UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

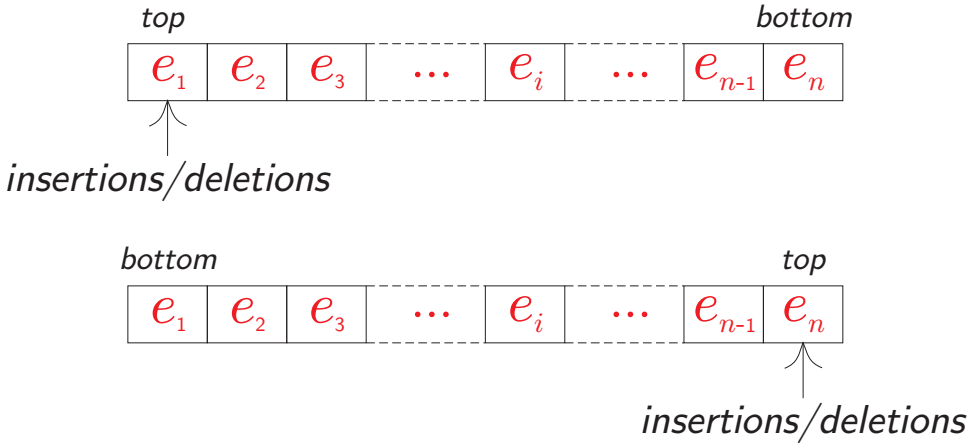
Sesión 5
Stack Data Structure

Yoan Pinzón

Table of Content Session 5

Stacks Data Structure

A **stack** is a linear list in which insertions (also called additions) and deletions take place at the *same* end. This end is called the **top**. The other end is called the **bottom**.



In other words, a stack is a LIFO (last-in-first out) list. Lists of this type appear frequently in computing.

The Abstract Data Type

AbstractDataType Stack

{

instances: linear list of elements; one end called the *bottom*; the other is the *top*

operations:

`isEmpty()`: return `true` if the stack is empty, `false` otherwise

`peek()`: return top element

`push(`x`)`: add element x at the top

`pop()`: remove the top element and return it

}

Interface Definition of Stack

```
1 package unal.datastructures;
2
3 public interface Stack<T>
4 {
5    boolean isEmpty( );
6    T peek( );
7    void push( T theObject );
8    T pop( );
9 }
```


Observations

- Stack is a specialized or restricted version of a more general data object linear list.
- Every instance of the data object stack is also an instance of the data object linear list. Moreover, all the stack operations can be performed as linear list operations.
- As a result of these observations, we will defined the stack class as a class which inherit all the data member and function from the linear list class.
- We will also use two methods of representation. namely, array-based and linked representation.

Stack Data Structure

Array-based Representation

- Implementation using inheritance
- Customised Implementation

Implementing Array Stack using Inheritance

Basic design decision:

designate the left end of the list as the bottom and the right end as the top

Implementing Array Stack using Inheritance

```
3  package unal.datastructures;

5  import java.util.*;

7  public class DerivedArrayStack<T> extends ArrayLinearList<T>
8                               implements Stack<T>
9  {
10    // constructors
11    /** create a stack with the given initial capacity */
12    public DerivedArrayStack( int initialCapacity )
13    {
14       super( initialCapacity );
15    }

17    /** create a stack with initial capacity 10 */
18    public DerivedArrayStack( )
19    {
20       this( 10 );
21    }
```



```
23    // methods
24    /** @return true iff stack is empty */
25    public boolean isEmpty( )
26    {
27       return super.isEmpty( );
28    }

30    /** @return top element of stack
31     * @throws EmptyStackException when the stack is empty */
32    public T peek( )
33    {
34       if( isEmpty( ) )
35          throw new EmptyStackException( );
36       return get( size( ) - 1 );
37    }

39    /** add theElement to the top of the stack */
40    public void push( T theElement )
41    {
42       add( size( ), theElement );
43    }
```

```
45      /** remove top element of stack and return it
46       * @throws EmptyStackException when the stack is empty */
47      public T pop( )
48      {
49         if( isEmpty( ) )
50            throw new EmptyStackException( );
51         return remove( size( ) - 1 );
52      }

54      /** test program */
55      public static void main( String[] args )
56      {
57         int x;
58         DerivedArrayStack<Integer> s = new DerivedArrayStack<>( 3 );

60         // add a few elements
61         s.push( new Integer( 1 ) );
62         s.push( new Integer( 2 ) );
63         s.push( new Integer( 3 ) );
64         s.push( new Integer( 4 ) );
```



```
66         // delete all elements
67         while( !s.isEmpty( ) )
68         {
69            System.out.println( "Top element is " + s.peek( ) );
70            System.out.println( "Removed the element " + s.pop( ) );
71         }
72      }
73   }
```

Compiling `DerivedArrayStack.java`

```
C:\2016699\code> javac unal\datastructures\DerivedArrayStack.java ↵
C:\2016699\code> java unal.datastructures.DerivedArrayStack ↵
Top element is 4
Removed the element 4
Top element is 3
Removed the element 3
Top element is 2
Removed the element 2
Top element is 1
Removed the element 1
```


Time Complexity of Operations

First Constructor	:	$O(\texttt{initialCapacity})$
Second Constructor	:	$\Theta(1)$
Other operations*	:	$\Theta(1)$

* The complexity of `push` is $\Theta(1)$ except when the addition of an element requires us to increase the capacity of the stack. In this latter case the complexity is $O(\texttt{capacity})$.

Implementing Array Stack as a Base Class

```
package unal.datastructures;

import java.util.*;

public class ArrayStack<T> implements Stack<T>
{
   // fields
   int top;   // current top of stack
   T[] stack; // element array

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

   // methods
   public boolean isEmpty( ) { /* ... */ }
```



```
   public T peek( ) { /* ... */ }
   public void push( T theElement ) { /* ... */ }
   public T pop( ) { /* ... */ }
   public static void main( String[] args ) { /* ... */ }
}
```

constructors

```
14    /** create a stack with the given initial capacity
15     * @throws IllegalArgumentException when initialCapacity < 1 */
16    @SuppressWarnings( "unchecked" )
17    public ArrayStack( int initialCapacity )
18    {
19       if( initialCapacity < 1 )
20          throw new IllegalArgumentException
21                ( "initialCapacity must be >= 1" );
22       stack = ( T[] ) new Object[ initialCapacity ];
23       top = -1;
24    }

26    /** create a stack with initial capacity 10 */
27    public ArrayStack( )
28    {
29       this( 10 );
30    }
```


isEmpty

```
33    /** @return true iff stack is empty */
34    public boolean isEmpty( )
35    {
36       return top == -1;
37    }
```

size

```
39    /** @return top element of stack
40     * @throws EmptyStackException when the stack is empty */
41    public T peek( )
42    {
43       if( isEmpty( ) )
44          throw new EmptyStackException( );
45       return stack[ top ];
46    }
```


push

```
48    /** add theElement to the top of the stack */
49    @SuppressWarnings( "unchecked" )
50    public void push( T theElement )
51    {
52       // increase array size if necessary
53       if( top == stack.length - 1 )
54       {
55          T[] old = stack;
56          stack = ( T[] ) new Object[ 2 * stack.length ];
57          System.arraycopy( old, 0, stack, 0, old.length );
58       }

60       // put theElement at the top of the stack
61       stack[ ++top ] = theElement;
62    }
```

pop

```
64  /** remove top element of stack and return it
65   * @throws EmptyStackException when the stack is empty */
66  public T pop( )
67  {
68     if( isEmpty( ) )
69        throw new EmptyStackException( );
70     T topElement = stack[ top ];
71     stack[ top-- ] = null; // enable garbage collection
72     return topElement;
73  }
```


main

```
75  /** test program */
76  public static void main( String[] args )
77  {
78     int x;
79     ArrayStack<Integer> s = new ArrayStack<>( 3 );

81     // add a few elements
82     s.push( new Integer( 1 ) );
83     s.push( new Integer( 2 ) );
84     s.push( new Integer( 3 ) );
85     s.push( new Integer( 4 ) );

87     // delete all elements
88     while( !s.isEmpty( ) )
89     {
90        System.out.println( "Top␣element␣is␣" + s.peek( ) );
91        System.out.println( "Removed␣the␣element␣" + s.pop( ) );
92     }
93  }
```

Compiling `ArrayStack.java`

```
C:\2016699\code> javac unal\datastructures\ArrayStack.java ↵
C:\2016699\code> java unal.datastructures.ArrayStack ↵
Top element is 4
Removed the element 4
Top element is 3
Removed the element 3
Top element is 2
Removed the element 2
Top element is 1
Removed the element 1
```


Stack Data Structure

Linked Representation

- Implementation using inheritance
- Customised Implementation

In both cases, we have to decide which end of the chain will be the top of the stack and which the bottom.

Implementing Linked Stack Using Inheritance

```
3  package unal.datastructures;

5  import java.util.*;

7  public class DerivedLinkedStack<T> extends Chain<T> ↙
     ↳ implements Stack<T>
8  {
9     // constructor
10    public DerivedLinkedStack( )
11    {
12       super( );
13    }

15    // methods
16    /** @return true iff stack is empty */
17    public boolean isEmpty( )
18    {
19       return super.isEmpty( );
20    }
```



```
22    /** @return top element of stack
23     * @throws EmptyStackException when the stack is empty */
24    public T peek( )
25    {
26       if ( isEmpty( ) ) throw new EmptyStackException( );
27       return get( 0 );
28    }

30    /** add theElement to the top of the stack */
31    public void push( T theElement )
32    {
33       add( 0, theElement );
34    }

36    /** remove top element of stack and return it
37     * @throws EmptyStackException when the stack is empty */
38    public T pop( )
39    {
40       if ( isEmpty( ) ) throw new EmptyStackException( );
41       return remove( 0 );
42    }
```

```
44    /** test program */
45    public static void main ( String[] args )
46    {
47       int x;
48       DerivedLinkedStack<Integer> s = new DerivedLinkedStack<>( );

50       // add a few elements
51       s.push( new Integer( 1 ) );
52       s.push( new Integer( 2 ) );
53       s.push( new Integer( 3 ) );
54       s.push( new Integer( 4 ) );

56       // delete all elements
57       while ( !s.isEmpty( ) )
58       {
59          System.out.println( "Top␣element␣is␣" + s.peek( ) );
60          System.out.println( "Removed␣the␣element␣" + s.pop( ) );
61       }
62    }
63 }
```


Compiling `DerivedLinkedStack.java`

```
C:\2016699\code> javac unal\datastructures\DerivedLinkedStack.java ↵
C:\2016699\code> java unal.datastructures.DerivedLinkedStack ↵
Top element is 4
Removed the element 4
Top element is 3
Removed the element 3
Top element is 2
Removed the element 2
Top element is 1
Removed the element 1
```

Implementing Linked Stack as a Base Class

```
package unal.datastructures;

import java.util.*;

public class LinkedStack<T> implements Stack<T>
{
   // fields
   protected ChainNode<T> topNode;

   // constructor
   public LinkedStack( ) { /* ... */ }

   // methods
   public boolean isEmpty( ) { /* ... */ }
   public T peek( ) { /* ... */ }
   public void push( T theElement ) { /* ... */ }
```



```
   public T pop( ) { /* ... */ }
   public static void main( String[] args ) { /* ... */ }
}
```

constructor

```
13  public LinkedStack( )
14  {
15      topNode = null;
16  }
```


isEmpty

```
19  /** @return true iff stack is empty */
20  public boolean isEmpty( )
21  {
22     return topNode == null;
23  }
```

peek

```
25    /** @return top element of stack
26     * @throws EmptyStackException when the stack is empty */
27    public T peek( )
28    {
29       if( isEmpty( ) ) throw new EmptyStackException( );
30       return topNode.element;
31    }
```


push

```
33    /** add theElement to the top of the stack */
34    public void push( T theElement )
35    {
36       topNode = new ChainNode<T>( theElement, topNode );
37    }
```

pop

```
39  /** remove top element of stack and return it
40   * @throws EmptyStackException when the stack is empty */
41  public T pop( )
42  {
43     if( isEmpty( ) ) throw new EmptyStackException( );
44     T topElement = topNode.element;
45     topNode = topNode.next;
46     return topElement;
47  }
```


main

```
49  /** test program */
50  public static void main( String[] args )
51  {
52     LinkedStack<Integer> s = new LinkedStack<>( );

54     // add a few elements
55     s.push( new Integer( 1 ) );
56     s.push( new Integer( 2 ) );
57     s.push( new Integer( 3 ) );
58     s.push( new Integer( 4 ) );

60     // delete all elements
61     while ( !s.isEmpty( ) )
62     {
63        System.out.println( "Top␣element␣is␣" + s.peek( ) );
64        System.out.println( "Removed␣the␣element␣" + s.pop( ) );
65     }
66  }
```

Compiling `LinkedStack.java`

```
C:\2016699\code> javac unal\datastructures\LinkedStack.java ↵
C:\2016699\code> java unal.datastructures.LinkedStack ↵
Top element is 4
Removed the element 4
Top element is 3
Removed the element 3
Top element is 2
Removed the element 2
Top element is 1
Removed the element 1
```


Time Complexity of Operations

Constructors	:	$\Theta(1)$
Other operations	:	$\Theta(1)$