Data Structures

Augustin Cosse.

Spring 2021

March 23, 2021

Lists and Iterator ADTs

- So far we have discussed linearly ordered sequences of elements.
- Among the types we studied, double ended queues are the most general, yet they only allow insertions and deletions at beginning and from the back of the sequence.
- We will now explore additional abstract data types that represent a linear sequence of elements, but with more general support for adding and removing elements at arbitrary positions
- As we saw, location within an array are described with an integer index. Recall that an index of an element e in a sequence is the number of elements before e in the sequence. I.e. the first element has index 0 and the last has index $n-1$

Lists and Iterator ADTs

- The notion of index is well defined for linked lists as well although not very convenient as in linked lists there is no efficient way to access an element without traversing a portion of the linked list that depends upon the index.
- Java defines a general interface **java.util.List** that includes the following methods (Part I)

size()	Returns the number of elements in the list
isEmpty()	Returns a boolean indicating whether the list is empty
get(i)	Returns the element of the list having index i; an error occurs if i is not in the range $[0,$ size()-1$]$
set(i,e)	Replaces the element at index i with e and returns the old element that was replaced; an error condition occurs if i is in the range [0, size()]

Lists and Iterator ADTs

add(i,e)	inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list. An error condition occurs if i is not in the range [0, size()]
remove(i)	Removes and returns the element at index i, moving all subsequent elements one index earlier in the list an error occurs if i is not in the range [0, size()-1]

- Note that the index of an existing element may change over time, as other elements are added or removed in from of it

Lists and Iterator ADTs

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	"error"	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	"error"	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	"error"	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

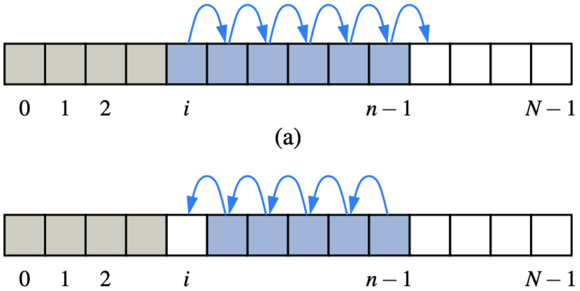
Lists and Iterator ADTs

```
/** simplified java.util.List interface. */
public interface List<E> {
/** num of elements in list */
int size( );
boolean isEmpty( );
/** Returns (but does not remove) element i. */
E get(int i) throws IndexOutOfBoundsException;
/** Replaces element i with e,
      and returns the replaced element. */
E set(int i, E e) throws IndexOutOfBoundsException;
/** Inserts e at index i, shifting subsequent elems */
void add(int i, E e) throws IndexOutOfBoundsException;

/** Removes/returns element i,
      shifting subsequent elems. */
E remove(int i) throws IndexOutOfBoundsException;
}
```

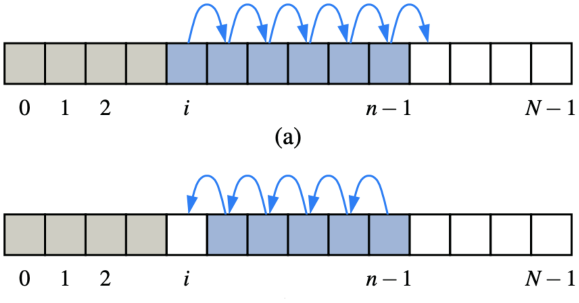
Lists and Iterator ADTs

- The simplest choice to implement the list ADT would be to rely on arrays for which $A[i]$ stores the element with index i
- We will begin by considering fixed capacity arrays and we will then study how to extend this idea to unbounded capacity. Such a list is known as an array list in Java
- With a representation based on arrays, the **get(i)** and **set(i,e)** methods are easy to implement by accessing elements as $A[i]$.



Lists and Iterator ADTs

- The methods add(i,e) and remove(i) are more time consuming as they require shifting elements up or down to maintain our rule of always maintaining an element whose list index is i at index i in the array



Lists and Iterator ADTs

- A fixed capacity implementation can be found below

```java
public class ArrayList<E> implements List<E> {
// instance variables
public static final int CAPACITY=16; // fixed capacity
private E[ ] data; // generic array for storage
private int size = 0;
// constructors
public ArrayList( ) { this(CAPACITY); } // constructor1
public ArrayList(int capacity) { // constructor2
data = (E[ ]) new Object[capacity];
// safe cast; compiler may give warning
}

// ... Part 2 and 3 on next slides
```

Lists and Iterator ADTs

```
public class ArrayList<E> implements List<E> {
// Part II
public int size( ) { return size; }
public boolean isEmpty( ) { return size == 0; }
/** Returns (but does not remove) the
     element at index i. */
public E get(int i) throws IndexOutOfBoundsException {
  checkIndex(i, size);
  return data[i];}
/** Replaces element at i with e, and
      returns the replaced elem. */
public E set(int i, E e) throws
              IndexOutOfBoundsException {
  checkIndex(i, size);
  E temp = data[i];
  data[i] = e;
  return temp;}
```

Lists and Iterator ADTs

```
public class ArrayList<E> implements List<E> {
// Part III
  public void add(int i, E e) throws
                 IndexOutOfBoundsException,
  IllegalStateException {
  checkIndex(i, size + 1);
  if (size == data.length) // not enough capacity
    throw new IllegalStateException("Array is full");
  for (int k=size1; k >= i; k--) // shifting rightmost
    data[k+1] = data[k];
  data[i] = e; // ready to place the new element
  size++;
}
```

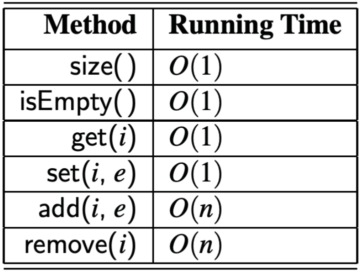
Lists and Iterator ADTs

```java
/** Removes/returns element at index i, with shift. */
public E remove(int i) throws
              IndexOutOfBoundsException {
checkIndex(i, size);
E temp = data[i];
for (int k=i; k < size1; k++) // shift to fill hole
data[k] = data[k+1];
data[size-1] = null; // help garbage collection
size--;
return temp;}
// utility method
/** Checks whether index is in range [0, n1]. */
protected void checkIndex(int i, int n) throws
             IndexOutOfBoundsException {
if (i < 0 || i >= n)
throw new IndexOutOfBoundsException
                ("Illegal index: " + i);}}
```

Lists and Iterator ADTs

- The insertion and removal methods can take much longer than $O(1)$. Indeed the worst cases for those operations occur when $i = 0$ since all the existing elements have to be shifted forward

- Assuming that each possible index is equally likely to be passed as an argument to these operations, the average running time is $O(n)$ since we have to shift $n/2$ elements on average



Method	**Running Time**
size( )	$O(1)$
isEmpty( )	$O(1)$
get(i)	$O(1)$
set(i, e)	$O(1)$
add(i, e)	$O(n)$
remove(i)	$O(n)$

Lists and Iterator ADTs

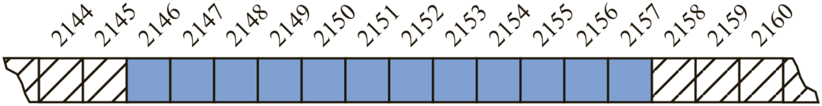
- With a little effort, one can produce an array based implementation of the array list ADT that achieves $O(1)$ time for insertions and removals at index 0 as well as insertions and removals at the end of the list
- Such an improvement in the efficienty, which can be obtained through circular arrays) requires that we give up on the rule that an element at index i is stored in the array at index i

Dynamic Arrays

- On top of the worst case running times, the ArrayList implementation has another serious limitation : it requires a fixed maximum capacity to be declared.
- This is a major limitation because if a user is unsure of the maximum size that will be reached for a collection, there is a risk that either too large of an array will be requested (causing inefficient waste of memory) or that too small an array will be requested, causing a fatal error when exhausting that capacity
- Fortunately, Java's ArrayList class provides a more robust abstraction, adding a user to add elements to the list with no apparent limit on the capacity
- To provide this abstraction, Java relies on an algorithmic sleight of hand that is known as a dynamic array

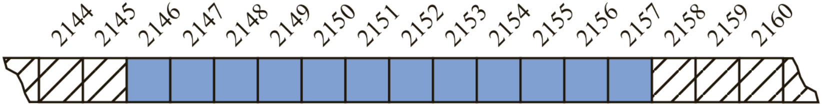
Dynamic Arrays

- In reality, elements of an array list are stored in a traditional array and the precise of such a traditional array must be declared
- Because the system may alocate neighboring memory locations to store other data, the capacity of an array cannot be increased by expanding into subsequent cells
- The first key to providing the semantics of an unbounded array is that an array list instance maintains an internal array that often has greater capacity than the current length of the list. I.e while a user may have created a list with five elements, the system may have reserved an underlying array capable of storing eight object references.



Dynamic Arrays

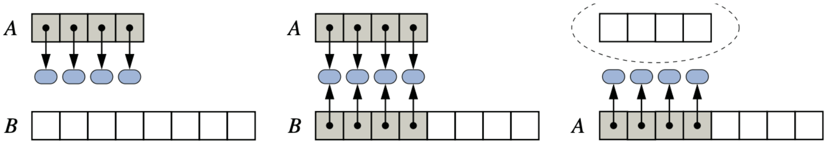
- The extra capcity makes it easy to add a new element to the end of the list by using the next available cells in the array
- If a user continues to add elements to a list, all reserved capacity in the underlying array will eventually be exhausted
- In that case, the class requests a new larger array from the system and copies all references from the smaller array into the beginning of the new array



Implementing a dynamic array

- To implement dynamic arrays, we rely on the same traditional array A, that is initialized wither as a default capacity or to one specified as a parameter to the constructor
- The key is to provide means to "grow" the array A, when more space is needed. Of course, we cannot actually grow that array, as its capacity is fixed. Instead, when a call to add a new element risks overflowing the current array, we perform the following additional steps:
  1. Allocate a new array B with larger capacity
  2. Set $B[k] = A[k]$, for $k = 0, \ldots, n-1$ where n denotes the current number of items
  3. Set $A = B$, that is we use the new array to support the list
  4. Insert the new element in the new array

Implementing a dynamic array



```
/** Resizes internal array to have given capacity
     >= size. */
protected void resize(int capacity) {
  E[ ] temp = (E[ ]) new Object[capacity];
 // safe cast; compiler may give warning
  for (int k=0; k < size; k++)
    temp[k] = data[k];
  data = temp; // start using the new array
}
```

Implementing a dynamic array

- Given the resize function, the remaining issue to consider is how large of a new array to create. A commonly used rule is for the new array to have twice the capacity of the existing array that has been filled (we will see later why this might be a good idea).

- From this, we can redesign the add method so that it calls the the new resize utility function when detecting that the new array is filled

```
/** Inserts element e to be at index i,
      then apply shifting*/
public void add(int i, E e) throws
           IndexOutOfBoundsException {
  checkIndex(i, size + 1);
  if (size == data.length) // not enough capacity
    resize(2  data.length); // double capacity
... // rest of method unchanged...
```

Implementing a dynamic array

- Given the resize function, the remaining issue to consider is how large of a new array to create. A commonly used rule is for the new array to have twice the capacity of the existing array that has been filled (we will see later why this might be a good idea).

- From this, we can redesign the add method so that it calls the the new resize utility function when detecting that the new array is filled

```
/** Inserts element e to be at index i,
      then apply shifting*/
public void add(int i, E e) throws
           IndexOutOfBoundsException {
  checkIndex(i, size + 1);
  if (size == data.length) // not enough capacity
    resize(2  data.length); // double capacity
... // rest of method unchanged...
```

Positional lists

- integer indices provide an excellent means for describing the location of an element , or the location at which an element should be deleted or inserted
- Numeric indices are however not a good choice for describing positions in a linked list as knowing only one element in a list will require to travers the list incrementally from its beginning (or its end) to reach it, counting elements along the way
- Moreover, indices do not give a good "local view" of a position in a sequence as the index change over time due to insertion and deletion that happened earlier in the sequence.
- We would prefer to have an abstract data type that provides a way to refer to elements anywhere in the sequence and to perform arbitrary insertions and deletions

Positional lists

- As an example, a text document can be viewed as a long sequence of characters. A word processor uses the abstraction of a cursor to describe the position within the document without explicit use of an integer index, allowing operations such as "deleting the character" or "insert the character just after the cursor" to be carried out efficiently
- We might also be able to refer to an inherent position within a document, such as a chapter without relying on a character index.
- For these reasons, we will temporarily leave aside the index based methods of the list type and instead design our own abstract data type that we will call a positional list

Positional lists

- Our objective with positional lists it that they implement $O(1)$ insertions and deletions at arbitrary positions in the array

- To achieve O(1) (i.e. constant time) insertions and deletions at arbitrary locations, we need a reference to the node at which the element is stored. It is tempting to develop and ADT in which the node reference serves As the mechanism for describing a position (in fact our previous implementation of Doubly linked lists contains methods addBetween and remove which accept node reference as parameters)

```
private void addBetween(E e, Node<E> pred, Node<E> succ){
 // create and link a new node
  node<E> newest = new Node<>(e pred, succ);
  pred.setNext(newest);
  succ.setPrev(newest);
  size++;}
```

Positional lists

- List nodes are private however and the use of public nodes would violate the object oriented design principles of abstraction and encapsulation.

- There are several reasons to prefer that we encapsulate the nodes of a linked list:

  1. It is always simpler for the user of the data structureif they are not bothered by with unnecessary details of our implementation such as low level manipulation of nodes or the use of sentinel (header, trailer) nodes

  2. We can provide a more robust data structure if we do not permit the users to directly access or manipulate the nodes (we for example prevent the addBetween or remove methods to be called with a node that does not belong to the list)

  3. By encapsulating the internal details of our implementation, we have greater flexibility to redesign the data structure or improve its performance

Positional lists

- Following encapsulation, we will therefore try to maintain as many methods and variables as private and introduce the concept of a position to formalize the relative location of an element relative to the others in the list
- To provide a general abstraction for the location of an element within a structure, we define a position abstract type
- A position support the method getElement() which return the element stored at the given position.
- We will want a position to act as a marker within a broader positional list. A position p which is associated to some element e in the list should not change even if the index of e changes in L due to insertions or deletions elsewhere in the list. Nor should the position change if we replace the element e stored at p with another element.

Positional lists

- We define our positional list as a collection of positions, each of which stores an element The accessor methods provided by a positional list include the following

first()	Returns position of first element (null if empty)
last()	Returns position of last element (null if empty)
before(p)	Returns position immediately before position p
after(p)	Returns position immediately after p
isEmpty() :	Returns true if list L does not contain any elements
size()	Returns number of elems

- Note that the first() and last() methods return the associated positions not the elements

Positional lists

- As a demonstration of the traversal of a positional list, consider the code fragment

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

- The code above relies on the assumption that the null reference is returned when the after method is called upon the last position

Positional lists

- We will also want our positional ADT to include the following set of 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 postion p, returning the position of the new element
remove(p)	Removes and return the element at position p in the li invalidating this position

Positional lists

- Positional lists work through node references. Each time we add a value to the list, the function returns the corresponding node
- We cannot directly access the nodes from the list but we can cast an node from outside the class to one of private node and use it to navigate in the list

Method	Return Value	List Contents
addLast(8)	p	(8_p)
first()	p	(8_p)
addAfter(p, 5)	q	$(8_p, 5_q)$
before(q)	p	$(8_p, 5_q)$
addBefore(q, 3)	r	$(8_p, 3_r, 5_q)$
r.getElement()	3	$(8_p, 3_r, 5_q)$
after(p)	r	$(8_p, 3_r, 5_q)$
before(p)	null	$(8_p, 3_r, 5_q)$
addFirst(9)	s	$(9_s, 8_p, 3_r, 5_q)$
remove(last())	5	$(9_s, 8_p, 3_r)$
set(p, 7)	8	$(9_s, 7_p, 3_r)$
remove(q)	"error"	$(9_s, 7_p, 3_r)$

Positional lists

- We first formalize the position ADT through the following interface

```
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;}
```

Positional lists

- As for the stacks and queues, we will consider two alternative implementations of the Positional List ADT. One based on arrays, and the other based on Lists. We will derive the two from a single interface which we call PositionalList

```
/** An interface for positional lists. */
public interface PositionalList<E> { //Part I
int size( );
boolean isEmpty( );
/** Returns first/last Position in list (null, if empty). */
Position<E> first( );
Position<E> last( );
/** Returns Position before/after p (null, if p is first). */
Position<E> before(Position<E> p) throws
                        IllegalArgumentException;
Position<E> after(Position<E> p) throws
                        IllegalArgumentException;
/** Inserts e at the front and returns its Position. */
Position<E> addFirst(E e);
```

Positional lists

- As for the stacks and queues, we will consider two alternative implementations of the Positional List ADT. One based on arrays, and the other based on Lists. We will derive the two from a single interface which we call PositionalList

```
/** An interface for positional lists. */
public interface PositionalList<E> {
// Part II
/* Inserts e at back and returns new Position. /
Position<E> addLast(E e);
/** Inserts e before p and returns new Position. */
Position<E> addBefore(Position<E> p, E e)
throws IllegalArgumentException;
/** Inserts e after p and returns new Position. */
Position<E> addAfter(Position<E> p, E e)
throws IllegalArgumentException;
/** Replaces and return element at Position p */
E set(Position<E> p, E e) throws IllegalArgumentException;
/** Removes and return element at Position p */
E remove(Position<E> p) throws IllegalArgumentException;}
```

Positional lists

- The preferred implementation of positional lists is clearly the one based on doubly linked lists
- The obvious way to identify the locations within a list is through the nodes themselves
- It thus makes sense to declare the nested Node class of our linked list so as to implement the the Position interface
- The Nodes are thus our position objects. However note that the nodes are declared as private to maintain proper encapsulation. All of the public methods of the positional lists relies on the Position type.
- Although we know that we are sending and receiving nodes, these are only known to be positions from the outside. In particular, this implies that users of the class cannot call any other method than other than **getElement()**

Positional lists

- We start by implementing the nested class Node

```
public class LinkedPositionalList<E>
                          implements PositionalList<E> {
// Nested Node Part I
  private static class Node<E> implements Position<E> {
    private E element;
    private Node<E> prev;
    private Node<E> next;
    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;}
```

Positional lists

- We start by implementing the nested class Node

```
public class LinkedPositionalList<E>
                          implements PositionalList<E> {

private static class Node<E> implements Position<E> {
// Nested Node Part II
  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;}} //
```

Positional lists

- From this, we continue the implementation with the rest of the PositionalList class. In particular the constructors

```
public class LinkedPositionalList<E>
                              implements PositionalList<E> {
// Part III
  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}
```

Positional lists

- We then add two important methods: First, the validate method throws an exception if the Position does not correspond to any node in the list. Otherwise, it returns the corresponding 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) // defunct node
    throw new IllegalArgumentException
                       ("p is no longer in the list");
  return node;
}
private Position<E> position(Node<E> node) {
  if (node == header || node == trailer)
    return null; // do not expose user to the sentinels
  return node;}
```

Positional lists

- Second, the position(node) method is used each time a position has to be returned to the user. it makes sure than we do not return a sentinel node, and return a **null** reference in that case.

```java
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) // defunct node
    throw new IllegalArgumentException
                          ("p is no longer in the list");
  return node;
}
private Position<E> position(Node<E> node) {
  if (node == header || node == trailer)
    return null; // do not expose user to the sentinels
  return node;}
```

Positional lists

- One can then define a number of public accessor methods

```
public int size( ) { return size; }
public boolean isEmpty( ) { return size == 0; }
/** Returns the first Position (null if empty). */
public Position<E> first( ) {
  return position(header.getNext( ));}
/** Returns the last Position (null if empty). */
public Position<E> last( ) {
  return position(trailer.getPrev( ));}
/** Returns Position before p (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 after 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( ));}
```

Positional lists

- As well as a set of public update methods

```
private Position<E> addBetween
                    (E e, Node<E> pred, Node<E> succ) {
  Node<E> newest = new Node<>(e, pred, succ);
  pred.setNext(newest);
  succ.setPrev(newest);
  size++;
  return newest;}

// public update methods
/** Inserts e at the front/ returns new Position. */
public Position<E> addFirst(E e) {
  return addBetween(e, header, header.getNext( )); }

/** Inserts e at the back and returns new Position. */
public Position<E> addLast(E e) {
  return addBetween(e, trailer.getPrev( ), trailer); }
```

Positional lists

- As well as a set of public update methods

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

Positional lists

- Finally we conclude with the main motivation for the PositionalList class: A method for removing elements stored at arbitrary positions

```
/** Removes 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;}}
```

Positional lists

- The Positional List ADT is ideally suited for implentation with a doubly linked list as all operations run in worst case constant time

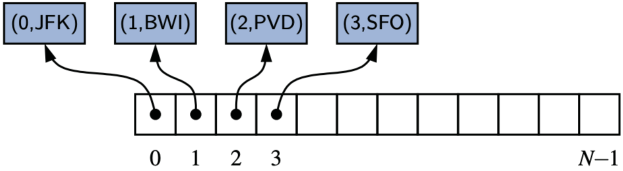
Method	**Running Time**
size()	$O(1)$
isEmpty()	$O(1)$
first(), last()	$O(1)$
before(p), after(p)	$O(1)$
addFirst(e), addLast(e)	$O(1)$
addBefore(p, e), addAfter(p, e)	$O(1)$
set(p, e)	$O(1)$
remove(p)	$O(1)$

Positional lists

- As for stacks and Queues, note that we can also implement a positional list using an array A for storage
- Some care has to be taken however when designing objects that will serve as positions
- At a first glance, it might seem that a position p only need to store the index i at which its associated element is stored in the array
- The problem with this approach is that the index of an element e changes when other insertions or deletions occur before it

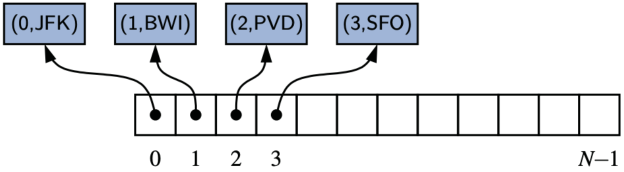
Positional lists

- If we have already returned a position p associated with element e that stores an outdated index i to a user, the wrong array cell would be accessed when the position was used (i.e. remember that positions in a positional list should always be defined relative to their neighboring positions, not indices)
- Hence, to implement a Positional List with an array, instead of storing the elements of L directly in array A, we should store a new kind of position object in each cell of A.
- A position p should store the element e as well as the current index i of that element within the list



Positional lists

- With this representation, we can determine the index currently associated with a position and we can determine the position currently associated with a specific index
- In particular, we can implement an accessor such as before(p), by finding the index of the given position and using the array to find the neighboring positions.



(0,JFK)
(1,BWI)
(2,PVD)
(3,SFO)
0
1
2
3
$N-1$