Part 5 13331388 Zhuang Zijia
Set 10
1.The isValid is specified in Grid Interface.The BoudedGrid<E> and UnboundedGrid<E> provide an implementation of this method.

2.The method getValidAdjacentLocations() call the isValid method. Because it only need to judge if some grid is valid.

3.getOccupiedAdjacentLocations() and get() are called in the getNeighbors method.The AbstractGrid class implements the getOcuppiedAdjacentLocations method. The get() is implemented in the BoundedGrid<E> and UnboundedGrid<E>.

4.The get method is the only way to test whether or not a given locations is empty or occupied.

5.The number of possible valid adjacent locations would decrease from eight to four. The valid adjacent locations would be those that are north, south, east, and west of the given location.

Set 11
1.The BoundedGrid<E> will throw an IllegalArgumentException if rows <= 0 or cols <= 0.

2.The getNumCols() returns the number of columns in row 0 of the occupantArray. The BoundedGrid<E> ensures that each BoundedGrid object has at least one row and one column.

3.What are the requirements for a Location to be valid in a BoundedGrid?
A Location’s has at least one row and one column.

4.An ArrayList<Location> is returned by the getOccupiedLocations method.The time complexity for this method is O(r*c).

5.The type returned is E, which is whatever type is stored in the occupantArray.And it need a two-dimensional array.The time complexity for this method is O(1).

6.If the object's location isn't valid or the object is null, it will cause an exception to be thrown by the put method.The time complexity for this method is O(1).

7.The type E.If an attempt is made to remove an item from an empty location, null is stored in the location and null is returned.The time complexity for this method is O(1).

8.It is an efficient implementation.The getOccupiedLocations method is inefficient, because its time complexity is O(r * c). The other method are efficient, they both are O(1).

Set 12
1.The Location class must implement the hashCode and the equals methods.The TreeMap requires keys of the map to be Comparable.The Location class satisfies all of these requirements.

2.The UnboundedGrid methods get, put, and remove must check the location parameter and throw a NullPointerException when the parameter is null.For the BoundedGrid, there are others method like isValid() to judge if a location is null and throw its exception.

3.The average time complexity for the get, put, and remove is O(1). If a TreeMap were used instead of a HashMap, the average time complexity would be O(log n), where n is the number of occupied locations in the grid.

4.Most of the time, the getOccupiedLocations method will return the occupants in a different order.Keys (locations) in a HashMap are placed in a hash table based on an index that is calculated by using the keys' hashCode and the size of the table. The order in which a key is visited when the keySet is traversed depends on where it is located in the hash table.
A TreeMap stores its keys in a balanced binary search tree and traverses this tree with an inorder traversal.The keys in the keySet (Locations) will be visited in ascending order (row major order) as defined by Location's compareTo method.

5.A map could be used to implement a bounded grid.If a HashMap were used to implement the bounded grid, the average time complexity for getOccupiedLocations would be O(n), where n is the number of items in the grid.

Exercise 1
/*
Suppose that a program requires a very large bounded grid that contains very few objects and that the program frequently calls the getOccupiedLocations method (as, for example, ActorWorld). Create a class SparseBoundedGrid that uses a "sparse array" implementation. Your solution need not be a generic class; you may simply store occupants of type Object.
The "sparse array" is an array list of linked lists. Each linked list entry holds both a grid occupant and a column index. Each entry in the array list is a linked list or is null if that row is empty.
You may choose to implement the linked list in one of two ways. You can use raw list nodes.
Or you can use a LinkedList<OccupantInCol> with a helper class.
*/

```
/* SparseBoundedGrid.java */
import info.gridworld.grid.Grid;
import info.gridworld.grid.AbstractGrid;
import info.gridworld.grid.Location;
import java.util.LinkedList;
import java.util.ArrayList;
/**
 * A <code>BoundedGrid</code> is a rectangular grid with a finite number of
 * rows and columns. <br />
 * The implementation of this class is testable on the AP CS AB exam.
 */
public class SparseBoundedGrid<E> extends AbstractGrid<E> {
   class OccupantInCol {
      private Object occupant;
      private int col;
      public OccupantInCol (Object obj, int colNum) {
         occupant = obj;
         col = colNum;
      }
      public Object getOccupant() {
         return occupant;
      }
```

```
    public int getCol() {
        return col;
    }
    public void setObj(Object obj) {
        occupant = obj;
    }
}
// the array storing the grid elements
private ArrayList<LinkedList<OccupantInCol> > occupantArray;
private int numCols;
private int numRows;

public SparseBoundedGrid(int rows, int cols) {
    if (rows <= 0) {
        throw new IllegalArgumentException("rows <= 0");
    }
    if (cols <= 0) {
        throw new IllegalArgumentException("cols <= 0");
    }
    numCols = cols;
    numRows = rows;
    occupantArray = new ArrayList<LinkedList<OccupantInCol>>();
    for (int i = 0; i < rows; i++) {
        occupantArray.add(new LinkedList<OccupantInCol>());
    }
}

public int getNumRows() {
    return numRows;
}

public int getNumCols() {
    return numCols;
}

public boolean isValid(Location loc) {
    return 0 <= loc.getRow() && loc.getRow() < getNumRows()
            && 0 <= loc.getCol() && loc.getCol() < getNumCols();
}

public ArrayList<Location> getOccupiedLocations() {
    ArrayList<Location> theLocations = new ArrayList<Location>();
    // Look at all grid locations.
    for (int r = 0; r < getNumRows(); r++) {
        //visit the arraylist and innitial the linkedlist
        LinkedList<OccupantInCol> list = occupantArray.get(r);
        if (list != null) {
            for (OccupantInCol occ : list) {
                Location loc = new Location(r, occ.getCol());
```

```
            if (get(loc) != null) {
                theLocations.add(new Location(r, occ.getCol()));
            }
          }
        }
    }
    return theLocations;
}

public E get(Location loc) {
    if (!isValid(loc)) {
        throw new IllegalArgumentException("Location " + loc + " is not valid");
    }
    int r = loc.getRow();
    LinkedList<OccupantInCol> list = occupantArray.get(r);
    if (list != null) {
        for (OccupantInCol occ : list) {
            if (loc.getCol() == occ.getCol()) {
                return (E) occ.getOccupant();
            }
        }
    }
    return null;
}

public E put(Location loc, E obj) {
    if (!isValid(loc)) {
        throw new IllegalArgumentException("Location " + loc + " is not valid");
    }
    if (obj == null) {
        throw new NullPointerException("obj == null");
    }
    // Add the object to the grid.
    int cou = 0;
    LinkedList<OccupantInCol> list = occupantArray.get(loc.getRow());
    E oldOccupant = get(loc);
    for (OccupantInCol occ : list) {
        if (occ.getCol() == loc.getCol()) {
            occ.setObj(obj);
            cou = 1;
            break;
        }
    }
        if (cou == 0) {
            occupantArray.get(loc.getRow()).add(new OccupantInCol(obj, loc.getCol()));
        }
    return oldOccupant;
}
```

```
    public E remove(Location loc) {
        if (!isValid(loc)) {
            throw new IllegalArgumentException("Location " + loc + " is not valid");
        }
        // Remove the object from the grid.
        E r = get(loc);
        if (r == null) {
            return null;
        }
        LinkedList<OccupantInCol> list = occupantArray.get(loc.getRow());
        int count = 0;
        for (OccupantInCol occ : list) {
            if (occ.getCol() == loc.getCol()) {
                list.remove(count);
                break;
            }
            count++;
        }
        return r;
    }
}
```

Exercise 2
Let r = number of rows, c = number of columns, and n = number of occupied locations.

Methods	SparseGridNode-version	LinkedList<OccupantInCol>-version	HashMap-version	TreeMap version
getNeighbors	O(c)	O(c)	O(1)	O(log n)
getEmptyAdjacentLocations	O(c)	O(c)	O(1)	O(log n)
getOccupiedAdjacentLocations	O(c)	O(c)	O(1)	O(log n)
getOccupiedLocations	O(r + n)	O(r + n)	O(n)	O(n)
get	O(c)	O(c)	O(1)	O(log n)
put	O(c)	O(c)	O(1)	O(log n)
remove	O(c)	O(c)	O(1)	O(log n)

```
/*
Consider using a HashMap or TreeMap to implement the SparseBoundedGrid. How could you use the
UnboundedGrid class to accomplish this task? Which methods of UnboundedGrid could be used without change?
*/

/* SparseBoundedGrid2.java */
import info.gridworld.grid.*;
import java.util.ArrayList;
import java.util.*;
```

```
/**
 * An <code>SparseBoundedGrid2</code> is a rectangular grid with an unbounded number of rows
and columns. <br />
 * The implementation of this class is testable on the AP CS AB exam.
 */
public class SparseBoundedGrid2<E> extends AbstractGrid<E> {
   private Map<Location, E> occupantMap;
   private int row;
   private int col;

   /**
    * Constructs an empty unbounded grid.
    */
   public SparseBoundedGrid2(int rows, int cols) {
      occupantMap = new HashMap<Location, E>();
      row = rows;
      col = cols;
   }

   public int getNumRows() {
      return row;
   }

   public int getNumCols() {
      return col;
   }

   public boolean isValid(Location loc) {
      return 0 <= loc.getRow() && loc.getRow() < getNumRows()
            && 0 <= loc.getCol() && loc.getCol() < getNumCols();
   }

   public ArrayList<Location> getOccupiedLocations() {
      ArrayList<Location> a = new ArrayList<Location>();
      for (Location loc : occupantMap.keySet()) {
         a.add(loc);
         }
      return a;
   }

   public E get(Location loc) {
      if (loc == null) {
         throw new NullPointerException("loc == null");
      }
      return occupantMap.get(loc);
   }

   public E put(Location loc, E obj) {
      if (loc == null) {
```

```
            throw new NullPointerException("loc == null");
        }
        if (obj == null) {
            throw new NullPointerException("obj == null");
        }
        return occupantMap.put(loc, obj);
    }

    public E remove(Location loc) {
        if (loc == null) {
            throw new NullPointerException("loc == null");
        }
        return occupantMap.remove(loc);
    }
}
```

Exercise 3

/*

Consider an implementation of an unbounded grid in which all valid locations have non-negative row and column values. The constructor allocates a 16 x 16 array. When a call is made to the put method with a row or column index that is outside the current array bounds, double both array bounds until they are large enough, construct a new square array with those bounds, and place the existing occupants into the new array.

Implement the methods specified by the Grid interface using this data structure. What is the Big-Oh efficiency of the get method? What is the efficiency of the put method when the row and column index values are within the current array bounds? What is the efficiency when the array needs to be resized?

*/

Big-Oh of get: O(1)
Big-Oh of put: O(1) when row and column index are within current array bounds
O(dim * dim), where dim is the number of rows and columns in the current array
before it is resized. Must traverse through.

```
/* UnboundedGrid2.java */
import info.gridworld.grid.Grid;
import info.gridworld.grid.AbstractGrid;
import info.gridworld.grid.Location;
import java.util.LinkedList;
import java.util.ArrayList;
import java.util.ArrayList;
/**
 * A <code>UnboundedGrid2</code> is a rectangular grid with a finite number of
 * rows and columns. <br />
 * The implementation of this class is testable on the AP CS AB exam.
 */
public class UnboundedGrid2<E> extends AbstractGrid<E> {
    // the array storing the grid elements
    private Object[][] occupantArray;
```

```
private int ini = 16;
public UnboundedGrid2() {
   occupantArray = new Object[ini][ini];
}

/**
 * Constructs an empty bounded grid with the given dimensions.
 * (Precondition: <code>rows > 0</code> and <code>cols > 0</code>.)
 * @param rows number of rows in BoundedGrid
 * @param cols number of columns in BoundedGrid
 */
/*public UnboundedGrid2(int rows, int cols) {
   if (rows <= 0)
      throw new IllegalArgumentException("rows <= 0");
   if (cols <= 0)
      throw new IllegalArgumentException("cols <= 0");
   occupantArray = new Object[rows][cols];
}
*/

public int getNumRows() {
   return -1;
}

public int getNumCols() {
   return -1;
}

public boolean isValid(Location loc) {
   return 0 <= loc.getRow() && 0 <= loc.getCol();
}

public ArrayList<Location> getOccupiedLocations() {
   ArrayList<Location> a = new ArrayList<Location>();
   // Look at all grid locations.
   for (int r = 0; r < ini; r++) {
      for (int c = 0; c < ini; c++) {
         // If there's an object at this location, put it in the array.
         Location loc = new Location(r, c);
         if (get(loc) != null) {
            a.add(loc);
         }
      }
   }
   return a;
}

public E get(Location loc) {
   if (!isValid(loc)) {
```

```
            throw new IllegalArgumentException("Location " + loc + " is not valid");
        }
        if (loc.getRow() >= ini || loc.getCol() >= ini) {
            return null;
        }
        return (E) occupantArray[loc.getRow()][loc.getCol()];
    }

    public E put(Location loc, E obj) {
        if (!isValid(loc)) {
            throw new IllegalArgumentException("Location " + loc + " is not valid");
        }
        if (obj == null) {
            throw new NullPointerException("obj == null");
        }
        if (loc.getRow() >= ini || loc.getCol() >= ini) {
            int size = ini;
            while (loc.getRow() >= size || loc.getCol() >= size) {
                size = size * 2;
            }s
            Object[][]temp = new Object[size][size];
            for (int row = 0; row < ini; row++) {
                for (int col = 0; col < ini; col++) {
                    temp[row][col] = occupantArray[row][col];
                }
            }
            occupantArray = temp;
            ini = size;
        }
        // Add the object to the grid.
        E oldOccupant = get(loc);
        occupantArray[loc.getRow()][loc.getCol()] = obj;
        return oldOccupant;
    }

    public E remove(Location loc) {
        if (!isValid(loc)) {
            throw new IllegalArgumentException("Location " + loc + " is not valid");
        }
        if (loc.getRow() >= ini || loc.getCol() >= ini) {
            return null;
        }
        // Remove the object from the grid.
        E r = get(loc);
        occupantArray[loc.getRow()][loc.getCol()] = null;
        return r;
    }
}
```