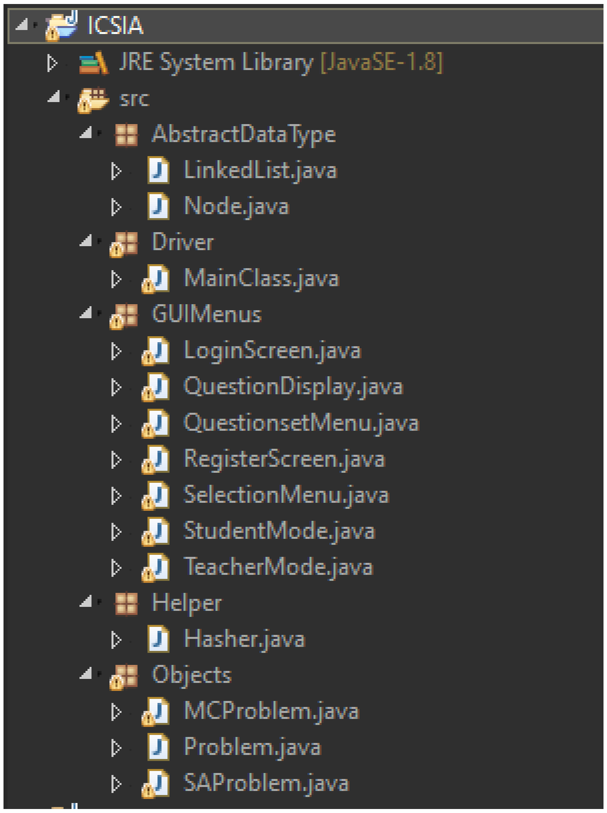
Criterion C: Development

This program is written using java. The purpose of the program is to serve as a user friendly question bank for math contest questions and interesting math problems in general. The user can store any math problems in the program and access them at any time.

Figure 1: Program Classes



```
ICSIA
  JRE System Library [JavaSE-1.8]
  src
    AbstractDataType
      LinkedList.java
      Node.java
    Driver
      MainClass.java
    GUIMenus
      LoginScreen.java
      QuestionDisplay.java
      QuestionsetMenu.java
      RegisterScreen.java
      SelectionMenu.java
      StudentMode.java
      TeacherMode.java
    Helper
      Hasher.java
    Objects
      MCProblem.java
      Problem.java
      SAProblem.java
```

Registration and Login Process:

Figure 2: Login Screen

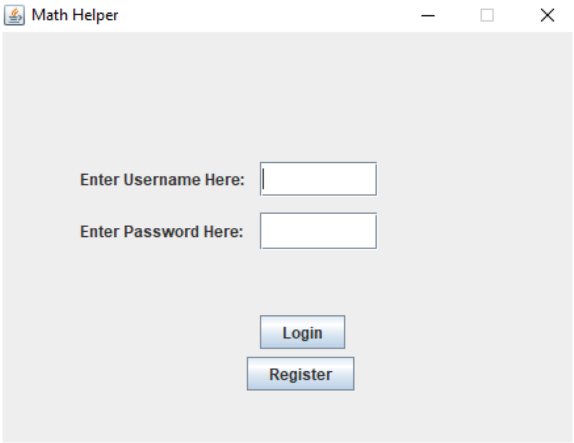
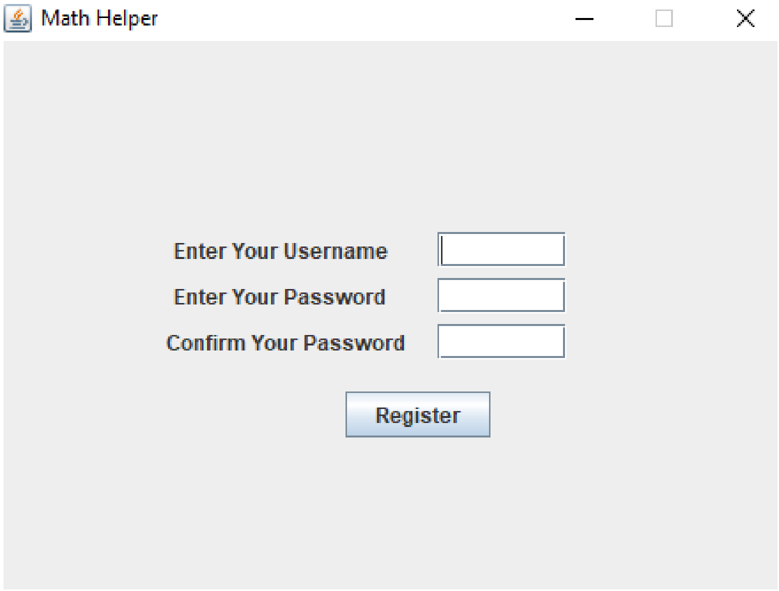
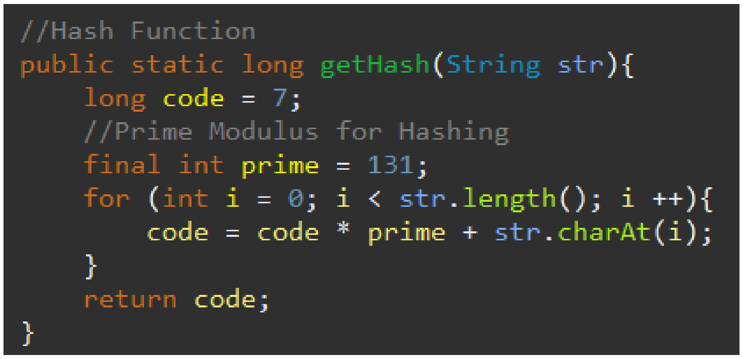


Figure 3: Registration and Account Creation Screen



An account is needed to use the program. The user will first have to create an account that fulfills a couple of requirements. The username of the account must also not be a duplicate. Assuming that the user info is valid, the user info will then be hashed and encrypted to protect the user. The created account will then be stored in a text file called “userinfo.txt”. A hash is used because it protects the user and allows for comparisons between strings to be faster once the hash is preprocessed as in the case of the text file.

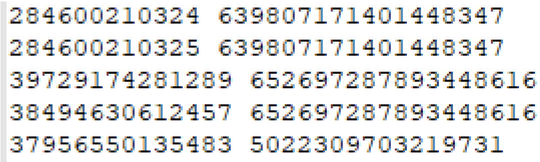
Figure 4: Hash Function



```java
//Hash Function
public static long getHash(String str){
    long code = 7;
    //Prime Modulus for Hashing
    final int prime = 131;
    for (int i = 0; i < str.length(); i ++){
        code = code * prime + str.charAt(i);
    }
    return code;
}
```

The resulting information in the text file will look the same as Figure 5 shown below.

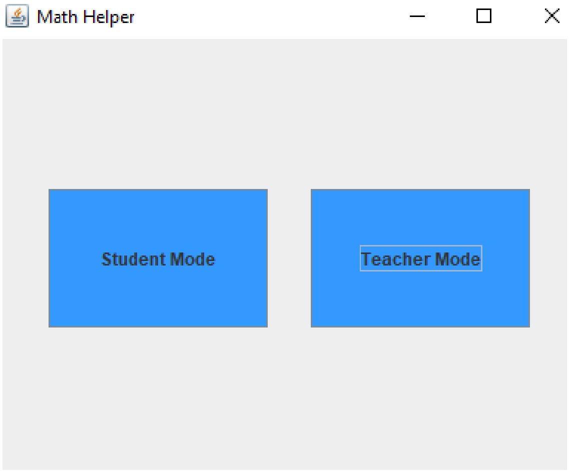
Figure 5: Hashed User Info



```
284600210324 6398071714014448347
284600210325 6398071714014448347
39729174281289 6526972878934448616
38494630612457 6526972878934448616
37956550135483 5022309703219731
```

After logging in, the user will choose between one of two modes as shown in Figure 6.

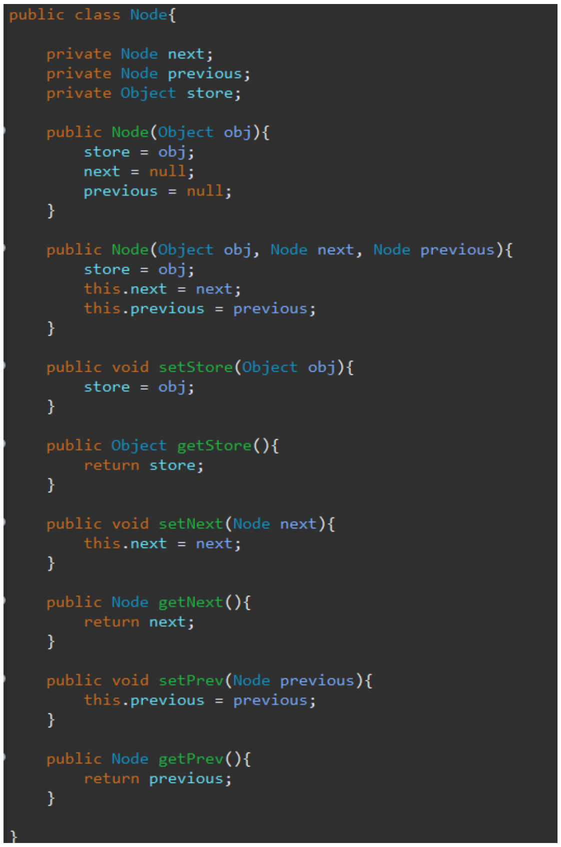
Figure 6: SelectionMenu



LinkedList and Node Class:

To store all of the objects and a lot of data, throughout the program I used the abstract data type of a doubly linked list alongside a node class. A doubly linked list allows for the support of adding on additional nodes instead and fixed in size when compared to an array. Furthermore, as opposed to a singly linked list, a doubly linked list allows for deletion of elements in O(1) or constant time.

Figure 7: Node Class

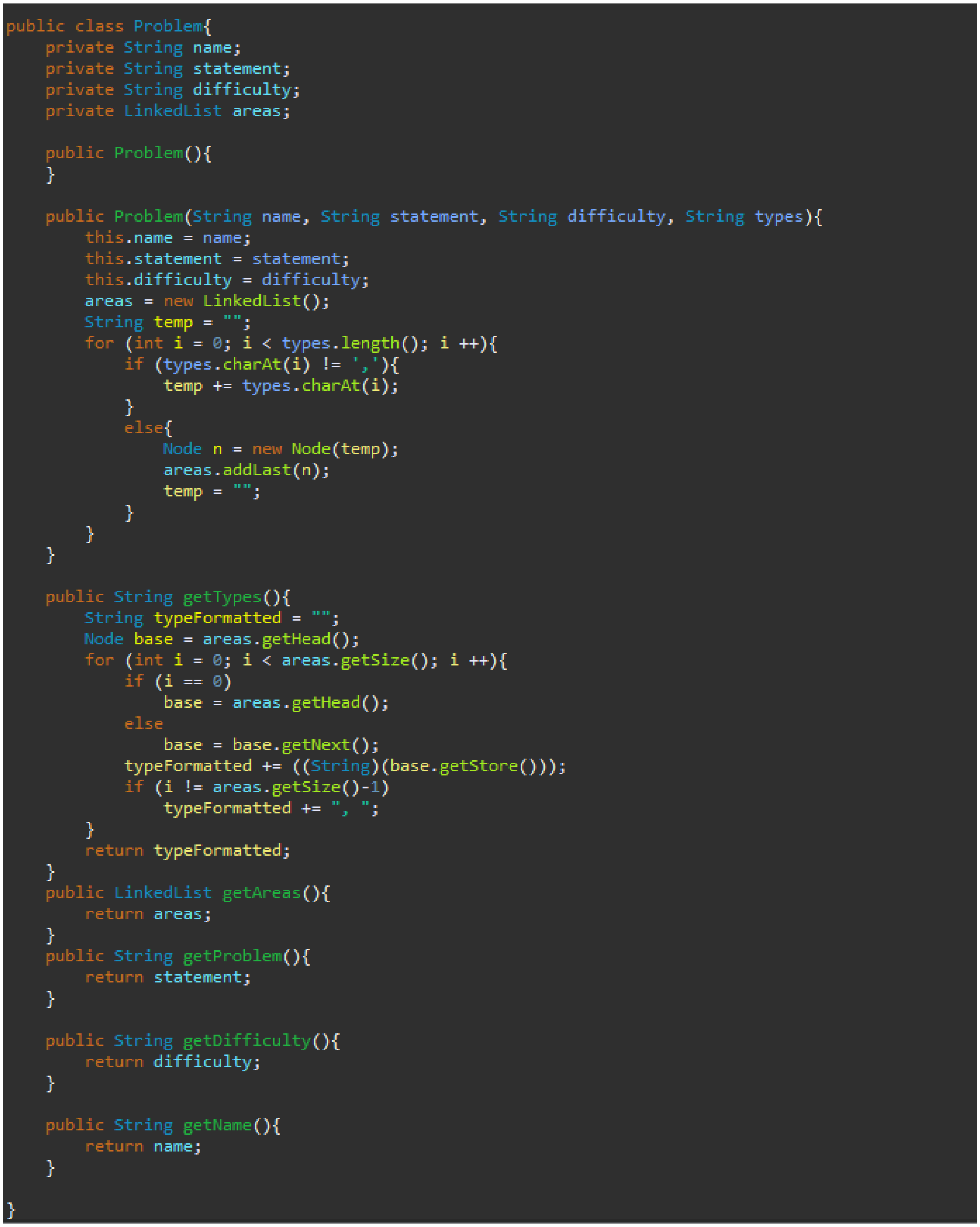


```
public class Node{

    private Node next;
    private Node previous;
    private Object store;

    public Node(Object obj){
        store = obj;
        next = null;
        previous = null;
    }

    public Node(Object obj, Node next, Node previous){
        store = obj;
        this.next = next;
        this.previous = previous;
    }

    public void setStore(Object obj){
        store = obj;
    }

    public Object getStore(){
        return store;
    }

    public void setNext(Node next){
        this.next = next;
    }

    public Node getNext(){
        return next;
    }

    public void setPrev(Node previous){
        this.previous = previous;
    }

    public Node getPrev(){
        return previous;
    }

}
```

Inheritance between Objects:

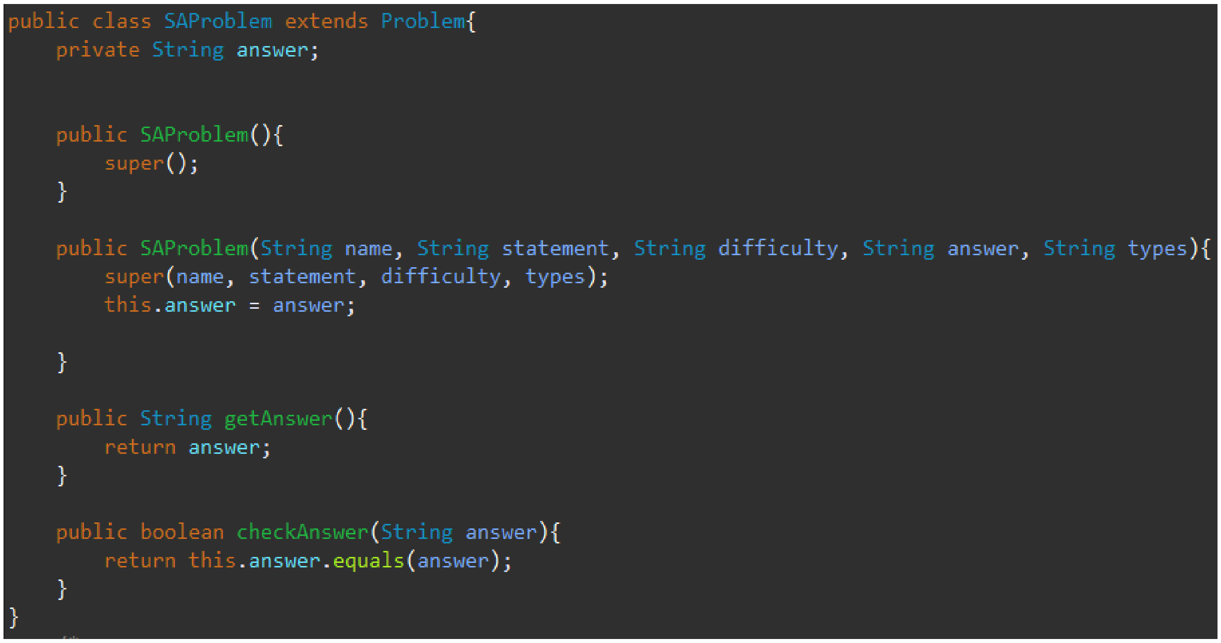
The question bank supports two different types of questions/problems. Both multiple choice and short answer problems are supported. To support this, inheritance is used to ensure that code can be reused and not repeated.

Figure 8: Problem class



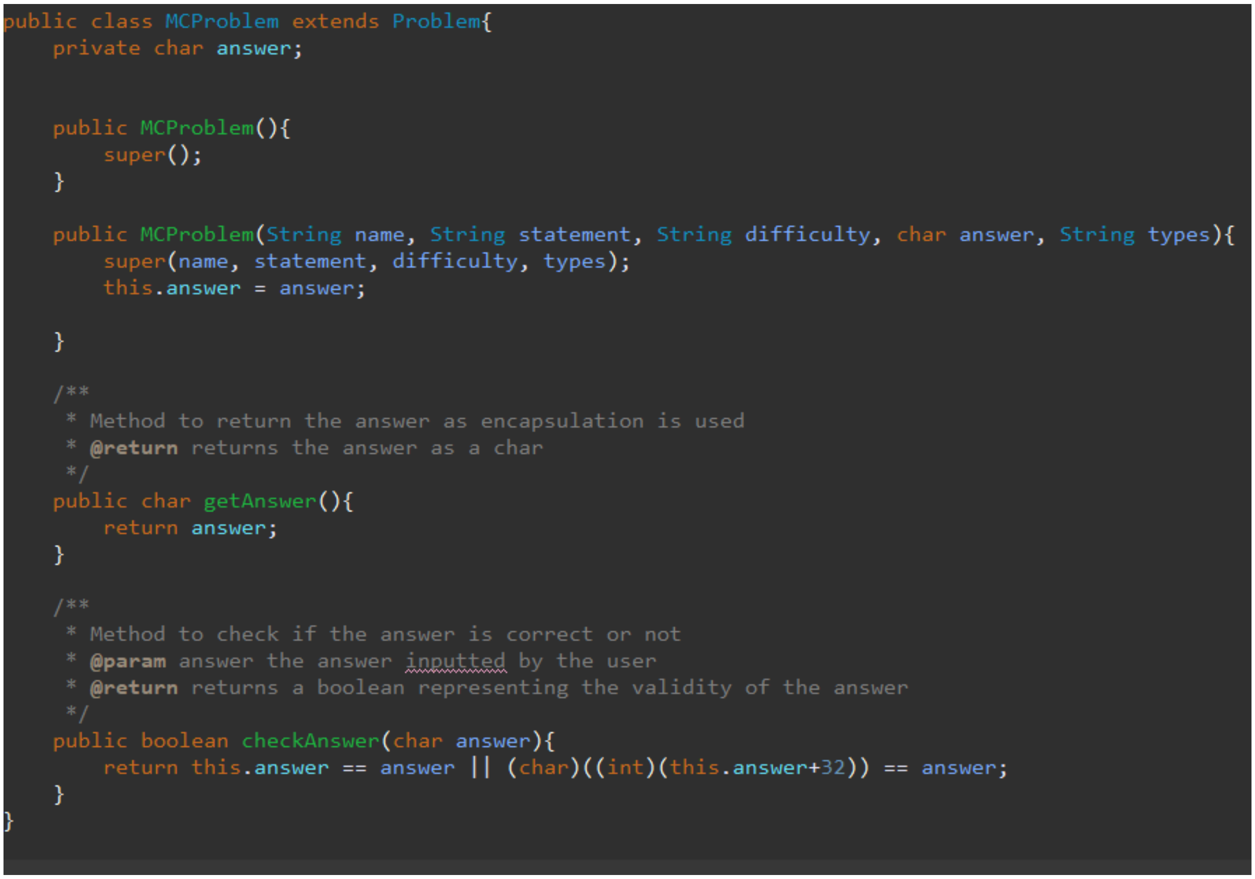
```
public class Problem{
    private String name;
    private String statement;
    private String difficulty;
    private LinkedList areas;

    public Problem(){
    }

    public Problem(String name, String statement, String difficulty, String types){
        this.name = name;
        this.statement = statement;
        this.difficulty = difficulty;
        areas = new LinkedList();
        String temp = "";
        for (int i = 0; i < types.length(); i ++){
            if (types.charAt(i) != ','){
                temp += types.charAt(i);
            }
            else{
                Node n = new Node(temp);
                areas.addLast(n);
                temp = "";
            }
        }
    }

    public String getTypes(){
        String typeFormatted = "";
        Node base = areas.getHead();
        for (int i = 0; i < areas.getSize(); i ++){
            if (i == 0)
                base = areas.getHead();
            else
                base = base.getNext();
            typeFormatted += ((String)(base.getStore()));
            if (i != areas.getSize()-1)
                typeFormatted += ", ";
        }
        return typeFormatted;
    }
    public LinkedList getAreas(){
        return areas;
    }
    public String getProblem(){
        return statement;
    }

    public String getDifficulty(){
        return difficulty;
    }

    public String getName(){
        return name;
    }

}
```

Figure 9: SAProblem Class



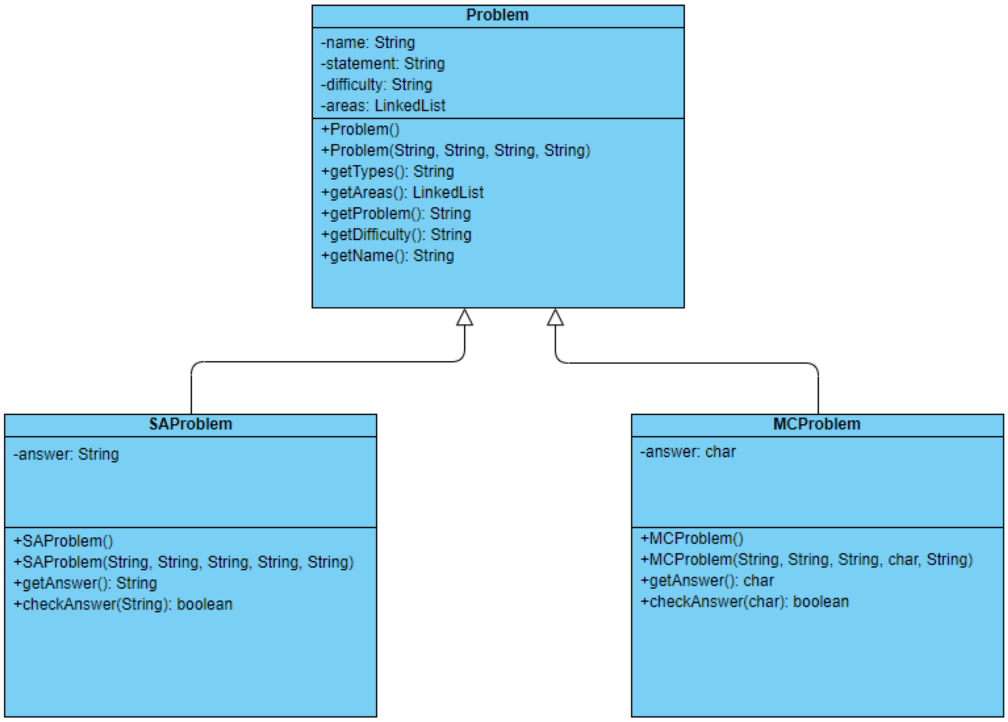
```java
public class SAProblem extends Problem{
    private String answer;


    public SAProblem(){
        super();
    }

    public SAProblem(String name, String statement, String difficulty, String answer, String types){
        super(name, statement, difficulty, types);
        this.answer = answer;

    }

    public String getAnswer(){
        return answer;
    }

    public boolean checkAnswer(String answer){
        return this.answer.equals(answer);
    }
}
```

Figure 10: MCProblem Class



```java
public class MCProblem extends Problem{
    private char answer;


    public MCProblem(){
        super();
    }

    public MCProblem(String name, String statement, String difficulty, char answer, String types){
        super(name, statement, difficulty, types);
        this.answer = answer;

    }

    /**
     * Method to return the answer as encapsulation is used
     * @return returns the answer as a char
     */
    public char getAnswer(){
        return answer;
    }

    /**
     * Method to check if the answer is correct or not
     * @param answer the answer inputted by the user
     * @return returns a boolean representing the validity of the answer
     */
    public boolean checkAnswer(char answer){
        return this.answer == answer || (char)((int)(this.answer+32)) == answer;
    }
}
```

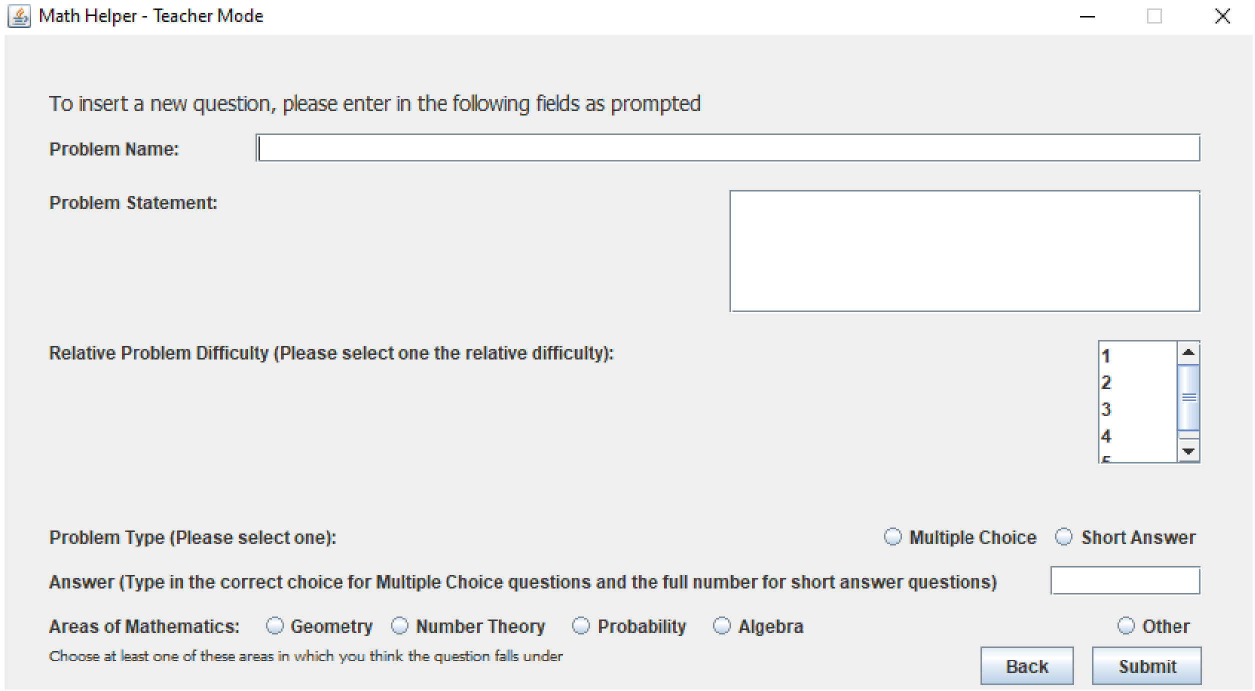
Figure 11: UML Diagram showing the hierarchy of the objects



The Problem class is the parent and the two child classes are the SAProblem and the MCProblem classes. The UML Diagram of the objects are shown as in Figure 12.

Teacher Mode:

Figure 12: Teacher Mode



Math Helper - Teacher Mode

To insert a new question, please enter in the following fields as prompted

Problem Name:

Problem Statement:

Relative Problem Difficulty (Please select one the relative difficulty): 1 2 3 4

Problem Type (Please select one): ○ Multiple Choice ○ Short Answer

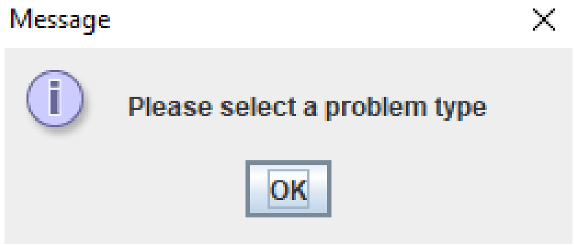
Answer (Type in the correct choice for Multiple Choice questions and the full number for short answer questions)

Areas of Mathematics: ○ Geometry ○ Number Theory ○ Probability ○ Algebra ○ Other

Choose at least one of these areas in which you think the question falls under

Teacher mode of the program allows for the addition of problems. Error messages like the following will be displayed for the future if they do not select or fill in some of the requested areas.

Figure 13: Error message

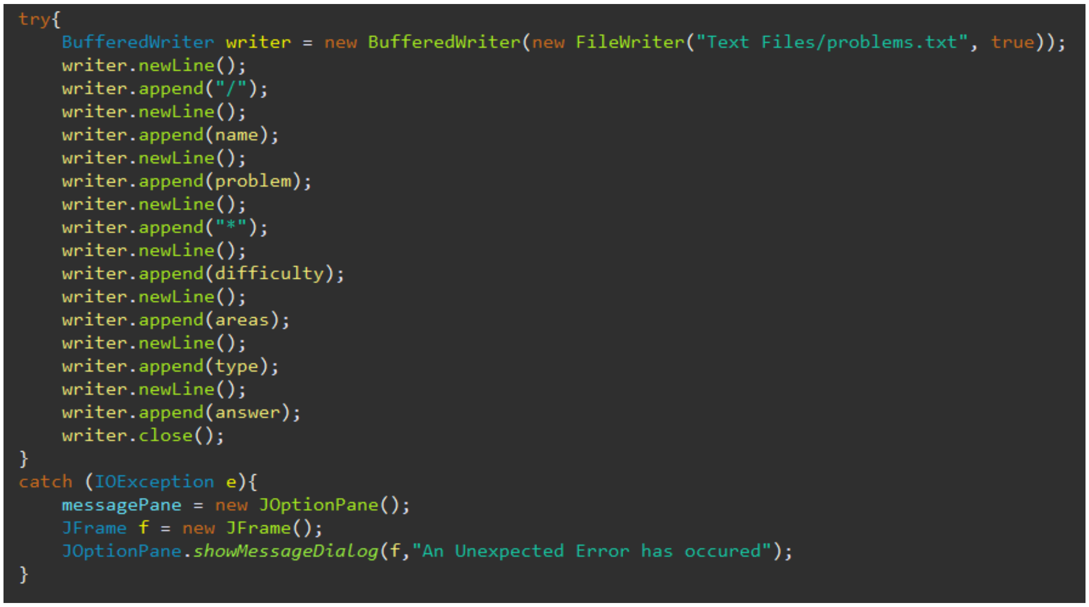


Message

Please select a problem type

By detecting instances that can cause errors, the program is also made more robust which means that it is less likely to crash. If the user input does fulfill all of the requests, the problem will be stored in a text file.

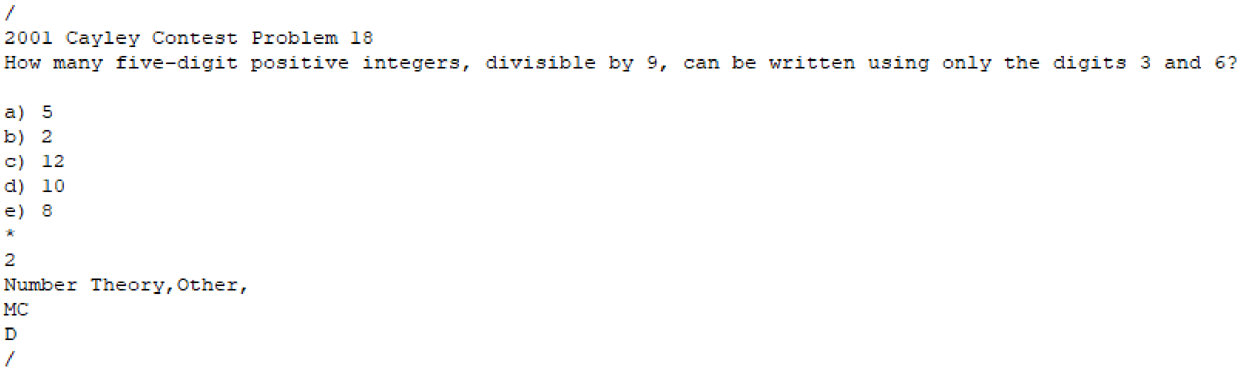
Figure 14: Writing of data into text file



```
try{
    BufferedWriter writer = new BufferedWriter(new FileWriter("Text Files/problems.txt", true));
    writer.newLine();
    writer.append("/");
    writer.newLine();
    writer.append(name);
    writer.newLine();
    writer.append(problem);
    writer.newLine();
    writer.append("*");
    writer.newLine();
    writer.append(difficulty);
    writer.newLine();
    writer.append(areas);
    writer.newLine();
    writer.append(type);
    writer.newLine();
    writer.append(answer);
    writer.close();
}
catch (IOException e){
    messagePane = new JOptionPane();
    JFrame f = new JFrame();
    JOptionPane.showMessageDialog(f,"An Unexpected Error has occured");
}
```

They are stored as such when in the text file. The random symbols such as the / and the * used inbetween are used to keep track of the different sections when reading the text file. There is also the use of try, catch to handle the exceptions in case they arise, making the program more robust.

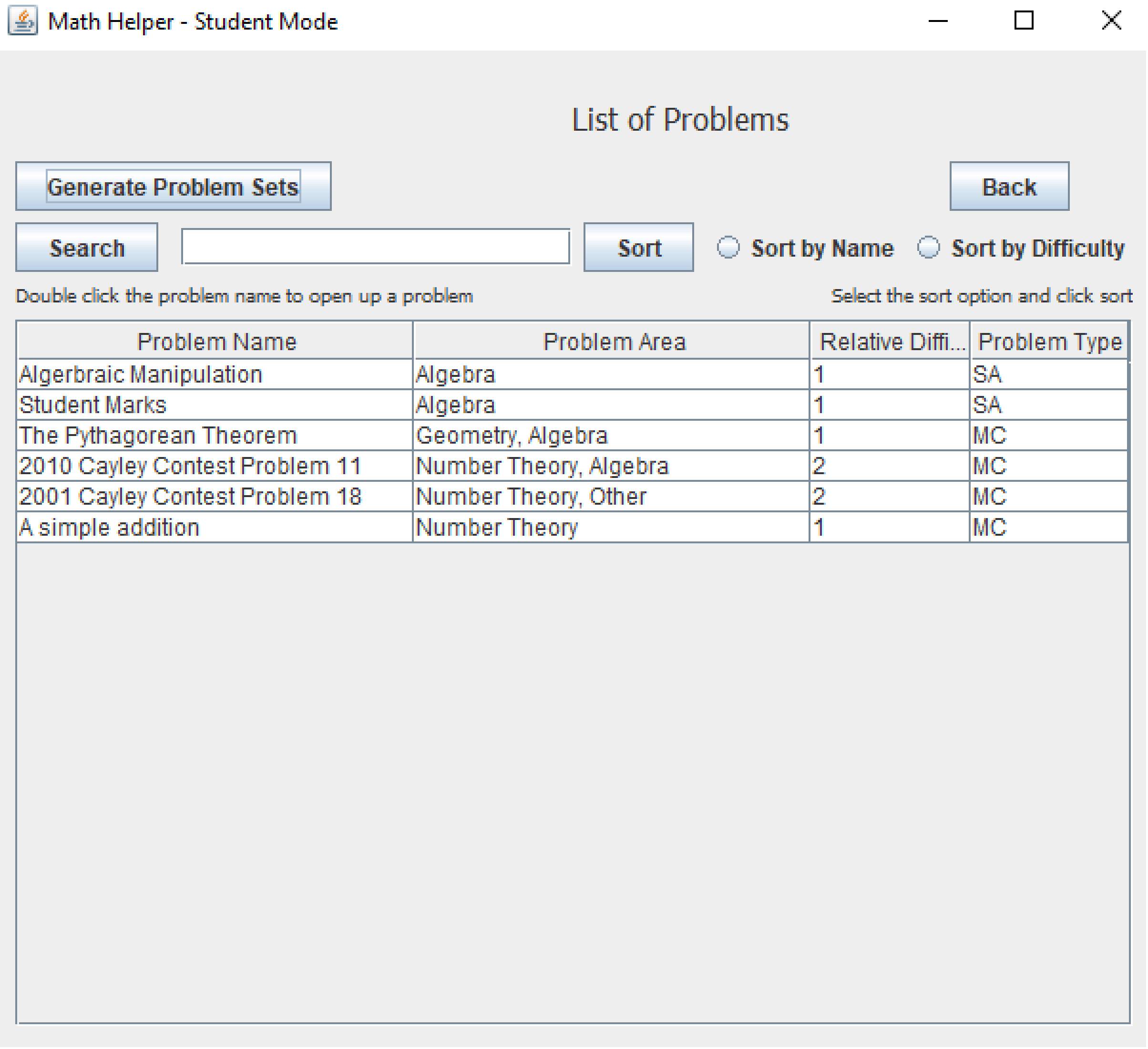
Figure 15: The stored data



```
/
2001 Cayley Contest Problem 18
How many five-digit positive integers, divisible by 9, can be written using only the digits 3 and 6?

a) 5
b) 2
c) 12
d) 10
e) 8
*
2
Number Theory,Other,
MC
D
/
```

Student Mode:

Figure 16: Student Mode main screen



Double click the problem name to open up a problem

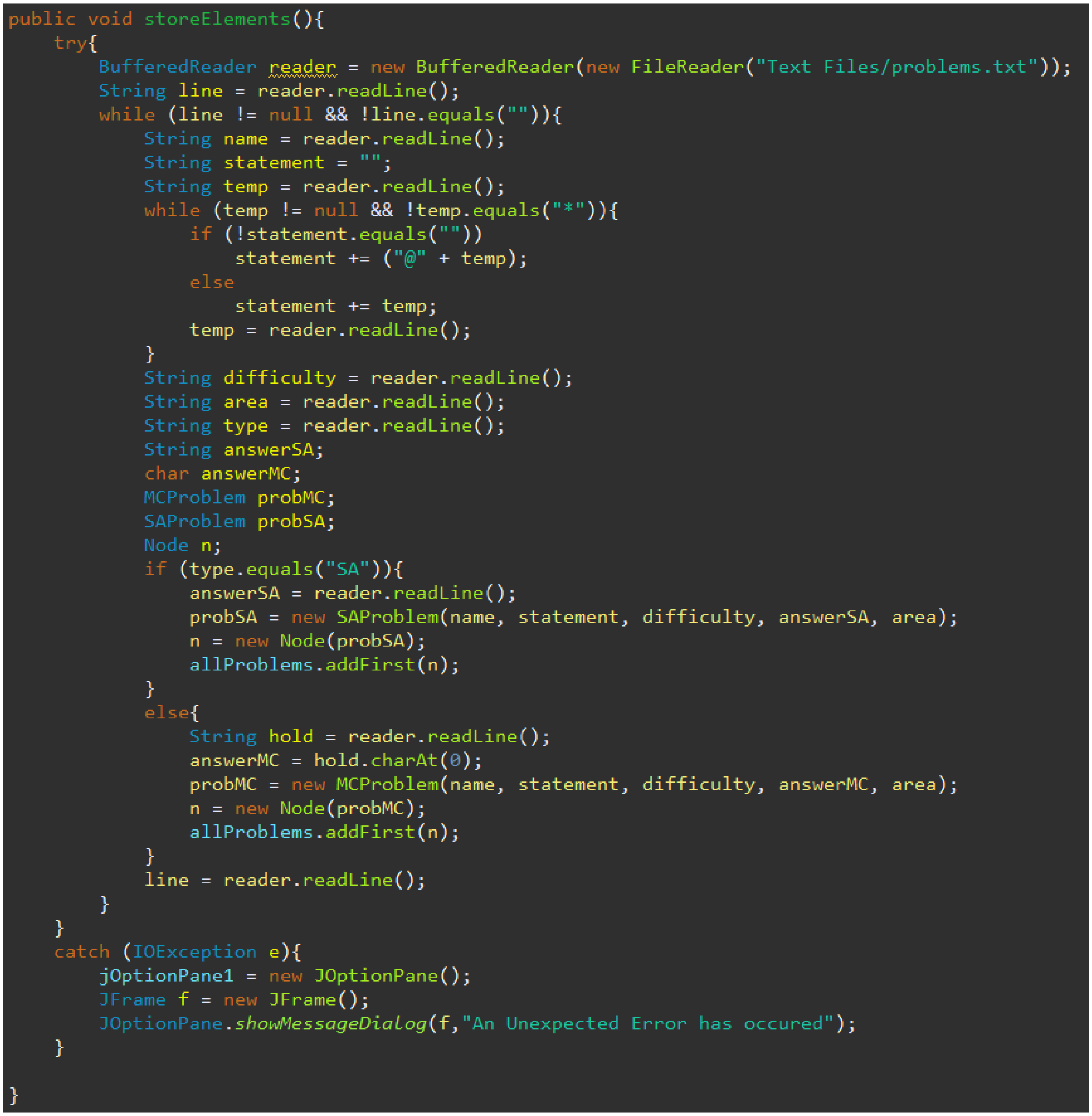
Select the sort option and click sort

Problem Name	Problem Area	Relative Diffi...	Problem Type
Algerbraic Manipulation	Algebra	1	SA
Student Marks	Algebra	1	SA
The Pythagorean Theorem	Geometry, Algebra	1	MC
2010 Cayley Contest Problem 11	Number Theory, Algebra	2	MC
2001 Cayley Contest Problem 18	Number Theory, Other	2	MC
A simple addition	Number Theory	1	MC

In the development process, the advisor recommended that the original design was not feasible. As such, I switched to a table design to showcase all of the problems to the user.

Inside of the student mode, the user has a number of functions that they can use including the search and sort function. To start off, the program first utilizes a store function as shown in Figure 16 to store the data from the written .txt file into a LinkedList. The data needs to be stored into a text document because it needs to be stored and accessed when running the program at a different time.

Figure 17: storeElements function/method

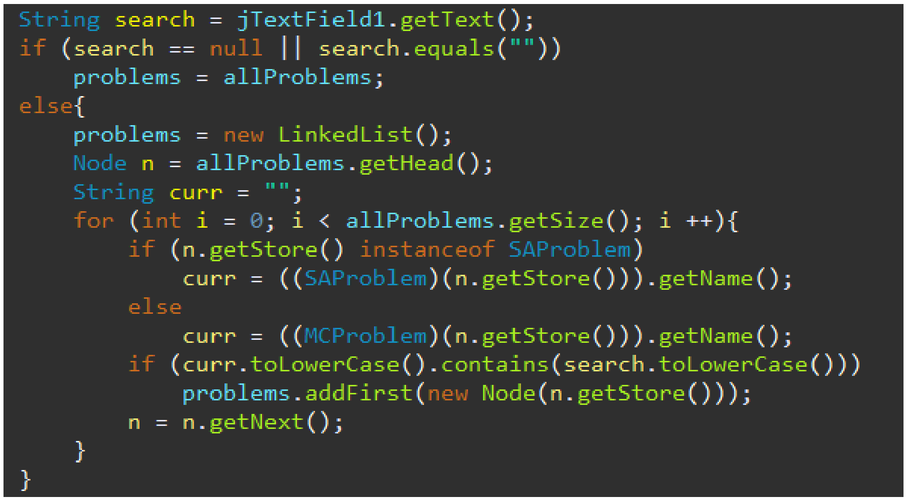


```
public void storeElements(){
    try{
        BufferedReader reader = new BufferedReader(new FileReader("Text Files/problems.txt"));
        String line = reader.readLine();
        while (line != null && !line.equals("")){
            String name = reader.readLine();
            String statement = "";
            String temp = reader.readLine();
            while (temp != null && !temp.equals("*")){
                if (!statement.equals(""))
                    statement += ("@" + temp);
                else
                    statement += temp;
                temp = reader.readLine();
            }
            String difficulty = reader.readLine();
            String area = reader.readLine();
            String type = reader.readLine();
            String answerSA;
            char answerMC;
            MCProblem probMC;
            SAProblem probSA;
            Node n;
            if (type.equals("SA")){
                answerSA = reader.readLine();
                probSA = new SAProblem(name, statement, difficulty, answerSA, area);
                n = new Node(probSA);
                allProblems.addFirst(n);
            }
            else{
                String hold = reader.readLine();
                answerMC = hold.charAt(0);
                probMC = new MCProblem(name, statement, difficulty, answerMC, area);
                n = new Node(probMC);
                allProblems.addFirst(n);
            }
            line = reader.readLine();
        }
    }
    catch (IOException e){
        jOptionPane1 = new JOptionPane();
        JFrame f = new JFrame();
        JOptionPane.showMessageDialog(f,"An Unexpected Error has occured");
    }

}
```

A BufferedReader is used to go through the text file itself. Based on the previous formatting that the text file was purposely written with, whilst reading it, all it is required to do is to undo the previous formatting to obtain the pieces of desired information.

The searching algorithm I used is shown below.

Figure 18: Searching Algorithm

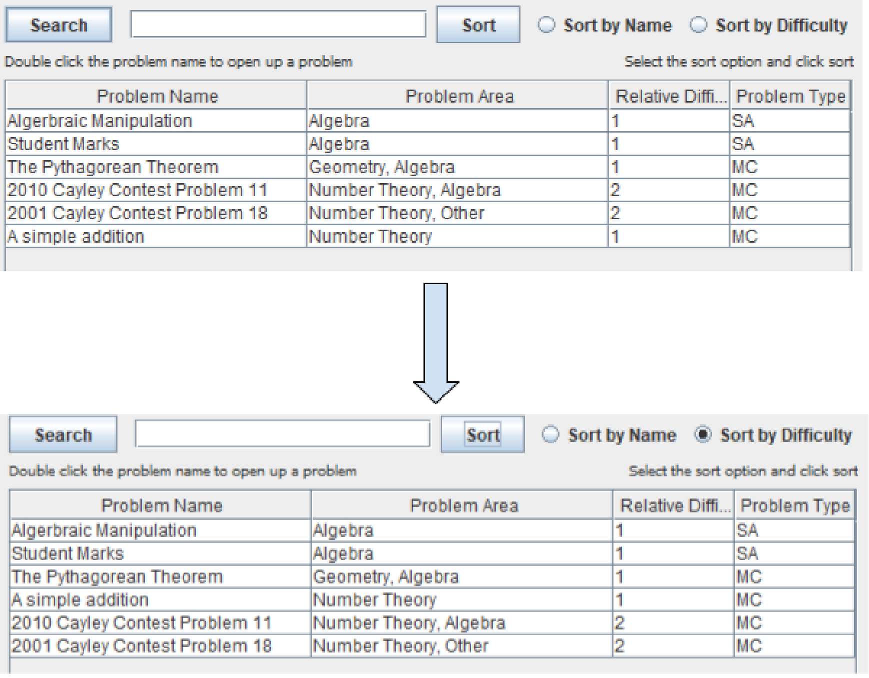


```
String search = jTextField1.getText();
if (search == null || search.equals(""))
    problems = allProblems;
else{
    problems = new LinkedList();
    Node n = allProblems.getHead();
    String curr = "";
    for (int i = 0; i < allProblems.getSize(); i ++){
        if (n.getStore() instanceof SAProblem)
            curr = ((SAProblem)(n.getStore())).getName();
        else
            curr = ((MCProblem)(n.getStore())).getName();
        if (curr.toLowerCase().contains(search.toLowerCase()))
            problems.addFirst(new Node(n.getStore()));
        n = n.getNext();
    }
}
```

The searching algorithm is a simple linear search through the LinkedList. If a problem is found, then it is added to a new LinkedList. In order to distinguish between the two possible problem types, the keyword instanceof is used.

The program also has features to sort the problems both by name alphabetically and by difficulty numerically.

Figure 19: Sorted and Unsorted


Search
Sort
○ Sort by Name
○ Sort by Difficulty
Double click the problem name to open up a problem
Select the sort option and click sort

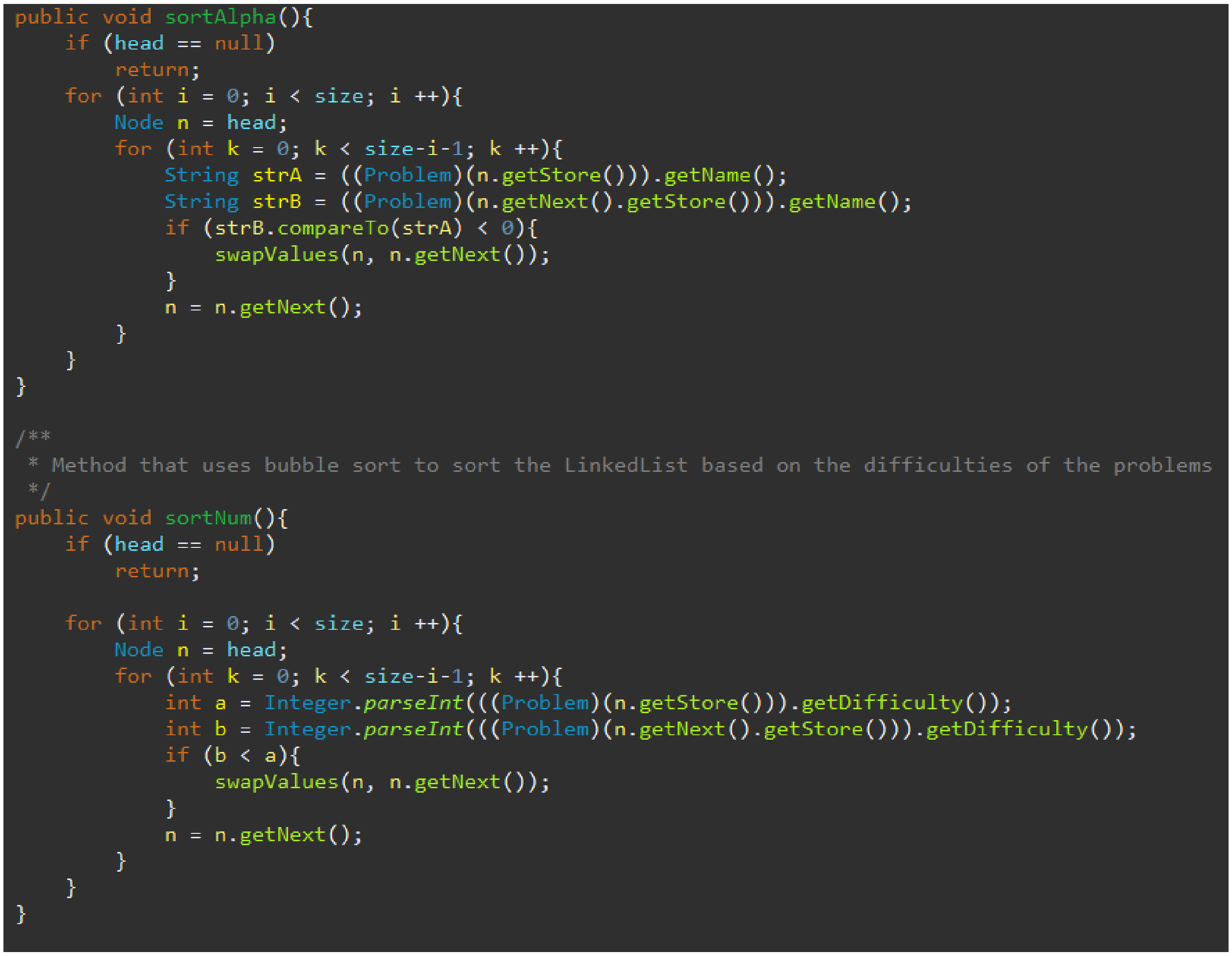
Problem Name	Problem Area	Relative Diffi...	Problem Type
Algerbraic Manipulation	Algebra	1	SA
Student Marks	Algebra	1	SA
The Pythagorean Theorem	Geometry, Algebra	1	MC
2010 Cayley Contest Problem 11	Number Theory, Algebra	2	MC
2001 Cayley Contest Problem 18	Number Theory, Other	2	MC
A simple addition	Number Theory	1	MC

Search
Sort
○ Sort by Name
● Sort by Difficulty
Double click the problem name to open up a problem
Select the sort option and click sort

Problem Name	Problem Area	Relative Diffi...	Problem Type
Algerbraic Manipulation	Algebra	1	SA
Student Marks	Algebra	1	SA
The Pythagorean Theorem	Geometry, Algebra	1	MC
A simple addition	Number Theory	1	MC
2010 Cayley Contest Problem 11	Number Theory, Algebra	2	MC
2001 Cayley Contest Problem 18	Number Theory, Other	2	MC

Figure 20 illustrates the sorting of the problems by difficulty.

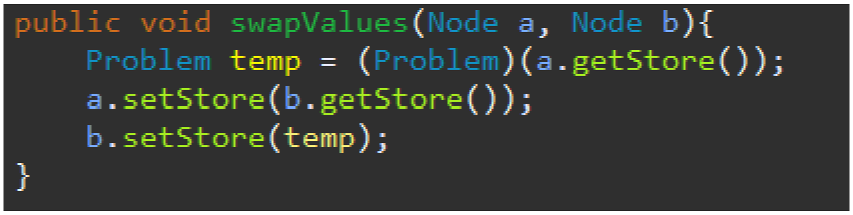
Figure 20: The Sorting Algorithms



```
public void sortAlpha(){
    if (head == null)
        return;
    for (int i = 0; i < size; i ++){
        Node n = head;
        for (int k = 0; k < size-i-1; k ++){
            String strA = ((Problem)(n.getStore())).getName();
            String strB = ((Problem)(n.getNext().getStore())).getName();
            if (strB.compareTo(strA) < 0){
                swapValues(n, n.getNext());
            }
            n = n.getNext();
        }
    }
}

/**
 * Method that uses bubble sort to sort the LinkedList based on the difficulties of the problems
 */
public void sortNum(){
    if (head == null)
        return;

    for (int i = 0; i < size; i ++){
        Node n = head;
        for (int k = 0; k < size-i-1; k ++){
            int a = Integer.parseInt(((Problem)(n.getStore())).getDifficulty());
            int b = Integer.parseInt(((Problem)(n.getNext().getStore())).getDifficulty());
            if (b < a){
                swapValues(n, n.getNext());
            }
            n = n.getNext();
        }
    }
}
```

The sorting algorithms themselves are as shown. Since for a LinkedList, one does not have access to a node at a specific index, bubble sort would be far superior to another sort such as insertion sort. As such, I used bubble sort to sort the LinkedList numerically and lexicographically. Both sorting algorithms make use of the swapping method shown in Figure 20.

Figure 21: The swapping method

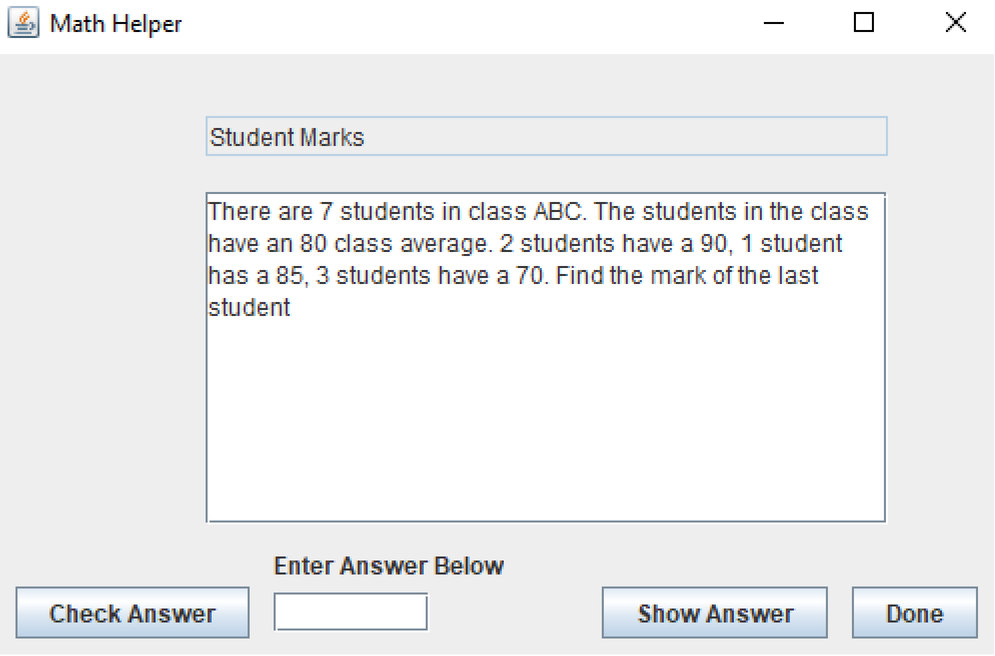


```
public void swapValues(Node a, Node b){
    Problem temp = (Problem)(a.getStore());
    a.setStore(b.getStore());
    b.setStore(temp);
}
```

The swapping method is quite simple. The simple observation can be made that instead of swapping the nodes themselves, it simply suffices to swap the data stored in each node.

QuestionDisplay:

Figure 22: QuestionDisplay

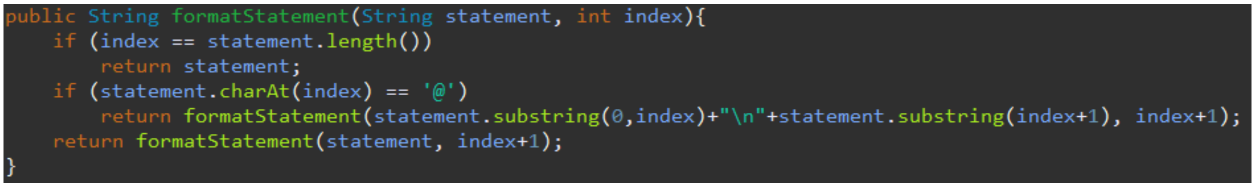


Student Marks

There are 7 students in class ABC. The students in the class have an 80 class average. 2 students have a 90, 1 student has a 85, 3 students have a 70. Find the mark of the last student

The QuestionDisplay menu is quite simple and makes use of previously shown methods found in other classes such as the checkAnswer methods in the SAProblem and the MCProblem classes.

Figure 23: Formatting Method

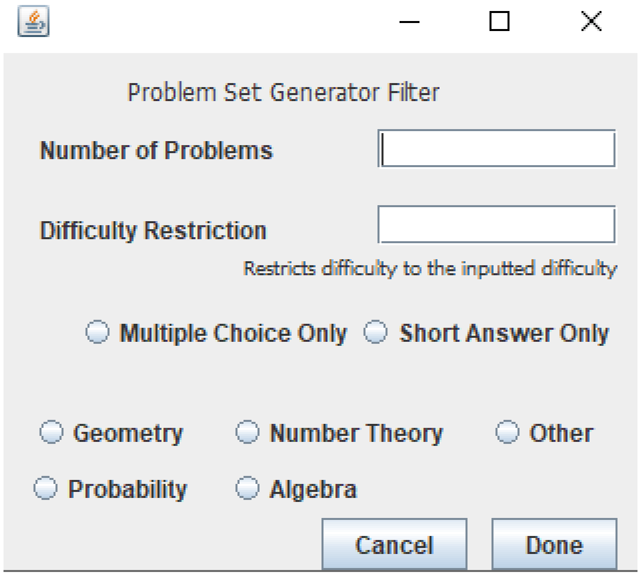


```
public String formatStatement(String statement, int index){
    if (index == statement.length())
        return statement;
    if (statement.charAt(index) == '@')
        return formatStatement(statement.substring(0,index)+"\n"+statement.substring(index+1), index+1);
    return formatStatement(statement, index+1);
}
```

A recursive method is used to format the text for the displayed question statement. If the method is not formatted, the lines separating the different choices for multiple choice questions will be messed up. A recursive method was chosen for this purpose because it requires less lines when compared to a non recursive method.

Question Set Generation:

Figure 24: Question Set Generation



Problem Set Generator Filter

Number of Problems

Difficulty Restriction

Restricts difficulty to the inputted difficulty

○ Multiple Choice Only ○ Short Answer Only

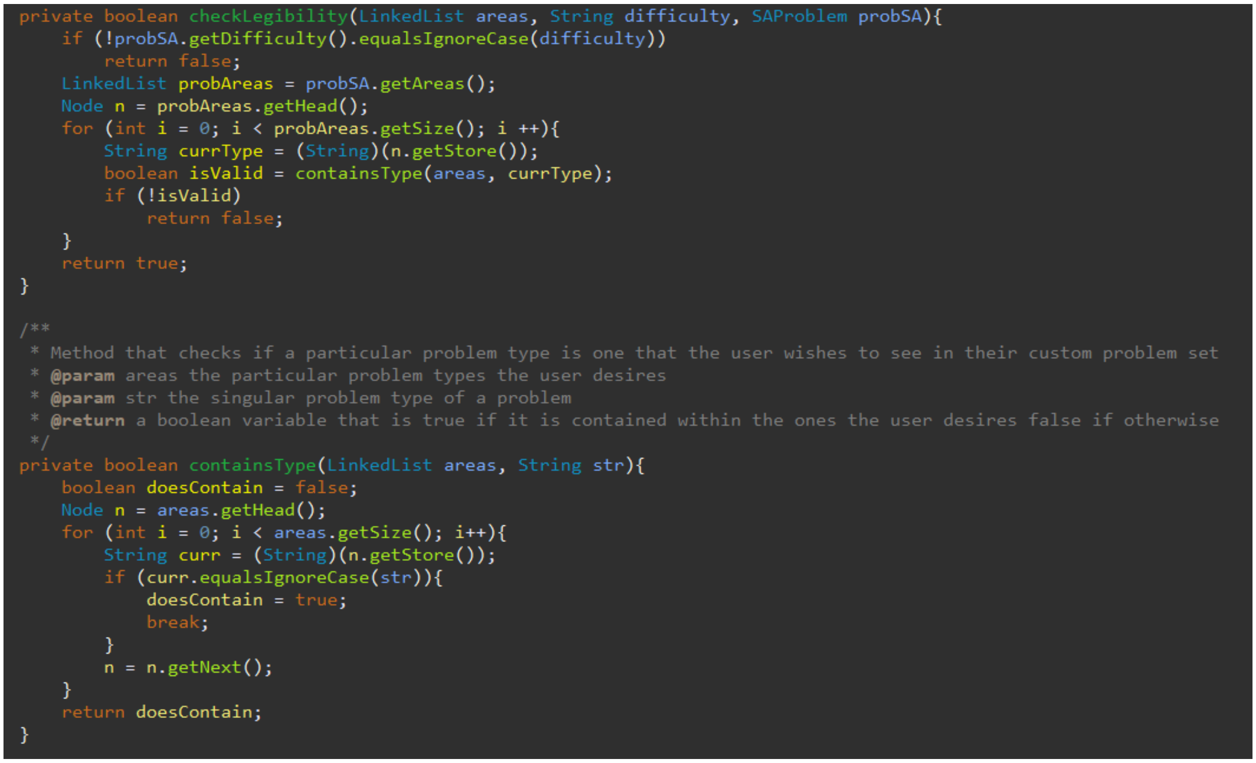
○ Geometry ○ Number Theory ○ Other

○ Probability ○ Algebra

Cancel Done

The question set generation filter enables the user to determine the type of question set wanted.

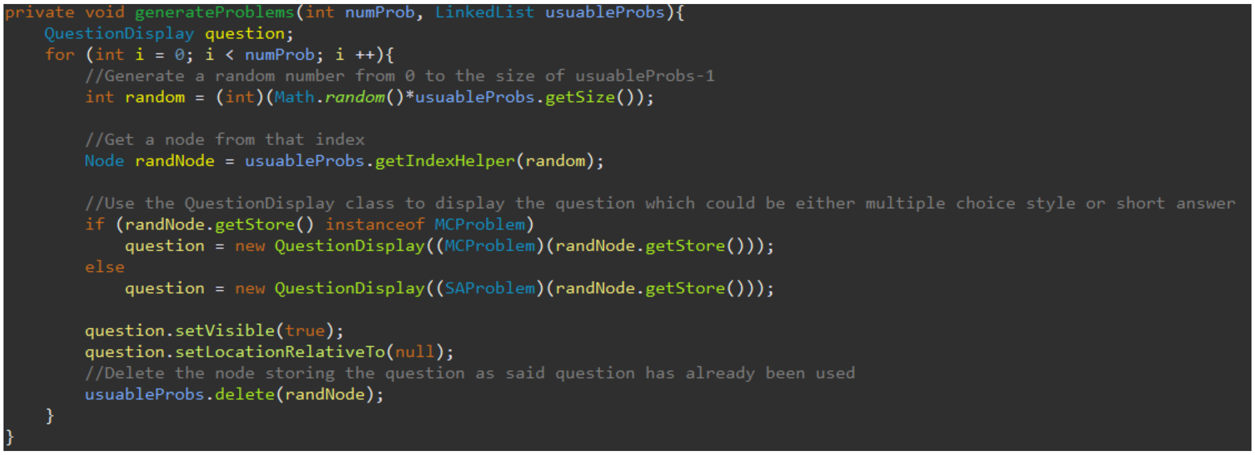
Figure 25: checkLegibility and containsType methods



```
private boolean checkLegibility(LinkedList areas, String difficulty, SAProblem probSA){
    if (!probSA.getDifficulty().equalsIgnoreCase(difficulty))
        return false;
    LinkedList probAreas = probSA.getAreas();
    Node n = probAreas.getHead();
    for (int i = 0; i < probAreas.getSize(); i ++){
        String currType = (String)(n.getStore());
        boolean isValid = containsType(areas, currType);
        if (!isValid)
            return false;
    }
    return true;
}

/**
 * Method that checks if a particular problem type is one that the user wishes to see in their custom problem set
 * @param areas the particular problem types the user desires
 * @param str the singular problem type of a problem
 * @return a boolean variable that is true if it is contained within the ones the user desires false if otherwise
 */
private boolean containsType(LinkedList areas, String str){
    boolean doesContain = false;
    Node n = areas.getHead();
    for (int i = 0; i < areas.getSize(); i++){
        String curr = (String)(n.getStore());
        if (curr.equalsIgnoreCase(str)){
            doesContain = true;
            break;
        }
        n = n.getNext();
    }
    return doesContain;
}
```

The checkLegibiltiy and containsType methods are both used in a 2 part process to check if a problem fulfills the desired filter requests of the user. The checkLegibility loops through each of the problem types within a specific problem and the containsType method checks if each of the specific problem types matches up with the filtered problem types of the user. It is also important to note that method overloading is used for the method checkLegibility to support the MCProblem class and the SAProblem class.

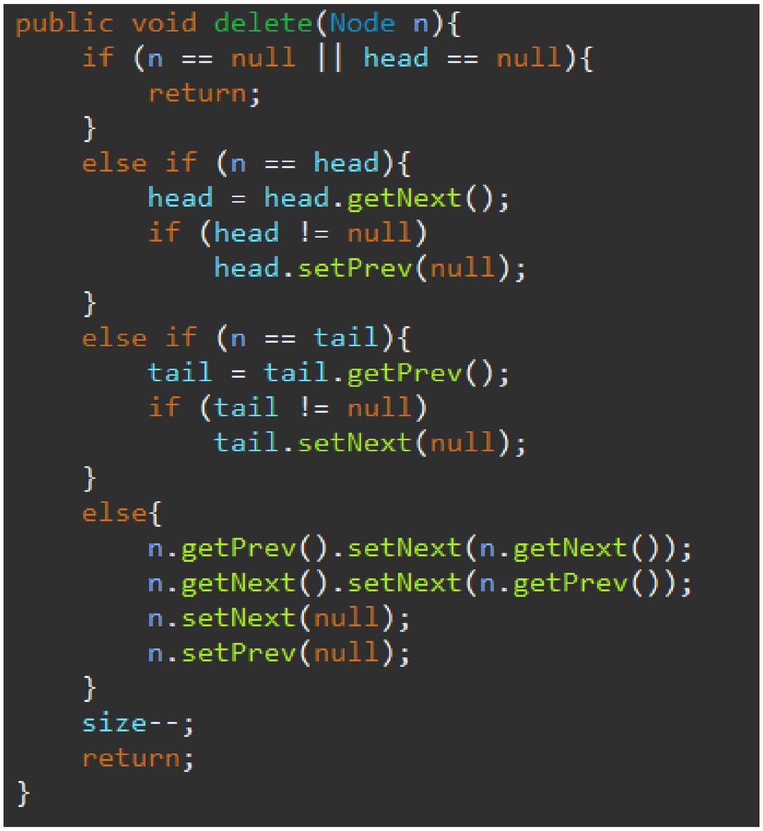
Figure 26: generateProblems Method



```
private void generateProblems(int numProb, LinkedList usuableProbs){
    QuestionDisplay question;
    for (int i = 0; i < numProb; i ++){
        //Generate a random number from 0 to the size of usuableProbs-1
        int random = (int)(Math.random()*usuableProbs.getSize());

        //Get a node from that index
        Node randNode = usuableProbs.getIndexHelper(random);

        //Use the QuestionDisplay class to display the question which could be either multiple choice style or short answer
        if (randNode.getStore() instanceof MCProblem)
            question = new QuestionDisplay((MCProblem)(randNode.getStore()));
        else
            question = new QuestionDisplay((SAProblem)(randNode.getStore()));

        question.setVisible(true);
        question.setLocationRelativeTo(null);
        //Delete the node storing the question as said question has already been used
        usuableProbs.delete(randNode);
    }
}
```

After generating the list of usable problems the Math.random() method is used to generate the problems at random.The constant deletion capabilities of a doubly linked list is utilized for maximum efficiency. The length of the LinkedList is then modified which is also reflected in the Math.random generation.

Figure 27: Constant Deletion



```
public void delete(Node n){
    if (n == null || head == null){
        return;
    }
    else if (n == head){
        head = head.getNext();
        if (head != null)
            head.setPrev(null);
    }
    else if (n == tail){
        tail = tail.getPrev();
        if (tail != null)
            tail.setNext(null);
    }
    else{
        n.getPrev().setNext(n.getNext());
        n.getNext().setNext(n.getPrev());
        n.setNext(null);
        n.setPrev(null);
    }
    size--;
    return;
}
```

Words: 1156