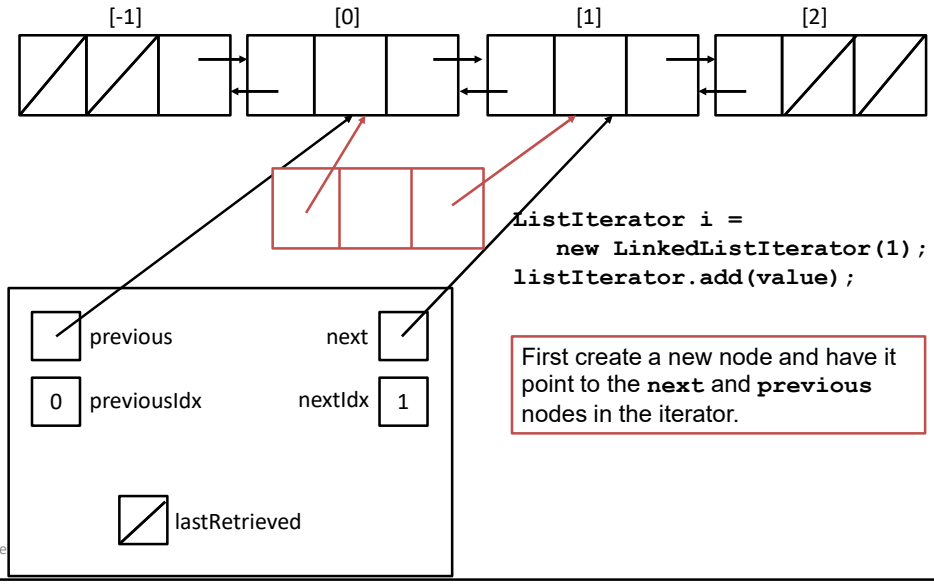
`ListIterator` Operations – add()




```
ListIterator i =
   new LinkedListIterator(1);
listIterator.add(value);
```


NC STATE UNIVERSITY

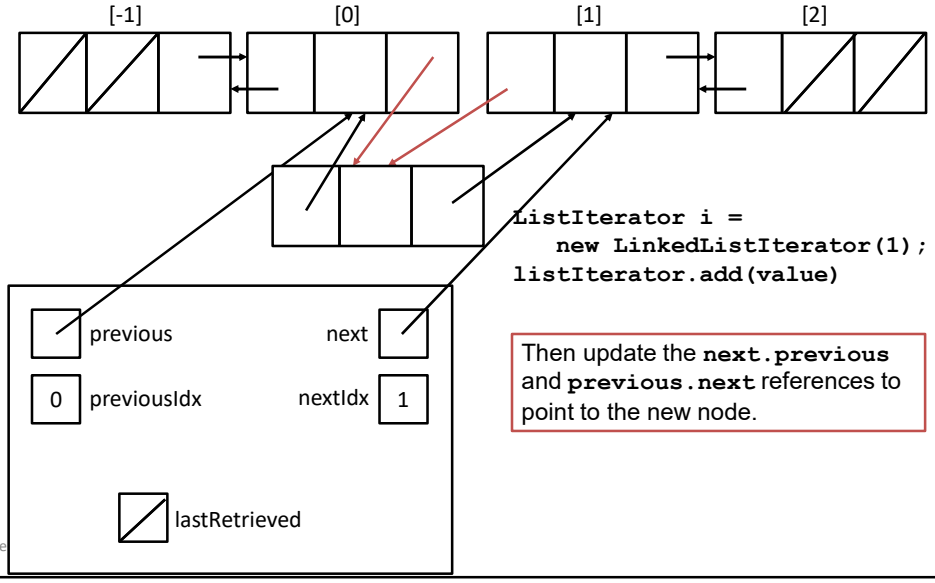
`ListIterator` Operations – add()




```
ListIterator i =
   new LinkedListIterator(1);
listIterator.add(value);
```

First create a new node and have it point to the **`next`** and **`previous`** nodes in the iterator.

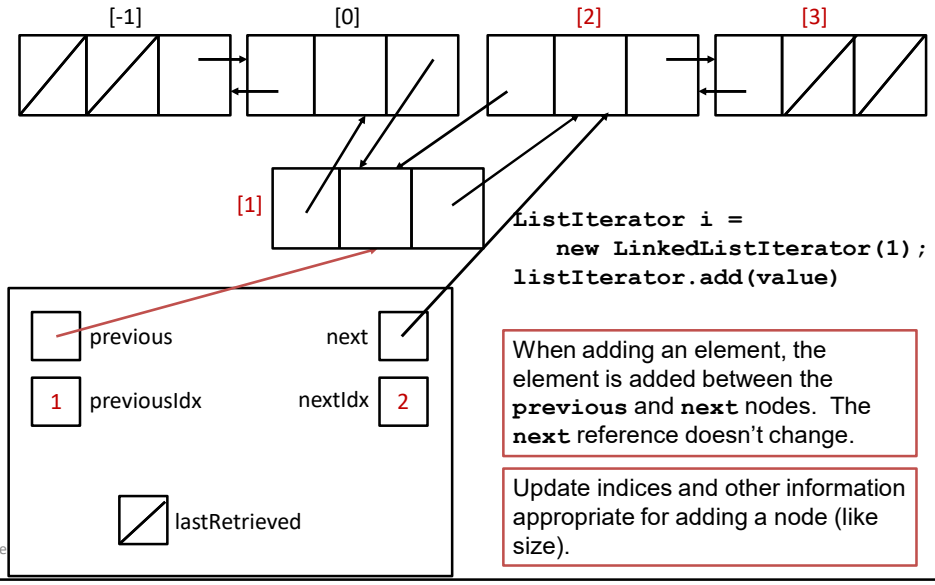
`ListIterator` Operations – add()




```
ListIterator i =
    new LinkedListIterator(1);
listIterator.add(value)
```

Then update the **`next.previous`** and **`previous.next`** references to point to the new node.

`ListIterator` Operations – add()




```
ListIterator i =
    new LinkedListIterator(1);
listIterator.add(value)
```

When adding an element, the element is added between the **`previous`** and **`next`** nodes. The **`next`** reference doesn't change.

Update indices and other information appropriate for adding a node (like size).