COMPUTER SCIENCE AB
SECTION II

Time—1 hour and 45 minutes
Number of questions—4
Percent of total grade—50

Directions: SHOW ALL YOUR WORK. REMEMBER THAT PROGRAM SEGMENTS ARE TO BE WRITTEN IN Java.

Write your answers in pencil only in the booklet provided.

Notes:

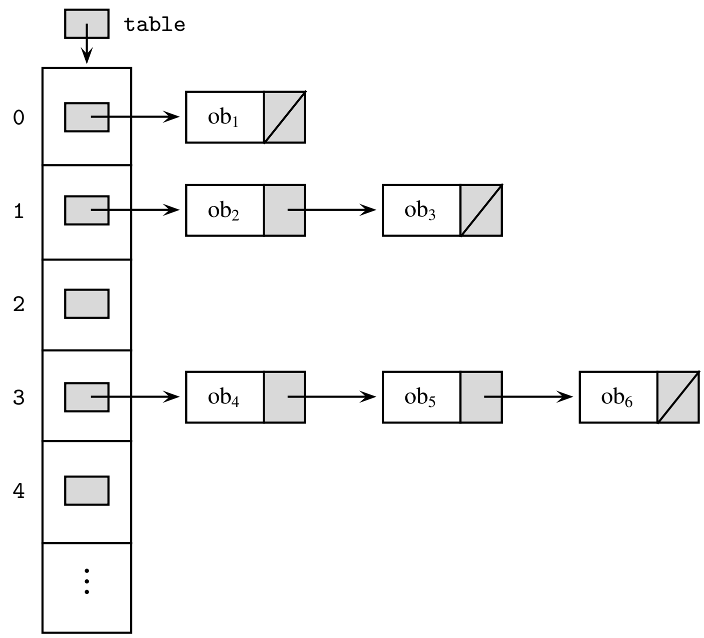
- Assume that the classes in the Quick Reference have been imported where needed.
- Assume that the implementation classes `ListNode` and `TreeNode` are used for any questions referring to linked lists or trees, unless otherwise specified.
- `ListNode` and `TreeNode` parameters may be `null`. Otherwise, unless noted in the question, assume that parameters in method calls are not `null`, and that methods are called only when their preconditions are satisfied.
- In writing solutions for each question, you may use any of the accessible methods that are listed in classes defined in that question. Writing significant amounts of code that can be replaced by a call to one of these methods may not receive full credit.

1. Consider a hash table that stores table entries of some type (`DataType`) with an associated key. Assume that classes `TableEntry` and `DataType` have been declared as follows:

```
/* Hash table entry. Consists of data and associated key */
public class TableEntry
{
    private Object key;
    private DataType data;

    public TableEntry(Object theKey, DataType theData)
    {
        key = theKey;
        data = theData;
    }

    public Object getKey()
    { return key; }

    public String toString()
    { return "" + key + "  " + data; }
}
```

```
public class DataType
{
    //private instance variables
        ...
    //constructor
        ...
    //toString method for DataType object
        ...
}
```

Assume that the hash table will be implemented using an `ArrayList` of linked lists, called buckets. Each linked list or bucket will contain table entries with the same hash address.



The diagram shows, for example, that ob_2 and ob_3 have hash address 1, while ob_4, ob_5, and ob_6 have hash address 3. It also shows that buckets labeled 2 and 4 are empty, since no entries have been inserted into them.

Here is the declaration for the `HashTable` class:

```
public class HashTable
{
    private int size;   //number of ArrayList elements (buckets)
    private int count;  //number of table entries
    //Each bucket is a LinkedList of TableEntry objects
    private ArrayList<LinkedList<TableEntry>> table;

    //constructor
    public HashTable(int tableSize)
    { /* to be implemented in part (a) */ }

    //Hash function returns hash address.
    public int hash(Object key)
    { return key.hashCode() % size; }
```

```
    //Delete TableEntry with given key.
    //Do nothing if key not in table.
    //Precondition: key does not occur more than once in table.
    public void delete(Object key)
    { /* to be implemented in part (b) */ }

    //Insert TableEntry with data and associated key into HashTable.
    //If key already in table, replace existing data with new data.
    //Postcondition: key occurs exactly once in table.
    public void insert(Object key, DataType data)
    { /* to be implemented in part (c) */ }

    //Print contents of HashTable.
    public void printTable()
    { /* implementation not shown */ }
}
```

(a) Complete the constructor for the `HashTable` class. Note that the `tableSize` parameter is the number of `ArrayList` elements (buckets). Each bucket in the list is initially empty since the number of table entries is zero.

```
//constructor
public HashTable(int tableSize)
```

(b) Write the `delete` method for the `HashTable` class. Method `delete` removes the `TableEntry` with the given `key`. If `key` is not in the table, the method does nothing.

Complete method `delete` as started below:

```
//Delete TableEntry with given key.
//Do nothing if key not in table.
//Precondition: key does not occur more than once in table.
public void delete(Object key)
```

(c) Write the `insert` method for the `HashTable` class. Method `insert` first removes any `TableEntry` with the given key. Then it inserts `data` with the given `key` into the table.

In writing `insert`, you may call any other methods of the `HashTable` class. Assume that all these methods work as specified.

Complete method `insert` as started below:

```
//Insert TableEntry with data and associated key into HashTable.
//If key already in table, replace existing data with new data.
//Postcondition: key occurs exactly once in table.
public void insert(Object key, DataType data)
```

2. This question involves reasoning about the code from the GridWorld Case Study. A Quick Reference to the case study is provided as part of this exam.

   In this question you will write three new methods for the `AbstractGrid` class. The current class manipulates and accesses adjacent neighboring locations. The new methods will be used to access "two-away" neighboring locations for any given location.

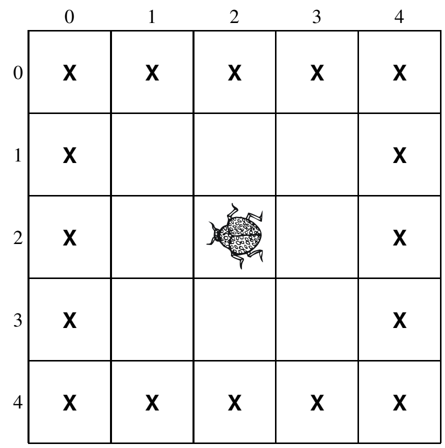


Fig. 1

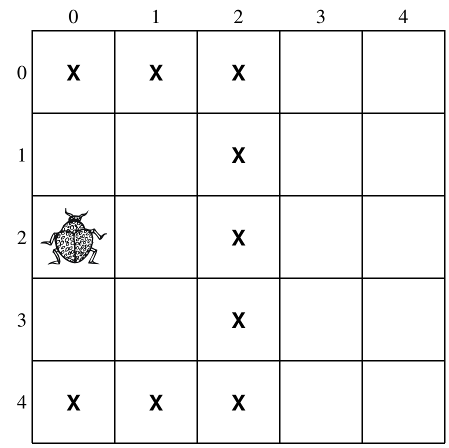


Fig. 2

   The bug in Fig. 1 has 16 valid two-away neighboring locations (marked with **X**'s), while the bug in Fig. 2 has nine. Notice that each of these locations is itself adjacent to one of the bug's adjacent neighboring locations.
   In writing the new methods below, you may use any of the existing methods in the `AbstractGrid` class.

   (a) Write a helper method called `addLoc` that has the following header:

```
private void addLoc(ArrayList<Location> list, Location loc, int dir)
```

   Method `addLoc` finds the location that is adjacent to `loc` in direction `dir`. If this adjacent location is valid, the method adds it to `list`, an existing `ArrayList` of `Location` objects. Note: This method does not deal with two-away locations. For example, in the bounded grids shown above, if the `addLoc` method is called with `loc` = (2, 1) and `dir` = `Location.RIGHT`, it will add location (2, 2) to the list. If it is called with `loc` = (1, 0) and `dir` = `Location.LEFT`, it will not add anything to `list` since the location to the left of (1, 0) is invalid.

   Complete method `addLoc` below.

```
/**
 * Adds to list the adjacent location that is in the given direction
 * from loc, if that location is valid.
 * Precondition: loc is valid in this grid.
 * @param list a list of Location objects
 * @param loc a given location
 * @param dir a specified direction from loc
 */
private void addLoc(ArrayList<Location> list, Location loc, int dir)
```

   (b) Write an `AbstractGrid` method `getValidTwoAwayLocations` that has the following header:

```
public ArrayList<Location> getValidTwoAwayLocations(Location loc)
```

This method should return all valid locations in the grid that are two-away from `loc`, where `loc` is a valid location in the grid. Note: You may use helper method, `addLoc`, specified in part (a), but you are not required to do so. If you do use the helper method, you may assume that it works as specified, irrespective of what you wrote.

Complete method `getValidTwoAwayLocations` below.

```
/**
 * Gets a list of valid two-away locations for loc.
 * Precondition: loc is valid in this grid.
 * @param loc the specified location
 * @return a list of valid two-away locations for loc
 */
public ArrayList<Location> getValidTwoAwayLocations(Location loc)
```

(c) Write an `AbstractGrid` method `getOccupiedTwoAwayLocations` that has the following header:

```
public ArrayList<Location> getOccupiedTwoAwayLocations(Location loc)
```

This method should return all valid occupied locations that are two-away from `loc` in this grid, where `loc` is a valid location in the grid. In writing `getOccupiedTwoAwayLocations`, you may call either of the methods in parts (a) or (b). You may assume that they work as specified, irrespective of what you wrote.

Complete method `getOccupiedTwoAwayLocations` below.

```
/**
 * Gets a list of valid occupied two-away locations for loc.
 * Precondition: loc is valid in this grid.
 * @param loc a specified location
 * @return a list of occupied two-away locations for loc
 */
public ArrayList<Location> getOccupiedTwoAwayLocations(Location loc)
```

3. Consider designing a simple line-oriented text editor. The text editor maintains a current line pointer and pointers to the first and last lines of the text. Each line of text is stored as a string. The text itself is stored as a linear doubly linked list of lines. The operations supported by the text editor are described in the following `TextEditor` class.

```
public class TextEditor
{
    private DoublyListNode current;    //refers to the current line
    private DoublyListNode topPtr;     //refers to the top line
    private DoublyListNode bottomPtr;  //refers to the bottom line

    //Constructor. Reads in lines of text from an input file inFile.
    //Precondition:  inFile is open for reading and contains at least
    //               one line of text.
    //Postcondition: All lines of inFile inserted into TextEditor.
    //               topPtr and current point to first line.
    //               bottomPtr points to last line.
    public TextEditor(FileReader inFile)
    { /* to be implemented in part (b) */ }
```

GO ON TO THE NEXT PAGE.

```
    //Moves current line pointer to next line, if line exists.
    //If current == bottomPtr, does nothing.
    public void next()
    { /* implementation not shown */ }

    //Moves current line pointer to previous line, if line exists.
    //If current == topPtr, does nothing.
    public void previous()
    { /* implementation not shown */ }

    //Moves current line pointer to first line.
    public void top()
    { /* implementation not shown */ }

    //Moves current line pointer to last line.
    public void bottom()
    { /* implementation not shown */ }

    //Precondition:  current and bottomPtr are not null.
    //Postcondition: line inserted following line pointed to by current.
    //               current remains unchanged.
    //               bottomPtr is updated to point to last line, if necessary.
    public void insert(String line)
    { /* to be implemented in part (a) */ }

    //Prints line pointed to by current line pointer to screen.
    //Postcondition: current line pointer still points to that line.
    public void printLine()
    { /* implementation not shown */ }

    //Returns true if current line pointer points to last line,
    // otherwise returns false.
    public boolean atEnd()
    { /* implementation not shown */ }
}
```

The doubly linked list for the text editor is implemented with the `DoublyListNode` class below:

```
public class DoublyListNode
{
    private Object value;
    private DoublyListNode next, prev;

    public DoublyListNode(DoublyListNode initPrev, Object initValue,
                    DoublyListNode initNext)
    {
        prev = initPrev;
        value = initValue;
        next = initNext;
    }

    public DoublyListNode getPrev()
    { return prev; }
```

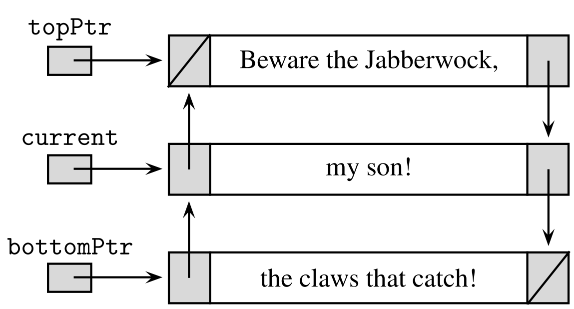
```
    public void setPrev(DoublyListNode theNewPrev)
    { prev = theNewPrev; }

    public Object getValue()
    { return value; }

    public void setValue(Object theNewValue)
    { value = theNewValue; }

    public DoublyListNode getNext()
    { return next; }

    public void setNext(DoublyListNode theNewNext)
    { next = theNewNext; }
}
```

The text editor uses the following `FileReader` class to read lines of text from an external file.

```
public class FileReader
{
    //Returns next line of file.
    public String getOneLine()
    { /* implementation not shown */ }

    //Returns true if the last line of the file
    // has been read, otherwise returns false.
    public boolean endOfFile()
    { /* implementation not shown */ }

    //other methods and private instance variables not shown
        ...
}
```

(a) Write the implementation code for the `insert` method as started below. Method `insert` should insert its parameter `line` after the line pointed to by the current pointer. After insertion, `current` should be unchanged, but `bottomPtr` should be adjusted to point to the last line of text if the insertion occurred at the last line.

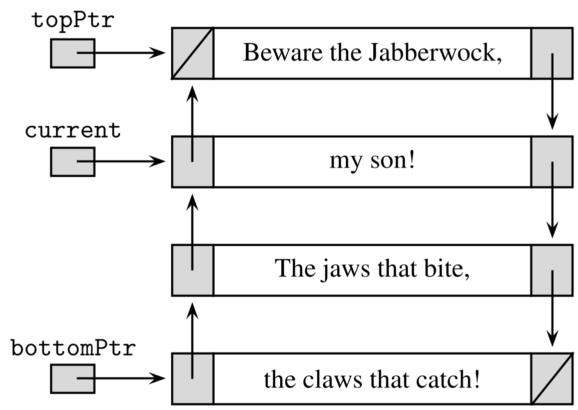
For example, if this is the state of the `TextEditor t`



the method call

```
t.insert("The jaws that bite,");
```

should result in



Complete method `insert` below:

```
//Precondition:  current and bottomPtr are not null.
//Postcondition: line inserted following line pointed to by current.
//               current remains unchanged.
//               bottomPtr is updated to point to last line, if necessary.
public void insert(String line)
```

(b) Write the implementation code for the `TextEditor` constructor. The constructor reads lines from the file with name `fileName`. You may assume that the file is open for reading and contains at least one line of text. Note that the postcondition specifies that both `current` and `topPtr` should be initialized to the top line of text, while `bottomPtr` should point to the last line of text.

In writing the constructor, you may wish to call the `insert` method specified in part (a), as well as the `getOneLine` and `endOfFile` methods of the `FileReader` class. You may assume that all of these methods work as specified.

Complete the constructor below:

```
//Constructor. Reads in lines of text from an input file inFile.
//Precondition:  inFile is open for reading and contains at least
//               one line of text.
//Postcondition: All lines of inFile inserted into TextEditor.
//               topPtr and current point to first line.
//               bottomPtr points to last line.
public TextEditor(FileReader inFile)
```

GO ON TO THE NEXT PAGE.

(c) A client method `printAlternate` prints every second line of text, starting with the first line and proceeding to the end of the text. In writing `printAlternate`, you may assume that all methods of the `TextEditor` class work as specified.

Write method `printAlternate` as started below:

```
//Precondition:  t contains at least one line of text.
//Postcondition: Alternate lines of text have been printed to
//               the screen, starting with the first line.
public static void printAlternate(TextEditor t)
```

4. Assume that binary search trees are implemented with the `TreeNode` class provided.

This question refers to the `BinaryTree` class below:

```
public class BinaryTree
{
    private TreeNode root;

    public BinaryTree()
    { root = null; }

    public TreeNode getRoot()
    { return root; }

    public void setRoot(TreeNode theNewNode)
    { root = theNewNode; }

    public boolean isEmpty()
    { return root == null; }

    public void postorder()
    { doPostorder(root); }

    //private helper method
    //Uses an iterative method to print the elements of t, postorder.
    private void doPostorder(TreeNode t)
    { /* to be implemented in part (b) */ }

    //other traversal methods, and methods to insert and find elements not shown
        ...
}
```

Consider the problem of writing an *iterative* algorithm for a postorder traversal of a binary tree. The iterative version of the traversal simulates recursion by maintaining a stack of tree nodes, each of which is labeled 1, 2, or 3. The stack stores nodes that have been visited, but whose recursive calls are not yet complete. The top of the stack represents the current node being visited, which can be at one of three places in the algorithm, indicated by its label:

label = 1: about to make a recursive call to the left subtree
label = 2: about to make a recursive call to the right subtree
label = 3: about to process the current node

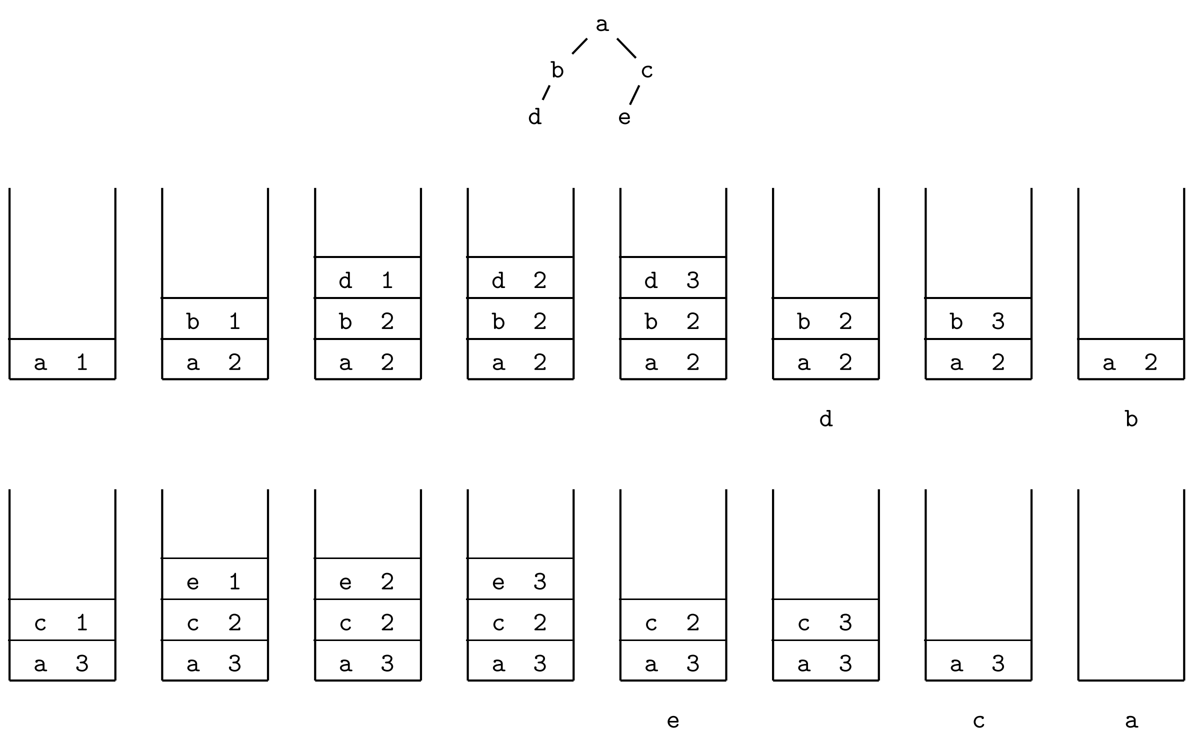
Thus, each node is placed on the stack three times during the postorder traversal. The third time the node is popped, it is processed.

GO ON TO THE NEXT PAGE.

Here is a summary of the algorithm:

1. Initialize the traversal by pushing the root onto the stack. Label it 1.
2. While the stack is not empty,
   - Pop the stack.
   - If the label is 1,
     (i) Increment the label and push that node back onto the stack.
     (ii) Push the root of its left subtree (if there is one) onto the stack with a label of 1.
   - If the label is 2,
     (i) Increment the label and push that node back onto the stack.
     (ii) Push the root of its right subtree (if there is one) onto the stack with a label of 1.
   - If the label is 3,
     (i) Process the node.

For example, here is the state of the stack for a postorder traversal of the tree shown.



The postorder traversal is `dbeca`.

(a) The algorithm requires a `StackNode` object. Write a `StackNode` class in which a `StackNode` has a `TreeNode` and an integer label. When a `StackNode` is constructed, it must be assigned a `TreeNode` and a label value of 1. The `StackNode` should have the following operations:

- Retrieve its `TreeNode`.
- Retrieve the value of its label.
- Increment its label by 1.

Write the `StackNode` class below.

GO ON TO THE NEXT PAGE.

(b) Write the implementation of the `doPostorder` helper method. Your algorithm should be iterative and should print the elements in the tree, one per line, with a postorder traversal. You may assume that the objects in the tree have a `toString` method defined. You may also assume that your `StackNode` class works as described, irrespective of what you wrote in part (a).

Complete the `doPostorder` method below.

```
//private helper method
//Uses an iterative method to print the elements of t, postorder.
private void doPostorder(TreeNode t)
```

END OF EXAMINATION