Fundamentals of Data Structure

Mahesh Shirole

VJTI, Mumbai-19

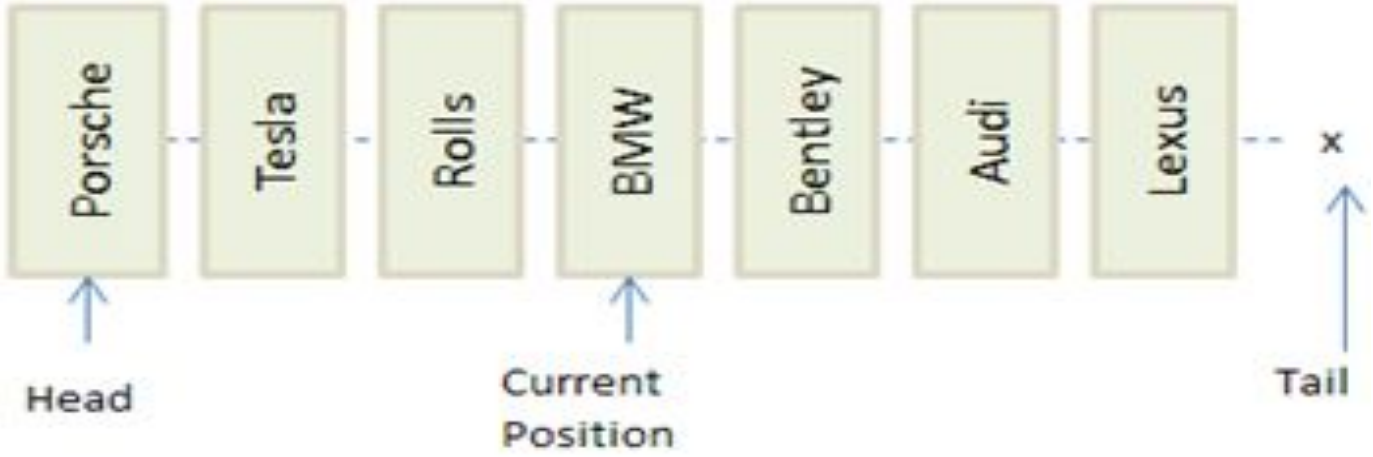
Slides are prepared from

1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

2.Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

Positional Lists

- **Array-based sequences** provide *integer indices* provide an excellent means for describing the location of an element, or the location at which an insertion or deletion should take place
- In a **linked list** knowing only an element's index is not sufficeint to access that element, the only way to reach it is to *traverse the list incrementally from its beginning or end*
- Indices are not a good abstraction for describing a more local view of a position in a sequence, because the index of an entry changes over time due to insertions or deletions that happen earlier in the sequence



Positional Lists

- Our goal is to design an abstract data type that provides a user a way to ***refer to elements anywhere in a sequence***, and to perform arbitrary insertions and deletions
  - For example, a **text document** can be viewed as a long sequence of characters. A word processor uses the abstraction of a ***cursor to describe a position*** within the document without explicit use of an integer index, allowing operations such as “delete the character at the cursor” or “insert a new character just after the cursor.”
- Therefore, in defining the positional list ADT, we also introduce the concept of a **position**, which formalizes the intuitive notion of the “location” of an element relative to others in the list

Positions

- To provide a general abstraction for the location of an element within a structure, we define a simple *position abstract data type*
- A position supports the following single method:

  getElement(): Returns the element stored at this position
- A position acts as a *marker or token* within a broader positional list
- A position *p*, which is associated with some element *e* in a list **L**, does not change, even if the index of *e* changes in **L** due to insertions or deletions elsewhere in the list
- Nor does position *p* change if we replace the element *e* stored at *p* with another element **e'**
- The only way in which a *position becomes invalid* is if that *position (and its element) are explicitly removed* from the list

The Positional List Abstract Data Type

- A positional list as a *collection of positions*, each of which stores an element
- The accessor methods provided by the positional list ADT include the following, for a list L:
  - **first()**: Returns the position of the first element of L (or null if empty)
  - **last()**: Returns the position of the last element of L (or null if empty)
  - **before(p)**: Returns the position of L immediately before position p (or null if p is the first position)
  - **after(p)**: Returns the position of L immediately after position p (or null if p is the last position)
  - **isEmpty()**: Returns true if list L does not contain any elements
  - **size()**: Returns the number of elements in list L
- An error occurs if a position ***p***, sent as a parameter to a method, is not a valid position for the list
- Note well that the *first() and last() methods* of the positional list ADT ***return the associated positions, not the elements***

Positional List

- As a demonstration of a typical traversal of a positional list named **guests**, that stores string elements, and prints each element while traversing from the beginning of the list to the end

```
Position<String> cursor = guests.first( );
while (cursor != null) {
  System.out.println(cursor.getElement( ));
  cursor = guests.after(cursor);                    // advance to the next position (if any)
}
```

- This code relies on the convention that the *null reference* is returned when the ***after*** method is called upon the last position
- The positional list ADT similarly indicates that the *null value* is returned when the ***before*** method is invoked at the front of the list

Updated Methods of a Positional List

- The positional list ADT also includes the following update methods:
  - **addFirst(e)**: Inserts a new element ***e*** at the front of the list, returning the position of the new element
  - **addLast(e)**: Inserts a new element ***e*** at the back of the list, returning the position of the new element
  - **addBefore(p, e)**: Inserts a new element ***e*** in the list, just before position ***p***, returning the position of the new element
  - **addAfter(p, e)**: Inserts a new element ***e*** in the list, just after position ***p***, returning the position of the new element
  - **set(p, e)**: Replaces the element at position ***p*** with element ***e***, returning the element formerly at position ***p***
  - **remove(p)**: Removes and returns the element at position ***p*** in the list, invalidating the position

Java Position Interface Definition

- A Java Position interface, representing the position ADT

```
public interface Position<E> {
  /**
   * Returns the element stored at this position.
   *
   * @return the stored element
   * @throws IllegalStateException if position no longer valid
   */
  E getElement( ) throws IllegalStateException;
}
```

Java PositionList Interface Definition -1

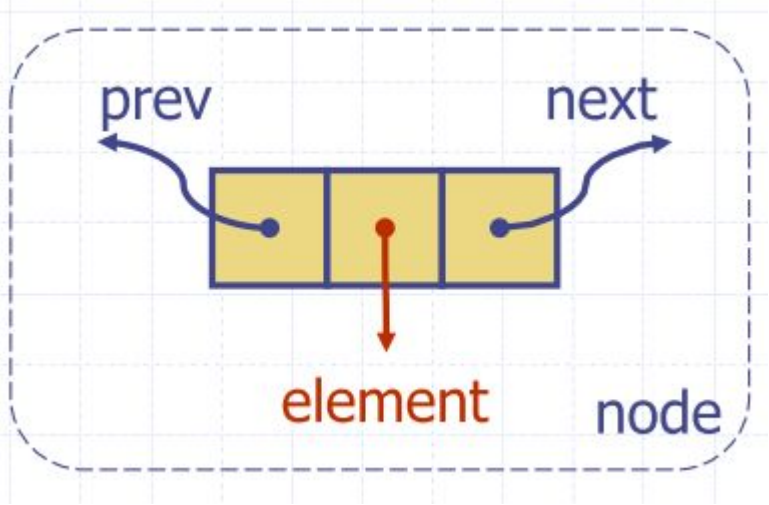
```
/** An interface for positional lists. */
public interface PositionalList<E> {

  /** Returns the number of elements in the list. */
  int size( );

  /** Tests whether the list is empty. */
  boolean isEmpty( );

  /** Returns the first Position in the list (or null, if empty). */
  Position<E> first( );

  /** Returns the last Position in the list (or null, if empty). */
  Position<E> last( );

  /** Returns the Position immediately before Position p (or null, if p is first). */
  Position<E> before(Position<E> p) throws IllegalArgumentException;

  /** Returns the Position immediately after Position p (or null, if p is last). */
  Position<E> after(Position<E> p) throws IllegalArgumentException;
```

Java PositionList Interface Definition -2

```
  /** Inserts element e at the front of the list and returns its new Position. */
  Position<E> addFirst(E e);

  /** Inserts element e at the back of the list and returns its new Position. */
  Position<E> addLast(E e);

  /** Inserts element e immediately before Position p and returns its new Position. */
  Position<E> addBefore(Position<E> p, E e)
    throws IllegalArgumentException;

  /** Inserts element e immediately after Position p and returns its new Position. */
  Position<E> addAfter(Position<E> p, E e)
    throws IllegalArgumentException;

  /** Replaces the element stored at Position p and returns the replaced element. */
  E set(Position<E> p, E e) throws IllegalArgumentException;

  /** Removes the element stored at Position p and returns it (invalidating p). */
  E remove(Position<E> p) throws IllegalArgumentException;
}
```

PositionList- Doubly Linked List Implementation

- The nested Node<E> class, which implements the Position<E> interface




```
//--------------- nested Node class ---------------
private static class Node<E> implements Position<E> {
  private E element;                    // reference to the element stored at this node
  private Node<E> prev;                 // reference to the previous node in the list
  private Node<E> next;                 // reference to the subsequent node in the list
  public Node(E e, Node<E> p, Node<E> n) {
    element = e;
    prev = p;
    next = n;
  }
  public E getElement( ) throws IllegalStateException {
    if (next == null)                   // convention for defunct node
      throw new IllegalStateException("Position no longer valid");
    return element;
  }
  public Node<E> getPrev( ) {
    return prev;
  }
  public Node<E> getNext( ) {
    return next;
  }
  public void setElement(E e) {
    element = e;
  }
  public void setPrev(Node<E> p) {
    prev = p;
  }
  public void setNext(Node<E> n) {
    next = n;
  }
} //---------- end of nested Node class ----------
```

PositionList- Doubly Linked List Implementation

```
/** Implementation of a positional list stored as a doubly linked list. */
public class LinkedPositionalList<E> implements PositionalList<E> {
  //---------- nested Node class ----------

    // instance variables of the LinkedPositionalList
    private Node<E> header;                    // header sentinel
    private Node<E> trailer;                   // trailer sentinel
    private int size = 0;                      // number of elements in the list

    /** Constructs a new empty list. */
    public LinkedPositionalList( ) {
      header = new Node<>(null, null, null);   // create header
      trailer = new Node<>(null, header, null); // trailer is preceded by header
      header.setNext(trailer);                 // header is followed by trailer
    }
```

PositionList- Doubly Linked List Implementation

```
// private utilities
/** Validates the position and returns it as a node. */
private Node<E> validate(Position<E> p) throws IllegalArgumentException {
  if (!(p instanceof Node)) throw new IllegalArgumentException("Invalid p");
  Node<E> node = (Node<E>) p;     // safe cast
  if (node.getNext( ) == null)       // convention for defunct node
    throw new IllegalArgumentException("p is no longer in the list");
  return node;
}

/** Returns the given node as a Position (or null, if it is a sentinel). */
private Position<E> position(Node<E> node) {
  if (node == header || node == trailer)
    return null;      // do not expose user to the sentinels
  return node;
}

// public accessor methods
/** Returns the number of elements in the linked list. */
public int size( ) { return size; }

/** Tests whether the linked list is empty. */
public boolean isEmpty( ) { return size == 0; }
```

PositionList- Doubly Linked List Implementation

```
/** Returns the first Position in the linked list (or null, if empty). */
public Position<E> first( ) {
  return position(header.getNext( ));
}

/** Returns the last Position in the linked list (or null, if empty). */
public Position<E> last( ) {
  return position(trailer.getPrev( ));
}

/** Returns the Position immediately before Position p (or null, if p is first). */
public Position<E> before(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  return position(node.getPrev( ));
}

/** Returns the Position immediately after Position p (or null, if p is last). */
public Position<E> after(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  return position(node.getNext( ));
}
```

PositionList-Doubly Linked List Implementation

```
// private utilities
/** Adds element e to the linked list between the given nodes. */
private Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
  Node<E> newest = new Node<>(e, pred, succ);  // create and link a new node
  pred.setNext(newest);
  succ.setPrev(newest);
  size++;
  return newest;
}

// public update methods
/** Inserts element e at the front of the linked list and returns its new Position. */
public Position<E> addFirst(E e) {
  return addBetween(e, header, header.getNext( ));          // just after the header
}

/** Inserts element e at the back of the linked list and returns its new Position. */
public Position<E> addLast(E e) {
  return addBetween(e, trailer.getPrev( ), trailer);          // just before the trailer
}

/** Inserts element e immediately before Position p, and returns its new Position.*/
public Position<E> addBefore(Position<E> p, E e)
                                    throws IllegalArgumentException {
  Node<E> node = validate(p);
  return addBetween(e, node.getPrev( ), node);
}
```

PositionList-Doubly Linked List Implementation

```
/** Inserts element e immediately after Position p, and returns its new Position. */
public Position<E> addAfter(Position<E> p, E e)
                                    throws IllegalArgumentException {
  Node<E> node = validate(p);
  return addBetween(e, node, node.getNext( ));
}

/** Replaces the element stored at Position p and returns the replaced element. */
public E set(Position<E> p, E e) throws IllegalArgumentException {
  Node<E> node = validate(p);
  E answer = node.getElement( );
  node.setElement(e);
  return answer;
}
/** Removes the element stored at Position p and returns it (invalidating p). */
public E remove(Position<E> p) throws IllegalArgumentException {
  Node<E> node = validate(p);
  Node<E> predecessor = node.getPrev( );
  Node<E> successor = node.getNext( );
  predecessor.setNext(successor);
  successor.setPrev(predecessor);
  size--;
  E answer = node.getElement( );
  node.setElement(null);                    // help with garbage collection
  node.setNext(null);                       // and convention for defunct node
  node.setPrev(null);
  return answer;
 }
}
```

The Performance of a Linked Positional List

Method	Running Time
size( )	$O(1)$
isEmpty( )	$O(1)$
first( ), last( )	$O(1)$
before(p), after(p)	$O(1)$
addFirst(e), addLast(e)	$O(1)$
addBefore(p, e), addAfter(p, e)	$O(1)$
set(p, e)	$O(1)$
remove(p)	$O(1)$

Iterators

- An **iterator** is a software design pattern that abstracts the *process of scanning through a sequence of elements, one element at a time*
- The underlying elements might be stored in a container class, streaming through a network, or generated by a series of computations
- Java defines the java.util.Iterator interface with the following three methods:
  - **hasNext()**: Returns true if there is at least one additional element in the sequence, and false otherwise
  - **next()**: Returns the next element in the sequence
  - **remove():** Removes from the collection the element returned by the most recent call to next(). Throws an IllegalStateException if next has not yet been called, or if remove was already called since the most recent call to next

Iterators - Usage

- The combination of these two methods allows a general loop construct for processing elements of the iterator
- For example, if we let variable, *iter*, denote an instance of the *Iterator<String>* type, then we can write the following:

```
while (iter.hasNext( )) {
  String value = iter.next( );
  System.out.println(value);
}
```

Iterable Interface

- A single iterator instance supports only one pass through a collection; calls to next can be made until all elements have been reported, but there is no way to "**reset**" the iterator back to the beginning of the sequence
- However, a data structure that wishes to allow *repeated iterations can support a method that returns a new iterator*, each time it is called
- To provide greater standardization, Java defines another ***parameterized interface, named Iterable***, that includes the following single method:
  - **iterator()**: Returns an iterator of the elements in the collection
- Each call to iterator() returns a new iterator instance, thereby allowing multiple (even simultaneous) traversals of a collection

```
Iterator<ElementType> iter = collection.iterator( );
while (iter.hasNext( )) {
  ElementType variable = iter.next( );
  loopBody                                    // may refer to "variable"
}
```

Iterable Interface - Example

- Consider the following loop can be used to remove all negative numbers from an ArrayList of floating-point values

```
ArrayList<Double> data;  // populate with random numbers (not shown)
Iterator<Double> walk = data.iterator( );
while (walk.hasNext( ))
  if (walk.next( ) < 0.0)
    walk.remove( );
```

Iterations with the LinkedPositionalList class

```
//---------- nested PositionIterator class ----------
private class PositionIterator implements Iterator<Position<E>> {
  private Position<E> cursor = first( );     // position of the next element to report
  private Position<E> recent = null;         // position of last reported element
  /** Tests whether the iterator has a next object. */
  public boolean hasNext( ) { return (cursor != null);    }
  /** Returns the next position in the iterator. */
  public Position<E> next( ) throws NoSuchElementException {
    if (cursor == null) throw new NoSuchElementException("nothing left");
    recent = cursor;                 // element at this position might later be removed
    cursor = after(cursor);
    return recent;
  }
  /** Removes the element returned by most recent call to next. */
  public void remove( ) throws IllegalStateException {
    if (recent == null) throw new IllegalStateException("nothing to remove");
    LinkedPositionalList.this.remove(recent);          // remove from outer list
    recent = null;                   // do not allow remove again until next is called
  }
} //---------- end of nested PositionIterator class ----------
```

Iterations with the LinkedPositionalList class

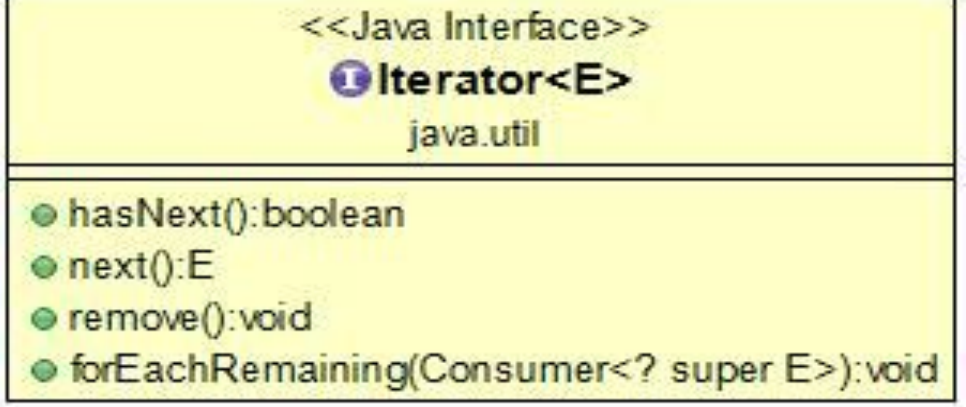
```
//---------- nested PositionIterable class ----------
private class PositionIterable implements Iterable<Position<E>> {
  public Iterator<Position<E>> iterator( ) { return new PositionIterator( ); }
} //---------- end of nested PositionIterable class ----------

/** Returns an iterable representation of the list's positions. */
public Iterable<Position<E>> positions( ) {
  return new PositionIterable( );          // create a new instance of the inner class
}

//---------- nested ElementIterator class ----------
/* This class adapts the iteration produced by positions() to return elements. */
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( ); } // return element!
  public void remove( ) { posIterator.remove( ); }
}
```

Lab Assignment-04

```
public interface Position<E> {
  /**
   * Returns the element stored at this position.
   *
   * @return the stored element
   * @throws IllegalStateException if position no longer valid
   */
  E getElement( ) throws IllegalStateException;
}
```

- Implement the positional list using linked list and implement iterator for the same. Use position interface and iterator interface.
- The accessor methods provided by the positional list ADT include the following, for a list L:
  - **first()**: Returns the position of the first element of L (or null if empty). last(): Returns the position of the last element of L (or null if empty)
  - **before(p)**: Returns the position of L immediately before position p (or null if p is the first position)
  - **after(p)**: Returns the position of L immediately after position p (or null if p is the last position)
  - **isEmpty()**: Returns true if list L does not contain any elements. size(): Returns the number of elements in list L

Lab Assignment-04



- The positional list ADT also includes the following update methods:
  - **addFirst(e)**: Inserts a new element ***e*** at the front of the list, returning the position of the new element
  - **addLast(e)**: Inserts a new element ***e*** at the back of the list, returning the position of the new element
  - **addBefore(p, e)**: Inserts a new element ***e*** in the list, just before position ***p***, returning the position of the new element
  - **addAfter(p, e)**: Inserts a new element ***e*** in the list, just after position ***p***, returning the position of the new element
  - **set(p, e)**: Replaces the element at position ***p*** with element ***e***, returning the element formerly at position ***p***
  - **remove(p)**: Removes and returns the element at position ***p*** in the list, invalidating the position