Imperial College of Science, Technology and Medicine	Department of Computing
Computing Science (CS) / Software Engineering (SE)	BEng and MEng
Examinations Part I	C Programming Course

Programming work is a continuously assessed part of the examinations

and is a required part of the degree assessment.

Programming work must be handed in for marking by the due date.

Late submissions may not be marked.

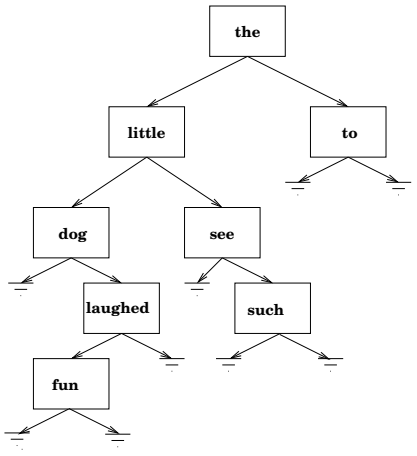
Exercise: 1	Working: Individual
Title: Dictionary in C	
Issue date: 9th June 2009	Due date: 17th June 2009
System: Linux	Language: C

Aim

- To gain introductory experience in programming in C using structures, pointers, and library functions, by implementing the *access procedures* for an Abstract Data Type - a *dictionary* which stores words in alphabetical order.

The Problem

- A dictionary is an *ordered collection of words* with no duplicates. (In real life we might like to also store a *meaning* for each word, but for the purposes of this exercise we will just store the words.) There are many possible representations that can be used to implement a dictionary, among which are a *list*, an *array* or a *binary tree*. In this exercise you will implement a dictionary as a *binary tree* of words in C. An example binary tree is shown in the diagram.



Every node of the tree should contain a word and two pointers to the two (left and right) subtrees of that node. For example, in Haskell a tree like this would be defined by:

```
type Word = [ Char ]
data Dict = Empty | Node Dict Word Dict
```

The equivalent C type declarations are:

```
#define MAXWORDSIZE 100
typedef char word[MAXWORDSIZE+1];  /* extra 1 to make room for null char */
typedef struct dictionary *dict;

/* the structure used to construct the dictionary */
struct dictionary {
      dict       left, right;
      word       theword;
 };
```

- Given the dictionary type (dict), you are required to implement the *access procedures* below (reproduced from the C header file `dict.h`):

```
dict addword(char aword[], dict adict);
/*
 * pre:  adict is a valid dictionary
 * post: the result contains all the words of adict and   'aword' is added
 *       in the correct (alphabetic) place
 *
 * note: it is wrong to write word aword in the first argument
 *       because it would include the size of the array, which the compile
 *       ignores
 */


void printdict(dict adict);
/*
 * pre:  'adict' is a valid dictionary
 * post: the words in 'adict' are written to standard output one per line in
 *       alphabetic order
 */


int lookup(char aword[], dict adict);
/*
 * pre:  adict is a valid dictionary
 * post: the value returned is non-zero if 'aword' appears in adict, 0 otherwise
 */


int countwords(dict adict);
/*
 * pre:  adict is a valid dictionary
 * post: the result is the number of words in adict
 */


int findword(dict adict, int n, char aword[]);
```

```
/*
 * pre:  adict is a valid dictionary
 * post: if  1 <= n <= countwords(adict) the nth word in alphabetic
 *       order is copied into aword and a non-zero value is returned.
 *       Otherwise aword is not changed and 0 is returned
 */


dict deleteword(char aword[], dict adict);
/*
 * pre:  adict is a valid dictionary
 * post: the result contains all the words in adict except aword and the
 *       storage used by aword is reclaimed
 */


dict deletedict(dict adict);
/*
 * pre:  adict is a valid dictionary
 * post: the result is empty; all storage used by adict is reclaimed
 */
```

- A valid dictionary `dict` (`struct dictionary *`) either is **NULL** or the `struct dictionary` it points to has been allocated by `malloc`, `left` and `right` are valid dictionaries, and `theword` contains a non-empty string.

  `malloc` returns a pointer to area of memory of the size given (in bytes) as the arguement, for example:

  ```
  dict adict = malloc(sizeof(struct dictionary))
  ```

  The `sizeof` function returns the size of a type or a variable in bytes. The memory that is returned is not initialised, and it is important to initialize it before using it.

  Like most C library functions `malloc` does not throw an exception if the request fails, but returns **NULL**, and it is important to test that the result is not **NULL** before using it, for example by using

  ```
  assert(adict!=NULL);
  ```

- Remember when a C function has an array argument, the size of the array is not part of the declaration. In this exercise you can assume any character array passed to one of the functions above will be a word. However, you must ensure that your program does not crash if the user tries to add a word which is too long. (No particular behaviour is required other than that it does not crash - but probably the simplest approach is to add the word in *truncated* form, *i.e.* an initial prefix of it that fits into the array space.)

Submit by Wednesday 17th June 2009

What To Do

- Copy the main program test harness **maindict.c**, the header file **dict.h**, and the skeleton implementation file **dict.c** into your current directory using the command: **exercise c1** (or from the CATE system).

maindict.c is the main program that provides a test harness for the dictionary.

dict.h contains the dictionary type and the prototypes (headers) and comments for the functions you will implement.

dict.c contains stubs for the functions you will implement.

- You can run a working lab version of the main program test harness by typing **labmaindict** at your Linux prompt. Try this out and familiarise yourself with its simple command menu.
- To create a compiled, linked, executable object code file **maindict** you can use the following GNU C Compiler command:

```
gcc -Wall -o maindict maindict.c dict.c
```

  at your Linux prompt. To execute it, type:

```
./maindict
```

- **maindict** is an interactive program; it prints a list of its commands and calls the functions listed above. Use it to test your functions as you implement them. Note that the supplied stubs return "minimal" results, so if you choose a menu item you have not yet implemented you will get the corresponding minimal result (such as an empty dictionary, or that a word could not be found, etc).
- Write the required functions one by one, testing them thoroughly as you go along. Your testing will be easier if you get `addword` and `printdict` working first, and then use them in testing the other functions. Remember to deal with an over-long word appropriately when adding it to the dictionary, for example by truncating it as suggested earlier.
- You can find out how to use the string compare function (**strcmp**) and string copy functions (**strcpy**, **strncpy**) by using the manual page display command **man** at your Linux prompt (**man strcmp**, **man strcpy**, **man strncpy**).
- Feel free to write any auxiliary functions you find helpful. Note that all your code, including any auxiliaries, should be in **dict.c**; you should not modify the supplied main program **maindict.c** or header file **dict.h**.

Submission

- Submit **dict.c** in the usual way using the **CATE** system.

Assessment

`addword`	3.0
`printdict`	1.0
`lookup, countwords, findword`	3.0
`deleteword, deletedict`	3.0
`Design, style, readability`	10.0
`Total`	20.0