Data Structures HW3

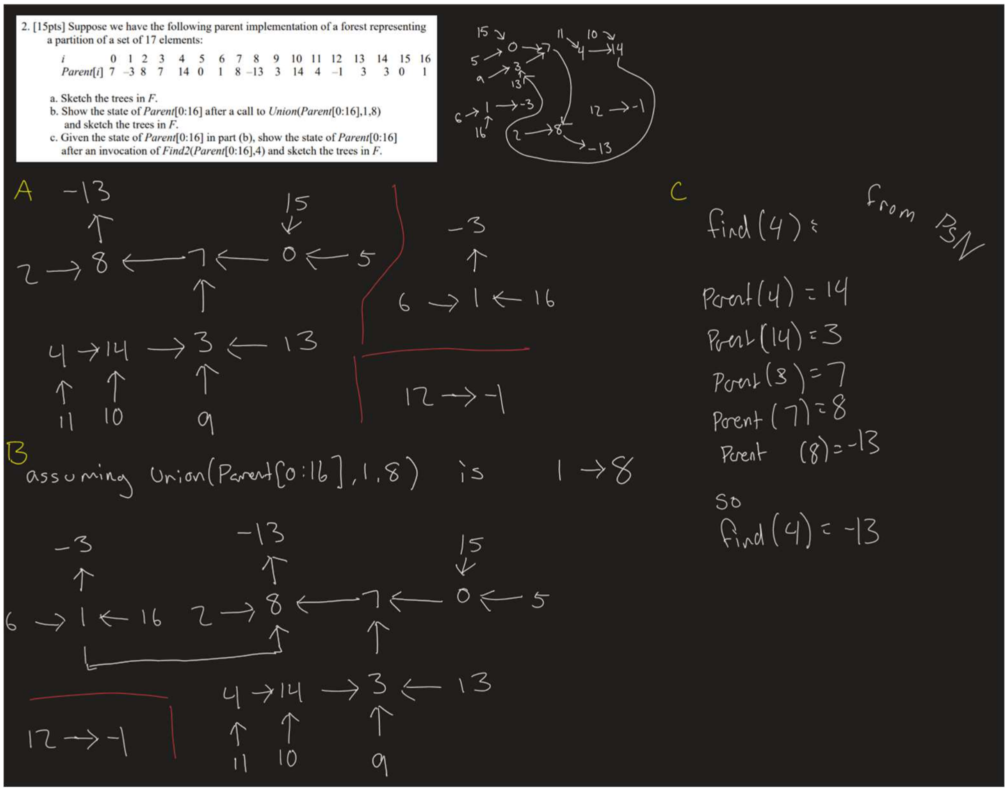
1. Problem 1
    a. To change the priority of an element in a min-heap priority queue, we must perform different operations depending on whether the priority is increased or decreased. If it is increased, then we must compare the new value of the node to the values of its children. If one of the children has a lower priority than the modified node, then we must swap the positions of the two nodes and then run the same check again recursively until no violations are detected. Similarly, if the priority of the node is decreased, then we must compare it to the value of its parent. If the parent now has a greater value than the modified node, then we must swap their positions and repeat until the parent node is less than the modified node
    b. Pseudocode:

```
ChangePriority(Q,x,v):
    i = FindIndex(Q,x)
    oldPriority = Q[i] -> priority
    Q[i] -> priority = v
    If(oldPriority == v):
        //The priority has not been changed. Do nothing
    Else if(oldPriority < v):
        //The priority has been increased
        Satisfied = false
        While(!satisfied):
            smallerChildIndex = 0
            If(Q[2i+1] -> priority < Q[i] -> priority):
                smallerChildIndex = 2i+1
            Else if(Q[2i+2] -> priority < Q[i] -> priority):
                smallerChildIndex = 2i+2
            If(smallerChildIndex == 0):
                Satisfied = true
            Else:
                i = swap(Q,i,smallerChildIndex)
    else:
        //The priority has been decreased
```

```
Satisfied = false
While(!satisfied):
    If(Q[i-1 / 2] -> priority > Q[i] -> priority):
        i = swap(Q,i,(i-1/2)
    else:
        satisfied = true
```

c. Analysis
   i. Worst case complexity
      1. In the worst case, either x is at the top of the tree and must be move to the bottom, or it is at the bottom of the tree and must be moved to the top
      2. So, x will have to be swapped once for each layer of the tree
      3. The depth of a complete tree is approximately log(n), so the worst case complexity of the algorithm will be approximately log(n) as well

2. Problem 2



2. [15pts] Suppose we have the following parent implementation of a forest representing a partition of a set of 17 elements:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Parent[*i*]	7	–3	8	7	14	0	1	8	–13	3	14	4	–1	3	3	0	1

a. Sketch the trees in *F*.
b. Show the state of *Parent*[0:16] after a call to *Union*(*Parent*[0:16],1,8) and sketch the trees in *F*.
c. Given the state of *Parent*[0:16] in part (b), show the state of *Parent*[0:16] after an invocation of *Find2*(*Parent*[0:16],4) and sketch the trees in *F*.

A

B
assuming Union(Parent[0:16],1,8) is 1 → 8

C
Find(4) = from BSN

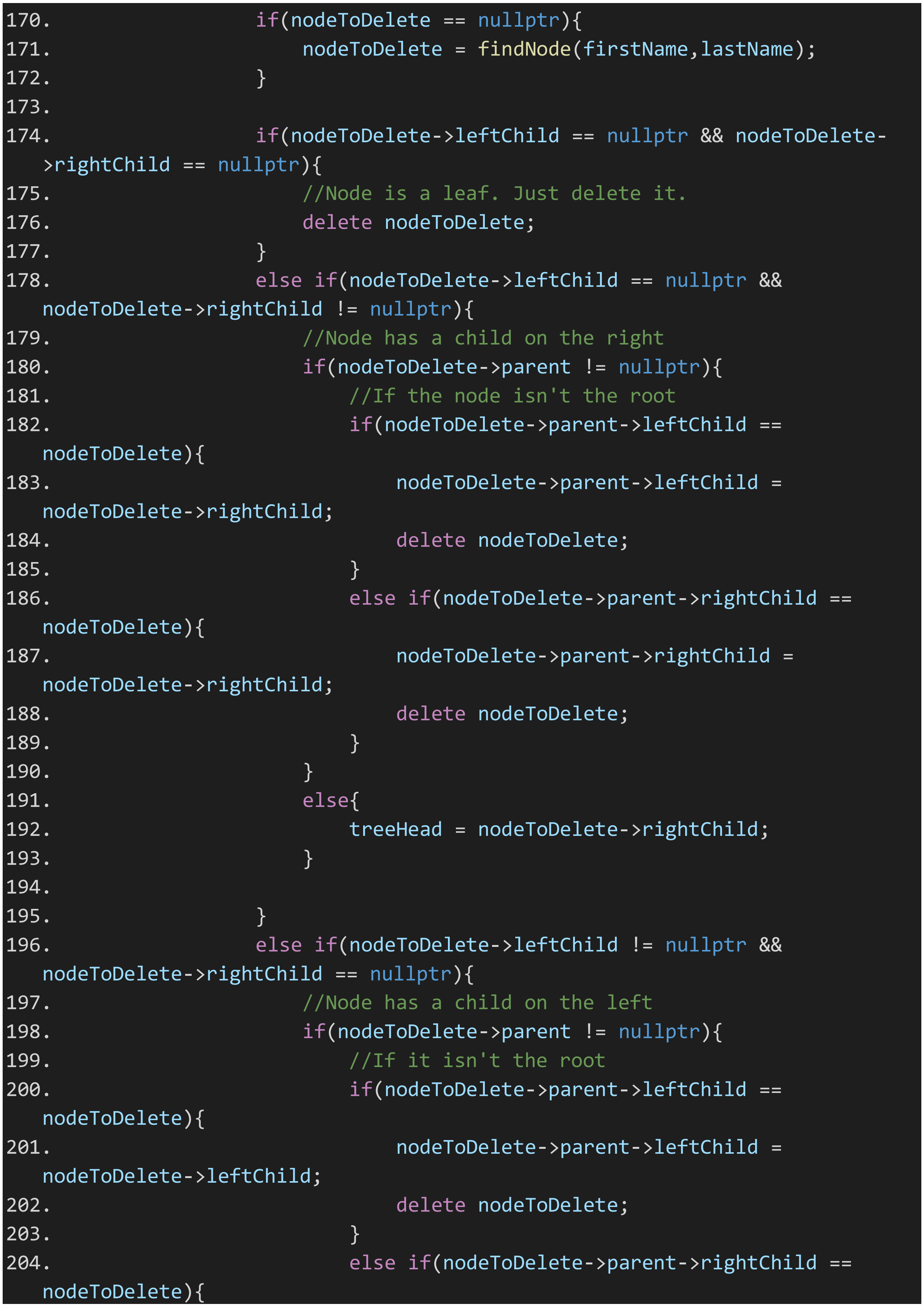
Parent(4) = 14
Parent(14) = 3
Parent(3) = 7
Parent(7) = 8
Parent (8) = -13

So
Find(4) = -13

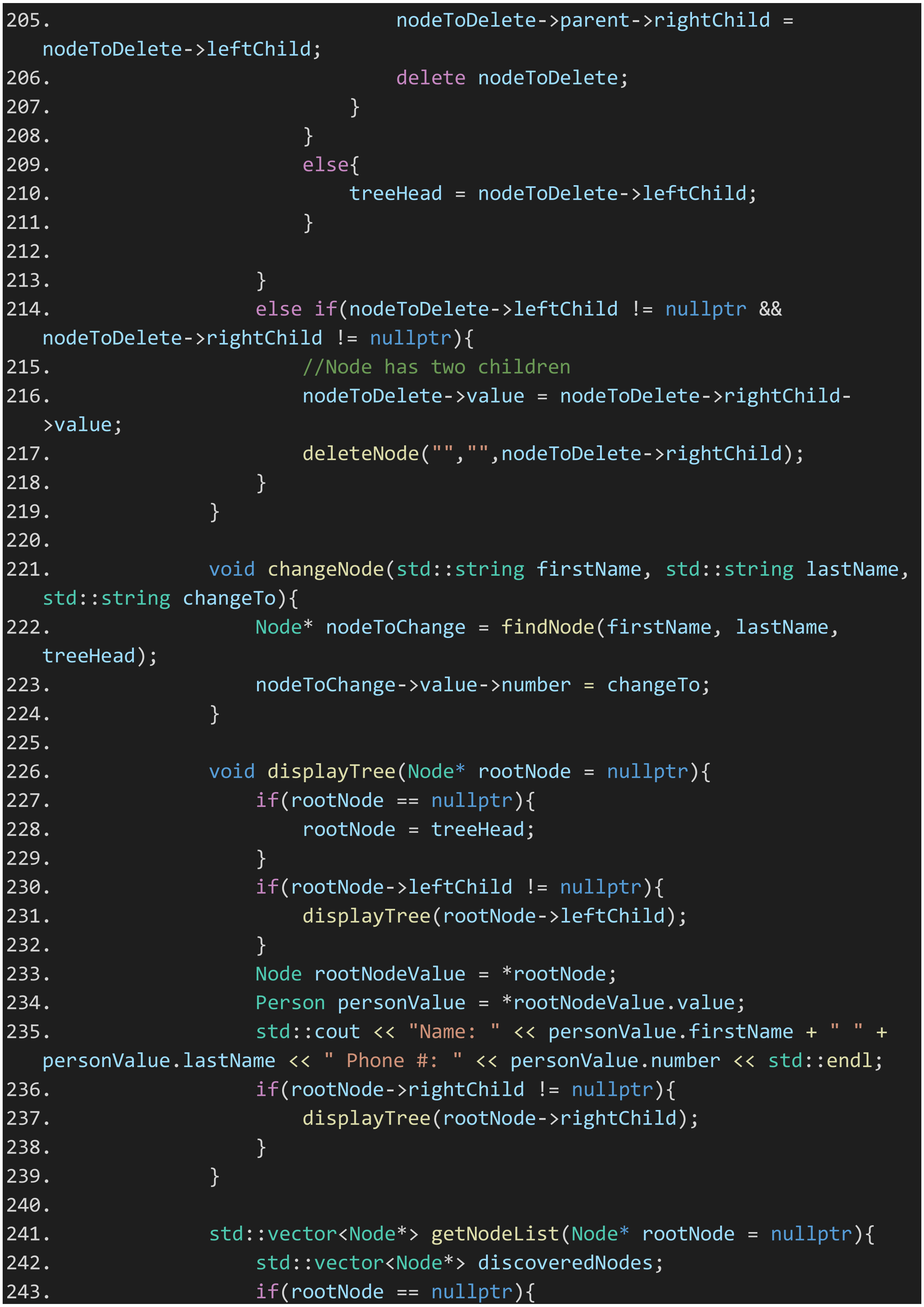
a.

3. Programming Assignment

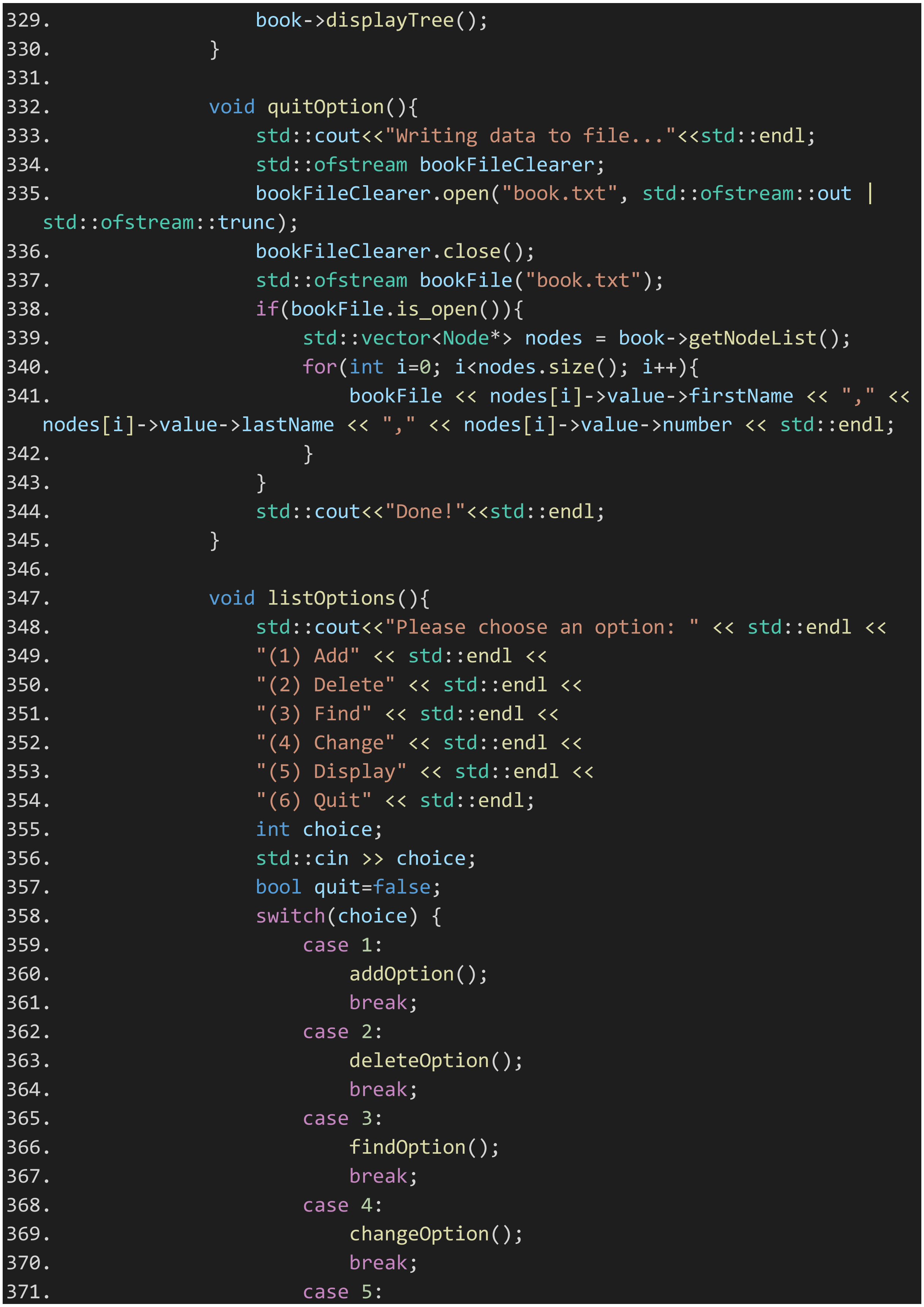
```cpp
#include <string>
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>

class Person {
    public:
        std::string firstName;
        std::string lastName;
        std::string number;

        Person(std::string first, std::string last, std::string newNumber){
            firstName = first;
            lastName = last;
            number = newNumber;
        }

        Person(const Person &oldPerson){
            firstName = oldPerson.firstName;
            lastName = oldPerson.lastName;
            number = oldPerson.number;
        }

        ~Person(){

        }

        std::string getFullName(){
            std::string fullName = firstName + " " + lastName;
            return fullName;
        };

};

class Node {
    public:
        Person* value;
        Node* leftChild;
        Node* rightChild;
        Node* parent;

        Node(Person* person){
            value = person;
            leftChild = nullptr;
            rightChild = nullptr;
            parent = nullptr;
        }

        Node(Person* value, Node* leftChild, Node* rightChild){
            value = value;
            leftChild = leftChild;
            rightChild = rightChild;
            parent = nullptr;
        }

        Node(const Node &oldNode){
            value = oldNode.value;
            leftChild = oldNode.leftChild;
            rightChild = oldNode.rightChild;
            parent = oldNode.parent;
        }

        ~Node(){
            if(leftChild != nullptr){
                leftChild->~Node();
            }
            if(rightChild != nullptr){
                rightChild->~Node();
            }

        }
};

class Book {
    public:
        Node* treeHead;

        Book(){
            treeHead = nullptr;
        }

        Book(const Book &oldBook){
            treeHead = oldBook.treeHead;
        }

        ~Book(){
            delete treeHead;
        }

        void treeInsert(Node* newNode, Node* rootNode){
            if(rootNode == 0x0){
                treeHead = newNode;
            }
            else if(rootNode->value->lastName < newNode->value->lastName){
                if(rootNode->rightChild == nullptr){
                    newNode->parent = rootNode;
                    rootNode->rightChild = newNode;
                }
                else{
                    treeInsert(newNode, rootNode->rightChild);
                }
            }
            else if(rootNode->value->lastName > newNode->value->lastName){
                if(rootNode->leftChild == nullptr){
                    newNode->parent = rootNode;
                    rootNode->leftChild = newNode;
                }
                else{
                    treeInsert(newNode, rootNode->leftChild);
                }
            }
            else if(rootNode->value->lastName == newNode->value->lastName){
                if(rootNode->value->firstName < newNode->value->firstName){
                    if(rootNode->rightChild == nullptr){
                        newNode->parent = rootNode;
                        rootNode->rightChild = newNode;
                    }
                    else{
                        treeInsert(newNode, rootNode->rightChild);
                    }
                }
                else if(rootNode->value->firstName > newNode->value->firstName){
                    if(rootNode->leftChild == nullptr){
                        newNode->parent = rootNode;
                        rootNode->leftChild = newNode;
                    }
                    else{
                        treeInsert(newNode, rootNode->leftChild);
                    }
                }
            }
        }

        void add(std::string firstName, std::string lastName, std::string number){
            Person* newPerson = new Person(firstName, lastName, number);
            Node* newNode = new Node(newPerson);
            treeInsert(newNode, treeHead);
        }

        Node* findNode(std::string firstName, std::string lastName, Node* rootNode = nullptr){
            if(rootNode == nullptr){
                rootNode = treeHead;
            }
            if(rootNode->value->lastName == lastName){
                if(rootNode->value->firstName == firstName){
                    return rootNode;
                }
                else if(rootNode->value->firstName < firstName) {
                return findNode(firstName,lastName,rootNode->rightChild);
                }
                else if(rootNode->value->firstName > firstName){
                    return findNode(firstName,lastName,rootNode->leftChild);
                }

            }
            else if(rootNode->value->lastName < lastName) {
                return findNode(firstName,lastName,rootNode->rightChild);
            }
            else if(rootNode->value->lastName > lastName){
                return findNode(firstName,lastName,rootNode->leftChild);
            }
            return nullptr;
        }

        void deleteNode(std::string firstName = "", std::string lastName = "", Node* nodeToDelete = nullptr){
```



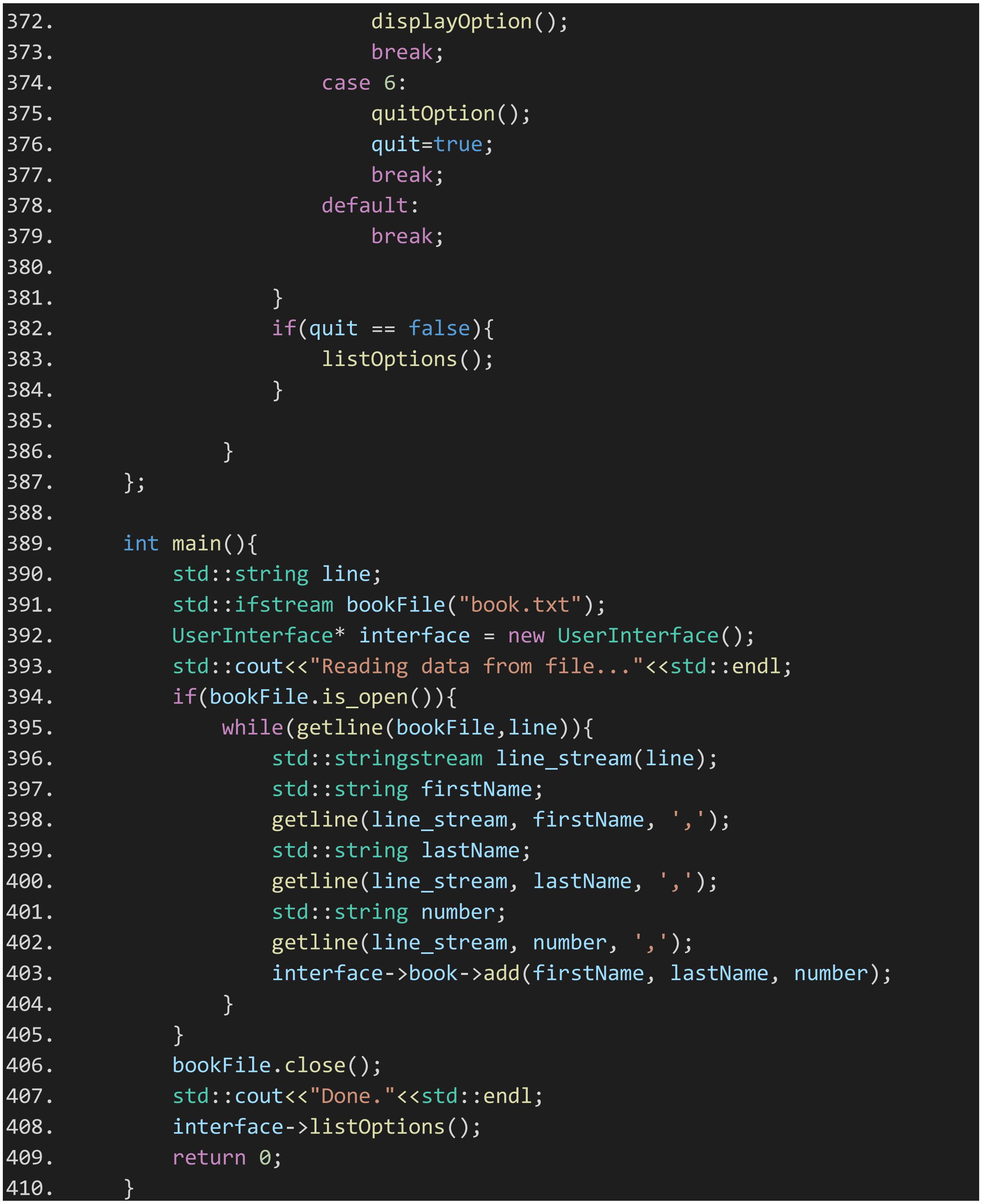
```cpp
                if(nodeToDelete == nullptr){
                    nodeToDelete = findNode(firstName,lastName);
                }

                if(nodeToDelete->leftChild == nullptr && nodeToDelete->rightChild == nullptr){
                    //Node is a leaf. Just delete it.
                    delete nodeToDelete;
                }
                else if(nodeToDelete->leftChild == nullptr && nodeToDelete->rightChild != nullptr){
                    //Node has a child on the right
                    if(nodeToDelete->parent != nullptr){
                        //If the node isn't the root
                        if(nodeToDelete->parent->leftChild == nodeToDelete){
                            nodeToDelete->parent->leftChild = nodeToDelete->rightChild;
                            delete nodeToDelete;
                        }
                        else if(nodeToDelete->parent->rightChild == nodeToDelete){
                            nodeToDelete->parent->rightChild = nodeToDelete->rightChild;
                            delete nodeToDelete;
                        }
                    }
                    else{
                        treeHead = nodeToDelete->rightChild;
                    }

                }
                else if(nodeToDelete->leftChild != nullptr && nodeToDelete->rightChild == nullptr){
                    //Node has a child on the left
                    if(nodeToDelete->parent != nullptr){
                        //If it isn't the root
                        if(nodeToDelete->parent->leftChild == nodeToDelete){
                            nodeToDelete->parent->leftChild = nodeToDelete->leftChild;
                            delete nodeToDelete;
                        }
                        else if(nodeToDelete->parent->rightChild == nodeToDelete){
```



```
                        nodeToDelete->parent->rightChild = nodeToDelete->leftChild;
                        delete nodeToDelete;
                    }
                }
                else{
                    treeHead = nodeToDelete->leftChild;
                }

            }
            else if(nodeToDelete->leftChild != nullptr && nodeToDelete->rightChild != nullptr){
                //Node has two children
                nodeToDelete->value = nodeToDelete->rightChild->value;
                deleteNode("","",nodeToDelete->rightChild);
            }
        }

        void changeNode(std::string firstName, std::string lastName, std::string changeTo){
            Node* nodeToChange = findNode(firstName, lastName, treeHead);
            nodeToChange->value->number = changeTo;
        }

        void displayTree(Node* rootNode = nullptr){
            if(rootNode == nullptr){
                rootNode = treeHead;
            }
            if(rootNode->leftChild != nullptr){
                displayTree(rootNode->leftChild);
            }
            Node rootNodeValue = *rootNode;
            Person personValue = *rootNodeValue.value;
            std::cout << "Name: " << personValue.firstName + " " + personValue.lastName << " Phone #: " << personValue.number << std::endl;
            if(rootNode->rightChild != nullptr){
                displayTree(rootNode->rightChild);
            }
        }

        std::vector<Node*> getNodeList(Node* rootNode = nullptr){
            std::vector<Node*> discoveredNodes;
            if(rootNode == nullptr){
                rootNode = treeHead;
            }
            discoveredNodes.push_back(rootNode);
            if(rootNode->leftChild != nullptr){
                std::vector<Node*> leftChildNodes = getNodeList(rootNode->leftChild);
                discoveredNodes.insert(discoveredNodes.end(),leftChildNodes.begin(),leftChildNodes.end());
            }
            if(rootNode->rightChild != nullptr){
                std::vector<Node*> rightChildNodes = getNodeList(rootNode->rightChild);
                discoveredNodes.insert(discoveredNodes.end(),rightChildNodes.begin(),rightChildNodes.end());
            }

            return discoveredNodes;
        }

};

class UserInterface{
    public:
        Book* book;

        UserInterface(){
            book = new Book();
        };

        UserInterface(const UserInterface &oldInterface){
            book = oldInterface.book;
        };

        ~UserInterface(){
            delete book;
        }

        void addOption(){
            std::cout<< "Enter first name: ";
            std::string firstName;
            std::cin>>firstName;
            std::cout<< "Enter last name: ";
            std::string lastName;
            std::cin>>lastName;
            std::cout<< "Enter phone #: ";
            std::string phoneNumber;
            std::cin>>phoneNumber;
            book->add(firstName, lastName, phoneNumber);
            std::cout<<"Done."<<std::endl;
        }

        void deleteOption(){
            std::cout<< "Enter first name: ";
            std::string firstName;
            std::cin>>firstName;
            std::cout<< "Enter last name: ";
            std::string lastName;
            std::cin>>lastName;
            book->deleteNode(firstName,lastName);
            std::cout<<"Done."<<std::endl;
        }

        void findOption(){
            std::cout<< "Enter first name: ";
            std::string firstName;
            std::cin>>firstName;
            std::cout<< "Enter last name: ";
            std::string lastName;
            std::cin>>lastName;
            Node* foundNode = book->findNode(firstName,lastName);
            std::cout<<"Phone number: "<<foundNode->value->number<<std::endl;
            std::cout<<"Done."<<std::endl;
        }

        void changeOption(){
            std::cout<< "Enter first name: ";
            std::string firstName;
            std::cin>>firstName;
            std::cout<< "Enter last name: ";
            std::string lastName;
            std::cin>>lastName;
            std::cout<< "Enter phone # to change to: ";
            std::string phoneNumber;
            std::cin>>phoneNumber;
            book->changeNode(firstName,lastName,phoneNumber);
            std::cout<<"Done."<<std::endl;
        }

        void displayOption(){
```

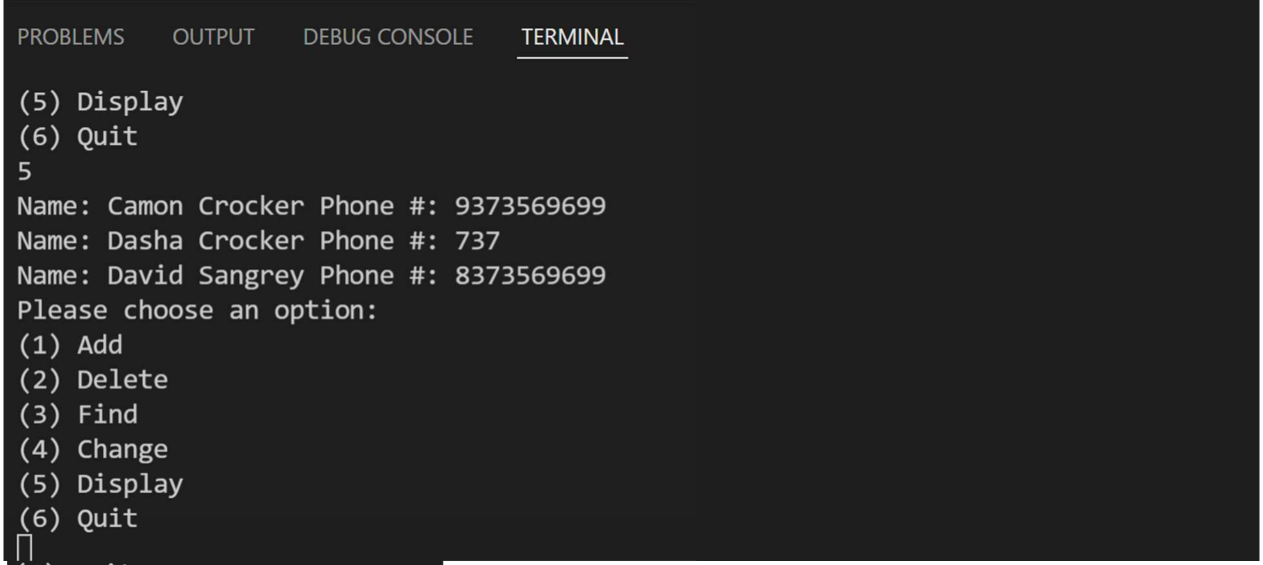


```cpp
        book->displayTree();
    }

    void quitOption(){
        std::cout<<"Writing data to file..."<<std::endl;
        std::ofstream bookFileClearer;
        bookFileClearer.open("book.txt", std::ofstream::out | std::ofstream::trunc);
        bookFileClearer.close();
        std::ofstream bookFile("book.txt");
        if(bookFile.is_open()){
            std::vector<Node*> nodes = book->getNodeList();
            for(int i=0; i<nodes.size(); i++){
                bookFile << nodes[i]->value->firstName << "," << nodes[i]->value->lastName << "," << nodes[i]->value->number << std::endl;
            }
        }
        std::cout<<"Done!"<<std::endl;
    }

    void listOptions(){
        std::cout<<"Please choose an option: " << std::endl <<
        "(1) Add" << std::endl <<
        "(2) Delete" << std::endl <<
        "(3) Find" << std::endl <<
        "(4) Change" << std::endl <<
        "(5) Display" << std::endl <<
        "(6) Quit" << std::endl;
        int choice;
        std::cin >> choice;
        bool quit=false;
        switch(choice) {
            case 1:
                addOption();
                break;
            case 2:
                deleteOption();
                break;
            case 3:
                findOption();
                break;
            case 4:
                changeOption();
                break;
            case 5:
```

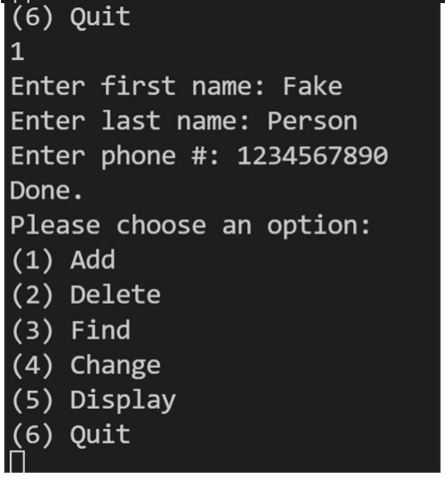


```
                    displayOption();
                    break;
                case 6:
                    quitOption();
                    quit=true;
                    break;
                default:
                    break;

            }
            if(quit == false){
                listOptions();
            }

        }
};

int main(){
    std::string line;
    std::ifstream bookFile("book.txt");
    UserInterface* interface = new UserInterface();
    std::cout<<"Reading data from file..."<<std::endl;
    if(bookFile.is_open()){
        while(getline(bookFile,line)){
            std::stringstream line_stream(line);
            std::string firstName;
            getline(line_stream, firstName, ',');
            std::string lastName;
            getline(line_stream, lastName, ',');
            std::string number;
            getline(line_stream, number, ',');
            interface->book->add(firstName, lastName, number);
        }
    }
    bookFile.close();
    std::cout<<"Done."<<std::endl;
    interface->listOptions();
    return 0;
}
```

```
Sample output of program
```



```
(5) Display
(6) Quit
5
Name: Camon Crocker Phone #: 9373569699
Name: Dasha Crocker Phone #: 737
Name: David Sangrey Phone #: 8373569699
Please choose an option:
(1) Add
(2) Delete
(3) Find
(4) Change
(5) Display
(6) Quit
```



```
(6) Quit
1
Enter first name: Fake
Enter last name: Person
Enter phone #: 1234567890
Done.
Please choose an option:
(1) Add
(2) Delete
(3) Find
(4) Change
(5) Display
(6) Quit
```