Question 3: Waiting List

Part A: `getKthNode` **3 points**

+1/2 create copy reference (used in loop) *if recursive solution:* returns `front` if $k == 0$

+1 1/2 iterate through list
- **+1/2** loop (or recursion) referring to the list
- **+1** update list reference *in context of loop/recursion*
  - **+ 1/2** attempt
  - **+ 1/2** correct

+1 return kth node

Part B: `transferNodesFromEnd` **6 points**

+1 1/2 get sublist from `other`
- **+1/2** attempt to access sublist (manual traversal OK)
- **+1** correctly access sublist w/ `getKthNode` (or previous node if later use `getNext`)

+2 add sublist to end of `this`
- **+1** get reference to last node (manual traversal OK)
  - **+ 1/2** attempt (must include valid reference to `null` or `size` or `numNodes`)
  - **+ 1/2** correct
- **+1** add nodes from `other` to end (w/o creating any `ListNodes`)
  - **+1/2** attempt
  - **+1/2** correct

+1 1/2 remove nodes from end of `other`
- **+ 1** remove nodes
  - **+1/2** attempt (manual traversal OK)
  - **+1/2** correct when `other.size()!= num` (must call `getKthNode`)
- **+1/2** set `other.front` to `null` if all moved (CANNOT define `setFront` public mutator)

+1 update counts
- **+1/2** add `num` to `numNodes`
- **+1/2** subtract `num` from `other.numNodes` (CANNOT define public `setNumNodes` mutator)

Common Usage: Moving `front` is **-1** for destruction of data structure.

AP® COMPUTER SCIENCE A/AB
2006 GENERAL USAGE

Most common usage errors are addressed specifically in rubrics with points deducted in a manner other than indicated on this sheet. The rubric takes precedence.

Usage points can only be deducted if the part where it occurs has earned credit.

A usage error that occurs once when the same usage is correct two or more times can be regarded as an oversight and not penalized. If the usage error is the only instance, one of two, or occurs two or more times, then it should be penalized.

A particular usage error should be penalized only once in a problem, even if it occurs on different parts of a problem.

Nonpenalized Errors

spelling/case discrepancies*

local variable not declared when any other variables are declared in some part

default constructor called without parens; for example, `new Fish;`

use keyword as identifier

`[r,c]`, `(r)(c)` or `(r,c)` instead of `[r][c]`

= instead of == (and vice versa)

`length`/`size` confusion for array, `String`, and `ArrayList`, with or without `()`

`private` qualifier on local variable

extraneous code with no side-effect, for example a check for precondition

common mathematical symbols for operators (x • ÷ ≤ ≥ < > ≠)

missing { } where indentation clearly conveys intent

missing `( )` on method call or around `if/while` conditions

missing `;`s

missing "new" for constructor call once, when others are present in some part

missing downcast from collection

missing `int` cast when needed

missing `public` on class or constructor header

Minor Errors (1/2 point)

confused identifier (e.g., `len` for `length` or `left()` for `getLeft()`)

no local variables declared

`new` never used for constructor calls

`void` method or constructor returns a value

modifying a constant (`final`)

use `equals` or `compareTo` method on primitives, for example
`int x; …x.equals(val)`

`[]` – `get` confusion if access not tested in rubric

assignment dyslexia, for example, `x + 3 = y;` for `y = x + 3;`

`super(method())` instead of `super.method()`

formal parameter syntax (with type) in method call, e.g., `a = method(int x)`

missing `public` from method header when required

"false"/"true" or 0/1 for boolean values

"null" for `null`

Major Errors (1 point)

extraneous code which causes side-effect, for example, information written to output

use interface or class name instead of variable identifier, for example `Simulation.step()` instead of `sim.step()`

`aMethod(obj)` instead of `obj.aMethod()`

use of object reference that is incorrect, for example, use of `f.move()` inside method of `Fish` class

use private data or method when not accessible

destruction of data structure (e.g., by using root reference to a `TreeNode` for traversal of the tree)

use class name in place of `super` either in constructor or in method call

**Note: Spelling and case discrepancies for identifiers fall under the "nonpenalized" category as long as the correction can be unambiguously inferred from context. For example, "Queu" instead of "Queue". Likewise, if a student declares "Fish fish;", then uses Fish.move() instead of fish.move(), the context allows for the reader to assume the object instead of the class.*

AP® COMPUTER SCIENCE AB
2006 CANONICAL SOLUTIONS

Question 3: Waiting List

PART A:

```
public ListNode getKthNode(int k)
{
  ListNode step = front;
  for (int i = 0; i < k; i++)
  {
    step = step.getNext();
  }
  return step;
}
```

ALTERNATE SOLUTION

```
public ListNode getKthNode(int k)
{
  if (k == 0)
  {
    return front;
  }
  return getKthNode(k-1).getNext();
}
```

PART B:

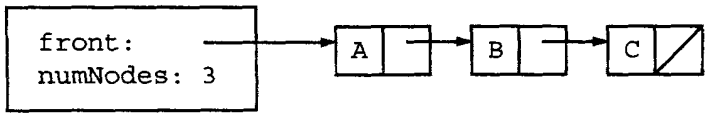
```
public void transferNodesFromEnd(WaitingList other, int num)
{
  ListNode lastNode = getKthNode(size()-1);
  lastNode.setNext(other.getKthNode(other.size()-num));

  if (other.numNodes == num)
  {
    other.front = null;
  }
  else
  {
    other.getKthNode(other.size()-num -1).setNext(null);
  }

  numNodes += num;
  other.numNodes -= num;
}
```

AB3 A1

(a) Write the `WaitingList` method `getKthNode`. This method should return a reference to the node at index `k`. Nodes in a `WaitingList` are indexed consecutively from the front, starting at 0.

In the diagram below, `list1.getKthNode(0)` returns a reference to the node containing `A`, `list1.getKthNode(1)` returns a reference to the node containing `B`, and `list1.getKthNode(2)` returns a reference to the node containing `C`.

`WaitingList list1`



Complete method `getKthNode` below.

```
// returns a reference to the node at index k,
// where the indexes are numbered 0 through size()-1
// precondition: 0 <= k < size()
private ListNode getKthNode(int k)
{
    ListNode curr = front;
    int count = 0;
    while (count < k)
    {
        curr = curr.getNext();
        count++;
    }
    return curr;
}
```

Part (b) begins on page 16.

AB3 A2

In writing `transferNodesFromEnd`, you may assume that `getKthNode` works as specified regardless of what you wrote in part (a).

Complete method `transferNodesFromEnd` below.

```
// removes the last num nodes from other and attaches them
// in the same order to the end of this WaitingList;
// updates the number of nodes in each list to reflect the move
// precondition:  size() > 0;
//                0 < num <= other.size()
public void transferNodesFromEnd(WaitingList other, int num)
{
    int sizeThis = size();
    int sizeOther = other.size();
    getKthNode(sizeThis - 1).setNext(other.getKthNode(sizeOther - num));
    if(sizeOther == num)
        other.front = null;
    else
        other.getKthNode(sizeOther - num - 1).setNext(null);
    numNodes += num;
    other.numNodes -= num;
}
```

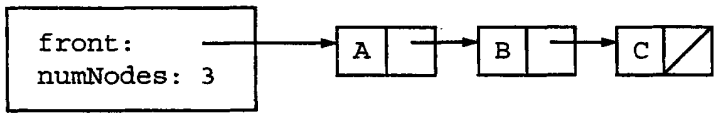
GO ON TO THE NEXT PAGE.

AB3B,

(a) Write the `WaitingList` method `getKthNode`. This method should return a reference to the node at index `k`. Nodes in a `WaitingList` are indexed consecutively from the front, starting at 0.

In the diagram below, `list1.getKthNode(0)` returns a reference to the node containing `A`, `list1.getKthNode(1)` returns a reference to the node containing `B`, and `list1.getKthNode(2)` returns a reference to the node containing `C`.

`WaitingList list1`



Complete method `getKthNode` below.

```
// returns a reference to the node at index k,
// where the indexes are numbered 0 through size()-1
// precondition: 0 <= k < size()
private ListNode getKthNode(int k)
{
   List Node  curr Node = front;
   for (int i=0; i < size(); i++)
   {
      if( i == k)
         return  curr Node;
      else
         curr Node = curr Node . get Next();
   }
   return null;
}
```

Part (b) begins on page 16.

In writing `transferNodesFromEnd`, you may assume that `getKthNode` works as specified regardless of what you wrote in part (a).

Complete method `transferNodesFromEnd` below.

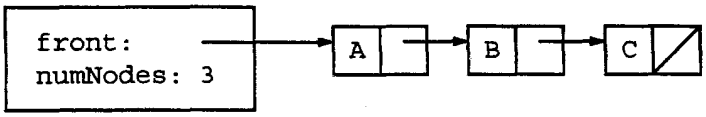
```
// removes the last num nodes from other and attaches them
// in the same order to the end of this WaitingList;
// updates the number of nodes in each list to reflect the move
// precondition:  size() > 0;
//                0 < num <= other.size()
public void transferNodesFromEnd(WaitingList other, int num)
{
   ListNode currLast = getKthNode(size()-1)
   if(other.size() > num)
   { ListNode otherLast, transferred;
     currNode = other.getFront();
     for(int i=0; i< other.size(); i++)
     {
        if (i == other.size - num -1)
           otherLast = currNode;
        if(i == other.size - num)
           transferred = currNode;
        currNode = currNode.getNext();
     }
     currLast.setNext(transferred);
   }
   else
   {
     currLast.setNext(other.getFront());
     other.setFront(null);
   }
}
```

AB3C,

(a) Write the `WaitingList` method `getKthNode`. This method should return a reference to the node at index `k`. Nodes in a `WaitingList` are indexed consecutively from the front, starting at 0.

In the diagram below, `list1.getKthNode(0)` returns a reference to the node containing `A`, `list1.getKthNode(1)` returns a reference to the node containing `B`, and `list1.getKthNode(2)` returns a reference to the node containing `C`.

`WaitingList list1`



Complete method `getKthNode` below.

```
// returns a reference to the node at index k,
// where the indexes are numbered 0 through size()-1
// precondition: 0 <= k < size()
private ListNode getKthNode(int k)
{
    private Object reference;

    reference = waitingList.get(k);

    return reference;

}
```

Part (b) begins on page 16.

AB3C2

In writing `transferNodesFromEnd`, you may assume that `getKthNode` works as specified regardless of what you wrote in part (a).

Complete method `transferNodesFromEnd` below.

```
// removes the last num nodes from other and attaches them
// in the same order to the end of this WaitingList;
// updates the number of nodes in each list to reflect the move
// precondition:  size() > 0;
//                0 < num <= other.size()
public void transferNodesFromEnd(WaitingList other, int num)
{
    private ListNode temp;

    temp = other.getKthNode(other.size() - num);

    WaitingList.addLast(temp);
    other.numNodes = other.numNodes - num;
    WaitingList.numNodes = WaitingList.numNodes + num;

}
```

GO ON TO THE NEXT PAGE.

AP® COMPUTER SCIENCE AB
2006 SCORING COMMENTARY

Question 3

Overview

This question focused on linked-list manipulation and abstraction. A `WaitingList` class that had a linked-list of `ListNodes` and a node count as private fields was provided. In part (a) students were required to complete the `private findKth` method, which returned the `ListNode` at a specified index of the list. This involved traversing the links in the list, keeping track of a counter, and returning the desired `ListNode` when reached. In part (b) students were required to complete the `transferNodesFromEnd` method, which transferred a specified number of `ListNodes` from the end of a different list onto the current one. This involved locating the desired `ListNode` in the other list (using the `findKth` method from part (a)), linking that node to the end of the current list, resetting the end of the other list, and updating the node counts for both lists. While the amount of code required to complete these tasks was modest, doing so required a solid understanding of linked structures and their manipulation.

Sample: AB3A
Score: 9

Part (a) demonstrates the canonical solution. A reference to a `ListNode` is created and this reference points at `front`. A loop is used to update the reference and return the correct node.

In part (b) the student's solution is also the same as the canonical solution. The current sizes of each list are saved, allowing the student to update the number of nodes at any point in this code. The third line of the code does three major tasks: accesses the sublist, references the end of this list, and adds the nodes. The student handles the special case of removing all nodes correctly and then updates the number of nodes in each list.

Sample: AB3B
Score: 5

Part (a) earned all 3 points, declaring a reference to the front of the list and creating a loop that correctly identifies the kth node and then returning it.

Part (b) begins by correctly accessing the last node of the list. No points were earned for accessing the sublist because there is no `getFront()` method available and the student does not use the `getKthNode` method written in part (a). The nodes are added correctly, but there is no attempt to remove the nodes from the other list. In the case of removing all of the nodes, the student does not correctly set the `front` reference in `other` to `null`. Additionally, there is no attempt to update the number of nodes in either list.

Sample: AB3C
Score: 2

Part (a) earned no points.

Part (b) correctly accesses the sublist in `other`, earning 1½ points, and updates the number of nodes in `other`, earning a ½ point. There is no `addLast` method and no `WaitingList` object (`WaitingList` is a class or object type), so no other points were awarded.