Data Structures

BSc in Computer Science
University of New York, Tirana

Assoc. Prof. Marenglen Biba

General info

- Course : Data Structures (3 credit hours)
- Instructor : Assoc. Prof. Marenglen Biba
- Office : Faculty building 2nd floor
- Office Hours : Wednesday 12-14 or by appointment
- Phone : 42273056 / ext. 112
- E-mail : marenglenbiba@unyt.edu.al
- Course page : http://www.marenglenbiba.net/ds/
- Course Location and Time
- Laboratory Room LAB3, Monday 14-17.

Course description

- This course couples work on program design, analysis, and verification with an introduction to the study of data structures.
- Data structures capture common ways to store and manipulate data, and they are important in the construction of sophisticated computer programs.
- Upon completion of this course, students should be able to: work on almost all widely used data structures.

Course Outcomes

- Upon course completion, students will have demonstrated the ability to do the following:
- understand the advantages of data structures as useful abstractions in programming
- understand the concept of a linked list structure, recite different implementations of linked lists.
- understand the concept of a LIFO structure; implement and use stacks
- understand the concept of a FIFO structure; implement and use queues
- understand the concept of the tree data structure; implement and use trees in applications
- understand the concept of hash tables; implement and use them effectively
- understand the concept of graphs; implement and use effectively
- understand algorithm analysis and evaluate performance

Required Readings

- Data Structures and Problem Solving Using Java, 4/E. Mark A. Weiss. Addison-Wesley, 2010. ISBN-10: 0321541405.

Content of the Course

- Introduction to Data Structures
- Algorithm analysis
- Linked Lists
- Stacks
- Queues
- Trees
- Hash tables
- Graphs
- Priority Queues

Assumptions for this Class

- Programming in Java

Grading Policy

Project	40%
Midterm	30%
Final	30%

Recommendations

- Start studying now
- Do not be shy! Ask any questions that you might have. Every questions makes you a good candidate.
- The professor is a container of knowledge and the goal is to get most of him, thus come and talk.
- Respect the deadlines
- Respect the appointments
- Try to study from more than one source, Internet is great!
- If you have any problems come and talk with me in advance so that we can find an appropriate solution

GOOD LUCK!

Outline

- Stacks
- Stack: Implementation with Array
- Stack: Implementation with Linked List

Chapter 16

Stacks

ADT and Classes

- **Abstract data type**
  - A type whose implementation is hidden from the rest of the system

- **Class:**
  - An abstraction in the context of object-oriented languages
  - A class encapsulates state and behavior

ADTs

- An ADT represents an abstract way to store data.
- This is different from normal data types, such as int or char.
- ADTs offer interesting ways to store data, depending on what the use of the data is for.

Linear Data Structures

- The defining property of a list is that the elements are organized linearly, that is, every element has one element immediately before it and another immediately after it (except, of course, the elements at the beginning and end of the list).

Stack

- The stack is an example of a constrained linear data structure.
- In a stack, the elements are ordered from most recently added (the top) to the least recently added (the bottom).
- All insertions and deletions are performed at the top of the stack.
- You use the push operation to insert an element onto the stack and the pop operation to remove the topmost stack element.

Stack: LIFO mode of operation

- The constraints on insertion and deletion produce the last in, first out (LIFO) behavior that characterizes a stack.
- Although the stack data structure is narrowly defined, it is used so extensively by systems software that support for a primitive stack is one of the basic elements of most computer architectures.

The first ADT: The Stack

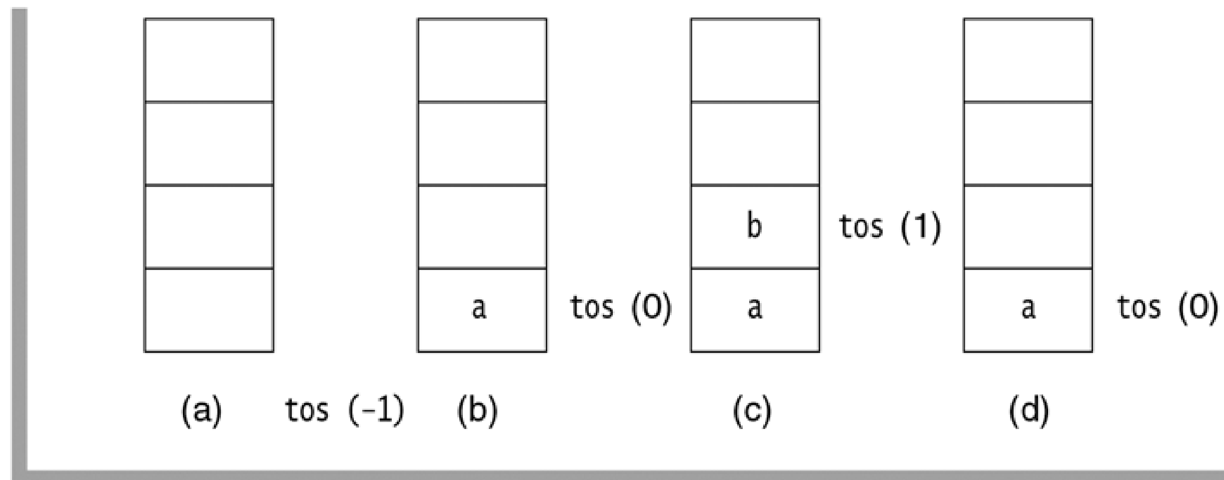

figure 16.1

How the stack routines work:
(a) empty stack;
(b) `push(a);`
(c) `push(b);`
(d) `pop( )`

Stack

- Two implementations:
  - Based on Array
  - Based on Linked List

Constructor

Stack ( )

Precondition:

- None.

Postcondition:

- creates an empty stack and allocates enough memory for a stack of default capacity.

Methods in the Interface

void push (Object newElement)

Precondition:

- newElement is not null.

Postcondition:

- Inserts newElement onto the top of a stack.

Methods in the Interface

pop ( )

Precondition:

- Stack is not empty.

Postcondition:

- Removes the most recently added (top) element from a stack.

Methods in the Interface

void makeEmpty ( )

Precondition:

- None.

Postcondition:

- Removes all the elements in a stack.

Methods in the Interface

boolean isEmpty ( )

- Precondition:

None.

- Postcondition:

Returns true if a stack is empty. Otherwise, returns false.

Methods in the Interface

Object top ( )

Precondition:

- Stack is not empty.

Postcondition:

- Get the most recently inserted item in the stack.

Methods in the Interface

Object topAndPop ( )

Precondition:

- Stack is not empty.

Postcondition:

- Removes the most recently added (top) element from a stack and returns it.

Method: doubleArray

- Internal method to extend the stack in case it is full.

Array Implementation of Stack

Before we start implementation: generic programming in Java

- Generics enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods.

Non generic class: Box

```
public class Box {
private Object object;
public void set(Object object)
{ this.object = object; }
public Object get() { return object; }
}
```

Generic class: Box

```
public class Box<T> {
// T stands for "Type"
private T t;
public void set(T t) { this.t = t; }
public T get() { return t; }
}
```

Invoking and Instantiating a Generic Type

- To reference the generic Box class from within your code, you must perform a *generic type invocation*, which replaces T with some concrete value, such as Integer:

```
Box<Integer> integerBox;
```

```
 1 package weiss.nonstandard;
 2
 3 // ArrayStack class
 4 //
 5 // CONSTRUCTION: with no initializer
 6 //
 7 // ******************PUBLIC OPERATIONS*********************
 8 // void push( x )         --> Insert x
 9 // void pop( )            --> Remove most recently inserted item
10 // AnyType top( )         --> Return most recently inserted item
11 // AnyType topAndPop( )   --> Return and remove most recent item
12 // boolean isEmpty( )     --> Return true if empty; else false
13 // void makeEmpty( )      --> Remove all items
14 // ******************ERRORS********************************
15 // top, pop, or topAndPop on empty stack
16
17 public class ArrayStack<AnyType> implements Stack<AnyType>
18 {
19     public ArrayStack( )
20       { /* Figure 16.3 */ }
21
22     public boolean isEmpty( )
23       { /* Figure 16.4 */ }
24     public void makeEmpty( )
25       { /* Figure 16.4 */ }
26     public Object top( )
27       { /* Figure 16.6 */ }
28     public void pop( )
29       { /* Figure 16.6 */ }
30     public AnyType topAndPop( )
31       { /* Figure 16.7 */ }
32     public void push( AnyType x )
33       { /* Figure 16.5 */ }
34
35     private void doubleArray( )
36       { /* Implementation in online code  */ }
37
38     private AnyType [ ] theArray;
39     private int         topOfStack;
40
41     private static final int DEFAULT_CAPACITY = 10;
42 }
```

figure 16.2

Skeleton for the array-based stack class

AnyType: Generic programming

Array that contains the elements

Index of top of stack

Constructor

figure 16.3

The zero-parameter constructor for the ArrayStack class

```
1      /**
2       * Construct the stack.
3       */
4      public ArrayStack( )
5      {
6          theArray = (AnyType []) new Object[ DEFAULT_CAPACITY ];
7          topOfStack = -1;
8      }
```

isEmpty() and makeEmpty()

figure 16.4

The `isEmpty` and `makeEmpty` routines for the `ArrayStack` class

```
 1    /**
 2     * Test if the stack is logically empty.
 3     * @return true if empty, false otherwise.
 4     */
 5    public boolean isEmpty( )
 6    {
 7        return topOfStack == -1;
 8    }
 9
10    /**
11     * Make the stack logically empty.
12     */
13    public void makeEmpty( )
14    {
15        topOfStack = -1;
16    }
```

Push( )

figure 16.5

The push method for the ArrayStack class

```
1   /**
2    * Insert a new item into the stack.
3    * @param x the item to insert.
4    */
5   public void push( AnyType x )
6   {
7       if( topOfStack + 1 == theArray.length )
8           doubleArray( );
9       theArray[ ++topOfStack ] = x;
10  }
```

Top( ) and Pop( )

```
1   /**
2    * Get the most recently inserted item in the stack.
3    * Does not alter the stack.
4    * @return the most recently inserted item in the stack.
5    * @throws UnderflowException if the stack is empty.
6    */
7   public AnyType top( )
8   {
9       if( isEmpty( ) )
10          throw new UnderflowException( "ArrayStack top" );
11      return theArray[ topOfStack ];
12  }
13
14  /**
15   * Remove the most recently inserted item from the stack.
16   * @throws UnderflowException if the stack is empty.
17   */
18  public void pop( )
19  {
20      if( isEmpty( ) )
21          throw new UnderflowException( "ArrayStack pop" );
22      topOfStack--;
23  }
```

figure 16.6

The `top` and `pop` methods for the `ArrayStack` class

topAndPop( )

```
1   /**
2    * Return and remove the most recently inserted item
3    * from the stack.
4    * @return the most recently inserted item in the stack.
5    * @throws Underflow if the stack is empty.
6    */
7   public AnyType topAndPop( )
8   {
9       if( isEmpty( ) )
10          throw new UnderflowException( "ArrayStack topAndPop" );
11      return theArray[ topOfStack-- ];
12  }
```

figure 16.7

The topAndPop method for the ArrayStack class

doubleArray( )

```
private void doubleArray( )
  {
     AnyType [ ] newArray;

     newArray = (AnyType []) new Object[ theArray.length * 2 ];
     for( int i = 0; i < theArray.length; i++ )
        newArray[ i ] = theArray[ i ];
     theArray = newArray;
  }
```

Linked List Implementation of Stack

Linked List concept

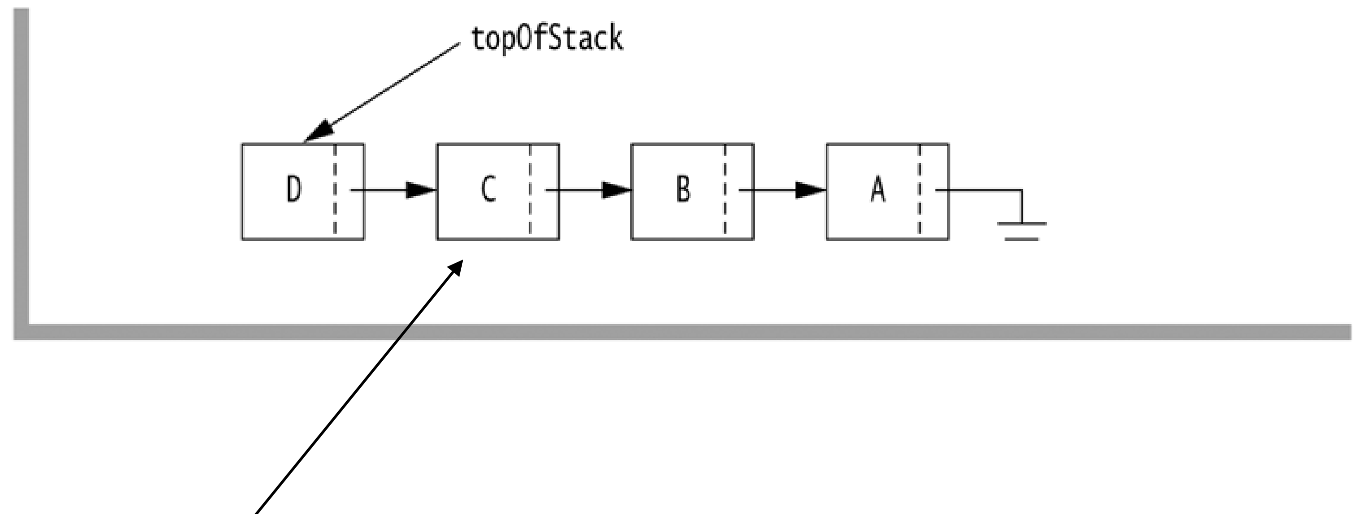

figure 16.18

Linked list implementation of the Stack class

This is the node containing two elements: the pointer/reference to next node and the object

```
 1 package weiss.nonstandard;
 2
 3 // ListStack class
 4 //
 5 // CONSTRUCTION: with no initializer
 6 //
 7 // ******************PUBLIC OPERATIONS*********************
 8 // void push( x )         --> Insert x
 9 // void pop( )            --> Remove most recently inserted item
10 // AnyType top( )         --> Return most recently inserted item
11 // AnyType topAndPop( )   --> Return and remove most recent item
12 // boolean isEmpty( )     --> Return true if empty; else false
13 // void makeEmpty( )      --> Remove all items
14 // ******************ERRORS********************************
15 // top, pop, or topAndPop on empty stack
16
17 public class ListStack<AnyType> implements Stack<AnyType>
18 {
19     public boolean isEmpty( )
20       { return topOfStack == null; }
21     public void makeEmpty( )
22       { topOfStack = null; }
23
24     public void push( AnyType x )
25       { /* Figure 16.20 */ }
26     public void pop( )
27       { /* Figure 16.20 */ }
28     public AnyType top( )
29       { /* Figure 16.21 */ }
30     public AnyType topAndPop( )
31       { /* Figure 16.21 */ }
32
33     private ListNode<AnyType> topOfStack = null;
34 }
35
36 // Basic node stored in a linked list.
37 // Note that this class is not accessible outside
38 // of package weiss.nonstandard
39 class ListNode<AnyType>
40 {
41     public ListNode( AnyType theElement )
42       { this( theElement, null ); }
43
44     public ListNode( AnyType theElement, ListNode<AnyType> n )
45       { element = theElement; next = n; }
46
47     public AnyType   element;
48     public ListNode next;
49 }
```

figure 16.19

Skeleton for linked list-based stack class

Reference to top of stack

Element of the node

Reference to next node

Push( ) and Pop( )

figure 16.20

The push and pop routines for the ListStack class

```
1     /**
2      * Insert a new item into the stack.
3      * @param x the item to insert.
4      */
5     public void push( AnyType x )
6     {
7         topOfStack = new ListNode<AnyType>( x, topOfStack );
8     }
9
10    /**
11     * Remove the most recently inserted item from the stack.
12     * @throws UnderflowException if the stack is empty.
13     */
14    public void pop( )
15    {
16        if( isEmpty( ) )
17            throw new UnderflowException( "ListStack pop" );
18        topOfStack = topOfStack.next;
19    }
```

Top( ) and topAndPop( )

```
1      /**
2       * Get the most recently inserted item in the stack.
3       * Does not alter the stack.
4       * @return the most recently inserted item in the stack.
5       * @throws UnderflowException if the stack is empty.
6       */
7      public AnyType top( )
8      {
9          if( isEmpty( ) )
10             throw new UnderflowException( "ListStack top" );
11         return topOfStack.element;
12     }
13
14     /**
15      * Return and remove the most recently inserted item
16      * from the stack.
17      * @return the most recently inserted item in the stack.
18      * @throws UnderflowException if the stack is empty.
19      */
20     public AnyType topAndPop( )
21     {
22         if( isEmpty( ) )
23             throw new UnderflowException( "ListStack topAndPop" );
24
25         AnyType topItem = topOfStack.element;
26         topOfStack = topOfStack.next;
27         return topItem;
28     }
```

figure 16.21

The `top` and `topAndPop` routines for the `ListStack` class

Stack: An Implementation with ArrayList

figure 16.28

A simplified Collections-style `Stack` class, based on the `ArrayList` class

```
 1 package weiss.util;
 2
 3 /**
 4  * Stack class. Unlike java.util.Stack, this is not extended from
 5  * Vector. This is the minimum respectable set of operations.
 6  */
 7 public class Stack<AnyType> implements java.io.Serializable
 8 {
 9     public Stack( )
10     {
11         items = new ArrayList<AnyType>( );
12     }
13
14     public AnyType push( AnyType x )
15     {
16         items.add( x );
17         return x;
18     }
19
20     public AnyType pop( )
21     {
22         if( isEmpty( ) )
23             throw new EmptyStackException( );
24         return items.remove( items.size( ) - 1 );
25     }
26
27     public AnyType peek( )
28     {
29         if( isEmpty( ) )
30             throw new EmptyStackException( );
31         return items.get( items.size( ) - 1 );
32     }
33
34     public boolean isEmpty( )
35     {
36         return size( ) == 0;
37     }
38
39     public int size( )
40     {
41         return items.size( );
42     }
43
44     public void clear( )
45     {
46         items.clear( );
47     }
48
49     private ArrayList<AnyType> items;
50 }
```

Running times of the two implementations

- Both the array and linked list versions run in constant time per operation.
- The array version is likely to be faster if an accurate estimation of capacity is performed.
- If an additional constructor is provided to specify the initial capacity and the estimate is correct, no doubling is performed.

Readings

- Book
  - Chapter 16

Lab Exercises

- Add to the ADT for both implementations that we have defined the following methods:
  - ShowElements: shows all the elements in the stack
  - ShowInverse: show the elements in inverse order.
  - New constructor which specifies size of stack (for the array implementation) as parameter.
  - Clone: replicate a stack in another stack.
  - Swap: exchange the two topmost items on the stack.
- Test all these in a testing class