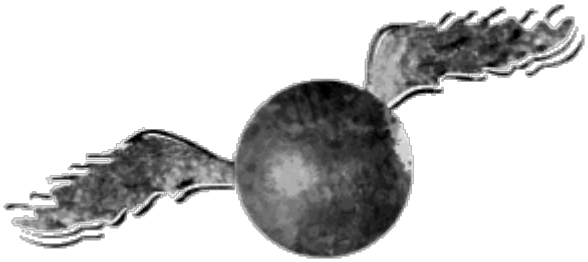
CMSC 206: Data Structures
Harry Potter and the Practice Final Exam

Harry Potter

and the

CS Exam



Many, far and wide, have read about the adventures of Harry, Ron, and Hermione.

What you have read is a fabrication – a simplified version, suitable for lay readers. Herein lies the true tale of how the trio saved life as we know it. Herein lies the true tale of Harry Potter and the Factorer's Stone.

Harry, Ron, and Hermione deftly snuck past Fluffy – the three-headed guard dog – by playing Hagrid's flute to lull him to sleep. They jumped down the trapdoor at his feet.

They landed on Devil's Snare, a plant that curls around the limbs (and sometimes necks) of unsuspecting Hogwarts students.

We represent each tentacle of the Devil's Snare as a linked list composed of `Node`s. (See the *Node.java* file posted with Lab #10 for CS206.) To save our heroes from the evil plant, we must identify the links in the list that have a grip on one of their body parts. Each student is very well aware of what part of the plant is touching them – they each have a `Set<Node>` containing all the `Node`s that are touching. In other words, each has a set of nodes that must be removed.

We shall use these class definitions:

```
public class Student
{
      …
      // a set of all nodes that must be removed to survive
      private Set<Node> touchingNodes =
                           new HashSet<ListNode>();

      public boolean isTouching(Node n)
      { /* to be implemented below */ }
      …
}

public class DevilsSnare
{
      …
      // the heads of the tentacle lists
      private Node[] heads;

      public Node[] getHeads() { return heads; }
      …
}
```

We wish to write methods, not placed in either of these classes, that free a student from the Devil's Snare. We will do this by iterating through the lists, looking for nodes that ought be removed, and removing them.

1. **(4 pts)** Neville Longbottom (another Hogwarts student), in reviewing the trio's actions, suggests that we simply iterate through a student's set of nodes, removing each one from its respective list. Which of the following statements is *false*?

   A. This approach, if implemented, would be slower than the approach above.

   B. It is impossible to write a method `remove(Node node)` without knowing the head of the list.

   C. The running time of Neville's approach, if properly implemented, would be $O(TSL)$, where T is the number of touching nodes, S is the number of different tentacles (the size of the `heads` array), and L is the maximum length of a tentacle (the maximum number of nodes in a list representing a tentacle).

   D. It is slow to iterate through a `HashSet` – that is why this approach is not used.

   E. Neville's approach would require significant searching, causing it to be slow.

2. **(4 pts)** We will first write a method that determines if a given `Node` is touching a `Student`. Write the `isTouching` method in the `Student` class:

```
/* preconditions:  the touchingNodes field is properly
 *                 set up to hold all the Nodes that
 *                 are touching us.
 * postconditions: the touchingNodes field is unchanged;
 *                 returns true if the node n is
 *                 touching, and false otherwise
 */
public boolean isTouching(Node n)
{









}
```

3. **(4 pts)** What is the running time of the `isTouching` method? Let T be the number of nodes stored in `touchingNodes` and let L be the length of the list referred to by `n`.

   A. $O(1)$

   B. $O(T)$

   C. $O(TL)$

   D. $O(L)$

   E. $O(T^2)$

The remaining methods to write are in neither the `Student` nor the `DevilsSnare` classes. The actual location of the methods is irrelevant.

4. **(16 pts)** Write a method that takes one tentacle and removes all the nodes touching a given student from that tentacle:

```
/* preconditions:  head refers to the head of a list of
 *                 Node objects; stu refers to a
 *                 proper Student object
 * postconditions: all nodes that are touching stu are
 *                 removed; the new head of the list is
 *                 returned
 */
public Node removeTouching(Node head, Student stu)
{









}
```

5. **(4 pts)** Write a method that takes a `DevilsSnare` and frees a student:

```
/* preconditions:  all parameters are properly set up
 * postconditions: all nodes from all lists that are
 *                 touching stu are removed.
 */
public void freeStudent(DevilsSnare snare, Student stu)
{









}
```

6. **(4 pts)** There are potentially many students involved. Write a method that will free all of them:

```
/* preconditions:  all parameters are set up.
 * postconditions: all nodes touching any student are
 *                 removed.
 */
public void freeAll(DevilsSnare snare, List<Student> studs)
{









}
```

7. **(4 pts)** What is the total running time of this method? Let T be the maximum number of nodes stored in one student's `touchingNodes`; let L be the maximum length of a list representing a tentacle; let S be the number of different tentacles (the size of the `heads` array); and let H be the number of students involved.

    A. $O(TLSH)$

    B. $O(LSH)$

    C. $O(TLH)$

    D. $O(TSH)$

    E. $O(TLS)$

After successfully escaping from the Devil's Snare, Harry, Ron, and Hermione continue into the next chamber. There, they see a heavy wooden door. Looking up, they see many glittering birds. After Ron discovers the door is locked, Harry suddenly realizes that the flying creatures aren't birds – they're keys. The challenge in this chamber is to find the key that fits the door.

Harry, Ron, and Hermione all take to brooms to fly about the chamber, searching for the key. Harry suggests binary search, saying, "Ron, you come at it from above – Hermione, stay below … – and I'll try and catch it."[1]

[1] *Harry Potter and the Sorcerer's Stone*, by J.K. Rowling, page 280.

8. **(4 pts)** We have a collection of objects of a class `Key` that we are searching among. What are the requirements in order to use binary search?

    I. The `Key`s must be stored in some kind of `List` or array.

    II. The class `Key` must implement the `Comparable` interface.

    III. The `Key`s must be in order.

    A. I only

    B. III only

    C. I and II only

    D. II and III only

    E. I, II, and III

9. **(4 pts)** In planning to use binary search for the key, Harry remembers the following method for binary search from his Algorithms class, taught by Professor Sedgewick.[2]

[2] Robert Sedgewick, though he has a name vaguely reminiscent of Professor Flitwick at Hogwarts, is a real Computer Science professor who wrote a widely-distributed book on algorithms.

```
      /* preconditions:  all necessary requirements for
       *                 binary search are met
       * postconditions: the array is unchanged; the index
       *                 of value is returned; -1 is
       *                 returned if the value is not in the
       *                 array
       */
1     public int binarySearch(int[] arr, int value)
2     {
3           int end = arr.length - 1;
4           int start = 0;
5           int mid = (end + start) / 2;
6
7           while(arr[mid] != value && end > start)
8           {
9                 if(value > arr[mid])
10                {
11                      start = mid + 1;
12                }
13                else
14                {
15                      end = mid;
16                }
17
18                mid = (end + start) / 2;
19          }
20
21          if(arr[mid] == value)
22          {
23                return mid;
24          }
25          else
26          {
27                return -1;
28          }
29    }
```

What statement is *true* about the preceding method?

A. It works as specified.

B. The method does not work properly; line 11 should read `start = mid;`

C. The method does not work properly; line 13 should read `else if(value < arr[mid])`

D. The method does not work properly; it will not find the element if `value` is the first element in the list.

E. The method does not work properly; it will not find the element if `value` is the last element in the list.

As it turns out, binary search will not work for finding the key to the locked wooden door – something about keys does not meet those requirements you discovered above. Instead, we have a `Set<Key>` that stores all the keys. Here are relevant class definitions:

```
public class Door
{
     public void unlock(RightKey k)
     {
          /* unlocks door so that our heroes may pass */
     }
}

public abstract class Key
{
     public abstract void unlockDoor(Door d);
}

public class WrongKey extends Key
{
     public void unlockDoor(Door d)
     {
          d.unlock(this);
     }
}

public class RightKey extends Key
{
     public class unlockDoor(Door d)
     {
          d.unlock(this);
     }
}
```

10. **(4 pts)** What statement is *true* about the preceding code? *(out of scope for CS206 exam, but still something we covered this semester)*

    A. All of the code compiles.

    B. The keyword `this` is used improperly in the `unlockDoor` methods.

    C. The code will not compile because there is an abstract method in a concrete class.

    D. The code will not compile because there are no constructors written.

    E. The code will not compile because of a type mismatch error.

11. **(6 pts)** Write a method that will find the right key among a `Set` of `Key`s and return it. Code that contains unnecessary casts will not receive full credit. *(out of scope for CS206 exam, but still something we covered this semester)*

```
public RightKey findKey(Set<Key> keys)
{









}
```

12. **(6 pts)** Write a method that will use `findKey` to find the key and unlock the door. Code that contains unnecessary casts will not receive full credit. *(out of scope for CS206 exam, but still something we covered this semester)*

```
public void useKey(Set<Key> keys, Door d)
{









}
```

With the help of those methods, our heroes proceed to the next room.

Before them appears a life-size chess board. As the three quickly figure out, they must take the place of three of the black chess pieces and win a chess game to cross the room. To do that, we need to help them figure out how some of the chess pieces move.

The chessboard is represented by an 8x8 matrix of `ChessPiece` elements. Here are some relevant classes:

```
public class Location
{
     private int rw; // both rw and cl count from 0
     private int cl; // thus, the range is 0 .. 7

     public int row() { return rw; }
     public int col() { return cl; }

     public Location(int r, int c) { rw = r; cl = c; }
}

public abstract class ChessPiece
{
     /* other methods, etc. shown below */

     public boolean isWhite()
     { /* implementation not shown */ }
}

public class Chess
{
     /* the matrix of chess pieces.  Empty squares are
        denoted by null */
     ChessPiece[][] board = new ChessPiece[8][8];

     /* checks whether a location is empty
      * preconditions:  The board is properly set up; loc
      *                 is a valid location in the chess
      *                 board.
      * postconditions: the board is unchanged; true is
      *                 returned if loc is empty, or false
      *                 otherwise
      */
     private boolean isEmpty(Location loc)
     { /* to be implemented below */ }
```

```
    /* checks whether a location contains a friendly piece
     * preconditions:  The board is properly set up;
     *                 isWhite indicates whether the
     *                 current player is white or black;
     *                 loc is a valid location in the
     *                 chess board.
     * postconditions: the board is unchanged; true is
     *                 returned if loc contains a friendly
     *                 piece, or false otherwise
     */
    private boolean containsFriendly(Location loc,
                                     boolean isWhite)
    { /* to be implemented below */ }

    /* checks whether a location contains an enemy piece
     * preconditions:  The board is properly set up;
     *                 isWhite indicates whether the
     *                 current player is white or black;
     *                 loc is a valid location in the
     *                 chess board.
     * postconditions: the board is unchanged; true is
     *                 returned if loc contains an enemy
     *                 piece, or false otherwise
     */
    private boolean containsEnemy(Location loc,
                                  boolean isWhite)
    { /* to be implemented below */ }

    /* Compute the Locations a given piece can move to.
     * preconditions:  The board is properly set up; the
     *                 piece under consideration can move
     *                 in the direction indicated by
     *                 rowStep and colStep
     * postconditions: The chess board is unchanged; a
     *                 List of Locations is returned that
     *                 contains all possible squares
     *                 accessible from the starting
     *                 location in the direction given by
     *                 rowStep and colStep.
     */
    public List<Location>
         getPossibleDestinations(Location startingLoc,
              int rowStep, int colStep, boolean isWhite)
    { /* to be implemented below */ }
}
```

13. **(4 pts)** Write the method `isEmpty`. *Hint*: Yes, this is as simple as it seems.

```
/* checks whether a location is empty
 * preconditions:  The board is properly set up; loc
 *                 is a valid location in the chess
 *                 board.
 * postconditions: the board is unchanged; true is
 *                 returned if loc is empty, or false
 *                 otherwise
 */
private boolean isEmpty(Location loc)
{









}
```

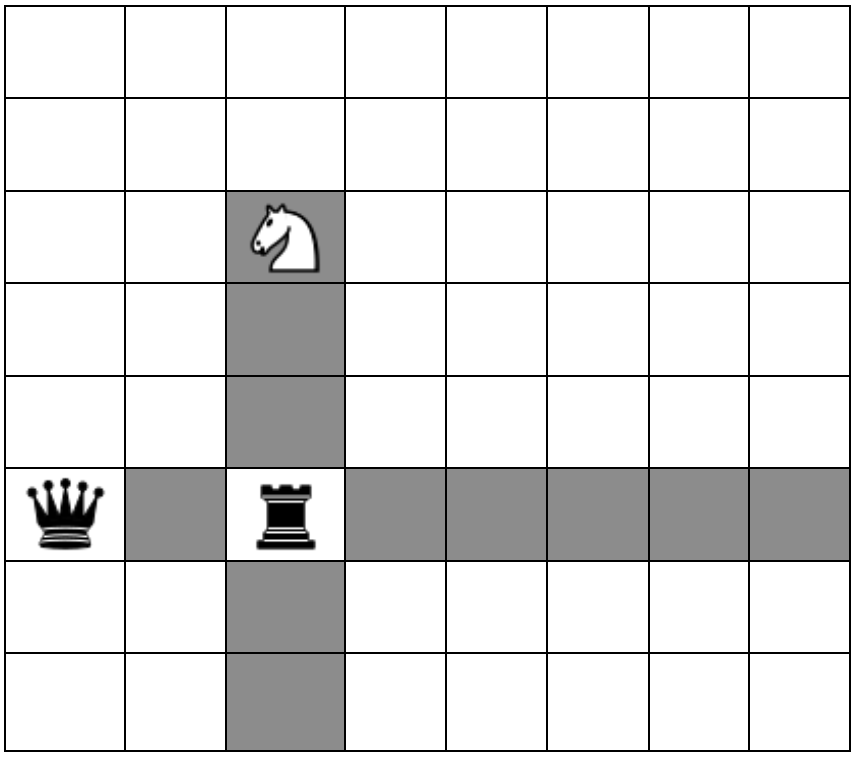
14. **(4 pts)** Write the method `containsFriendly`. This method checks a given location to see if it contains a friendly piece. In other words, this returns `true` if the location given contains a piece and that piece's color agrees with the `isWhite` parameter to this method.

```
/* checks whether a location contains a friendly piece
 * preconditions:  The board is properly set up;
 *                 isWhite indicates whether the
 *                 current player is white or black;
 *                 loc is a valid location in the
 *                 chess board.
 * postconditions: the board is unchanged; true is
 *                 returned if loc contains a friendly
 *                 piece, or false otherwise
 */
private boolean containsFriendly(Location loc,
                                 boolean isWhite)
{









}
```

15. **(4 pts)** Write the method `containsEnemy`. This method is just like `containsFriendly`, but it returns true for a location containing a piece whose color does not match what is in the `isWhite` parameter.

```
/* checks whether a location contains an enemy piece
 * preconditions:  The board is properly set up;
 *                 isWhite indicates whether the
 *                 current player is white or black;
 *                 loc is a valid location in the
 *                 chess board.
 * postconditions: the board is unchanged; true is
 *                 returned if loc contains an enemy
 *                 piece, or false otherwise
 */
private boolean containsEnemy(Location loc,
                              boolean isWhite)
{









}
```

16. **(20 pts)** Write the `getPossibleDestinations` method. The idea behind this method is that it returns possible destinations for a chess piece. Rooks, Bishops, and Queens move similarly in chess, and this method helps simplify their implementation. Any of these pieces move along a line. Depending on the piece, that line can be horizontal, vertical, or along either diagonal. The line starts at the piece's current location and extends until the end of the board is reached or a non-empty location is reached. If the non-empty location holds a friendly piece, the line ends one square before it; if the non-empty location holds an enemy piece, the line ends on that square (and the enemy piece is liable to be captured). Here is an illustration, using a Rook, which can move along any vertical or horizontal line:



Here, the Rook (denoted by the ♜), can move to any of the shaded squares. It is blocked by the edge of the board on the right and bottom, blocked by a friendly piece on the left, and blocked by an enemy piece above. Note that the Rook can travel to the location the enemy piece occupies, but no further. The Rook cannot travel to the location the friendly piece occupies.

The `getPossibleDestinations` method works only along one line. Thus, for a Rook, one would have to call `getPossibleDestinations` four times to collect all possible destinations. The `rowStep` and `colStep` parameters control which line is being investigated. For the Rook, we would call `getPossibleDestinations` once with a `rowStep` of 1 and a `colStep` of 0; this would get the destinations below the Rook. We would call the method again with 0 and 1 to get the destinations to the right of the Rook. The next call would be with -1 and 0 to get the destinations above, and the last would be 0 and -1 to get the destinations on the left. The order of these calls does not matter.

To seek destinations along a diagonal, both the `rowStep` and `colStep` parameters would be non-zero.

Write the method here. It is worth **4 extra credit points** if written recursively. A helper method is unnecessary.

```
/* Compute the Locations a given piece can move to.
 * preconditions:  The board is properly set up; the
 *                 piece under consideration can move
 *                 in the direction indicated by
 *                 rowStep and colStep
 * postconditions: The chess board is unchanged; a
 *                 List of Locations is returned that
 *                 contains all possible squares
 *                 accessible from the starting
 *                 location in the direction given by
 *                 rowStep and colStep.
 */
private List<Location>
     getPossibleDestinations(Location startingLoc,
          int rowStep, int colStep, boolean isWhite)
{









}
```

Now that we have a functioning chess board, we must use it to move some of the pieces.

Here are the relevant classes:

```
public abstract class ChessPiece // repeated from above
{
     private boolean white; // true if white; false if black
     private Location loc; // current location of piece

     /* returns true if white; false if black */
     public boolean isWhite()
     {
          return white;
     }

     /* returns the current location */
     public Location location()
     {
          return loc;
     }

     /* preconditions:  The board is properly set up
      * postconditions: The board is unchanged; a list of
      *                 all possible destinations for this
      *                 chess piece is returned
      */
     public abstract List<Location> getAllDestinations(Chess board);

     /* other methods not shown */
}

public class Rook extends ChessPiece
{
     /* a Rook can move vertically or horizontally */

     public List<Location> getAllDestinations(Chess board)
     { /* to be implemented below */ }

     /* other methods not shown */
}
```

```
public class Bishop extends ChessPiece
{
     /* a Bishop can move only along either diagonal */

     public List<Location> getAllDestinations(Chess board)
     { /* to be implemented below */ }

     /* other methods not shown */
}
```

17. **(6 pts)** Write the `Rook` method `getAllDestinations`. A Rook can move either way along both a horizontal and vertical line.

```
// IN THE Rook CLASS:
/* preconditions:  The board is properly set up
 * postconditions: The board is unchanged; a list of
 *                 all possible destinations for this
 *                 chess piece is returned
 */
public List<Location> getAllDestinations(Chess board)
{









}
```

18. **(6 pts)** Write the `Bishop` method `getAllDestinations`. A Bishop can either way along either diagonal.

```
// IN THE Bishop CLASS:
/* preconditions:  The board is properly set up
 * postconditions: The board is unchanged; a list of
 *                 all possible destinations for this
 *                 chess piece is returned
 */
public List<Location> getAllDestinations(Chess board)
{









}
```

19. **(4 pts)** In a method tying all of this together, we use this code:

```
ChessPiece piece = /* code unimportant */;
Chess board = /* code unimportant */;

List<Location> dests =
               piece.getAllDestinations(board);
```

What statement is *true* of the preceding code? *(out of scope for CS206 exam, but still something we covered this semester)*

A. This is an example of inheritance.

B. This is an example of polymorphism.

C. This code contains an implicit upcast.

D. This code will not work; it calls an abstract method.

E. This code will not work; it causes a `ClassCastException`.

Using the above methods, our Harry, Ron, and Hermione take the place of a Rook, a Knight, and a Bishop, and they battle it out to cross the room. In the course of the game, Ron tragically sacrifices himself to win. Victorious but shaken, Harry and Hermione pass through the darkened door into the next chamber.

There, in the dim light, they start to make out a strange but familiar sight. A voice rings out.

TO BE CONTINUED…

Image credits:

Harry Potter logo:
http://toysinaction.nl/oscommerce/catalog/images/Harry_Potter-logo_90894o.jpg

Harry Potter font:
http://www.mugglenet.com/downloads/fonts/files/lumos.zip

Golden snitch:
http://www.encyclopedie-hp.org/images/sva/snitch2-sva.gif

Hat:
http://www.badgersden.com/General_Store/Garb/Hats/HP/LU2230.jpg