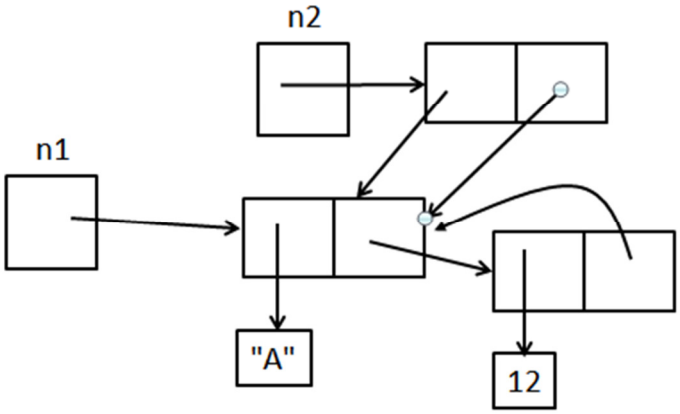
CS314 Fall 2012 Midterm 2 Solution and Grading Criteria.

Grading acronyms:
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.
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, and braces

A. Sorted in asciibetical order. (Alphabetical, Lexigrahical okay, or words to that effect)
B. HashMap
C. So constructors in sub classes can call them to initialize data, or words to that effect
D. O(N)
E. O(N^2)



F.

G. 9

H. runtime error or stack overflow error or infinite loop

I. abbabbabbabb

J. 46

K. Must show work

16,000,000 * x = 32,0000,000 log2 32,000,000 +
x * log2 32,000,000

16,000,000 * x – x *25 = 32,000,000 * 25

16,000,000 * x = 32,000,000 * 25

x = 2 * 25 = 50

about 50 searches. (25 okay as well if use 32,000,000x)

L. 44 seconds

M. insertion sort

N. -18

O. 6810 (Spaces in between okay)

EXTRA CREDIT ANSWER: OVERWORKED

2. Comments. Not algorithmically difficult, just working with several data structures. Two and three pass O(N) algorithms were okay. We saw a couple of tidy single loop solutions.

Common problems: Some people had very inefficient solutions that tried to dump all the references to an ArrayList and search for equality pairwise. O(N^2)

Suggested Solution:

```
public static ArrayList<String> getMoviesWithMaxReferences
        (Map<String, ArrayList<String>> movieRefs) {

    HashMap<String, Integer> counts = new HashMap<String, Integer>();
    for(String movie : movieRefs.keySet())
        for(String ref : movieRefs.get(movie))
            if(counts.containsKey(ref))
                counts.put(ref, counts.get(ref) + 1);
            else
                counts.put(ref, 1);

    ArrayList<String> result = new ArrayList<String>();
    int max = 0;
    for(String movie : counts.keySet()) {
        int count = counts.get(movie);
        if(count > max) {
            max = count;
            result.clear();
            result.add(movie);
        }
        else if(count == max)
            result.add(movie);
    }
    return result;
}
```

General Grading Criteria: 18 points

- find frequency of references for given movie - 6 total
  attempt: 2 points
  correct: 4 points
- find maximum number of references (may be done as separate step) 6 total
  attempt: 2 points
  correct: 4 points
- create resulting array list: 1 point
- find movie(s) with max references and add to result correctly: 3 points
- return correct answer (local var or multiple returns) 1 point
- Many errors were -2:
  - try to instantiate iterator (new Iterator())
  - attempt to iterator through map, instead of map keyset
  - calling non existent methods

3. Comments: I thought this would be an easy problem, but a lot of people simply didn't understand how to approach it. The approach was to push opening symbols onto the stack. When a closing symbol is seen check if the stack is empty or if the symbol doesn't match. If never returned null, then just dump matching symbols to the resulting String. No need to use a StringBuilder.

Common problems:

- A lot of people used the Stack to reverse the String which really doesn't make anything easier. The purpose of the Stack was to hold opening symbols NOT to reverse the String.
- Not checking stack empty when see closing symbol

Suggested Solution:

```
    public static String getMissingGroupingSymbols(String str) {
        Stack<Character> symbols = new Stack<Character>();
        for(int i = 0; i < str.length(); i++) {
            char ch = str.charAt(i);
            // if opening symbol push
            if(isOpening(ch);
                symbols.push(ch);
            else
                // must be a closing symbol!
                if(symbols.isEmpty() || !matches(ch, symbols.top())
                   // no matching symbol or incorrect match
                    return null;
                else
                    // matches, remove symbol and throw away
                    symbols.pop();
        }
        String result = "";
        while(!symbols.isEmpty()) {
             result += getMatch(symbols.top());
            symbols.pop();
        }
        return result;
    }

public static boolean matches(char ch1, char ch2) {
     return ch1 == ']' ? ch2 == '[' : ch2 == '('; }

public static boolean isOpening(char ch) {return ch == '[' || '('; }

public static char getMatch(char ch) { return ch == '(' ? ')' : ']'; }
```

General Grading Criteria: 17 points

- loop through chars in String: 2 points
- if opening symbol, push: 3 points
- if closing symbol and stack empty, return null: 2 points
- if closing symbol and top symbol doesn't match return null: 2 points
- if closing symbol and top matches, pop: 2 points
- create result by emptying stack: 4 points
- concat correct closing symbol: 1 point
- return result: 1 point

4. Comments: Students did very well on this question. It required tracking multiple nodes in the list and moving through it. The one special case was if the list was empty.

Common problems: Not moving through list. Off by one errors. Creating unnecessary node objects. Possible null pointer exceptions. Not handling the empty case correctly. Using disallowed methods

Suggested Solution:
the trailer solution

```
public void removeRepeats() {
        if(first != null) {
            Node<E> trailer = first;
            Node<E> lead = first.getNext();
            while(lead != null) {
                if(lead.getData().equals(trailer.getData()))
                    trailer.setNext(lead.getNext());
                else
                    trailer = lead;
                lead = lead.getNext();
            }
        }
    }
```

using look ahead, very tidy

```
    public void removeRepeats() {
        if(first != null) {
            Node<E> temp = first;
            while(temp.getNext() != null)
                if(temp.getData().equals(temp.getNext().getData()))
                    temp.setNext(temp.getNext().getNext());
                else
                    temp = temp.getNext();
        }
    }
```

General Grading Criteria: 17 points

Handle empty list case correctly: 2 points
loop: 1 point
stopping condition on loop correct: 3 points
compare data in consecutive nodes correctly: 3 points (== instead of .equals -> -2)
if equal remove node correctly: 4 points (partial credit possible)
move / traverse temporary references correctly: 4 points

5. Comments: Classic recursive backtracking problem. Really not that different than the flowing off the map problem.

Common problems:

- Not using return values correctly
- early return
- not ensuring that cells are not reused (rule of the game)
- changing board (tempBoard = theBoard does not prevent changes, recall pointers!)

Suggested Solution:

```
public boolean isPresent(String word, int row, int col) {
     if(word.length() == 0)
          return true; // success base case
     else if(!inbounds(row, col) || board[row][col] != word.charAt(0))
          return false; // failure base cases
     else {
          // recursive case, deal with current letter / cell
          char oldChar = theBoard[row][col];
          theBoard[row][col] = '*'; // won't match
          String newWord = newWord.substring(1);
          // check surrounding cells
          for(int r = row - 1; r <= row + 1; r++)
               for(int c = col - 1; c <= col + 1; c++)
                    if(isPresent(word, r, c,)) {
                         // solved!
                         theBoard[row][col] = oldChar;
                         return true;
                    }
          // tried all cells, no solution
          theBoard[row][col] = oldChar;
          return false;
     }
}
```

General Grading criteria: 18 points

- base case if out of bounds, or guard against, 2 points
- base case completed correctly, 3 points
- base case char does not match, 2 points
- recursive case, check all surrounding cells, 3 points
- make recursive calls correctly with new parameters(string, row, col) in context of surrounding cells, 4 points
- if solved correctly return true, 2 points
- return failure if all surrounding cells checked, 2 points
- Penatlies: board altered -4, efficiency (stop when not possible) -3, early return -7, don't reuse cells in same word, -4,