AP® COMPUTER SCIENCE AB
2008 SCORING GUIDELINES

Question 2: Cache List

Part A: get **5 points**

+2 determine start location
- **+1** correct in first call to `get (front)`
- **+1** correct in subsequent calls to `get (front` or `remNode)`

+2 traverse to desired node (in context of loop)
- **+1/2** call to `getNext()`
- **+1/2** accesses more than one successive node (if needed)
- **+1** identifies desired node (may assume that `remNode` is not null)

+1/2 update `remNode` and `remIndex`

+1/2 return value at identified node

Part B: addFirst **2 1/2 points**

+1 add object
- **+1/2** create `ListNode` object containing `obj` and `front`
- **+1/2** update `front` to refer to new node

+1 1/2 update state
- **+1/2** increment `listSize`
- **+1** increment `remIndex` (if not previously -1)

Part C: Big-Oh **1 1/2 points**

+1/2 $O(n^2)$ for `LinkedList printForward` and `printReverse`
+1/2 O(n) for `APList printForward`
+1/2 $O(n^2)$ for `APList printReverse`

AP® COMPUTER SCIENCE AB
2008 CANONICAL SOLUTIONS

Question 2: Cache List

PART A:

```
public Object get(int n)
{
  if (remIndex == -1 || n < remIndex)
  {
    remNode = front;
    remIndex = 0;
  }

  for (int i = remIndex; i < n; i++)
  {
    remNode = remNode.getNext();
  }
  remIndex = n;

  return remNode.getValue();
}
```

PART B:

```
public void addFirst(Object obj)
{
  front = new ListNode(obj, front);
  listSize++;
  if (remIndex != -1)
  {
    remIndex++;
  }
}
```

PART C:

SomeListType	printForward	printReverse
LinkedList<Object>	$O(n^2)$	$O(n^2)$
APList	$O(n)$	$O(n^2)$

(a) Write the `APList` method `get`. This method returns the value contained in the list node at index `n`. If the index `n` is greater than or equal to the remembered index, the method should start its traversal at the remembered node; otherwise, the method should start at the front of the list. The remembered node and index should be updated to refer to the node at the given index.

Complete method `get` below.

```
/** Gets a value at a given index in this list.
 *  @param n the index at which to get a value
 *         Precondition: 0 ≤ n < size()
 *  @return the object at the given index
 *  Postcondition: The remembered node and index refer to the node at index n
 */
public Object get(int n)
{ if(n > remIndex && remIndex != -1)
    {
      if (n >= listSize)
       { remIndex = -1
         return null;}
      for(int x= remIndex; x < n; x++)
       { remNode = remNode.getNext();}
       remIndex= n;
       return remNode.getValue();
    }
  else
   { remNode = front;
     if( n >= listSize)
      { remIndex = -1;
        return null;}
      for( int x=0; x<n; x++)
      { remNode = remNode.getNext();}
        remIndex = n;
        return remNode.getValue();
   }
}
```

GO ON TO THE NEXT PAGE.

(b) Write the `APList` method `addFirst`. Instance variables should be updated as necessary. This method should not change the value of `remNode` since there is no advantage to remembering a node at the front of the list.

Complete method `addFirst` below.

```
/** Adds a new node containing obj to the front of this list.
 *  @param obj the value to add to the list
 */
public void addFirst(Object obj)
{ ListNode temp = new ListNode(obj, front);
  front = temp;
  if(remIndex != -1)
     remIndex++;
  listSize++;
}
```

Part (c) begins on page 12.

(c) Consider the following methods.

```
public static void printForward(SomeListType myList)
{
  int n = myList.size();
  for (int k = 0; k < n; k++)
  {
    Object obj = myList.get(k);
    System.out.println(obj);
  }
}


public static void printReverse(SomeListType myList)
{
  int n = myList.size();
  for (int k = n - 1; k >= 0; k--)
  {
    Object obj = myList.get(k);
    System.out.println(obj);
  }
}
```

Give the big-Oh running time (in terms of n) of these methods for the following list types, where n is the number of elements in the list.

SomeListType	printForward	printReverse
LinkedList<Object>	$O(n^2)$	$O(n^2)$
APList	$O(n)$	$O(n^2)$

GO ON TO THE NEXT PAGE.

AB24

(a) Write the `APList` method `get`. This method returns the value contained in the list node at index `n`. If the index `n` is greater than or equal to the remembered index, the method should start its traversal at the remembered node; otherwise, the method should start at the front of the list. The remembered node and index should be updated to refer to the node at the given index.

Complete method `get` below.

```
/** Gets a value at a given index in this list.
 *  @param n the index at which to get a value
 *         Precondition: 0 ≤ n < size()
 *  @return the object at the given index
 *  Postcondition: The remembered node and index refer to the node at index n
 */
public Object get(int n)
{
    if(n == remIndex)
       return remNode.getValue();
else if(n > remIndex)
    { while (remNode.getNext()!= null && remIndex != n)
       {
         remNode = remNode.getNext();
         remIndex ++;
       }
       return remNode.getValue();
    }
    else
    {
       remIndex = 0;
       remNode = front;
       while (remNode.getNext()!=null) && remIndex!=n)
       {
         remNode = remNode.getNext();
         remIndex++;
       }
       return remNode.getValue();
    }
}
```

GO ON TO THE NEXT PAGE.

AB2b

(b) Write the `APList` method `addFirst`. Instance variables should be updated as necessary. This method should not change the value of `remNode` since there is no advantage to remembering a node at the front of the list. 2

Complete method `addFirst` below.

```
/** Adds a new node containing obj to the front of this list.
 *  @param obj the value to add to the list
 */
public void addFirst(Object obj)
{
    ListNode oldFront = front;
    front = new ListNode(obj, oldFront);
    listSize++;
    if(remIndex > 0)
        remIndex++;
}
```

Part (c) begins on page 12.

AB26
3

(c) Consider the following methods.

```
public static void printForward(SomeListType myList)
{
  int n = myList.size();
  for (int k = 0; k < n; k++)
  {
    Object obj = myList.get(k);
    System.out.println(obj);
  }
}


public static void printReverse(SomeListType myList)
{
  int n = myList.size();
  for (int k = n - 1; k >= 0; k--)
  {
    Object obj = myList.get(k);
    System.out.println(obj);
  }
}
```

Give the big-Oh running time (in terms of n) of these methods for the following list types, where n is the number of elements in the list.

SomeListType	printForward	printReverse
LinkedList<Object>	O(n)	O(n)
APList	O(n)	O(n)

GO ON TO THE NEXT PAGE.

AB2C, 1

(a) Write the `APList` method `get`. This method returns the value contained in the list node at index `n`. If the index `n` is greater than or equal to the remembered index, the method should start its traversal at the remembered node; otherwise, the method should start at the front of the list. The remembered node and index should be updated to refer to the node at the given index.

Complete method `get` below.

```
/** Gets a value at a given index in this list.
 *  @param n the index at which to get a value
 *         Precondition: 0 ≤ n < size()
 *  @return the object at the given index
 *  Postcondition: The remembered node and index refer to the node at index n
 */
public Object get(int n)
{
    int startHere = -1;
    Object mystery;
    if(n >= remIndex)
    {
        startHere = remIndex;
        mystery = remNode;
    }
    else
    {
        startHere = 0;
        mystery = front;
    }
    while((count < n) && (count < size()))
    {
        mystery = remNode.getNext();
    }
    remIndex = n;
    remNode = mystery;
    return mystery;
}
```

AB2C 2

(b) Write the `APList` method `addFirst`. Instance variables should be updated as necessary. This method should not change the value of `remNode` since there is no advantage to remembering a node at the front of the list.

Complete method `addFirst` below.

```
/** Adds a new node containing obj to the front of this list.
 *  @param obj the value to add to the list
 */
public void addFirst(Object obj)
{
    ListNode firstNode = new ListNode(obj, null);
    if (!front.equals(null))
    {
        firstNode.setNext(front);
    }
    front = firstNode;
}
```

Part (c) begins on page 12.

AB2 C3

(c) Consider the following methods.

```
public static void printForward(SomeListType myList)
{
  int n = myList.size();
  for (int k = 0; k < n; k++)
  {
    Object obj = myList.get(k);
    System.out.println(obj);
  }
}


public static void printReverse(SomeListType myList)
{
  int n = myList.size();
  for (int k = n - 1; k >= 0; k--)
  {
    Object obj = myList.get(k);
    System.out.println(obj);
  }
}
```

Give the big-Oh running time (in terms of n) of these methods for the following list types, where n is the number of elements in the list.

SomeListType	printForward	printReverse
LinkedList<Object>	$O(n)$	$O(n)$
APList	$O(\frac{n}{2})$	$O(\frac{n}{2})$

GO ON TO THE NEXT PAGE.

AP® COMPUTER SCIENCE AB
2008 SCORING COMMENTARY

Question 2

Overview

This question focused on creating, updating, and effectively accessing a linked structure of `ListNodes`. The linked structure described in the problem extended a simple linked list by adding additional state (a reference to the last accessed node and its index). In part (a) students were required to implement the `get` method of the `APList` class, which involved using the state variables to more efficiently locate the desired value. In part (b) students had to implement the `addFirst` method, which meant adding the value to the list and updating the state variables appropriately. Finally, part (c) required reasoning about the efficiency of this implementation when compared with a standard linked list structure. Two code samples were provided, and students had to identify the Big-Oh complexity of each sample using each list structure.

Sample: AB2a
Score: 8

In part (a) 1 point was earned for correctly determining the start location in the first call to `get`. The student did not earn 1 point for correctly determining the start location for subsequent calls to `get`; if `n = remIndex`, the student's traversal starts at `front` rather than `remNode`. The student earned ½ point for the call to `getNext` and ½ point for accessing more than one successive node. The student earned 1 point for identifying the desired node. The student earned ½ point for correctly updating `remNode` and `remIndex`. The student earned ½ point for returning the value at the identified node.

In part (b) the student earned ½ point for creating a new `ListNode` with `obj` and `front` and ½ point for updating `front` to refer to the new node. The student earned ½ point for incrementing `listSize`. The student earned 1 point for a properly guarded increment of `remIndex.`

In part (c) the student earned ½ point for $O(n^2)$ for `LinkedList printForward` and `printReverse`, ½ point for $O(n)$ for `APList printForward`, and ½ point for $O(n^2)$ for `APList printReverse`.

Sample: AB2b
Score: 6

In part (a) the student did not earn a point for correctly determining the start location in the first call to `get`; the student's traversal starts at `remNode` rather than `front`. The student earned 1 point for correctly determining the start location for subsequent calls to `get`, ½ point for the call to `getNext`, and ½ point for accessing more than one successive node. The student earned 1 point for identifying the desired node, ½ point for correctly updating `remNode` and `remIndex`, and ½ point for returning the value at the identified node.

In part (b) the student earned ½ point for creating a new `ListNode` with `obj` and `front` and ½ point for updating `front` to refer to the new node. The student earned ½ point for updating `listSize` but did not earn 1 point for a properly guarded increment of `remIndex`; when `remIndex = 0`, the increment should occur.

In part (c) the student did not earn ½ point for $O(n^2)$ for `LinkedList printForward` and `printReverse`. The student earned ½ point for $O(n)$ for `APList printForward` but did not earn ½ point for $O(n^2)$ for `APList printReverse`.

Question 2 (continued)

Sample: AB2c
Score: 3

In part (a) the student did not earn 1 point for correctly determining the start location in the first call to `get`; the student's traversal starts at `remNode` rather than `front`. The student earned 1 point for correctly determining the start location for subsequent calls to `get` and ½ point for the call to `getNext`. The student did not earn ½ point for accessing more than one successive node and did not earn 1 point for identifying the desired node; `mystery` is always set to the successor to `remNode`. The student earned ½ point for correctly updating `remNode` and `remIndex` in the context of what the student has written in the loop. The student did not earn ½ point for returning the value at the identified node; a call to `getValue` is required.

In part (b) the student did not earn ½ point for creating a new `ListNode` with `obj` and `front`; one cannot use `equals()` to test for `null` values. The student earned ½ point for updating `front` to refer to the new node. The student did not earn ½ point for updating `listSize` or 1 point for a guarded increment of `remIndex`.

In part (c) the student did not earn ½ point for $O(n^2)$ for `LinkedList printForward` and `printReverse`. The student earned ½ point for $O(n)$ for `APList printForward` but did not earn ½ point for $O(n^2)$ for `APList printReverse`.