```
      isPalindrome: list = 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 3
        isPalindrome: list = 4 ) 3 ) 2 ) 1 ) 0. len = 1
        returns node 3b, true
      returns node 2b, true
    returns node 1b, true
  returns node 0b, true
returns null, true
```

Implementing this code is now just a matter of filling in the details.

```java
boolean isPalindrome(LinkedListNode head) {
  int length = lengthOfList(head);
  Result p = isPalindromeRecurse(head, length);
  return p.result;
}

Result isPalindromeRecurse(LinkedListNode head, int length) {
  if (head == null || length <= 0) { // Even number of nodes
    return new Result(head, true);
  } else if (length == 1) { // Odd number of nodes
    return new Result(head.next, true);
  }

  /* Recurse on sublist. */
  Result res = isPalindromeRecurse(head.next, length - 2);

  /* If child calls are not a palindrome, pass back up
   * a failure. */
  if (!res.result || res.node == null) {
    return res;
  }

  /* Check if matches corresponding node on other side. */
  res.result = (head.data == res.node.data);

  /* Return corresponding node. */
  res.node = res.node.next;

  return res;
}

int lengthOfList(LinkedListNode n) {
  int size = 0;
  while (n != null) {
    size++;
    n = n.next;
  }
  return size;
}
```

Some of you might be wondering why we went through all this effort to create a special `Result` class. Isn't there a better way? Not really—at least not in Java.

However, if we were implementing this in C or C++, we could have passed in a double pointer.

```cpp
bool isPalindromeRecurse(Node head, int length, Node** next) {
  ...
}
```

It's ugly, but it works.

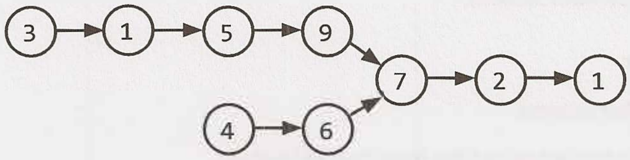
2.7 **Intersection**: Given two (singly) linked lists, determine if the two lists intersect. Return the intersecting node. Note that the intersection is defined based on reference, not value. That is, if the kth node of the first linked list is the exact same node (by reference) as the `j`th node of the second linked list, then they are intersecting.

pg 95

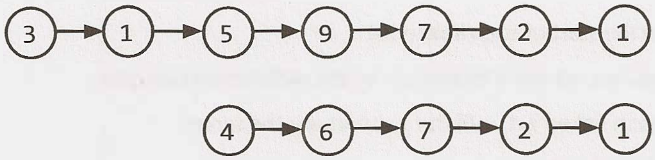
SOLUTION

Let's draw a picture of intersecting linked lists to get a better feel for what is going on.

Here is a picture of intersecting linked lists:



And here is a picture of non-intersecting linked lists:



We should be careful here to not inadvertently draw a special case by making the linked lists the same length.

Let's first ask how we would determine if two linked lists intersect.

Determining if there's an intersection.

How would we detect if two linked lists intersect? One approach would be to use a hash table and just throw all the linked lists nodes into there. We would need to be careful to reference the linked lists by their memory location, not by their value.

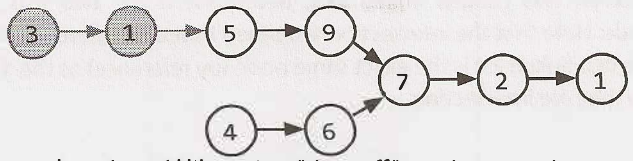
There's an easier way though. Observe that two intersecting linked lists will always have the same last node. Therefore, we can just traverse to the end of each linked list and compare the last nodes.

How do we find where the intersection is, though?

Finding the intersecting node.

One thought is that we could traverse backwards through each linked list. When the linked lists "split", that's the intersection. Of course, you can't really traverse backwards through a singly linked list.

If the linked lists were the same length, you could just traverse through them at the same time. When they collide, that's your intersection.



When they're not the same length, we'd like to just "chop off"—or ignore—those excess (gray) nodes.

How can we do this? Well, if we know the lengths of the two linked lists, then the difference between those two linked lists will tell us how much to chop off.

We can get the lengths at the same time as we get the tails of the linked lists (which we used in the first step to determine if there's an intersection).

Putting it all together.

We now have a multistep process.

1. Run through each linked list to get the lengths and the tails.
2. Compare the tails. If they are different (by reference, not by value), return immediately. There is no intersection.
3. Set two pointers to the start of each linked list.
4. On the longer linked list, advance its pointer by the difference in lengths.
5. Now, traverse on each linked list until the pointers are the same.

The implementation for this is below.

```
LinkedListNode findIntersection(LinkedListNode list1, LinkedListNode list2) {
  if (list1 == null || list2 == null) return null;

  /* Get tail and sizes. */
  Result result1 = getTailAndSize(list1);
  Result result2 = getTailAndSize(list2);

  /* If different tail nodes, then there's no intersection. */
  if (result1.tail != result2.tail) {
    return null;
  }

  /* Set pointers to the start of each linked list. */
  LinkedListNode shorter = result1.size < result2.size ? list1 : list2;
  LinkedListNode longer = result1.size < result2.size ? list2 : list1;

  /* Advance the pointer for the longer linked list by difference in lengths. */
  longer = getKthNode(longer, Math.abs(result1.size - result2.size));

  /* Move both pointers until you have a collision. */
  while (shorter != longer) {
    shorter = shorter.next;
    longer = longer.next;
  }

  /* Return either one. */
  return longer;
}

```

```
class Result {
  public LinkedListNode tail;
  public int size;
  public Result(LinkedListNode tail, int size) {
    this.tail = tail;
    this.size = size;
  }
}

Result getTailAndSize(LinkedListNode list) {
  if (list == null) return null;

  int size = 1;
  LinkedListNode current = list;
  while (current.next != null) {
    size++;
    current = current.next;
  }
  return new Result(current, size);
}

LinkedListNode getKthNode(LinkedListNode head, int k) {
  LinkedListNode current = head;
  while (k > 0 && current != null) {
    current = current.next;
    k--;
  }
  return current;
}
```

This algorithm takes `O(A + B)` time, where `A` and `B` are the lengths of the two linked lists. It takes `O(1)` additional space.

2.8 **Loop Detection:** Given a circular linked list, implement an algorithm that returns the node at the beginning of the loop.

DEFINITION

Circular linked list: A (corrupt) linked list in which a node's next pointer points to an earlier node, so as to make a loop in the linked list.

EXAMPLE

Input: `A -> B -> C -> D -> E -> C` [the same C as earlier]

Output: `C`

pg 95

SOLUTION

This is a modification of a classic interview problem: detect if a linked list has a loop. Let's apply the Pattern Matching approach.

Part 1: Detect If Linked List Has A Loop

An easy way to detect if a linked list has a loop is through the `FastRunner` / `SlowRunner` approach. `FastRunner` moves two steps at a time, while `SlowRunner` moves one step. Much like two cars racing around a track at different steps, they must eventually meet.

An astute reader may wonder if `FastRunner` might "hop over" `SlowRunner` completely, without ever colliding. That's not possible. Suppose that `FastRunner` *did* hop over `SlowRunner`, such that `SlowRunner` is at spot `i` and `FastRunner` is at spot `i + 1`. In the previous step, `SlowRunner` would be at spot `i - 1` and `FastRunner` would at spot `((i + 1) - 2)`, or spot `i - 1`. That is, they would have collided.

Part 2: When Do They Collide?

Let's assume that the linked list has a "non-looped" part of size k.

If we apply our algorithm from part 1, when will `FastRunner` and `SlowRunner` collide?

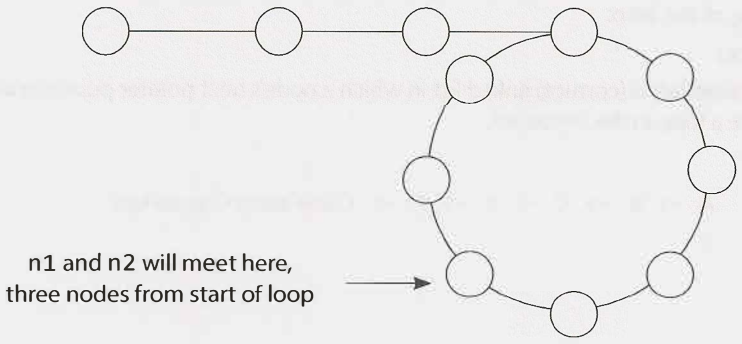
We know that for every p steps that `SlowRunner` takes, `FastRunner` has taken 2p steps. Therefore, when `SlowRunner` enters the looped portion after k steps, `FastRunner` has taken 2k steps total and must be `2k - k` steps, or k steps, into the looped portion. Since k might be much larger than the loop length, we should actually write this as `mod(k, LOOP_SIZE)` steps, which we will denote as K.

At each subsequent step, `FastRunner` and `SlowRunner` get either one step farther away or one step closer, depending on your perspective. That is, because we are in a circle, when A moves q steps away from B, it is also moving q steps closer to B.

So now we know the following facts:

1. `SlowRunner` is 0 steps into the loop.
2. `FastRunner` is K steps into the loop.
3. `SlowRunner` is K steps behind `FastRunner`.
4. `FastRunner` is `LOOP_SIZE - K` steps behind `SlowRunner`.
5. `FastRunner` catches up to `SlowRunner` at a rate of 1 step per unit of time.

So, when do they meet? Well, if `FastRunner` is `LOOP_SIZE - K` steps behind `SlowRunner`, and `FastRunner` catches up at a rate of 1 step per unit of time, then they meet after `LOOP_SIZE - K` steps. At this point, they will be K steps before the head of the loop. Let's call this point `CollisionSpot`.



Part 3: How Do You Find The Start of the Loop?

We now know that `CollisionSpot` is K nodes before the start of the loop. Because `K = mod(k, LOOP_SIZE)` (or, in other words, `k = K + M * LOOP_SIZE`, for any integer M), it is also correct to say that it is k nodes from the loop start. For example, if node N is 2 nodes into a 5 node loop, it is also correct to say that it is 7, 12, or even 397 nodes into the loop.

Therefore, both `CollisionSpot` and `LinkedListHead` are k nodes from the start of the loop.

Now, if we keep one pointer at CollisionSpot and move the other one to LinkedListHead, they will each be k nodes from LoopStart. Moving the two pointers at the same speed will cause them to collide again—this time after k steps, at which point they will both be at LoopStart. All we have to do is return this node.

Part 4: Putting It All Together

To summarize, we move FastPointer twice as fast as SlowPointer. When SlowPointer enters the loop, after k nodes, FastPointer is k nodes into the loop. This means that FastPointer and SlowPointer are LOOP_SIZE - k nodes away from each other.

Next, if FastPointer moves two nodes for each node that SlowPointer moves, they move one node closer to each other on each turn. Therefore, they will meet after LOOP_SIZE - k turns. Both will be k nodes from the front of the loop.

The head of the linked list is also k nodes from the front of the loop. So, if we keep one pointer where it is, and move the other pointer to the head of the linked list, then they will meet at the front of the loop.

Our algorithm is derived directly from parts 1, 2 and 3.

1. Create two pointers, FastPointer and SlowPointer.
2. Move FastPointer at a rate of 2 steps and SlowPointer at a rate of 1 step.
3. When they collide, move SlowPointer to LinkedListHead. Keep FastPointer where it is.
4. Move SlowPointer and FastPointer at a rate of one step. Return the new collision point.

The code below implements this algorithm.

```
LinkedListNode FindBeginning(LinkedListNode head) {
  LinkedListNode slow = head;
  LinkedListNode fast = head;

  /* Find meeting point. This will be LOOP_SIZE - k steps into the linked list. */
  while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next.next;
    if (slow == fast) { // Collision
      break;
    }
  }

  /* Error check - no meeting point, and therefore no loop */
  if (fast == null || fast.next == null) {
    return null;
  }

  /* Move slow to Head. Keep fast at Meeting Point. Each are k steps from the
   * Loop Start. If they move at the same pace, they must meet at Loop Start. */
  slow = head;
  while (slow != fast) {
    slow = slow.next;
    fast = fast.next;
  }

  /* Both now point to the start of the loop. */
  return fast;
}
```

3

Solutions to Stacks and Queues

3.1 Three in One: Describe how you could use a single array to implement three stacks.

pg 98

SOLUTION

Like many problems, this one somewhat depends on how well we'd like to support these stacks. If we're okay with simply allocating a fixed amount of space for each stack, we can do that. This may mean though that one stack runs out of space, while the others are nearly empty.

Alternatively, we can be flexible in our space allocation, but this significantly increases the complexity of the problem.

Approach 1: Fixed Division

We can divide the array in three equal parts and allow the individual stack to grow in that limited space. Note: We will use the notation "[" to mean inclusive of an end point and "(" to mean exclusive of an end point.

- For stack 1, we will use `[0,` $^{n}/_{3}$ `)`.
- For stack 2, we will use `[` $^{n}/_{3}$ `,` $^{2n}/_{3}$ `)`.
- For stack 3, we will use `[` $^{2n}/_{3}$ `, n)`.

The code for this solution is below.

```
class FixedMultiStack {
  private int numberOfStacks = 3;
  private int stackCapacity;
  private int[] values;
  private int[] sizes;

  public FixedMultiStack(int stackSize) {
    stackCapacity = stackSize;
    values = new int[stackSize * numberOfStacks];
    sizes = new int[numberOfStacks];
  }

  /* Push value onto stack. */
  public void push(int stackNum, int value) throws FullStackException {
    /* Check that we have space for the next element */
    if (isFull(stackNum)) {
      throw new FullStackException();
    }

    /* Increment stack pointer and then update top value. */
    sizes[stackNum]++;
    values[indexOfTop(stackNum)] = value;
  }

  /* Pop item from top stack. */
  public int pop(int stackNum) {
    if (isEmpty(stackNum)) {
      throw new EmptyStackException();
    }

    int topIndex = indexOfTop(stackNum);
    int value = values[topIndex]; // Get top
    values[topIndex] = 0; // Clear
    sizes[stackNum]--; // Shrink
    return value;
  }

  /* Return top element. */
  public int peek(int stackNum) {
    if (isEmpty(stackNum)) {
      throw new EmptyStackException();
    }
    return values[indexOfTop(stackNum)];
  }

  /* Return if stack is empty. */
  public boolean isEmpty(int stackNum) {
    return sizes[stackNum] == 0;
  }

  /* Return if stack is full. */
  public boolean isFull(int stackNum) {
    return sizes[stackNum] == stackCapacity;
  }

  /* Returns index of the top of the stack. */
  private int indexOfTop(int stackNum) {
    int offset = stackNum * stackCapacity;
    int size = sizes[stackNum];
    return offset + size - 1;
  }
}
```

If we had additional information about the expected usages of the stacks, then we could modify this algorithm accordingly. For example, if we expected Stack 1 to have many more elements than Stack 2, we could allocate more space to Stack 1 and less space to Stack 2.

Approach 2: Flexible Divisions

A second approach is to allow the stack blocks to be flexible in size. When one stack exceeds its initial capacity, we grow the allowable capacity and shift elements as necessary.

We will also design our array to be circular, such that the final stack may start at the end of the array and wrap around to the beginning.

Please note that the code for this solution is far more complex than would be appropriate for an interview. You could be responsible for pseudocode, or perhaps the code of individual components, but the entire implementation would be far too much work.

```
public class MultiStack {
   /* StackInfo is a simple class that holds a set of data about each stack. It
    * does not hold the actual items in the stack. We could have done this with
    * just a bunch of individual variables, but that's messy and doesn't gain us
    * much. */
   private class StackInfo {
      public int start, size, capacity;
      public StackInfo(int start, int capacity) {
         this.start = start;
         this.capacity = capacity;
      }

      /* Check if an index on the full array is within the stack boundaries. The
       * stack can wrap around to the start of the array. */
      public boolean isWithinStackCapacity(int index) {
         /* If outside of bounds of array, return false. */
         if (index < 0 || index >= values.length) {
            return false;
         }

         /* If index wraps around, adjust it. */
         int contiguousIndex = index < start ? index + values.length : index;
         int end = start + capacity;
         return start <= contiguousIndex && contiguousIndex < end;
      }

      public int lastCapacityIndex() {
         return adjustIndex(start + capacity - 1);
      }

      public int lastElementIndex() {
         return adjustIndex(start + size - 1);
      }

      public boolean isFull() { return size == capacity; }
      public boolean isEmpty() { return size == 0; }
   }

   private StackInfo[] info;
   private int[] values;

   public MultiStack(int numberOfStacks, int defaultSize) {
      /* Create metadata for all the stacks. */
      info = new StackInfo[numberOfStacks];
      for (int i = 0; i < numberOfStacks; i++) {
         info[i] = new StackInfo(defaultSize * i, defaultSize);
      }
      values = new int[numberOfStacks * defaultSize];
   }

   /* Push value onto stack num, shifting/expanding stacks as necessary. Throws
    * exception if all stacks are full. */
   public void push(int stackNum, int value) throws FullStackException {
```