UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 1

Methods of Representing Data, List Data Structure (Part 1)

Yoan Pinzón

© 2014

Table of Content Session 1

Preliminaries

Data Structure: Data object along with the relationships that exist among the instances and elements, and which are provided by specifying the operations of interest.

Our *main* concern will be:

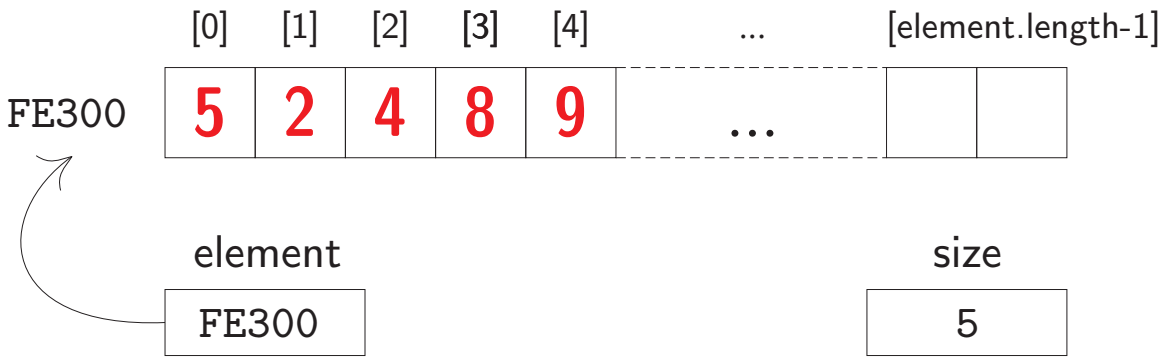
- The representation of data objects (actually of their instances)
- The representation should facilitate an *efficient* implementation of the operations

ADT - Abstract Data Type: A general way that provides a specification of the instances as well as of the operations that are to be performed.

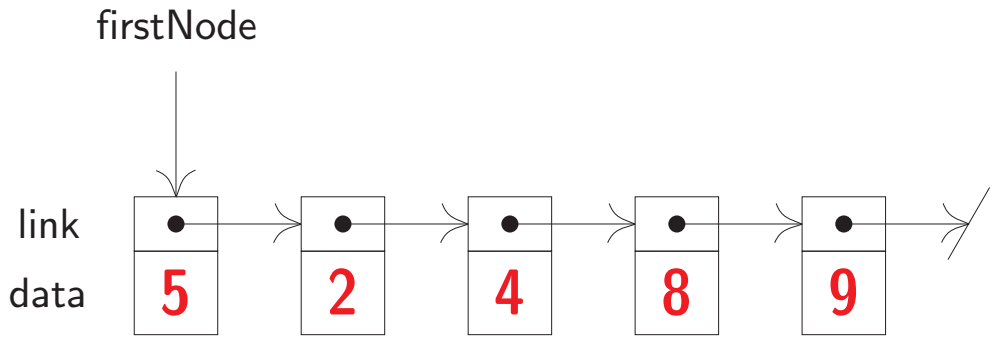
ADT representation is completely *independent* of the implementation

Methods of Representing Data

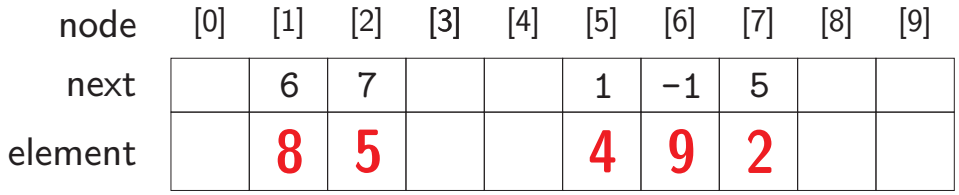
Array-based Representation: Uses an array to store either the list of elements or references to these elements.

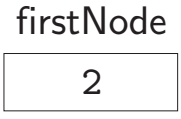


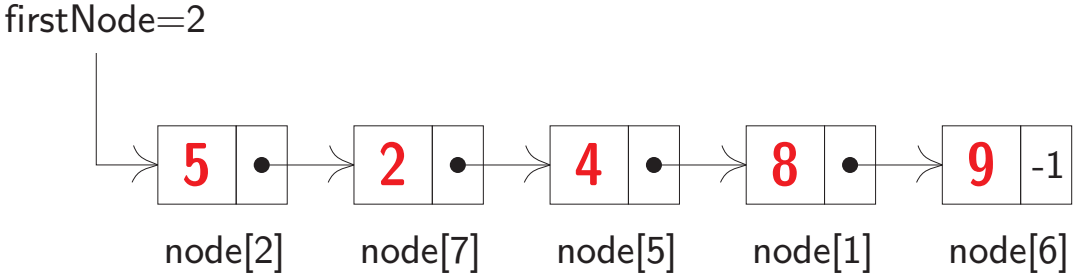
Linked Representation: The elements may be stored in any arbitrary set of memory locations. Each element keeps explicit information about the location of the next element called pointer (or link).



Simulated-pointers Representation: Similar to the linked list representation. However, pointers are replaced by integers.







Linear List Data Structure

Preliminaries

A **linear list** is a data object whose instances are of the form $(e_0, e_1, \ldots, e_{n-1})$. A linear list may be specified as an abstract data type (ADT) as follows:

AbstractDataType `LinearList`
{
instances: ordered finite collection of zero or more elements
operations:

`isEmpty( )`:	return `true` if the list is empty, `false` otherwise
`size( )`:	return the number of elements in the list
`get(`$index$`)`:	return the $index$th element of the list
`indexOf(`x`)`:	return the position of the first occurrence of x in the list, return -1 if x is not in the list
`remove(`$index$`)`:	remove and return the $index$th element, elements with higher index have their index reduced by 1
`add(`$index$`,` x`)`:	insert x as the $index$th element, elements with index $\geq$ $index$ have their index increased by 1
`output( )`:	output the list elements from left to right

}

Interface Definition of LinearList

```
3  package unal.datastructures;

5  public interface LinearList<T>
6  {
7     boolean isEmpty( );
8     int size( );
9     T get( int index );
10    int indexOf( T theElement );
11    T remove( int index );
12    void add( int index, T theElement );
13    String toString( );
14 }
```

We have changed the name of the output operation to `toString` because the standard output methods of Java invoke a method by this name for output.

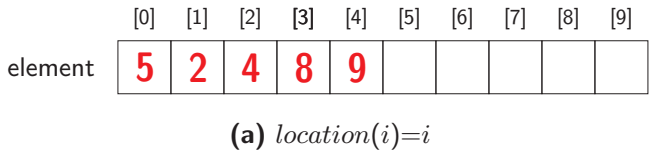
All methods of an interface are abstract (provides an implementation for no methods).

All methods declared in an interface are implicitly public, so the public modifier can be omitted.

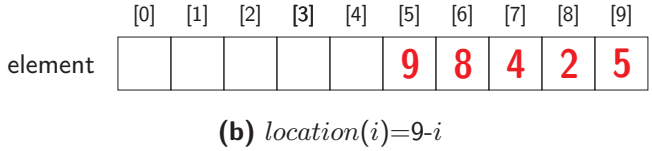
Linear List Data Structure

Array-based Representation

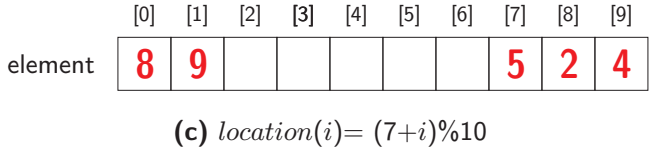
This representation uses an *array* to represent the instances of a linear list. A formula is used to map list elements into array positions. There are different ways of mapping:



(a) $location(i)=i$



(b) $location(i)$=9-i



(c) $location(i)= (7+i)\%10$

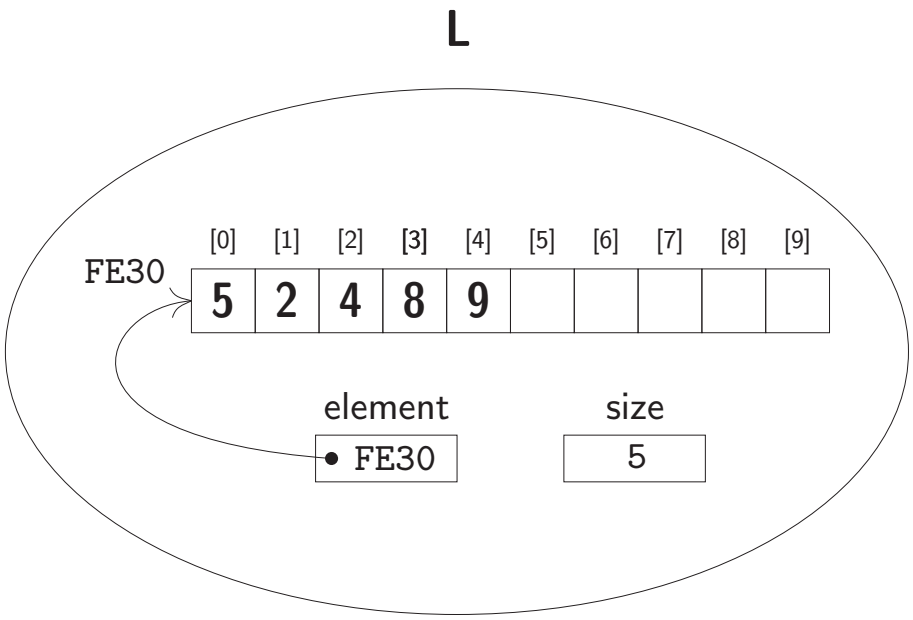
We will use formula **(a)** $location(i) = i$, thus the ith element of the list (if it exists) is at position i of the array.

Class Definition of ArrayLinearList

```
package unal.datastructures;

import java.util.*;

public class ArrayLinearList<T> implements LinearList<T>, ↙
  ↳ Iterable<T>
{
   // fields
   protected T[] element; // array of elements
   protected int size; // number of elements in array

   // constructors
   public ArrayLinearList(int initialCapacity) { /* ... */ }
   public ArrayLinearList( ) { /* ... */ }

   // methods
```

```
    public boolean isEmpty( ) { /* ... */ }
    public int size( ) { /* ... */ }
    void checkIndex( int index ) { /* ... */ }
    public T get( int index ) { /* ... */ }
    public int indexOf( T theElement ) { /* ... */ }
    public T remove( int index ) { /* ... */ }
    public void add( int index, T theElement ) { /* ... */ }
    public String toString( ) { /* ... */ }
    public Iterator<T> iterator( ) { /* ... */ }
    public static void main( String[] args ) { /* ... */ }
}
```


An instance of this class (ArrayLinearList L of integers) with `size=5` will look like this:



The length of the array `element` is often difficult to estimate. To overcome this, we ask the user to provide an estimate and then dynamically increase the length in case the user underestimated.

constructors

```
14  /** create a list with initial capacity initialCapacity
15   * @throws IllegalArgumentException when
16   * initialCapacity < 1 */
17  @SuppressWarnings( "unchecked" )
18  public ArrayLinearList(int initialCapacity)
19  {
20     if( initialCapacity < 1 )
21        throw new IllegalArgumentException
22              ( "initialCapacity must be >= 1" );
23     element = ( T[] ) new Object[ initialCapacity ];
24     size = 0;
25  }

27  /** create a list with initial capacity 10 */
28  public ArrayLinearList( )
29  {
30     this( 10 );
31  }
```


isEmpty

```
34  /** @return true iff list is empty */
35  public boolean isEmpty( )
36  {
37     return size == 0;
38  }
```

size

```
40    /** @return current number of elements in list */
41    public int size( )
42    {
43       return size;
44    }
```


checkIndex

```
46    /** @throws IndexOutOfBoundsException when
47     * index is not between 0 and size - 1 */
48    void checkIndex( int index )
49    {
50       if( index < 0 || index >= size )
51          throw new IndexOutOfBoundsException
52                ( "index = " + index + "  size = " + size );
53    }
```

get

```
55  /** @return element with specified index
56   * @throws IndexOutOfBoundsException when
57   * index is not between 0 and size - 1 */
58  public T get( int index )
59  {
60     checkIndex( index );
61     return element[ index ];
62  }
```


indexOf

```
64  /** @return index of first occurrence of theElement,
65   * return -1 if theElement not in list */
66  public int indexOf( T theElement )
67  {
68     // search element[] for theElement
69     for( int i = 0; i < size; i++ )
70        if( element[ i ].equals( theElement ) )
71           return i;
72     // theElement not found
73     return -1;
74  }
```

remove

```
76    /** Remove the element with specified index.
77     * All elements with higher index have their
78     * index reduced by 1.
79     * @throws IndexOutOfBoundsException when
80     * index is not between 0 and size - 1
81     * @return removed element */
82    public T remove( int index )
83    {
84       checkIndex( index );

86       // valid index, shift elements with higher index
87       T removedElement = element[ index ];
88       for( int i = index + 1; i < size; i++ )
89          element[ i - 1 ] = element[ i ];

91       element[ --size ] = null; // enable garbage collection
92       return removedElement;
93    }
```


add

```
95    /** Insert an element with specified index.
96     * All elements with equal or higher index
97     * have their index increased by 1.
98     * @throws IndexOutOfBoundsException when
99     * index is not between 0 and size */
100   @SuppressWarnings( "unchecked" )
101   public void add( int index, T theElement )
102   {
103      if( index < 0 || index > size )
104      // invalid list position
105         throw new IndexOutOfBoundsException
106               ( "index␣=␣" + index + "␣␣size␣=␣" + size );

108      // valid index, make sure we have space
109      if( size == element.length )
110      {
111         // no space, double capacity
112         T[] old = element;
113         element = ( T[] ) new Object[ 2 * size ];
```

```
114         System.arraycopy( old, 0, element, 0, size );
115      }

117      // shift elements right one position
118      for( int i = size - 1; i >= index; i-- )
119         element[ i + 1 ] = element[ i ];

121      element[ index ] = theElement;

123      size++;
124   }
```

Why do we double the array length and not simply increase the length by 1 or 2?

toString

```
126   /** convert to a string */
127   @Override
128   public String toString( )
129   {
130      StringBuilder s = new StringBuilder( "[" );

132      // put elements into the buffer
133      for( T x : this )
134         s.append( Objects.toString( x ) + ",␣" );

136      if( size > 0 )
137         s.setLength( s.length( ) - 2 ); // remove last ", "

139      s.append( "]" );

141      // create equivalent String
142      return new String( s );
143   }
```

iterator

```
145    /** create and return an iterator */
146    @Override
147    public Iterator<T> iterator( )
148    {
149       return new ArrayLinearListIterator<T>( this );
150    }
```


main

```
152    /** test program */
153    public static void main( String[] args )
154    {
155       // test default constructor
156       ArrayLinearList<Integer> x = new ArrayLinearList<>( );

158       // test size
159       System.out.println( "Initial␣size␣is␣" + x.size( ) );

161       // test isEmpty
162       if( x.isEmpty( ) )
163          System.out.println( "The␣list␣is␣empty" );
164       else System.out.println( "The␣list␣is␣not␣empty" );

166       // test put
167       x.add( 0, new Integer( 2 ) );
168       x.add( 1, new Integer( 6 ) );
169       x.add( 0, new Integer( 1 ) );
170       x.add( 2, new Integer( 4 ) );
```

```
171       System.out.println( "List␣size␣is␣" + x.size( ) );
173       // test toString
174       System.out.println( "The␣list␣is␣" + x );
176       // output using an iterator
177       Iterator y = x.iterator( );
178       while( y.hasNext( ) )
179          System.out.print( y.next( ) + "␣" );
180       System.out.println( );
182       // test indexOf
183       int index = x.indexOf( new Integer( 4 ) );
184       if( index < 0 )
185          System.out.println( "4␣not␣found" );
186       else System.out.println( "The␣index␣of␣4␣is␣" + index );
188       index = x.indexOf( new Integer(3) );
189       if( index < 0 )
190          System.out.println( "3␣not␣found" );
191       else System.out.println( "The␣index␣of␣3␣is␣" + index );
```



```
193       // test get
194       System.out.println( "Element␣at␣0␣is␣" + x.get( 0 ) );
195       System.out.println( "Element␣at␣3␣is␣" + x.get( 3 ) );
197       // test remove
198       System.out.println( x.remove( 1 ) + "␣removed" );
199       System.out.println( "The␣list␣is␣" + x );
200       System.out.println( x.remove( 2 ) + "␣removed" );
201       System.out.println( "The␣list␣is␣" + x );
203       if( x.isEmpty( ) )
204          System.out.println( "The␣list␣is␣empty" );
205       else System.out.println( "The␣list␣is␣not␣empty" );
207       System.out.println( "List␣size␣is␣" + x.size( ) );
208    }
```

Compiling `ArrayLinearList.java`

```
C:\2016699\code> javac unal\datastructures\ArrayLinearList.java ↵
C:\2016699\code> java unal.datastructures.ArrayLinearList ↵
Initial size is 0
List is empty
List size is 4
The list is [1, 2, 4, 6]
1 2 4 6
The index of 4 is 2
3 not found
Element at 0 is 1
Element at 3 is 6
2 removed
The list is [1, 4, 6]
6 removed
The list is [1, 4]
List is not empty
List size is 2
```


class ArrayLinearListIterator

```
211 class ArrayLinearListIterator<T> implements Iterator<T>
212 {
213    // fields
214    private ArrayLinearList<T> list; // list to be iterated
215    private int nextIndex; // index of next element

217    // constructor
218    public ArrayLinearListIterator( ArrayLinearList<T> theList )
219    {
220       list = theList;
221       nextIndex = 0;
222    }

224    // methods
225    /** @return true iff the list has a next element */
226    public boolean hasNext( )
227    {
228       return nextIndex < list.size;
229    }
```

```
231     /** @return next element in list
232      * @throws NoSuchElementException
233      * when there is no next element */
234     public T next( )
235     {
236        if( nextIndex < list.size )
237           return list.element[ nextIndex++ ];
238        else
239           throw new NoSuchElementException( "No next element" );
240     }

242     /** unsupported method */
243     public void remove( )
244     {
245        throw new UnsupportedOperationException
246              ( "remove not supported" );
247     }
248 }
```