Data Structures

Augustin Cosse.

Spring 2021

April 8, 2021

- Recall many data structures in java (such as ArrayList, Trees,...) are iterable but are not themselves iterators.
- For a structure to be iterable (i.e. to implement the iterable interface), it must provide an implementation for the iterator() function which should return an iterator on the elements of the collection
- An iterator comes up with two kind of functions : hasNext() and next().
- Also recall that when dealing with the Linked Positional List class, we found it convenient to have two types of iterators: an iterator on the elements of the list and an iterator on the positions

- In particular, we decided to define the iterators such that they could be used with the syntax

```
for (ElementType e : collection)
```

```
for (Position<ElementType> p : collection.positions)
```

- To define the Element Iterator, we used the following nested class to construct the iterator

```
private class ElementIterator implements Iterator<E> {
  Iterator<Position<E>> posIterator = new PositionIterator( );
  public boolean hasNext( ) { return posIterator.hasNext( ); }
  public E next( ) { return posIterator.next( ).getElement( ); }
  public void remove( ) { posIterator.remove( ); }
}
```

- And we returned the iterator through the **iterator()** method, as requested by the Iterable interface

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

- To define the iterator on positions, in order to make the distinction with the **iterator** on the elements of the collection, we fisrt implemented a method **positions()** and we had it return an instance of a nested class that implemented the **iterable** interface (hence providing an implementation of the **iterator** method that returned the iterator on positions).

- We then defined the nested **PositionIterator** class, formally defining the iterator on position (hence implementing the Iterator interface)

- Note that we don't have to implement a Position Iterator but we have to provide at least one **iterator()** method if our class implement the **iterable()** interface

- For trees as well, since we defined the tree interface to extend the Iterable interface, and since our AbstractTree class (from which the class BinaryTree inherits) implements this interface, we must provide at least an implementation of the **iterator()** method.

- On top of this, since our Tree interface provide the signature of a **position()** method, the binaryTree class which is the first concrete class in our inheritance scheme, must provide an implementation for this method as well.

```
public interface Tree<E> extends Iterable<E> {
  Position<E> root( );
  Position<E> parent(Position<E> p) throws IllegalArgumentException;
  Iterable<Position<E>> children(Position<E> p)
                          throws IllegalArgumentException;
  int numChildren(Position<E> p) throws IllegalArgumentException;
  boolean isInternal(Position<E> p) throws IllegalArgumentException;
  boolean isExternal(Position<E> p) throws IllegalArgumentException;
  boolean isRoot(Position<E> p) throws IllegalArgumentException;
  int size( );
  boolean isEmpty( );
  Iterator<E> iterator( );
  Iterable<Position<E>> positions( );}
```

- In order to provide implementations for those two iterator() and positions() functions, we will rely on the traversal algorithms

- The iterator on the elements of the tree can be easily produced from the iterator on positions. Indeed if the **positions()** method returns the iterator on positions, we can define the iterator on the elements by defining a nested class as for Positional LinkedLists

```
private class ElementIterator implements Iterator<E> {
  Iterator<Position<E>> posIterator = positions( ).iterator( );
  public boolean hasNext( ) { return posIterator.hasNext( ); }
  public E next( ) { return posIterator.next( ).getElement( ); }
  public void remove( ) { posIterator.remove( ); }
}
/** Returns an iterator of the elements stored in the tree. */
public Iterator<E> iterator( ) { return new ElementIterator( ); }
```

- We can thus focus on implementing the **position()** method whch will rely on any of the traversal approaches that we covered (below we choose to follow preorder traversal)

```
public Iterable<Position<E>> positions( )
{ return preorder( ); }
```

- Recall that when defining an iterator we have the choice between lazy iterators (does not make a copy of the sequence) and snapshot iterators (maintains its own copy of the sequence).

- For trees, we will use a snapshot iterator. We start by defining a private method **preorderSubtree** which takes a list as a parameter and add all the positions from the tree following a preorder traversal, starting from the root p

```
private void preorderSubtree(Position<E> p,
                             List<Position<E>> snapshot) {
  snapshot.add(p);
 // for preorder, we add position p
// before exploring subtrees
  for (Position<E> c : children(p))
  preorderSubtree(c, snapshot);
}
```

- We then use this private method with an empty list and with the root to define the preorder

```
public Iterable<Position<E>> preorder( ) {
  List<Position<E>> snapshot = new ArrayList<>( );
  if (!isEmpty( ))
  preorderSubtree(root( ), snapshot);
// fill the snapshot recursively
  return snapshot;
}
```

- ArrayList extends AbstractList which implements Iterable and our list of Position therefore comes with the appropriate **hasNext()**, **next()** and **remove()** methods.
- We can thus return the ArrayList as the result of our **positions()** method

```
public Iterable<Position<E>> positions( )
{ return preorder( ); }
```

- The exact same idea holds for postorder traversal. The only difference is that a visited position is not added to the postorder snapshot until after all its subtrees have been traversed

```
private void postorderSubtree(Position<E> p,
          List<Position<E>> snapshot) {
  for (Position<E> c : children(p))
  postorderSubtree(c, snapshot);
  snapshot.add(p); }
/* for postorder, we add position p
   after exploring subtrees */
```

```
public Iterable<Position<E>> postorder( ) {
  List<Position<E>> snapshot = new ArrayList<>( );
  if (!isEmpty( ))
  postorderSubtree(root( ), snapshot);
  return snapshot;}
```

- Similarly we can implement a Breadth First traversal

```
public Iterable<Position<E>> breadthfirst( ) {
  List<Position<E>> snapshot = new ArrayList<>( );
  if (!isEmpty( )) {
    Queue<Position<E>> fringe = new LinkedQueue<>( );
    fringe.enqueue(root( )); // start with the root
    while (!fringe.isEmpty( )) {
      Position<E> p = fringe.dequeue( );
// remove from front of the queue
      snapshot.add(p); // report this position
      for (Position<E> c : children(p))
        fringe.enqueue(c);
// add children to back of queue
}}
return snapshot;
}
```

- Recall that in BF traversal, we rely on a queue of positions. In order to do this, we will use the LinkedQueue class which we have introduced in previous courses

```
public Iterable<Position<E>> breadthfirst( ) {
  List<Position<E>> snapshot = new ArrayList<>( );
  if (!isEmpty( )) {
    Queue<Position<E>> fringe = new LinkedQueue<>( );
    fringe.enqueue(root( )); // start with the root
    while (!fringe.isEmpty( )) {
      Position<E> p = fringe.dequeue( );
// remove from front of the queue
      snapshot.add(p); // report this position
      for (Position<E> c : children(p))
        fringe.enqueue(c);
// add children to back of queue
}}
return snapshot;
}
```

- The preorder, postorder and Breadth First Traversal approaches can be used with ervy type of tree. For binary tree, we can rely on the more specific inorder traversal (which is based on the notions of left and right children)
- In particular, we can include its definition in the Abstract Binary class
- We can use a recursive traveral as we did for the pre and postorder iterators
- When dealing with binary trees, the inorder traversal is the most natural iterator , we thus override the **position()** method with a **position()** method based on this last approach.

```
private void inorderSubtree(Position<E> p,
            List<Position<E>> snapshot) {
  if (left(p) != null)
    inorderSubtree(left(p), snapshot);
  snapshot.add(p);
  if (right(p) != null)
    inorderSubtree(right(p), snapshot);}
/** Returns an iterable collection of
positions of the tree, reported in inorder. */
public Iterable<Position<E>> inorder( ) {
  List<Position<E>> snapshot = new ArrayList<>( );
  if (!isEmpty( ))
    inorderSubtree(root( ), snapshot);
// fill the snapshot recursively
  return snapshot;}
/** Overrides positions to make inorder
the default order for binary trees. */
public Iterable<Position<E>> positions( ) {
  return inorder( );}
```