School of Electrical Engineering and Computing

SEENG2200/6220 PROGRAMMING LANGUAGES & PARADIGMS (S1, 2020)

Iterators

Dr Nan Li
Office: ES222
Nan.Li@newcastle.edu.au

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Outline

- Collections
  - *Data structures*
  - *Algorithms*
- Iterators
- Implementation
  - *Iterators*
  - *LinkedListPT*

Collections and Containers

- A collection is simply an object that groups multiple elements into a single unit.
- The term collection gives the connotation of a special type of organization within the container.
- Linear
  - *Arrays, Stacks, Queues & Deques, Lists, etc.*
- Hierarchical
  - *Trees, General Tree, Binary Tree, Heap*
- Graph
  - *Undirected Graph, Directed Graph*
- Unordered
  - *Set, Bag, Map (table)*

Collections Framework

- A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.
  - *Reduces programming effort – provides standard data structures and algorithms.*
  - *Increases performance – optimised design and high-performance implementation of data structures and algorithms.*
  - *Reduces the effort required to design and implement APIs.*
  - *Software reuse*

Example - 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 organising features*
- Lists
  - *General insertion and deletion*

Iterators and Collections

- Iterators, basically, are used to access every item of a collection.

- It does not need to follow a specific (visiting) order, but if a specific ordering exists, it may make the iterator's task easier when it follows that ordering.

- Iterators need extra data about the particular collection.
  - *e.g., the items have been visited.*

- 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

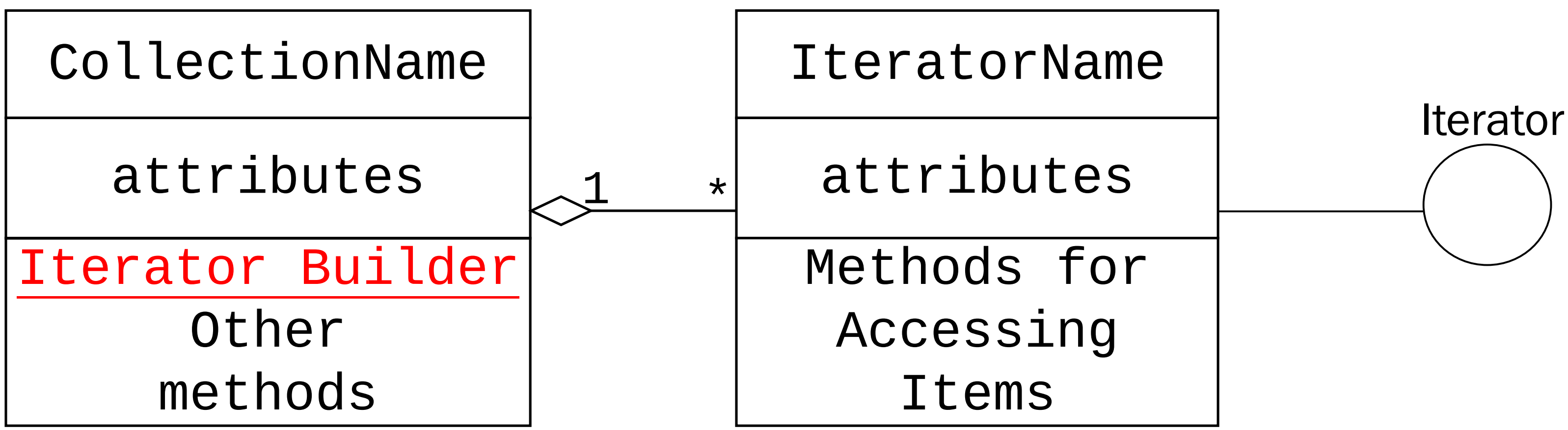
- <u>Test</u> whether there are more items to be visited.
- <u>Visit</u> (look at) the next item.
  - *Obtain access to the next item via the iterator*
- <u>Remove</u> items from the container.
  - *Not an essential part of all iterators*
- <u>Check</u> consistency
  - *The container itself is still 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

- The relationship between a collection and its iterator is Aggregation, a Collection can HAVE any number of iterators.
  - *i.e there is not the same 1 to 1, or lifetime equivalence nature in this relationship, when compared to the composition relationship.*
- But 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.

UML Modelling of Iterators

- There may be a number of ways to draw these relationships in UML, for example

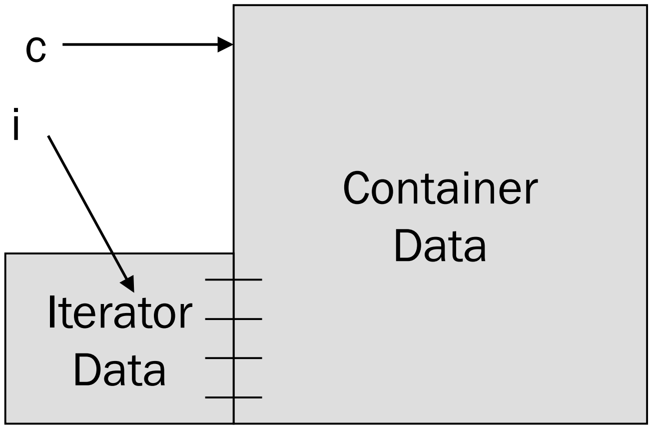


How to construct an iterator

- An iterator IS an object – but a container object HAS AN iterator, 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 <u>have been processed</u> and which have not, as well as providing a standard means of returning information about the items.

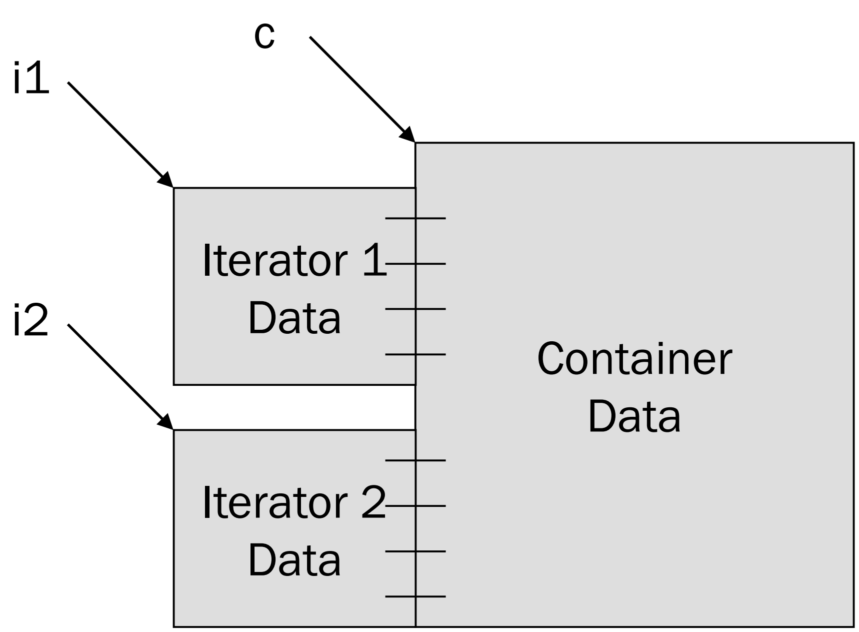
O-O and Iterators

- 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



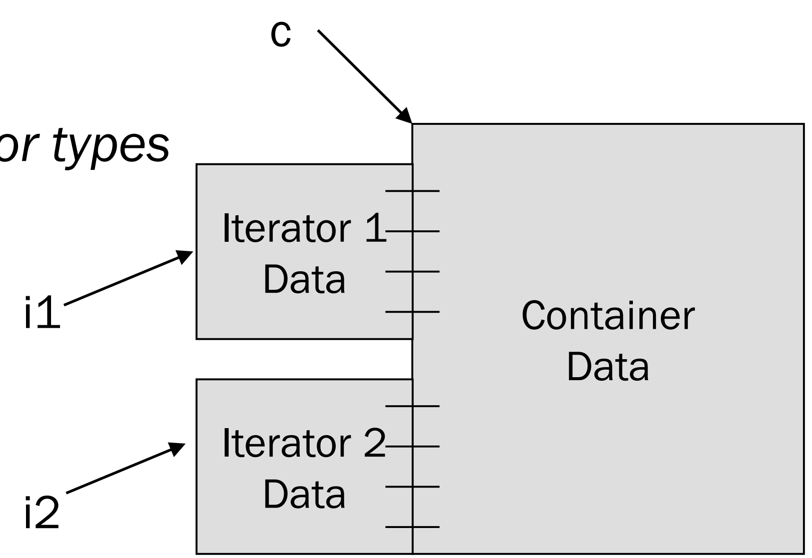
O-O and Iterators

- For multiple iterators of a container, each iterator remembers where it is up to, independently of other iterators or the container itself.



Referring to an Iterator Object

- 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 works



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, such 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 methods:

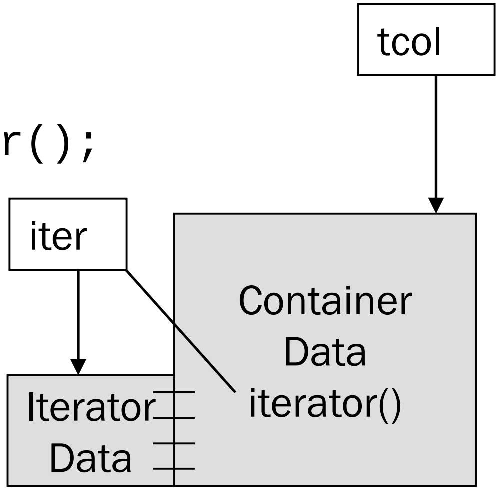
```
public boolean hasNext()
public E next()
public void remove()
```

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

Iterator Construction

- Classes that support iteration must provide an `iterator()` method and this is most often enforced by labelling the container as `implements Iterable<E>`
- The `iterator()` method returns an instance of a class that supports the `Iterator<E>` interface:

```
Iterator<MyClass> iter = tcol.iterator();
```



Use Iterator

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

```
while (iter.hasNext()) {
    MyClass item = iter.next();
    System.out.println(item.toString());
}
```

Creating an Iterator

- The iterator must support the following methods, while it may also support remove() method

```
public boolean hasNext()

public E next()
```

Creating an Iterator

- `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*

hasNext()

- 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();
    // do something
}
```

- If hasNext() has returned FALSE

```
MyClass myObject = iter.next();
```

  - *Should throw a NoSuchElementException*

next()

- Preconditions
  - *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()`

Mutators and Iterators

- 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();
}
myObject = iter.next();
```

- This should throw a ConcurrentModificationException at the second `iter.next()` call, because the collection has been altered independent of the iterator `iter`.

Simple Container

```
public interface<E> SimpleI {// A simple container Interface
      public void AddItem(E e);
      public E TakeItem( );
      public int CountItems( );
}

public class SimpleC<E> implements SimpleI<E>,Iterable<E>{
      private final Object[] items;
      private int last;
      private int cap;

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

Simple Container

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

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() {
   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.

The SimpleIterator Class

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

    ...........    // attribute data for iterator - see later

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

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

SimpleIterator is nested within SimpleC

Checking Consistency of Access

- A SimpleC instance variable is 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`()

We therefore need to go back and modify the ***`AddItem`*** *() and* ***`TakeItem`*** *() methods of class SimpleC.*

`SimpleIterator` 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(){
        curPos = 0;
        expectedModCount = modCount;
    }
    // Other methods
}
```

- Private constructor – an iterator can only be created by a collection object from the correct class.

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

```
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
}
```

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");
    }
    // Other methods
}
```

Exercise: write a meaningful remove() method.

Lists and 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 Operations

- 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 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

`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

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

Position-based List 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 List Operations

- For 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 abstract data types*

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, movements)

- Once we have these classes we can construct (instantiate) list objects of any parameterised type.
- E.g. to hold and navigate through a list of strings:

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

Implementation of List Prototype

- 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()).

ListPT<E> Interface

<u>Supporting Methods</u>

`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.

<u>General Methods</u>

`ListIteratorPT<E> listIterator()`

Returns a list iterator over this list.

ListIteratorPT<E> Interface

<u>Navigation Methods</u>

`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.

ListIteratorPT<E> Interface

<u>Modification Methods</u>

`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.

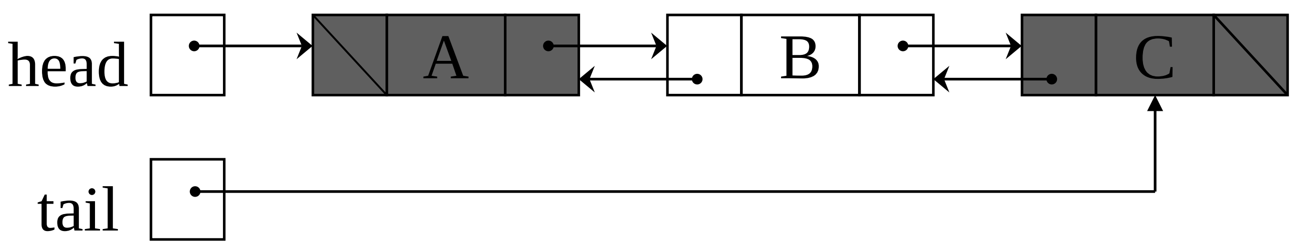
`void set(E o)`

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.

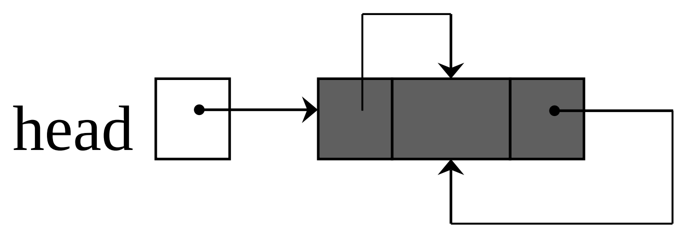
A Problem

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



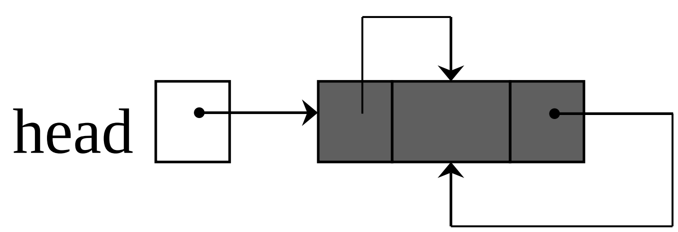
A Solution

- 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



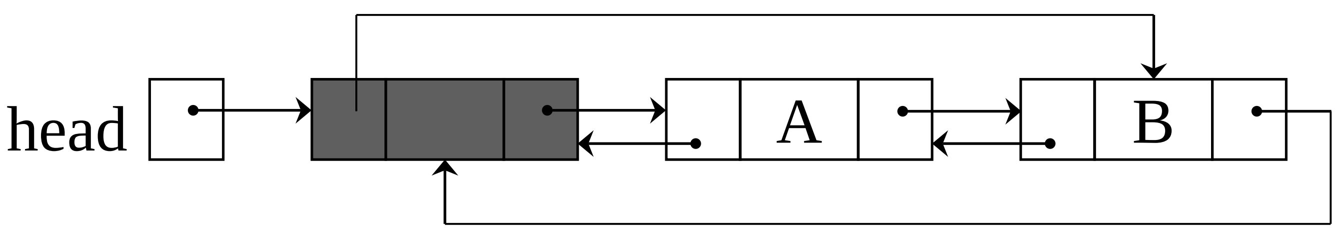
A Solution

- 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*



A Solution

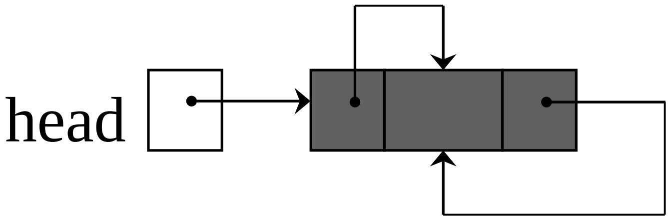
- After addition of a second "data" node
  - *Note that insertions and removals anywhere are handled in the same way*



Data for the Linked Implementation

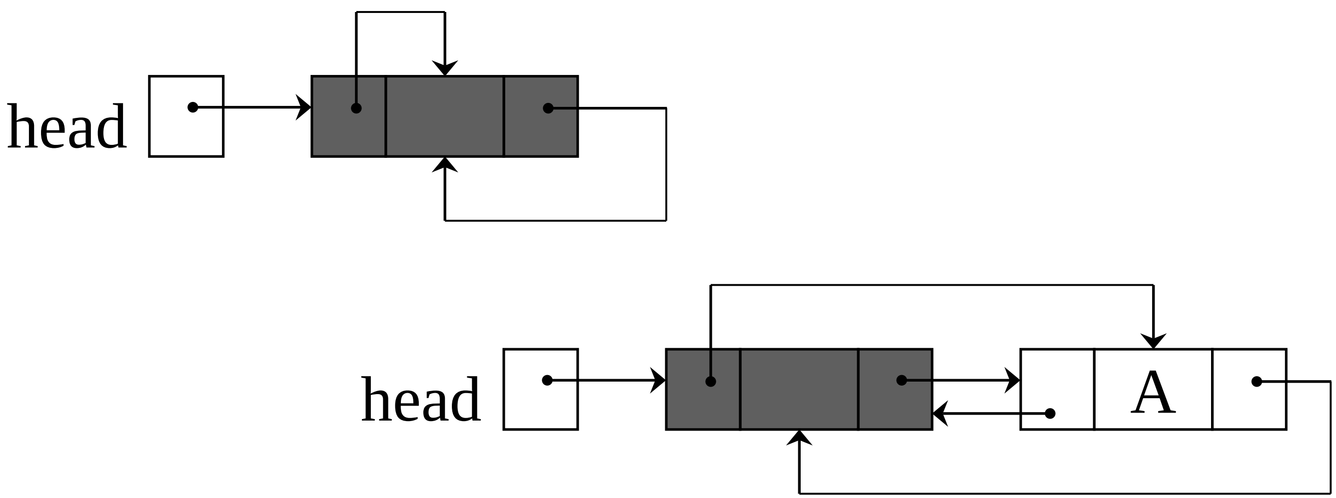
```
public class LinkedListPT<E> implements ListPT<E>,
                                        Serializable {
   private TwoWayNode head;
   private int size;
   private int modCount;

   public LinkedListPT() {
      head = new TwoWayNode(null, null, null);
      head.next = head;
      head.previous = head;
      size = 0;
      modCount = 0;
   }
```



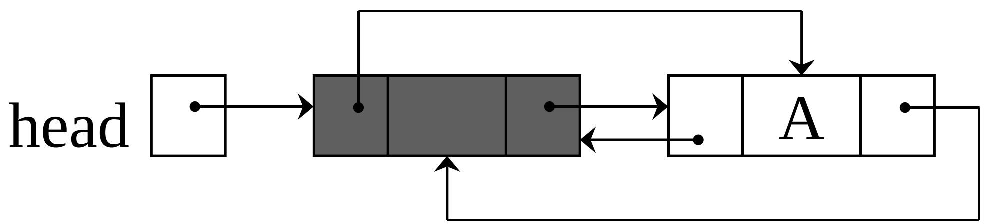
The `add(i,E)` Method

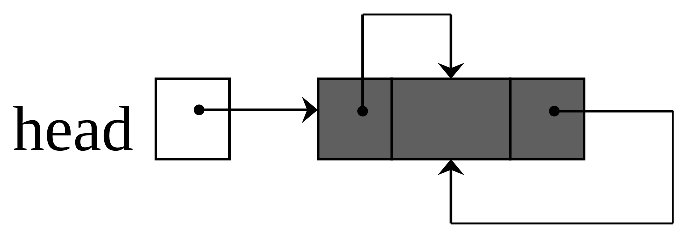
- Locate the node at position `i - 1`
  - *Operate on that node's* `next` *pointer and the new node's* `previous` *and* `next` *pointers, e.g.* `add(0,E)`



The remove(i,E) Method

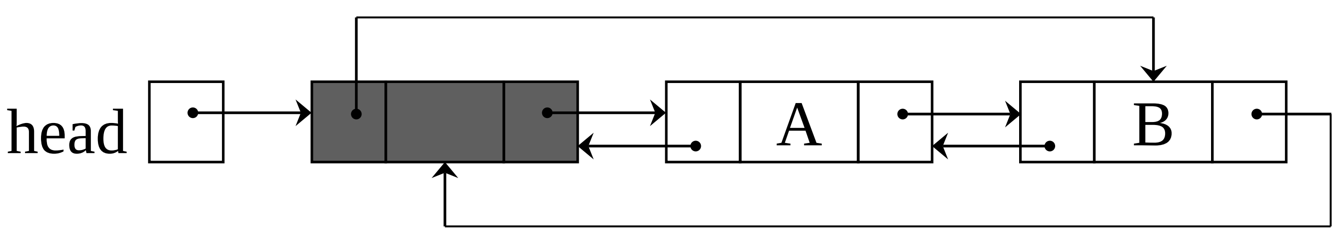
- Locate the node at position `i - 1`
  - *Operate on that node's* `next` *pointer and the node following the* `i`*th node's* `previous` *pointer*





The `getNode(int)` Method

- 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
  - *E.g., `add()`, `remove()`, `get()`, `set()`*

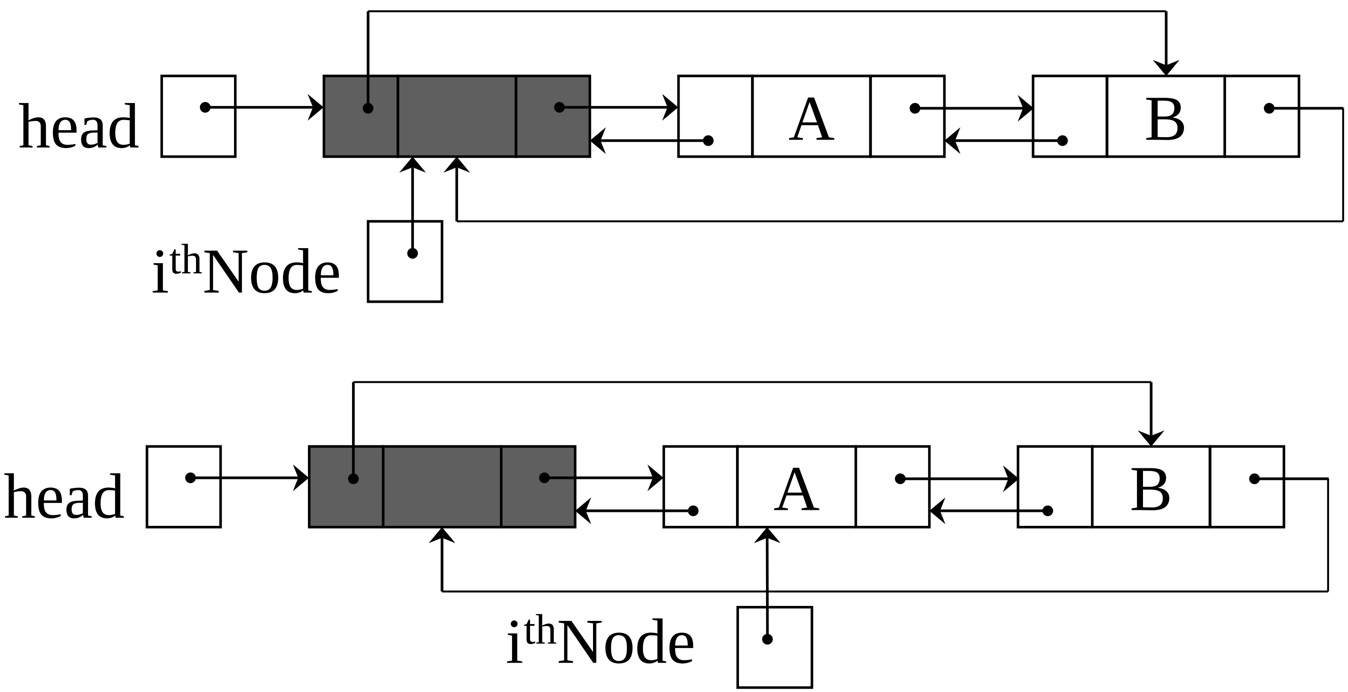


The getNode(int) Method

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

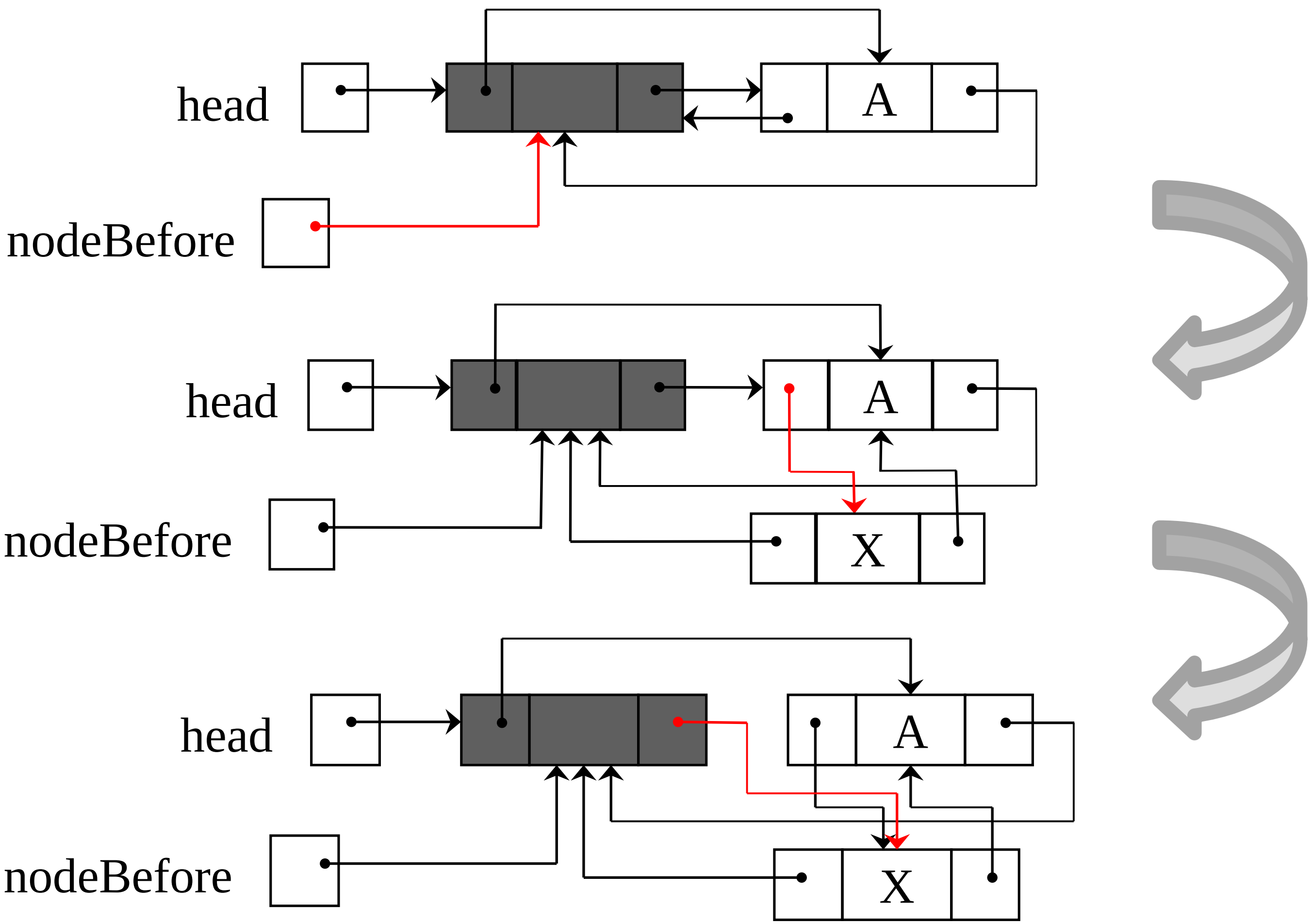
```
private TwoWayNode getNode(int i)
{
    TwoWayNode ithNode = head;
    for (int k = -1; k < i; k++)
        ithNode = ithNode.next;
    return ithNode;
}
```

The getNode(int) Method



getNode(0)

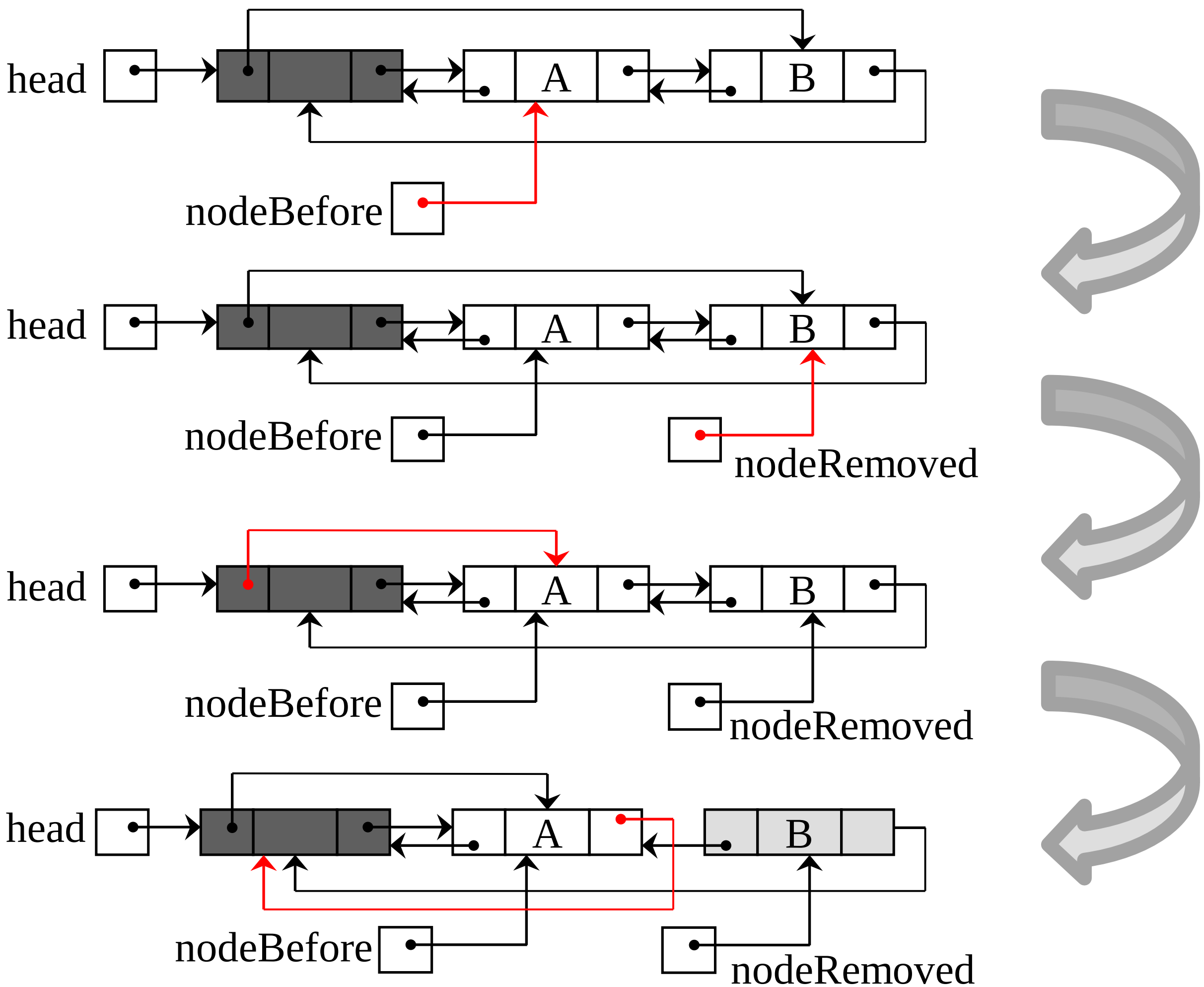
The add(int,E) Method



The add(int,E) Method

```
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++;
}
```

The remove(int) Method



The remove(int) Method

```
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

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

Position-based 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 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.

Exercise: How to implement an Array-based iterator?