Assignment 2: Dictionary Database

The deadline for the second assignment is **Tuesday 13th of may at 4pm**. The aim of this assignment is to implement a program to maintain a library database, with each book represented by its title, author and year of publication. This is a fairly large program, and you will only have to implement part of it. You will be provided with the source code for some basic functions, the structure you can use to represent the information for each book, and also the code for the top-level keyboard interface to the program. Your task will be to build the lower level functions that build and manipulate the database.

You will represent the library database as either an array or a tree of structures, one structure per book, each containing the relevant title, author name and year of publication for the book. The `struct Book` structure you will be provided with contains this information, as follows:

```
#define MAX_TITLE_LENGTH  100
#define MAX_AUTHOR_LENGTH 100

/* Book structure
 */
struct Book
{
   /* Book details */
   char title[MAX_TITLE_LENGTH+1];   /* title string */
   char author[MAX_AUTHOR_LENGTH+1]; /* author name string */
   int  year;                        /* year of publication */

   /* pointers to left and right branches pointing down to next level in
      the binary tree (for if you use a binary tree instead of an array) */
   struct Book *prev, *next;
};
```

So the book title is represented by a string of maximum length `MAX_TITLE_LENGTH` (+1 to include the string termination character), the author name as another string of maximum length `MAX_AUTHOR_LENGTH`, and the year of publication as an integer. If either the title or author name is longer than the specified maximum length, it should be truncated to the maxmimum length.

Representing a Database with an Array

In your program you will manipulate `struct Book` structures to build and modify the database. The simplest way to represent several structures at once is as an array of structures. Let us set the maximum number of books in our database to a defined constant `MAX_BOOKS`:

```
/* maximum number of books that can be stored at once in the library
   database (relevant only to storage using an array) */
#define MAX_BOOKS 200
```

Then we can define an array of `MAX_BOOKS` structures, and use an integer to represent the number of books currently represented in the program, which of course is initialized to zero:

```
/* array of books */
static struct Book book_array[MAX_BOOKS];
```

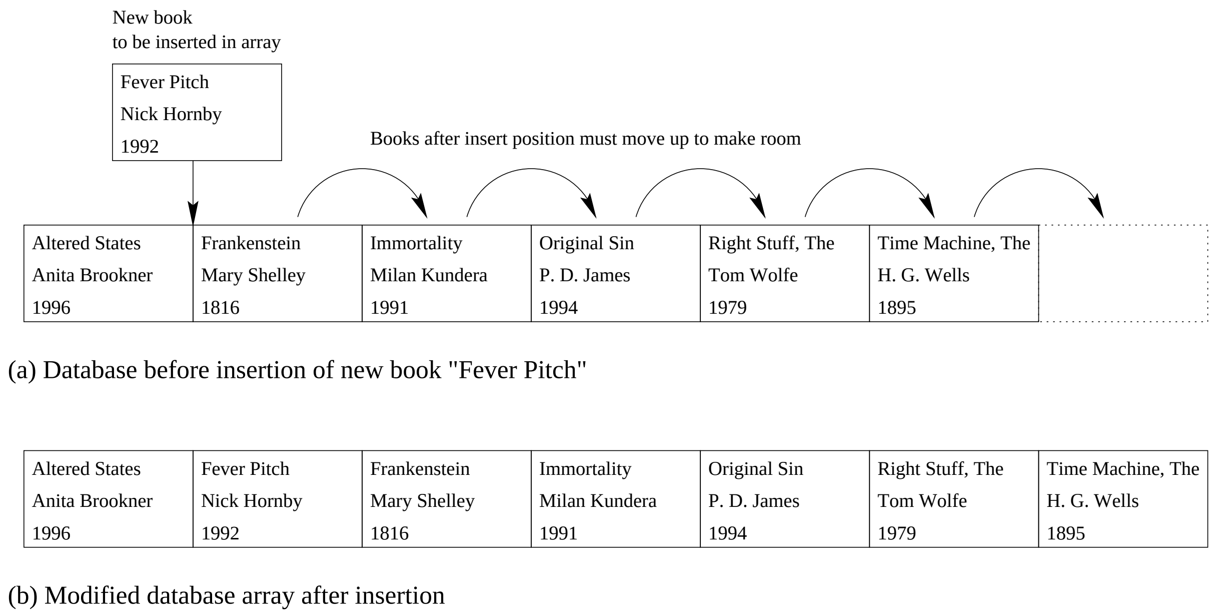

Figure 1: Illustration of the inefficiency of the array representation of an ordered database. Here the database of books is ordered alphabetically by title name, and a new book "Fever Pitch" is to be inserted in the existing array (a) of five books. To insert the new book, all the books with titles alphabetically "greater" than "Fever Pitch" have to shift up one place in the array to make room for it.

```
/* number of books stored */
static int no_books = 0;
```

This array/integer representation may be used to represent our database. However there are some problems with using an array to represent a database:

- An array has a fixed size. The array `book_array` of `struct Book` structures declared above can obviously not hold details for more than `MAX_BOOKS` books. This is only OK if you know beforehand the maximum number of books you are likely ever to have, which is not likely.
- An array is inefficient in storing the information, again because of its fixed size. If you guess the maximum number `MAX_BOOKS` of books, and declare an array of that size, but in fact the program actually only creates a much smaller database, most of the memory block used for the array is wasted.
- An array is inefficient in implementing important operations, especially if the elements of the array are maintained in a specific order. Let us assume that our array is stored in alphabetical order of title name, and we wish to insert a new book "Fever Pitch" into the array shown in figure 1. As you can see, insertion into an ordered array is a very inefficient operation, because every array element (in this case `struct Book` structure) beyond the insertion point must shift up one place to make room. This involves lots of copying data. The same applies to deleting a book, when all the book structures beyond the deletion point would have to shift back one place.

The *binary tree* data structure gets around all these problems. It is a *dynamic* data structure, which expands and contracts as the database size goes up and down. Implementing binary trees involves the use of an advanced C feature called *dynamic memory allocation.* We shall first introduce the concept of dynamic memory allocation and provide a simple example, before discussing how we can use it to efficiently build a binary tree. Once you are comfortable with the concepts and examples, it will be easier to write the programs for this assignment using binary trees than using arrays. However if you really do not want to use binary trees, you may if you like use the array representation, but in this case you will be marked out of 90% for this assignment (i.e. you will lose 10% of your marks, for this assignment only). An alternative would be to convert to the binary tree representation when you have successfully implemented the array representation. If you do not want to know about binary trees right now, you may turn immediately to part 1 of the assignment description on page 6.

Dynamic memory allocation

We must first introduce the mechanism provided in C to create temporary blocks of memory within a program. The two most important functions that implement dynamic memory allocation in C are `malloc()` and `free()`. `malloc()` is used to allocate a block of unused computer memory which can then subsequently be used in the program. It is declared as:–

```
void *malloc(size_t size);
```

The `size` argument specifies how many bytes you want in your block of memory. The memory block is returned as a generic pointer `(void *)` type, and can be cast (converted) to whatever C pointer type you wish. When the program has finished with the memory block, the `free()` function is used to discard it so that it may be used again:-

```
void free(void *ptr);
```

The `void *` memory block returned by `malloc()` should be passed to `free()` when it is not needed any more. Here is a simple program using dynamic memory allocation.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main ( int argc, char *argv[] )
{
   char *string;
   int length = 200;

   string = (char *) malloc(length);
   strcpy ( string, "This is a string that fits easily into 200 bytes" );
   printf ( "String: %s\n", string );
   free ( string );
   return 0;
}
```

Firstly the character pointer `string` in declared. As you should know by now, a pointer can be used to point to a single object, or it may specify the start of an array. In this case we dynamically create an array of characters by calling `malloc()`, and set `string` to point to the start of it. `string` can then be used exactly as if it had been declared as an array of characters, and we show an example, copying another string into it and printing the copied string. The code for the alternative array version of the program would be

```
#include <string.h>
#include <stdio.h>

int main ( int argc, char *argv[] )
```

```
{
   char string[200];

   strcpy ( string, "This is a string that fits easily into 200 bytes" );
   printf ( "String: %s\n", string );
   return 0;
}
```

Comparing the two versions of the program, we note that

1. The version using dynamic memory allocation version is more flexible, in that the size of the string allocated need not be a constant (200 in this case) but can be any integer value, for instance (as here) the value of an integer variable. The size of the allocated block of memory is determined when the program is run. In the second version, the array size is fixed.
2. On the other hand, the array version is shorter and simpler. If `malloc()` is called, you must remember to add a `free()` statement at the point where the program has finished using the memory block, because otherwise the memory will be wasted. The memory taken up by arrays (allocated from the "stack") is returned automatically when the program block in which it was declared terminates, in this simple case the end of the program. Predefined arrays are in this sense easier to use than dynamically allocated arrays.

Another use of dynamic memory allocation is to create an instance of a structure. So for instance the program segment

```
{
   struct Book *new;

   new = (struct Book *) malloc ( sizeof(struct Book) );
   strcpy ( new->title, "Titus Groan" );
   strcpy ( new->author, "Mervyn Peake" );
   new->year = 1946;
   free ( new );
}
```

creates a memory block the right size to hold a `struct Book` structure, sets `new` to point to it, and then fills the fields of the structure with details of a book. The call to `free()` discards the memory block so that it can be used again. Note the `->` operator, which allows you to access fields of a structure via a pointer to it. In fact `new->author` is equivalent to `(*new).author`, but a little simpler and more concise.

Without dynamic memory allocation one could write this program segment as

```
{
   struct Book book, *new;

   new = &book;
   strcpy ( new->title, "Titus Groan" );
   strcpy ( new->author, "Mervyn Peake" );
   new->year = 1946;
}
```

Again this program segment is exactly equivalent. The `book` structure is here allocated from the "stack", and returned to the stack automatically at the end of the program segment.

NOTE: To use `malloc()` and `free()` in your program you must first `#include` the header file `stdlib.h`.

Binary Trees

So far the only benefit we have mentioned of dynamic memory allocation is the extra flexibility of being able to create blocks of memory with size specified as the program runs rather than being fixed beforehand. The other major benefit becomes clear if we extend the above program segment as follows:–

```
static struct Book *book_tree = NULL;

{
   struct Book *new;

   /* create the first book */
   new = (struct Book *) malloc ( sizeof(struct Book) );
   strcpy ( new->title, "Something Happened" );
   strcpy ( new->author, "Joseph Heller" );
   new->year = 1973;

   /* add first book to binary tree */
   new->left = new->right = NULL;
   book_tree = new;

   /* create the second book */
   new = (struct Book *) malloc ( sizeof(struct Book) );
   strcpy ( new->title, "House of Mirth, The" );
   strcpy ( new->author, "Edith Wharton" );
   new->year = 1905;

   /* add second book to binary tree, to left of "Something Happened" */
   new->left = new->right = NULL;
   book_tree->left = new;

   /* create the third book */
   new = (struct Book *) malloc ( sizeof(struct Book) );
   strcpy ( new->title, "Suitable Boy, A" );
   strcpy ( new->author, "Vikram Seth" );
   new->year = 1993;

   /* add third book to binary tree, to right of "Something Happened" */
   new->left = new->right = NULL;
   book_tree->right = new;

   /* create the fourth book */
   new = (struct Book *) malloc ( sizeof(struct Book) );
   strcpy ( new->title, "Plague, The" );
   strcpy ( new->author, "Albert Camus" );
   new->year = 1946;

   /* add fourth book to binary tree, to right of "House of Mirth, The" */
   new->left = new->right = NULL;
   book_tree->left->right = new;

   /* create the fifth book */
```

```
        new = (struct Book *) malloc ( sizeof(struct Book) );
        strcpy ( new->title, "Handel's Operas" );
        strcpy ( new->author, "W. Dean and J. M. Knapp" );
        new->year = 1987;

        /* add fifth book to binary tree, to left of "House of Mirth, The" */
        new->left = new->right = NULL;
        book_tree->left->left = new;
    }
```

Here we have created five books, and built a simple *binary tree* `book_tree` to hold them, using the `left` and `right` fields of the `struct Book` structure to hold the links between the book structures. The binary tree generated by this simple example is illustrated in figure 2. Note that `free()` is not called. This means that the book data stored in the binary tree is maintained outside the program segment in which it is created, which is not possible with locally declared arrays.

Note the manner in which books are inserted in the tree. Given a new book, we firstly check whether the new book title is alphabetically greater than or less than that of the top book in the tree. If it is greater, we follow the `right` branch, if less the `left` branch. We repeat this process with the book we find along the given branch, and continue down the tree, until we either find a book with the same title as the new book, or a book `left` or `right` pointer to `NULL`. In the latter case we create a structure for the new book and replace the `NULL` pointer with a pointer to the new book structure.

If a book is to be deleted from the database, for instance "A Suitable Boy" in the above database, one can achieve this with the following code:

```
    /* free the memory used for the book structure */
    free ( (Book *) book_tree->right );

    /* resets the pointer from the top-most node of the tree to NULL */
    book_tree->right = NULL;
```

So `free()` is applied to the book structures as they become redundant. In ge though, deleting books which point to other books lower in the tree involves a tricky algorithm which you will not have to implement. Instead there is a simpler alternative method, described in the assignment description below.

This binary tree method of holding book information gets around the previously mentioned problems with arrays, because

1. The tree can grow and shrink as required. It is only limited by the computer's memory.
2. It is usually more efficient in terms of using memory, because only the data required at any time is stored. This usually compensates for the extra pointer fields(s) in the structure used to represent the branches, which are not required in the array representation.
3. Insertion, deletion and other operations can be implemented efficiently as above. The difference between binary trees and arrays here is that with a binary tree the operation can be implemented as a "local" operation, with only two or three nodes of the tree involved, whereas with ordered arrays sometimes the whole of the array must be modified to implement a single operation.

Part 1: Add books and print database (40 marks)

Now you should have decided whether to use arrays or (as recommended) binary trees for the assignment, and we can now describe the details of the assignment. Firstly, `cd` to your `eeclabs` directory. Then

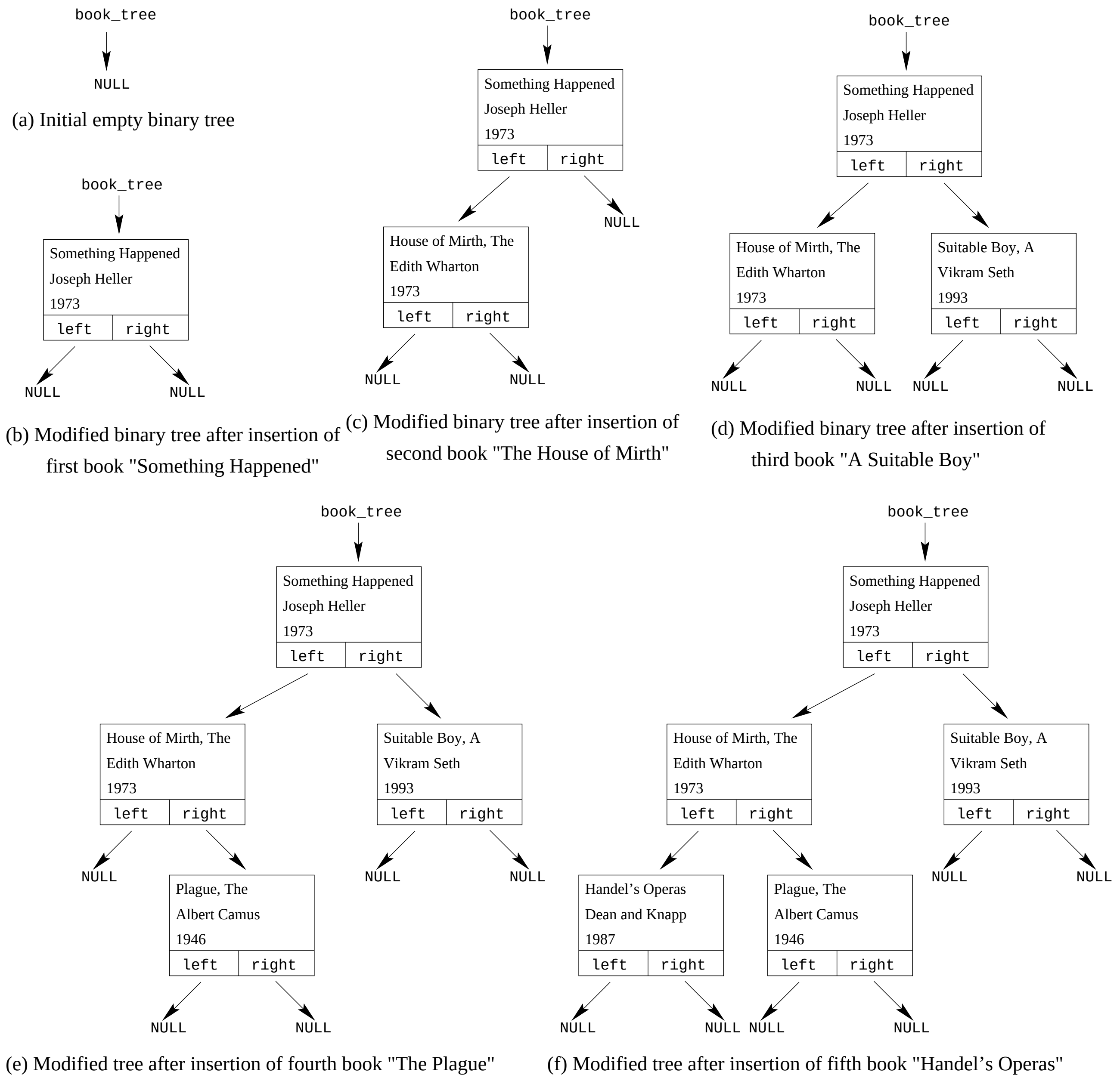

Figure 2: A simple binary tree. (a) Initial empty tree pointing to `NULL`. (b) After the first book is added. (c) After the second book is added. (d) After the third book is added. (e) & (f) after the fourth and fifth books. See program segment for code to create this tree.

download `database.c` and input file "input1" from SurreyLearn (`http://surreylearn.surrey.ac.uk`) or from `http://info.ee.surrey.ac.uk/Teaching/Courses/C/`, and then put them into your directionary. `database.c` is a "template" program with some empty functions, which you shall fill in with code as the assignment proceeds. Compile the program with the command

```
gcc -ansi -Wall database.c -o database
```

It should compile successfully (although it may give a warning that the `read_string()` function is "defined but not used", which you can ignore). Run it using the command:

```
./database
```

You will find that none of the menu options do anything, except 5 (Exit), which should work! When a menu option is selected, the relevant function (`menu_add_book()`, `menu_get_book_details()`, `menu_delete_book()` or `menu_print_database()`) is called, but these functions are empty. Your assignment is to build this program into a useful database program by filling in these empty functions.

Copy the `database.c` file into a new file `database1.c`, and work with this file in this part of the assignment. If you're using the binary tree representation, you can delete the `book_array` and `no_books` variables. If you want to use arrays, you can delete the definition of the `book_tree` variable and the `get_tree_depth()` function, and remove the body of the `menu_print_tree()` function.

Please note that, as with the other assignments, the testing that will be applied to your program involves comparing what your program prints to standard output (`stdout`) with the correct output specified in the assignment sheet and produced by our reference program. Your program must conform to the specified output format to be classified as a working program. Therefore please take note of any instructions regarding output. The use of the standard error (`stderr`) is however unrestricted, so any debugging messages, prompts etc. should be `fprintf`'d to `stderr`.

In the first part of the assignment, you are to fill in the functions `menu_add_book()` and `menu_print_database()` by modifying `database1.c`. These functions are called when menu options 0 (add new book) and 3 (print database to screen) respectively are selected by the user.

`menu_add_book()` should prompt the user to enter the title, author and publication year of the new book, in that order, and add the new book to the array/binary tree database. The prompt messages (which you can choose for yourself) should be printed to standard error (use `fprintf(stderr,"...)`, NOT standard output (i.e. DO NOT use `printf("...)`. The title and author should be entered as strings, truncated to `MAX_TITLE_LENGTH` and `MAX_AUTHOR_LENGTH` characters respectively. The publication year is an integer. If any part of the description is entered incorrectly (i.e. empty title/author strings are typed, or non-number year entered), the program should continue to prompt for the relevant personal detail until a legal version is provided, before going on to the next part of the book description. A "The" or "A" at the start of a title should be relegated with a comma to the end of the title string, as in the examples above.

To help you implement `menu_add_book()`, the function `read_line()` is provided, which reads a line of input and stores it in a string (the `read_string()` function also provided will be useful in part 3).

The function `menu_print_database()` should print the details of each book in the library database in alphabetical order of title to standard output (i.e. DO use `printf()` here), with each part being printed on a line by itself and prefixed by the strings `"Title:"`, `"Author:"` and `"Year:"` respectively, and also a space separating the prefix from the relevant following string/character/integer. There should be a blank line between each book. Do not print any extra invisible spaces at the end of lines.

Compile the program with the command

```
gcc -ansi -Wall database1.c -o database1
```

To test your program, the file `input1` can be used as input to the `database` program, to simulate keyboard input. Once you have adapted your program as above, and compiled it, run the command

```
./database1 < input1 > tmp
```

The contents of the standard output are sent to the file `tmp`, which should then contain

```
Title: Dispossessed, The
Author: Ursula Le Guin
Year: 1974

Title: Don Quixote
Author: Cervantes
Year: 1605

Title: Feersum Endjinn
Author: Iain M. Banks
Year: 1994

Title: Longitude
Author: Dava Sobel
Year: 1996

Title: Structures
Author: J. E. Gordon
Year: 1978
```

You should generate other test files in the same way, to make sure your program is working, particularly to test its behaviour with illegal inputs.

NOTE 1: You should not change the `main()` function or the values of the menu codes (`ADD_CODE` etc.). You should use either the global variable `book_tree` to represent your tree, or if using arrays use the `book_array` and `no_books` variables defined for you. If you're using binary trees you should not change the provided `get_tree_depth()` or `menu_print_tree()` functions, which are used by the testing programs when marking your assignment. Apart from these conditions you are free to implement the program as you wish, so long as it produces the correct output to `stdout` for a given keyboard input (from `stdin`). So for instance you do not have to use the `read_line()` function; it is only intended as a helpful suggestion. This note applies to all parts of this assignment.

NOTE 2: Most of the marks will be for correct operation of the program given legal input. The tests and actions described above for illegal/over-long inputs in `menu_add_book()` will gain you a few extra bonus marks.

Once you are sure the `menu_add_book()` and `menu_print_database()` functions are working, and have written comment blocks into the source code for any extra internal functions you have added to the `database1.c` program (as described in the rules for assignments on the Web page), re-name your file according to the following specification and then submit the program to **SurreyLearn**.

NOTE 3: The name of the submitted file MUST be proceeded with your surname and initial followed by the name of the program. For example, if your surname is “Brown”, and your first name is “Peter”, you need to name the file as follows:

BROWNP-database1.c

The first part is your surname and initial followed by a hyphen, and then followed by the original filename (`database1.c`). If you also have middle names, ignore them in the file name. `If your files do not follow this name specification, 10 percent will be deducted from your final mark.`

Part 2: Get/Delete books from the database (36 marks)

Download the input file "input2" into your directory from SurreyLearn (`http://surreylearn.surrey.ac.uk`) or from `http://info.ee.surrey.ac.uk/Teaching/Courses/C/`. In the second part of the assignment, which follows on from the program `database1.c` you wrote in part 1, you are to fill in the functions `menu_get_book_details()` and `menu_delete_book()`. These functions are called when menu options 1 (get details of book) and 2 (delete book from database) respectively are selected by the user.

Firstly copy the `database1.c` file from part 1 into a new file `database2.c`, and work with this file in this part of the assignment.

`menu_get_book_details()` should prompt the user to enter the full title of the book to be deleted (as it appears in the database), and then print the title, author and year of publication of the book in the same format as for `menu_print_database()`. If the title does not match any book in the database, the function should `fprintf` an error message to `stderr` (e.g. "Book not found") and return.

`menu_delete_book()` should prompt the user to enter the full title of the book to be deleted (as it appears in the database), and delete the named book from the book database. If the title does not match anyone in the database, the function should `fprintf` an error message to `stderr` (e.g. "Book not found") and return with no change to the database.

SPECIAL INSTRUCTIONS FOR TREE USERS

If you are using a tree to represent the database, you can either attempt to delete the book from the tree, which is very tricky to implement, or retain the book to be deleted in the tree, and simply "mark" it as deleted. You can mark a book as deleted by, for instance, setting its publication year to a special value, say 9999, or alternatively setting its author string to empty (as you will see below, you should not modify the title string). If you choose this simpler option, you will need to modify your code for `menu_add_book()`, `menu_print_database()` and `menu_get_book_details()` to recognise the special year and take the appropriate action (or inaction). For instance, `menu_print_database()` should ignore marked books, and if you are adding a book into the database that has previously been deleted, you merely need to "revive" the book by overwriting the author and publication year with the new values. Note that in order to do the latter, you must have retained the title string so that it can be recognised.

If you want to try and delete a book "properly" from a tree, here are a few hints. If book A is the book being deleted, find the book B that lies "above" book A and points to it. Set the relevant left/right pointer of B (whichever pointed originally to A) to the left (or right, if you prefer) pointer of A. Then you have to insert the sub-tree attached to the right (or left) pointer of A in the correct place in the tree, which is a `NULL` left or right branch in the position in the sub-tree contained by book B that maintains the alphabetical order of the tree. Finally free the memory for the deleted book structure. Deleting the top-most node should be treated as a special case. Good luck if you try this!

END OF SPECIAL INSTRUCTIONS.

Compile the program with the command

```
    gcc -ansi -Wall database2.c -o database2
```

To test your program, the file `input2` can be used as input to the `database2` program, to simulate keyboard input. Once you have adapted your program as above, and compiled it, run the command

```
   ./database2 < input2 > tmp
```

The contents of the standard output are sent to the file `tmp`, which should then contain

```
Title: Something Happened
Author: Joseph Heller
Year: 1973

Title: Frankenstein
Author: Mary Shelley
Year: 1816

Title: House of Mirth, The
Author: Edith Wharton
Year: 1905

Title: Frankenstein
Author: Mary Shelley
Year: 1816

Title: House of Mirth, The
Author: Edith Wharton
Year: 1905

Title: Something Happened
Author: Joseph Heller
Year: 1973

Title: Frankenstein
Author: Mary Shelley
Year: 1816

Title: Something Happened
Author: Joseph Heller
Year: 1973

Title: Frankenstein
Author: Mary Shelley
Year: 1816
```

You should generate other test files in the same way, to make sure your program is working, particularly to test its behaviour with illegal inputs.

Once you are sure the `menu_delete_book()` function is working, and have written comment blocks into the source code for any extra internal functions you have added to the `database2.c` program (as described in the rules for assignments on the Web page), re-name your file according to the following specification and then submit the program to **SurreyLearn**.

NOTE 1: The name of the submitted file MUST be proceeded with your surname and initial followed by the name of the program. For example, if your surname is "Brown", and your first name is "Peter", you need to name the file as follows:

BROWNP-database2.c

The first part is your surname and initial followed by a hyphen, and then followed by the original filename (`database2.c`). If you also have middle names, ignore them in the file name. `If your files do not follow this name specification, 10 percent will be deducted from your final mark.`

Part 3: Read initial book database from file (24 marks)

Download the input file "data" into your directory from SurreyLearn (`http://surreylearn.surrey.ac.uk`) or from `http://info.ee.surrey.ac.uk/Teaching/Courses/C/`. In the third part of the assignment, which follows on from the program `database2.c` you wrote in part 2, you are to fill in the function `read_book_database()`. This function is called when the program is run with an argument, which specifies a file of books with which to initialize the databse.

Firstly copy the `database2.c` file from part 2 into a new file `database3.c`, and work with this file in this part of the assignment.

`read_book_database()` should read the database in the same format that `menu_print_database()` prints them, except that it should be able to accept books not in alphabetical order. If any illegal title/author string or publication year appears in the file, the program should exit with an error message sent to `stderr`. Otherwise, the file is read, the database is built, and the menu interface starts as normal, with the pre-loaded book database.

Compile the modified program with the command

```
gcc -ansi -Wall database3.c -o database3
```

To test your program, the file `data` can be used as input to the `database3` program, to simulate keyboard input. Once you have adapted your program as above, and compiled it, run the command

```
./database3 data > tmp
```

and then select options "3" and "5" in turn to print the database to standard output and exit. The contents of the standard output are sent to the file `tmp`, which should then contain

```
Title: Altered States
Author: Anita Brookner
Year: 1996

Title: Dispossessed, The
Author: Ursula Le Guin
Year: 1974

Title: Don Quixote
Author: Cervantes
Year: 1605

Title: Feersum Endjinn
Author: Iain M. Banks
```

```
Year: 1994

Title: Fever Pitch
Author: Nick Hornby
Year: 1992

Title: Frankenstein
Author: Mary Shelley
Year: 1816

Title: Handel’s Operas
Author: W. Dean and J. M. Knapp
Year: 1987

Title: House of Mirth, The
Author: Edith Wharton
Year: 1905

Title: Immortality
Author: Milan Kundera
Year: 1991

Title: Longitude
Author: Dava Sobel
Year: 1996

Title: Original Sin
Author: P. D. James
Year: 1994

Title: Plague, The
Author: Albert Camus
Year: 1946

Title: Right Stuff, The
Author: Tom Wolfe
Year: 1979

Title: Something Happened
Author: Joseph Heller
Year: 1973

Title: Structures
Author: J. E. Gordon
Year: 1978

Title: Suitable Boy, A
Author: Vikram Seth
Year: 1993

Title: Time Machine, The
Author: H. G. Wells
```

```
Year: 1895
```

You should generate other test files in the same way, to make sure your program is working, particularly to test its behaviour with illegal inputs.

Once you are sure the `read_book_database()` function is working, and have written comment blocks into the source code for any extra internal functions you have added to the `database3.c` program (as described in the rules for assignments on the Web page), re-name your file according to the following specification and then submit the program to **SurreyLearn**.

NOTE 1: The name of the submitted file MUST be proceeded with your surname and initial followed by the name of the program. For example, if your surname is "Brown", and your first name is "Peter", you need to name the file as follows:

BROWNP-database3.c

The first part is your surname and initial followed by a hyphen, and then followed by the original filename (`database3.c`). If you also have middle names, ignore them in the file name. `If your files do not follow this name specification, 10 percent will be deducted from your final mark.`

If you have implemented all the functions correctly, you will have built a database program that can read an existing database and modify it under interactive keyboard control.

Printout of `database.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

/* maximum number of books that can be stored at once (relevant only
   to storage using an array) */
#define MAX_BOOKS 200

#define MAX_TITLE_LENGTH  100
#define MAX_AUTHOR_LENGTH 100

/* Book structure
   */
struct Book
{
   /* Book details */
   char title[MAX_TITLE_LENGTH+1];   /* name string */
   char author[MAX_AUTHOR_LENGTH+1]; /* job string */
   int  year;                        /* year of publication */

   /* pointers to left and right branches pointing down to next level in
      the binary tree (for if you use a binary tree instead of an array) */
   struct Book *left, *right;
};
```

```
/* tree of books, initialized to NULL. Ignore if using an array */
static struct Book *book_tree = NULL;

/* array of books, and number of books stored initialized to zero. Ignore
 * if using a binary tree */
static struct Book book_array[MAX_BOOKS];
static int no_books = 0;

/* read_line():
 *
 * Read line of characters from file pointer "fp", copying the characters
 * into the "line" string, up to a maximum of "max_length" characters, plus
 * one for the string termination character '\0'. Reading stops upon
 * encountering the end-of-line character '\n', for which '\0' is substituted
 * in the string. If the end of file character EOF is reached before the end
 * of the line, the failure condition (-1) is returned. If the line is longer
 * than the maximum length "max_length" of the string, the extra characters
 * are read but ignored. Success is returned (0) on successfully reading
 * a line.
 */
static int read_line ( FILE *fp, char *line, int max_length )
{
   int i;
   char ch;

   /* initialize index to string character */
   i = 0;

   /* read to end of line, filling in characters in string up to its
      maximum length, and ignoring the rest, if any */
   for(;;)
   {
      /* read next character */
      ch = fgetc(fp);

      /* check for end of file error */
      if ( ch == EOF )
         return -1;

      /* check for end of line */
      if ( ch == '\n' )
      {
         /* terminate string and return */
         line[i] = '\0';
         return 0;
      }

      /* fill character in string if it is not already full*/
      if ( i < max_length )
         line[i++] = ch;
   }
   /* the program should never reach here */
   return -1;
}

/* read_string():
 *
 * Reads a line from the input file pointer "fp", starting with the "prefix"
 * string, and filling the string "string" with the remainder of the contents
 * of the line. If the start of the line does not match the "prefix" string,
 * the error condition (-1) is returned. Having read the prefix string,
 * read_string() calls read_line() to read the remainder of the line into
 * "string", up to a maximum length "max_length", and returns the result.
 */
static int read_string ( FILE *fp,
                         char *prefix, char *string, int max_length )
{
   int i;

   /* read prefix string */
   for ( i = 0; i < strlen(prefix); i++ )
      if ( fgetc(fp) != prefix[i] )
         /* file input doesn't match prefix */
         return -1;

   /* read remaining part of line of input into string */
   return ( read_line ( fp, string, max_length ) );
}

/* menu_add_book():
 *
 * Add new book to database
 */
static void menu_add_book(void)
{
   /* fill in the code here in part 1, and add any extra functions you need */
}

/* menu_print_database():
 *
 * Print database of books to standard output in alphabetical order of title.
 */
static void menu_print_database(void)
{
   /* fill in the code here in part 1, and add any extra functions you need */
}

/* menu_get_book_details():
 *
 * Get details of book from database.
 */
static void menu_get_book_details(void)
{
   /* fill in the code here in part 2, and add any extra functions you need */
}

/* menu_delete_book():
 *
 * Delete new book from database.
 */
static void menu_delete_book(void)
{
   /* fill in the code here in part 2, and add any extra functions you need */

}

/* read file containing database of books */
static void read_book_database ( char *file_name )
{
   /* fill in the code here in part 3, and add any extra functions you need */
}

/* get_tree_depth():
 *
 * Recursive function to compute the number of levels in a binary tree.
 */
static int get_tree_depth ( struct Book *book, int level )
{
   int level1, level2;

   /* return with the current level if we've reached the bottom of this
      branch */
   if ( book == NULL ) return level;

   /* we need to go to the next level down */
   level++;

   /* count the number of levels down both branches */
   level1 = get_tree_depth ( book->left,  level );
   level2 = get_tree_depth ( book->right, level );

   /* return the depth of the deepest branch */
   if ( level1 > level2 ) return level1;
   else return level2;
}

/* menu_print_tree():
 *
 * Print tree to standard output. You can use this function to print out the
 * tree structure for debugging purposes. It is also used by the testing
 * software to check that the tree is being built correctly.
 *
 * The first letter of the title of each book is printed.
 */
static void menu_print_tree(void)
```

```
{
   int no_levels, level, size, i, j, k;
   struct Book **row;

   /* find level of lowest node on the tree */
   no_levels = get_tree_depth ( book_tree, 0 );

   /* abort if database is empty */
   if ( no_levels == 0 ) return;

   /* compute initial indentation */
   assert ( no_levels < 31 );

   row = (struct Book **) malloc((1 << (no_levels-1))*sizeof(struct Book *));
   row[0] = book_tree;
   printf ( "\n" );
   for ( size = 1, level = 0; level < no_levels; level++, size *= 2 )
   {
      /* print books at this level */
      for ( i = 0; i < size; i++ )
      {
         if ( i == 0 )
            for ( j = (1 << (no_levels - level - 1)) - 2; j >= 0; j-- )
               printf ( " " );
         else
            for ( j = (1 << (no_levels - level)) - 2; j >= 0; j-- )
               printf ( " " );

         if ( row[i] == NULL )
            printf ( " " );
         else
            printf ( "%c", row[i]->title[0] );
      }

      printf ( "\n" );

      if ( level != no_levels-1 )
      {
         /* print connecting branches */
         for ( k = 0; k < ((1 << (no_levels - level - 2)) - 1); k++ )
         {
            for ( i = 0; i < size; i++ )
            {
               if ( i == 0 )
                  for ( j = (1 << (no_levels - level - 1))-3-k; j >= 0; j-- )
                     printf ( " " );
               else
                  for ( j = (1 << (no_levels - level)) - 4 - 2*k; j >= 0; j-- )
                     printf ( " " );

               if ( row[i] == NULL || row[i]->left == NULL )
                  printf ( " " );
               else
                  printf ( "/" );

               for ( j = 0; j < 2*k+1; j++ )
                  printf ( " " );

               if ( row[i] == NULL || row[i]->right == NULL )
                  printf ( " " );
               else
                  printf ( "\\" );
            }

            printf ( "\n" );
         }

         /* adjust row of books */
         for ( i = size-1; i >= 0; i-- )
         {
            row[2*i+1] = (row[i] == NULL) ? NULL : row[i]->right;
            row[2*i]   = (row[i] == NULL) ? NULL : row[i]->left;
         }
      }
   }

   free(row);
}

/* codes for menu */
#define ADD_CODE     0
#define DETAILS_CODE 1
#define DELETE_CODE  2
#define PRINT_CODE   3
#define TREE_CODE    4
#define EXIT_CODE    5

int main ( int argc, char *argv[] )
{
   /* check arguments */
   if ( argc != 1 && argc != 2 )
   {
      fprintf ( stderr, "Usage: %s [<database-file>]\n", argv[0] );
      exit(-1);
   }

   /* read database file if provided, or start with empty database */
   if ( argc == 2 )
      read_book_database ( argv[1] );

   for(;;)
   {
      int choice, result;
      char line[301];
```

```
/* print menu to standard error */
fprintf ( stderr, "\nOptions:\n" );
fprintf ( stderr, "%d: Add new book to database\n",      ADD_CODE );
fprintf ( stderr, "%d: Get details of book\n",       DETAILS_CODE );
fprintf ( stderr, "%d: Delete book from database\n",  DELETE_CODE );
fprintf ( stderr, "%d: Print database to screen\n",    PRINT_CODE );
fprintf ( stderr, "%d: Print tree\n",                   TREE_CODE );
fprintf ( stderr, "%d: Exit database program\n",        EXIT_CODE );
fprintf ( stderr, "\nEnter option: " );

if ( read_line ( stdin, line, 300 ) != 0 ) continue;

result = sscanf ( line, "%d", &choice );
if ( result != 1 )
{
   fprintf ( stderr, "corrupted menu choice\n" );
   continue;
}

switch ( choice )
{
   case ADD_CODE: /* add book to database */
   menu_add_book();
   break;

   case DETAILS_CODE: /* get book details from database */
   menu_get_book_details();
   break;

   case DELETE_CODE: /* delete book from database */
   menu_delete_book();
   break;

   case PRINT_CODE: /* print database contents to screen
                       (standard output) */
   menu_print_database();
   break;

   case TREE_CODE: /* print tree to screen (standard output) */
   menu_print_tree();
   break;

   /* exit */
   case EXIT_CODE:
   break;

   default:
   fprintf ( stderr, "illegal choice %d\n", choice );
   break;
}
```

```
      /* check for exit menu choice */
      if ( choice == EXIT_CODE )
         break;
   }

   return 0;
}
```