```cpp
// head file might use
// #include <vector>
// #include <string>
// #include <algorithm>
// #include <numeric>
// #include <iostream>
// #include <iterator>
// #include <sstream>
// #include <cassert>
// #include <algorithm>
// #include <numeric>
// #include <string>

//Alphabetical Ordered Lines

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <iterator>
#include <sstream>

int main() {
    std::string line;
    while (std::getline(std::cin, line)) {
        std::istringstream iss(line);
        std::vector<std::string> words;

        // Extract words from the line
        std::copy(std::istream_iterator<std::string>(iss),
                  std::istream_iterator<std::string>(),
                  std::back_inserter(words));

        // Create a sorted version of the words
        std::vector<std::string> sorted_words(words);
        std::sort(sorted_words.begin(), sorted_words.end(),
                  [](const std::string& a, const std::string& b) {
                      std::string lower_a = a, lower_b = b;
                      std::transform(lower_a.begin(), lower_a.end(), lower_a.begin(), ::tolower);
                      std::transform(lower_b.begin(), lower_b.end(), lower_b.begin(), ::tolower);
                      return lower_a < lower_b;
                  });

        // Check if the sorted version matches the original order
        if (std::equal(words.begin(), words.end(), sorted_words.begin(),
                       [](const std::string& a, const std::string& b) {
                           std::string lower_a = a, lower_b = b;
                           std::transform(lower_a.begin(), lower_a.end(), lower_a.begin(), ::tolower);
                           std::transform(lower_b.begin(), lower_b.end(), lower_b.begin(), ::tolower);
                           return lower_a == lower_b;
                       })) {
            // Output the line as is if the condition is satisfied
            std::cout << line << std::endl;
        }
    }
    return 0;
}

// stack

// stack.cpp
#include "Stack.hpp"

void Stack::push(char c) {
  Node* new_top = new Node(c);
  new_top->down = top_;
  top_ = new_top;
}
```

```cpp
char Stack::top() { return top_->letter; }
void Stack::pop() {
  Node* new_top = top_->down;
  delete top_;
  top_ = new_top;
}

// WRITE YOUR CODE HERE
int Stack::size(){
  int cnt = 0;
  Node* current = top_;
  while(current != nullptr){
    cnt ++;
    current = current-> down;
  }
  return cnt;
}

// stack.hpp
#pragma once
// This file can't be changed

class Node {
 public:
  char letter;
  Node* down;
  Node(char c) : letter(c), down(nullptr) {}
};

class Stack {
 private:
  Node* top_ = nullptr;

 public:
  Stack() = default;
  void push(char c);
  char top();
  void pop();
  int size(); // TODO
};

// main.cpp
#include "Stack.hpp"
#include <cassert>
int main() {
    Stack s;
    assert(s.size() == 0);
    s.push('a');
    assert(s.top() == 'a');
    assert(s.size() == 1);
    s.push('b');
    assert(s.top() == 'b');
    assert(s.size() == 2);
    s.pop();
    assert(s.top() == 'a');
    assert(s.size() == 1);
}



// comlex class problem

// complex.cpp
#include "Complex.hpp"
#include <iostream>

// Overload the << operator
std::ostream& operator<<(std::ostream& out, const Complex& c) {
    out << c.real();  // Output the real part
    if (c.imaginary() >= 0) {
        out << "+";   // Add '+' for positive imaginary part
    }
    out << c.imaginary() << "i";  // Output the imaginary part
    return out;
}

// Overload the * operator for complex multiplication
Complex Complex::operator*(const Complex& other) const {
    double real_part = (this->real() * other.real()) - (this->imaginary() * other.imaginary());
    double imaginary_part = (this->real() * other.imaginary()) + (this->imaginary() * other.real());
    return Complex(real_part, imaginary_part);
}


// complex.hpp
#pragma once

#include <iostream>

class Complex {
private:
    double re, im;

public:
    // Constructor
    Complex(double r, double i) : re{r}, im{i} {}

    // Getters
    double real() const { return re; }
    double imaginary() const { return im; }

    // Setters (optional, not needed for this problem)
    void real(double r) { re = r; }
    void imaginary(double i) { im = i; }

    // Operator overloads
    friend std::ostream& operator<<(std::ostream& out, const Complex& c);
    Complex operator*(const Complex& other) const;
};


// Reorder Words
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <sstream>

// Function to reorder names based on their parts
void Reorder(std::vector<std::string>& names) {
    std::vector<std::pair<std::string, std::string>> m;

    // Split names into pairs of parts before and after the dot
    for (const auto& name : names) {
        size_t pos = name.find('.');
        if (pos != std::string::npos) {
            m.emplace_back(name.substr(0, pos), name.substr(pos + 1));
        }
    }

    // Stable sort based on the second part, then the first part
    std::stable_sort(m.begin(), m.end(), [](const auto& a, const auto& b) {
        if (a.second == b.second) {
            return a.first < b.first;
        } else {
            return a.second < b.second;
        }
    });

    // Rebuild the names vector
    names.clear();
    for (const auto& pair : m) {
        names.push_back(pair.first + "." + pair.second);
    }
}

int main() {
    std::vector<std::string> words;
    std::string word;

    // Read words from standard input
    while (std::cin >> word) {
        words.push_back(word);
    }

    // Reorder the words
    Reorder(words);

    // Print the reordered words
    for (const std::string& output_word : words) {
        std::cout << output_word << " ";
    }

    return 0;
}
```