CS302

Assignment 7: Document Analyzer

Description

In this project you are assigned a task; of course a critical task. The task is that you are given a document consisting of lowercase letters, numbers and punctuations. You have to analyze the document and separate the words first. Words are consecutive sequences of lower case letters. After listing the words, in the order same as they occurred in the document, you have to number them from $1, 2, ..., 3$. After that you have to find the range p and q ($p \leq q$) in the list such that all of words occur between p and q (inclusive). If there are multiple such solutions you have to find the one where the difference of p and q is smallest. If still there is a tie, then find the solution where p is smallest.

Input

First line of input will contain T (≤ 20) denoting number of test cases. Each test case will be denoted by one or more lines, each line having no more than 150 characters. A test case will contain either lowercase letters or numbers or punctuations. The last line of a document will contain the word 'END' which is of course not the part of the document. You can assume that a document will contain between 1 and 105 words (inclusive).

Output

For each test case, print the test case number first. After that, print p and q as described above. See sample output for formatting.

Hash Map Structure

There is a simple brute force solution to this problem but we would like to have code to solves this problem quickly, thus we will need to write a custom hash map class.

```
template <class type1, class type2>
class hashMap
{
public:
  const int MAX_ELEMENTS;
  hashMap(int = 10);
  hashMap(const hashMap<type1, type2>&);
  const hashMap<type1, type2>& operator=(const hashMap<type1, type2>&);
  ~hashMap();
  type2& operator[](std::string);
```

```
private:
  int hashFunction(std::string) const;
  void resize(int);

  struct node
  {
    type1 key;
    type2 value;
    node * next;
    node * prev;
  };

  struct list
  {
    node * head;
    node * middleElement;
    int amount;
  };

  int filledEntries;
  int tableSize;

  list * table;
};
```

Each member will contain/perform the following

- `struct node` - contains the pair (key, value) for each entry, along with the left and right neighbor pointer (since we're implementing a doubly linked list)
- `struct list` - a linked list struct, `node * head` points to the first node in the linked list, `node * middleElement` points to middle node in the list, and `int amount` denotes the amount of node in the linked list (you want this list to be sorted in ascending order by keys)
- `list * table` - an array of linked lists that contains all the (key, item) for all entries in the `hashMap`
- `int filledEntries` - denotes how many linked lists in `table` are full, an element in `table[i]` is full if `table[i].amount == MAX_ELEMENTS`
- `const int MAX_ELEMENTS` - a constant that is set to 10 in either constructor
- `int tableSize` - denotes the size of the `table` array
- `hashMap<type1, type2>::hashMap(int init) : MAX_ELEMENTS(10)` - constructor that sets `MAX_ELEMENTS` to be 10 (as seen in the heading), sets `tableSize` with the value of `init`, sets `filledEntries` with 0, allocates an array of size `tableSize` to the `table` pointer (sets each field in `table[i]` with `NULL` or 0)
- `hashMap<type1, type2>::hashMap(const hashMap<type1, type2>& copy) : MAX_ELEMENTS(10)` - copy constructor, sets `MAX_ELEMENTS` to 10 (as seen in the heading), deep copies the `copy` object into the `*this` object
- `const hashMap<type1, type2>& hashMap<type1, type2>::operator=(const hashMap<type1, type2>& rhs)` - assignment operator, deep copies `rhs` object into `*this` object, make sure you deallocate the `*this` object first and check for self assignment as always
- `hashMap<type1, type2>::~hashMap()` - destructor

- `int hashMap<type1, type2>::hashFunction(std::string key) const` - hash function, takes in the key of type `string` (pretty much gives away the key type that will be used in main when declaring the `hashMap` object), this function adds up all the ASCII values for each character in the `string` parameter passed in, then mods this sum by the `tableSIze` and returns that value
- `type2& hashMap<type1, type2>::operator[](std::string key)` - finds the `key` (the parameter) in the hash structure or creates a new entry and returns the value field of the node with a matching key. Each `table[i].head` will point to a doubly linked lists sorted by keys, and `table[i].midleElement` will point to the node in the list that is roughly the middle element (this is used to potentially speed up the search time). If `table[i].amount == MAX_ELEMENTS` and the key you're trying to find is not in this linked list, you need to linear probe to the next element in table `table[i + collisionCounter]` and search in that linked list. Here is a rough sketch of the algorithm
  1. If the load factor is 50% or larger, double the size of the table, the load factor would be `filledEntries / tableSize`
  2. Use the hash function using the key passed into the function and assign the result to a variable say `index`
  3. If `table[index].head` is `NULL` then you just simply add a node to this linked list, set `table[index].middleElement` to this node as well, set this node's key field with the key passed into this function, set the value field to `type2()` (a default value), increment `table[index].amount` by one and then return `table[index].head->value`
  4. If `table[index].head` is not empty then you will do the following
     (a) Create a pointer call it say `it` and assign it to `table[index].middleElement`
     (b) Compare `it->key` with `key` (the parameter passed into this function), and move `it` to the right or the left depending whether `key` is larger or smaller than `it->key`
     (c) As you traverse the linked list (left direction or right direction), if the following occurs
     - A matching key is found return that node's value field
     - If you can determine the key is not in the list and `table[index].amount` is less than `MAX_ELEMENTS` then insert a new node before or after `it`, set this newly created node's key to the key passed into the function, set its value to `type2()`, increment `table[index].amount` by 1 (and potentially updated `filledEntries` by 1 if `table[index].amount` equals `MAX_AMOUNT` after inserting this node), potentially move `table[index].middleElement` to the left or right, and then return this newly created node's value field
     - If you determine the key is not in the list but the linked list in `table[index]` contains `MAX_ELEMENTS`, then updated `index` to `(index + 1) % tableSize` to move to the next index in table (this is sort of a linear probe) and go to step 3
- `void hashMap<type1, type2>::resize(int amt)` - resizes the hash table by `amt`, you will need to rehash all the nodes from the smaller table to the newer table

Contents of main

After the filestream is ready to read input. You read in the amount of test cases and for each test case you output the p and q values that contains all the words in the test case in that range. You need to extract all the words from each test case and insert into a vector (this process does not need a hash, and it's ok to have duplicate words, in fact you will most likely have duplicate words), to find the range that contains all the words would need a `hashMap` however...

You probably can think of a brute force way that would probably take $O(n^3)$ time, but we want to use the hash map to get a much faster runtime. The key for the hash map will be a string, and the value...well I'll let

you think of it. Of course for testing your algorithm, you can use an `unordered_map` but when submitting you need to use `hashMap`.

Specifications

- Comment your code and your functions
- Do not add extra class members or remove class members and do not modify the member functions of the class
- Use `hashMap` for your searching, do not use linear or binary search or any other algorithm/data structure other than the `hashMap` structure
- No global variables (global constants are ok)
- Make sure your program is memory leak free

Sample Run

```
$ g++ main.cpp
$ ./a.out

You know the drill: words.txt

Test case 1: 7 10
Test case 2: 1 5
Test case 3: 5 6

Before this program ends, just remember...
Words can in fact hurt >:(
```

Submission

Compressed your source files and upload to class website by deadline

References

- Link to the top image can be found at $https://iconarchive.com/show/papirus-places-icons-by-papirus-team/folder-green-documents-icon.html$

Supplemental Videos

- Video 1 $https://youtu.be/IzQI9pGMIc0$
- Video 2 $https://youtu.be/-0Xb41_E3iE$
- Video 3 $https://youtu.be/GhRvd-_S Ldw$