Computer Science

Professor Tom Ellman
Lecture 17
package java.util;

public interface Iterator<D> {
    boolean hasNext();
    D next();
    void remove();
}

An **Iterator** provides sequential access to the elements of a container.
Iterable Interface

package java.lang;

public interface Iterable<D> {
    Iterator<T> iterator();
}

A container is Iterable if it has a method that returns an Iterator for the container.
public interface List<D> extends Iterable<D>{
    void add(D data);
    D remove (D data);
    boolean contains(D data);
    int size();
    boolean isEmpty();
    void clear();
    void insert(D d, int index);
    void delete(int index);
    D elementAt(int index);
}

Now we modify our List interface to require that all lists implement the Iterable interface.
public class ListSL<D> implements List<D>{

    //...

    public Iterator<D> iterator() {
        return new ListIteratorSL<D>(this);
    }
}

Our ListSL class will implement the Iterable interface by including an iterator method that returns a ListIteratorSL object implementing the Iterator interface.
ListIteratorSL

- An iterator for the ListSL class.
- ListIteratorSL stores:
  - The ListSL over which it is iterating.
  - The current ListElementSL, which holds the next datum to be returned.
- ListSL now needs an iterator method that returns an iterator for the ListSL.
ListIteratorSL

ListSL
  - count: 3
  - head: 

ListIteratorSL
  - list: 
  - current: 

next data

next data

next data

ParkingSpace

ParkingSpace

ParkingSpace
package utilities;
import java.util.Iterator;

public class ListIteratorSL<D> implements Iterator<D> {

    private ListSL<D> list;
    private ListElementSL<D> current;

    ListIteratorSL(ListSL<D> list) {
        this.list = list;
        this.current = list.head;
    }

    // Omitted ...
}

The variable `list` stores the `ListSL` over which we are iterating. The variable `current` stores the `ListElementSL` containing the next datum to be returned. Why do we need to store the `list` itself in the `ListIteratorSL` object?
public boolean hasNext() {
    return current != null;
}

public D next() {
    D nextItem = current.data;
    current = current.next;
    return nextItem;
}

public void remove() {
    // Not Implemented.
}

The iterator has more data to generate, i.e., `hasNext` returns `true`, as long as `current` is not `null`. The `next` method must return the `data` in the `current` element, after moving `current` one step down the linked list. Since the `remove` method is optional, we choose not to implement it. (But we might later.)
Using an Iterator to Access a Container

Iterator<ParkingSpace> iter = free.iterator();
while (iter.hasNext()) {
    space = iter.next();
    println(space.toString());
}

We can use the iterator to access the contents of a container, such as the ListSL<ParkingSpace> free of ParkingSpace objects. First we invoke free.iterator() to get an iterator for the container. Then we use the iterator’s hasNext and next methods to access data in the free list.
Java for-loop with Iterable Containers

```java
for (ParkingSpace space : free) {
    println(space.toString());
}
```

Since `free` is a `ListSL<ParkingSpace>` variable, Java will use the `iterator` method of the `ListSL<ParkingSpace>` class to obtain a `ListIteratorSL<ParkingSpace>` object. Then it will use the `isEmpty` and `next` methods to successively set `space` equal to each `ParkingSpace` in the `free` container. The loop terminates when the iterator’s `isEmpty` method returns `true`. 
private void printStatus(List<ParkingSpace> free, 
        List<Association<String, ParkingSpace>> rented) {

    if (free.isEmpty()) println("Free Parking ParkingSpaces: None.");
    else {
        println("Free Parking ParkingSpaces: ");
        for (ParkingSpace space : free) {
            println(space.toString());
        }
    }

    if (rented.isEmpty()) println("Rented Parking ParkingSpaces: None.");
    else {
        println("Rented Parking ParkingSpaces: ");
        for (Association<String, ParkingSpace> association : rented) {
            println(association.toString());
        }
    }
}

Code in boxes illustrates a special version of the for-loop that works with Iterable containers.
Iterating Over a **FlexibleArray**

- Iterator should give access to all non-zero entries in the **FlexibleArray**.
- Keep track of the current array index in the iteration process.
- Need to move current to index of next non-zero element on each iteration step.
The variable `array` stores the `FlexibleArray` over which we are iterating. The variable `current` stores the index of the next datum to be returned. It is initialized to be the lowest indexed element in the `FlexibleArray`. 
public boolean hasNext() {
    return current < array.high;
}

public Object next() {
    int old = current;
    do {
        current++;
    } while (array.retrieve(current)==0 && current<array.high);
    return array.retrieve(old);
}

public void remove() {
}

The iterator has more data to generate, i.e., **hasNext** returns **true**, as long as **current** is less than **array.high**. The **next** method must return element retrieved at the **current** index. Also, **current** must be moved up to the index of the next non-zero entry in the array, if there is one, or else up to the end of the array.
Iterating Over a Binary Search Tree

• Keep track of the current node in the iteration process.
• Need to move current up and down the tree.
• How can we move up the tree?
  – Store pointer to parent in BSTreeNode class.
  – Is there an alternative?
The variable `bsTree` stores the `BSTree` over which we are iterating. The variable `current` stores the `BSTreeNode` containing the next datum to be returned. Why do we need to store the `bsTree` itself in the `BSTreeIterator` object?
public BSTreeIterator(BSTree<T> bsTree) {
    this.bsTree = bsTree;
    this.current = bsTree.root;
    while (current!=null && current.left!=null) {
        current = current.left;
    }
}

The current node must be initialized to be the first element in the BSTree ordering. Start with current at the root and move it down and left as far as possible.
The iterator has more data to generate, i.e., `hasNext` returns `true`, as long as `current` is not `null`. The `next` method must return the `data` in the `current` element, after moving `current` to the next node in the tree. Since the `remove` method is optional, we choose not to implement it. (But we might later.)
private BSTreeNode<T> nextNode(BSTreeNode<T> current) {
    if (current != null && current.right != null) {
        current = current.right;
        while (current.left != null) current = current.left;
    } else {
        while (current != null && !current.isLeft()) current = current.parent;
        if (current != null) current = current.parent;
    }
    return current;
}
Is there an alternative?

• Yes!
• Use a stack.
• Store the path back to the root of the tree.
The variable `bsTree` stores the `BSTree` over which we are iterating. The variable `current` stores the `BSTreeNode` containing the next datum to be returned. The variable `parentStack` stores the path from `current` to the `BSTree` root. Why use a stack?
public BSTreeIterator(BSTree<T> bsTree) {
    this.bsTree = bsTree;
    this.current = bsTree.root;
    this.parentStack = new StackListSL<BSTreeNode<T>>()
    while (current!=null && current.left!=null) {
        parentStack.push(current);
        current = current.left;
    }
}

The current node must be initialized to be the first element in the BSTree ordering. Start with current at the root and move it down and left as far as possible. Each time we move current from a node to it’s child, we push the parent node on to the parentStack.
public boolean hasNext() {
    return current!=null;
}

public T next() {
    T data = current.data;
    current = nextNode(current);
    return data;
}

public void remove() {
}

The iterator has more data to generate, i.e., hasNext returns true, as long as current is not null. The next method must return the data in the current element, after moving current to the next node in the tree. Since the remove method is optional, we choose not to implement it. (But we might later.)
private BSTreeNode<T> nextNode(BSTreeNode<T> current) {
    if (current!=null && current.right!=null) {
        parentStack.push(current);
        current = current.right;
        while (current.left!=null){
            parentStack.push(current);
            current = current.left;
        }
    } else {
    while (!parentStack.isEmpty() && isRight(current)) {
        current = parentStack.pop();
    }
    if (!parentStack.isEmpty()) {
        current = parentStack.pop();
    } else {
        current = null;
    }
}
    return current;
}

If the current node has a right child, we move to the root of the right sub-tree and then move current down and left as far as possible. (As we move current from parent to child, we push the parent on to the parentStack.) Otherwise, we move up the tree until reaching an ancestor node that is itself a left child, and set current to that ancestor’s parent. (As we move current from child to parent, we obtain the parent by popping the parentStack.)
Why shouldn’t we be surprised that a stack is useful to traverse a binary search tree in order?

Have we ever used a stack for this purpose before?
package utilities;
import java.util.Iterator;

public interface SortedSet<T extends Comparable>
    extends Iterable<T> {
    public boolean isEmpty();
    public T insert(T item);
    public T delete(T item);
    public T member(T item);
    public Iterator<T> iterator();
}

Here we have modified our SortedSet interface to extend the Iterable interface, i.e., it includes an iterator method that returns and Iterator object.
public void run() {  
    SortedSet<String> set = new BSTree<String>();  
    // Omitted ...
}

public void printSet(SortedSet<String> set) {  
    for (String item : set) {  
        print(item + " ");  
    }
    println();
}

Now it’s easier than ever to print out the data in a BSTree in its proper ordering!
Of course, if you like to know what’s going on under the hood, you can do it this way.