School of Electrical Engineering and Computing

SENG2200/6220 Programming Languages & Paradigms

Topic 6

Container and Iterator Implementation with Generic Specifications

Dr Nan Li
Office: ES222
Phone: 4921 6503
Nan.Li@newcastle.edu.au

Topic 6 Overview

What are data structures, really?

What do we normally use data structures for?

What is an Iterator?

How do we implement iterators?

Containers & Collections

Linear
- Arrays
- Stacks
- Queues & Deques
- Lists
- Priority Queue
- Heap

Hierarchical
- Trees
- General Tree
- Binary Tree
- Heap

Graph
- Undirected Graph
- Directed Graph

Unordered
- Set
- Bag
- Map (table)

The term **collection** gives the connotation of a special type of organization within the container.

E.G. Linear Collections

All have an explicit predecessor and successor item.

Arrays
- Capacity and Random Access

Stacks, Queues, & Deques
- Time of entry and exit are the crucial organizing features

Lists
- General insertion and deletion

Iterators and Collections

Basically a way of visiting every item in a collection.

Therefore, it does not need to follow a specific order, but if a specific ordering exists, it may make the iterator's task easier if it follows that ordering.

Needs extra data about the particular collection such as which items have been visited and which have not.

Therefore, we can identify particular behaviours that will be common to all iterators (ie an Interface), that will need to be implemented for each possible collection.

Iterator Behaviour (1)

Provide the ability for clients to traverse collections visiting the items stored in it

The basic operations (in Iterator interface) are:
- Test whether there are more items to be visited
- Visit (look at) the next item
- Some iterators may need to remove items from the container

Iterator Behaviour (2)

7

Create an Iterator

Test if there are more items that have not yet been visited

Obtain access to the next item via the iterator

Remove item from container

- Not an essential part of all iterators

Check consistency

- The container itself is still an object in its own right
- The container can be modified without knowledge of the iterator, and this should lead to the iterator refusing to do more work.

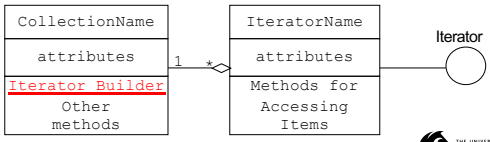
UML Modelling of Iterators

8

The relationship between a Collection and its Iterator is Aggregation, a Collection can HAVE any number of iterators. Note that there is not the same 1 to 1, or lifetime equivalence nature in this relationship, when compared to the composition relationship.

The *iterator* has no meaningful semantics separate from its Collection, so this aggregation is different in its structure than is "normal". A Collection HAS AN iterator (maybe > 1), but an Iterator must also have a particular single collection in order to be meaningful.

So there may be a number of ways to draw these relationships in UML, the following is (probably) the best way to do so:



How to construct an iterator

9

An iterator IS an object – but a container object HAS AN iterator (or maybe more than one), that is, the iterator is an integral part of the container object itself, and vice versa.

The iterator attribute data can maintain information about which items have been processed and which have not, as well as providing a standard means of returning information about the items.

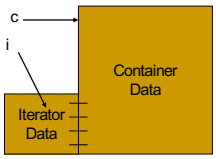
O-O and Iterators

10

Modularizing the data as well as the functionality, and then encapsulating them into an object

The iterator object has no meaning apart from its container

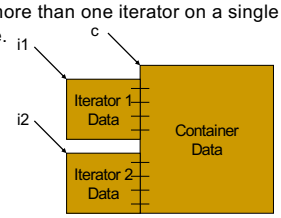
A special class that can be instantiated each time an iterator is needed on any container



O-O and Iterators

11

Can even have more than one iterator on a single container at the one time.

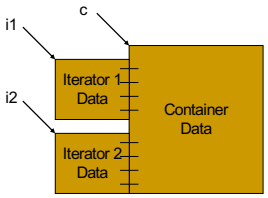


Each iterator remembers where it is up to, independently of other iterators or the container itself.

Referring to an Iterator Object

12

Iterators can be referred to (using a C++ pointer or a Java reference) by any of the types in the inheritance hierarchy of the object, in exactly the same way that any other object can.



In the example:

- c is any container type
- i1 and i2 can be any Iterator types

Polymorphism can still be used

C++ and Java Iterators

C++ and Java view iterators in different ways

C++

- A special object that can OPERATE on the container object in special (but standard) ways using the ++ and --operators.
- This follows through from the idea of C++ as an operator-rich language, e.g. the ++ and -- operators can even be used as an effective array iterator, and this carries through to iterators.

Java

- A special object with a special (well-known) interface that can be requested to respond to messages in the same way that other objects might follow the O-O paradigm
- This follows through from Java being a pure O-O language, so that even iteration through an array can be done using the standard Iterator interface or the generic Iterator<E> interface.
- A collection that is able to supply an iterator (over itself), can do so by implementing the standard Iterable<E> interface.

Java Iterators

JDK provides the standard `Iterator<E>` interface in `java.util`

`Iterator<E>` is now the preferred general interface

- This is `extended` as `ListIterator<E>` and `TreeIterator<E>` for more specific collection types
- Note that an interface can extend another interface

Standard Iterator Interface

The `Iterator<E>` interface provides for the following:

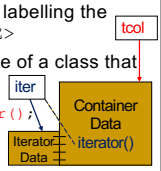
- `remove()` deletes the object most recently accessed by `next()`
  - Though `remove()` need not be implemented

```
public boolean hasNext()

public E next()

public void remove()
```


Iterator Construction & Use

Classes that support iteration must provide an `iterator()` method and this is most often enforced by labelling the copntainer as `implements Iterable<E>`

The `iterator()` method returns an instance of a class that supports the `Iterator<E>` interface:




```
Iterator<MyClass> iter = tcol.iterator();
while (iter.hasNext()) {
   MyClass myObject = iter.next();
   System.out.println(myObject.toString());
}
```

ie. the `iterator()` method returns an object `iter` that is a `Iterator<E>` and supplies `hasNext()` returning a boolean and `next()` returning an object of type MyClass.

Creating an Iterator - 1

```
// The iterator must support the following
//  methods.
//  It may also support remove()


public boolean hasNext()

public E next()
```


Creating an Iterator - 2

`hasNext` has no preconditions

`next` has two preconditions:

- `hasNext` returns true
- the underlying collection has not been modified by one of that collection's mutators during the lifetime of that iterator

Running Out Of Elements

19

```
// The client should be aware that the
// collection can run out of elements
Iterator<MyClass> iter = tcol.iterator();

while (iter.hasNext()) {
   MyClass myObject = iter.next();
   // … … code to process the object accessed …
}
// hasNext() has now returned FALSE

MyClass myObject = iter.next(); // This should cause
            // a NoSuchElementException to be thrown
```


Mutators and Iterators

20

```
// It should not be possible to mutate a
// collection while an iterator is being used
// on it
Iterator<MyClass> iter = tcol.iterator();

while (iter.hasNext()) {
   MyClass myObject = iter.next();
   if ( … some condition … )
      tcol.removeLast();  // This should throw a
            // ConcurrentModificationException the
            // next time iter.next() is called
            // because the collection has been altered
            // been altered independent of the
            // iterator.
}
```


Creating an Iterator - 3

21

```
// Preconditions for next() are as follows:

// Throws a NoSuchElementException if
// hasNext() is false

// Throws a ConcurrentModificationException if
// the iterator's backing store
// (the collection) has been modified
// by the collection's mutators

public E next()
```


Simple Interface & Container

22

```
public interface<E> SimpleI {       // A Simple
      public void AddItem( E );     // Container
      public E TakeItem( );   // Interface
      public int CountItems( );
}
public class SimpleC<E> implements SimpleI<E>,
                                   Iterable<E>, …… {
      private E[] items;
      private int last;
      private int cap;

      public SimpleC(int sz) {
            items = new E[sz];
            last = -1;
            cap = sz;
      }
```


Simple Container (cont)

23

```
      public void AddItem(E o) {
            if (last = cap-1) return;
            last++;
            items[last] = o;
      }
      public E TakeItem( ) {
            if (last == -1) return null;
            E o = items[last];
            items[last] = null;
            last--;
            return o;
      }
      public int CountItems( ) {
            return last+1;
      }
}
```


Creating an Iterator - 4

24

```
import java.util.*;  // To access Iterator<E>
                     // i/f and exceptions

public class SimpleC<E> implements SimpleI<E>,
                                   Iterable<E>,
                                   Cloneble,
                                   Serializable {

  // Code for the SimpleC data, constructor,
  // and methods presented on previous slides.

      // new code goes here
  // Code for the iterator method.
  // Code for the class that implements the
  //   Iterator interface.

}
```

The `iterator()` Method

The `iterator()` method uses `new` to construct an iterator for the container and returns a reference to it

```
public Iterator<E> iterator() { // returns ref of
                                // std type Iterator<E>
   return new SimpleIterator<E>();
}
```

SimpleIterator is a class, which implements the Iterator interface methods.

SimpleIterator is a private inner class, within the SimpleC class, so that only the `iterator()` method can create a SimpleC iterator. Next slide .....

The SimpleIterator Class

Note these well .....

```
private class SimpleIterator<E> implements Iterator<E>
{
   ..........   // attribute data for iterator - see later

   ..........   // constructor for iterator - see later

   public boolean hasNext() {  // stub only
      return false;            // see later
   }
   public E next() {           // stub only
      return null;             // see later
   }
   public void remove() {  // if required
   }
}
```

SimpleIterator is nested within **SimpleC**

Checking Consistency of Access


```
// A SimpleC instance variable used to
// test for concurrent modifications

private int modCount;
```

`modCount` is set to 0 when the collection is created.
`modCount` is incremented whenever the collection is modified by one of its mutators.
`modCount` is compared to the iterator's expected mod count as a precondition for `next` (see next slide)
We therefore need to go back and modify the AddItem( ) and TakeItem( ) methods of class SimpleC.

The SimpleIterator Constructor

Also note ... private constructor

```
// The iterator must privately store its own
// view of modification state within itself

private class SimpleIterator<E> implements Iterator<E> {
   private int curPos, expectedModCount;

   private SimpleIterator<E>(){ // private constructor
      curPos = 0;               // - an iterator can
      expectedModCount = modCount; // only be created
   }                            // by a collection
                                // object from the
                                // correct class.
   // Other methods
}
```


The `hasNext()` Method


```
private class SimpleIterator<E> implements Iterator<E> {

   private int curPos, expectedModCount;

   public boolean hasNext() {
      return curPos <= last;
   }

   // Other methods
}
```


The `next()` Method


```
// Demonstrates implementation of the next() method
private class SimpleIterator<E> implements Iterator<E> {
   private int curPos, expectedModCount;

   public E next() {
      if (modCount != expectedModCount)
         throw new ConcurrentModificationException
            ("Cannot mutate in context of iterator");
      if (! hasNext())
         throw new NoSuchElementException
                  ("There are no more elements");
      E obj = items[curPos];
      curPos++;
      return obj;
   }                    // Other methods follow
}
```

The `remove()` Method

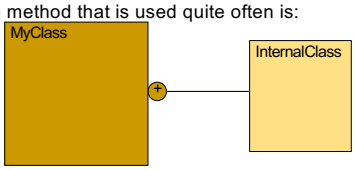

```
private class SimpleIterator<E> implements Iterator<E> {

   private int curPos, expectedModCount;

   public void remove() {
      throw new UnsupportedOperationException
            ("remove not supported by SimpleC");
   }

// exercise – write a meaningful remove() method

   // Other methods
}
```


UML and Internal Classes

Internal Classes are a particularly Java way of implementing certain relationships between tightly coupled classes.

There is no standard way of showing such a relationship because of how "implementation specific" it is.

One method that is used quite often is:



Remember that an important part of the specification is whether the constructor is private or public.

Lists & List Iterators

A lot of what follows will has been updated to line up with Generics in Java.

A list is a linear collection that supports access to any item

Lists are more general-purpose than stacks, queues and deques.

There is no defined set of standard operations, but most lists support some typical ones,

A List Iterator is likely to have a "closer-to-standard" set of operations, but these will still be dependent on the types of access that a List might have.

List Abstract Data Type (ADT) & Behaviour

A list is a linear collection that supports access to any item.

Lists are more general-purpose than stacks, queues and deques.

There is no defined set of standard operations, but most lists support some typical ones.

We will use a prototype List (called ListPT) to explore a list's behavior and how to implement a list. Because the list operations will normally be independent of the type of object they store, we use the generic spec ListPT<E>.

List Operations

List operations may be categorised as:

- Supporting
  - E.g. size(), isFull(), iterator()
- Index-based
  - using an index position
- Content-based
  - using an object
- Position-based
  - moving a current position pointer

Index-based List Operations - 1

`void add(i, E)`	opens up a slot in the list at index `i` and inserts object o in this slot
`E get(i)`	returns the object at index `i`
`E remove(i)`	removes and returns the object at index `i`
`E set(i, E)`	replaces the object at index `i` with the given object and returns the original object

Index-based List Operations - 2

For example, if we have a list that can hold `string`(s), which we know by the reference `list`:

```
// Add some strings
for (int i = 0; i < 5; i++)
   list.add(i, "" + i);

// And display them
for (int i = 0; i < list.size(); i++)
   System.out.println(list.get(i));
```


Content-based List Operations

`void add(E)`	adds object `o` at a list's tail
`boolean contains(E)`	returns `true` if a list contains an object equal to object `o`
`int indexOf(E)`	returns the index of the first instance of object `o` in a list
`boolean remove(E)`	removes the first instance of object **`o`** from a list and returns `true` if `o` is removed, else returns `false`

Position-based List Operations - 1

Are used for navigation or mutation

- Navigation:

`boolean hasNext()`	returns `true` if there are any items following the current position
`E next()`	returns the next item and advances the position
`boolean hasPrevious()`	returns `true` if there are any items preceding the current position
`E previous()`	returns the previous item and moves the position backward
`int nextIndex()`	returns the index of the next item or -1 if none
`int previousIndex()`	returns the index of the previous item or -1 if none

Position-based List Operations - 2

Are used for navigation or mutation

- Mutation:

`add(E)`	inserts object `o` at the current position
`remove()`	removes the last item returned by `next` or `previous`
`set(E)`	replaces the last item returned by `next` or `previous`

Uses of Lists

Lists may be used for many purposes, including:

- object heap storage management
- documents
- files
- implementation of other ADTs

Implementation of Lists

`List<E>`	(interface)
`ArrayList<E>`	(uses dynamic array)
`LinkedList<E>`	(uses doubly linked list)
`Iterator<E>`	(interface)
`ListIterator<E>`	(allows insertions, removals, movement to previous items)

Once we have these classes we can construct (instantiate) list objects of any parameterised type <E>.

E.g. to hold and navigate through a list of strings:

```
List<String> list1 = new LinkedList<String>();
ListIterator<String> iter1 = list1.iterator();
```


Implementation of Lists Prototype - 1

The following provide a cut-down version of the Java List implementations

- Interfaces:
  - `ListPT<E>`
  - `ListIteratorPT<E>`
- Implementation classes:
  - `ArrayListPT<E>`
  - `LinkedListPT<E>`

ListPT<E> Interface

Fundamental Methods

`void add(int i, E o)`
Adds the object o to the list at index i.
Throws an exception if the object o is null or the list is full or if i is out of range (i < 0 || i > size()).

`boolean contains(E o)`
Returns true if the object o is in the list, else returns false.

`E get(int i)`
Returns the object at index i.
Throws an exception if i is out of range (i < 0 || i >= size()).

`int indexOf(E o)`
Returns the index of the first object equal to object o or -1 if there is none.

`E remove(int i)`
Removes and returns the object at index i.
Throws an exception if i is out of range (i < 0 or i >= size()).

`E set(int i, E o)`
Returns the object at index i after replacing it with the object o.
Throws an exception if the object o is null or if i is out of range (i < 0 or i >= size()).

Supporting Methods

`boolean isEmpty()`
Returns true if this list contains no items.

`boolean isFull()`
Returns true if this list is full and can accept no more items.

`int size()`
Returns the number of items in this list.

General Methods

`ListIteratorPT<E> listIterator()`
Returns a list iterator over this list.

ListIteratorPT<E> Interface

Navigation Methods

`boolean hasNext()`
Returns true if there are any items after the current position, else returns false.

`boolean hasPrevious()`
Returns true if there are any items preceding the current position, else returns false.

`E next()`
Returns the item following the current position and advances the current position.
Throws an exception if `hasNext` would return false.

`E previous()`
Returns the item preceding the current position and moves the current position back.
Throws an exception if `hasPrevious` would return false.

Modification Methods

`void add(E o)`
Inserts the object o at the current position.
After insertion, the current position is located immediately after the newly inserted item.
Throws an exception if the object o is null or the list is full.

`void remove()`
Removes the last object returned by next or previous.
Throws an exception if add or remove has occurred since the last next or previous.

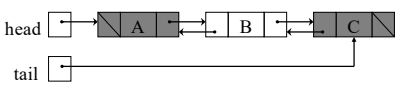
`void set(E o)`
Replaces the last object returned by next or previous with object o.
Throws an exception if add or remove has occurred since the last next or previous.

Linked Implementation of ListPT<E> Interface

Two way nodes (doubly linked) are preferable, because the `iterator` must support movement in both directions

Problem with Linked Implementation

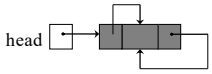
- Additions or removals at either end are special cases and require extra code (setting external pointers to `null`, etc.)



Solution to Problem with Linked Implementation - 1

A solution is to use a circular linked structure with a (single) dummy header node

There will always be a node before the first “data” node and a node after the last “data” node

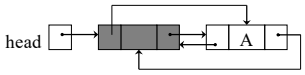


Solution to Problem with Linked Implementation - 2

49

After addition of the first “data” node

- The next pointer of a “data” node is never null
- The previous pointer of a “data” node is never null
- The head pointer never changes
- There is no tail pointer to worry about, but there is still direct access to the last node

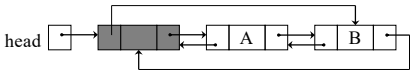


Solution to Problem with Linked Implementation - 3

50

After addition of a second “data” node

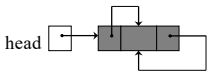
- Note that insertions and removals anywhere are handled in the same way



Data for the Linked Implementation

51

```
public class LinkedListPT<E>
                implements ListPT<E>, Serializable {
   private TwoWayNode head;
   private int size;
   private int modCount;

// Constructor sets up circular linked structure
//                              with a dummy
   header node
   public LinkedListPT() {
      head = new TwoWayNode(null, null, null);
      head.next = head;
      head.previous = head;
      size = 0;
      modCount = 0;
   }
```

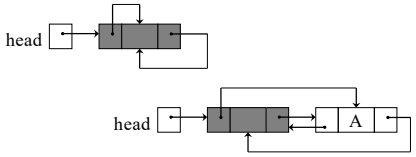


Implementing `add(i, E)`

52

Locate the node at position i - 1

- Operate on that node’s `next` pointer and the new node's `previous` and `next` pointers, e.g. `add(0,E)`

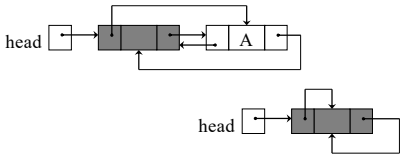


Implementing `remove(i, E)`

53

Locate the node at position i - 1

- Operate on that node’s `next` pointer and the node following the `i`th node's `previous` pointer



Method `getNode(int)` - 1

54

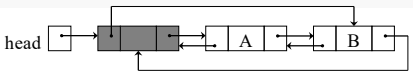
`getNode(int)`

- Searches for the node at the position specified by `int`
- Returns a reference (pointer) to that node
- Is used by all the index-based operations `add()`, `remove()`, `get()`, `set()`

Method getNode(int) - 2

55

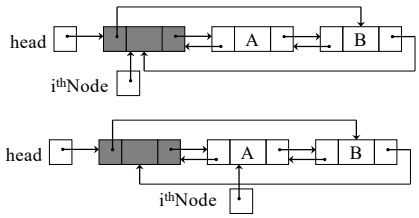
```
// A helper method that returns a reference to the
// ith node of a doubly-linked list

private TwoWayNode getNode (int i)  // why private?
{
   TwoWayNode ithNode = head;
   for (int k = -1; k < i; k++)
      ithNode = ithNode.next;
   return ithNode;
}
```



Method getNode(int) - 3

56

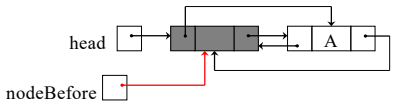
```
// e.g. getNode(0);
```



Method add(int, E) - 1

57

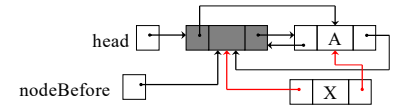
```
// Adds item to list at position specified by index

public void add (int index, E item) {
   // Check preconditions
   …
   // Locate node before insertion point
   TwoWayNode nodeBefore = getNode (index - 1);
```



Method add(int, E) - 2

58

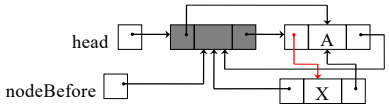
```
// Adds item to list at position specified by index
public void add (int index, E item) {
   // Check preconditions
   …
   // Locate node before insertion point
   TwoWayNode nodeBefore = getNode (index - 1);

   // Create new node and link it into the list
   TwoWayNode newNode = new TwoWayNode(item,
                  nodeBefore, nodeBefore.next);
```



Method add(int, E) - 3

59

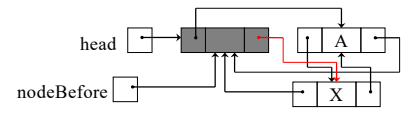
```
public void add (int index, E item) {
   // Check preconditions
   …
   // Locate node before insertion point
   TwoWayNode nodeBefore = getNode (index - 1);

   // Create new node and link it into the list
   TwoWayNode newNode = new TwoWayNode(item,
                  nodeBefore, nodeBefore.next);
   nodeBefore.next.previous = newNode; // fix ptr
```



Method add(int, E) - 4

60

```
public void add (int index, E item) {
   // Check preconditions
   …
   // Locate node before insertion point
   TwoWayNode nodeBefore = getNode (index - 1);
   // Create new node and link it into the list
   TwoWayNode newNode = new TwoWayNode(item,
                        nodeBefore, nodeBefore.next);
   nodeBefore.next.previous = newNode;
   nodeBefore.next = newNode; // fix pointer
```



Method add(int, E) - 5

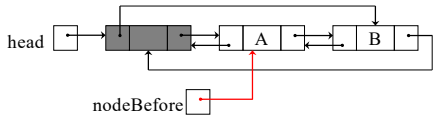
61

```
public void add (int index, E item) {
    // Check preconditions
    …
    // Locate node before insertion point
    TwoWayNode nodeBefore = getNode (index - 1);
    // Create new node and link it into the list
    TwoWayNode newNode = new TwoWayNode(item,
                         nodeBefore, nodeBefore.next);
    nodeBefore.next.previous = newNode;
    nodeBefore.next = newNode;
    // Adjust List instance data to reflect addition
    size++;
    modCount++;
}
```


remove(int) - 1

62

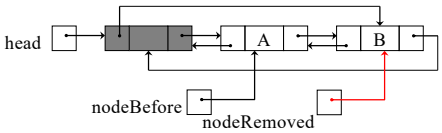
```
//Removes the node and item at the indicated index
public E remove(int index) {
    // Check preconditions
    …
    //Locate the node before the one being deleted
    TwoWayNode nodeBefore = getNode(index - 1);
```



remove(int) - 2

63

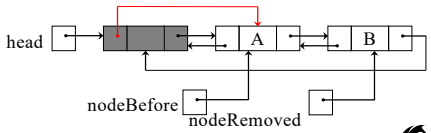
```
//Removes the node and item at the indicated index
public E remove(int index) {
    // Check preconditions
    …
    //Locate the node before the one being deleted
    TwoWayNode nodeBefore = getNode(index - 1);
    //Remember the node about to be removed
    TwoWayNode nodeRemoved = nodeBefore.next;
```



remove(int) - 3

64

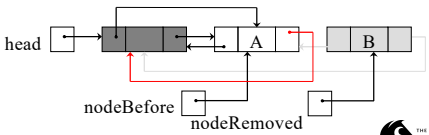
```
public E remove(int index) {
    // Check preconditions
    …
    //Locate the node before the one being deleted
    TwoWayNode nodeBefore = getNode(index - 1);
    //Remember the node about to be removed
    TwoWayNode nodeRemoved = nodeBefore.next;
    //Link around the removed node
    nodeRemoved.next.previous = nodeBefore; // Step 1
```



remove(int) - 4

65

```
public E remove(int index) {
    // Check preconditions
    …
    //Locate the node before the one being deleted
    TwoWayNode nodeBefore = getNode(index - 1);
    //Remember the node about to be removed
    TwoWayNode nodeRemoved = nodeBefore.next;
    //Link around the removed node
    nodeRemoved.next.previous = nodeBefore;
    nodeBefore.next = nodeRemoved.next; // Step two
```



remove(int) - 5

66

```
public E remove(int index) {
    // Check preconditions
    …
    //Locate the node before the one being deleted
    TwoWayNode nodeBefore = getNode(index - 1);
    //Remember the node about to be removed
    TwoWayNode nodeRemoved = nodeBefore.next;
    //Link around the removed node
    nodeRemoved.next.previous = nodeBefore;
    nodeBefore.next = nodeRemoved.next;

    // Finish off by fixing instance data
    size--;
    modCount++;
    return nodeRemoved.value;
}
```

List Iterators

An iterator over a list:

- Supports extended navigation – can move to `previous` as well as `next`
- Supports extended mutation – can replace and insert as well as remove

Position-based (Iterator) Operations for Navigation

`boolean hasNext()`	returns `true` if there are any items following the current position
`E next()`	returns the next item and advances the position
`boolean hasPrevious()`	returns `true` if there are any items preceding the current position
`E previous()`	returns the previous item and moves the position backward
`int nextIndex()`	returns the index of the next item or -1 if none
`int previousIndex()`	returns the index of the previous item or -1 if none

Position-based (Iterator) Operations for Mutation

`add(E)`	inserts object `E` at the current position
`remove()`	removes the last item returned by `next` or `previous`
`set(E)`	replaces the last item returned by `next` or `previous`

Using a List Iterator


```
ListPT<MyClass> list = new LinkedListPT<MyClass>( )
                                    // Create a list
ListIteratorPT<MyClass> iter = list.iterator();
                                // Open an iterator on it
for (int i = 1; i <= 10; i++)     // Insert some strings
   iter.add("" + i);

while (iter.hasNext())            // Move forwards
   System.out.println(iter.next());

while (iter.hasPrevious())        // Move backwards
   System.out.println(iter.previous());

while (iter.hasNext()) {          // Replace all strings
   iter.next();
   iter.set("");
}
```


The `iterator()` Method


```
public ListIteratorPT<E> iterator() {
   return new ListIter<E>();
}
```

`ListIter` is a private inner class in both array-based and linked implementations

Array-based `ListIter` - 1


```
private class ListIter<E> implements ListIteratorPT<E>,
                                     Serializable {

   private int curPos;     // Logically current position
                           // is just before this node

   private int lastItemPos;
      // Equals index of last item returned by next or
      // previous.  Equals -1 initially and after add
      // and remove

   private int expectedModCount;

   private ListIter() {
      curPos = 0;
      lastItemPos = -1;
      expectedModCount = modCount;
   }
```


Array-based ListIter - 2


```
public boolean hasNext() {
    return curPos < size;
}

public boolean hasPrevious() {
    return curPos > 0;
}
```


Array-based ListIter - 3


```
public E next() {
     // Check preconditions
                        // The current position is
                        // logically just before the
                        // node pointed to by curPos
    lastItemPos = curPos; // Store index of the last
                        // item returned
    curPos++;           // Advance the current position
    return items[lastItemPos];
}

public E previous() {
     // Check preconditions
     lastItemPos = curPos - 1;
     curPos--;
     return items[lastItemPos];
}
```


Array-based ListIter - 4


```
public void add(E o) {
// Adds object o at the current position

// You should write the code for this method yourselves.
// Throw an exception if expectedModCount != modCount
// Make the addition at curPos via the backing list's
//     add method.
// That method will increment modCount or will throw an
//     exception if object o is null or the list is full
// Increment curPos and expectedModCount
// Set lastItemPos equal to -1
}
```


Array-based ListIter - 5


```
public void remove() {
   if (modCount != expectedModCount)
      throw new ConcurrentModificationException();
   if (lastItemPos == -1)
      throw new IllegalStateException
              ("There is no established item to remove.");
   ArrayListPT.this.remove(lastItemPos); // Call the
             // backing list's remove method, which will
             // increment modCount
   expectedModCount++; // Inc expectedModCount as well
   if (lastItemPos < curPos) // If the item removed was
      curPos--;       // obtained via next, then move the
                      // current position back
   lastItemPos = -1;  // Block remove and set until after
                      // a successful next or previous
}
```