Iterators

Professor Frank Kreimendahl

School of Computing and Data Science

Wentworth Institute of Technology

September 19, 2022

Iterators

Iterator Definition

- An iterator is an object designed to traverse a data structure in a standardized way
- An iterator keeps track of a position in a specific data structure
- A specific iterator always refers to the data object it was initialized to
- A data object can have multiple independent iterators that all refer to it

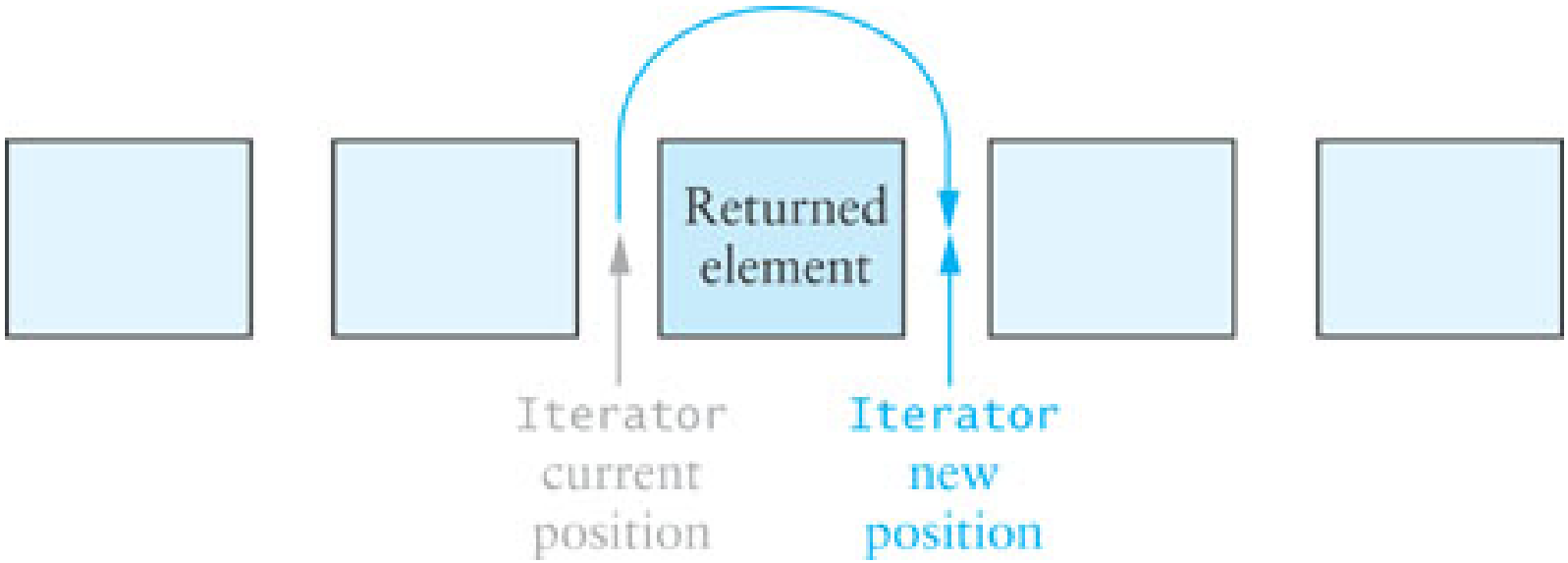
Method	Behavior
`boolean hasNext()`	Returns `true` if the `next` method returns a value.
`E next()`	Returns the next element. If there are no more elements, throws the `NoSuchElementException`.
`void remove()`	Removes the last element returned by the `next` method.

Iterator interface

List Iterators

- The iterator in a list functions like a cursor symbol in text: it resides between two elements
- Iterators only move if you call `.next()` on them
- The Iterator is defined within a List class
- For a list, calling `.next()` repeatedly has the same result as looping through indices in an array
- Iterators still work on data structures that don't map integers to values (which we will see later)

Iterator Changing Positions



Result of calling `.next()` on an Iterator in a list

Iterator Usage Example

Iterating with Iterator

```
Iterator<String> iter = aList.iterator();
while (iter.hasNext()) {
  String value = iter.next();
  // Do something with value
}
```

Iterating with int index

```
int size = aList.size();
for (int i = 0; i < size; ++i) {
  String value = aList.get(i);
  // Do something with value
}
```

Remove Operation

- The `.remove()` call removes an item from a list after you access the item
- `.remove()` deletes the most recently returned element
- You must call `.next()` before each `.remove()` call, or an exception is thrown

Remove Example

Removing all strings that start with a certain char

```
public static void removeChar(LinkedList<String>
                                    aList, char c) {
  Iterator<String> iter = aList.iterator();
  while (iter.hasNext()) {
    String nextStr = iter.next();
    if (nextStr.charAt(0) == c)
      iter.remove();
  }
```

ListIterator

ListIterator Definition

- `ListIterator` is a subinterface of `Iterator`
- `ListIterator` extends requirements of the `Iterator` iterface
- `Iterator` interface requires fewer methods and applies to a wider range of data structures

ListIterator Interface

Method	Behavior
`void add(E obj)`	Inserts object `obj` into the list just before the item that would be returned by the next call to method `next` and after the item that would have been returned by method `previous`. If method `previous` is called after `add`, the newly inserted object will be returned.
`boolean hasNext()`	Returns `true` if `next` will not throw an exception.
`boolean hasPrevious()`	Returns `true` if `previous` will not throw an exception.
`E next()`	Returns the next object and moves the iterator forward. If the iterator is at the end, the `NoSuchElementException` is thrown.
`int nextIndex()`	Returns the index of the item that will be returned by the next call to `next`. If the iterator is at the end, the list size is returned.
`E previous()`	Returns the previous object and moves the iterator backward. If the iterator is at the beginning of the list, the `NoSuchElementExcepton` is thrown.
`int previousIndex()`	Returns the index of the item that will be returned by the next call to `previous`. If the iterator is at the beginning of the list, −1 is returned.
`void remove()`	Removes the last item returned from a call to `next` or `previous`. If a call to `remove` is not preceded by a call to `next` or `previous`, the `IllegalStateException` is thrown.
`void set(E obj)`	Replaces the last item returned from a call to `next` or `previous` with `obj`. If a call to `set` is not preceded by a call to `next` or `previous`, the `IllegalStateException` is thrown.

A larger interface than a general `Iterator`

ListIterator Constructors

Method	Behavior
`public ListIterator<E> listIterator()`	Returns a `ListIterator` that begins just before the first list element.
`public ListIterator<E> listIterator(int index)`	Returns a `ListIterator` that begins just before position `index`.

Two possible ways to construct an Iterator

`for` Loop with Iterators

Standard index iteration

```
count = 0;
for (int i = 0; i < myList.size(); ++i) {
  nextStr = myList.get(i);
  if (target.equals(nextStr))
    count++;
}
```

Equivalent Implicit Iterator

```
count = 0;
for (String nextStr : myList) {
  if (target.equals(nextStr))
    count++;
}
```

`for` Statement

- The `for` statement constructs an Iterator on `myList` (or whatever iterable object is to the right of the : symbol)
- `.hasNext()` and `.next()` are implicitly called each time through the loop
- `.remove()` is not available since the Iterator has no name

KWListIter

ListIterator Implementation on Double-Linked Lists

```
private class KWListIter implements
                    ListIterator<E> {
  private Node <E> nextItem;
  private Node <E> lastItemReturned;
  private int index = 0;
  // methods go here
}
```

ListIterator Constructor

```
public KWListIter(int i) {
  // Validate i parameter.
  if (i < 0 || i > size)
    throw new IndexOutOfBoundsException("Error: " + i);
  lastItemReturned = null; // No item returned yet.
  // Special case of last item
  if (i == size) {
    index = size;
    nextItem = null;
  }
  else { // Start at the beginning
    nextItem = head;
    for (index = 0; index < i; index++)
      nextItem = nextItem.next;
  }
}
```

.hasNext() and .next()

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

```
public E next() {
  if (!hasNext())
    throw new NoSuchElementException();
  lastItemReturned = nextItem;
  nextItem = nextItem.next;
  index++;
  return lastItemReturned.data;
}
```

.hasPrevious() and .previous()

```
public boolean hasPrevious() {
  return (nextItem == null && size != 0)
         || nextItem.prev != null;
}
```

```
public E previous() {
  if (!hasPrevious())
    throw new NoSuchElementException();
  if (nextItem == null) // Beyond list end
    nextItem = tail;
  else
    nextItem = nextItem.prev;
  lastItemReturned = nextItem;
  index--;
  return lastItemReturned.data;
}
```

.add(E)

- Inserts object just before the `next` element
- Four cases to consider when adding an element:
  - adding to empty list
  - adding to head of list
  - adding to tail of list
  - adding to interior of list
- Treat four cases as separate code blocks

.add(E)

```
public void add(E obj) {
  // four cases
  size++;
  index++;
  lastItemReturned = null;
}
```

.add(E) – empty list

```
if (head == null) { // Add to an empty list.
  head = new Node<>(obj);
  tail = head;
}
```

.add(E) – add to head


```
else if (nextItem == head) { // Insert at head.
  // Create a new node.
  Node<E> newNode = new Node<>(obj);
  // Link it to the nextItem.
  newNode.next = nextItem; // Step 1
  // Link nextItem to the new node.
  nextItem.prev = newNode; // Step 2
  // The new node is now the head.
  head = newNode; // Step 3
}
```

.add(E) – add to tail


```
else if (nextItem == null) { // Insert at tail.
  // Create a new node.
  Node<E> newNode = new Node<>(obj);
  // Link the tail to the new node.
  tail.next = newNode; // Step 1
  // Link the new node to the tail.
  newNode.prev = tail; // Step 2
  // The new node is the new tail.
  tail = newNode; // Step 3
}
```

.add(E) – add to middle


```
else { // Insert into the middle.
  // Create a new node.
  Node<E> newNode = new Node<>(obj);
  // Link it to nextItem.prev.
  newNode.prev = nextItem.prev; // Step 1
  nextItem.prev.next = newNode; // Step 2
  // Link it to the nextItem.
  newNode.next = nextItem; // Step 3
  nextItem.prev = newNode; // Step 4
}
```

Inner Classes – Static vs. Nonstatic

- `KWLinkedList` contains two inner classes: `Node` and `KWListIter`
- `Node` is declared `static` because it never accesses data fields of its outer class
- `KWListIter` cannot be declared `static` because its methods access and modify data fields in its outer class
- An inner class which is not static can reference fields of its outer class