## Container Classes

Bags & Sequences

### Operations on a Bag

- **Constructors**
  - default: creates bag with capacity of 10
  - with int argument: creates bag with specified capacity

### Operations on a Bag

- **Modifiers:**
  - add -- place an item in a bag
  - addMany, addAll -- variations on add
  - remove -- take an item out of a bag
  - ensureCapacity -- enlarges bag if necessary
  - trimToSize -- shrinks bag to current content capacity

### Operations on a Bag

- **Observers:**
  - size -- returns total # of items in bag
  - countOccurrences -- returns # of items of a specific target value present in bag
  - getCapacity -- returns current size of bag
  - clone -- returns a copy of the calling bag

### Invariant of an ADT

- An invariant is an explicit statement of the rules dictating how member variables are to be used
- All methods may assume that the invariant is valid when they are called
- Each method is responsible for ensuring continuing validity of the invariant when method returns
Invariant for Bag
1. The number of items contained in the bag is stored in member variable manyItems
2. The bag entries are stored in member variable data, from position data[0] to position data[manyItems-1]

Bag implementation: class heading & private members

public class IntArrayBag implements Cloneable
{
    // we add the “implements” clause so that the clone() method can be used to create duplicate Bags
    private int[] data;
    private int manyItems;
    // purpose of each of these is described in the class invariant

Bag definition: constructors

public IntArrayBag() // default constructor
{
    final int INITIAL_CAPACITY = 10;
    manyItems = 0;
    data = new int[INITIAL_CAPACITY];
}

Bag definition: constructors

public IntArrayBag(int initialCapacity) {
    // write code that enforces precondition that
    // parameter is greater than 0 and sets up initial bag
    // if precondition is met:

    }

Implementation of add()

public void add(int element) {
    if (manyItems == data.length)
        ensureCapacity((manyItems + 1)*2);
    data[manyItems] = element;
    manyItems++;
}

Notes on add()

• If adding an element to the array would exceed the current array capacity, the array is resized to accommodate the new data
• This resizing is accomplished with a call to ensureCapacity (next slide)
• Note that the array is resized to be twice its current capacity; this is meant to minimize the calls to ensureCapacity, a relatively costly operation
**Implementation of ensureCapacity()**

```java
public void ensureCapacity(int minimumCapacity) {
    int[] biggerArray;
    if (data.length < minimumCapacity)
        {  
            biggerArray = new int[minimumCapacity];
            System.arraycopy(data, 0, biggerArray, 0, manyItems);
            data = biggerArray;
        }
}
```

**Notes on ensureCapacity**

- This method and the next couple use the method System.arraycopy to copy elements from the parameter object’s array into the calling object’s array
- Syntax:
  System.arraycopy (source array, source index, destination array, destination index, number of elements to copy)

**Implementation of addMany()**

```java
public void addMany(int... elements) {
    if (manyItems + elements.length > data.length)
        ensureCapacity((manyItems + elements.length)*2);
    System.arraycopy(elements, 0, data, manyItems, elements.length);
    manyItems += elements.length;
}
```

**Notes on addMany()**

- The parameter notation (int ... elements) indicates the method takes a variable number of int arguments
- The argument(s) passed to the method are treated like an array; for example, their number can be found by accessing the length member of the parameter (elements.length)

**Implementation of addAll()**

```java
public void addAll(IntArrayBag addend) {
    ensureCapacity(manyItems + addend.manyItems);
    System.arraycopy(addend.data, 0, data, manyItems, addend.manyItems);
    manyItems += addend.manyItems;
}
```

**Implementation of remove()**

```java
public boolean remove(int target) {
    // precondition: none
    // postcondition: seeks target and, if it exists, removes an instance of
    // target value from bag, returning true; if target not present, returns false
}
```
Implementation of trimToSize()

public void trimToSize() {
    int[] trimmedArray;
    if (data.length != manyItems) {
        trimmedArray = new int[manyItems];
        System.arraycopy(data, 0, trimmedArray, 0, manyItems);
        data = trimmedArray;
    }
}

Implementation of countOccurences

public int countOccurences(int target) {
    // precondition: none
    // postcondition: returns number of times target
    // value appears in array

}

Implementation of clone()

public IntArrayBag clone() {
    IntArrayBag answer;
    try {
        answer = (IntArrayBag) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new RuntimeException("This class does not implement Cloneable");
    }
    answer.data = data.clone();
    return answer;
}

Implementation of observer methods

public int getCapacity() {
    }

public int size() {
    }

Notes on clone()

- The call to super.clone() is used to clone the object, but doesn’t clone all of the object’s contents
- Note the separate housekeeping issue of cloning the array; in general, you will face this issue when cloning objects that contain object types as member variables

Implementation of union

public static IntArrayBag union(IntArrayBag b1, IntArrayBag b2) {
    IntArrayBag answer = new IntArrayBag(b1.getCapacity() +
    b2.getCapacity());
    System.arraycopy(b1.data, 0, answer.data, 0, b1.manyItems);
    System.arraycopy(b2.data, 0, answer.data, b1.manyItems,
    b2.manyItems);
    answer.manyItems = b1.manyItems + b2.manyItems;
    return answer;
}
Sequence ADT -- a different kind of container

- Items in a sequence are arranged in order
- Items are kept in sequence deliberately: member functions can step through the sequence one item at a time, in order
- Can control where items are inserted and removed

Invariant for the Sequence ADT

- The number of elements in the Sequence is stored in member variable manyItems
- The data are stored in array data, from data[0] to data[manyItems - 1]
- The current item is found in data[currentIndex]; if there is no current item, currentIndex == manyItems

Class definition for Sequence ADT

- Like the Bag class, Sequence implements the Cloneable interface
- Private members include the three mentioned in the invariant

Class definition for Sequence ADT -- iterators

- Iterators: methods that provide capability to step through items in a container
  - Keep in mind: although we are currently implementing our ADTs using arrays, this will not always be the case
  - Furthermore, client programmers can’t access private class members – so array indexing isn’t available as an option for clients

Iterator methods for Sequence

- start( ) makes the first Item in the List the current item
- advance( ) moves to the next item
- isCurrent( ) ensures that there is an item at the current index
- getCurrent( ) allows access to the current item

Example of use of iterators & observers: printing a list

```java
Sequence myList = new Sequence();

// code omitted here – presumably, something gets added
for (myList.start( ); myList.isCurrent( ); myList.advance( ))
  System.out.println(myList.getCurrent( ) );
```
Class definition for Sequence ADT -- modifiers

- methods addAfter() and addBefore insert items into the sequence
  - addAfter() places the new element after the current element
  - addBefore() places the new element before the current element
  - In either case, the added element becomes the new current item
- method removeCurrent removes the current item, provided one exists at the current index

Other methods

- Like Bag, Sequence is a container class, so many of the same methods (or very similar methods) are also provided; they include:
  - constructors
  - clone()
  - ensureCapacity()
  - trimToSize()
  - addAll()
  - getCapacity
  - a static method called concatenation: similar to Bag class’s union method

Member function implementations

```java
// constructor -- creates empty sequence
public Sequence (int size) {
    manyItems = 0;
    data = new int[size];
    currentIndex = manyItems;
}
```

Member function implementations

```java
// size method returns count of Items in List
public int size () {
    }
}
```

Member function implementations

```java
// start method -- places cursor at start of List
public void void start() {
    }
}
```

Member function implementations

```java
// getCurrent method returns value of current Item
public int getCurrent() {
    try {
        isCurrent() == true;
    } catch (Exception e) {
        System.err.print (“Error – no current item”);
        System.exit(1);
    }
    return data[currentIndex];
}
```
Member function implementations

// advance method moves to next position in List
public void advance( )
{
    currentIndex++;}
}

Continuation of insert()

// make room for new Item from current_index on
// if valid Item exists
else
for (int n=manyItems; n>=currentIndex; n--)
    data[n] = data[n-1];

// add new entry
data[currentIndex] = entry;

// restore invariant
manyItems++;

Member function implementations

// addBefore method-- add item in front of current item
public void addBefore(int entry)
{
    if (size() >= manyItems)
        ensureCapacity((manyItems + 1)*2);
    if (!isCurrent())
    {
        for (int x=manyItems; x>0; x--)
            data[x] = data[x-1];
        start();
    }
}

Member function implementations

// addAfter method-- add entry after current Item
public void addAfter(int entry)
{
    int n;
    if (size() <= manyItems)
        ensureCapacity((manyItems+1)*2);
    // make room for new entry
    for (n=manyItems; n>currentIndex+1; n--)
        data[n] = data[n-1];
    // add new entry
data[n] = entry;
    // restore invariant
currentIndex = n;
    manyItems++;
}

Member function implementations

public boolean isCurrent()
{
    if (currentIndex < manyItems)
        return true;
    return false;
}
Container Classes

Bags & Sequences