Problem 1: [20 points]

Finish the definition of the method power below. It should return its first argument raised to the power of its second. For example, if passed 2 and 4 it should return 16 since $2^4 = 16$. (Or, put another way, $2*2*2*2 = 16$.) For full credit, define the method using recursion.

```java
public int power(int base, int exp) {
```
Problem 2: [28 points]

Assume we have a stack of Integers that we keep ordered, from smallest at the top to largest at the bottom. Below, define a method called `orderedInsert` that could take such an ordered stack and add a new value to the stack such that the stack remains ordered. Your method should take a Stack of Integers and an int as inputs, and modify the input stack such that it contains the new value at the appropriate position. For full credit, your solution should only use methods from the Stack class, not any of the methods it inherits from Vector. (See documentation at the end of the exam.)
Problem 3: [28 points]

```java
public boolean binarySearch(List<Integer> nums, int value, int low, int high) {
    if (high == low+1) {
        return (nums.get(low) == value);
    }
    else {
        int midIndex = (low+high)/2;
        if (value < nums.get(midIndex)) {
            return binarySearch(nums, value, low, midIndex);
        }
        else {
            return binarySearch(nums, value, midIndex, high);
        }
    }
}
```

a) Our Binary Search code is shown above, but edited slightly so that it works on Lists rather than arrays. What is the complexity of this method when passed an ArrayList containing N nodes? You don’t need to define a precise \( T(n) \) function — you can just skip to the Big-O term, but for full credit you need to explain your answer.

b) What is the complexity of this method when passed a LinkedList containing N nodes? Here again you can just skip to the Big-O term, but for full credit you need to explain your answer.
Problem 4: [24 points]

Define the method `buildEvensQueue` that creates and returns a queue full of even numbers. It should take a single argument, the largest even number to be placed in the queue, and return a queue containing 0 through that value, with 0 at the head of the queue. For example `buildEvens(10)` would return a queue containing [0,2,4,6,8,10]. For full credit, define the method recursively, and don’t use methods other than those defined in the Queue interface.
Overview Package Class Use Tree Deprecated Index Help
Java™ Platform Standard Ed. 6
PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAILS: FIELD | CONSTR | METHOD
Prepared by javadoc - 31 Oct 2007

package java.util;
class Stack{
    extends Vector

    The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.

    When a stack is first created, it contains no items.

    A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations, which should be used in preference to this class. For example:

        Deque<Integer> stack = new ArrayDeque<Integer>();

    Since:
        JDK1.0
    See Also:
        Serialized Form

Field Summary

<table>
<thead>
<tr>
<th>Fields inherited from class java.util.Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>capacityIncrement, elementCount, elementData</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fields inherited from class java.util.AbstractList</th>
</tr>
</thead>
<tbody>
<tr>
<td>modCount</td>
</tr>
</tbody>
</table>

Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean empty()</td>
<td>Tests if this stack is empty.</td>
</tr>
<tr>
<td>E peek()</td>
<td>Looks at the object at the top of this stack without removing it from the stack.</td>
</tr>
<tr>
<td>E pop()</td>
<td>Removes the object at the top of this stack and returns that object as the value of this function.</td>
</tr>
<tr>
<td>E push(E item)</td>
<td>Pushes an item onto the top of this stack.</td>
</tr>
<tr>
<td>int search(Object o)</td>
<td>Returns the 1-based position where an object is on this stack.</td>
</tr>
</tbody>
</table>

Methods inherited from class java.util.Vector

add, add, addAll, addAll, addElement, capacity, clear, clone, contains, containsAll, copyInto, elementAt, elements, ensureCapacity, equals, firstElement, get, hashCode, indexOf, indexOf, insertElementAt, isEmpty, lastElement, lastIndexOf, lastIndexOf, remove, remove, removeAll, removeAllElements, removeElement, removeElementAt, removeRange, retainAll, set, setElementAt, setSize, size, subList, toArray, toArrayList, toString, trimToSize

Methods inherited from class java.util.AbstractList

iterator, listIterator, listIterator

Methods inherited from class java.lang.Object

finalize, getClass, notify, notifyAll, wait, wait, wait

Constructor Summary

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack()</td>
<td>Creates an empty Stack.</td>
</tr>
</tbody>
</table>

Constructor Detail

Stack

public Stack()

Creates an empty Stack.
java.util

**Interface Queue<E>**

**Type Parameters:**
- E - the type of elements held in this collection

**All Superinterfaces:**
- Collection<E>, Iterable<E>

**All Known Subinterfaces:**
- BlockingDeque<E>, BlockingQueue<E>, Deque<E>

**All Known Implementing Classes:**
- AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, PriorityBlockingQueue, PriorityQueue, SynchronousQueue

public interface Queue<E>

extends Collection<E>

A collection designed for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations. Each of these basic methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation). The latter form of the insert operation is designed specifically for use with capacity-restricted Queue implementations; in most implementations, insert operations cannot fail.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.</td>
</tr>
<tr>
<td>offer</td>
<td>Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.</td>
</tr>
<tr>
<td>remove</td>
<td>Removes and removes the head of this queue, or returns null if this queue is empty.</td>
</tr>
<tr>
<td>poll</td>
<td>Retrieves and returns the head of this queue, or returns null if this queue is empty.</td>
</tr>
</tbody>
</table>

Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LILO queues (or stacks) which order the elements LIPO (last-in-first-out). Whatever the ordering used, the head of the queue is that element which would be removed by a call to remove() or poll(). In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.

The offer method inserts an element if possible, otherwise returning false. This differs from the Collection.add method, which can fail to add an element only by throwing an unchecked exception. The offer method is designed for use when failure is a normal, rather than exceptional occurrence, for example, in fixed-capacity (or "bounded") queues.