Data structures

- A **data structure** is data type

  that is

  either simple or aggregate,

  built-in or user-defined, or

  an organized collection of these

- a particular way of organizing data in a computer memory so that it can be accessed efficiently

Data type

- A data type is characterized by:
  - a set of *values*
  - a *data representation,* which is common to all these values, and
  - a set of *operations,* which can be applied uniformly to all these values

Primitive data types in Java

- Java provides eight primitive types:
  - boolean(1), char(16)
  - byte(8), short(16), int(32), long(64)
  - float(32), double(64)
- Each primitive type has
  - a set of values (int: -2^{31}… $2^{31}-1$)
  - a data representation (32-bit)
  - a set of operations (+, -, *, /, etc.)
- These are "**fixed**"—the programmer cannot change anything

ADT = Abstract + Data Type

- To Abstract is to leave out information, keeping (hopefully) the more important parts

  What part of a Data Type does an ADT leave out?

- An Abstract Data Type (ADT) has:
  - a set of *values*
  - a set of *operations*, which can be applied uniformly to all these values

It is NOT characterized by its data representation.

- Data representation is private, and changeable, with no effect on application code.

Example ADT

- Simplest ADT is a Bag
  - no implied order to the items
  - duplicates allowed
  - items can be added, removed, accessed
- Another ADT is a Set
  - same as a bag, except duplicate elements not allowed
  - union, intersection, difference, subset

Other types of ADT

- List
- Stack
- Queue
- Tree
- Binary Search Trees
- Heaps
- AVL and Red-Black Trees
- B Trees
- Hash Tables
- Maps
- Graphs

ADTs

- many, many different ADTs
  - picking the right one for the job is an important step in design
- High level languages often provide built in ADTs,
  - the Java standard library

Implementation of ADTs

ADTs are implemented

in Java using classes, and

in C++ using classes and structures

Model for an abstract data type

The ADT model is shown in Figure below

There are two different parts of the model:

data structure and operations (public and private).

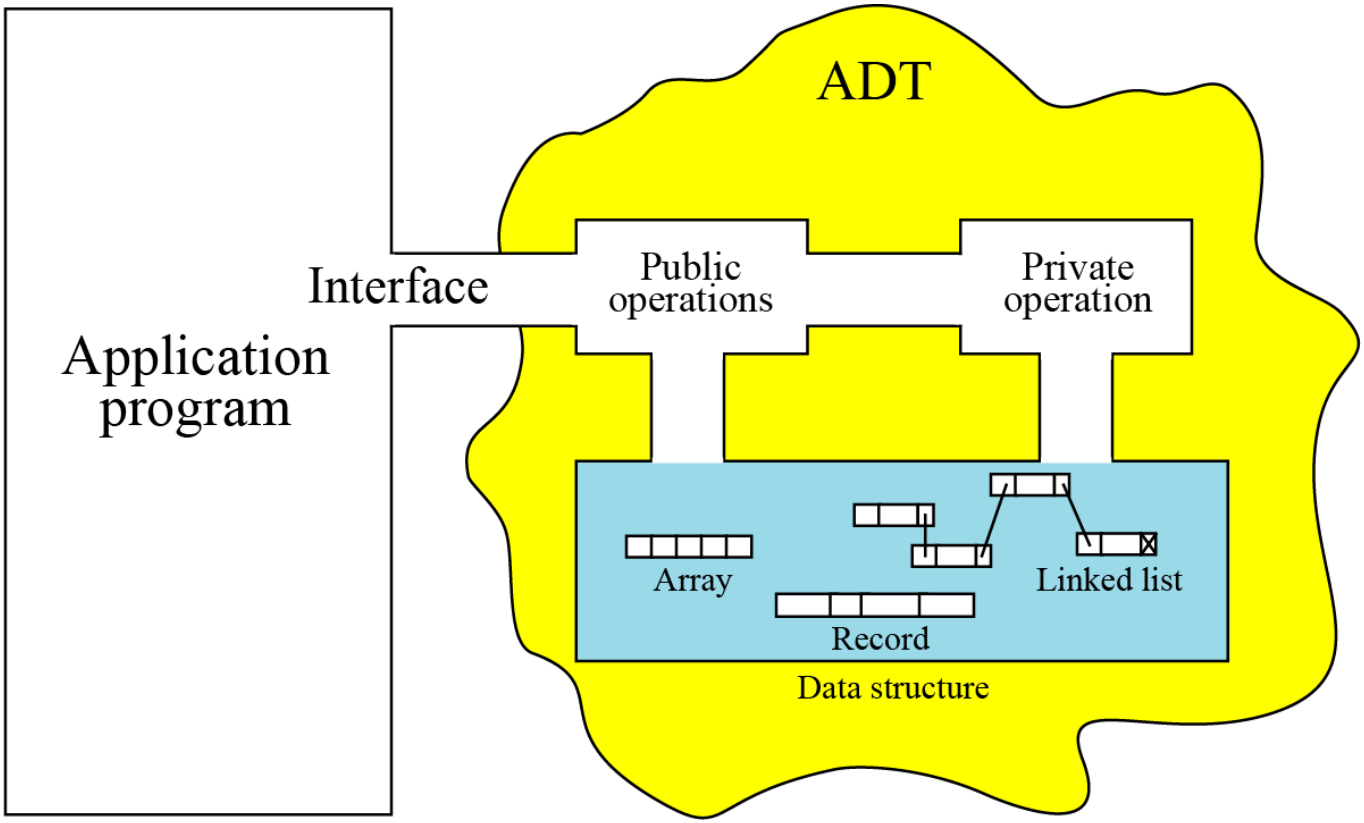


Figure: The model for an ADT

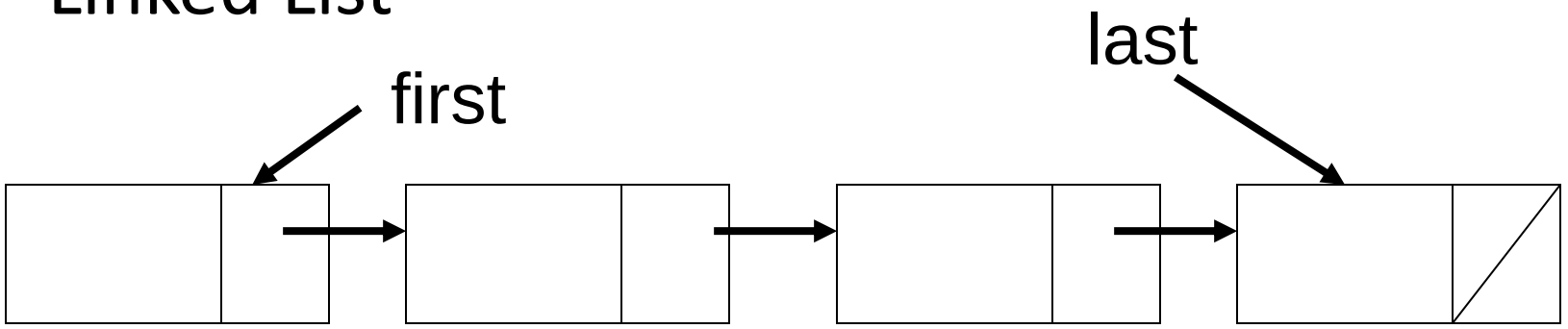
Lists ADT

- Linear collection of items (or elements, data)
- Operations: add, delete,…

Implementation of List ADT

- Array List

- Linked List



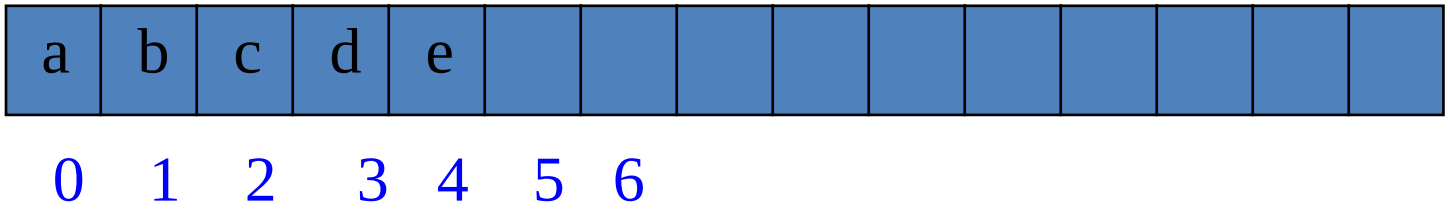
Data representation methods

array

linked

Array List Representation

use a one-dimensional array element[]



L = (a, b, c, d, e)

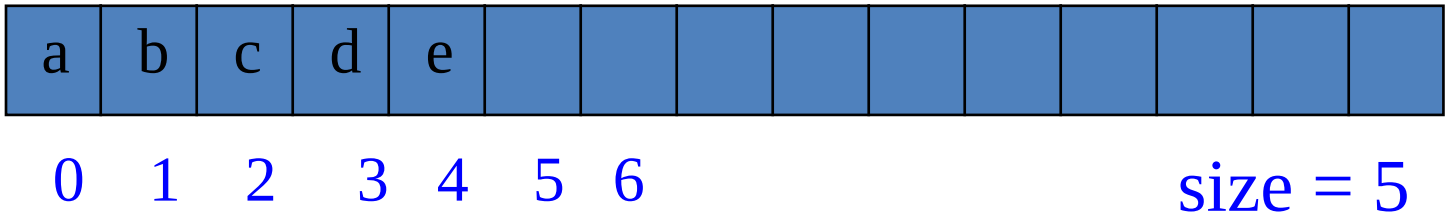
Store element i of list in element[i].

Right to left mapping

Mapping that skips every other position

Wrap around mapping

Representation used



put element i of list in element[i]

use a variable size to record current number of elements

Add/Remove an element

size = 5

add(position,element)

add(1,g)

size = 6

Data type of array element[]

Data type of list elements is unknown.

Define element[] to be of data type Object.

data types (int, float, double, char, etc.)

Primitive Type	Wrapper Type
`int`	`Integer`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

Two questions

- Difference between int and Integer?
  (Similarly, double and Double)

- Can we implement a generic list class?

- Can we create a dynamic size array?

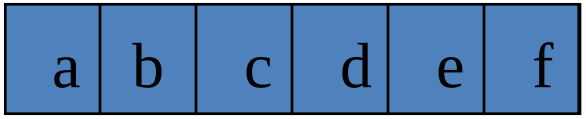
Length of array element[]

Don’t know how many elements will be in your list.

Must pick an initial length and dynamically increase as needed.

Increasing array length

Length of array element[] is 6.

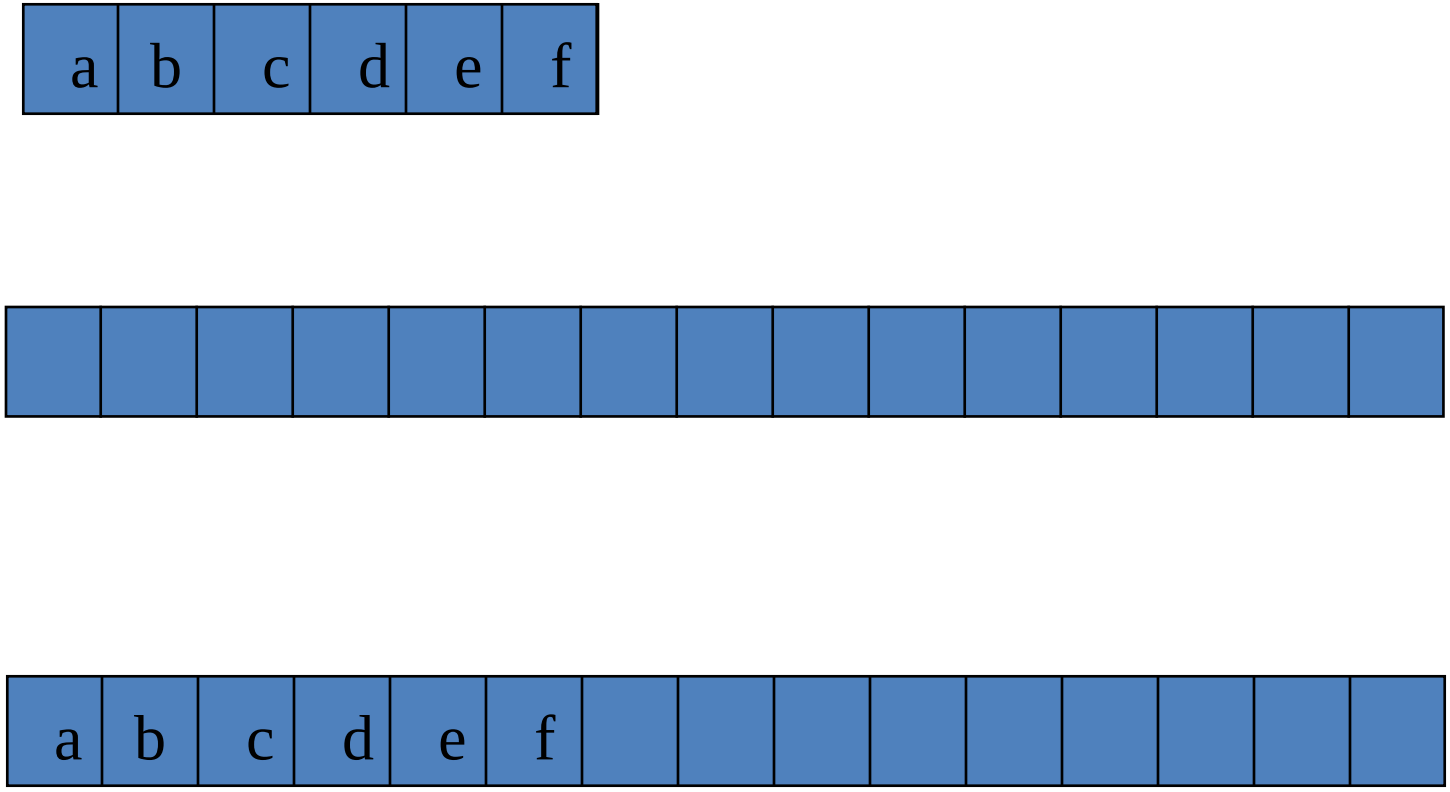


First create a new and larger array

newArray = new Object[15];

Increasing array length

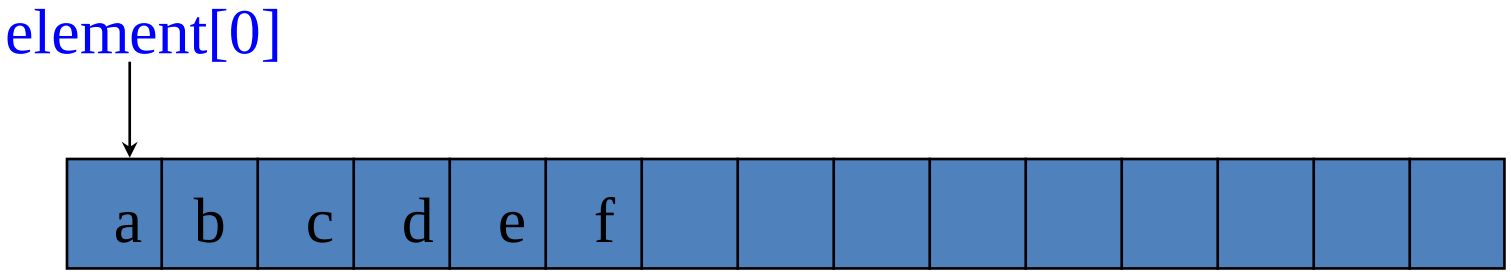
Now copy elements from old array to new one.



Increasing array length

Finally, rename new array.

element = newArray;



element.length = 15

Altogether now

```
// create a new array of proper length and data type
Object [] newArray = new Object [newLength];

// copy all elements from old array into new one
System.arraycopy(element, 0, newArray, 0,
        element.length);

// rename array
element = newArray;
```

```
public static Object [] changeLength(Object [] a,
                                     int newLength)
{
    Object [] newArray = new Object [newLength];
    System.arrayCopy(…);
    return newArray;
}
……
element = changeLength(element, 100);
```

How big should the new array be?

At least 1 more than current array length.

Cost of increasing array length is

O(new length)

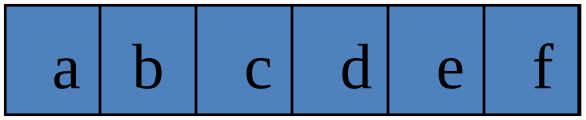
Cost of n add operations done on an initially empty linear list increases by

$O(n^2)$

When array length is increased by 1 each time we need to resize the array element[], the cost of n add operations goes up by $O(n^2)$.

Space complexity

element[6]

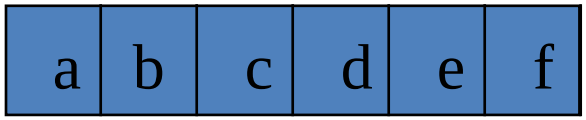


newArray = new char[7];

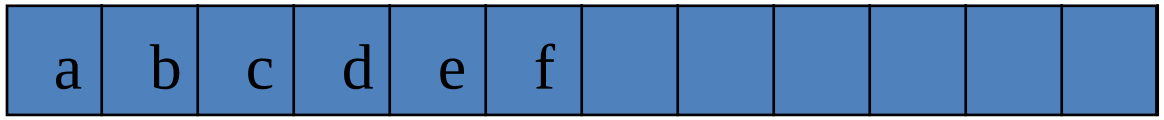
space needed = 2 * newLength – 1

Array doubling

Double the array length.



newArray = new char[12];



Time for n adds goes up by O(n).

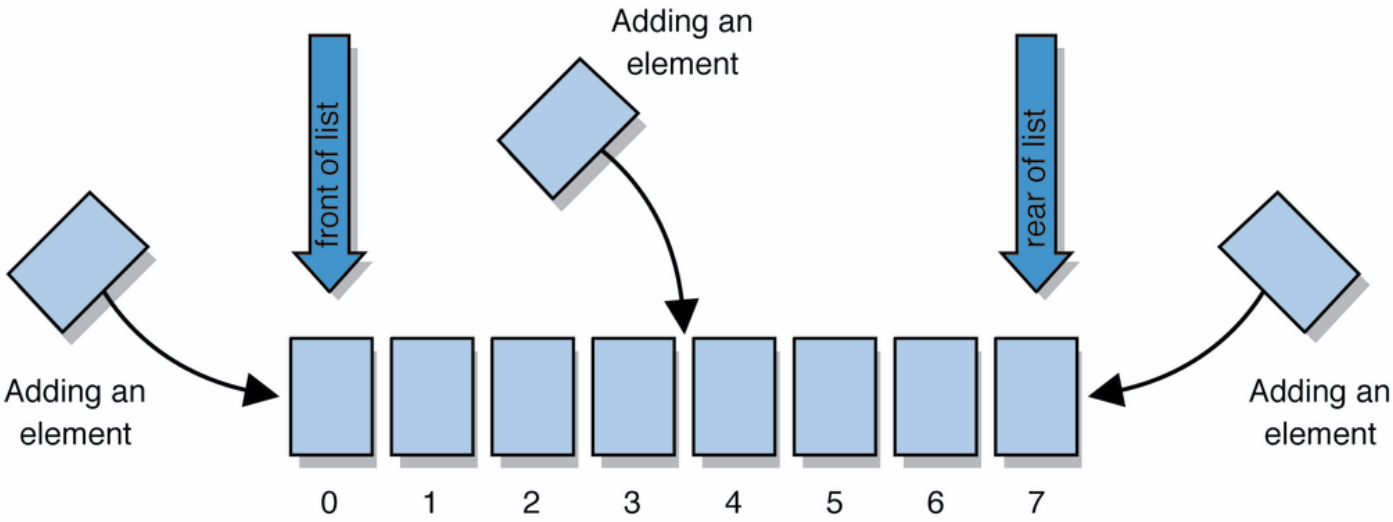
Space needed = 1.5*newLength.

The Class ArrayLinearList

- General purpose implementation of linear lists.

Lists

- a collection storing a sequence of elements
  - each element is accessible by a 0-based **index**
  - a list has a **size** (# of elements present)
  - element can be added to the front, back, or elsewhere
  - in Java, a list can be represented as an **`ArrayLinearList`** object



Linear List

Method Summary

void **addToFront**(Object element)
Adds the given element to the front of the current list

void **addToRear**(Object element)
Adds the given element to the rear of the current list

boolean **isEmpty**()
Predicate returns true if the list is empty and false otherwise

Object **removeFront**()
Removes the first element in the list and returns a reference to it.

Object **removeRear**()
Removes the last element in the list and returns a reference to it.

int **size**()
Returns the number of nodes in the list

String **toString**()
Returns a string representation of the list.

Create an empty List

```
ArrayLinearList a = new ArrayLinearList(100);

        b = new ArrayLinearList()
```

The class ArrayLinearList

```
/** array implementation of  LinearList */
import java.util.*;    // has Iterator interface
import utilities.*;    // has array resizing class

public class ArrayLinearList  // implements LinearList
{
    // data members
       protected Object [] element;  // array of elements
       protected int size;    // number of elements in array
   // constructors and other methods come here
}
```

A Constructor

```
/** create a list with initial capacity initialCapacity
   * @throws IllegalArgumentException when
   * initialCapacity < 1 */
  public ArrayLinearList(int initialCapacity)
   {
      if (initialCapacity < 1)
        throw new IllegalArgumentException
                      ("initialCapacity must be >= 1");
      element = new Object [initialCapacity];
       size = 0;
   }
```

Another Constructor

```
/** create a list with initial capacity 10 */

  public ArrayLinearList()
  {// use default capacity of 10
    this(10);
  }
```

The Method isEmpty

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

The Method size()

```
/** @return current number of elements in list */
 public int size()
    {return size;}
```

The Method checkIndex

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

The Method get

```
/** @return element with specified index
  * @throws IndexOutOfBoundsException when
  * index is not between 0 and size - 1 */
 public Object get(int index)
 {
   checkIndex(index);
   return element[index];
 }
```

The Method indexOf

```
/** @return index of first occurrence of theElement,
  * return -1 if theElement not in list */
public int indexOf(Object theElement)
{
   // search element[] for theElement
   for (int i = 0; i < size; i++)
      if (element[i].equals(theElement))
         return i;

   // theElement not found
   return -1;
}
```

The Method remove

```
public Object remove(int index)
 {
    checkIndex(index);

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

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

The Method add

```
public void add(int index, Object theElement)
   {
     if (index < 0 || index >= size)
       // invalid list position
       throw new IndexOutOfBoundsException("index = " + index + "
                                                  size = " + size);
      // valid index, make sure we have space
     if (size == element.length)
       // no space, double capacity
       element = ChangeArrayLength.changeLength1D(element,
                                                  2*size);
```

The Method add

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

    element[index] = theElement;

    size++;
}
```