13331093
黄雄镖

Set 10

1. Where is the isValid method specified? Which classes provide an implementation of this method?

The isValid method is specified in the Grid interface, and implemented in the BoundedGrid and UnboundedGrid class.

2.Which AbstractGrid methods call the isValid method? Why don't the other methods need to call it?

The getValidAdjacentLocations method. Because the other methods can use getValidAdjacentLocations method, that is, to call the isValid method indirectly.

3.Which methods of the Grid interface are called in the getNeighbors method? Which classes provide implementations of these methods?

The getOccupiedAdjacentLocations and get methods. The AbstractGrid class provide implementations of getOccupiedAdjacentLocations, the BoundedGrid and the UnboundedGrid class provide implementations of get.

4.Why must the get method, which returns an object of type E, be used in the getEmptyAdjacentLocations method when this method returns locations, not objects of type E?

The getEmptyAdjacentLocations method use the get method to find whether the location is empty or not, then return the location.

5.What would be the effect of replacing the constant Location.HALF_RIGHT with Location.RIGHT in the two places where it occurs in the getValidAdjacentLocations method?

It will decrease the number of the valid adjacent locations get, become a half of the origin's.

Set 11

1.What ensures that a grid has at least one valid location?

If the number of rows or colums is less than 1, if will throw an IllegalArgumentException.

2.How is the number of columns in the grid determined by the getNumCols method? What assumption about the grid makes this possible?

The getNumCols method return the occupantArray[0].length, the constructor make it possible, occupantArray[0] must exist.

3.What are the requirements for a Location to be valid in a BoundedGrid?

The row number of the location should bigger than zero and smaller than the grid's row number, the colum number of the location should bigger than zero and smaller than the grid's colum number.

In the next four questions, let r = number of rows, c = number of columns, and n = number of occupied locations.

4.What type is returned by the getOccupiedLocations method? What is the time complexity (Big-Oh) for this method?

The returned type is ArrayList<Location>, the time complexity for this method is O(r*c)

5.What type is returned by the get method? What parameter is needed? What is the time complexity (Big-Oh) for this method?

The returned type is E, it need the Location as a parameter. The time complexity for this method is O(1).

6.What conditions may cause an exception to be thrown by the put method? What is the time complexity (Big-Oh) for this method?

If the location is not valid, it will cause an IllegalArgumentException, if the object is null, it will cause an NullPointerException. The time complexity for this method is O(1).

7.What type is returned by the remove method? What happens when an attempt is made to remove an item from an empty location? What is the time complexity (Big-Oh) for this method?

The returned type is E, if an attempt is made to remove an item from an empty location, it will return null. The time complexity for this method is O(1).

8.Based on the answers to questions 4, 5, 6, and 7, would you consider this an efficient implementation? Justify your answer.

Yes, the time complexity of many method in the class is O(1), and these method are use frequently, the lower complexity make it efficient. In a word, the implementation is very good.

Set 12

1.Which method must the Location class implement so that an instance of HashMap can be used for the map? What would be required of the Location class if a TreeMap were used instead? Does Location satisfy these requirements?

The Location class implement the hashCode method so that an instance of HashMap can be used for the map. Because the Location class implements the Comparable interface, the compareTo method must be implemented. If a TreeMap were used instead, the key is required. Yes, the Location satisfy these requirements.

2.Why are the checks for null included in the get, put, and remove methods? Why are no such checks included in the corresponding methods for the BoundedGrid?

Because in the unboundedgrid, it use a HashMap to store the object, the isValid method should return true all the time. Null is not a valid location or obj in the unboundedgrid, so the get or push or remove methods should checks for null, if it is null, throws the NullPointerException.In the BoundedGrid, it use the isValid to check whether it is valid or not before it try to get the occupantArray. If the method get null location, it will throw a NullPointerException.

3.What is the average time complexity (Big-Oh) for the three methods: get, put, and remove? What would it be if a TreeMap were used instead of a HashMap?

The time complexity for get method is O(1)
The time complexity for put method is O(1)
The time complexity for remove method is O(1)
If a TreeMap were used instead of a HashMap, the time complexity for them become O(n)

4.How would the behavior of this class differ, aside from time complexity, if a TreeMap were used instead of a HashMap?

Some of the object list order is different, such as the getOccupiedAdjacentLocations method. Because the TreeMap keeps the object in an order, it stores them in an balanced binary search tree.

5.Could a map implementation be used for a bounded grid? What advantage, if any, would the two-dimensional array implementation that is used by the BoundedGrid class have over a map implementation?

Yes, the time complexity for some method can reduce, such as the getOccupiedLocations method, use the map for a bounded grid can reduce its time complexity from O(r*c) to O(n). But if the bounded grid is very big and it contains many object, it may cost more memery, because the map need more space to store the key and value.

Exercises

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.

```
import info.gridworld.grid.Grid;
import info.gridworld.grid.AbstractGrid;
import info.gridworld.grid.Location;
import java.util.LinkedList;
import java.util.ArrayList;

/**
 * SparseBoundedGrid, LinkedList version
 */
public class SparseBoundedGrid<E> extends AbstractGrid<E>
{
    private ArrayList<LinkedList> occupantArray; // the array storing
the grid elements
    private int rowNum;
    private int colNum;

    public SparseBoundedGrid(int rows, int cols)
    {
        if (rows <= 0)
            throw new IllegalArgumentException("rows <= 0");
        if (cols <= 0)
            throw new IllegalArgumentException("cols <= 0");
```

```
    rowNum = rows;
    colNum = cols;
    initOccupantArray();
}

public int getNumRows()
{
    return rowNum;
}

public int getNumCols()
{
    return colNum;
}

private void initOccupantArray() {
    occupantArray = new ArrayList<LinkedList>();
 for  (int i = 0; i < rowNum; i++) {
      occupantArray.add(new LinkedList<OccupantInCol>());
 }
}

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

public ArrayList<Location> getOccupiedLocations()
{
    ArrayList<Location> theLocations = new ArrayList<Location>();

    // Look at all grid locations.
    for (int r = 0; r < getNumRows(); r++)
```

```
        {
            for (int c = 0; c < getNumCols(); c++)
            {
                // If there's an object at this location, put it in
the array.
                Location loc = new Location(r, c);
                if (get(loc) != null)
                    theLocations.add(loc);
            }
        }

        return theLocations;
    }

    public E get(Location loc)
    {
        if (!isValid(loc))
            throw new IllegalArgumentException("Location " + loc
                    + " is not valid");
        LinkedList<OccupantInCol> occupantCol =
occupantArray.get(loc.getRow());
        for (OccupantInCol object: occupantCol) {
          if (object.getCol() == loc.getCol()) {
               return (E) object.getObject();
          }
        }
        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.
        E oldOccupant = get(loc);
        if (oldOccupant != null)
        {
            LinkedList<OccupantInCol> occupantCol =
occupantArray.get(loc.getRow());
          for (OccupantInCol object: occupantCol) {
               if (object.getCol() == loc.getCol()) {
                    object.setObject(obj);
                    break;
               }
          }
        }
        else
        {
          (occupantArray.get(loc.getRow())).add(new
OccupantInCol(loc.getCol(), obj));
        }
        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);
        LinkedList<OccupantInCol> occupantCol =
occupantArray.get(loc.getRow());
```

```
        for (OccupantInCol object: occupantCol) {
          if (object.getCol() == loc.getCol()) {
               occupantCol.remove(object);
               break;
          }
     }
        return r;
    }
}
```

2. 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?

Fill in the following chart to compare the expected Big-Oh efficiencies for each implementation of the SparseBoundedGrid.

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

If using a HashMap or TreeMap to implement the SparseBoundedGrid, the get, put, remove and getOccupiedLocations methods of UnboundedGrid could be used without change.

```
import info.gridworld.grid.Grid;
import info.gridworld.grid.AbstractGrid;
import info.gridworld.grid.Location;
import java.util.HashMap;
import java.util.*;

/**
 * SparseBoundedGrid2, HashMap version
 */
public class SparseBoundedGrid2<E> extends AbstractGrid<E>
{
    private int rowNum;
    private int colNum;
```

```
private Map<Location, E> occupantMap;

/**
 * Constructs an empty SparseBounded Grid.
 */
public SparseBoundedGrid2(int row, int col)
{
    occupantMap = new HashMap<Location, E>();
    rowNum = row;
    colNum = col;
}

public int getNumRows()
{
    return rowNum;
}

public int getNumCols()
{
    return colNum;
}

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);
    }
}
```

Methods	SparseGridNode version	LinkedList<OccupantInCol> version	HashMap version	TreeMap version
getNeighbors	O(c)	O(c)	O(1)	O(logn)
getEmptyAdjacentLocations	O(c)	O(c)	O(1)	O(logn)
getOccupiedAdjacentLo	O(c)	O(c)	O(1)	O(logn)

cations				
getOccupiedLocations	O(r+n)	O(r+n)	O(n)	O(n)
get	O(c)	O(c)	O(1)	O(logn)
put	O(c)	O(c)	O(1)	O(logn)
remove	O(c)	O(c)	O(1)	O(logn)

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?

The time complexity for the get methos is O(1)

The time complexity for the put methos is O(1) when the row and column index values are within the current array bounds.

The time complexity for the put methos is O(r*c) when the array needs to be resized.

```
import info.gridworld.grid.Grid;
import info.gridworld.grid.AbstractGrid;
import info.gridworld.grid.Location;
import java.util.HashMap;
import java.util.*;

/**
 * UnboundedGrid2
```

```
 */

public class UnboundedGrid2<E> extends AbstractGrid<E>
{
    private int rowNum;
    private int colNum;
    private Object[][] occupantArray;

    /**
     * Constructs an empty unbounded grid.
     */
    public UnboundedGrid2()
    {
        rowNum = colNum = 16;
        occupantArray = new Object[rowNum][colNum];
    }

    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> theLocations = new ArrayList<Location>();

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

        return theLocations;
    }

    public void enlargeGrid(int newrow, int newcol) {
        Object[][] newoccupantArray = new  Object[newrow][newcol];
        for (int i = 0; i < rowNum; i++) {
            for (int j = 0; j < colNum; j++) {
                newoccupantArray[i][j] = occupantArray[i][j];
            }
        }
        rowNum = newrow;
        colNum = newcol;
        occupantArray = newoccupantArray;
    }

    public E get(Location loc)
    {
        if (!isValid(loc))
            throw new IllegalArgumentException("Location " + loc
```

```
                + " is not valid");
        if (loc.getRow() >= rowNum || loc.getCol() >= colNum) {
            return null;
        }
        return (E) occupantArray[loc.getRow()][loc.getCol()]; //
unavoidable warning
    }

    public E put(Location loc, E obj)
    {
        if (loc == null)
            throw new NullPointerException("loc == null");
        if (obj == null)
            throw new NullPointerException("obj == null");

        if (loc.getRow() >= rowNum || loc.getCol() >= colNum) {

            int newRowSize = rowNum;
            int newColSize = colNum;
            while (loc.getRow() >= newRowSize || loc.getCol() >=
newColSize) {
                    newRowSize *= 2;
                    newColSize *= 2;
            }

            enlargeGrid(newRowSize, newColSize);
            System.out.println(colNum);
        }

        // Add the object to the grid.
        E oldOccupant = get(loc);
        occupantArray[loc.getRow()][loc.getCol()] = obj;
        return oldOccupant;
    }
```

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