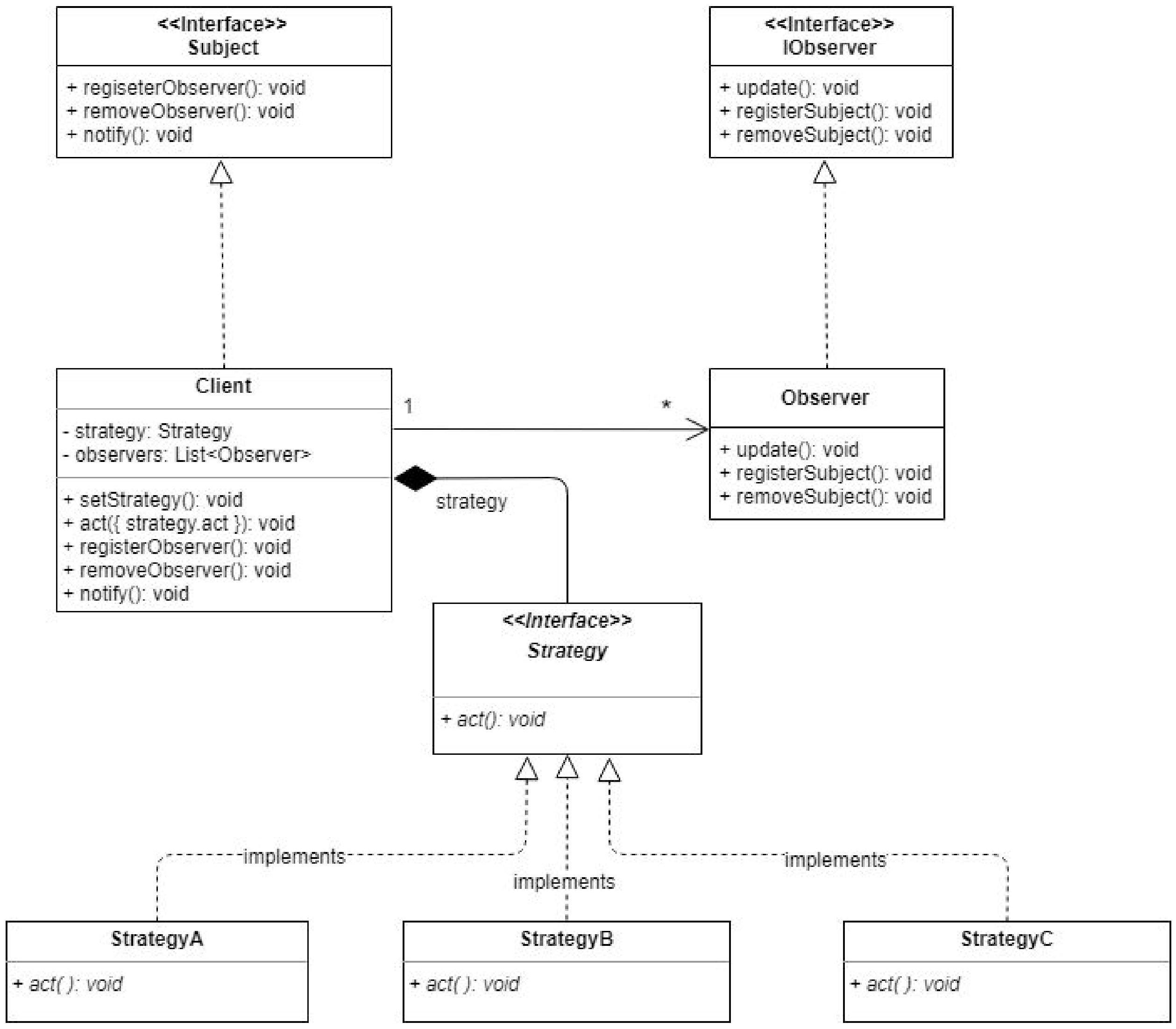
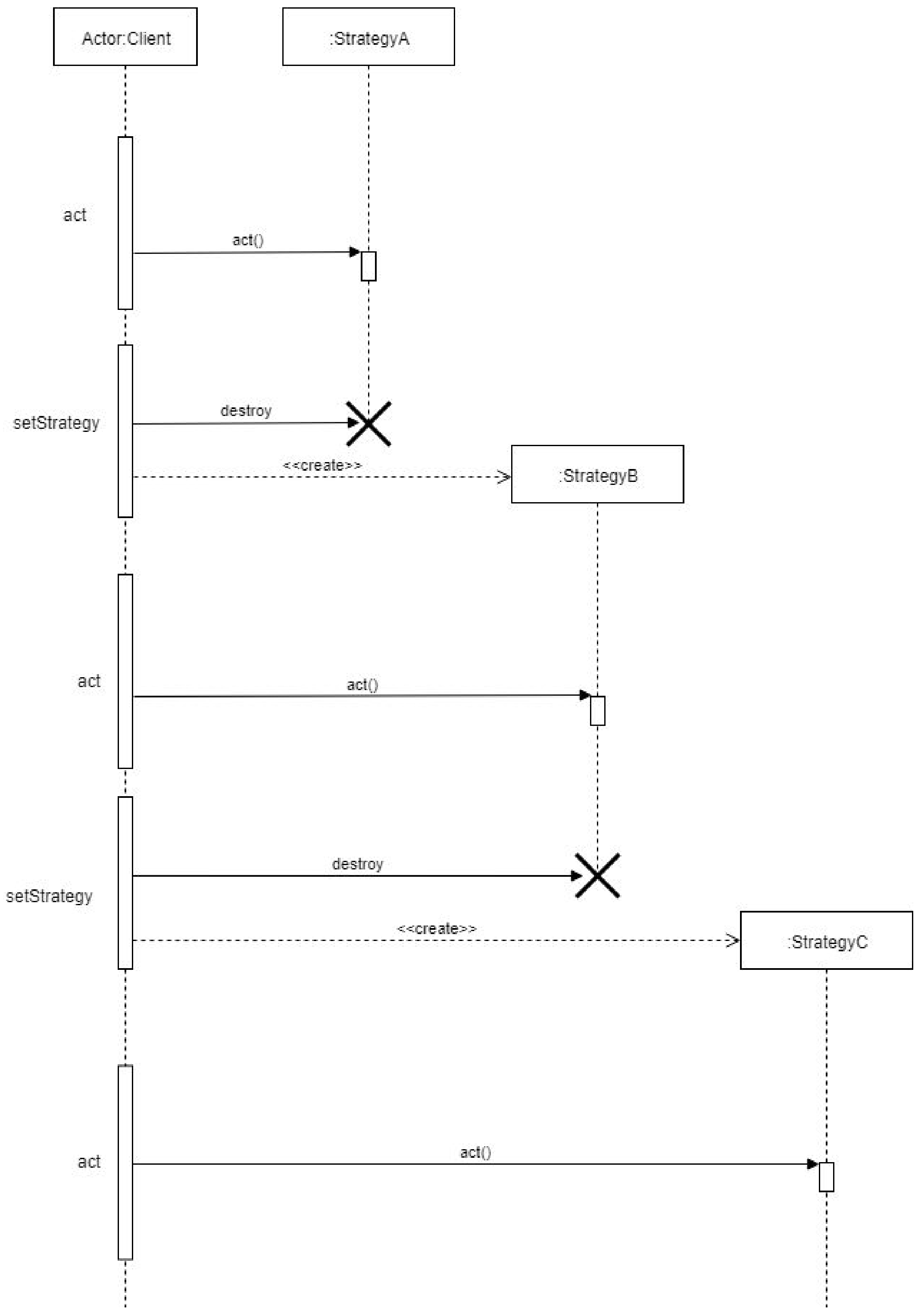
Exercise 1:

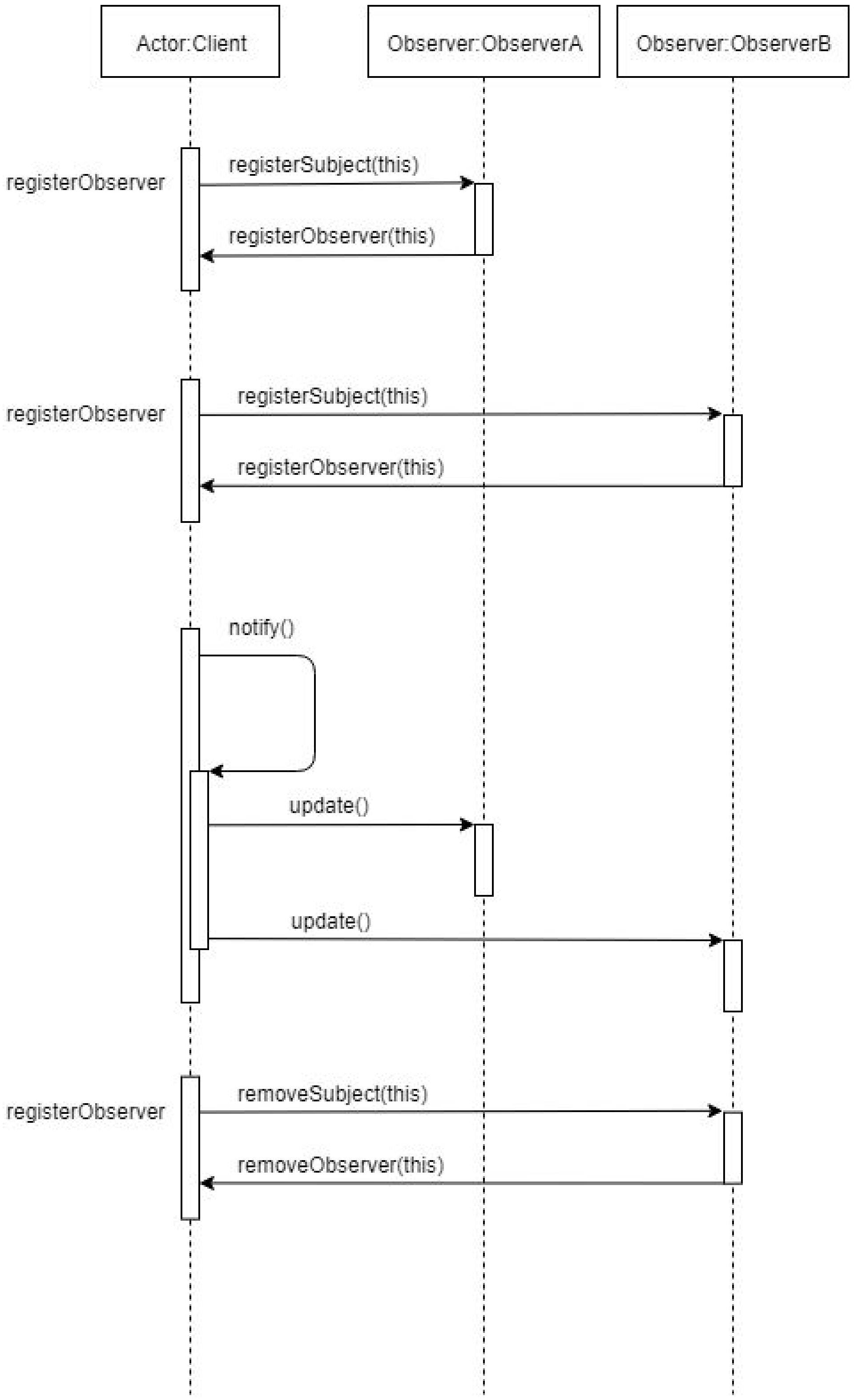
Below, the *Client* class participates in both the Strategy Pattern and Observer Pattern.



For the second portion of this exercise, we made the decision to create two separate sequence diagrams for each pattern, due to the fact that neither strategy pattern in our above model affects the operation of the other. Below is a sequence diagram from the perspective of the Strategy Pattern:



Below is a sequence diagram from the perspective of the Observer Pattern:



Exercise 2:
a) Assume during your team's last sprint, that they completed 32 story points using a 3-person team working in sprints of 3 weeks for a total of 45-man days. Calculate your team's estimated velocity for the next sprint if we still have 3-week sprints, but you now added 2 engineers to the team, and one of them can only work 80% of the time.

Focus factor = 32/45 = 71%

45 man days from the original team + 15 man days from the full time new engineer + 12 man days from the engineer that can work 80% of the time = 72 man days

72 man days this sprint * 71% focus factor last sprint = Estimated Velocity is 51.2 (story points)

b) How would you estimate a focus factor for a brand-new team?

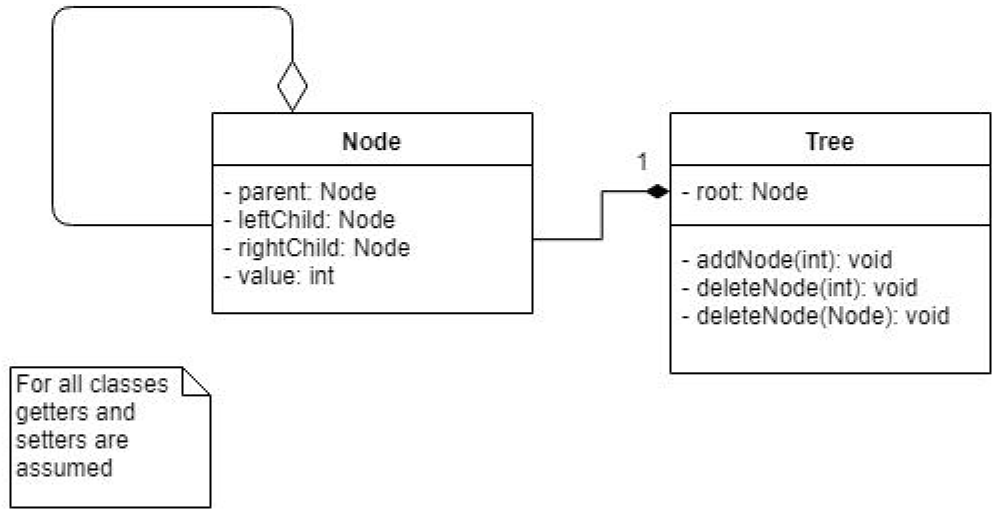
We can assume a brand-new team has a focus factor of 70%. After the first sprint the focus factor will start to adjust to the team.

c) We looked at using poker using semi-Fibonacci sequences to estimate story points. Think of another way to estimate story points and explain it. Is it better or worse than poker?

You could have a silent bidding system where the total story points are incremented up. Everyone in the room raises their hand if they think the story points are larger. If at any time less than half the room thinks it will take more time then there will be a discussion on if the story points should go higher or stay the same. After 7 points it increments up by 2 and after 20 points it increments up by 10 to 40. After 40 everyone has a discussion on the project and if they think the project should get split up more or if there are any issues.

This would be worse than poker as it would probably take more time to come to a conclusion about the number of story points a project will take. Participants would also be able to see each other which might make people more likely to put their hands down if they see others putting their hands down, unlike poker where participants don't know what other people are putting down while they are choosing their card.

d) Draw a UML class diagram of a binary tree. Each node contains an integer.

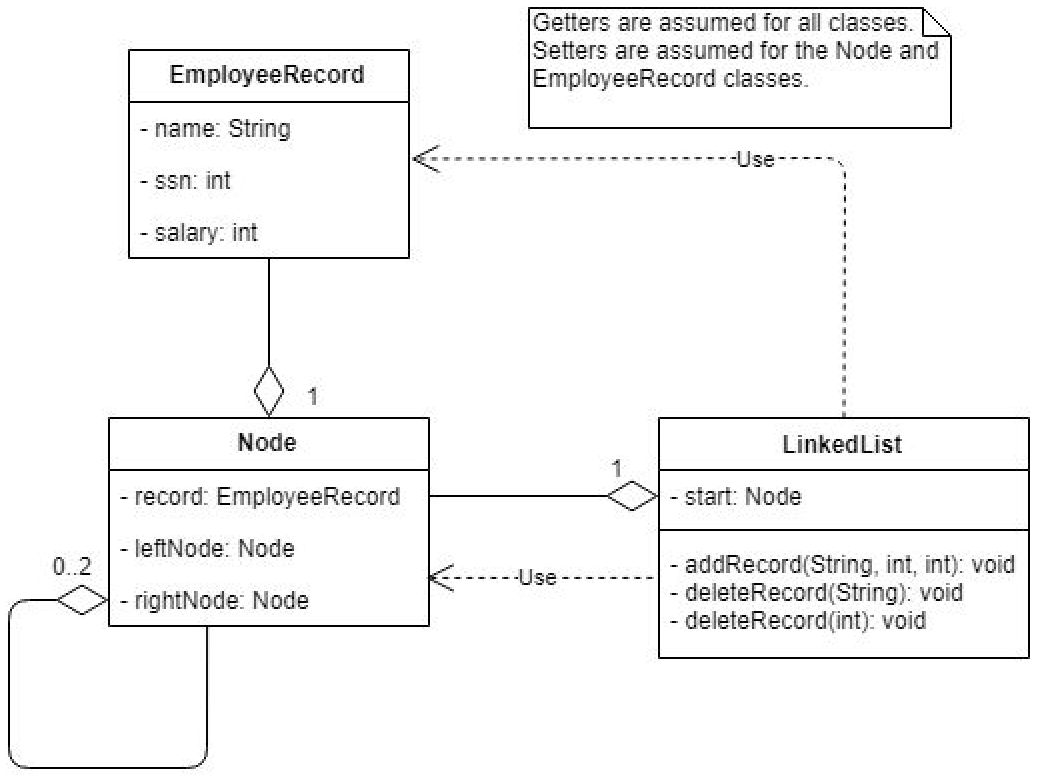


e) Provide the corresponding object-oriented code that implements your binary tree design from part d.

```
private class Node {
   int value;
   Node parent;
   Node leftChild;
   Node rightChild;

   private Node(int value) {
       this.value = value
       leftChild = null;
       rightChild = null;
       parent = null;
   }

   public int getValue() {
       return value;
   }

   public Node getLeftChild() {
       return leftChild;
   }

   public Node getRightChild() {
       return rightChild;
   }

   public Node getParent() {
       return parent;
   }
```

```
    public void setLeftChild(Node leftChild) {
        this.leftChild = leftChild;
    }

    public void setRightChild(Node rightChild) {
        this.rightChild = rightChild;
    }

    public void setValue(int value) {
        this.value = value;
    }

    public void setParent(Node parent) {
        this.parent = parent;
    }
}

private class Tree {
    Node root;

    private Tree() {
        root = new Node();
    }

    private Node getRoot() {
        return root;
    }

    private void setRoot(Node root) {
        this.root = root;
    }

    private void addNode(int newNodeValue) {
        // This method creates a new Node object with the given value, then adds it to the tree
    }

    private void deleteNode(int nodeValue) {
        // This method finds a node in the tree based on its value, then deletes it from the tree
    }

    private void deleteNode(Node toDelete) {
        // This method finds a given node in the tree, then deletes it
    }
}
```

f) Draw a UML class diagram of a linked list that contains Employee records as data. An Employee record has a name, a social security number, and a salary.


Getters are assumed for all classes. Setters are assumed for the Node and EmployeeRecord classes.
EmployeeRecord
- name: String
- ssn: int
- salary: int
Use
1
Node
- record: EmployeeRecord
- leftNode: Node
- rightNode: Node
0..2
1
LinkedList
- start: Node
- addRecord(String, int, int): void
- deleteRecord(String): void
- deleteRecord(int): void
Use

g) Provide the corresponding object-oriented code that implements your linked List from part f

.

```
public class EmployeeRecord {

    private String name;

    private Integer ssn;
    private Integer salary;

    public EmployeeRecord(String name, Integer ssn, Integer salary) {
        this.name = name;
        this.ssn = ssn;
        this.salary = salary;
    }

    public Integer getSalary() {
        return salary;
    }

    public Integer getSSN() {
        return ssn;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setSalary(Integer salary) {
        this.salary = salary;
    }

    public void setSSN(Integer ssn) {
        this.ssn = ssn;
    }
}
```

```java
public class Node {

    private Node leftNode;
    private Node rightNode;

    private EmployeeRecord record;

    public Node(Node leftNode, Node rightNode, EmployeeRecord record) {
        this.leftNode = leftNode;
        this.rightNode = rightNode;
        this.record = record;
    }

    public Node getLeftNode() {
        return leftNode;
    }

    public EmployeeRecord getRecord() {
        return record;
    }

    public Node getRightNode() {
        return rightNode;
    }

    public void setRightNode(Node rightNode) {
        this.rightNode = rightNode;
    }

    public void setLeftNode(Node leftNode) {
        this.leftNode = leftNode;
    }

    public void setRecord(EmployeeRecord record) {
        this.record = record;
    }
}
```

```java
public class LinkedList {
    private Node start;
    private Integer length;

    public LinkedList() {
        start = null;
        length = 0;
    }

    public Node getStart() {
        return start;
    }

    public Integer getLength() {
        return length;
    }

    public void addRecord(String name, Integer ssn, Integer salary) {
        Node toAdd = new Node(null, null, null);
        toAdd.setRecord(new EmployeeRecord(name, ssn, salary));
        if (start != null) {
            toAdd.setRightNode(start);
            start.setLeftNode(toAdd);
            start = toAdd;
            length++;
        } else {
            start = toAdd;
            length++;
        }
    }
    public void deleteRecord(String name) {
        Node current = start;
        for (int i = 0; i < length; i++) {
            if (!current.getRecord().getName().equals(name)) {
                current = current.getRightNode();
            } else {
                if (current.getLeftNode() != null) {
                    current.getLeftNode().setRightNode(current.getRightNode());
                }

                if (current.getRightNode() != null) {
                    current.getRightNode().setLeftNode(current.getLeftNode());
                }

                if (current == start) {
                    start = current.getRightNode();
                }

                length--;
                break;
            }
        }
    }

    public void deleteRecord(int ssn) {
        Node current = start;
        for (int i = 0; i < length; i++) {
            if (current.getRecord().getSSN() != ssn) {
                current = current.getRightNode();
            } else {
                if (current.getLeftNode() != null) {
                    current.getLeftNode().setRightNode(current.getRightNode());
                }

                if (current.getRightNode() != null) {
                    current.getRightNode().setLeftNode(current.getLeftNode());
                }

                if (current == start) {
                    start = current.getRightNode();
                }

                length--;
                break;
            }
        }
    }
}
```