11. ADTs and Algorithms II

11.1 MultiValue Maps

Many data types concern a single value (e.g. a hash table), so that a string (say) acts as both the key (by which we search for the data) and also as the object we need to store (the value). An example of this a spelling checker, where one word is stored (and searched for) at a time. However, sometimes there is a need to store a value based on a particular key - for instance an associative array in Python allows you to perform operations such as :

```
population["Bristol"] = 536000
```

where a value (the number 536000) is stored using the key (the string "Bristol"). One decision you need to make when designing such a data type is whether multiple values are allowed for the same key; in the above example this would make no sense - Bristol can only have one population size. But if you wanted to store people as the key, with their salary as the value, you might need to use a MultiValue Map (MVM) since people can have more than one job.

Here we write the abstract type for a MultiValueMap that stores key-value pairs, where both the key and the value are strings.

Exercise 11.1 The definition of an MVM ADT is given in `mvm.h`, and a file to test it is given in `testmvm.c`. Write `mvm.c`, so that:

```
% make -f mvm_adt.mk
./testmvm
Basic MVM Tests ... Start
Basic MVM Tests ... Stop
```

works correctly. Use a simple linked list for this, inserting new items at the head of the list. Make no changes to any of my files.

Submit : `mvm.c`. ■

11.2 Rhymes

In the usual place is a dictionary which, for every word, lists the phonemes (a code for a distinct unit of sound) used to pronounce that word in American English. In this file the word itself, and its phonemes, are separated by a '#'). For instance:

```
BOY#B OY1
```

shows that the word `BOY` has two phonemes : `B` and `OY1`.

A simple attempt at finding rhymes for boy would match every word that has `OY1` as its final phoneme. This gives you:

```
POLLOI MCVOY LAFOY ALROY ILLINOIS CROIX DECOY REDEPLOY CLOY
LAVOY MOYE LOYE STOY PLOY KNOY EMPLOY ELROY JOY COY LACROIX
DEVROY ENJOY LOY COYE FOYE MOY DOI BROY TOY LABOY ROI HOY
ROYE NEU CROY SOY YOY MCCOY CHOY GOY ROY BOLSHOI MALLOY JOYE
DESTROY DELACROIX(1) DEBOY MCROY CHOI UNDEREMPLOY FLOY MCKOY
TOYE AHOY BOY OYE SGROI FOIE(1) TROY DEPLOY SAVOY UNEMPLOY
SCHEU WOY BOYE HOYE FOY OI HOI KROY EMPLOY(1) FLOURNOY OIE
MCCLOY ANNOY OY DEJOY
```

Using two phonemes to do the matching is too many, since the only matches are for words that have exactly the same pronunciation:

```
LABOY DEBOY BOY BOYE
```

which are not really rhymes, but homophones. Therefore, using the **correct** number of phonemes will be key to finding 'good' rhyming words.

Here we will use the MutliValue Map written in Exercise 11.1 to create two maps. An MVM `map1` stores the word (as the key) and its final n phonemes as a single string (the value). Now `map2` stores the word (value), keyed by its final n phonemes as a single string. Looking up the phonemes associated with a word can be done using the word as a key [1] (via `map1`), and looking up a word given its phonemes can be achieved using `map2`.

Exercise 11.2 Read in the dictionary, and for each line in turn, store the data in the two maps. Now for a requested word to be 'rhymed', search for its phonemes using `map1` and then search for matching words using `map2`. The number of phonemes specified for this rhyming is given via the command line, as are the words to be rhymed:

```
$ ./homophones -n 3 RHYME
RHYME (R AY1 M): PRIME RHYME ANTICRIME(1) CRIME ANTICRIME
GRIME RIME
```

The `-n` flag specifies the number of phonemes to use (you may assume the number associated with it always is always separated from the flag by a space). If no `-n` flag is given then the value 3 is assumed. It only makes sense to use a value of $n \leq$ the number of phonemes in the two words being checked. If you use a value greater than this, the results are undefined.

The list of words to be matched may be found on the command line:

```
$ ./homophones -n 4 CHRISTMAS PROGRAM PASSING
CHRISTMAS (S M AH0 S): ISTHMUS CHRISTMAS CHRISTMAS'
PROGRAM (G R AE2 M): CENTIGRAM ENGRAM HISTOGRAM WOLFGRAM
MONOGRAM LOGOGRAM HOLOGRAM MICROGRAM SONOGRAM ANGIOGRAM
TELEGRAM PROGRAMME ELECTROCARDIOGRAM ELECTROPHORETOGRAM
REPROGRAM MILLIGRAM ANAGRAM PEGRAM POLYGRAM DIAGRAM
EPIGRAM PROGRAM MAILGRAM MILGRAM INGHRAM CABLEGRAM
MAMMOGRAM KILOGRAM
PASSING (AE1 S IH0 NG): GASSING MASENG SURPASSING KASSING
```

[1] Strictly speaking, we don't need the *map*1 to be capable of storing multiple values, since every word in the dictionary is unique. We'll use it here though for simplicity.

```
HASSING PASSING AMASSING MASSING CLASSING HARASSING
```

Use the Makefile supplied for this task. Make the output as similar to that shown above as possible.

Submit : `homophones.c`. ■

11.3 Faster MVMs

The ADT for MVMs used in Exercise 11.1 is a simple linked list - insertion is fast, but searching is slow.

Exercise 11.3 Write a new version of this MVM ADT called `fmvm.c` that implements exactly the same functionality but has a faster search time. The file `fmvm.h` will change very little from `mvm.h`, with maybe only the structures changing, but not the function prototypes. A similar testing file to that used previously, now called `testfmvm.c` should also be written. Note that any ordering of data when using the `mvm_print` and `mvm_multisearch` functions is acceptable, so these can't be tested in exactly the same manner.

By simply changing the `#include` from `<mvm.h>` to `<fmvm.h>` in your `homephones.c` file from Exercise 11.2, and compiling it against `fmvm.c`, I can test that program works identically.

Make it clear what you have done to speed up your searching (and how) using comments at the top of `fmvm.h`

Submit : `fmvm.c`, `fmvm.h` and `testfmvm.c` ■