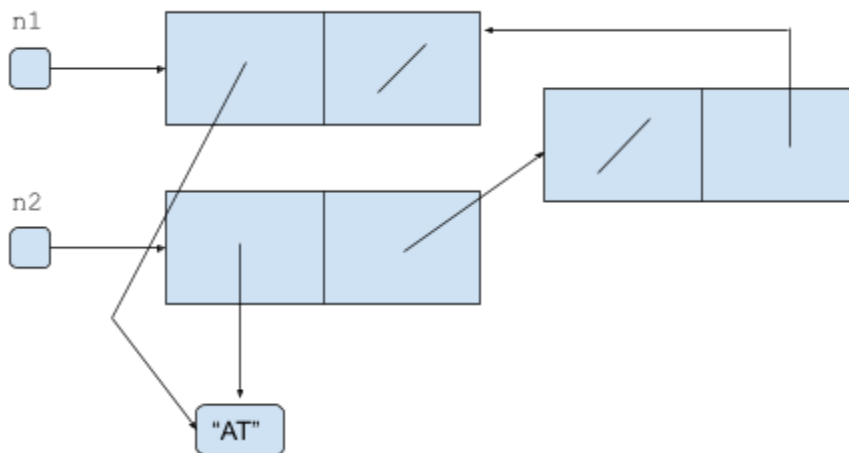
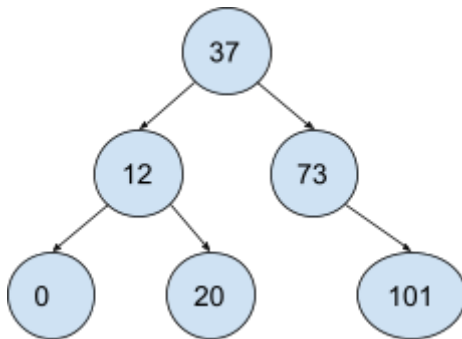


Skyler's Mega Awesome (and ugly ... and probably incorrect) Psuedo-Key. Email me at skylervestal@utexas.edu if anything is incorrect.

- A. 13
- B. -5
- C. $L20 \times 3G1$
- D. 58
- E. Either is fine.
- F. 9 seconds
- G. Selection, insertion, and quick sort.
- H. .20 seconds



- I.
- J. 80 seconds
- K. Move `t1.get(i)` above the for loop with `j` (`t1` is a `LinkedList`)



- L.
- M. 0 20 12 101 73 37
- N. 7 (Only the left node has 2 children -- repeat to height 4)
- O. 20 seconds ($N \log(N)$)

2.

```
public boolean rangeEqualsTarget(int tgt, int start, int stop) {
    int sum = 0;
    int index = 0;
    IntNode temp = first;
    while (temp != null && index < start) {
        temp = temp.next;
        index += 1;
    }
    while (temp != null && index < stop) {
        sum += temp.data;
        temp = temp.next;
        index += 1;
    }
    return sum == tgt;
}
```

Thought Process:

- The conditions Mike gives (if stop is out of bounds go until the end of the list and consider that sum) makes the iteration clear -- get to the first node at index start, and then sum each node's data until you either hit the node at index stop or the end
- Only thing to consider is if we're off by one getting to the start node (maybe it should be index <= start?). Let's try it out:
 - If start == 0, then the first while loop will never iterate. We'll start adding at 0
 - If start == 1, then the first while loop will iterate a single time (0 < 1) and then stop. We'll start adding at 1
 - This is CS 314 and not CS 331. Good enough for me!

Remarks:

- If you get to index start you could just iterate while temp != null and stop - start times. Around the same amount of code and the one I wrote above

3.

```
public LinkedList combineIgnoreValue(LinkedList other, int tgt) {
    LinkedList res = new LinkedList();
    // res.first right now is null (ugh) -- important to work around this
    IntNode tempR = null;

    IntNode tempT = first;
    IntNode tempO = other.first;
    // Hit the end of the shorter
    while (tempT != null && tempO != null) {
        // Pretty straight forward, hard part is the helper method
        // tempR = ... is kinda weird -- look at the comments below
        if (tempT.val != tgt && tempO.val != tgt) {
            tempR = addNode(res, tempR, tempT.val + tempO.val);
        }
        tempT = tempT.next;
        tempO = tempO.next;
    }
    // One list may be longer -- add the data in from the longer
    tempT = tempT != null ? tempT : tempO;
    while (tempT != null) {
        if (tempT.val != tgt) {
            tempR = addNode(res, tempR, tempT.val);
        }
    }
    return res;
}
```

```

// The idea is to add in the first node if needed, and otherwise add a new node
to the end
// In both cases we return the last node in the list so we can add to this node
in the future
private IntNode addNode(LinkedIntList list, IntNode curr, int val) {
    IntNode next;
    if (list.first == null) {
        list.first = new IntNode();
        next = list.first;
    } else {
        curr.next = new IntNode();
        next = curr.next;
    }
    next.val = val;
    return next;
}

```

Thought Process:

- This one is very tedious. Note if I was a student on the exam I wouldn't make the helper method. I'd just repeat the same code twice and let my TA be sad about my choices
- The idea of the question is pretty simple, but dealing with if first is null or not is very tedious to work around. As a result, I think it's easiest to make a helper method that will take care of the complicated logic for me, and it'll return the last node in the list to make adding to the end easy in the future.
- I would not be ashamed if I lost a few points on this question for the sake of covering the rest of the exam with enough time (5 coding questions?!?). Lots of details that are diminishing gains vs making sure the rest of your exam is strong. If you have the time though a solution like this is very satisfying. Really good question for distinguishing efficiency and details.

4.

```
public static <E> boolean sameBottomN(Stack<E> s1, Stack<E> s2, int n) {
    if (s1.size() < n || s2.size() < n) {
        return false;
    }
    Stack<E> t1 = new Stack<>();
    Stack<E> t2 = new Stack<>();
    // We'll have to remove all the elements to get to the bottom anyway, so
    // let's make it easier on ourselves and start from the bottom element
    while (!s1.isEmpty()) {
        t1.push(s1.pop());
    }
    while (!s2.isEmpty()) {
        t2.push(s2.pop());
    }
    // Compare the first n -- stop if there's a different
    boolean same = true;
    int count = 0;
    while (count < n && same) {
        E e1 = t1.pop();
        E e2 = t2.pop();
        same = e1.equals(e2);
        s1.push(e1);
        s2.push(e2);
    }
    // Add everything back
    while (!t1.isEmpty()) {
        s1.push(t1.pop());
    }
    while (!t2.isEmpty()) {
        s2.push(t2.pop());
    }
    return same;
}
```

Thought Process:

- First handle any weird cases with out of bounds w/ n. Mike makes this easy.
- Next we need to get to the bottom of the stack. Pop everything off, compare the first n (stop at the first mismatch), then add everything back
- Double and triple check the 4 while loops are all correct. Easy for your brain to make a silly mistake here even though we all know what you mean

5.

```
private Node helper(BNode curr) {
    if (curr == null) {
        return null;
    }
    // At times like these I wish java would let me use ^ for boolean vals
    // "Umm ackshutally you could do that with two ternaries 🤔"
    // yeah yeah whatever I would if I wasn't making an answer key
    boolean leftEmpty = curr.left == null;
    boolean rightEmpty = curr.right == null;
    boolean needsFix = (leftEmpty && !rightEmpty) || (!leftEmpty &&
rightEmpty);
    if (needsFix) {
        if (leftEmpty) {
            curr.left = new BNode(curr.data);
        } else {
            curr.right = new BNode(curr.data);
        }
    }
    // Run recursion to fix any subtrees -- base case handles newly created
fixes easily
    curr.left = helper(curr.left);
    curr.right = helper(curr.right);
    return curr;
}
```

Thought Process:

- Not much to say here. Classic recursion.
- This is why you shouldn't spend too much time on question 3. Imagine if you knew how to do this one and missed 10 points here being incomplete because you spent so much time getting an extra +2 for details before 🤖

6.

```
private int help(int start, int dest, TNode n, int distanceFromStart) {
    // REMEMBER -- NO DUPLICATES! Make this much easier for the base cases:
    // If the tree is empty OR if we reach the end (if we find end before start
    it works out!)
    if (n == null || n.data == dest) {
        return distanceFromStart;
    }
    // We can finally start counting
    if (n.data == start) {
        distanceFromStart = 0;
    }
    // Only add an edge if we've reached start to count
    int addEdge = distanceFromStart >= 0 ? 1 : 0;
    for (TNode c : children) {
        int res = help(start, dest, c, distanceFromStart + addEdge);
        if (res > 0) {
            return res;
        }
    }
    return -1;
}
```

Thought Process:

- Last question and is not easy to handle. I'd be running out of time at this point so I'd write the first thing that seems right and only check if I have time to.
- Base case:
 - If our tree is empty no point to going on. Otherwise since each node's data is unique if we find dest then we know we're done!
 - Does this work if we found end before start? Well if we do then distanceFromStart is -1 (case below). Yup!
- Lastly -- How do we handle recursion?
 - Mark distanceFromStart to be 0 if we actually find start -- we only want to add to our current distance if we've already found start (which we can know by checking if distanceFromStart is positive)
 - Tricky part: There is only one path, so if we never find it we should return -1. This means if we ever get a non-negative value we found the path. Return it!
 - Sanity Check:
 - Can we ever return a positive value without finding the path? We only return not -1 if we find the end after the start. (Base case second condition). Works for me!