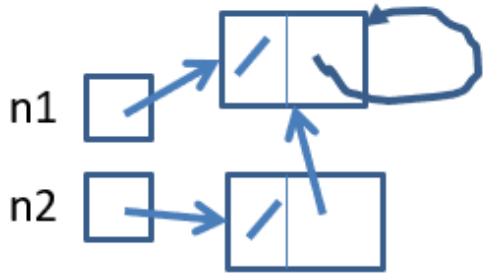
CS314 Spring 2022 Exam 2 Solution and Grading Criteria.

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.
Gacky or Gack - Code very hard to understand even though it works. (Solution is not elegant.)
LE - Logic Error in code.
MCE - Major Conceptual Error. Answer is way off base, question not understood.
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.
RTQ - Read The question. Violated restrictions or made incorrect assumption.

1. Answer as shown or -2 unless question allows partial credit.
No points off for minor differences in spacing, capitalization, commas, and brace unless noted.
Text in parenthesis not required. It is simply grading guidance and / or a brief explanation for answer.

A. -4
B. 9753*----
C. 15
D. 13
E. {A=5, C=2, S=25} (diff in { okay)
F. best: O(N) worst: O(NlogN) (base 2 okay, LinkedList get is O(N))
G. 100 seconds or 102.4 seconds. Method is $O(2^N)$
H. O(N)
I. $O(N^2)$ (removing the last element of a singly linked list. Even with last reference, must update that reference to refer to second to the last node.)
J. O(N)
K. $O(N^3)$
L. O(N)
M. O(NlogN) (base 2 okay, already sorted, but quicksort from lecture picks middle element as pivot, so still O(NlogN))
N. 2 seconds
O. 4 seconds
P. answers between 18 and 22 inclusive accepted
Q. front (if back pop is O(N) as explained in I.)
R. 9 11 7
S. Yes



T.
U. $O(N^2)$
V. {1=2, 3=4, 5=5} (diff in { okay)
W. 3
X. CT!SU
Y. TCU!S

Extra Credit: Chariots of Fire +1

2. Comments

```
  public static TreeMap<String, ArrayList<Integer>>
     getIndex(String[][] book, String[] keyWords) {
     TreeMap<String, ArrayList<Integer>> index;
     index = new TreeMap<>();
     // We know all elements of keyWords are present
     // in book so add them all up front.
     for (String word : keyWords) {
         index.put(word, new ArrayList<>());
     }
     for (int i = 0; i < book.length; i++) {
         int pageNum = i + 1;
         for (String word : book[i]) {
             if (index.containsKey(word)) {
                 ArrayList<Integer> pageNums = index.get(word);
                 // Make sure we have not already
                 // added this page.
                 if (pageNums.size() == 0
                     || pageNums.get(pageNums.size() - 1) != pageNum) {
                     pageNums.add(pageNum);
                  // pageNums is a reference shared by map, so no
                  // need to put back.
                 }
             }
         }
     }
     return index;
}
```

14 points, Criteria:

- Creating resulting map, 1 point
- Add all keywords and empty ArrayLists for each to result at start instead of searching array of keywords for every word. 1 point (Makes checking if a word is index much more efficient)
- loop through all elements of book. Recall, book is an array of array. 2 points
- check if current word is a keyword, any approach okay including searching keyword array and / or contains key method, 1 point (lose if contains() on array)
- change from 0 based indexing of page arrays to 1 based indexing for result, 1 point
- ensure duplicate page number not added for a given word. If word appear on page multiple times the page is only included once in the result. 2 points
- Add new ArrayLists to map (okay if in nested loop for these points), 2 points
- access values already in map with get method correctly, (Not necessary if outer loop is keywords), 2 points (not necessary if outer loop is keywords.)
- Add new pages to ArrayList value for key word (Even if off by one or duplicates), 1 point
- return result, 1 point

Other deductions:

- create ArrayLists unnecessarily -1 add / index all words, -3

3. Comments:

```
public int compareTo(LinkedIntList other) {
    return sum() - other.sum();
}

private int sum() {
    int total = 0;
    IntNode temp = first;
    while (temp != null) {
        total += temp.val;
        temp = temp.next;
    }
    return total;
}
```

10 points, Criteria:

- variable(s) for sum of list(s) initialized to 0, 1 point
- temp node(s) initialized to refer to same node as first(s), 1 point (Lose if Node<E> instead of IntNode)
- correct logic to access all elements in list. -2 if off by one error, 3 points (lose if assume iterator, hasNext, or size variable)
- access and add value from current node to cumulative sum variable, 1 point
- correctly make temp Node variable refer to the next node in the structure, 3 points
- calculate and return result per specification. Okay if only -1, 0, 1 as long as logic correct., 1 point (Lose if assumptions made on empty list. e.g. return +1 if this is empty. What if other list is all negative numbers?)

Others:

- altering either list in any way, -4
- public method for get first, -2
- assuming size method or variables, -3
- iterator or for-each loop, -3

4. Comments:

```
    public boolean removeLast(E tgt) {
        if (first == null) {
            return false;
        }
        // special case when 1 element and it is tgt
        // and have to set first to null.
        if (first.next == null) {
            if (first.data.equals(tgt)) {
                first = null;
                return true;
            } else {
                return false;
            }
        }
        // general case
        Node<E> temp = first;
        // Actually want to stop on the second to the last node.
        while (temp.next.next != null) { // OOF!
            temp = temp.next;
        }
        // temp now referring to the second to the last node
        if (temp.next.data.equals(tgt)) {
            temp.next = null;
            return true;
        } else {
            return false;

        }
    }
```

13 points, Criteria:

- handle empty list case (can be part of general solution), 1 point
- handle case when first must be set to null due to removing only element in list. (This is the only time first will change)., 2 points
- general case, temp node variable made to refer to first node, 1 point
- correctly stop on second to the last node. Can be look ahead, trailer, two pass algorithm or other O(N) approach that works. 3 points (OBOE -2)
- correctly make temp Node variable refer to next node in structure in context of loop, 2 points
- Check equality. equals method, not ==, 1 point
- General case, if last element is equal to target correctly set second to the last node's next reference to null. 2 points
- return boolean result, 1 point

Other:

- > Not O(1) space, -4 Destroys list -4 disallowed methods, -2 to -5

5.
```
public static boolean knightCanReach(int rows,
            Position knight, Position target, int numMoves) {

    if (knight.row == target.row
        && knight.col == target.col) {
            return true; // Got there!!! Pop, pop, pop!!!
    } else if (numMoves == 0) {
        return false; // out of moves
    }
    numMoves--; // local copy, okay to change.
    for (int[] deltas : KNIGHT_DIRECTIONS) {
        int newRow = knight.row + deltas[0];
        int newCol = knight.col + deltas[1];
        if (0 <= newRow && newRow < rows
                && 0 <= newCol && newCol < rows) {
            // New position inbounds, okay to have as base case at
            // start of method. Maybe that's better?
            Position newPos = new Position(newRow, newCol);
            if (knightCanReach(rows, newPos, target, numMoves)) {
                return true; // Pop, pop, pop!!!
            }
         }
         // nothing to undo as we made a new Position
    }
    return false; // never found a solution.
```

13 points, Criteria:

- failure base case if moves left < 0, 2 points
- success base case, positions the same and return true, 2 points
- recursive case, loop through move choices, 1 point
- Create new position or update current knight's position from current move and determine new position is in bounds. Check on in bounds failure base case at top of method. Lose if don't check new position also >= 0 for row and column. Also must undo if alter Knight's position instead of undoing. 2 points
- correct recursive call, 2 points
- return true IFF recursive call is successful, 3 points
- return false after loop if no success, 1 point

Other:

- early return, -5
- don't use return value (Very common. Made recursive call but didn't check to see if result was true.) -5
- equals method assumed for Position (not method indicated), -1
- added helper methods, -3 (disallowed by instructions)