1. (15 pts) Short questions (3 pts each)

a) What's the difference between the size and the capacity of a data container?
b) What is the disadvantage of using a linear array to implement a queue?
c) Why did we have a empty/dummy slot in a queue's circular array container?
d) If you're sorting a sequence of integers which are already sorted, which sorting algorithm (one of the 3 we have studied so far) would you use and why?
e) If you want to extract the k'th (1 <= k <= N) minimum from a sequence of unique integers, how would you do it?

2. (15 pts) Linked Lists

We have a circular doubly linked list, which uses a dummy header node.
It uses the same design used by the linked list discussed in class.

```
public class Node {                     public class List {
  int val;                                Node header;
  Node next;                              int size;
  Node prev;
                                          public List();
  Node(int v);                            public void sort();
  Node(int v, Node n, Node p);            // ... other methods
}                                       }
```

This version of List has a sort() method that sorts the elements in ascending order. You can choose to use any of the following sorting algorithms: bubble, insertion and selection. The List class uses a dummy header node.

```
class List {
  public List() {
    header = new Node(0);       // value in header node is irrelevant.
    header.next = header.prev = header;
    size = 0;
  }

  /**
   * Sorts the elements of the list in ascending order.
   * Postcondition: the list is now sorted in ascending order.
   */
  public void sort() {
    // TODO - write the sort method. You have to mention which algorithm
    // you're using to get marks for it.
  }
}
```

3. (40 pts) Queues

You have to implement a Queue that using a circular array.

```
public class Queue {
  // Initial capacity of the queue.
  public final int DEFAULT_CAPACITY = 10;
  // Create a queue of DEFAULT_CAPACITY capacity.
  public Queue() {
    queue = new int[DEFAULT_CAPACITY];
    size = front = rear = 0;
  }
  public void enqueue(int val); // add the element with value at end
  public int dequeue();         // removes and returns the front element
  public int getFront();        // get the item at front of queue
  public void remove(int val);  // remove element matching value

  private int[] queue;          // the array holding the queue elements
  private int size;             // number of elements in queue
  private int front;            // index of first queue element
  private int rear;             // index of where the next element would go

  // Note this is how we can use this Queue class
  public void main(String[] args) {
    Queue queue = new Queue();  // empty queue
    //  ________________________________
    //  |   |    |    |   |   |  |  |  | ...
    //  --------------------------------
    //    ^
    //  front
    //  rear
    queue.enqueue(9);
    queue.enqueue(5); queue.enqueue(15); queue.enqueue(-31);
    //  __________________________________
    //  | 9 | 5  | 15 | -31 |   |  |  |  | ...
    //  ----------------------------------
    //    ^                   ^
    //  front                rear
    queue.dequeue();
    //  __________________________________
    //  | 9 | 5  | 15 | -31 |   |  |  |  | ...
    //  ----------------------------------
    //        ^               ^
    //      front            rear
    queue.remove(15);
    //  __________________________________
    //  | 9 | 5  | -31| -31 |   |  |  |  | ...
    //  ----------------------------------
    //        ^          ^
    //      front       rear
  }
}
```

a) (15 pts) Write Queue.enqueue(int val) method.

```
/**
 * Add a value at the end of the queue. If the queue is at full capacity,
 * then resize it to double the capacity, and then add the new element.
 * Postcondition: Adjusts size and rear.
 * @param val the integer value to enqueue
 */
public void enqueue(int val) {
  // YOUR CODE GOES HERE
}
```

b) (10 pts) Write Queue.dequeue() method.

```
/**
 * Removes the item at the head of the queue, and returns that value. Throws
 * QueueException if the queue is empty.
 * Precondition: Queue must not be empty.
 * Postcondition: Adjusts size and front.
 * @return the value of the dequeued element.
 * @throws QueueException if the queue is empty.
 */
public int dequeue() throws QueueException {
  // YOUR CODE GOES HERE
}
```

c) (15 pts) Write Queue.remove(int val) method.

```
/**
 * Removes the queue entry matching the given value. Throws QueueException
 * if the value is not found in the queue.
 * Preconditions: there is a queue element with the given value.
 * Postcondition: adjust size and rear.
 * @param val the element with value to remove.
 * @throws QueueException if the value does not exist in the queue.
 */
public void remove(int val) throws QueueException {
  // YOUR CODE GOES HERE
}
```

4. (30 pts) Stacks

You're asked to write a stack of integers, and you have decided to create one based on (non-circular) doubly-linked lists to avoid problems of resizing arrays and so on. Instead of using a separate linked list class, the stack contains its own linked list using the Node class from Question 2. The linked list does not use a dummy head, so it's null when the stack is empty. When pushing a value, it's inserted in the beginning, so we don't need a separate pointer for the top of stack ("header" is the reference to the first node, which is the top of stack).
shown below.

```
public class Stack {
  public Stack() {
    header = null;
    size = 0;
  }

  public void push(int val);    // pushes the element with value
  public int pop ();            // removes and return top of the stack
  public int getTop();          // gets the item at the top of stack

  // Remove the stack element matching the value
  public bool remove (int val);

  private Node header;          // reference to the first node
  private int size;             // the number of items on stack

  // Note this is how we can use this Stack class
  public void main(String[] args) {
    Stack stack = new Stack();  // empty stack
    // header = null
    stack.push(5); stack.push(15); stack.push(-31); stack.push(1);
    //
    //   1  ->  -31  ->  15  -> 5
    //   ^
    // header
    stack.pop();
    //
    //  -31  ->  15  -> 5
    //   ^
    // header
    stack.remove(15);
    //
    //  -31  -> 5
    //   ^
    // header
    stack.pop(); stack.pop();
    // header = null
  }
}
```

a) (10 pts) Write Stack.push(int val) method.

```
/**
 * Pushes a value on the stack.
 * Postcondition: val is new top of stack; adjust size.
 * @param val the value to push.
 */
public void push(int val) {
  // YOUR CODE GOES HERE
}
```

b) (10 pts) Write Stack.pop() method.

```java
/**
 * Pops the top of stack, and return the value. Throws StackException if
 * the stack is empty.
 * Precondition: stack is non-empty.
 * Postcondition: adjust size.
 * @return the value that was popped.
 * @throws StackException if the stack is empty.
 */
public int pop() throws StackException {
  // YOUR CODE GOES HERE
}
```

c) (10 pts) Write Stack.remove(int val) method.

```java
/**
 * Removes the stack entry matching the given value. Throws StackException
 * if the value is not found in the stack.
 * Preconditions: there is a stack element with the given value.
 * Postcondition: adjust size.
 * @param val the element with value to remove.
 * @throws StackException if the value does not exist in the stack.
 */
public void remove(int val) throws StackException {
  // YOUR CODE GOES HERE
}
```

1. (10 pts) Short questions (2.5 pts each)

(a) What is the maximum of comparisons that you need to do to search for an element in an (unsorted) array of N elements?
(b) If instead the array was already sorted, how can you (easily) reduce that by at least 50%?
(c) Why is adding a node at the end of a list efficient if you had a tail reference?
(d) Why is removing the last element difficult even if you had a tail reference?

2. (30 pts) Array algorithms. You're given the following Array class.

```
public class Array {
  private Comparable[] data;    // the underlying built-in array container
  private int size;             // the number of elements being used

  /**
   * Sorts the data in this array using insertion sort.
   */
  void sort() {
    // TODO
  }

  /**
   * Reverses the order of the elements of this array. If the data
   * container has [ "a" "b" "c" "d" ], reversing it will result in
   * [ "d" "c" "b" "a" ].
   */
  void reverse() {
    // TODO
  }

  /**
   * Searches for the last occurrence of the given key and returns its index
   * if found, or -1 if not found. If the data contains [ "a" "b" "c" "b" ],
   * revSearch("a") will return 0, but revSearch("b") will return 3, not 1.
   * @param key the key being searched for.
   * @returns the index of its last occurrence in the array, or -1 if
   *          it's not in the array.
   */
   int revSearch(Comparable key) {
     // TODO
   }

   /**
    * Computes the index of the largest element (determined by the element's
    * compareTo method) in this array. If the data contains [ "a" "b" "c" ],
    * maxIndex() will return 2, the index of "c" which is the largest element.
    * @return the index of the largest element in this array.
    */
   int maxIndex() {
     // TODO
    }
}
```

(a) (10 pts) Write the Array.sort() method (MUST use insertion sort).
(b) (5 pts) Write the Array.reverse() method.
(c) (7.5 pts) Write the Array.revSearch() method.
(d) (7.5 pts) Write the Array.maxIndex() method.

3. (30 pts) Linked lists
Assume you have a singly linked linear list, with a reference to the head node. The Node class is shown below.

```
public class Node {
  public Object element;
  public Node next;
  public Node(Object e, Node n) {
    element = e;
    next = n;
  }

  /**
   * Make a list from the specified array, with the elements in the same
   * order as in the array. If the array contains [ "hello" "world" "!" ],
   * the list would contain the nodes [ "hello" "world" "!" ].
   * @param the array of objects to create the list from.
   * @return reference to the head node of the list.
   */
  public static Node makeList(Object[] a) {
    // TODO
  }

  /**
   * Returns the string representation of the specified list. If you have
   * a list [ "hello" "world" "!" ], returns the string "[ hello world ! ]".
   * @param list the reference to the head node.
   * @return a string representation of the specified list.
   */
  public static String makeString(Node list) {
    // TODO
  }

  /**
   * Removes the specified node in the list, and returns its element.
   * @param list reference to the head node in the list.
   * @param node reference to the node to remove from the list.
   * @return the element within the removed node.
   */
  public static Object removeNode(Node list, Node node) {
    // TODO
  }
}
```

(a) (10 pts) Write the static Node.makeList() method. Basically, iterate

```
  through the array, creating a node for each element and adding it to the
  end of the list.
(b) (10 pts) Write the static Node.makeString() method. Start with an empty
  String (or a StringBuffer), add "[ " to start with. Now iterate through
  the nodes, adding each to the string (or string buffer) and a space.
  Finally add "]", and return the string.
(c) (10 pts) Write the static Node.removeNode() method. You'll have to find
  reference to the node *just before* the node the remove, and then use
  the normal relinking to remove it. Make sure you take care of special
  cases (e.g., when list == node, ie., you're removing the first node)!

4. (30 pts) Queues using circular array container.
You're given the following Queue class using a built-in circular array as
the underlying container, but with the following differences from our
previous one:
 - this queue is not automatically resized when enqueuing a new element;
 - this queue only stores the index of the front element. The index to
   the next available location ("rear") is not essential - you can always
   calculate the "rear" given the front and index variables, along with
   the array capacity.

       0    1    2   3   4   5    6    7   8   9
     ---------------------------------------------
     | -4 | 17 |    |     |    |  5  | 7 | 9 | 15 |      size = 6
     ---------------------------------------------  capacity = 10
                                    ^--front

       0    1    2   3   4   5    6    7   8   9
     ---------------------------------------------
     | 5  | 7  | 9  | 15  | -4 | 17  |    |   |   |      size = 6
     ---------------------------------------------  capacity = 10
      ^--front

The queues shown above are "equal".
NOTE the absense of an explicit "rear" variable.

public class Queue {
  /** The size and index of front element. */
  int size, front;
  /** The data container. */
  Object[] queue;
  /**
   * Adds the specified element the end of this queue.
   * @param elem the element to add to the end of this queue.
   * @throws FullQueueException if the queue is at full capacity.
  void enqueue(Object elem) throws FullQueueException {
    // TODO
  }
  /**
   * Removes the front element of this queue, and returns it.
   * @return the object that was at the front of this queue.
   * @throws EmptyQueueException if the queue is empty.
  Object dequeue() throws EmptyQueueException {
    // TODO
  }
  /**
   * Checks if this queue is equal to the specified one.
   * @param o the other queue to check for equality.
   * @return true if this queue is equal to the specified one.
   */
  boolean equals(Object o) {
    // TODO
  }
}
```

(a) (10 pts) Write Queue.enqueue(). Note that the circular array container
    is not automatically resized when capacity is exhausted.
(b) (10 pts) Write Queue.dequeue().
(c) (10 pts) Write Queue.equals().

5. [BONUS] (5 pts) Complexity analysis of bubble sort.
The bubble sort algorithm is given below:

```
BUBBLESORT(A):

  FOR i <- 0 TO A.length-1 DO
      FOR j <- A.length-1 DOWNTO i+1 DO
          IF A[j] < A[j-1] THEN
              EXCHANGE A[j] <-> A[j-1]
```

To sort an array of `n' elements, how many operations does this algorithm perform? Assume that each assignment, addition, comparison, etc is a single operation. Your answer should be a function of `n'.

No. of Pages	**6**
No. of Questions	**5**

Department of Computer Science and Engineering
MIDTERM EXAMINATION FALL 2009

CSE220: Data Structures

Total Marks: 100 Time Allowed: 1 Hour 20 Minutes

- Answer all questions, in any order; number each answer.

Section	*Marks*
Q1. Short questions	15
Q3. Sorted Arrays	20
Q4. Queues	30
Q5. Stacks	30
Q6. Recursion	05
Total	**100**

- You must write your name and student ID on the script and on the question paper

Name : ___________________________

ID : ___________________________

- Write all answers, including scratch notes, in the answer script
- You must turn in your question paper

1. (15 pts) Short questions

(a) [2.5 pts] Why did we use a circular array when implementing an array-based queue?
(b) [2.5 pts] If you are to implement a LIFO stack with a singly-linked circular list with a tail reference, where would you put the new element that's being pushed (and why)?
(c) [2.5 pts] What is the advantage of using a dummy node (head or tail)?
(d) [2.5 pts] How do you iterate through a tail-referenced singly-linked circular list? Show code.
(e) [5 pts] To find an element in an array using linear search requires a maximum of `n' comparisons. What is the maximum number of comparisons needed when using binary search as a function of `n'? Show the math!

2. (20 pts) Let's say that you're creating a Set ADT, providing the methods to add an element to (and remove from) a set, checking if an object is a member of the set, etc. Since a set contains only distinct elements, adding an element that is already a member is ignored. You can use the set in the following way to check if a voter is voting multiple times:

```
  Set alreadyVoted = new Set(10000);        // maximum 10000 elements.
  // Add each person who votes to the set alreadyVoted.
  // ...
  // voter 591351 has come to vote, but first check if he/she has already
  // voted or not.
  if (!alreadyVoted.isMember(591351))
    alreadyVoted.add(591351);
  else
    throw new VoterFraudException("Call Police!").
```

This Set ADT maintains the elements (all distinct) in a non-resizable array. A requirement is that the isMember() method must be better than linear search, so we will have to use binary search, and the array must be kept sorted at all times.

```
public class Set {
  private Object[] data;              // the sorted array container
  private int size;                   // cardinality of the set

  /**
   * Creates an empty Set.
   * @param capacity the capacity of this set.
   */
  public Set(int capacity) {
    data = new Object[capacity];
    size = 0;
  }
  /**
   * Adds a new element to the set, but only if it's not already present.
   * @param e new element to add to this set.
   * @return true if added, false if it is already a member.
   * @throws NoMoreSpaceException if the capacity is exhausted.
   */
   public boolean add(Object e) {
     // TODO
  }

  /**
   * Checks whether the given element is a member of this set.
   * @param e the element to check for membership in this set.
   * @return true if it is a member, false otherwise.
  public boolean isMember(Object e) {
    // NOTE: The search must not take longer than log(n) time, where n
    // is the size of the set.
    // TODO
  }

  /**
   * Removes an element from this set.
   * @param e element to be removed from this set.
   * @return true if e was removed, or false if e was not a member.
   */
  public boolean remove(Object e) {
    // TODO
  }
```

(a) (10 pts) Write the Set.add() method. Note: array must remain sorted.
(b) (5 pts) Write the Set.isMember() method. Note: search must be faster than linear search.
(c) (5 pts) Write the Set.remove() method.

3. (30 pts) Queues using circular resizable-array container.
You're given the following Queue class using a built-in circular array as the underlying container (resizable). It maintains an index to the front of this queue ("front"), and the number of elements in this queue ("size").

```
  0    1    2    3    4    5    6   7   8
--------------------------------------------
| -4 | 17 |    |    |    | 5  | 7 | 9 | 15 |      size = 6
--------------------------------------------  capacity = 9
                             ^--front

  0    1    2    3    4    5    6   7   8
--------------------------------------------
| 5  | 7  | 9  | 15 |-4  | 17 |   |   |    |      size = 6
--------------------------------------------  capacity = 9
 ^--front
```

```
public class Queue {
  /** The queue size and the index of front element. */
  private int size, front;
  /** The data container. */
  private Object[] queue;

  /**
   * Creates an empty queue of given initial capacity.
   * @param initialCapacity the initial capacity of this queue.
   */
  public Queue(int initialCapacity) {
    queue = new Object[initialCapacity];
    size = 0;
    front = 0;
  }

  public Queue() { return this(100); }

  /**
   * Adds the specified element the end of this queue.
   * @param elem the element to add to the end of this queue.
   */
  public void enqueue(Object elem) {
    // TODO
  }

  /**
   * Dequeues the element at the front this queue, and returns it.
   * @return the element at the front of this queue.
   * @throws EmptyQueueException if this queue is empty.
   */
  public Object dequeue() {
    // TODO
  }

  /**
   * Resizes the underlying container to have at least the given capacity.
   * @param minCapacity the minimum capacity requested.
   */
  public void resize(int minSize) {
    // TODO
  }
}
```

(a) (10 pts) Write Queue.enqueue(). Note that the circular array container is resizable, so use the Queue.resize() method.

(b) (10 pts) Write Queue.dequeue().

(c) (10 pts) Write Qeueue.resize().

4. (30 pts) Stack using a head-referenced single-linked linear list.
You're free to choose where to add the new stack elements (to the beginning or the end of the list). You CANNOT add any new instance variable.

```
public class Stack {
  /**
   * The nodes in the linked list.
   */
  public class Node {
    public Object element;
    public Node next;
    /**
     * Creates a new node.
     */
    public Node(Object e, Node n) {
      element = e;
      next = n;
    }
  }

  // The reference to the head.
  private Node head;

  /**
   * Creates an empty stack.
   */
  public Stack() {
    head = null;
  }

  /**
   * Pushes the given element on this stack.
   * @param e the element to push onto this stack.
   */
  public void push(Object e) {
    // TODO
  }

  /**
   * Pops the top of stack from this stack.
   * @return the element that was on top before it was removed.
   * @exception EmptyStackException if this stack is empty.
   */
  public Object pop() throws EmptyStackException {
    // TODO
  }
  /**
   * Searches for the given element in this stack.
   * @param e the element to search for.
   * @return the offset from top of the stack if it exists, or -1 otherwise.
  public int search(Object e) {
    // If you search for element that is on the top of the stack, it should
    // return 0 (it's 0 offset from itself); if you search for the element
    // that is at the bottom of the stack, it would return 1 less than the
    // the size of this stack.
    // TODO
  }
}
```

(a) (10 pts) Write the Stack.push() method.
(b) (10 pts) Write the Stack.pop() method.
(c) (10 pts) Write the Stack.search() method.

5. (5 pts) Recursion
Write the recursive method to compute the power of an number using the Divide & Conquer strategy. The recursive definition is given below:

```
  a^n = 1,                                if n == 0
  a^n = a^(n/2) * a^(n/2),                if n is even
  a^n = a * a^((n-1)/2) * a^((n-1)/2),    if n is odd

  /**
   * Computes a^n using a Divide and Conquer strategy.
   * @param a the base.
   * @param n the exponent.
   * @return a^n.
  public static int pow(int a, int n) {
    // TODO
  }
```

No. of Pages	**6**
No. of Questions	**4**

Department of Computer Science and Engineering
MIDTERM EXAMINATION FALL 2010

CSE220: Data Structures

Total Marks: 100 Time Allowed: 1 Hour 20 Minutes

- Answer all questions, in any order; number each answer.

Section	*Marks*
Q1. Short questions	25
Q2. Array algorithms	30
Q3. Queues	25
Q4. Stack	20
Total	**100**

- You must write your name and student ID on the script and on the question paper

Name : ____________________________

ID : ____________________________

- Write all answers, including scratch notes, in the answer script
- You must turn in your question paper

1. (25 pts) Short questions

(a) [2.5 pts] What is the advantage of using a dummy head?
(b) [2.5 pts] What would you have to do if you want to support fast append operation for a singly-linked list?
(c) [2.5 pts] If you are to implement LIFO queue using a single-linked list, where would you put the new element in enqueue? Justify.
(d) [2.5 pts] How do you iterate through a cyclic array? Show code.
(e) [5 pts] You're reading keys one by one from the network (and you don't know how keys there are), and at any point in time, you want the keys to be sorted. Which sorting algorithm would you use? Justify.
(f) [5 pts] To find an element in an array using linear search requires a maximum of `n' comparisons. What is the maximum number of comparisons needed when using binary search as a function of `n'? Show the math!
(g) [5 pts] What search algorithm outperforms even binary search? Describe two limitations of this search algorithm. Can you use this to sort as well?

2. (30 pts) Let's say that you're creating a Set ADT, providing the methods to add an element to (and remove from) a set, checking if an object is a member of the set, etc. Since a set contains only distinct elements, adding an element that is already a member is ignored. You can use the set in the following way to check if a voter is voting multiple times:

```
Set alreadyVoted = new Set(10000);        // maximum 10000 elements.
// Add each person who votes to the set alreadyVoted.
// ...
// voter ID 591351 has come to vote, but first check if he/she has
// already voted or not.
if (!alreadyVoted.isMember(591351))
  alreadyVoted.add(591351);
else
  throw new VoterFraudException("Call Police!").
```

This Set ADT maintains the elements (all distinct) in a *non-resizable* array. One requirement we have is that the isMember() method must perform much better than linear search, so we will have to use binary search, and for that the array must be kept sorted at all times.

```
public class Set {
  private Object[] data;                // the sorted array container
  private int size;                     // cardinality of the set

  /**
   * Creates an empty Set.
   * @param capacity the capacity of this set.
   */
  public Set(int capacity) {
    data = new Object[capacity];
    size = 0;
  }

  /**
   * Checks whether the given element is a member of this set.
   * @param e the element to check for membership in this set.
   * @return true if it is a member, false otherwise.
   */
  public boolean isMember(Object e) {
    // NOTE: The search must not take longer than log(n) time, where n
    // is the size of the set. Basically, use binary search.
    // TODO
  }

  /**
   * Adds a new element to the set, but only if it's not already present.
   * @param e new element to add to this set.
   * @return true if added, false if it is already a member.
   * @throws SetFullException if the capacity is exhausted.
   */
   public boolean add(Object e) {
     // TODO
  }

  /**
   * Removes an element from this set.
   * @param e element to be removed from this set.
   * @return true if e was removed, or false if e was not a member.
   */
  public boolean remove(Object e) {
    // TODO
  }
}
```

(a) (10 pts) Write the Set.isMember() method. Note: search must be faster than linear search.
(b) (10 pts) Write the Set.add() method. Note: array must remain sorted.
(c) (10 pts) Write the Set.remove() method.

3. (25 pts) Queues using circular array container.
You're given the following Queue class using a built-in circular array as the underlying container (non-resizable). It maintains an index to the front of this queue ("front"), and the number of elements in this queue ("size").

```
   0    1    2    3    4    5    6   7   8
 -------------------------------------------
 | -4 | 17 |    |    |    | 5  | 7 | 9 | 15 |      size = 6
 -------------------------------------------   capacity = 9
                               ^--front
```

```
       0    1    2    3    4    5    6   7   8
     --------------------------------------------
     | 5  | 7  | 9  | 15 |-4 | 17  |    |   |    |      size = 6
     --------------------------------------------  capacity = 9
      ^--front

public class Queue {
  /** The queue size and the index of front element. */
  int size, front;
  /** The data container. */
  Object[] queue;

  /**
   * Creates an empty queue of given capacity.
   * @param capacity the capacity of this queue.
   */
  public Queue(int capacity) {
    queue = new Object[capacity];
    size = 0;
    front = 0;
  }

  /**
   * Adds the specified element the end of this queue.
   * @param elem the element to add to the end of this queue.
   * @throws QueueFullException if the underlying container is full.
   */
  public void enqueue(Object elem) {
    // TODO
  }

  /**
   * Dequeues the element at the front this queue, and returns it.
   * @return the element at the front of this queue.
   * @throws EmptyQueueException if this queue is empty.
   */
  public Object dequeue() {
    // TODO
  }

  /**
   * Searches for the element in the queue, and returns its offset from
   * the front of the queue if found.
   *
   * @param elem the element to search for.
   * @return the offset from front of the queue if found, or -1 otherwise.
   */
  public int search(Object elem) {
    // TODO
  }
}
```

(a) (7.5 pts) Write Queue.enqueue(). Note that the circular array container is not resizable.
(b) (7.5 pts) Write Queue.dequeue().
(c) (10 pts) Write Qeueue.search().

4. (20 pts) Stack using a single-linked linear list. We follow the usual design of push adding a new node to the beginning, and consequently, pop removing a node from the beginning as well. You cannot add any new instance variable.

```
public class Stack {
  /**
   * The nodes in the linked list.
   */
  public class Node {
    public Object element;
    public Node next;
    /**
     * Creates a new node.
     */
    public Node(Object e, Node n) {
      element = e;
      next = n;
    }
  }

  // The reference to the head.
  private Node head;

  /**
   * Creates an empty stack.
   */
  public Stack() {
    head = null;
  }

  /**
   * Pops the top of stack from this stack.
   * @return the element that was on top before it was removed.
   * @exception EmptyStackException if this stack is empty.
   */
  public Object pop() throws EmptyStackException {
    // TODO
  }

  /**
   * Moves the given object to the top of this stack (where the other stack
   * elements above the given object moves down by one, but the elements
   * already below the given object remain in place).
   * @param o the object to move to the top of this stack.
   * @return o if o was moved, or null if o is not an element on this
   *           stack.
   */
  public Object moveToTop(Object o) {
    // TODO
  }
}
```

(a) (10 pts) Write the Stack.pop() method.
(b) (10 pts) Write the Stack.moveToTop() method.

No. Of Pages	5
No. Of Questions	5

Department of Computer Science and Engineering MIDTERM EXAMINATION FALL 2011

CSE 220: Data Structures

Total Marks: 100   Time Allowed: 80 minutes

- Answer all questions, in any order; number each answer.

Category	*Marks*
Short questions	25
Stacks	25
Queues	25
Searching and Sorting	25
Total	**100**

- You must write your name and ID on the script ***and*** on the question paper.

Name : ____________________

ID   : ____________________

- Write all answers, including scratch notes, in the answer script.
- You must turn in ***both*** the answer script and the question paper.

Short questions [25 pts]

Problem 1. [25 pts]

a. You have to sort an array of 10^6 integers in the range $[-10^5, 10^5]$. What sorting algorithm would you pick to minimize running time? What if the input contains 10^2 integers instead? Justify your answer. [5 pts]

b. You're continously getting data from the Internet, which you want to maintain in a sorted array to facilitate searching using binary search algorithm. What sorting algorithm would you use and why? Must justify your answer to get credit. [5 pts]

c. Derive the maximum number of comparisons and shifts or exchanges needed to sort an array of "n" keys using *insertion sort*. [7.5 pts]

d. To find an element in an array using linear search requires a maximum of "n" comparisons. What is the maximum number of comparisons needed when using binary search as a function of "n"? Show the math, or else you get no credit! [7.5 pts]

Stacks [25 pts]

Problem 2. [**25 pts**] You have to implement a stack using a *head referenced singly-linked linear* list, given the following Node class.

```
public class Node {
    public Object element;      // The element within this Node
    public Object next;         // The reference to the next node
    /**
     * Creates a new Node.
     * @param e the element within
     * @param n the next node
     */
    public Node(Object e, Node n) {
        item = e;
        next = n;
    }
}

public class Stack {
    private Node head;  // The reference to the head node
    private int  size;  // The number of items on this stack

    /**
    * Pushes the given item on this stack.
    * @param item the item to push onto this stack
    */
    public void push(Object item) {
        // TODO
    }

    /**
    * Pops the top of stack from this stack.
    * @return the item that was on top before it was removed
    * @exception EmptyStackException if this stack is empty
    */
    public Object pop() throws EmptyStackException {
        // TODO
    }

    /**
    * Removes the given item from this stack, if it exists.
    * @param item the item to remove from this stack
    * @return true if found and removed, false otherwise
    */
    public boolean remove(Object item) {
        // TODO
    }
}
```

a. Write the Stack.push() method. [7.5 pts]
b. Write the Stack.pop() method. [7.5 pts]
c. Write the Stack.remove() method. [10 pts]

Queues [25 pts]

Problem 3. [**25 pts**] Assume the following implementation of a *resizable* queue using a *circular array* for the data container.

```
public class Queue {
    static final int DEFAULT_CAPACITY = 100;
    private Object[] queue;           // data container
    private int front;                // index of item at front
    private int size;                 // number of items in queue
    public Queue() {
        queue = new Object[DEFAULT_CAPACITY];
        size = 0; front = 0;
    }
    /**
     * Enqueues an item in this queue, resizing first if full.
     * @param item the item to put in this queue
     */
    public void enqueue(Object item) {
        // TODO
    }
    /**
```

```
     * Dequeues the item from the front of this queue.
     * @return the item that was at the front of this queue
     * @exception EmptyQueueException if this queue is empty
     */
    public Object dequeue() throws EmptyQueueException {
        // TODO
    }
    /**
     * Returns an string representation of this queue, starting with
     * the front of this queue. The queue items are enclosed in "[]",
     * the items are separated by a space.
     * @return the string representation of this queue
     */
    public String toString() {
        // TODO
    }
}
```

a. Write the Queue.enqueue() method, which resizes the underlying array if it's full. [12.5 pts]

b. Write the Queue.dequeue() method. [5 pts]

c. Write the Queue.toString() method. [7.5 pts]

Searching and Sorting [25 pts]

Problem 4. [12.5 pts] Implement a binary search algorithm that returns the index of the *first* occurrence of a key if it exists in the given array, or −1 otherwise.

```
/**
 * Finds the first occurrence of the given key.
 * @param data the sorted array of keys to search
 * @param key  the key to search for
 * @return index of the first occurrence if found, or -1 otherwise
 */
public static int search(Object[] data, Object key) {
    // TODO
}
```

Problem 5. [12.5 pts] You are to write a method that sorts a given array in *non-increasing order* using insertion sort.

```
/**
 * Sorts the given array using insertion sort
 * @param data the array of keys to sort in non-increasing order
 */
public static void sort(Object[] data) {
    // TODO
}
```

End of examination

1. (12 pts) Short questions (3 pts each)

a) Among the 3 sort algorithms - insertion, selection and bubble - which one has the lowest best-case cost, and when?
b) When we implemented an array-based stack, where in the array did each "push" add data to (and why)?
c) What type of sequence container would provide quick prepend and append operation? What abstract data type would benefit from such a container? Explain with an example.
d) What property of the sequence container is necessary for binary search algorithm to be efficient? Give an example of a container with this property, and one without.

2. (15 pts) Algorithm and recursion

You have to write the recursive and iterative solutions to the Fibonacci sequence, starting with the recursive definition.

(a) (5 pts) Write the recursive definition - that is, the recurrence relation and the base condition for fib(n) as we had done in class. Assume that n > 1, and fib(1) = fib(2) = 1.

(b) (5 pts) Write the recursive method using the answer from 2(a).

```
  /**
   * Recursively computes the fibonacci number for the specified number.
   * Pre-condition: n >= 1
   * @param n the number whose fibonacci to compute.
   * @return the fibnoacci value for the specified number.
   * @throws InvalidArgumentException if n < 1.
   */
  public static int fib(int n) {
    // TODO - your code goes here - write in the answer script
  }
```

(c) (5 pts) Write the iterative method using the answer from 2(a).

```
  /**
   * Iteratively computes the fibonacci number for the specified number.
   * Pre-condition: n >= 1
   * @param n the number whose fibonacci to compute.
   * @return the fibnoacci value for the specified number.
   * @throws InvalidArgumentException if n < 1.
   */
  public static int fibI(int n) {
    // TODO - your code goes here - write in the answer script
  }
```

3. (18 pts) Linked lists

(a) (6 pts) Circular linked list without dummy header
You have a circular doubly linked list, which does not use a dummy header node. It uses the same design used by the linked list discussed in class

(except for the dummy header node).

```
public class Node {                  public class List {
  int val;                             Node header;
  Node next;                           int size;
  Node prev;
                                       public List();
  Node(int v);                         public void addBefore(Node x, Node n);
  Node(int v, Node n, Node p);         // ... other methods
}                                    }
```

This version of List has an addBefore() method that adds a new node 'n' before the specified node 'x'.

```
class List {
  public List() {
    header = null;
    size = 0;
  }

  /**
   * Adds the node 'n' before the specified node 'x'. If 'x' refers to the
   * header, then prepend the new node (and as a result, header will change).
   * @param x the node before which the new node will go.
   * @param n the new node to add before 'x'.
   */
  public void addBefore(Node x, Node n) {
    // TODO - write the addBefore method. You cannot USE any other method,
    // so write all the code.
  }
}
```

(b) (6 pts) Circular linked list with dummy header
Now, let's change the design so that we use a dummy header, just like we had done in assignment 4.

```
class List {
  public List() {
    header = new Node(0);       // value in dummy header node is irrelevant.
    header.next = header.prev = header;
    size = 0;
  }

  /**
   * Adds the node 'n' before the specified node 'x'.
   * @param x the node before which the new node will go.
   * @param n the new node to add before 'x'.
   */
  public void addBefore(Node x, Node n) {
    // TODO - write the addBefore method. You cannot USE any other method,
    // so write all the code.
  }
}
```

(c) (6 pts) Emulating random access for the list
Now we want a List.get() method that gives the impression of random access for the doubly-linked circular list with a dummy header from part (b). The usual implementation has a worst-case cost of retrieving a node given its index is N (N is the number of nodes in the list); however, we want the worst case to be N/2. Write the code so that it is true.

```
  /**
   * Returns the element at the specified position in this list. The worst
   * case cost must be N/2, not N.
   * @param index index of element to return.
   * @return the element at the specified position in this list.
   * @throws IndexOutOfBoundsException if index is out of range
   *             (index < 0 || index >= size()).
   */
  public int get(int index) {
      // TODO - return the value in the node at the specified index.
  }
```

4. (15 pts) Queues

Assume the following List interface:

```
public class List {
  public List();                // creates and empty list.
  public int size();            // returns the size of the list.
  public int indexOf(int val);  // returns the index of the node with value.
  public int get(int index);    // returns the value at the given index.
  public void add(int index, int val);// adds at the given index.
  public int removeAt(int index);// removes node at the specified index,
                                   and returns the value of removed node.
}
```

You have to implement a Queue that using the List class above (and only using methods in the interface).

```
public class Queue {
  // Create an empty queue.
  public Queue() {
    queue = new List();
  }
  public void enqueue(int val); // adds the element with value at end
  public int dequeue();         // removes and returns the front element
  public int peek();            // gets the item at front of queue
  public void remove(int val);  // removes element matching value

  private List list;            // the underlying List container
}
```

You can ONLY use the List methods listed here.

a) (5 pts) Write Queue.enqueue(int val) method.

```
/**
 * Adds a value at the end of the queue.
 * @param val the integer value to enqueue
 */
public void enqueue(int val) {
  // TODO - YOUR CODE GOES HERE
}
```

b) (5 pts) Write Queue.dequeue() method.

```
/**
 * Removes the item at the head of the queue, and returns that value. Throws
 * QueueException if the queue is empty.
 * Precondition: Queue must not be empty.
 * @return the value of the dequeued element.
 * @throws QueueException if the queue is empty.
 */
public int dequeue() throws QueueException {
  // TODO - YOUR CODE GOES HERE
}
```

c) (5 pts) Write Queue.remove(int val) method.

```
/**
 * Removes the queue entry matching the given value. Throws QueueException
 * if the value is not found in the queue.
 * Preconditions: there is a queue element with the given value.
 * @param val the element with value to remove.
 * @throws QueueException if the value does not exist in the queue.
 */
public void remove(int val) throws QueueException {
  // TODO - YOUR CODE GOES HERE
}
```

5. (40 pts) Stacks

You're asked to write a stack of integers, and you have decided to create one based on (circular) doubly-linked lists to avoid problems of resizing arrays and so on. Instead of using a separate linked list class, the stack implements its own. Also, it maintains a reference to the tail node (ie., the last node) instead of to the head node (ie., the first node). There is no dummy header nor tail, so "tail" is null when the stack is empty.

```
public class Stack {
  public Stack() {
    tail = null;
    size = 0;
  }

  public void push(int val);    // pushes the element with value
  public int pop ();            // removes and return top of the stack
  public int peek();            // gets the item at the top of stack
  // Remove the stack element matching the value
  public bool remove (int val);

  private Node tail;            // reference to the last node
  private int size;             // the number of items on stack

  // Note this is how we can use this Stack class
  public void main(String[] args) {
    Stack stack = new Stack();  // empty stack
    // tail = null and size = 0 at this point
    stack.push(5); stack.push(15); stack.push(-31); stack.push(1);
    //
    //   1  ->  -31  ->  15  -> 5
    //                          ^
    //                         tail
    stack.pop();
    //
    //  -31  ->  15  -> 5
    //                  ^
    //                 tail
    stack.remove(15);
    //
    //  -31  -> 5
    //          ^
    //        tail
    stack.pop(); stack.pop();
    // tail = null and size = 0 at this point.
  }
}
```

a) (15 pts) Write Stack.push(int val) method.

```
/**
 * Pushes a value on the stack.
 * Postcondition: val is new top of stack; adjust size.
 * @param val the value to push.
 */
public void push(int val) {
  // YOUR CODE GOES HERE
}
```

b) (15 pts) Write Stack.pop() method.

```
/**
 * Pops the top of stack, and return the value. Throws StackException if
 * the stack is empty.
 * Precondition: stack is non-empty.
 * Postcondition: adjust size.
 * @return the value that was popped.
 * @throws StackException if the stack is empty.
 */
public int pop() throws StackException {
  // YOUR CODE GOES HERE
}
```

c) (10 pts) Write Stack.remove(int val) method.

```
/**
 * Removes the stack entry matching the given value. Throws StackException
 * if the value is not found in the stack.
 * Preconditions: there is a stack element with the given value.
 * Postcondition: adjust size.
 * @param val the element with value to remove.
 * @throws StackException if the value does not exist in the stack.
 */
public void remove(int val) throws StackException {
  // YOUR CODE GOES HERE
}
```

No. of Pages	**5**
No. of Questions	**5**

Department of Computer Science and Engineering
MIDTERM EXAMINATION SPRING 2010

CSE220: Data Structures

Total Marks: 100 Time Allowed: 1 Hour 20 Minutes

- Answer all questions, in any order; number each answer.

Section	*Marks*
Q1. Short questions	15
Q2. Short code fragments	10
Q3. Array algorithms	20
Q4. Queues	30
Q5. Stack	25
Total	**100**

- You must write your name and student ID on the script and on the question paper

Name : ____________________________

ID : ____________________________

- Write all answers, including scratch notes, in the answer script
- You must turn in your question paper

1. (15 pts) Short questions

(a) [2.5 pts] What would you need to efficiently add a new node to the end of a head-referenced single-linked list?
(b) [2.5 pts] Can you efficiently remove the node at the end of a head-referenced single-linked list? Justify your answer.
(c) [2.5 pts] If you are to implement a FIFO queue with a double-linked circular list with a tail reference, where would you put the new element that's being enqueued (and why)?
(d) [2.5 pts] What is the advantage of using a dummy head node?
(e) [5 pts] To find an element in an array using linear search requires a maximum of `n' comparisons. What is the maximum number of comparisons needed when using binary search as a function of `n'? Show the math!

2. (10 pts) Write the following methods (5 pts each)
(a) Given a reference to an array and the number of elements in the array, reverse the order of the elements in the array.

```
/**
 * Reverses the elements of the given array.
 * @param a reference to the array of elements.
 * @param size the number of elements in the array.
 */
static void reverse(Object[] a, int size) { ... }
```

(b) Given a reference to an circular array, index of the first element, and the number of elements in the array, print the elements in the reverse order.

```
/**
 * Prints the elements of the circular array in the reverse order.
 * @param a reference to the circular array of elements.
 * @param front the index of the first element.
 * @param size the number of elements in the array.
 */
static void printReverse(Object[] a, int front, int size) { ... }
```

3. (20 pts) You have a non-resizable SortedArray class that uses a built-in array to maintain its data in the non-decreasing order.

```
public class SortedArray {
  private Object[] data;        // the underlying built-in array container
  private int size;             // the number of elements being used

  /**
   * Creates an empty sorted array of given capacity.
   * @param capacity the capacity of the underlying array container.
   */
  public SortedArray(int capacity) {
    data = new Object[capacity];
    size = 0;
  }
  /**
   * Returns the index of the element if it exists, or -1 otherwise. Must
   * perform better than a sequential search algorithm.
   *
   * @param item the item to find.
   * @return the index if found, or -1 otherwise.
   */
  public int search(Object item) {
    // TODO
  }

  /**
   * Inserts an item in the sorted array in its appropriate place.
   *
   * @param item the item to insert, maintaining the sorted order.
   * @return the index where it was inserted.
   */
  public int insert(Object item) {
    // TODO
  }
}
```

(a) (10 pts) Write the Array.search() method.
(b) (10 pts) Write the Array.insert() method.

4. (30 pts) Queues using circular array container.
You're given the following Queue class using a built-in circular array as the underlying container (non-resizable). It maintains an index to the front of this queue ("front"), and the number of elements in this queue ("size").

```
       0    1    2    3    4    5    6   7   8
     --------------------------------------------
     | -4 | 17 |    |    |    |  5  | 7 | 9 | 15 |      size = 6
     --------------------------------------------  capacity = 9
                                    ^--front

       0    1    2    3    4    5    6   7   8
     --------------------------------------------
     | 5  | 7  | 9  | 15 |-4 | 17  |    |   |    |      size = 6
     --------------------------------------------  capacity = 9
      ^--front
```

```
public class Queue {
  /** The queue size and the index of front element. */
  int size, front;
  /** The data container. */
  Object[] queue;

  /**
   * Creates an empty queue of given capacity.
   * @param capacity the capacity of this queue.
   */
  public Queue(int capacity) {
    queue = new Object[capacity];
    size = 0;
    front = 0;
  }

  /**
   * Adds the specified element the end of this queue.
   * @param elem the element to add to the end of this queue.
   * @throws QueueFullException if the underlying container is full.
   */
  public void enqueue(Object elem) {
    // TODO
  }

  /**
   * Dequeues the element at the front this queue, and returns it.
   * @return the element at the front of this queue.
   * @throws EmptyQueueException if this queue is empty.
   */
  public Object dequeue() {
    // TODO
  }

  /**
   * Moves the given object to the front this queue (where the other queue
   * elements ahead of the given object moves back by one, but the elements
   * already behind the given object remain in place).
   * @param o the object to move to the front of this queue.
   * @return o if o was moved, or null if o is not an element in this queue.
   */
  public Object moveToFront(Object o) {
    // TODO
  }
}
```

(a) (10 pts) Write Queue.enqueue(). Note that the circular array container is not resizable.
(b) (10 pts) Write Queue.dequeue().
(c) (10 pts) Write Qeueue.moveToFront().

5. (25 pts) Stack using a head-referenced single-linked linear list.
You're free to choose where to add the new stack elements (to the beginning or the end of the list). You MAY NOT add any new instance variable.

```
public class Stack {
  /**
   * The nodes in the linked list.
   */
  public class Node {
    public Object element;
    public Node next;
    /**
     * Creates a new node.
     */
    public Node(Object e, Node n) {
      element = e;
      next = n;
    }
  }

  // The reference to the head.
  private Node head;

  /**
   * Creates an empty stack.
   */
  public Stack() {
    head = null;
  }

  /**
   * Pushes the element onto the stack.
   */
  public void push(Object e) {
    // TODO
  }

  /**
   * Pops the top of stack from this stack.
   * @return the element that was on top before it was removed.
   * @exception EmptyStackException if this stack is empty.
   */
  public Object pop() throws EmptyStackException {
    // TODO
  }

  /**
   * Removes the given element, if it exists, from the stack.
   *
   * @param e the element to remove, if it exists.
   * @return e if the element was removed, or null otherwise.
   */
   public Object remove(Object e) {
     // TODO
   }
}
```

(a) (7.5 pts) Write the Stack.push() method.
(b) (7.5 pts) Write the Stack.pop() method.
(c) (10 pts) Write the Stack.remove() method.

No. Of Pages	6
No. Of Questions	5

Department of Computer Science and Engineering MIDTERM EXAMINATION SPRING 2011

CSE 220: Data Structures

Total Marks: 100 Time Allowed: 80 minutes

- Answer all questions, in any order; number each answer.

Part	*Category*	*Marks*
I	Short questions	20
II	Arrays and Lists	30
III	Stacks	25
IV	Queues	25
	Total	**100**

- You must write your name and ID on the script ***and*** on the question paper.

Name : ____________________

ID : ____________________

- Write all answers, including scratch notes, in the answer script.
- You must turn in ***both*** the answer script and the question paper.

Part I: Short questions [20 pts]

Problem 1. [20 pts]

a. Adding a tail reference helps speeding up appending to a linear singly-linked list, but it also add a couple of special cases. What are theses special case and how would you handle these? [2.5 pts]

b. Would binary search be effective for a linked list? Just a "yes" or "no" answer is not sufficient. [2.5 pts]

c. To find an element in an array using linear search requires a maximum of "n" comparisons. What is the maximum number of comparisons needed when using binary search as a function of "n"? Show the math, or else you get no credit! [7.5 pts]

d. What is the maximum number of shifts (or exchanges) and comparisons for insertion sort on an *n-element* sequence? Show details, or else you get no credit. [7.5 pts]

Part II: Array and Lists [30 pts]

Problem 2. [**20 pts**] In assignment 2, we used an *unsorted array* to store the underlying data for a SET, which made the membership test very slow because of the necessity of sequential search. If we store the data in a *sorted array*, say in non-decreasing order, then we could use binary search to speed up the search signficantly. So, let's modify our Set class to store the set's elements in a auto-resizable built-in array in *non-decreasing* order.

```
public class Set {
    private Object[] data;      // the underlying built-in array container
    private int size;           // the number of elements in this set
    private static final int DEF_INIT_CAPACITY = 101;

    /**
     * Creates an empty set of given initial capacity.
     * @param initialCapacity the initial capacity of the array container.
     */
    public Set(int initialCapacity) {
        data = new Object[initialCapacity];
        size = 0;
    }

    /**
     * Creates an empty set of default capacity.
     */
    public Set() {
        this(DEFAULT_INIT_CAPACITY);
    }

    /**
     * Resizes the underlying if needed.
     *
     * @param minCapacity the minimum capacity needed.
     */
    private void resizeIfNeeded(int minCapacity) {
      // ASSUME CODE GIVEN, YOU CAN USE THIS METHOD IF YOU WISH.
    }

    /**
     * Returns true if the the given item is a member of this set, or
     * false otherwise. Must NOT use sequential search algorithm to locate
     * the item in this set!
     *
     * @param item the item to check for membership in this set.
     * @return true if it's a member, false otherwise.
     */
    public boolean isMember(Object item) {
        // TODO
    }

    /**
     * Adds an item to this set, only if it doesn't already exist.
     *
     * @param item the item to add to this set, maintaining the sorted
     *             order.
     */
    public void add(Object item) {
        // TODO
    }
}
```

a. Write the Set.isMember() method. Note: **cannot** use sequential search – too slow for real use. [10 pts]

b. Write the Set.add() method. Hint: it's basically the inner loop of insertion sort. [10 pts]

Problem 3. [**10 pts**] You are given the following Node class for a List data structure.

```
public class Node {
    public Object element;      // The element within this Node.
    public Object next;         // The reference to the next node in list.
    /**
     * Creates a new Node.
     * @param e the element within
     * @param n the next node
     */
    public Node(Object e, Node n) {
        element = e;
        next = n;
    }
}
```

Write the reverse() method that takes a head reference, and returns a reference to the head of the reversed list. You must not modify the original list, but rather create a new one and return a reference to its head node.

```
/**
 * Return a list that is the reverse of the given list. You MUST NOT
 * modify the given list, rather you should create and return a new list.
 *
 * @param list reference to the list to reverse
 * @return the reference to the reversed list
 */
public static Node reverse(Node list) {
    // TODO
}
```

Part III: Stacks [25 pts]

Problem 4. [**25 pts**] You have to implement a stack using a head-referenced single-linked linear linked list. Use the Node class from Problem 3.

```
public class Stack {
    private Node head;  // The reference to the head node.
    private int size;   // The number of items on this stack.

    /**
    * Pushes the given element on this stack.
    * @param e the element to push onto this stack.
    */
    public void push(Object e) {
        // TODO
    }

    /**
    * Pops the top of stack from this stack.
    * @return the element that was on top before it was removed.
    * @exception EmptyStackException if this stack is empty.
    */
    public Object pop() throws EmptyStackException {
        // TODO
    }

    /**
    * Removes the given element from this stack, if it exists.
    * @param item the item to remove from this stack.
    * @return true if found and removed, false otherwise.
    */
    public boolean remove(Object item) {
        // TODO
    }

    /**
    * Moves the given element, if it exists, to the top of this stack.
    * @param item the item to move to the top of this stack.
    * @return true if found and moved, false otherwise.
    */
    public boolean moveToTop(Object item) {
        // TODO
    }
}
```

a. Write the Stack.push() method. [5 pts]
b. Write the Stack.pop() method. [5 pts]
c. Write the Stack.remove() method. [7.5 pts]
d. Write the Stack.moveToTop() method. [7.5 pts]

Part IV: Queues [25 pts]

Problem 5. [**25 pts**] Assume the following implementation of a *resizable* queue using a *cyclic array* for the data container.

```
public class Queue {
    static final int DEFAULT_CAPACITY = 100;
    private Object[] queue;           // data container
    private int front;                // index of item at front
    private int size;                 // number of elements in queue
    public Queue() {
        queue = new Object[DEFAULT_CAPACITY];
        size = 0; front = 0;
    }
    /**
     * Enqueues an element in this queue, resizing first if full.
     * @param el the element to put in this queue
     */
    public void enqueue(Object el) {
        // TODO
    }
    /**
     * Dequeues the element from the front of this queue.
     * @return the element that was at the front of this queue
     * @exception QueueEmptyException if this queue is empty.
     */
    public Object dequeue() {
        // TODO
    }
    /**
     * Returns an array representation, with the front of the queue in
     * the first position.
     * @return the array representation.
     */
    public Object[] toArray() {
        // TODO
    }
}
```

a. Write the Queue.enqueue() method, which resizes the underlying array if it's full. [12.5 pts]

b. Write the Queue.dequeue() method. [7.5 pts]

c. Write the Queue.toArray() method. [5 pts]

End of examination

No. Of Pages	5
No. Of Questions	5

Department of Computer Science and Engineering
MIDTERM EXAMINATION SPRING 2012

CSE 220: Data Structures
Total Marks: 100 Time Allowed: 80 minutes

- Answer all questions, in any order; number each answer.

Category	*Marks*
Short questions	25
Stacks	25
Queues	25
Searching and Sorting	25
Total	**100**

- You must write your name and ID on the script ***and*** on the question paper.

Name : ____________________
ID : ____________________

- Write all answers, including scratch notes, in the answer script.
- You must turn in ***both*** the answer script and the question paper.

Short questions [25 pts]

Problem 1. [25 pts]

a. You have to sort an array of integer keys in the range $[-10^5, 10^5]$. If $n \leq 100$, which sorting algorithm would you pick to minimize running time, without regard for space? Which sorting algorithm would you use to minimize both running time and space. Justify your answer. [5 pts]

b. Given that insertion and selection sorts have quadratic worst-case performance, give two reasons why one would choose insertion sort over selection sort in real life. [5 pts]

c. Derive the maximum number of comparisons and shifts or exchanges needed to sort an array of “n” keys using *insertion sort*. [7.5 pts]

d. To find an element in an array using linear search requires a maximum of “n” comparisons. What is the maximum number of comparisons needed when using binary search as a function of “n”? Show the math, or else you get no credit! [7.5 pts]

Stacks [25 pts]

Problem 2. [**25 pts**] You have to implement a stack using a *dummy-tail referenced doubly-linked circular* list, given the following Node class.

```
public class Node {
    public Object element;      // The element within this Node
    public Object next;         // The reference to the next node
    public Object prev;         // The reference to the prev node
    /**
     * Creates a new Node.
     * @param e the element within
     * @param n the next node
     */
    public Node(Object e, Node n, Node p) {
        item = e;
        next = n;
        prev = p;
    }
}

public class Stack {
    private Node tail;  // The reference to the dummy tail node
    private int  size;  // The number of items on this stack

    /**
    * Creates a new empty stack.
    */
    public Stack() {
        // Create the dummy tail node, and make it circular.
        tail = new Node(null, null, null);
        tail.next = tail;
        tail.prev = tail;
        size = 0;
    }
    /**
    * Pushes the given item on this stack.
    * @param item the item to push onto this stack
    */
    public void push(Object item) {
        // TODO
    }

    /**
    * Pops the top of stack from this stack.
    * @return the item that was on top before it was removed
    * @exception EmptyStackException if this stack is empty
    */
    public Object pop() throws EmptyStackException {
        // TODO
    }

    /**
    * Removes the given item from this stack, if it exists.
    * @param item the item to remove from this stack
    * @return true if found and removed, false otherwise
    */
    public boolean remove(Object item) {
        // TODO
    }
}
```

a. Write the Stack.push() method. [7.5 pts]
b. Write the Stack.pop() method. [7.5 pts]
c. Write the Stack.remove() method. [10 pts]

Queues [25 pts]

Problem 3. [25 pts] Assume the following implementation of a *resizable* queue using a *circular array* for the data container.

```
public class Queue {
    static final int DEFAULT_CAPACITY = 100;
    private Object[] queue;            // data container
    private int front;                 // index of item at front
    private int size;                  // number of items in queue
    public Queue() {
        queue = new Object[DEFAULT_CAPACITY];
        size = 0; front = 0;
    }
    /**
     * Enqueues an item in this queue, resizing first if full.
     * @param item the item to put in this queue
     */
    public void enqueue(Object item) {
        // TODO
    }
    /**
     * Dequeues the item from the front of this queue.
     * @return the item that was at the front of this queue
     * @exception EmptyQueueException if this queue is empty
     */
    public Object dequeue() throws EmptyQueueException {
        // TODO
    }
    /**
     * Checks if the specified item exists in this queue.
     * @return offset from the front if it exists, or -1 otherwise.
     */
    public int indexOf(Object item) {
        // TODO
    }
}
```

a. Write the Queue.enqueue() method, which resizes the underlying array if it's full. [12.5 pts]

b. Write the Queue.dequeue() method. [5 pts]

c. Write the Queue.indexOf() method. [7.5 pts]

Searching and Sorting [25 pts]

Problem 4. [12.5 pts] Implement a binary search algorithm that returns the index of the *last* occurrence of a key if it exists in the given array, or −1 otherwise. Note that binary search returns the index of some occurrence, not necessarily the first or the last. You have to modify binary search to return the index of the *last* occurrence.

```
/**
 * Finds the last occurrence of the given key.
 * @param data the sorted array of keys to search
 * @param key  the key to search for
 * @return index of the last occurrence if found, or -1 otherwise
 */
public static int searchLast(Object[] data, Object key) {
    // TODO
}
```

Problem 5. [12.5 pts] You are to write a method that sorts a given array in *non-increasing order* using insertion sort.

```
/**
 * Sorts the given array using insertion sort
 * @param data the array of keys to sort in non-increasing order
 */
public static void sort(Object[] data) {
    // TODO
}
```

End of examination

CSE 220 Summer 2007 ID : ____________________
Midterm, July 05, 2007 Name : ____________________
Marks: ____ out of 100

1. (15 pts) Short questions

a) Describe two different situations where a copy constructor is used.
b) What is a good reason for using a const reference to a object when passing it to a function?
c) Why did we use a cyclic buffer (array) for a queue?
d) Why did we NOT use a cyclic linked list for a queue?
e) When a stack uses a linked list, what's more efficient for pushing - appending or inserting? Why?

2. (15 pts) Linked Lists

A student wrote the insertSorted() member function shown below for the following List class:

```
class Node {
private:
  int element;
  Node* next;
  Node* prev;

public:
  Node (int val);
  // ... other member function
};
```

```
class List {
private:
  Node* m_head;
  int m_size;
public:
  // ... other member functions
};
```

The function insertSorted() inserts an element in its sorted position in a doubly linked list (where the head's prev and tail's next pointers are both null). It iterates through the entire list and tries to find the position of the new element using the '<' operator. The List class does not use a dummy head node, which means that, for an empty, m_head = 0 and m_size = 0.

```
 1  // insertSorted: insert an element in its sorted position
 2  // Parameters:
 3  //   val (input)        : value to insert maintaining the sorted order
 4  // Returns: void
 5  //
 6  void List::insertSorted (int val) {
 7    Node* node = new Node(val);
 8
 9    if (m_head == 0) {            // empty list
10      m_size = 1;
11      m_head = node;
12    } else {
13      Node* current = m_head;
14      // find where to insert to maintain sorting order
15      while (current != 0) {
   if (current->val < val)
    current = current->next;
   else
    break;
  }
  // found location, now insert
  node->next = current;
  node->prev = current->prev;
  if (node->prev != 0)
   node->prev->next = node;
  node->next->prev = node;
  m_size++;
 }
}
```

The function has no syntax errors, but it does have logical errors that may cause problems in certain situations. Identify those errors and explain how you would modify the code shown above so that it works correctly. Oh, and write the corrected code of course!

3. (35 pts) Queues

a) (5 pts) Consider the following resizable integer-valued Array class interface:

```
class Array {
public:
  Array (int initialSize);
  Array (const Array &);
  ~Array ();

  int getSize () const;
  void setSize (int newSize);

  int get (int index) const;
  void set (int index, int val);

private:
  int* m_data;
  int m_size;
};
```

You're now asked to write a basic integer-valued Queue class that implements the push, pop and getFront member functions. Explain (using array functionality that you need, and whether Array class provides it or not).

b) (30 pts) You're asked to write a Queue class using this Array class, with the following interface:

```
class Queue {
```

```
public:
  Queue ();
  Queue (const Queue &);
  ~Queue ();

  // Normal queue operations.

  void enqueue (int val);       // add the element with value at end
  void dequeue ();              // remove the front the of the queue
  int getFront () const;        // get the item at front of queue

  // Special queue operations

  // Random access to get any element in the queue (0 <= index < size)
  int get (int index) const;
  // Find the index of the element matching the value in the queue
  int indexOf (int val) const;
  // Remove the queue element matching the value
  bool remove (int val);

  // Print functions
  void print () const;          // print items in the queue, front to back

private:
  Array m_array;                // the underling array object.
  int m_first;                  // the first element
  int m_last;                   // the last element
  int m_size;                   // current number of queued items
};
```

You have to use the member functions. Note this is how we can use this Queue class:

```
int main () {
  Queue queue;                  // empty queue
  queue.enqueue(5); queue.enqueue(15); queue.enqueue(-31);
  queue.print();                // prints: 5, 15, -31

  queue.remove(15);
  queue.print();                // prints: 5, -31

  int idx;
  idx = queue.indexOf(-31);     // idx is set to 1
  idx = queue.indexOf(5);       // idx is set to 0
  idx = queue.indexOf(15);      // idx is set to -1 (not found)

  int val;
  val = queue.get(0);           // val is set to 5
  val = queue.get(1);           // val is set to -31
  val = queue.get(2);           // invalid index, prints error message!
}
```

i) Write Queue::enqueue member function.

```
// Queue::enqueue: Add a value at the end of the queue
// Parameters:
//  val (input)         : the value to enqueue
//
// Returns: void
//
// Postcondition: adjust the m_size member data.
//
void Queue::enqueue (int val) {
  // YOUR CODE GOES HERE
}
```

ii) Write Queue::dequeue member function

```
// Queue::dequeue: Removes the item at the head of the queue
// Parameters:
//
// Returns: void
//
// Preconditions: queue must not be empty.
// Postcondition: adjust the m_size member data.
//
void Queue::dequeue () {
  // YOUR CODE GOES HERE
}
```

iii) Write Queue::get member function

```
// Queue::get: Get any element in the queue (0 <= index < size). If index
//  is out of bounds, Print error message and return 0.
// Parameters:
//  index (input)       : the index of the element to get
//
// Returns: the value at that index
//
// Preconditions: queue must not be empty (0 <= index < size).
//
int Queue::get (int index) const {
  // YOUR CODE GOES HERE
}
```

4. (35 pts) Stacks

You're asked to write a stack of integers, and you have decided to create one based on doubly-linked lists to avoid problems of resizing arrays and so on. The linked list uses a dummy head. When pushing a value, it's inserted in the beginning, so we don't need a separate pointer for the top of stack (it's head->next). The declarations are given below:

```
struct Node {
  int val;
  Node* next;
  Node* prev;

  Node (int val1) : val(val1), next(0), prev(0) { }
};

class Stack {
public:
  Stack ();
  Stack (const Stack &);
  ~Stack ();

  // Normal stack operations.

  void push (int val);          // push the element with value
  void pop ();                  // remove the top of the stack
  int getTop () const;          // get the item at the top of stack

  // Special stack operations

  // Random access to get any element in the stack (0 <= index < size)
  int get (int index) const;
  // Find the index of the element matching the value in the stack
  int indexOf (int val) const;
  // Remove the stack element matching the value
  bool remove (int val);

  // Print functions
  void print () const;          // print items in the stack, top to bottom

private:
  Node* m_head;                 // pointer to dummy head
  int m_size;                   // number of items on stack
};
```

You have to use the member functions. Note this is how we can use this Stack class:

```
int main () {
  Stack stack;                  // empty stack
  stack.push(5); stack.push(15); stack.push(-31);
  stack.print();                // prints: -31, 15, 5

  stack.remove(15);
  stack.print();                // prints: -31, 5

  int idx;
  idx = stack.indexOf(-31);     // idx is set to 0
  idx = stack.indexOf(5);       // idx is set to 1
  idx = stack.indexOf(15);      // idx is set to -1 (not found)

  int val;
  val = stack.get(0);           // val is set to -31
  val = stack.get(1);           // val is set to 5
  val = stack.get(2);           // invalid index, prints error message!
}
```

i) Write Stack::push member function.

```
// Stack::push: Add a value at the end of the stack
// Parameters:
//  val (input)         : the value to push
//
// Returns: void
//
// Postcondition: adjust the m_size member data.
//
void Stack::push (int val) {
  // YOUR CODE GOES HERE
}
```

ii) Write Stack::pop member function

```
// Stack::pop: Removes the item at the head of the stack
// Parameters:
//
// Returns: void
//
// Preconditions: stack must not be empty.
// Postcondition: adjust the m_size member data.
//
void Stack::pop () {
  // YOUR CODE GOES HERE
}
```

iii) Write Stack::remove member function

```
// Stack::remove: Remove the stack element matchine the given value. If
//  the value does not exist, return false.
// Parameters:
//  val (input)       : the value of the element to remove from stack
//
// Returns: true if value found (and removed), false otherwise
//
// Preconditions:
//
bool Stack::remove (int val) const {
  // YOUR CODE GOES HERE
}
```

1. (10 pts) Short questions (2.5 pts each)

(a) For a doubly-linked list of sorted values, which one is more efficient - linear or binary search, and why?
(b) What is the pattern for checking if two sequence containers are equal? You can use cse220.Array as an example.
(c) How do you iterate through a singly-linked circular list with a head reference? Assume the variable "head" is a reference to the first node in the list, so it's null if the list is empty.
(d) How do you iterate through a circular singly-linked list with a dummy tail reference? Assume the variable "tail" is a reference to the dummy tail node in the list.

2. (10 pts) Algorithm and recursion

You have to write the recursive solution to finding the maximum value in an array, starting with the recursive definition.

(a) (5 pts) Write the recursive definition based on the following observation: the maximum value in array is the larger of the maximum of the left half and the maximum of the right half.
Assume that we have another method max(x,y) that returns the larger of the two objects x and y.

(b) (5 pts) Write the recursive method using the answer from 2(a).

```
  /**
   * Recursively computes the maximum value in an array.
   * Pre-condition: a != null; a.length > 0
   * @param a the array of values.
   * @param lo the lower bound of the array 0 <= lo < a.length.
   * @param hi the upper bound of the array 0 <= hi < a.length.
   * @return the maximum value in the array.
   * @throws InvalidArgumentException if preconditions are not met.
   */
  public static int max(Comparable[] a, int lo, int hi) {
    // TODO - your code goes here - write in the answer script
  }
```

3. (20 pts) Arrays and sorting. You're given the following Array class.

```
public class Array {
  Comparable[] data;            // the underlying built-in array container
  int size;                     // the number of elements being used

  /**
   * Sorts the data in the array using insertion sort.
   */
  void sort() {
    // TODO
  }

  /**
   * Searches for the given key using binary search and returns its index
   * if found, or -1 if not found.
   * @param key the key being searched for
   * @returns the index of its first occurrence in the array, or -1 if
   *          it's not in the array.
   */
   int search(Comparable key) {
     // TODO
   }
}
```

(a) (10 pts) Write the Array.sort() method (MUST use insertion sort).
(b) (10 pts) Write the Array.search() method (MUST use binary search). You may use either the iterative or the recursive method; if you choose a recursive method, you will need to write a "helper" method which does the actual searching, given the range.

4. (30 pts) Queues using circular array container.
You're given the following Queue class using a built-in circular array as the underlying container, but with the following differences from our previous one:
- this queue is not automatically resized when enqueuing a new element; however, the user may choose to increase capacity by explicitly calling the Queue.resize(int) method.
- this queue only stores the index of the front element. The index to the next available location ("rear") is not essential - you can always calculate the "rear" given the front and index variables, along with the array capacity.

```
    0    1    2   3   4   5    6    7   8   9
  ----------------------------------------------
  | -4 | 17 |    |     |    |  5  | 7 | 9 | 15 |      size = 6
  ----------------------------------------------  capacity = 10
                              ^--front

    0    1    2   3   4   5    6    7   8   9
  ----------------------------------------------
  | 5  | 7  | 9  | 15  | -4 | 17  |   |   |    |      size = 6
  ----------------------------------------------  capacity = 10
   ^--front
```

The queues shown above are "equal".
NOTE the absense of an explicit "rear" variable.

```
public class Queue {
  /** The size and index of front element. */
  int size, front;
  /** The data container. */
  Object[] queue;
  /**
   * Adds the specified element the end of this queue.
   * @param elem the element to add to the end of this queue.
   * @throws FullQueueException if the queue is at full capacity.
  void enqueue(Object elem) throws FullQueueException {
    // TODO
  }
  /**
   * Removes the front element of this queue, and returns it.
   * @return the object that was at the front of this queue.
   * @throws EmptyQueueException if the queue is empty.
  Object dequeue() throws EmptyQueueException {
    // TODO
  }
  /**
   * Resizes the array container to have at least the given capacity.
   * @param minCapacity the minimum capacity to resize this queue to.
  void resize(int minCapacity) {
    // TODO
  }
}
```

(a) (10 pts) Write Queue.enqueue(). Note that the circular array container is not automatically resized when capacity is exhausted.
(b) (10 pts) Write Queue.dequeue().
(c) (10 pts) Write Queue.resize().

5. (30 pts) Stacks

You're asked to write a stack using a doubly-linked circular list as the underlying container. It maintains a reference to the head (ie., the first) node in the list. There is no dummy header, so "head" is null when the stack is empty. See the pictures after each operation to understand the policy Stack class uses to place the element in the linked list.

```
public class Stack {
  /** Creates an empty stack */
  public Stack() {
    head = null;
    size = 0;
  }

  public void push(Object val); // pushes the element with value
  public Object pop ();         // removes and return top of the stack

  private Node head;            // reference to the head node
  private int size;             // the number of elements on stack

  static class Node {
    Object item;                // the item in this node.
    Node next;
    Node prev;
    Node(Object item, Node next, Node prev) {
      this.item = item; this.next = next; this.prev = prev;
    }
  }

  // Note this is how we can use this Stack class
  public void main(String[] args) {
    Stack stack = new Stack();  // empty stack
    // head = null and size = 0 at this point
    stack.push(5); stack.push(15); stack.push(-31); stack.push(1);
    // The stack now looks like the following:
    //
    //   5 <-> 15 <-> -31 <-> 1
    //   ^--head
    //
    stack.pop();
    //
    //   5 <-> 15 <-> -31
    //   ^--head
    stack.pop(); stack.pop();
    //
    //   5
    //   ^--head
    stack.pop();
    // head = null and size = 0 at this point.
  }
}
```

a) (15 pts) Write Stack.push(Object elem) method.

```
/**
 * Pushes an element on the stack.
 * Postcondition: elem is new top of stack; adjust size.
 * @param elem the element to push.
 */
public void push(Object elem) {
  // YOUR CODE GOES HERE
}
```

b) (15 pts) Write Stack.pop() method.

```
/**
 * Pops the top of stack, and return the element.
 * Precondition: stack is non-empty.
 * Postcondition: adjust size.
 * @return the element that was popped.
 * @throws EmptyStackException if the stack is empty.
 */
public Object pop() throws EmptyStackException {
  // YOUR CODE GOES HERE
}
```

6. [BONUS] (5 pts) Complexity analysis of selection sort.
The selection sort algorithm is given below:

```
for i <- 0 to n-2 do
    min <- i
    for j <- (i + 1) to n-1 do
        if A[j] < A[min]
            min <- j
    swap A[i] and A[min]
```

Derive the complexity for this algorithm for an array of size `n'.

No. of Pages	**5**
No. of Questions	**6**

Department of Computer Science and Engineering
MIDTERM EXAMINATION SUMMER 2009

CSE220: Data Structures

Total Marks: 100   Time Allowed: 1 Hour 20 Minutes

- Answer all questions, in any order; number each answer.

Section	*Marks*
Q1. Short questions	15
Q2. Short code fragments	10
Q3. Array algorithms	15
Q4. Queues	30
Q5. Stack	20
Q6. Recursion	10
Total	**100**

- You must write your name and student ID on the script and on the question paper

Name : ____________________________

ID : ____________________________

- Write all answers, including scratch notes, in the answer script
- You must turn in your question paper

1. (15 pts) Short questions

(a) [2.5 pts] What would you need to efficiently add a new node to the end of a head-referenced single-linked list?
(b) [2.5 pts] Can you efficiently remove the node at the end of a head-referenced single-linked list?
(c) [2.5 pts] If you are to implement a FIFO queue with a double-linked circular list with a tail reference, where would you put the new element that's being enqueued (and why)?
(d) [2.5 pts] What is the advantage of using a dummy head node?
(e) [5 pts] To find an element in an array using linear search requires a maximum of \`n' comparisons. What is the maximum number of comparisons needed when using binary search as a function of \`n'? Show the math!

2. (10 pts) Write the following methods (5 pts each)
(a) Given a reference to an array and the number of elements in the array, reverse the order of the elements in the array.

```
/**
 * Reverses the elements of the given array.
 * @param a reference to the array of elements.
 * @param size the number of elements in the array.
 */
static void reverse(Object[] a, int size) { ... }
```

(b) Given a reference to an circular array, index of the first element, and the number of elements in the array, print the elements in the reverse order.

```
/**
 * Prints the elements of the circular array in the reverse order.
 * @param a reference to the circular array of elements.
 * @param front the index of the first element.
 * @param size the number of elements in the array.
 */
static void printReverse(Object[] a, int front, int size) { ... }
```

3. (15 pts) You have a non-resizable Array class.

```
public class Array {
  private Object[] data;        // the underlying built-in array container
  private int size;             // the number of elements being used

  /**
   * Creates an empty array of given capacity.
   * @param capacity the capacity of the underlying array container.
   */
  public Array(int capacity) {
    data = new Object[capacity];
    size = 0;
  }
  /**
   * Returns the index of the last occurrence of the item in this array.
   * @param item the item to find.
   * @return the index if found, or -1 otherwise.
   */
  public int lastIndexOf(Object item) {
    // TODO
  }

  /**
   * Sorts this array using insertion sort.
   */
  public void sort() {
    // TODO
  }
}
```

(a) (5 pts) Write the Array.lastIndexOf() method.
(b) (10 pts) Write the Array.sort() method. MUST use insertion sort.

4. (30 pts) Queues using circular array container.
You're given the following Queue class using a built-in circular array as the underlying container (non-resizable). It maintains an index to the front of this queue ("front"), and the number of elements in this queue ("size").

```
       0    1    2    3    4    5    6   7   8
     --------------------------------------------
     | -4 | 17 |    |    |    | 5  | 7 | 9 | 15 |      size = 6
     --------------------------------------------  capacity = 9
                                  ^--front

       0    1    2    3    4    5    6   7   8
     --------------------------------------------
     | 5  | 7  | 9  | 15 |-4 | 17  |    |   |   |      size = 6
     --------------------------------------------  capacity = 9
      ^--front
```

```
public class Queue {
  /** The queue size and the index of front element. */
  int size, front;
  /** The data container. */
  Object[] queue;

  /**
   * Creates an empty queue of given capacity.
   * @param capacity the capacity of this queue.
   */
  public Queue(int capacity) {
    queue = new Object[capacity];
    size = 0;
    front = 0;
  }

  /**
   * Adds the specified element the end of this queue.
   * @param elem the element to add to the end of this queue.
   * @throws QueueFullException if the underlying container is full.
   */
  public void enqueue(Object elem) {
    // TODO
  }

  /**
   * Dequeues the element at the front this queue, and returns it.
   * @return the element at the front of this queue.
   * @throws EmptyQueueException if this queue is empty.
   */
  public Object dequeue() {
    // TODO
  }

  /**
   * Moves the given object to the front this queue (where the other queue
   * elements ahead of the given object moves back by one, but the elements
   * already behind the given object remain in place).
   * @param o the object to move to the front of this queue.
   * @return o if o was moved, or null if o is not an element in this queue.
   */
  public Object moveToFront(Object o) {
    // TODO
  }
}
```

(a) (10 pts) Write Queue.enqueue(). Note that the circular array container
    is not resizable.
(b) (10 pts) Write Queue.dequeue().
(c) (10 pts) Write Qeueue.moveToFront().

5. (20 pts) Stack using a single-linked linear list with a dummy head.
You're free to choose where to add the new stack elements (to the beginning or the end of the list). You cannot add any new instance variable.

```
public class Stack {
  /**
   * The nodes in the linked list.
   */
  public class Node {
    public Object element;
    public Node next;
    /**
     * Creates a new node.
     */
    public Node(Object e, Node n) {
      element = e;
      next = n;
    }
  }

  // The reference to the dummy head.
  private Node head;

  /**
   * Creates an empty stack.
   */
  public Stack() {
    // Create the dummy tail.
    head = new Node("dummy", null);
  }

  /**
   * Pops the top of stack from this stack.
   * @return the element that was on top before it was removed.
   * @exception EmptyStackException if this stack is empty.
   */
  public Object pop() throws EmptyStackException {
    // TODO
  }

  /**
   * Moves the given object to the top of this stack (where the other stack
   * elements above the given object moves down by one, but the elements
   * already below the given object remain in place).
   * @param o the object to move to the top of this stack.
   * @return o if o was moved, or null if o is not an element on this stack.
   */
  public Object moveToTop(Object o) {
    // TODO
  }
}
```

(a) (10 pts) Write the Stack.pop() method.
(b) (10 pts) Write the Stack.moveToTop() method.

6. (10 pts) Recursion
You have to print the elements of a head-referenced singly-linked list using recursion.
(a) Write the recursive statement(s) and base conditions(s).
(b) Write the code that takes a reference to the starting node, and prints all the elements till the end of the list.

```
  private static void print(Node n) {
    // TODO
  }
```

No. of Pages	**6**
No. of Questions	**4**

Department of Computer Science and Engineering
MIDTERM EXAMINATION SUMMER 2010

CSE220: Data Structures

Total Marks: 100 Time Allowed: 1 Hour 20 Minutes

- Answer all questions, in any order; number each answer.

Section	*Marks*
Q1. Short questions	20
Q2. Array algorithms	20
Q3. Queues	30
Q4. Stack	30
Total	**100**

- You must write your name and student ID on the script and on the question paper

Name : ____________________________

ID : ____________________________

- Write all answers, including scratch notes, in the answer script
- You must turn in your question paper

1. (20 pts) Short questions

(a) [2.5 pts] Adding a tail reference helps speeding up appending to a linear singly-linked list, but it also adds a special case. What is this special case and how would you handle it?
(b) [2.5 pts] Why did we use a circular array when implementing an array-based queue?
(c) [2.5 pts] Would binary search be effective for a linked list? Just a "yes" or "no" answer is not sufficient.
(d) [2.5 pts] How do you iterate through a circular array? Show code.
(e) [5 pts] To find an element in an array using linear search requires a maximum of \`n' comparisons. What is the maximum number of comparisons needed when using binary search as a function of \`n'? Show the math!
(f) [5 pts] What is the maximum number of exchanges (or shifts) and comparisons for insertion sort on an n-element sequence?

2. (20 pts) In our FriendGraph assignment, we used an array to store the information for each person in the graph, and used sequential search to find a person given a name. To speed this up, we want to maintain the persons sorted in non-decreasing manner, and then we can use binary search to speed up the search significantly. So, let's create a new class called SortedArray that does exactly that. It's a resizable array container that adds element in such a way that the elements remain sorted.

```
public class SortedArray {
  private Object[] data;     // the underlying built-in array container
  private int size;          // the number of elements being used
  private static final int DEF_CAPACITY = 101;

  /**
   * Creates an empty sorted array of given capacity.
   * @param initialCapacity the initial capacity of the array container.
   */
  public SortedArray(int initialCapacity);

  /**
   * Creates an empty sorted array of default capacity.
   */
  public SortedArray();
```

```
    /**
     * Returns the index of the element if it exists, or -1 otherwise.
     * Must perform better than a sequential search algorithm.
     *
     * @param item the item to find.
     * @return the index if found, or -1 otherwise.
     */
    public int indexOf(Object item) {
      // TODO
    }

    /**
     * Adds an item to the sorted array in its appropriate place.
     *
     * @param item the item to add, maintaining the sorted order.
     * @return the index where it was added.
     */
    public int add(Object item) {
      // TODO
    }
  }
```

(a) (2 pts) What is the precondition for the elements that can be added to a SortedArray?
(b) (8 pts) Write the SortedArray.indexOf() method. Note: CANNOT USE sequential search - too slow for real use.
(c) (10 pts) Write the SortedArray.add() method. MUST USE TECHNIQUE USED IN INSERTION SORT.

3. (30 pts) Queues using circular resizable-array container.
You're given the following Queue class using a built-in circular array as the underlying container (resizable). It maintains an index to the front of this queue ("front"), and the number of elements in this queue ("size").

```
  0    1    2    3    4    5    6   7   8
--------------------------------------------
| -4 | 17 |    |    |    | 5  | 7 | 9 | 15 |      size = 6
--------------------------------------------  capacity = 9
                             ^--front

  0    1    2    3    4    5    6   7   8
--------------------------------------------
| 5  | 7  | 9  | 15 |-4 | 17  |   |   |    |      size = 6
--------------------------------------------  capacity = 9
 ^--front
```

```
public class Queue {
  /** The queue size and the index of front element. */
  private int size, front;
  /** The data container. */
  private Object[] queue;

  /**
   * Creates an empty queue of given initial capacity.
   * @param initialCapacity the initial capacity of this queue.
   */
  public Queue(int initialCapacity) {
    queue = new Object[initialCapacity];
    size = 0;
    front = 0;
  }

  public Queue() { return this(100); }

  /**
   * Adds the specified element the end of this queue.
   * @param elem the element to add to the end of this queue.
   */
  public void enqueue(Object elem) {
    // TODO
  }

  /**
   * Dequeues the element at the front this queue, and returns it.
   * @return the element at the front of this queue.
   * @throws EmptyQueueException if this queue is empty.
   */
  public Object dequeue() {
    // TODO
  }

  /**
   * Resizes this queue to have at least the given capacity.
   * @param minCapacity the minimum capacity requested.
   */
  public void resize(int minSize) {
    // TODO
  }
}
```

(a) (10 pts) Write Queue.enqueue(). Note that the circular array container is resizable, so call the Queue.resize() method in (c) instead of duplicating the same code.
(b) (10 pts) Write Queue.dequeue().
(c) (10 pts) Write Qeueue.resize().

4. (30 pts) Stack using a head-referenced single-linked linear list.

```
public class Stack {
  /**
   * The nodes in the linked list.
   */
  public class Node {
    public Object element;
    public Node next;
    /**
     * Creates a new node.
     */
    public Node(Object e, Node n) {
      element = e;
      next = n;
    }
  }

  // The reference to the head. NOT A DUMMY!
  private Node head;
  // The number of items on the stack.
  private int size;

  /**
   * Pushes the given element on this stack.
   * @param e the element to push onto this stack.
   */
  public void push(Object e) {
    // TODO
  }

  /**
   * Pops the top of stack from this stack.
   * @return the element that was on top before it was removed.
   * @exception EmptyStackException if this stack is empty.
   */
  public Object pop() throws EmptyStackException {
    // TODO
  }
```

```
    /**
     * Moves the first matching item (starting from the current top of
     * the stack) to the top of the stack. Must not use another stack!
     * @param item the item to search for and move.
     * @return true if found and moved, false otherwise.
     */
    public boolean moveToTop(Object item) {
      // Cannot use another stack, so you'll have to manipulate the low
      // level data structure instead.
      // TODO
    }
  }
```

(a) (7.5 pts) Write the Stack.push() method.
(b) (7.5 pts) Write the Stack.pop() method.
(c) (15 pts) Write the Stack.moveToTop() method.

No. Of Pages	5
No. Of Questions	5

Department of Computer Science and Engineering MIDTERM EXAMINATION SUMMER 2011

CSE 220: Data Structures

Total Marks: 100 Time Allowed: 80 minutes

- Answer all questions, in any order; number each answer.

Category	*Marks*
Short questions	25
Stacks	25
Queues	25
Searching and Sorting	25
Total	**100**

- You must write your name and ID on the script ***and*** on the question paper.

Name : ______________________
ID : ______________________

- Write all answers, including scratch notes, in the answer script.
- You must turn in ***both*** the answer script and the question paper.

Short questions [25 pts]

Problem 1. [25 pts]

a. Why did we choose a circular array over a linear one to implement array-based Queue ADT? [2.5 pts]

b. Given that insertion and selection sorts have the same worst-case performance, give two reasons why one would choose insertion sort over selection sort in real life. [5 pts]

c. In key-indexed searching, what is the **setup** cost for an input of size n, and with a range of k? [5 pts]

d. How would you model a FIFO queue and a LIFO stack using a priority queue? [5 pts]

e. To find an element in an array using linear search requires a maximum of "n" comparisons. What is the maximum number of comparisons needed when using binary search as a function of "n"? Show the math, or else you get no credit! [7.5 pts]

Stacks [25 pts]

Problem 2. [25 pts] You have to implement a stack using a **dummy-head referenced doubly-linked circular** list, given the following Node class.

```
public class Node {
    public Object item;         // The item within this Node.
    public Object next;         // The reference to the next node.
    public Object prev;         // The reference to the previous node.
    /**
     * Creates a new Node.
     * @param e the item within
     * @param n the next node
     * @param p the previous node
     */
    public Node(Object e, Node n, Node p) {
        item = e;
        next = n;
        prev = p;
    }
}


public class Stack {
    private Node head;  // The reference to the dummy head node.
    private int size;   // The number of items on this stack.

    /**
    * Pushes the given item on this stack.
    * @param item the item to push onto this stack
    */
    public void push(Object item) {
        // TODO
    }

    /**
    * Pops the top of stack from this stack.
    * @return the item that was on top before it was removed
    * @exception EmptyStackException if this stack is empty
    */
    public Object pop() throws EmptyStackException {
        // TODO
    }

    /**
    * Removes the given item from this stack, if it exists.
    * @param item the item to remove from this stack
    * @return true if found and removed, false otherwise
    */
    public boolean remove(Object item) {
        // TODO
    }
}
```

a. Write the Stack.push() method. [7.5 pts]
b. Write the Stack.pop() method. [7.5 pts]
c. Write the Stack.remove() method. [10 pts]

Queues [25 pts]

Problem 3. [**25 pts**] Assume the following implementation of a *resizable* queue using a *circular array* for the data container.

```
public class Queue {
    static final int DEFAULT_CAPACITY = 100;
    private Object[] queue;           // data container
    private int front;                // index of item at front
    private int size;                 // number of items in queue
    public Queue() {
        queue = new Object[DEFAULT_CAPACITY];
        size = 0; front = 0;
    }
    /**
     * Enqueues an item in this queue, resizing first if full.
     * @param item the item to put in this queue
     */
    public void enqueue(Object item) {
        // TODO
    }
    /**
     * Dequeues the item from the front of this queue.
     * @return the item that was at the front of this queue
     * @exception EmptyQueueException if this queue is empty
     */
    public Object dequeue() throws EmptyQueueException {
        // TODO
    }
    /**
     * Returns an array representation, with the front of the queue in
     * the first position.
     * @return the array representation
     */
    public Object[] toArray() {
        // TODO
    }
}
```

a. Write the Queue.enqueue() method, which resizes the underlying array if it's full. [12.5 pts]
b. Write the Queue.dequeue() method. [5 pts]
c. Write the Queue.toArray() method. [7.5 pts]

Searching and Sorting [25 pts]

Problem 4. [**15 pts**] Given an array of n **integers** in the range $[-\frac{n}{2} \ldots \frac{n}{2}]$, your task is to write a *key-indexed* search method that takes a key and returns a position within the array if it exists, or -1 otherwise. You need two methods – one to setup the **auxiliary** array, and other to perform the actual search (which uses the auxiliary array). Firstly, note that you need to return the **position**, and not just whether it exists or not; this affects what you put in your auxiliary array. Secondly, note that the numbers are not necessarily **non-negative**.

```
    /**
     * Creates the auxiliary array for key-indexed search.
     * @param data the input data
     * @return the auxiliary array needed for key-indexed search
     */
    public static int[] setup(int[] data) {
        // TODO
    }
    /**
     * Returns the position of the key in the array, if it exists.
     * @param aux the auxiliary array
     * @param key the key to search for
     * @return the position if found, -1 otherwise
     */
    public static int search(int[] aux, int key) {
        // TODO
    }
```

a. Write the setup method that returns the auxiliary array given the input data. [10 pts]

b. Write the search method that returns the position of a key, if it exists. Note that you need to return the **position**, and not just whether it exists or not. This affects what you put in your auxiliary array. [5 pts]

Problem 5. [**10 pts**] You are to write a method that sorts a given array in *non-decreasing order* using insertion sort.

```
/**
 * Sorts the given array using insertion sort.
 * @param data the array of keys to sort
 */
public static void sort(Object[] data) {
    // TODO
}
```

End of examination

No. Of Pages	5
No. Of Questions	5

Department of Computer Science and Engineering MIDTERM EXAMINATION SUMMER 2012

CSE 220: Data Structures
Total Marks: 100   Time Allowed: 80 minutes

- Answer all questions, in any order; number each answer.

Category	*Marks*
Short questions	30
Stacks	20
Queues	25
Searching and Sorting	25
Total	**100**

- You must write your name and ID on the script ***and*** on the question paper.

Name : ____________________
ID   : ____________________

- Write all answers, including scratch notes, in the answer script.
- You must turn in ***both*** the answer script and the question paper.

Short questions [30 pts]

Problem 1. [**30 pts**]

a. Why did we choose a circular array over a linear one to implement array-based Queue ADT? [2.5 pts]

b. Given that insertion and selection sorts have the same worst-case performance, give two reasons why one would choose insertion sort over selection sort in real life. [5 pts]

c. What auxiliary data structure would you need (and why) to sort a set of *(key,value)* pairs using key-indexed sorting, if they keys are not necessarily distinct? Assume that the keys are all non-negative integers. [5 pts]

d. How can you sort using a priority queue? [5 pts]

e. To find an element in an array using linear search requires a maximum of "n" comparisons. What is the maximum number of comparisons needed when using binary search as a function of "n"? Show the math, or else you get no credit! [7.5 pts]

f. Fill in the following table (in your answer sheet please), using **n** as the number of keys in the input. Consider the *worst-case* behavior when filling in the columns. [5 pts]

	# comparisons	# shifts/swaps	stable?	adaptive?
insertion sort				
selection sort				

Stacks [20 pts]

Problem 2. [**20 pts**] You have to implement a stack using a **dummy-head referenced singly-linked linear** list, given the following Node class.

```
public class Node {
    public Object item;         // The item within this Node.
    public Object next;         // The reference to the next node.
    /**
     * Creates a new Node.
     * @param e the item within
     * @param n the next node
     */
    public Node(Object e, Node n) {
        item = e;
        next = n;
    }
}

public class Stack {
    private Node head;  // The reference to the dummy head node.
    private int size;   // The number of items on this stack.

    /**
     * Creates an empty stack.
     */
    public Stack() {
        // head refers to the dummy header node.
        head = new Node(null, null);
        size = 0;
    }
    /**
     * Pushes the given item on this stack.
     * @param item the item to push onto this stack
     */
    public void push(Object item) {
        // TODO
    }

    /**
    * Pops the top of stack from this stack.
    * @return the item that was on top before it was removed
    * @exception EmptyStackException if this stack is empty
    */
    public Object pop() throws EmptyStackException {
        // TODO
    }

    /**
    * Removes the given item from this stack, if it exists.
    * @param item the item to remove from this stack
    * @return true if found and removed, false otherwise
    */
    public boolean remove(Object item) {
        // TODO
    }
}
```

a. Write the Stack.push() method. [5 pts]

b. Write the Stack.pop() method. [5 pts]

c. Write the Stack.remove() method. [10 pts]

Queues [25 pts]

Problem 3. [**25 pts**] Assume the following implementation of a *resizable* queue using a *circular array* for the data container.

```
public class Queue {
    static final int DEFAULT_CAPACITY = 100;
    private Object[] queue;           // data container
    private int front;                // index of item at front
    private int size;                 // number of items in queue
    /**
     * Creates an empty queue of default capacity.
     */
    public Queue() {
        queue = new Object[DEFAULT_CAPACITY];
        size = 0; front = 0;
    }
    /**
     * Enqueues an item in this queue, resizing first if full.
     * @param item the item to put in this queue
     */
    public void enqueue(Object item) {
        // TODO
    }
    /**
     * Dequeues the item from the front of this queue.
     * @return the item that was at the front of this queue
     * @exception EmptyQueueException if this queue is empty
     */
    public Object dequeue() throws EmptyQueueException {
        // TODO
    }
    /**
     * Returns an array representation, with the front of the queue in
     * the first position.
     * @return the array representation
     */
    public Object[] toArray() {
        // TODO
    }
}
```

a. Write the Queue.enqueue() method, which resizes the underlying array if it's full. [12.5 pts]

b. Write the Queue.dequeue() method. [5 pts]

c. Write the Queue.toArray() method. [7.5 pts]

Searching and Sorting [25 pts]

Problem 4. [**15 pts**] Given a sorted array of *integers*, you are to find the index of the *last* occurrence of a given key using a modified *binary search* algorithm.

```
/**
 * Finds the index of the last occurrence of the given key
 * @param data sorted input data
 * @param key key to find the index of
 * @return the index if found, or -1 otherwise
 */
public static int indexOf(int[] data, int key) {
    // TODO -- Must use a modified binary search
}
```

Problem 5. [**10 pts**] You are to write a method that sorts a given array in *non-increasing order* using *insertion sort*.

```
/**
 * Sorts the given array in non-increasing order using insertion sort.
 * @param data the array of keys to sort
 */
public static void sort(Object[] data) {
    // TODO
}
```

End of examination

No. of Pages	**5**
No. of Questions	**4**

Department of Computer Science and Engineering
MIDTERM EXAMINATION SPRING 2009

CSE220: Data Structures

Total Marks: 100   Time Allowed: 1 Hour 20 Minutes

- Answer all questions, in any order; number each answer.

Section	*Marks*
Q1. Short questions	10
Q2. Array algorithms	30
Q3. Queues	30
Q4. Stack	30
Total	**100**

- You must write your name and student ID on the script and on the question paper

Name : ___________________________

ID : ___________________________

- Write all answers, including scratch notes, in the answer script
- You must turn in your question paper

1. (10 pts) Short questions (5 pts each)

(a) Given a linked list, getting the element at a given index requires a maximum of `n' advances, where `n' is the number of elements. If you have a circular doubly linked list with head reference, and if you know the number of elements in the list, can you do any better? If so, how? If not, why?

(b) To find an element in an array using linear search requires a maximum of `n' comparisons. What is the maximum number of comparisons needed when using binary search as a function of `n'? Show the math!

2. (30 pts) You have an SortedArray class that maintains the data in non-decreasing order. The array is non-resizable.

```
public class SortedArray {
  private Object[] data;        // the underlying built-in array container
  private int size;             // the number of elements being used

  /**
   * Creates an empty sorted array of given capacity.
   * @param capacity the capacity of the underlying array container.
   */
  public SortedArray(int capacity) {
    // TODO
  }

  /*
   * Returns the index of the item in this sorted array (MUST use binary
   * search).
   * @param item the item to find.
   * @return the index if found, or -1 otherwise.
   */
  public int indexOf(Object item) {
     // TODO
  }

  /**
   * Inserts the given item maintaining sorted order.
   * @param item the item to insert.
   * @return the index where the item was inserted.
   * @exception ArrayFullException if there is not sufficient capacity.
   */
  public int add(Object item) {
    // TODO
  }

  /**
   * Removes the given item from this sorted array.
   * @param item the item to remove from this sorted array.
   * @return reference to the item removed, or null if not found.
   */
  public Object remove(Object item) {
    // TODO
  }

}
```

(a) (5 pts) Write the SortedArray.SortedArray() constructor.
(b) (10 pts) Write the SortedArray.indexOf() method. Must use binary search.
(c) (10 pts) Write the SortedArray.add() method. Hint: this is the inner loop in insertion sort.
(d) (5 pts) Write the SortedArray.remove() method.

3. (30 pts) Queues using circular array container.
You're given the following Queue class using a built-in circular array as the underlying container, where the container is automatically resized when needed (using the "resize" method). It maintains an index to the front of the queue ("front"), and the number of elements in the queue ("size").

```
      0    1    2    3    4    5    6   7   8
     --------------------------------------------
     | -4 | 17 |    |    |    | 5  | 7 | 9 | 15 |      size = 6
     --------------------------------------------  capacity = 9
                                ^--front

      0    1    2    3    4    5    6   7   8
     --------------------------------------------
     | 5  | 7  | 9  | 15 |-4 | 17  |    |   |    |      size = 6
     --------------------------------------------  capacity = 9
      ^--front

public class Queue {
  /** The queue size and the index of front element. */
  int size, front;
  /** The data container. */
  Object[] queue;

  /**
   * Creates an empty queue of given initial capacity.
   * @param initialCapacity the initial capacity of this queue.
   */
  public Queue(int initialCapacity) {
    queue = new initialCapacity;
    size = 0;
    front = 0;
  }

  /**
   * Resizes the underlying circular array to have given capacity.
   * @param newCapacity the new capacity of the underlying array.
   */
  public void resize(int newCapacity) {
    // TODO
  }

  /**
   * Adds the specified element the end of this queue.
   * @param elem the element to add to the end of this queue.
  public void enqueue(Object elem) {
    // TODO
  }

  /**
   * Removes the front element of this queue, and returns it.
   * @return the object that was at the front of this queue.
   * @throws EmptyQueueException if the queue is empty.
  public Object dequeue() throws EmptyQueueException {
    // TODO
  }
}
```

(a) (10 pts) Write Queue.resize().
(b) (10 pts) Write Queue.enqueue(). Note that the circular array container is automatically resized when capacity is exhausted.
(c) (10 pts) Write Queue.dequeue().

4. (30 pts) Stack using a circular doubly-linked list
Assume you are using a doubly-linked list with a dummy tail to implement a Stack. You're free to choose where to add the new stack elements (to the beginning or the end of the list). You are also free to add new instance variables as needed.

```
public class Stack {

  /**
   * The nodes in the linked list.
   */
  public class Node {
    public Object element;
    public Node next;
    public Node prev;
    /**
     * Creates a new node.
     */
    public Node(Object e, Node n, Node p) {
      element = e;
      next = n;
      prev = p;
    }
  }

  // The reference to the dummy tail.
  private Node tail;

  /**
   * Creates an empty stack.
   */
  public Stack() {
    // Create the dummy tail, and make it circular.
    tail = new Node("dummy", null, null);
    tail.next = tail.prev = tail;
  }

  /**
   * Pushes a new element on this stack.
   * @param elem the element to push on this stack.
   */
  public void push(Object elem) {
    // TODO
  }

  /**
   * Pops the top of stack from the stack.
   * @return the element that was on top before it was removed.
   * @exception EmptyStackException if the stack is empty.
   */
  public Object pop() throws EmptyStackException {
    // TODO
  }

  /**
   * Returns an array representation of the stack, where the first element
   * is the top of stack.
   * @return an array representation of the stack elements.
   */
  public Object[] toArray() {
    // TODO
  }
}
```

(a) (10 pts) Write the Stack.push() method.
(b) (10 pts) Write the Stack.pop() method.
(c) (10 pts) Write the Stack.toArray() method.

```
CSE 220 Fall 2011                       ID   : ____________________
Quiz 1, Sep 27, 2011                    Name : ____________________
Marks: ____ out of 10
```

Let's say we have a class called Array, which is a resizable container using Java's built-in array. It has a set of operations that we've already seen - add, remove, contains, indexOf, toString, toArray, etc. We now want another one - countHowMany - which counts how many instances of the specified element is in the array.

```
    Array a = new Array();                    // Array of objects.
    a.add("xy"); a.add("ab"); a.add("xy"); a.add("mn"); a.add("xy");
    System.out.println("Array a = " + a);
    System.out.println("Number of \"xy\" = " + a.countHowMany("xy"));
```

should print:

```
    Array a = [ "xy" "ab" "xy" "mn" "xy" ]
    Number of "xy" = 3
```

```
class Array {
  private Object[] data;        // the underlying data array
  private int      size;        // the number of elements in this array
  /**
   * Returns the number of occurrences of specified element in this Array
   * @param e the element to find the number of occurrences of
   * @return the number of occurrences (>= 0) of specified element
   */
  public int countHowMany(Object e) {







  }
  /**
   * Removes the element at the given position in this Array
   * @param index the position of the element to remove
   * @return the element if it was successfully removed, or null otherwise
   *         (if the position is invalid for example)
   */
  public Object remove(int index) {

















  }
}
```

```
CSE 220 Fall 2011                       ID   : ____________________
Quiz 1, Sep 27, 2011                    Name : ____________________
Marks: ____ out of 10
```

Let's say we have a class called Array, which is a resizable container using Java's built-in array. It has a set of operations that we've already seen - add, remove, contains, indexOf, toString, toArray, etc. We now want another one - countHowMany - which counts how many instances of the specified element is in the array.

```
    Array a = new Array();                    // Array of objects.
    a.add("xy"); a.add("ab"); a.add("xy"); a.add("mn"); a.add("xy");
    System.out.println("Array a = " + a);
    System.out.println("Number of \"xy\" = " + a.countHowMany("xy"));
```

should print:

```
    Array a = [ "xy" "ab" "xy" "mn" "xy" ]
    Number of "xy" = 3
```

```
class Array {
  private Object[] data;        // the underlying data array
  private int      size;        // the number of elements in this array
  /**
   * Returns the number of occurrences of specified element in this Array
   * @param e the element to find the number of occurrences of
   * @return the number of occurrences (>= 0) of specified element
   */
  public int countHowMany(Object e) {
      int numOccur = 0;
      for (int i = 0; i < size; i++) {
          if (e.equals(data[i]))
              ++numOccur;
      }
      return numOccur;
  }
  /**
   * Removes the element at the given position in this Array
   * @param index the position of the element to remove
   * @return the element if it was successfully removed, or null otherwise
   *         (if the position is invalid for example)
   */
  public Object remove(int index) {
      Object oldElement = null;
      if (index >= 0 && index < size)
          // Save a reference to the element to remove at given position
          oldElement = data[index];

          // Shift elements at positions [index+1,...,size-1] left by
          // one to fill the gap left by removing the element at specified
          // index.
```

```
            for (int i = index; i < size - 1; i++)
                data[i] = data[i + 1];

            // (Optional) Null out the unused slot at the end to help the GC
            data[size - 1] = null;

            // Update number of elements currently in array
            size--;
        }
        return oldElement;
    }
  }
```

```
CSE 220 Spring 2011                     ID   : ____________________
Quiz 1, Jan 23, 2011                    Name : ____________________
Marks: ____ out of 10
```

We have to design a Student class, where a Student's properties are "ID" (integer) and "name" (String), and want to override the equals() method and also implement the Comparable interface (which means that we have to implement the compareTo() method).

For equality, two Student instances s1 and s2 are equal if and only if both the IDs and names are the same. For comparison, we first check the student ID. If the IDs are the same, then the comparison is based on the name fields. Since String class is Comparable, we can use String's compareTo() method.

```
  public class Student {
    public int ID;
    public String name;
    public Student(int ID, String name) {
      this.ID = ID;
      this.name = name;
    }

    /**
     * Returns true if this Student is equal to the given one.
     * @param o reference to the other student
     * @return true if equal, false otherwise.
     */
    public boolean equals(Object o) { // TODO
      // Optimization - if this and 'o' refer to the same object, they must
      // be equal.
      if (this == o)
        return true;
      // Enhancement - if 'o' is not a Student, or if 'o' is null,
      // they must not be equal.
      if (o == null || !(o instanceof Student))
        return false;

      // Now we actually compare the two objects! But 'o' doesn't refer
      // specifically to a Student, so we must first cast it.
      Student other = (Student) o;
      return ID == other.ID && name.equals(other.name);

      // Another way to write the return statement above is:
      // if (ID == other.ID && name.equals(other.name))
      //   return true;
      // else
      //   return false;
    }
```

```
/**
 * Returns 0 if this Student is equal to the given Student, a negative
 *   value if smaller than the given one, and positive value if larger.
 *   The checking is done first on the IDs, and then the names (if the
 *   IDs are equal).
 * @param o reference to the other student
 * @return 0 if equal, < 0 if smaller or > 0 if larger.
 */
public int compareTo(Object o) { // TODO
  // Optimization - if this and 'o' refer to the same object, they must
  // be equal.
  if (this == o)
    return 0;
  // Note: unlike the case of equals() method, we do not try to handle
  // the case of o == null and o not an instance of Student.

  // Now we actually compare the two objects! But 'o' doesn't refer
  // specifically to a Student, so we must first cast it.
  Student other = (Student) o;
  // First check the IDs, and compare the names only if the IDs are
  // equal.
  if (ID < other.ID)
    return -1;
  else if (ID > other.ID)
    return +1;
  else
    return name.compareTo(other.name);
}
```

```
CSE 220 Summer 2011                     ID   : ____________________
Quiz 1, May 29, 2011                    Name : ____________________
Marks: ____ out of 10
```

Given a resizable Array class, we want to be able to remove elements. We have to find the element by sequentially searching for it using the indexOf() method, and if the index is valid, remove it from the array. You will write the indexOf() and the remove() methods.

```
    Array a = new Array();                    // Array of Strings.
    a.add("xy"); a.add("ab"); a.add("xy"); a.add("mn"); a.add("xy");
    System.out.println("Array a = " + a);
    a.remove("xy"); System.out.println("after removing \"xy\" = " + a);
```

should print:

```
    Array a = [ "xy" "ab" "xy" "mn" "xy" ]
    after removing "xy" = [ "ab" "xy" "mn" "xy" ]
```

```
class Array {
  private Object[] data;        // the underlying data array
  private int      size;        // the number of elements in this array
  /**
   * Find the index of the given element in this array.
   * @param e the element to find the index of
   * @return the index if found, or -1 otherwise
   */
  public int indexOf(Object e) {
      // We sequentially search the array, keeping track of the current
      // index, to find the element within the array.
      for (int i = 0; i < size; i++)
          if (data[i].equals(e))
              return i;

      // If we reach here, then the element was not in the array, so
      // return a sentinel for invalid index.
      return -1;
  }

  /**
   * Removes the given element from this Array.
   * @param e the element to remove, if it exists
   * @return true if the element was removed, false otherwise
   */
  public boolean remove(Object e) {
      // Use the indexOf() method!
      int index = indexOf(e);

      // If not found, return false right away.
      if (index < 0)
          return false;
      // Since the array is not sorted, or at least the order of the
      // elements within it does not matter, we can remove it in two ways:
      // 1. the hard, but always correct, way - shift all the elements
      //    from [index+1,...,size-1] left by one position to fill the
      //    "hole" left by the removed element; or
      // 2. the easy way, which only works if the element ordering does
      //    not matter - simply move the element in the last slot into
      //    the "hole" left by removed element. No shifting required!
      // Either way, we decrease the size by one, so the last slot is
      // considered empty after this.
      // We'll do it the 1st way since it's always going to give you
      // the correct answer (the element ordering in the array does
      // matter at times, eg., when the elements are sorted).

      // Shift all the subsequent elements one position to the left.
      for (int i = index; i < size - 1; i++)
          data[i] = data[i + 1];

      // Help the GC by null'ing out the unused last slot
      data[size - 1] = null;

      // and declare the array smaller by one element!
      size--;

      return true;
  }
}
```

```
CSE 220 Summer 2012                     ID   : ____________________
Quiz 1, June 03, 2012                   Name : ____________________
Marks: ____ out of 10
```

Given a resizable Array class, we want to be able to find elements, and shift left or right by k positions. Write contains() and shiftLeft() methods.

```
    Array a = new Array();                    // Array of Strings.
    a.add("xy"); a.add("ab"); a.add("xy"); a.add("mn"); a.add("xy");
    System.out.println("Array a = " + a);
    println("\"xy\" in the array? " + a.contains("xy"));
    a.shiftLeft(2);
    println("after shifting left by 2, a = " + a);
```

should print:

```
    Array a = [ "xy" "ab" "xy" "mn" "xy" ]
    "xy" in the array? true
    after shifting left by 2, a = [ "xy" "mn" "xy" ]
```

```
class Array {
  private Object[] data;        // the underlying data array
  private int      size;        // the number of elements in this array
  /**
   * Returns true if the given element is in this array.
   * @param e the element to find
   * @return true if found, false otherwise
   */
  public boolean contains(Object e) {
      for (int i = 0; i < size; i++)
          if (e.equals(data[i]))
              return true;

      return false;
  }

  /**
   * Shifts the elements of this array left by k positions.
   * @param k the number of positions to shift left by
   * @throws IllegalShiftException if k < 0 or k > size
   */
  public void shiftLeft(int k) {
      if (k < 0 || k > size)
          throw new IllegalShiftException();

      for (int i = k; i < size; i++)
          data[i - k] = data[i];

      // help garbage collector by null'ing out the last k positions
      for (int i = size - k; i < size; i++)
```

```
          data[i] = null;

      size = size - k;
  }
}
```

```
CSE 220 Summer 2012                     ID   : ____________________
Quiz 1, June 03, 2012                   Name : ____________________
Marks: ____ out of 10
```

Given a resizable Array class, we want to be able to find elements, and shift left or right by k positions. Write contains() and shiftLeft() methods.

```
    Array a = new Array();                      // Array of Strings.
    a.add("xy"); a.add("ab"); a.add("xy"); a.add("mn"); a.add("xy");
    System.out.println("Array a = " + a);
    println("\"xy\" in the array? " + a.contains("xy"));
    a.shiftLeft(2);
    println("after shifting left by 2, a = " + a);
```

should print:

```
    Array a = [ "xy" "ab" "xy" "mn" "xy" ]
    "xy" in the array? true
    after shifting left by 2, a = [ "xy" "mn" "xy" ]
```

```
class Array {
  private Object[] data;        // the underlying data array
  private int      size;        // the number of elements in this array
  /**
   * Returns true if the given element is in this array.
   * @param e the element to find
   * @return true if found, false otherwise
   */
  public boolean contains(Object e) {
  }
  /**
   * Shifts the elements of this array left by k positions.
   * @param k the number of positions to shift left by
   * @throws IllegalShiftException if k < 0 or k > size
   */
  public void shiftLeft(int k) {











  }
}
```

```
CSE 220 Fall 2001                       ID   : ____________________
Quiz 2, Oct 11, 2011                    Name : ____________________
Marks: ____ out of 10
```

You are given the following doubly-linked Node class:

```
private static class Node {
    public Object elem;
    public Node   next;
    public Node   prev;
    public Node(Object e, Node n, Node p)
        { elem = e; next = n; prev = p; }
}
```

Which is being used in your dummy-headed doubly-linked circular list:

```
public class LinkedList {
    private Node header;        // dummy head node of this list
    private int  size;          // number of elements in this list
    // ...
}
```

1. [5 pts] Write a method to find the index of the given element.

```
  /**
   * Finds the index of the given element, if it exists, in this list.
   * @param e element to find the index of in the given list
   * @return the element's index if found, or -1 otherwise
   */
  public int indexOf(Object e) {











  }
```

2. [5 pts] Write a method to remove the given node from this list.

```
/**
 * Removes the given node from this list.
 * @param node reference to the node to remove from this list
 * @return the element within the node just removed
 */
private Object removeNode(Node node) {









}
```

```
CSE 220 Fall 2001                       ID   : ____________________
Quiz 2, Oct 11, 2011                    Name : ____________________
Marks: ____ out of 10

You are given the following doubly-linked Node class:
private static class Node {
    public Object elem;
    public Node   next;
    public Node   prev;
    public Node(Object e, Node n, Node p)
        { elem = e; next = n; prev = p; }
}

Which is being used in your dummy-headed doubly-linked circular list:
public class LinkedList {
    private Node header;        // dummy head node of this list
    private int  size;          // number of elements in this list
    // ...
}

1. [5 pts] Write a method to find the index of the given element.
  /**
   * Finds the index of the given element, if it exists, in this list.
   * @param e element to find the index of in the given list
   * @return the element's index if found, or -1 otherwise
   */
  public int indexOf(Object e) {
      int i = 0;
      for (Node n = header.next; n != header; n = n.next, i++) {
          if (e.equals(n.elem))
              return i;
      }
      return -1;        // not found.
  }

2. [5 pts] Write a method to remove the given node from this list.
  /**
   * Removes the given node from this list.
   * @param node reference to the node to remove from this list
   * @return the element within the node just removed
   */
  private Object removeNode(Node node) {
      Object ret = node.elem;
      Node p = node.prev;
      Node q = node.next;

      p.next = q;
      q.prev = p;
      node.next = node.prev = null;
      return ret;
  }
```

```
CSE 220 Spring 2011                    ID   : ____________________
Quiz 2, Feb 01, 2011                   Name : ____________________
Marks: ____ out of 10

1. (10 pts) Given the Array class in Assignment 1, implement the remove()
   method, and add a reverse() method. Assume that you're given indexOf().

    Array a = new Array();                    // Array of Strings.
    a.add("xy"); a.add("ab"); a.add("xy"); a.add("mn"); a.add("xy");
    System.out.println("Array a = " + a);
    a.remove("xy"); System.out.println("after removing \"xy\" = " + a);
    a.reverse(); System.out.println("after reversing = " + a);

should print:
    Array a = [ "xy" "ab" "xy" "mn" "xy" ]
    after removing "xy" = [ "ab" "xy" "mn" "xy" ]
    after reversing = [ "xy" "mn" "xy" "ab" ]

class Array {
  private Object[] data;
  private int size;

  public int indexOf(Object e);

  /**
   * Removes the given element, if it exists, from this Array.
   * @param e the element to remove.
   * @return true if it was removed, false otherwise.
   */
  public boolean remove(Object e) {
    // Get the location of the object within this array first.
    int index = indexOf(e);
    if (index < 0)
      return false;                      // Does not exist!

    // Now shift all elements in the range [index + 1 ... size - 1] left
    // by one position to fill the gap left by the given element.
    for (int i = index; i < size - 1; i++)
      data[i] = data[i + 1];

    // Set the last position (now unused) to null to help GC
    data[size - 1] = null;

    // Adjust size and return.
    size--;
    return true;
  }
  /**
   * Reverses the elements of this Array in-place.
   */
  public void reverse() {
    // Use two indices - left and right - that meet halfway.
    int l = 0;
    int r = size - 1;

    while (l < r) {
      // Exchange the elements at positions l and r.
      Object tmp = data[l];
      data[l] = data[r];
      data[r] = tmp;

      // Now move the indices towards the middle.
      l++;
      r--;
    }
  }
}
```

```
CSE 220 Summer 2011                      ID   : ____________________
Quiz 2, Jun 07, 2011                     Name : ____________________
Marks: ____ out of 10
```

You are given the following singly-linked Node class:

```
private static class Node {
  public Object e;
  public Node n;
  public Node(Object e, Node n) { this.e = e; this.n = n; }
}
```

Write the following static methods to manipulate a linked list:

```
/**
 * Appends the specified element at the end of the specified list.
 * @param head reference to first node of the list
 * @param e    reference to the element to append to the list
 * @return reference to the head (may be modified)
 */
public static Node append(Node head, Object e) {
    // Create the new node to append to the list
    Node newNode = new Node(e, null);

    // Special case - the specified list may be empty!
    if (head == null)
        head = newNode;         // note: this changes head!
    else {
        // Scan the list for the tail node first, and then append
        Node tail = head;
        while (tail.n != null)
            tail = tail.n;

        // Now append
        tail.n = newNode;
    }

    // Return the head reference, which may have changed
    return head;
}

/**
 * Returns a copy of the specified list.
 * @param head reference to the first node in the list
 * @return the reference to the copy of the specified list
 */
public static Node copy(Node head) {
    // Create an empty list, and then iterate over the elements of the
    // specified list, appending each node to the new list. Keep a
    // tail reference so that it's easy to append.
    Node h = null;
    Node t = null;

    while (head != null) {
        // Create a copy of the current node
        Node newNode = new Node(head.e, null);

        // Noew append it
        // Special case - the copy is empty the first time
        if (h == null) {
            h = newNode;
            t = newNode;
        } else {
            // it's not empty, so use tail to append it quickly.
            t.n = newNode;
            t = newNode;
        }

        // Advance the original list
        head = head.n;
    }

    // Return reference to the copy list's head.
    return h;
}
```

```
CSE 220 Summer 2012                     ID   : ____________________
Quiz 2, Jun 19, 2012                    Name : ____________________
Marks: ____ out of 10
```

You are given the following head-referenced singly-linked list class:

```
public class List {
    private Node head;  // reference to first node
    private int  size;  // number of nodes in this list

    private static class Node {
        public Object e;
        public Node   n;
        public Node(Object e, Node n) { this.e = e; this.n = n; }
    }
    public void insertFirst(Object e) { ... }
    public Object removeLast() { ... }
}

/**
 * Inserts the specified element in the beginning of this list.
 * @param e element to insert in the beginning of this list
 */
public void insertFirst(Object e) { // TODO
    // Create the new "standalone" node first
    Node n = new Node(e, null);

    // Now insert in the beginning. Note that head changes!
    n.n = head;

    // You could "combine" the two statements above this way:
    // Node n = new Node(e, head);

    // Now make head point the new node.
    head = n;

    size++;
}

/**
 * Removes the last element of this list.
 * @return the element removed, or null if list is empty.
 */
public Object removeLast() { // TODO
    // If the list is empty, we're done!
    if (head == null) // or equivalently, if (size == 0)
        return null;

    // Find the predecessor, which is the node *before* the tail. We use
    // two different references to make our life easier.
    Node p = null;
    Node tail = head;
    while (tail.n != null) {
        p = tail;
        tail = tail.n;
    }

    // There are two possibilities now:
    // (1) list has a single element, so p is null and tail is head; or
    // (2) p refers to the tail node and tail is null
    // The first one is a special case. If we remove the last node from a
    // list with a single element, it becomes empty (head changes).

    if (p == null)
        // Special case -- a single element in the list
        head = null;
    else {
        // Now tail refers to the node to remove, and p refers
        // to the predecessor
        p.n = null;
    }

    size--;
    return tail.e;
}
```

```
CSE 220 Summer 2012                     ID   : ____________________
Quiz 2, Jun 19, 2012                    Name : ____________________
Marks: ____ out of 10
```

You are given the following head-referenced singly-linked list class:

```
public class List {
    private Node head;  // reference to first node
    private int  size;  // number of nodes in this list

    private static class Node {
        public Object e;
        public Node   n;
        public Node(Object e, Node n) { this.e = e; this.n = n; }
    }
    public void insertFirst(Object e) { ... }
    public Object removeLast() { ... }
}

/**
 * Inserts the specified element in the beginning of this list.
 * @param e element to insert in the beginning of this list
 */
public void insertFirst(Object e) { // TODO













}
```

```
/**
 * Removes the last element of this list.
 * @return the element removed, or null if list is empty.
 */
public Object removeLast() { // TODO
















}
```

```
CSE 220 Fall 2011                       ID   : ____________________
Quiz 3, Oct 25, 2011                    Name : ____________________
Marks: ____ out of 10
```

1. We have an a cyclic-array based Queue implementation. Complete remove().

```
public class Queue {
  private Object[] data;        // The underlying array container
  private int size;             // Number of items in the queue
  private int front;            // Index of the front item
  /**
   * Returns the offset the given item relative to the front item if it
   * exists, or -1 otherwise.
   * @param item the desired item
   * @return distance from front if found, -1 otherwise
   */
  public int search(Object item);       // ALREADY DONE FOR YOU.
  /**
   * Removes the given item from this queue, if it exists.
   * @param item the item to remove
   * @return true if removed, false otherwise
   */
  public boolean remove(Object item) {
  }
}
```

2. Use binary search to find the given key in a sorted array.

```
/**
 * Finds the index of the given key in the sorted array.
 * @param data the sorted array of elements to search in
 * @param key  the key to find the index of, if it exists
 * @return the index if found, or -1 otherwise.
 */
static public int binarySearch(Object[] data, Object key) {









}
```

```
CSE 220 Fall 2011                       ID   : ____________________
Quiz 3, Oct 25, 2011                    Name : ____________________
Marks: ____ out of 10

1. We have an a cyclic-array based Queue implementation. Complete remove().
public class Queue {
  private Object[] data;        // The underlying array container
  private int size;             // Number of items in the queue
  private int front;            // Index of the front item
  /**
   * Returns the offset the given item relative to the front item if it
   * exists, or -1 otherwise.
   * @param item the desired item
   * @return distance from front if found, -1 otherwise
   */
  public int search(Object item);       // ALREADY DONE FOR YOU.
  /**
   * Removes the given item from this queue, if it exists.
   * @param item the item to remove
   * @return true if removed, false otherwise
   */
  public boolean remove(Object item) {
      // Find the offset of the given item, if it exists, relative to
      // front.
      int offset = search(item);
      if (offset == -1)
          return false;

      // Find the actual index given the offset.
      int i = (front + offset) % data.length;

      // Now remove the item by shifting all subsequent queue elements
      // left by one position.
      for (int j = offset; j < size - 1; j++) {
          int k = (i + 1) % data.length;
          data[i] = data[k];
          i = k;
      }
      // Set the last (now unused) slot to be null to help GC.
      data[(front + size - 1) % data.length] = null;

      // Decrement size and return the removed item.
      size--;
      return true;
  }
}

2. Use binary search to find the given key in a sorted array.

/**
```

```
 * Finds the index of the given key in the sorted array.
 * @param data the sorted array of elements to search in
 * @param key  the key to find the index of, if it exists
 * @return the index if found, or -1 otherwise.
 */
static public int binarySearch(Object[] data, Object key) {
    int l = 0;
    int r = data.length - 1;    // size equals capacity here.
    while (l <= r) {
        int mid = (l + r)/2;
        int compare = ((Comparable) key).compareTo(data[mid]);
        if (compare == 0)
            return mid;
        else if (compare < 0)   // key < data[mid]
            r = mid - 1;
        else
            l = mid + 1;
    }
    return -1;                  // not found.
}
```

```
CSE 220 Spring 2011                     ID   : ____________________
Quiz 3, Feb 08, 2011                    Name : ____________________
Marks: ____ out of 10
```

You are given the following singly-linked Node class:

```
private static class Node {
  public Object e;
  public Node n;
  public Node(Object e, Node n) { this.e = e; this.n = n; }
}
```

Write the following methods to manipulate a linked list:

```
/**
 * Adds the element after the given node.
 * @param e reference to the element to add.
 * @param p reference to the node after which to add the element.
 * @return reference to the new node just added.
 */
public static Node addAfter(Object e, Node p) {
  // Create a new node to add
  Node newNode = new Node(e, null);

  // Attach newNode to the list first
  newNode.next = p.next;

  // Now attach the list to the new node
  p.next = newNode;

  return newNode;
}

/**
 * Adds the element before the given node.
 * @param e    reference to the element to add.
 * @param q    reference to the node before which to add the element.
 * @param head reference to the head node.
 * @return the reference to the head node (will change if q == head).
 */
public static Node addBefore(Object e, Node q, Node head) {

  // In the general case, we first iterate from the head, and find the
  // node *before* q (find the predecessor that is), and then use
  // the addAfter method.
  // There is a special case however - if we're adding before the head
  // (ie., q == head), then head changes. Handle the special case first.

  Node newNode = null;

  // Special case first - adding before the head changes head!
  if (q == head) {
    newNode = new Node(e, null);
    newNode.next = head;
    head = newNode;
  } else {
    // find the predecessor node
    Node p = head;
    while (p.next != q)
      p = p.next;

    // We could create a new node with "e" in it, and add after node p, but
    // we can of course use the addAfter() method which is already done!
    newNode = addAfter(e, p);
  }

  return head;
}
```

```
CSE 220 Summer 2011                     ID   : ____________________
Quiz 3, Jun 14, 2011                    Name : ____________________
Marks: ____ out of 10
```

Implement the Stack.push and Stack.pop methods in a Stack implementation using a dummy-head referenced doubly-linked circular list.

```
public class Stack {
  private static class Node {
    Object e; Node n; Node p;
    public Node(Object e1, Node n1, Node p1) { e = e1; n = n1; p = p1; }
  }
  private Node head;             // The reference to the dummy head node.
  private int size;              // The number of elements in the stack.
  /**
   * Pushes a new element on this stack.
   * @param item the new element to push on this stack.
   */
  public void push(Object item) {
      // Create a new node to put the item in
      Node node = new Node(item, null, null);

      // Link it in the beginning of the list
      node.n = head.next;
      node.p = head;
      head.n = node;
      node.n.p = node;

      size++;
  }

  /**
   * Pops this stack and returns the element that was on the top.
   * @return the element that was on top of this stack.
   * @exception EmptyStackException if this stack is empty.
   */
  public Object pop() {
      if (size == 0)             // alternatively: head.next == head
          throw new EmptyStackException();

      Node node = head.n;
      head.n = node.n;
      head.n.p = head;

      size--;

      node.next = null;
      node.prev = null;
      return node.e;
  }
}
```

```
CSE 220 Summer 2012                        ID   : ____________________
Quiz 3, Jul 15, 2012                       Name : ____________________
Marks: ____ out of 10
```

You need to find the number of odd integers in an array of integers (with at least 1 element) using recursion.
There are two ways to do this:
(1) linear recursion - find the number of odd integers in the rest of the array, and add 1 or 0 if the current element is odd or not.
(2) binary recursion - find the number of odd integers in the left and right halves of the array (recursively), and add the two.

1. Write a recursive static method to return the result using way #1.

```
/**
 * Finds the number of odd integers in array of integers
 * @param a reference to the array of integers
 * @param l the left boundary
 * @param r the right boundary
 * @return the number of odd integers
 */
public static int countOdd(int[] a, int l, int r) {
    if (l > r)
        return 0;
    else {
        // Find the number of odd integers in the rest of the array
        int numOdd = countOdd(a, l + 1, r);
        // And add one if the current one is also odd
        if (a[l] % 2 != 0)
            numOdd = numOdd + 1;

        return numOdd;
    }
}
```

2. Write a recursive static method to return the result using way #2.

```
/**
 * Finds the number of odd integers in array of integers
 * @param a reference to the array of integers
 * @param l the left boundary
 * @param r the right boundary
 * @return the number of odd integers
 */
public static int countOdd(int[] a, int l, int r) {
    if (l > r)
        return 0;
    else if (l == r) {
        if (a[l] % 2 == 0)
            return 0;
        else
            return 1;
    } else {
        int mid = l + (r - l)/2;
        int numOddLeft = countOdd(a, l, mid);
        int numOddRight = countOdd(a, mid + 1, r);
        return numOddLeft + numOddRight;
    }
}
```

```
CSE 220 Summer 2012                     ID   : ____________________
Quiz 3, Jul 15, 2012                    Name : ____________________
Marks: ____ out of 10
```

You need to find the number of odd integers in an array of integers (with at least 1 element) using recursion.
There are two ways to do this:

(1) linear recursion - find the number of odd integers in the rest of the array, and add 1 or 0 if the current element is odd or not.

(2) binary recursion - find the number of odd integers in the left and right halves of the array (recursively), and add the two.

1. Write a recursive static method to return the result using way #1.

```
/**
 * Finds the number of odd integers in array of integers
 * @param a reference to the array of integers
 * @param l the left boundary
 * @param r the right boundary
 * @return the number of odd integers
 */
public static int countOdd(int[] a, int l, int r) {









}
```

2. Write a recursive static method to return the result using way #2.

```
/**
 * Finds the number of odd integers in array of integers
 * @param a reference to the array of integers
 * @param l the left boundary
 * @param r the right boundary
 * @return the number of odd integers
 */
public static int countOdd(int[] a, int l, int r) {









}
```

```
CSE 220 Fall 2011                       ID   : ____________________
Quiz 4, Nov 22, 2011                    Name : ____________________
Marks: ____ out of 10
```

1. Write a recursive static method to return the number of occurrences of a key in a singly-linked list.

```
/**
 * Finds the number of occurrences of key in the given list
 * @param list reference to the head node of the list
 * @param key  the number of occurrences of the key to find
 * @return the number of occurrences of key in the given list
 */
public static int countKey(Node list, Object key) {
    if (list == null)
        return 0;
    else {
        int match = key.equals(list.element) ? 1 : 0;
        return match + countKey(list.next, key);
    }
}
```

2. Write a recursive static method to return the number of occurrences of a key in a binary tree. Assume "left" and "right" instance variables in the Node class.

```
/**
 * Finds the number of occurrences of key in the given binary tree
 * @param root reference to the root node of the binary tree
 * @param key  the number of occurrences of the key to find
 * @return the number of occurrences of key in the given binary tree
 */
public static int countKey(Node root, Object key) {
    if (root == null)
        return 0;
    else {
        int match = key.equals(root.element) ? 1 : 0;
        return match
               + countKey(root.left, key)
               + countKey(root.right, key);
    }
}
```

```
CSE 220 Spring 2011                   ID   : ____________________
Quiz 4, Feb 15, 2011                  Name : ____________________
Marks: ____ out of 10
```

1. [4 pts] We looked at different design choices when pushing/popping elements in a stack when using an array or a linked list container.
(a) When using an array, should a push put the element in the beginning of an array or at the end? Justify in one or two sentences.

ANS: It's easier to add to and remove from the end of the array (no shifting!), so push should put the new element at the end. The top of the stack is then simply "size-1".

(b) When using a linked list, should a push put the element in the beginning of a list or at the end? Justify in one or two sentences.

ANS: It's easier to add to and remove from the beginning of the list, so push should put the new element at the beginning of the list. We can add to the end easily as well, using a tail reference for example, but removing from the end is much harder.

2. [6 pts] Implement the Stack.push and Stack.pop methods in a Stack implementation that uses a singly-linked list.

```
public class Stack {
  private static class Node {
    Object e; Node n;
    public Node(Object e, Node n) { this.e = e; this.n = n; }
  }
  private Node head;            // The reference to the head node.
  private int size;             // The number of elements in the stack.
  /**
   * Pushes a new element on this stack.
   * @param item the new element to push on this stack.
   */
  public void push(Object item) {
    // Create a new node to put the item in.
    Node newNode = new Node(item, null);
    // Now add to the beginning. Note: no special cases here, even
    // if stack is empty.
    newNode.n = head;
    head = newNode;
    size++;
  }

  /**
   * Pops this stack and returns the element that was on the top.
   * @return the element that was on top of this stack.
   * @exception EmptyStackException if this stack is empty.
   */
  public Object pop() {
    // Throw exception if trying to pop an empty stack!
    if (size == 0)              // equivalently: head == null
      throw new EmptyStackException();

    // Save a reference to the current head, and then advance head.
    Node oldHead = head;
    head = head.n;

    size--;

    // Save a reference to the item to return.
    Object ret = oldHead.e;

    // Detach old head to help GC.
    oldHead.e = null;
    oldHead.n = null;

    return ret;
  }
}
```

```
CSE 220 Summer 2011                     ID   : ____________________
Quiz 4, Jun 21, 2011                    Name : ____________________
Marks: ____ out of 10

We have an *bounded* cyclic-array based Queue implementation.
public class Queue {
  private Object[] data;        // The underlying array container
  private int size;             // Number of items in the queue
  private int front;            // Index of the front item
  /**
   * Adds the given item to the back of the queue.
   * @param item the item to add to the back of the queue
   * @exception QueueOverflowException if the queue is full
   */
  public void enqueue(Object item) throws QueueOverflowException {
    // Note that this is a bounded queue, so check for overflow
    if (size == data.length)
      throw new QueueOverflowException();

    // Put the item in the next available position
    int nextPos = (front + size) % data.length;
    data[nextPos] = item;
    size++;
  }

  /**
   * Removes the given item from this queue.
   * @param item the item to remove
   * @return true if removed, false otherwise
   */
  public boolean remove(Object item) {
    // Need to find it first, otherwise return false. Use one variable to
    // index into the circular array, and the other to control the loop.
    int j = front;
    int i = 0;
    for (i = 0; i < size; i++) {
      if (item.equals(data[j]) {
        break;
      }
    }
    if (i == size)              // did not find it
      return false;

    // Now that we know both the position and the offset (relative to
    // front) of the item within the circular array, we can remove it
    // by shifting the subsequent items to the left. Start by figuring
    // out how many to shift, and then start from where "j" is right
    // now.
    int numToShift = size - offset - 1;
    int to = j;
    for (i = 0; i < numToShift; i++) {
      int from = (j + 1) % data.length;
      data[to] = data[from];
      from = to;
    }

    // Help GC
    data[from] = null;

    size--;
    return true;
  }

  // If you didn't want to manipulate the circular array, there is also
  // an out-of-place solution that uses an auxiliary queue.

  /**
   * Removes the given item from this queue.
   * @param item the item to remove
   * @return true if removed, false otherwise
   */
  public boolean remove(Object item) {
    Queue auxQueue = new Queue();
    // Add all the items, except for the first one that matches the
    // specified one, to the aux queue. Then move those back to this
    // one.
    boolean removed = false;
    while (!isEmpty()) {
      Object t = dequeue();
      if (!removed && item.equals(t))
        removed = true;
      else
        auxQueue.enqueue(t);
    }

    // auxQueue now has all but the first occurrence of the specified item.
    // Now we move those back to this queue.
    while (!auxQueue.isEmpty())
      enqueue(auxQueue.dequeue());

    return removed;
  }
}
```

```
CSE 220 Summer 2012                     ID   : ____________________
Quiz 4, Aug 7, 2012                     Name : ____________________
Marks: ____ out of 10
```

1. Write a recursive static method that checks whether the given key is in the binary tree. Note: not a binary search tree, just a binary tree. Assume "left", "right", "element" instance variables in the Node class.

```
/**
 * Returns true if the key is in the binary tree.
 * @param tree reference to the root node of the binary tree
 * @param key  the key to search for in the tree
 * @return true if found, false otherwise
 */
public static boolean contains(Node tree, Object key) {
    if (tree == null)
        return false;
    else if (key.equals(tree.element))
        return true;
    else return contains(tree.left, key)
         || contains(tree.right, key);
}
```

2. Write a recursive static method that checks whether the given key is in the binary search tree using binary search tree algorithm.

```
/**
 * Returns true if the key is in the binary search tree.
 * @param tree reference to the root node of the binary search tree
 * @param key  the key to search for in the binary search tree
 * @return true if found, false otherwise
 */
public static boolean contains(Node tree, Object key) {
    if (tree == null)
        return false;
    else {
        int compare = ((Comparable) key).compareTo(tree.element);
        if (compare == 0)
            return true;
        else if (compare > 0)
            return contains(tree.right, key);
        else
            return contains(tree.left, key)
    }
}
```

```
CSE 220 Summer 2012                     ID   : ____________________
Quiz 4, Aug 7, 2012                     Name : ____________________
Marks: ____ out of 10
```

1. Write a recursive static method that checks whether the given key is in the binary tree. Note: not a binary search tree, just a binary tree. Assume "left", "right", "element" instance variables in the Node class.

```
/**
 * Returns true if the key is in the binary tree.
 * @param tree reference to the root node of the binary tree
 * @param key  the key to search for in the tree
 * @return true if found, false otherwise
 */
public static boolean contains(Node tree, Object key) {









}
```

2. Write a recursive static method that checks whether the given key is in the binary search tree using binary search tree algorithm.

```
/**
 * Returns true if the key is in the binary search tree.
 * @param tree reference to the root node of the binary search tree
 * @param key  the key to search for in the binary search tree
 * @return true if found, false otherwise
 */
public static boolean contains(Node tree, Object key) {




}
```

```
CSE 220 Fall 2011                       ID   : ____________________
Quiz 5, Dec 4, 2011                     Name : ____________________
Marks: ____ out of 10
```

1. Write a recursive static method that checks whether the given key is in the binary tree. Note: not a binary search tree, just a binary tree. Assume "left" and "right" instance variables in the Node class.

```java
/**
 * Returns true if the key is in the binary tree.
 * @param tree reference to the root node of the binary tree
 * @param key  the key to search for in the tree
 * @return true if found, false otherwise
 */
public static boolean contains(Node tree, Object key) {
    if (tree == null)
        return false;
    else if (key.equals(tree.element))
        return true;
    else return contains(tree.left, key)
         || contains(tree.right, key);
}
```

2. Write a recursive static method that checks whether the given key is in the binary search tree using binary search tree algorithm.

```java
/**
 * Returns true if the key is in the binary search tree.
 * @param tree reference to the root node of the binary search tree
 * @param key  the key to search for in the binary search tree
 * @return true if found, false otherwise
 */
public static boolean contains(Node tree, Object key) {
    if (tree == null)
        return false;
    else {
        int compare = ((Comparable) key).compareTo(tree.element);
        if (compare == 0)
            return true;
        else if (compare > 0)
            return contains(tree.right, key);
        else
            return contains(tree.left, key)
    }
}
```

```
CSE 220 Spring 2011                    ID   : ____________________
Quiz 5, Feb 22, 2011                   Name : ____________________
Marks: ____ out of 10
```

1. [2 pts] Why do we use a cyclic array to implement array-based queue? (in one or two sentences at most).

ANS: A linear array would run out of space in the back, even if there's space available in the front. Cyclic queue avoid this problem by wrapping around, utilizing all available space in the array.

2. [8 pts] We have an auto-resizable cyclic-array based Queue implementation.

```
public class Queue {
  private Object[] data;        // The underlying array container.
  private int size;             // Number of elements in the queue.
  private int front;            // Index of the front element.
  /**
   * Adds the given element to the back of the queue.
   * @param e the element to add to the back of the queue.
   */
  public void enqueue(Object e) {
    // Resize if needed. Note that front is reset to 0 after resizing.
    if (size == data.length) {
      Object[] oldData = data;
      data = new Object[size * 2];
      int k = front;
      for (int i = 0; i < size; i++) {
        data[i] = oldData[front];
        k = (k + 1) % oldData.length;
      }
      front = 0;                // Don't forget to reset front!
      oldData = null;           // Help GC
    }

    // Now add it to the next available slot at the end.
    int nextPos = (front + size) % data.length;
    data[nextPos] = e;
    size++;
  }

  /**
   * Reverses the order of the elements in this queue.
   */
  public void reverse() {
    // The trick is to learn how to iterate backwards in a cyclic array.
    // You cannot use the modulus ("%") operator when moving backwards!
    int l = front;
    int r = (front + size - 1) % data.length;
    // The second trick is to learn that you cannot use "l < r" as the
    // loop condition!
    int i = 0;
    while (i < size/2) {                // Exchange only half the elements.
      Object tmp = data[l];
      data[l] = data[r];
      data[r] = tmp;
      // Count the number of elements being copied.
      i++;
      // Can wrap using % when going forwards.
      l = (l + 1) % data.length;
      // Must cycle explicitly when going backwards.
      r = r--;
      if (r < 0)
        r = data.length - 1;
    }
  }
}
```

```
CSE 220 Summer 2011                     ID   : ____________________
Quiz 5, Jul 16, 2011                    Name : ____________________
Marks: ____ out of 10
```

1. Write a recursive (static) method to return the *position* of the minimum key in an array.

```
/**
 * Finds the index of the minimum key in data[l..r]
 * @param data the array of keys
 * @param l    the left boundary of data to consider
 * @param r    the right boundary of data to consider
 * @return the index of the minimum key in the array
 */
public static int findIndexOfMin(Object[] data, int l, int r) {
    if (l >= r)
        return l;
    else {
        int minIndex = l;
        int minIndexRest = findIndexOfMin(data, l + 1, r);
        if (((Comparable) data[minIndexRest]).compareTo(data[l]) < 0)
            minIndex = minIndexRest;

        return minIndex;
    }
}
```

We can also formulate the solution using binary recursion, which splits the array into half, finds the position of the minimum of each half, then finds the overall minimum from that.

```
public static int findIndexOfMin(Object[] data, int l, int r) {
    if (l >= r)
        return l;
    else {

        int mid = (l + r)/2;
        int minIndexLeft = findIndexOfMin(data, l, mid);
        int minIndex = minIndexLeft;
        int minIndexRight = findIndexOfMin(data, mid + 1, r);
        if (((Comparable) data[minIndexRight]).compareTo(data
[minIndexLeft]) < 0)
            minIndex = minIndexRight;

        return minIndex;
    }
}
```

2. Selection sort can be made recursive with the following observation: to sort an array of keys, swap the minimum key with the leftmost key,

and sort the rest of the array. It stops when there is either 1 or 0 keys in the array. You can now use the findIndexOfMin from question 1 to implement a recursive version of selection sort.

```java
/**
 * Sorts data[l..r] using selection sort
 * @param data the array of keys to sort
 * @param l    the left boundary of data to consider
 * @param r    the right boundary of data to consider
 */
public static void selectionSort(Object[] data, int l, int r) {
    if (l >= r)
        return;
    else {
        int minIndex = findIndexOfMin(data, l, r);
        Object tmp = data[l];
        data[l] = data[minIndex];
        data[minIndex] = tmp;

        sort(data, l + 1, r);
    }
}
```

```
CSE 220 Spring 2011                     ID   : ____________________
Quiz 6, Mar 15, 2011                    Name : ____________________
Marks: ____ out of 10
```

1. [3 pts] What value does method mystery return when called with a value of 4 (ie., mystery(4))? Explain with a recursion tree.

```
  static int mystery(int n) {
      if (n <= 1)
          return 1;
      else
          return n * mystery(n - 1);
  }
```

SOLUTION:

```
  mystery(4) = 4 * mystery(3)
               |           3  * mystery(2)
               |           |            2  * mystery(1)
               |           |            |            1
               |           |            |            |
               4 *         3  *         2  *         1  = 24
```

It's obviously an implementation of the factorial function!

2. [7 pts] The recursive form of sequential/linear search in a linked list is the following: if the element in the first node matches the value, return true; else find the element in the rest of the list.

a) [3 pts] Write down the base case(s) and the recursive case(s) for this algorithm. You can use the mathematical notation we used in class. YOU WILL NOT CREDIT FOR PART (b) IF YOU DO NOT COMPLELETE PART (a).

SOLUTION:

```
                          -
                          | false,  if list is empty
                          | true, if the list's node value equals value
    find(list,val) =      |
                          | otherwise, find the value in the rest of the
                          | list
                          -
```

Many other ways to write the same. For example,

```
                          -
                          | if list == null, return false
                          | if list.val == val, return true
    find(list,val) =      |
                          | otherwise, return find(list.next,val)
                          -
```

b) [4 pts] Write the recursive function (you must complete part(a) of this question to credit for this part):

```
public class Node {
    public Object val;
    public Node next;
}
/**
 * Returns true if "what" is found in the list referenced by "n".
 * @param n    the reference to first node (null if empty list).
 * @param what the value to find.
 * @return true if value is found, false otherwise.
public static boolean find(Node n, Object what) {
    if (n == null)
        return false;
    else if (n.val.equals(what))
        return true;
    else
        return find(n.next, what);
}
```

CSE 220 Summer 2011   ID   : ____________________
Quiz 6, Jul 31, 2011   Name : ____________________
Marks: ____ out of 10

1. [5 pts] Given the following class for a binary tree node:

```
   public class Node {
       public Object element;
       public Node left;
       public Node right;
   }
```

You are to find how many times (that is the number of occurrences of) a given element in the given tree. Each node the contains the given element contributes one to the total. An empty tree contributes 0 of course.

```
   /**
    * Returns the number of occurrences of the given element in the tree
    * @param n the root of the binary (sub)tree
    * @param e the element for which to find the number of occurrences of
    * @return the number of occurrences of the element in the tree
    */
   public static int countHowMany(Node n, Object e) {
       // TODO
   }
```

SOLUTION:
NOTE: This is NOT a binary search tree, just a binary tree.

This is exactly the same problem as counting the number of occurrences of an element in a sequence:
- Recursively: If the current element matches the given one, set its contribution to 1, otherwise 0. Add to this the contribution from the *rest of the sequence*, and return the total.
- Iteratively: iterate over the sequence, counting the occurrence of the given element, and return the total.

Here we have a tree instead of a sequence, so there are a few small superficial differences:
- Recursively: the "rest of the tree" consists of multiple branches, instead of the just the rest of the sequence. For a binary tree, the rest means the left subtree AND the right subtree.
- Iteratively: we "traverse" instead of iterate

Let's try the recursive solution first:

```
   /**
    * Returns the number of occurrences of the given element in the tree
    * @param n the root of the binary (sub)tree
    * @param e the element for which to find the number of occurrences of
```

```
 * @return the number of occurrences of the element in the tree
 */
public static int countHowMany(Node n, Object e) {
    if (n == null)
        return 0;
    else {
        int nOccur = 0;
        // If the node has the element, add 1
        if (e.equals(n.element))
            nOccur = nOccur + 1;

        // Add the occurrences in the left subtree
        nOccur = nOccur + countHowMany(n.left, e);
        // and the right subtree
        nOccur = nOccur + countHowMany(n.right, e);

        return nOccur;

        // An equivalent "one-liner" is:
        // return (e.equals(n.element) ? 0 : 1)
        //        + countHowMany(n.left, e)
        //        + countHowMany(n.right, e);
    }
}
```

Note that this really is a version of Pre-order traversal, where we "visit" self first (to check whether it's element matches or not), and then visit left and right subtrees in that order.

We *could* use pre-order traversal directly, but need a "helper" method to update a global count variable. Here's a complete class that does that.

```
public class Tree {
    private static class Node {
        public Object element;
        public Node left;
        public Node right;
        public Node(Object e, Node l, Node r) {
            element = e; left = l; right = r;
        }
        public Node(Object e) {
            this(e, null, null);
        }
    }

    private static class Counter {
        public int count = 0;
        public Counter(int count) {
           this.count = count;
        }
        public Counter() {
           this(0);
        }
    }

    private static void visit(Node n, Object e, Counter count) {
        if (e.equals(n.element))
           count.count++;
    }

    private static void preOrderVisit(Node n, Object e, Counter count) {
        if (n == null)
           return;
        else {
           visit(n, e, count);
           preOrderVisit(n.left, e, count);
           preOrderVisit(n.right, e, count);
        }
    }

    public static int countHowMany(Node n, Object e) {
        Counter count = new Counter(0);
        preOrderVisit(n, e, count);
        return count.count;
    }

    public static void main(String[] args) {
        Node r = new Node("x",
          new Node("b", new Node("foo"), null),
          new Node("y", new Node("x"), new Node("foo")));

        System.out.println("count(foo) = " + countHowMany(r, "foo"));
        System.out.println("count(x) = " + countHowMany(r, "x"));
        System.out.println("count(b) = " + countHowMany(r, "b"));
    }
}
```

How about an iterative solution? We have to iterate over the nodes of the binary tree using one of the standard traversals, and count the number of occurrences of the given element. Let's use the level order traversal first.

```
/**
 * Returns the number of occurrences of the given element in the tree
 * @param n the root of the binary (sub)tree
 * @param e the element for which to find the number of occurrences of
 * @return the number of occurrences of the element in the tree
 */
public static int countHowMany(Node n, Object e) {
    if (n == null)
        return 0;            // empty tree => no occurrences, done

    Queue q = new Queue();
    q.enqueue(n);

    int count = 0;
    while (!q.isEmpty()) {
        Node self = (Node) q.dequeue();
        if (e.equals(self.element))
            count++;

        // enqueue node's children to get to later
        if (self.left != null)
            q.enqueue(self.left);
        if (self.right != null)
            q.enqueue(self.right);
   }

   return count;
}
```

What if we change the Queue to a Stack?

```
/**
 * Returns the number of occurrences of the given element in the tree
 * @param n the root of the binary (sub)tree
 * @param e the element for which to find the number of occurrences of
 * @return the number of occurrences of the element in the tree
 */
public static int countHowMany(Node n, Object e) {
    if (n == null)
        return 0;            // empty tree => no occurrences, done

    Stack s = new Stack();
    s.push(n);

    int count = 0;
    while (!s.isEmpty()) {
        Node self = (Node) s.pop();
        if (e.equals(self.element))
            count++;

        // push node's children to get to later
        if (self.left != null)
            s.push(self.left);
        if (self.right != null)
            s.push(self.right);
   }
```

```
      return count;
   }
```

We have pre-order traversal! Basically, we just implemented recursion using our own Stack instead of letting Java do it for us!

2. [5 pts] What's the maximum number of comparisons as a function of N to find a key in a perfectly balanced binary search tree of N keys (internal nodes)?

   The maximum number of comparisons for any binary search tree is h + 1, where h is the height of the tree. We need to find the relationship between the height h and the number of nodes N.

   A perfectly balanced binary search tree has all the levels filled, so we must have:

```
    N = 2^0 + 2^1 + ... + 2^h = 2^{h+1} - 1
      = 2^{h+1} - 1
    N + 1 = 2^{h+1}
    log_2(N + 1) = h + 1
    h = log_2(N + 1) - 1
    h + 1 = log_2(N + 1)
```

   Since h is an integer, we must have:

```
    h + 1 = ceil(log_2(N + 1)
```

   So the maximum number of comparisons to find an element in a perfectly balanced binary search tree is ceil(log_2(N + 1)).

   The other way is to note that a perfectly balanced binary tree has all the external nodes, N + 1 of those, at level h + 1. In any binary tree, the maximum number of nodes at any level x is 2^x, and since all levels are filled in a perfectly balanced binary tree, it has the maximum
number
   of nodes at each level.

```
    N + 1 = 2^{h+1}
    h + 1 = log_(N + 1)
```

   We have the same answer as before.

```
CSE 220 Spring 2011                    ID   : ____________________
Quiz 7, Mar 22, 2011                   Name : ____________________
Marks: ____ out of 10
```

1. [4 pts] You are to write an iterative method to copy a linked list. You're given:

```java
public class Node {
  public Object element;
  public Node next;
  public Node(Object e, Node n) { element = e; next = n; }
}
```

The iterative algorithm to copy a linked list works like this: keep a head and tail for the new list (initially null); for each node in the given list (iterate of course), create a new node with the given node's element, and add this new node to the *end* of the new list.
YOU MUST NOT MODIFY THE GIVEN LIST!

```java
/**
 * Returns a copy of the given linked list. Must be iterative.
 * @param n the reference to the given list.
 * @return the reference to the list copy.
 */
public static Node copy(Node n) {
    Node newList = null;
    Node tail = null;
    while (n != null) {
        Node newNode = new Node(n.element);
        if (newList == null)
            newList = tail = newNode;
        else {
            tail.next = newNode;
            tail = newNode;
        }
        n = n.next;
    }
    return newList;
}
```

2. [6 pts] You are to write a recursive method to copy a linked list.

The recursive algorithm to copy a linked list works like this: if it's an empty list, return an empty list (the base case). Otherwise, create a new node with the element in the first node of the given list, and make it's next refer to a copy of the rest of the given list (note the recursion). YOU MUST NOT MODIFY THE GIVEN LIST!

```java
/**
 * Returns a copy of the given linked list. Must be recusive.
 * @param n the reference to the given list.
 * @return the reference to the list copy.
 */
public static Node copy(Node n) {
    if (n == null)
        return null;
    else {
        Node h = new Node(n.element, null);
        h.next = copy(n.next);
        return h;
        // Note: we could shorten the whole block to a single statement:
        // return new Node(n.element, copy(n.next));
    }
}
```

```
CSE 220 Spring 2011                    ID   : ____________________
Quiz 8, Mar 29, 2011                   Name : ____________________
Marks: ____ out of 10
```

1. [2.5 pts] Given the following binary tree (external nodes are not shown):

```
         M
       /   \      Pre  : M Q K T V P N F X Y
      Q     P
     / \   / \    In   : K T Q V M F N P Y X
    K   V N  X
     \   /  /     Post : T K V Q F N Y X P M
      T F  Y
```

(a) Show the pre-, in- and post-order traversals (next to the tree).
(b) What is the leftmost node?   Leftmost node = K_____
(c) What is the rightmost node?   Rightmost node = X_____

2. [3.0 pts] Derive the expressions for the maximum and minimum heights of a 2-ary (binary) tree.

We're using the book's definition of height, which defines the height of an empty tree to be -1.

The tallest tree is when all the nodes are arranged in a linear fashion, either all leaning left or all leaning right. If there are N internal nodes, then the maximum height is N-1.

```
h <= N - 1
```

The shortest tree is when all the levels are filled, with the exception of possibly the last level. If all the levels are filled, then we must have all N+1 external nodes at level h+1. The maximum number of nodes at level h+1 in a binary tree is 2^{h+1}, so we must have:

```
N + 1 <= 2^{h+1}
h + 1 >= log_2(N + 1)
h >= log_2(N + 1) - 1
```

Now we have the bounds on the height, namely it's approximately between N and log_2(N).

```
log_2(N + 1) - 1 <= h <= N - 1
```

3. [4.5 pts] Given the following class for a binary tree node:

```
public class Node {
    public Object element;
    public Node left;
    public Node right;
}
```

You are to find if an element exists in a tree. Return true if it does, or false otherwise. If the tree is empty, return false. If not, look in the root of the tree, and return true if found. If not, search the left subtree, and return true if found. Otherwise, search in the right subtree and return whatever that returns.

```
/**
 * Returns true if the tree rooted at n contains the given element.
 * @param n the root of the (sub)tree.
 * @param e the element to find.
 * @return true if found, false otherwise.
 */
public static boolean contains(Node n, Object e) {
    // Do an exhaustive search, using pre-order traversal.
    if (n == null)
        return false;
    else {
        if (n.element.equals(e))
            return true;
        else
            return contains(n.left, e) || contains(n.right, e);
    }
}
```

```
CSE 220 Spring 2011                     ID   : ____________________
Quiz 9, Apr 5, 2011                     Name : ____________________
Marks: ____ out of 10
```

1. [3 pts] Insert the following sequence in an initially-empty binary search tree, and show the resulting binary search tree. Assume that we're using <= and > relations for the left and right subtrees.

          13 2 4 17 9 7 18 2 13

   After inserting 13, 2, 4, and 17:

```
     13
   /   \
  2    17
   \
    4
```

   Now after inserting 9, 7, and 18:

```
     13
   /   \
  2    17
   \     \
    4    18
     \
      9
     /
    7
```

   and the final tree after inserting the remaining 2, and 13:

```
     13
   /   \
  2    17
 / \     \
2   4    18
     \
      9
     / \
    7  13
```

2. [3 pts] Consider this binary search tree:

```
     13
   /   \
  2     19
 / \   / \
1   5 15  21
   /   \   \
  4    17  33
```

Remove the root node, and draw the resulting binary search tree.

If we choose to use the predecessor, we exchange 13 with 5, and remove the node with the key 5 in it:

```
     5
   /   \
  2     19
 / \   / \
1   4 15  21
       \   \
       17  33
```

If we choose to use the successor, we exchange 13 with 15, and remove the node with the key 15 in it:

```
     15
   /   \
  2     19
 / \   / \
1   5 17  21
   /       \
  4        33
```

3. [4 pts] Consider the following binary search tree class, where all the methods are already done for you, except for the static sort() method. Write the sort() method that uses a binary search tree to sort the given keys.

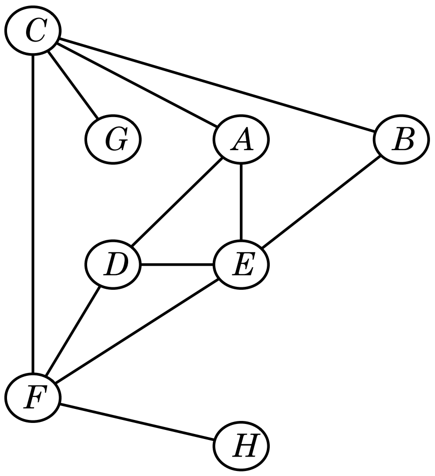
```
public class BST {
    // The reference to the root node of the tree, null if empty
    private Node root;
    // The number of nodes in the tree
    private int size;

    // Constructs a new empty binary search tree.
    public BST();
    // Returns the number of keys in this binary search tree.
    public int size();
    // Adds the given key to this binary search tree.
    public void add(Object key);
    // Finds the leftmost node of the tree rooted at the given node.
    private static Node findLeftMost(Node node);
    // Finds the successor of the given node.
    private static Node findSuccessor(Node node);
    /**
     * Sorts the elements in the given array.
     * @param data the array whose elements are to be sorted.
     */
    public static void sort(Object[] data) {
        // Add the sequence to a new binary search tree.
        BST tree = new BST();
        for (Object key : data)
            tree.add(key);

        // Now extract the elements in sorted order using
        // in-order traversal. Easiest way is to start from
        // the leftmost, and iterate using successor.
        Node n = BST.findLeftMost(tree.root);
        for (int i = 0; i < data.length; i++) {
            data[i] = n.element;
            n = BST.findSuccessor(n);
        }
        return data;
    }
}
```

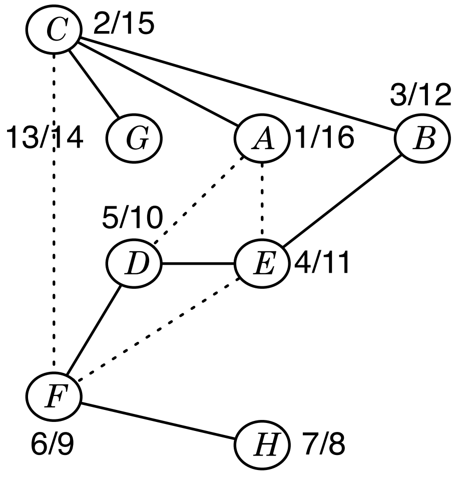
CSE 220 Spring 2011	Name : ______________
Quiz 10, Apr 12, 2011	ID : ______________
Marks: __ out of 10	

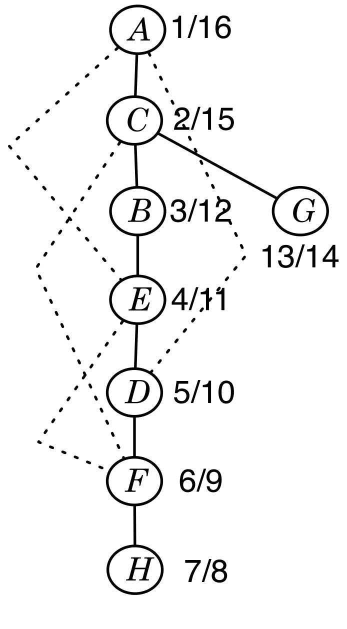
Problem 1. [**6 pts**] Given the following graph G:



a. Show the DFS tree starting at vertex A. Mark the tree edges with solid and non-tree edges with dashed lines. Annotate each vertex with the discovery and finish times. Visit adjacent vertices in lexicographic (alphabetical) order. [5 pts]

The DFS tree is shown below. The tree on the right is drawn to be more "tree-like".





b. Is there a cycle in G? Answer using the DFS tree/forest. [1 pts]

The presence of back-edges during DFS prove that G has at least one cycle.

Problem 2. [4 pts]

a. Given the following sequence, what are the steps to search any key using key-indexed search? [2.5 pts]

9	-5	10	-3	9	10
0	1	2	3	4	5

The first step is to compute the minimum/maximum keys, and the range: $min = -5$, $max = 10$, $range = 15$. Since there are negative values, we must shift the keys right such that the minimum is a valid index, so $\delta = +5$. We now have the shifted keys, shown below:

14	0	15	2	14	15
0	1	2	3	4	5

At this point, we create the intermediate array of length *range*, and insert the keys one by one.

1	0	1	0	0	0	0	0	0	0	0	0	0	0	2	2
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now to search for a key, we first add δ to the key, and then see if the element at the index is 0 (not found), or positive (found).

b. Given the problem size n and range of the input k, show the worst-case space and time cost for setting up the intermediate array, and then searching for a key. [1.5 pts]

The space cost for the intermediate array is k. To setup the intermediate array ,we have to iterate through the n input keys, and do constant work for each key, so the it costs approximately n operations. The search for a key is done in constant time. For sorting, we have to iterate through the intermediate array (k elements), and output the elements in sorted order. That means that sorting requires additional k operations.

Space cost	k
Time cost for setup	n
Searching cost	constant
Sorting cost	k