CS121 Data Structures A, C
Positional Lists

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am

Fall 2021

Lists

We explore several abstract data types that represent

a linear sequence of elements, with more general support for adding or removing elements at arbitrary positions,

unlike the stack, queue and deque ADTs.

An index of an element e in a sequence is equal to the number of elements before e in that sequence.

A **position** formalizes the intuitive notion of the 'location' of an element relative to others in the list

The Position Abstract Data Type

A position supports the following single method:

getElement(): Returns (a reference to) 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

The position p does not change if we replace the element e stored at p with another element

A position becomes invalid if that position (and its element) are explicitly removed from the list

The Positional List Abstract Data Type

A **positional list** is a collection of positions, each storing an element

The positional list ADT supports **accessor** methods:

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

The Positional List Abstract Data Type (cont'd)

A **positional list** is a collection of positions, each storing an element

The positional list ADT supports **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

For a nonempty list, `addFirst(`e`)` and `addBefore(first(), `e`)`, `addLast(`e`)` and `addAfter(last(), `e`)` are pairwise equivalent

Traversing the List

`first()` and `last()` methods return the associated *positions*, not *elements* (in contrast to the Deque ADT)

First element accessed by: `first().getElement()`

Traverse and print each element of a list, named `guests`, that stores string elements:

```
1  Position<String> cursor = guests.first( );
2  while (cursor != null) {
3    System.out.println(cursor.getElement( ));
4    cursor = guests.after(cursor);          // advance to the next position (if any)
5  }
```

Using the convention of returning `null` references, the above code fragment works correctly on an empty `guests` list

Example

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"	$(9_s, 7_p, 3_r)$

The Position API

```
1  public interface Position<E> {
2    /**
3     * Returns the element stored at this position.
4     *
5     * @return the stored element
6     * @throws IllegalStateException if position no longer valid
7     */
8    E getElement( ) throws IllegalStateException;
9  }
```

The Positional List API

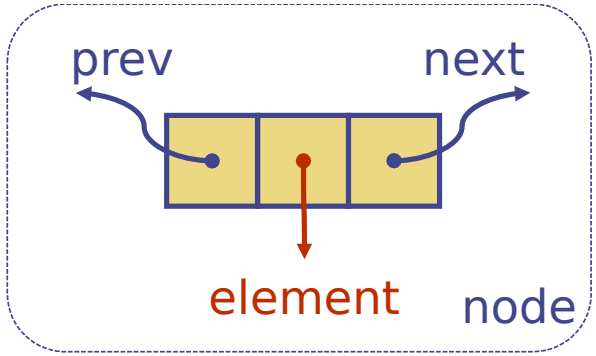
```
1   /** An interface for positional lists. */
2   public interface PositionalList<E> {
3
4     /** Returns the number of elements in the list. */
5     int size( );
6
7     /** Tests whether the list is empty. */
8     boolean isEmpty( );
9
10    /** Returns the first Position in the list (or null, if empty). */
11    Position<E> first( );
12
13    /** Returns the last Position in the list (or null, if empty). */
14    Position<E> last( );
15
16     /** Returns the Position immediately before Position p (or null, if p is first). */
17    Position<E> before(Position<E> p) throws IllegalArgumentException;
18
19    /** Returns the Position immediately after Position p (or null, if p is last). */
20    Position<E> after(Position<E> p) throws IllegalArgumentException;
21
```

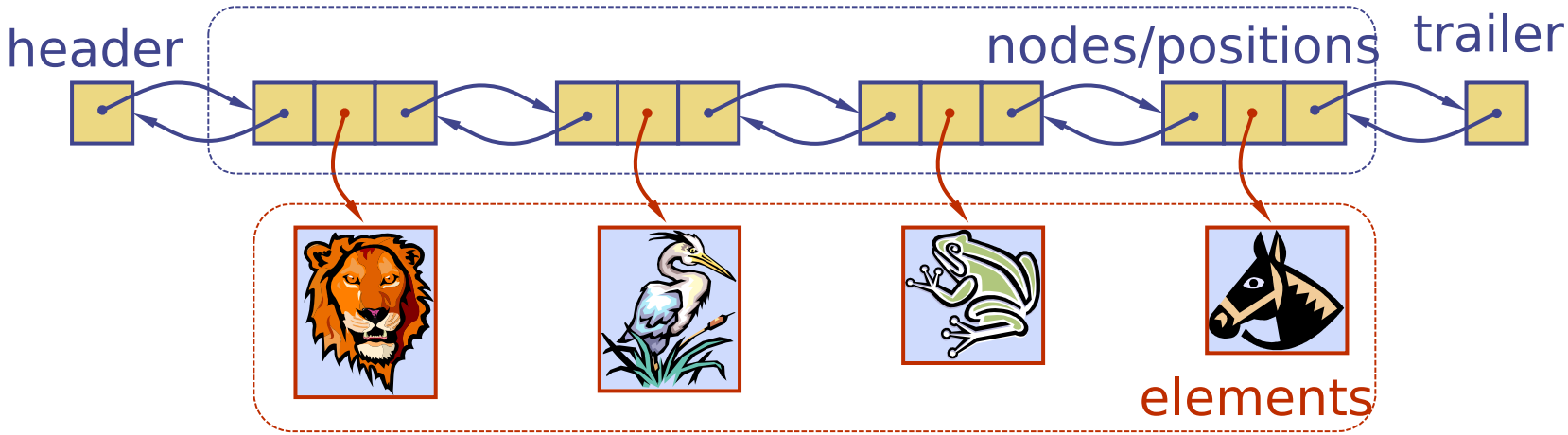
The Positional List API (cont'd)

```
21
22      /** Inserts element e at the front of the list and returns its new Position. */
23      Position<E> addFirst(E e);
24
25      /** Inserts element e at the back of the list and returns its new Position. */
26      Position<E> addLast(E e);
27
28      /** Inserts element e immediately before Position p and returns its new Position. */
29      Position<E> addBefore(Position<E> p, E e) throws IllegalArgumentException;
30
31      /** Inserts element e immediately after Position p and returns its new Position. */
32      Position<E> addAfter(Position<E> p, E e) throws IllegalArgumentException;
33
34      /** Replaces the element stored at Position p and returns the replaced element. */
35      E set(Position<E> p, E e) throws IllegalArgumentException;
36
37      /** Removes the element stored at Position p and returns it (invalidating p). */
38      E remove(Position<E> p) throws IllegalArgumentException;
39  }
```

Doubly Linked List Implementation

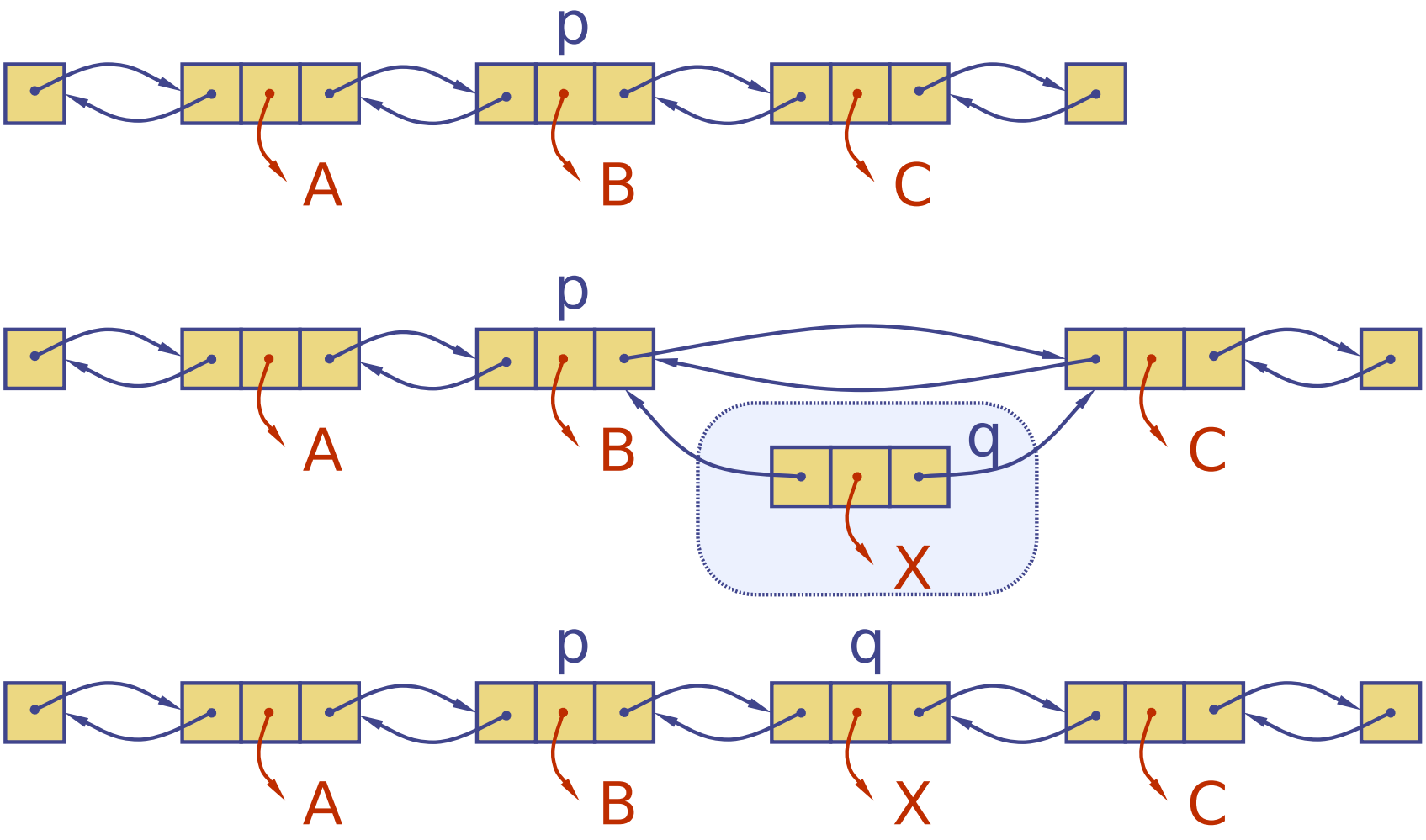
A simple way of implementing the positional list ADT uses a doubly linked list, where the nodes are the positions





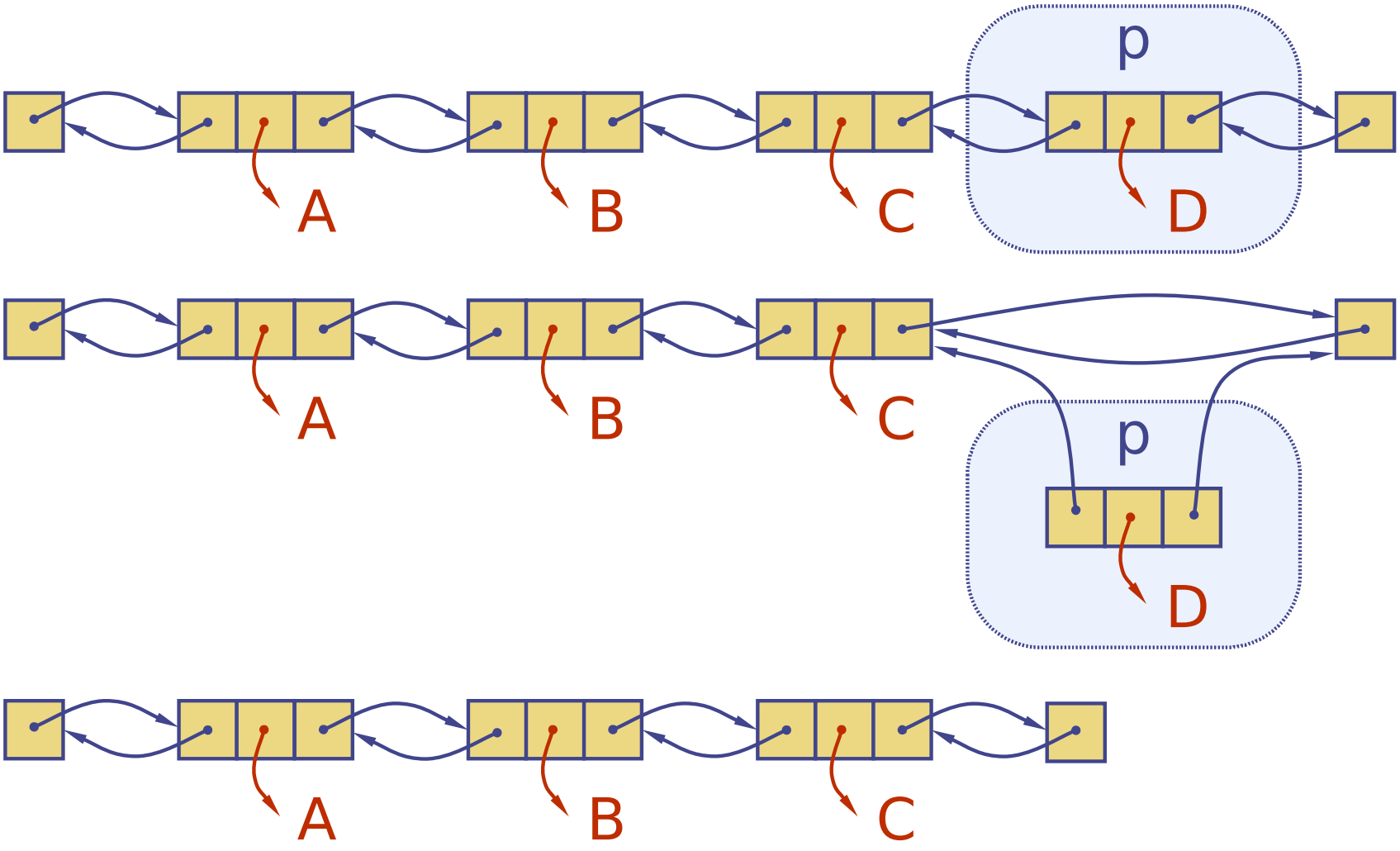
Linked Positional List: Element Insertion

Insert a new node q between p and its successor



Linked Positional List: Element Removal

Remove a node p from a doubly-linked list



Linked Positional List Implementation I

```
1  /** Implementation of a positional list stored as a doubly linked list. */
2  public class LinkedPositionalList<E> implements PositionalList<E> {
3    //---------- nested Node class ----------
4    private static class Node<E> implements Position<E> {
5      private E element;                    // reference to the element stored at this node
6      private Node<E> prev;                 // reference to the previous node in the list
7      private Node<E> next;                 // reference to the subsequent node in the list
8      public Node(E e, Node<E> p, Node<E> n) {
9        element = e;
10       prev = p;
11       next = n;
12     }
13     public E getElement() throws IllegalStateException {
14       if (next == null)                   // convention for defunct node
15         throw new IllegalStateException("Position no longer valid");
16       return element;
17     }
18     public Node<E> getPrev() { return prev; }
19     public Node<E> getNext() { return next; }
20     public void setElement(E e) { element = e; }
21     public void setPrev(Node<E> p) { prev = p; }
22     public void setNext(Node<E> n) { next = n; }
23   } //---------- end of nested Node class ----------
24
```

Linked Positional List Implementation II

```
25    // instance variables of the LinkedPositionalList
26    private Node<E> header;                          // header sentinel
27    private Node<E> trailer;                         // trailer sentinel
28    private int size = 0;                            // number of elements in the list
29
30    /** Constructs a new empty list. */
31    public LinkedPositionalList() {
32      header = new Node<>(null, null, null);         // create header
33      trailer = new Node<>(null, header, null);      // trailer is preceded by header
34      header.setNext(trailer);                       // header is followed by trailer
35    }
36
37    // private utilities
38    /** Validates the position and returns it as a node. */
39    private Node<E> validate(Position<E> p) throws IllegalArgumentException {
40      if (!(p instanceof Node)) throw new IllegalArgumentException("Invalid p");
41      Node<E> node = (Node<E>) p;                    // safe cast
42      if (node.getNext() == null)                    // convention for defunct node
43        throw new IllegalArgumentException("p is no longer in the list");
44      return node;
45    }
46
47    /** Returns the given node as a Position (or null, if it is a sentinel). */
48    private Position<E> position(Node<E> node) {
49      if (node == header || node == trailer)
50        return null;                                 // do not expose user to the sentinels
51      return node;
52    }
53
```

Linked Positional List Implementation III

```
54    // public accessor methods
55    /** Returns the number of elements in the linked list. */
56    public int size() { return size; }
57
58    /** Tests whether the linked list is empty. */
59    public boolean isEmpty() { return size == 0; }
60
61    /** Returns the first Position in the linked list (or null, if empty). */
62    public Position<E> first() {
63      return position(header.getNext());
64    }
65
66    /** Returns the last Position in the linked list (or null, if empty). */
67    public Position<E> last() {
68      return position(trailer.getPrev());
69    }
70
71    /** Returns the Position immediately before Position p (or null, if p is first). */
72    public Position<E> before(Position<E> p) throws IllegalArgumentException {
73      Node<E> node = validate(p);
74      return position(node.getPrev());
75    }
76
77    /** Returns the Position immediately after Position p (or null, if p is last). */
78    public Position<E> after(Position<E> p) throws IllegalArgumentException {
79      Node<E> node = validate(p);
80      return position(node.getNext());
81    }
82
```

Linked Positional List Implementation IV

```
83      // private utilities
84      /** Adds element e to the linked list between the given nodes. */
85      private Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {
86        Node<E> newest = new Node<>(e, pred, succ); // create and link a new node
87        pred.setNext(newest);
88        succ.setPrev(newest);
89        size++;
90        return newest;
91      }
92
93      // public update methods
94      /** Inserts element e at the front of the linked list and returns its new Position. */
95      public Position<E> addFirst(E e) {
96        return addBetween(e, header, header.getNext()); // just after the header
97      }
98
99      /** Inserts element e at the back of the linked list and returns its new Position. */
100     public Position<E> addLast(E e) {
101       return addBetween(e, trailer.getPrev(), trailer);    // just before the trailer
102     }
103
104     /** Inserts element e immediately before Position p, and returns its new Position. */
105     public Position<E> addBefore(Position<E> p, E e) throws IllegalArgumentException {
106       Node<E> node = validate(p);
107       return addBetween(e, node.getPrev(), node);
108     }
109
```

Linked Positional List Implementation V

```
110     /** Inserts element e immediately after Position p, and returns its new Position. */
111     public Position<E> addAfter(Position<E> p, E e) throws IllegalArgumentException {
112       Node<E> node = validate(p);
113       return addBetween(e, node, node.getNext());
114     }
115
116     /** Replaces the element stored at Position p and returns the replaced element. */
117     public E set(Position<E> p, E e) throws IllegalArgumentException {
118       Node<E> node = validate(p);
119       E answer = node.getElement();
120       node.setElement(e);
121       return answer;
122     }
123
124     /** Removes the element stored at Position p and returns it (invalidating p). */
125     public E remove(Position<E> p) throws IllegalArgumentException {
126       Node<E> node = validate(p);
127       Node<E> predecessor = node.getPrev();
128       Node<E> successor = node.getNext();
129       predecessor.setNext(successor);
130       successor.setPrev(predecessor);
131       size--;
132       E answer = node.getElement();
133       node.setElement(null);                    // help with garbage collection
134       node.setNext(null);                       // and convention for defunct node
135       node.setPrev(null);
136       return answer;
137     }
138   }
```

Linked Positional List: Analysis

For a positional list ADT implementation with a doubly linked list all operations run in worst-case constant time

This is in contrast to the `ArrayList` structure.
Which operations required linear time with an `ArrayList`?

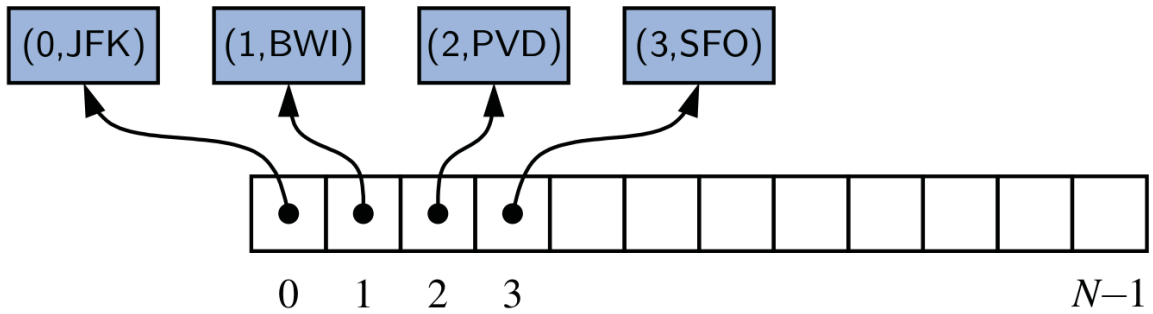
Method	**Time**
size()	$O(1)$
isEmpty()	$O(1)$
first(), last()	$O(1)$
before(p), after(p)	$O(1)$
addFirst(e), addLast(e)	$O(1)$
addBefore(p, e), addAfter(p, e)	$O(1)$
set(p, e)	$O(1)$
remove(p)	$O(1)$

Space usage: $O(n)$, where n is the number of elements in the list

Implementing a Positional List with an Array

We store a new kind of position object in each cell of array A

A position p stores the element e as well as the current index i of that element within the list



We can determine the index currently associated with a position

And we can determine the position currently associated with a specific index

During insertions/deletions, we loop through the array to update the index variable stored in all positions in the list that are shifted

`addFirst`, `addBefore`, `addAfter`, `remove` take $O(n)$ time

All the other position-based methods take $O(1)$ time

Summary

Reading

Section 7.3 Positional Lists

Questions?