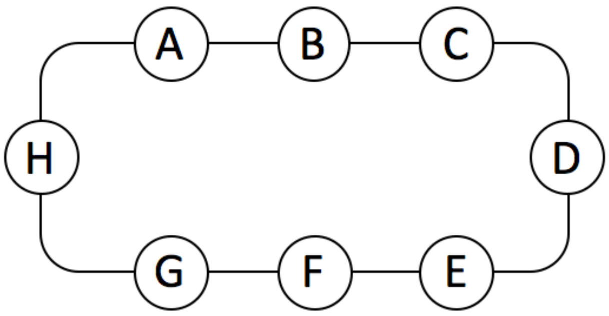
Story of My Love

A love story is often a heart-warming one. This is the story of a man who fell in love. We will keep his name a secret for this problem. This man lives in the ever-bustling Celadon city, the so-called city of love. He has a girl, a charming and beautiful girl, deep inside his heart. It is undeniable that he loves her very dearly from the bottom of his heart. He wants to make her happy and as long as she is happy, he is happy.

He wants to buy a gift for her. To buy the gift, he needs to figure out where to purchase the gift so that he can get the best gift available. He wants to visit many places and compare the possible gifts that he can buy for her. Therefore, he decides to travel by himself in Celadon's subway network. Before travelling, he has learned some interesting facts about the subway network. The subway network in Celadon city consists of only one line, and it is a complete loop. An example of the subway network is illustrated in the following diagram:



In the subway network, there are trains going both directions, i.e. one can go from station A to either B or H in one stop. The time to travel from one stop to the next one is 2 minutes. For example, if he wants to go from A to C, he needs to go to B from A first (which takes 2 minutes) and then from B to C (which takes another 2 minutes). However, each train must stop at all stations to accommodate alighting and boarding passengers. This takes 1 minute at each stop. Hence, the total time from A to C (by B) is 5 minutes. He can also go to C from A by taking the other way, i.e. A-H-G-F-E-D-C but it takes too much time.

He wants to know how much is the minimum time he needs to travel between any two given stations. Moreover, sometimes he wants to know all the stations that are in the shortest path between any two given stations. This is to help him plan his travels efficiently and meticulously.

At the same time, however, the government of Celadon is also building new stations for the already brilliant subway network. The government has this magical device which is able to surprise people by yielding tremendous results all the time. This device can modify the existing network almost instantly, i.e. it can build new stations and railroads that connect to it in a flash. For example, if the government soon learns that a new station is needed and wants to build station "I" between station "H" and "A" based on the above network map, this device will do it instantly. Now that station "I" has been built, you can verify that going from station H to station A now takes 5 minutes. Although the travel time between each station remains the same (i.e. 2 minutes between stations), now going to "A" from "H" requires you to go to station "I" first. (The train needs 2 minutes from station "H" to station "I" and 2 minutes from station "I" to station "A", for a total journey of 5 minutes, including the 1-minute stop).

He is now going to plan his journey. In order to do so, he needs a fast query-answering machine that can quickly tell him some information that he needs. Your job is to help him by building this simple query machine.

Input

The first line of the input consists of a single integer, **N (1 <= N <= 100)**, the number of initial stations in the Celadon subway network. The following line contains **N** station names, ordered in a clockwise fashion. It is guaranteed that each station name is unique, containing only small English letters ('a' - 'z') and not exceeding 50 characters each.

The next line has a single integer, **Q (1 <= Q <= 500)**, the number of queries you need to answer. It is then followed by **Q** lines, each containing a single query for you to answer. The queries follow the following specification:

Query Type Input Format: `<QUERY_TYPE> <APPROPRIATE_PARAMETERS>`

1. `TIME STARTING_STATION ENDING_STATION`
   Print the minimum travel time from STARTING_STATION to ENDING_STATION.

2. `BUILD PRECEEDING_STATION NEW_STATION`
   Build a new station right after the station PRECEEDING_STATION (clockwise direction). Print "**station NEW_STATION has been built**" after executing this query. It is guaranteed that the station PRECEEDING_STATION already exists and the station NEW_STATION does not exist when this query is called.

3. `PATH STARTING_STATION ENDING_STATION`
   Print, in a single line, all station names in the shortest path between STARTING_STATION and ENDING_STATION (both inclusive), separated by a single space. If there is more than one shortest path, print the one in a clockwise order. There is no space after the last station's name.

4. `PRINT STARTING_STATION`
   Print, in a single line, all station names, in a clockwise order, starting from the station STARTING_STATION, separated by a single space. There is no space after the last station's name.

It is guaranteed that all station names are valid, i.e. all are existing stations (except for the newly-built station for query 2 as explained above).

Output

Print the result of all queries, as described above. The last line of the output should contain a newline character.

Sample Input

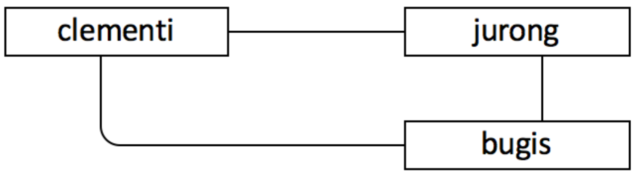
```
3
clementi
jurong
bugis
7
TIME clementi bugis
TIME clementi clementi
PATH clementi bugis
BUILD bugis tampines
TIME clementi Bugis
PATH clementi bugis
PRINT jurong
```

Sample Output

```
2
0
clementi bugis
station tampines has been built
5
clementi jurong bugis
jurong bugis tampines clementi
```

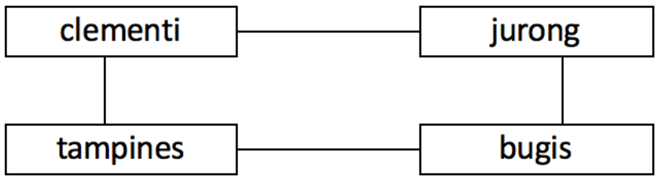
Explanation

The initial subway network is represented in the following diagram:



For each query:

1. It takes 2 minutes from `clementi` to `bugis` since it is only 1 station apart.
2. Travelling from `clementi` station to `clementi` station takes 0 minute.
3. You can go directly to `bugis` station from `clementi`, hence the path `clementi bugis`.
4. The new subway network is represented in the following diagram:



5. Now it takes 5 minutes to travel from `clementi` to `bugis` (see explanation below).
6. You can travel from `clementi` to `bugis` in 5 minutes going in either direction, but you need the one in clockwise order: `clementi jurong bugis`. It takes 2 minutes from `clementi` to `jurong` and 2 minutes from `jurong` to `bugis`, with 1-minute waiting time in `jurong`.
7. Refer to the above updated station map for details.

Skeleton

You are given the skeleton file `Subway.java`. You should see a non-empty file when you open the skeleton file. Otherwise, you might be in the wrong working directory.

Notes:

1. You should develop your program in the subdirectory `ex1` and use the skeleton java file provided. You should not create a new file or rename the file provided.
2. If your algorithm is different from the given skeleton, you are free to write a solution according to your own algorithm. However, **your algorithm must use linked list** to solve this problem. Solution that does not use linked list will receive 0 marks. You are allowed to use other Java's classes as helpers, but not as the main idea of the algorithm.
3. You are given a **fully-functional doubly-linked list** in the skeleton file. You are free to modify the provided implementation to suit your needs. Alternatively, you are allowed to use Java's API instead. Please take note to **remove the provided linked list (and list node) class** in case you want to use Java's linked list.
4. You are **free to define your own classes (or remove existing ones)** if it is suitable.
5. Please be reminded that the marking scheme is:

Input	: 10%
Output	: 10%
Correctness	: 50%
Programming Style	: 30% (awarded if you score **at least 20% from the above**):

   - Meaningful comments (pre- and post- conditions, comments inside the code): 10%
   - Modularity (modular programming, proper modifiers [public / private]): 10%
   - Proper Indentation: 5%
   - Meaningful Identifiers (for both method and variable names): 5%

Compilation Error : Deduction of **50% of the total marks obtained**.

Skeleton File – Subway.java

You should see the following contents when you open the skeleton file:

```java
/**
 * Name        :
 * Matric. No  :
 * PLab Acct.  :
 */

import java.util.*;

public class Subway {

    public Subway() {
        //constructor
    }

    public void run() {
        //implement your "main" method here
    }

    public static void main(String[] args) {
        Subway newSubwayNetwork = new Subway();
        newSubwayNetwork.run();
    }
}

class DoublyLinkedList<E> {

    //Data attributes
    private ListNode<E> head;
    private ListNode<E> tail;
    private int size;

    public DoublyLinkedList() {
        this.head = null;
        this.tail = null;
        this.size = 0;
    }

    // returns the size of the linked list
    public int size() {
        return this.size;
    }

    // returns true if the list is empty, false otherwise
    public boolean isEmpty() {
        return this.size == 0;
    }

    // adds the specified element to the beginning of the list
    public void addFirst(E element) {
        ListNode<E> newNode = new ListNode<E>(element);

        if (size == 0) {
            this.head = newNode;
            this.tail = this.head;
        } else {
            ListNode<E> oldHead = this.head;
            this.head = newNode;
            newNode.setNext(oldHead);
            oldHead.setPrev(newNode);
        }

        this.size++;
    }

    // retrieves the first element of the list
    public E getFirst() throws NoSuchElementException {
        if (head == null) {
            throw new NoSuchElementException("Cannot get from an empty list");
        } else {
            return head.getElement();
        }
    }
    // returns true if the list contains the element, false otherwise
    public boolean contains(E element) {
        for (ListNode<E> current = head; current != null; current = current.getNext()) {
            if (current.getElement().equals(element)) {
                return true;
            }
        }

        return false;
    }

    // removes the first element in the list
    public E removeFirst() throws NoSuchElementException {
        if (head == null) {
            throw new NoSuchElementException("Cannot remove from an empty list");
        } else {
            ListNode<E> currentHead = head;
            head = head.getNext();
            if (head == null) {
                tail = null;
            } else {
                head.setPrev(null);
            }
            this.size--;
            return currentHead.getElement();
        }
    }

    // Returns reference to first node.
    public ListNode<E> getHead() {
        return this.head;
    }

    // Returns reference to last node of list.
    public ListNode<E> getTail() {
        return this.tail;
    }

}

class ListNode<E> {
    private E element;
    private ListNode<E> next;
    private ListNode<E> prev;

    public ListNode(E newElement) {
        this.element = newElement;
        this.next = null;
        this.prev = null;
    }

    public void setElement(E newElement) {
        this.element = newElement;
    }

    public E getElement() {
        return this.element;
    }

    public void setPrev(ListNode<E> previous) {
        this.prev = previous;
    }

    public void setNext(ListNode<E> next) {
        this.next = next;
    }

    public ListNode<E> getNext() {
        return next;
    }

    public ListNode<E> getPrev() {
        return prev;
    }
}
```