

Section II

1. (a)

```
public HashTable(int tableSize)
{
    table = new ArrayList<LinkedList<TableEntry>>();
    size = tableSize;
    count = 0;
    for (int i = 0; i < size; i++)
    {
        table.add(new LinkedList<TableEntry>());
    }
}
```
- (b)

```
public void delete(Object key)
{
    int pos = hash(key);
    boolean found = false;
    ListIterator<TableEntry> i = table.get(pos).listIterator();
    while (i.hasNext() && !found)
    {
        if (i.next().getKey().equals(key))
        {
            i.remove();
            found = true;
            count--;
        }
    }
}
```
- (c)

```
public void insert(Object key, DataType data)
{
    int pos = hash(key);
    TableEntry entry = new TableEntry(key, data);
    delete(key);          //does nothing if key not in table
    table.get(pos).addFirst(entry);
    count++;
}
```

NOTE

- The for loop in part (a) initializes each of size slots in table to be an empty `LinkedList`.
- Note that a for-each loop will not work in part (b): You need to use an iterator explicitly if an element may be removed from the `LinkedList` being traversed.
- In parts (b) and (c), the hash method determines which `LinkedList` must be searched.
- The boolean variable `found` in part (b) is used to exit the loop if key is found. It is inefficient to continue the traversal after key is found, since key does not occur more than once in the table.
- In part (c), `delete(key)` must be done before adding new data, otherwise the postcondition will be violated.
- In part (c), when inserting into the `LinkedList` `table.get(pos)`, it is also correct to use `addLast(entry)` instead of `addFirst(entry)`.

2 PRACTICE EXAMS

2. (a)

```
private void addLoc(ArrayList<Location> list, Location loc, int dir)
{
    Location adj = loc.getAdjacentLocation(dir);
    if (isValid(adj))
        list.add(adj);
}
```

(b) Here is a solution that uses the helper method addLoc.

```
public ArrayList<Location> getValidTwoAwayLocations(Location loc)
{
    ArrayList<Location> locs = new ArrayList<Location>();
    int d = Location.NORTH;
    for (int i = 0;
        i < Location.FULL_CIRCLE / Location.HALF_RIGHT; i++)
    {
        Location neighborLoc = loc.getAdjacentLocation(d);
        Location twoAwayLoc = neighborLoc.getAdjacentLocation(d);
        if (isValid(twoAwayLoc))
            locs.add(twoAwayLoc);

        if (d == Location.NORTH || d == Location.NORTHWEST)
            addLoc(locs, twoAwayLoc, Location.RIGHT);
        else if (d == Location.NORTHEAST || d == Location.EAST)
            addLoc(locs, twoAwayLoc, Location.HALF_CIRCLE);
        else if (d == Location.SOUTHEAST || d == Location.SOUTH)
            addLoc(locs, twoAwayLoc, Location.LEFT);
        else if (d == Location.SOUTHWEST || d == Location.WEST)
            addLoc(locs, twoAwayLoc, Location.AHEAD);

        d = d + Location.HALF_RIGHT;
    }
    return locs;
}
```

Alternatively, here is a solution that does not use the helper method.

```
public ArrayList<Location> getValidTwoAwayLocations(Location loc)
{
    ArrayList<Location> twoAwayLocs = new ArrayList<Location>();
    ArrayList<Location> adjLocs = getValidAdjacentLocations(loc);
    Set<Location> adjLocsSet = new HashSet<Location>(adjLocs);
    Set<Location> manyLocsSet = new HashSet<Location>();

    for (Location locat : adjLocs)
    {
        for (Location l : getValidAdjacentLocations(locat))
            manyLocsSet.add(l);
    }

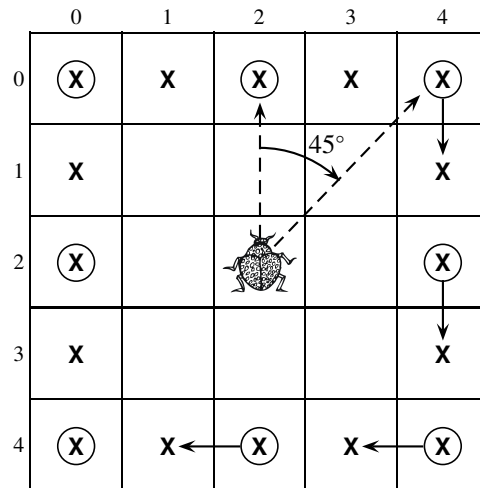
    for (Location l : manyLocsSet)
        if (!l.equals(loc) && !adjLocsSet.contains(l))
            twoAwayLocs.add(l);

    return twoAwayLocs;
}
```

```
(c) public ArrayList<Location> getOccupiedTwoAwayLocations(Location loc)
{
    ArrayList<Location> locs = new ArrayList<Location>();
    for (Location twoAwayLoc : getValidTwoAwayLocations(loc))
    {
        if (get(twoAwayLoc) != null)
            locs.add(twoAwayLoc);
    }
    return locs;
}
```

NOTE


- The solution in part (b) that uses the helper method gets each two-away location in the compass directions from `loc` (the circled locations below), using a similar algorithm to that used in `getValidAdjacentLocations`.



The locations with the uncircled **X**'s are added to the list using the helper method. For example, notice that if the direction `d` is 45 (which gets location (0, 4)), or 90 (which gets location (2, 4)), you want to add (1, 4) and (3, 4) to the list, namely the locations that are `Location.HALF_CIRCLE` from (0, 4) and (2, 4). When `d` is 135 or 180, you want to add the locations that are `Location.LEFT` from (4, 4) and (4, 2). The idea is that for each value of `d`, you get the “easy-to-reach” two-away location in that compass direction, and then use the helper method with the appropriate parameters to get the “in-between” two-away location.

- Here are the steps in the alternative solution to part (b).
 - Get all of the valid adjacent locations for `loc`.
 - Place these adjacent locations in a set, `adjLocsSet`.
 - Now create another set, `manyLocsSet`. This set gets all of the valid adjacent locations of each location in `adjLocsSet`.
 - Traverse `manyLocsSet`, and for each location in it, if that location is not equal to `loc` and is not adjacent to `loc`, add it to the two-away list.

4 PRACTICE EXAMS

| | | | | | | |
|--|---|---|--|---|---|--|
| | | | | | | |
| | X | X | X | X | X | |
| | X | O | O | O | X | |
| | X | O |  | O | X | |
| | X | O | O | O | X | |
| | X | X | X | X | X | |
| | | | | | | |

Refer to the diagram. Suppose `loc` is where the bug is. It is required that you return the list of locations with **X**'s in them. The circled locations go into `adjLocsSet`. The shaded locations go into `manyLocsSet`. (Note that since a set is being used, there are no duplicates.) In the final step of the algorithm, all of the non-**X** locations are weeded out with the test

```
if (!l.equals(loc) && !adjLocsSet.contains(l))
```

- The statement in the alternative solution,

```
Set<Location> adjLocsSet = new HashSet<Location>(adjLocs);
```

that inserts the elements of an `ArrayList` into a `HashSet`, uses a constructor of `HashSet` that is not in the AP Java subset. The same effect can be achieved with a traversal of the `ArrayList`:

```
for (Location lo : adjLocs)
    adjLocsSet.add(lo);
```

```
3. (a) public void insert(String line)
    {
        DoublyListNode newLine = new DoublyListNode(current,
            line, current.getNext());
        if (!atEnd())
        {
            current.getNext().setPrev(newLine);
        }
        else
        {
            bottomPtr = newLine;
        }
        current.setNext(newLine);
    }
```

```

(b) public TextEditor(FileReader inFile)
    {
        String line = inFile.getOneLine();
        topPtr = new DoublyListNode(null, line, null);
        top();
        bottomPtr = topPtr;
        while (!inFile EOF())
        {
            line = inFile.getOneLine();
            insert(line);
            next();
        }
        top();
    }

(c) public static void printAlternate(TextEditor t)
    {
        t.top();
        t.printLine();
        while (!t.atEnd())
        {
            t.next();
            if (!t.atEnd())
            {
                t.next();
                t.printLine();
            }
        }
    }

```

NOTE

- There are two cases for part (a); either current points to the last line, or it doesn't. If it does, there's no prev field to be adjusted for a node that will follow the new node. The bottomPtr, however, must be adjusted.
- For part (b), the first node must be filled first because the precondition for insert requires that current and bottomPtr are not null. At the end of the algorithm, don't forget to reassign current to be at the top. You don't need to update bottomPtr—it's automatically updated by insert.
- Part (c) is tricky! You have to be sure to have the right combination of calls to next() and atEnd() so that your algorithm handles odd and even numbers of lines, as well as one line and two lines.

6 PRACTICE EXAMS

```
4. (a) public class StackNode
    {
        private TreeNode node;
        private int label;

        public StackNode(TreeNode n)
        {
            node = n;
            label = 1;
        }

        public TreeNode getNode()
        { return node; }

        public int getLabel()
        { return label; }

        public void incrementLabel()
        { label++; }
    }

    (b) private void doPostorder(TreeNode t)
    {
        Stack<StackNode> s = new Stack<StackNode>();
        if (t != null)
        {
            s.push(new StackNode(t));
            while (!s.isEmpty())
            {
                StackNode sNode = s.pop();
                if (sNode.getLabel() == 1)
                {
                    sNode.incrementLabel();
                    s.push(sNode);
                    if (sNode.getNode().getLeft() != null)
                        s.push(new StackNode(sNode.getNode().getLeft()));
                }
                else if (sNode.getLabel() == 2)
                {
                    sNode.incrementLabel();
                    s.push(sNode);
                    if (sNode.getNode().getRight() != null)
                        s.push(new StackNode(sNode.getNode().getRight()));
                }
                else //label == 3
                    System.out.println(sNode.getNode().getValue());
            }
        }
    }
```

NOTE Don't forget the three null tests!