```
import java.util.*;
public class BinarySearch
{
    public static final int NOT_FOUND = -1;

    /**
     * Performs the standard binary search
     * using two comparisons per level.
     * @return index where item is found, or NOT_FOUND.
     */
    public static int binarySearch( Comparable [ ] a, Comparable x )
    {
        int low = 0;
        int high = a.length - 1;
        int mid;

        while( low <= high )
        {
            mid = ( low + high ) / 2;

            if( a[ mid ].compareTo( x ) < 0 )
                low = mid + 1;
            else if( a[ mid ].compareTo( x ) > 0 )
                high = mid - 1;
            else
                return mid;
        }

        return NOT_FOUND;     // NOT_FOUND = -1
    }

    // just a linear search method
    public static int search ( int [] a, int x)
    {

      int NOT_FOUND=-1;

      for (int i=0; i<a.length;i++)
      {
        if(a[i]==x)
         return i;
      }

         return NOT_FOUND;

    }
```

```
 public static int binarySearch( int [ ] a, int x )
    {
        int NOT_FOUND=-1;
        int low = 0;
        int high = a.length - 1;
        int mid;

        while( low <= high )
        {
            mid = ( low + high ) / 2;

            if ( x> a[ mid ]  )
                low = mid + 1;  // leta i högra halvan
            else if(x< a[ mid ] )
                high = mid - 1; // leta i vänstra halvan
            else
                return mid;
        }

        return NOT_FOUND;     // NOT_FOUND = -1
    }


    // Test program
    public static void main( String [ ] args )
    {

        Scanner scan =new Scanner( System.in);

        Comparable [ ] names = new String [ 3 ];

        for( int i = 0; i < 3; i++ )
            products[ i ] =  scan.next();


    System.out.println( "Found at " + binarySearch( products,
"Kalle"));

    }
}
```

```
public class ArrayStack<AnyType>

{

    private AnyType [ ] theArray;
    private int         topOfStack;

    private static final int DEFAULT_CAPACITY = 10;


    /**
     * Construct the stack.
     */
    public ArrayStack( )
    {
        theArray = (AnyType []) new Object[ DEFAULT_CAPACITY ];
        topOfStack = -1;
    }

    /**
     * Test if the stack is logically empty.
     * @return true if empty, false otherwise.
     */
    public boolean isEmpty( )
    {
        return topOfStack == -1;
    }

    /**
     * Make the stack logically empty.
     */
    public void makeEmpty( )
    {
        topOfStack = -1;
    }

    /**
     * Get the most recently inserted item in the stack.
     * Does not alter the stack.
     * @return the most recently inserted item in the stack.
     * @throws UnderflowException if the stack is empty.
     */
    public AnyType top( )
    {
        if( isEmpty( ) )
            throw new UnderflowException( "ArrayStack top" );    //
UnderflowException
        return theArray[ topOfStack ];
    }

    /**
     * Remove the most recently inserted item from the stack.
     * @throws UnderflowException if the stack is empty.
     */
    public void pop( )
    {
```

```
        if( isEmpty( ) )
            throw new UnderflowException( "ArrayStack pop" );
        topOfStack--;
    }

    /**
     * Return and remove the most recently inserted item
     * from the stack.
     * @return the most recently inserted item in the stack.
     * @throws Underflow if the stack is empty.
     */
    public AnyType topAndPop( )
    {
        if( isEmpty( ) )
            throw new UnderflowException( "ArrayStack topAndPop" );
        return theArray[ topOfStack-- ];
    }

    /**
     * Insert a new item into the stack.
     * @param x the item to insert.
     */
    public void push( AnyType x )
    {
        if( topOfStack + 1 == theArray.length )
            doubleArray( );
        theArray[ ++topOfStack ] = x;
    }

    /**
     * Internal method to extend theArray.
     */
    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;
    }



public static void main (String [] arg)
{
   ArrayStack s=new ArrayStack();
   try{
  s.pop();

   }
   catch( UnderflowException e)
   {
     System.out.println(e);
     System.out.println("Skack empty");
```

```
      }

    }



}
```

```
public class ArrayQueue<AnyType>
{
    private AnyType [ ] theArray;
    private int         currentSize;
    private int         front;
    private int         back;

    private static final int DEFAULT_CAPACITY = 10;


/**
     * Construct the queue.
     */
    public ArrayQueue( )
    {
        theArray = (AnyType []) new Object[ DEFAULT_CAPACITY ];
        makeEmpty( );
    }

    /**
     * Test if the queue is logically empty.
     * @return true if empty, false otherwise.
     */
    public boolean isEmpty( )
    {
        return currentSize == 0;
    }

    /**
     * Make the queue logically empty.
     */
    public void makeEmpty( )
    {
        currentSize = 0;
        front = 0;
        back = -1;
    }

    /**
     * Return and remove the least recently inserted item
     * from the queue.
     * @return the least recently inserted item in the queue.
     * @throws UnderflowException if the queue is empty.
```

```
     */
    public AnyType dequeue( )
    {
        if( isEmpty( ) )
            throw new UnderflowException( "ArrayQueue dequeue" );
        currentSize--;

        AnyType returnValue = theArray[ front ];
        front = increment( front );
        return returnValue;
    }

    /**
     * Get the least recently inserted item in the queue.
     * Does not alter the queue.
     * @return the least recently inserted item in the queue.
     * @throws UnderflowException if the queue is empty.
     */
    public AnyType getFront( )
    {
        if( isEmpty( ) )
           throw new UnderflowException( "ArrayQueue getFront" );
        return theArray[ front ];
    }

    /**
     * Insert a new item into the queue.
     * @param x the item to insert.
     */
    public void enqueue( AnyType x )
    {
        if( currentSize == theArray.length )
            doubleQueue( );
        back = increment( back );
       theArray[ back ] = x;
        currentSize++;
    }

    /**
     * Internal method to increment with wraparound.
     * @param x any index in theArray's range.
     * @return x+1, or 0 if x is at the end of theArray.
     */
    private int increment( int x )
    {
        if( ++x == theArray.length )
            x = 0;
        return x;
    }

    /**
     * Internal method to expand theArray.
     */
    private void doubleQueue( )
    {
        AnyType [ ] newArray;
```

```
        newArray = (AnyType []) new Object[ theArray.length * 2 ];

            // Copy elements that are logically in the queue
        for( int i = 0; i < currentSize; i++, front = increment(
front ) )
            newArray[ i ] = theArray[ front ];

        theArray = newArray;
        front = 0;
        back = currentSize - 1;
    }


}
```

```
public class ListStack<AnyType>

{
     private ListNode<AnyType> topOfStack;

     /**
      * Construct the stack.
      */
    public ListStack( )
    {
        topOfStack = null;
    }

    /**
     * Test if the stack is logically empty.
     * @return true if empty, false otherwise.
     */
    public boolean isEmpty( )
    {
       return topOfStack == null;
    }

    /**
     * Make the stack logically empty.
     */
    public void makeEmpty( )
    {
        topOfStack = null;
    }

    /**
     * Insert a new item into the stack.
     * @param x the item to insert.
     */
```

```
    public void push( AnyType x )
    {
        topOfStack = new ListNode<AnyType>( x, topOfStack );
    }

    /**
     * Remove the most recently inserted item from the stack.
     * @throws UnderflowException if the stack is empty.
     */
    public void pop( )
    {
        if( isEmpty( ) )
            throw new UnderflowException( "ListStack pop" );
        topOfStack = topOfStack.next;
    }

    /**
     * Get the most recently inserted item in the stack.
     * Does not alter the stack.
     * @return the most recently inserted item in the stack.
     * @throws UnderflowException if the stack is empty.
     */
    public AnyType top( )
    {
        if( isEmpty( ) )
            throw new UnderflowException( "ListStack top" );
        return topOfStack.element;
    }

    /**
     * Return and remove the most recently inserted item
     * from the stack.
     * @return the most recently inserted item in the stack.
     * @throws UnderflowException if the stack is empty.
     */
    public AnyType topAndPop( )
    {
        if( isEmpty( ) )
            throw new UnderflowException( "ListStack
topAndPop" );

        AnyType topItem = topOfStack.element;
        topOfStack = topOfStack.next;
        return topItem;
    }


}
```

```
public class ListQueue<AnyType>
{

    private ListNode<AnyType> front;
    private ListNode<AnyType> back;


     /**
      * Construct the queue.
      */
    public ListQueue( )
    {
        front = back = null;
    }

    /**
     * Test if the queue is logically empty.
     * @return true if empty, false otherwise.
     */
    public boolean isEmpty( )
    {
        return front == null;
    }

   /**
     * Insert a new item into the queue.
     * @param x the item to insert.
     */
    public void enqueue( AnyType x )
    {
        if( isEmpty( ) )     // Make queue of one element
            back = front = new ListNode<AnyType>( x );
        else                 // Regular case
            back = back.next = new ListNode<AnyType>( x );
   }

    /**
     * Return and remove the least recently inserted item
     * from the queue.
     * @return the least recently inserted item in the queue.
     * @throws UnderflowException if the queue is empty.
     */
    public AnyType dequeue( )
    {
        if( isEmpty( ) )
            throw new UnderflowException( "ListQueue dequeue"
);

        AnyType returnValue = front.element;
        front = front.next;
```

```
        return returnValue;
    }

    /**
     * Get the least recently inserted item in the queue.
     * Does not alter the queue.
     * @return the least recently inserted item in the queue.
     * @throws UnderflowException if the queue is empty.
     */
    public AnyType getFront( )
    {
        if( isEmpty( ) )
            throw new UnderflowException( "ListQueue getFront"
);
        return front.element;
    }

    /**
    * Make the queue logically empty.
    */
    public void makeEmpty( )
    {
        front = null;
        back = null;
    }

    }
```