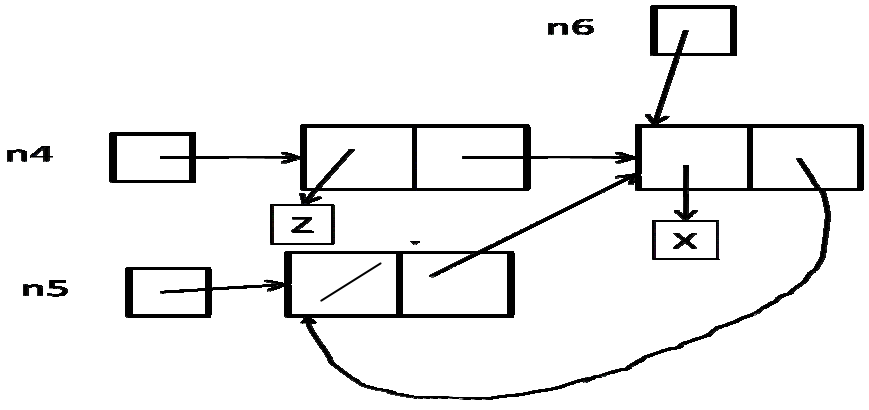
CS307 Spring 2011 Midterm 2 Solution and Grading Criteria.

Grading acronyms:
ABA - Answer by Accident
AIOBE - Array Index out of Bounds Exception may occur
BOD - Benefit of the Doubt. Not certain code works, but, can't prove otherwise
ECF - Error carried forward.
Gacky or Gack - Code very hard to understand even though it works or solution is not elegant. (Generally no points off for this.)
GCE - Gross Conceptual Error. Did not answer the question asked or showed fundamental misunderstanding
LE - Logic error in code.
NAP - No answer provided. No answer given on test
NN - Not necessary. Code is unneeded. Generally no points off
NPE - Null Pointer Exception may occur
OBOE - Off by one error. Calculation is off by one.

1. Answer as shown or -2 unless question allows partial credit.
No points off for differences in spacing, capitalization, commas, or braces
On Big O questions, okay if O() left off and just function

A. 30
B. 26
C. BBDBBBACCCECC
D. 21
E. $O(N^2)$
F. O(1)
G. 15N + 5 range of +/- 2 on each Coefficient
H. $O(N^4)$ (recall, linked list get is O(N))
I. $O(N^2)$
J. 2.5 seconds
K. 160 seconds
L. $O(N^3)$
M. 10,000 seconds (10,000 - 10,240 or $10 * 2^{10}$)
N. -7 (Z and X in nodes okay)

O.



2. Comments. I thought this would an easy question. We did a large number of array based list methods in class and lots of old midterms had array based list questions. At its core it was a simple array question. There wasn't even another list involved. Surprisingly a lot of students had problems with this question.

Common problems:
- using methods that were not allowed on the question
- not nulling out the elements in the unused portions of the array
- not calculating indices correctly
- solution not O(1) space as required

Suggested Solution:

```
public void removeFrontPortion(int num) {
    assert 0 <= num && num <= size();
    // shift references for remaining elements
    int frontIndex = 0;
    for(int i = num; i < size; i++) {
        container[frontIndex] = container[i];
        frontIndex++;
    }

    // null out unused portion of list
    for(int i = size - num; i < size; i++)
        container[i] = null;

    size = size - num;
}
```

General Grading Criteria: 15 points

shift elements to front of array: attempt: 3, correct 6. (can lose partial points on this)
null out elements that are no longer part of list: attempt 2, correct 2(can lose partial points on this)
update size: attempt: 1, correct 1

not O(1) space - 7
calling method not allowed -6

3. Comments: did you do assignment 8? If so, this should have been an easy question. Do you know how to use iterators? Students, generally did well on this question.

Common problems:
- determining size of other.difference(this) instead of this.difference(other)
- not using iterators correctly.
- misunderstanding what the difference of two sets is
- using compareTo instead of equals. (objects were not guaranteed to be comparable)

Suggested Solution:

```
public int sizeOfDifference(ISet<E> other) {
    int result = 0;
    Iterator<E> thisIt = iterator();
    while(thisIt.hasNext()) {
        E temp = thisIt.next();
        Iterator<E> otherIt = other.iterator();
        boolean found = false;
        while(!found && otherIt.hasNext())
            found = otherIt.next().equals(temp);
        if(!found)
            result++;
    }
    return result;
}
```

General Grading Criteria: 15 points

(partial credit possible on items worth more than 1)
get iterator for this set: 1 point
loop correctly with iterator through all elements of this set: 4
store thisIt.next() in temp for use correctly: 1 point
get iterator for other set, must be new one each time: 1 point
use other iterator to check if temp element in other set correctly: 3 points
check if temp equals otherIt.next: 1 point
track number of elements in difference of this set and other, and update correctly: 3 points
return value: 1 point

4. Comments. Students did relatively well on this question. A good linked list and object based question. Also we have done the equals method in class and on several assignments so I thought the standard stuff would be easy. (Check not null, check if other is same data type, cast to LinkedList if it is a LinkedList.)

Common problems:
- not verifying other != null and not verifying other is a LinkedList using the instanceof operator
- not casting other to a LinkedList
- not checking sizes of lists. If they are not the same size they cannot be equal
- not setting up temp nodes and moving through the list correctly
- not stopping search as soon as a mismatch occurs between two elements of list

Suggested Solution

```
public boolean equals(Object other) {
    boolean result = other instanceof LinkedList;
    if(result && this != other) {
        // safe to cast now
        LinkedList otherList = (LinkedList) other;
        // check sizes the same
        result = size == otherList.size;
        if(result) {
            // now have to check elements!
            Node tempInThis = first;
            Node tempInOther = otherList.first;
            while(result && tempInThis != null) {
              result = tempInThis.getData().equals(tempInOther.getData());
              tempInThis = tempInThis.getNext();
              tempInOther = tempInOther.getNext();
            }
        }
    }
    return result;
}
```

General Grading criteria: 20 points
return false if other null: 1 point
return false it other not a LinkedList: 1 point (getClass() okay)
corrent casting: 1 point
check sizes of two lists equal and if not return false and return true if this == other: 3 points
Temp nodes to move through both lists: attempt, 1 points, correct 2 points
loop with correct check for null at end: attempt 2 points, correct 2 points
stop loop as soon as unmatched objects found: 3 points
access data from node: 1 point
call equals on data: 1 point (safe to assume no nulls as data in list)
advance both temp nodes correctly: 5 points
return statement, with correct answer: 1 point

5. Comments. Not too bad of a problem. A lot like flowing off the map with a catch. Students did fairly well on this question

Common problems:
- Not marking a square that has been visited. If this is not done then two open squares can lead to an infinite loops. Consider: from square 1 I can move north to square 2, from square 2 I can move south to square 1, from square 1 I can move north to square 2, from square 2 I can move south to square 1, and so forth until you run out of stack space. Some people handled this by sending a String or character showing where they had come from. This works because the sail boat can't go west and circle around. That would not work if the boat could have gone west.
- not using the return value in any way, just calling canSail and not using the return value is pointless
- using loops to check all cells. This demonstrates a huge misconception. From the current cell the boast call only reach at most 3 other cells
- not checking the bounds to ensure the cell is in bounds

Suggested Solution

```
public static boolean canSail(char[][] map, int startRow, int startCol,
       int destinationRow, int destinationCol) {

   // if out of bounds can't reach, col won't ever be < 0
   if(startRow < 0 || startRow == map.length || startCol == map[0].length)
       return false;

   // if destination or current is rock, can't reach.
   if(map[destinationRow][destinationCol] == '%'
           || map[startRow][startCol] == '%')
       return  false;

   // check if destination is to the west, can't reach
   if(destinationCol < startCol)
       return false;

   // success base case
   if(startRow == destinationRow && startCol == destinationCol)
       return true;

   // On a cell that is legal, but not destination. Need to make sure
   // we don't end up in a loop back and forth into cell. So mark as rock
   map[startRow][startCol] = '%';

   // try the 3 directions
   return canSail(map, startRow, startCol + 1, destinationRow, destinationCol)
      || canSail(map, startRow - 1, startCol, destinationRow, destinationCol)
       || canSail(map, startRow + 1, startCol, destinationRow, destinationCol);
}
```

General Grading criteria: 20 points
check failure base cases. out of bounds, current rock (or prevent), or can't reach (columns)
    2 each (can avoid base case if code prevents), 6 total
success base case: 4 points

recursive case: prevents infinite loop between cells by marking: 3 points
recursive step: check all 3 directions if necessary: 7 points (-5 early return)