CS2002 Practical 4 - Stable Circuits

Matriculation Number: 160001362

10/04/2018

Contents

1 Overview

This practical specified the development of a system to simulate digital circuits with feedback (generated from a circuit description language) using appropriate data structures and functions. In addition, the practical specified that the output as well as the truth table of the simulated circuit should be appropriately displayed after having been run with suitable inputs.

2 Design and Implementation

2.1 Summary

The representation of the circuit in the program is an array list (custom data type) of logical gates which is generated by parsing the user input or input from a file into the program. Each gate points to an output wire, as well as to zero to two input wires. These wires are stored in a linked list (also a custom data type). After the circuit has been generated (provided the input is valid) then the truth table for the circuit is output which to be achieved results in the program evaluating the circuit given the possible inputs to check for stabilisation. After the truth table has been output, the data structures utilised are freed.

2.2 Data Structures

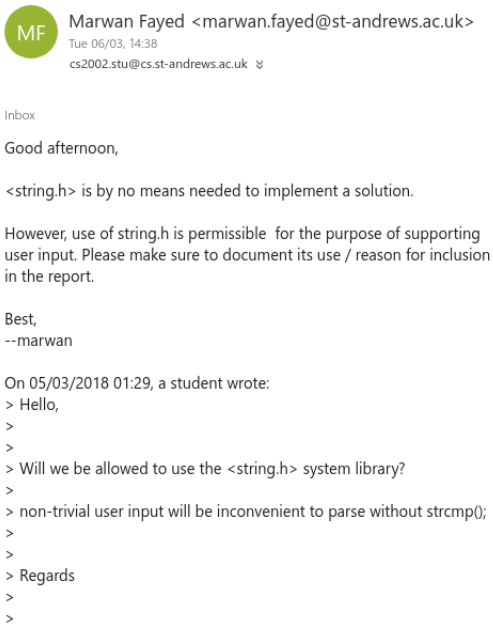
- **Wire**:
  - **name** - a string to store the name of the wire (used as its identifier).
  - **val** - the integer value representing the current state of the wire (either 1 or 0).
  - **nextVal** - the integer value representing the next state of the wire when the circuit is evaluated (either 0 or 1).
- **Gate** (represents a logic gate in the circuit):
  - **op** - stores a string representing the logical operator of the gate.
  - **output** - stores a Wire* pointing to the output wire of the gate.
  - **input1** - stores a Wire* pointing to the first input wire of the gate (if none then NULL).
  - **input2** - stores a Wire* pointing to the second input wire of the gate (if none then NULL).
- **LinkedList** (used to store wires):
  - **wire** - a Wire* pointing to the wire stored at that node in the linked list.
  - **next** - a LinkedList* pointing to the next node in the list.

- **ArrayList** (used to store the list of logical gates):
  - **gates** - a Gate* (array of type Gate) which stores each gate.
  - **size** - stores the current size of the array list (i.e. number of Gate structs in gates).
  - **capacity** - stores the number of locations in the gates (size cannot exceed capacity).

2.2.1 Files and Libraries

- **circuits.c** - This file contains the main function of the program which coordinates the sequence of operations. This file also contains the functions that directly relate to the creation and manipulation of the wires and gates such as creating wires and gates, generating the circuit, generating inputs to the circuit, evaluating the circuit and computing the next state, as well as stabilising the circuit. This file is also where the fixed value wires 'one' and 'zero' are declared.
- **io.c** - Handles the input and output of the system. The functions contained within this file include the likes of outputting the truth table of the circuit, tokenizing the file/console input, as well as parsing the file/console input and validating it. Also contained within the file is a constant array of strings which stores all of the valid operators (NOT, AND, OR, NAND, NOR, XOR, EQ and IN).
- **linkedlist.c** - This file contains functions that operate on the linkedlist of wires in the program. This file is similar to the linkedlist file used in the CardTrick practical, only now the linkedlist has been adapted to store wires instead of cards and has had some additional functions defined. Functions in this file include those to create new nodes, add nodes to the linked list, get the size of the linked list, check if the linked list contains some value, get a node from the list with some given value, reset all of the values in the list, and finally freeing nodes of the list.
- **readmymind.h** - Contains definitions for the size of data structure dimensions (e.g. COLUMN_SIZE, SUIT_SIZE), the structs representing the data structures, as well as for the signatures of functions used throughout the program.
- **stdbool.h** - Used to allow true and false values to be returned when validating the user's input.
- **stdlib.h** - Used to set appropriate values to NULL as well as enabling the use of malloc and free.
- **stdio.h** - Used for input and output.
- **assert.h** - Used to terminate program if branches of code are reached that should be unreachable.

- **string.h -** Used for the strlen function to check if the size of the user's input is greater than a single character. Although not explicitly stated as one of the permitted libraries in the specification, Dr Fayed stated in a query that its use is permissible:



MF

Marwan Fayed <marwan.fayed@st-andrews.ac.uk>
Tue 06/03, 14:38
cs2002.stu@cs.st-andrews.ac.uk

Inbox

Good afternoon,

<string.h> is by no means needed to implement a solution.

However, use of string.h is permissible for the purpose of supporting user input. Please make sure to document its use / reason for inclusion in the report.

Best,
--marwan

On 05/03/2018 01:29, a student wrote:
> Hello,
>
>
> Will we be allowed to use the <string.h> system library?
>
> non-trivial user input will be inconvenient to parse without strcmp();
>
>
> Regards
>
>

2.3 Index Based

2.3.1 Summary

Declares and initialises a deck array of 52 cards and uses it to fill a Columns struct of 21 cards (of 3 column structs of 7). The columns are filled by randomly generating a number (signifying a card to select from the deck) and checking an array of selected values to see if the card has already been picked. It does this until 21 cards have been selected.

The user then inputs their chosen column (this data is validated using an array of valid inputs) and this value is converted to an integer format and is then used to gather the columns. The columns are gathered by assigning the selected column of the previous Columns struct to the middle Column struct of the new Columns struct. The other columns are then iterated over and assigned.

Finally the Columns struct with the selected column in the middle is dealt to a new set of columns by iterating over each element in the new Columns struct and calculating which old element should be dealt.

The above process of getting user input, gathering the cards, and dealing the cards is then repeated three times with the user's card being displayed at the end.

2.3.2 Data Structures

- **Card** - Has two integer attributes:
  - **suit** - a number from 0 - 3 representing either spade, heart, diamond, or club.
  - **rank** - a number from 0 - 12 representing and Ace, 2, 3, 4, 5, 6, 7, 8, 9, Jack, Queen, or King respectively.
- **Deck** - Has one attribute 'cards' which is an array of type Card which has the size of the constant DECK_SIZE (in the general case 52).
- **Column** - Like the Deck struct - has one attribute 'cards' which is an array of type Card which has the size of constant COLUMN_SIZE (typically 7).
- **Columns** - Has an attribute 'column' which is an array of columns of size NUM_COLUMNS (typically 3).

2.4 Pointer Based

2.4.1 Summary

The pointer based implementation has an additional C file '**linkedlist.c**' which has functions to interface with the linkedlist data structure declared in readmymind.h. These include functions to create and add nodes to a linked list as well as retrieve nodes from a linked list. In addition there is a function to free the memory occupied by nodes of a linked list to avoid memory leakage and a function to update the indices of a given linked list.

A linked list implementation by HackerEarth[1] was used to give an idea of how to implement a basic linked list data structure in C.

The pointer version uses almost the exact same process as described in section 2.1.1. Aside from the way the data structure is accessed during the trick, the main difference between the two implementations is the way that the user input and output works. In the pointer based implementation instead of accessing card icons to be displayed using a constant array, a switch statement containing print statements is used.

Similarly, instead of storing a constant array of valid inputs and accessing it to validate the user's input, a switch statement is used for input validation. A

[1] https://www.hackerearth.com/practice/data-structures/linked-list/singly-linked-list/tutorial/

switch statement rather than iteration is also used in the gather stage. While the implementation of these features is more elegant in the array based implementation, this alternate implementation demonstrates that arrays are not strictly necessary.

The pointer implementation shares a common Card struct with the index based implementation, however the other following structures differ in implementation:

2.4.2 Data Structures

- **Card** - Has two integer attributes:
  - **suit** - a number from 0 - 3 representing either spade, heart, diamond, or club.
  - **rank** - a number from 0 - 12 representing and Ace, 2, 3, 4, 5, 6, 7, 8, 9, Jack, Queen, or King respectively.
- **LinkedList** - Has the following attributes:
  - **card** - Stores a Card attribute that can be thought of as the 'value' of the node (a card in the deck or columns).
  - **next** - A linkedlist pointer to the next node the in the list.
  - **index** - An integer signifying a node's position in its linkedlist - manually set and used so that it would be easier to translate the functions in the index/array based implementation into the pointer implementation.
  - **chosen** - A boolean used when selecting the random 21 cards from the deck of cards to signify whether a particular card has already been selected.
- **Node** - A typedef that allows a LinkedList pointer to be declared using the keyword 'Node'.
- **Deck** - A struct that has one attribute 'node' which is a LinkedList Node. Used to allow 52 nodes to be added, each of which stores a unique playing card.
- **Columns** - A struct that contains three Node attributes called 'first', 'second' and 'third'. Each attribute stores the heads of their respective linkedlists which represent the three columns of the card trick.

3 Testing

The following is a sample of output generated from testing the implementations. The function scanf was used to get the input from the user and proved surprisingly robust given the limited scope of this program.

3.1 Input Validation

Figure 1: Invalid inputs being validated. Includes multiple character input, character input larger than allocated memory for the input char*, invalid single character input, as well as some miscellaneous input.

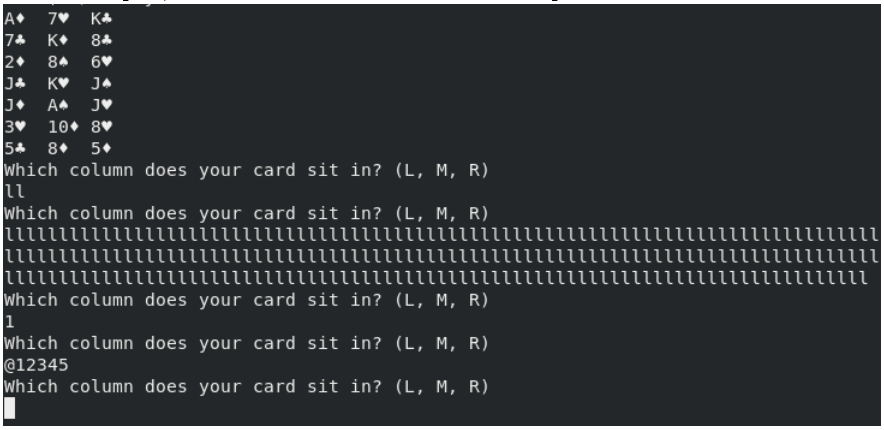


Figure 2: Example of valid inputs being accepted. Valid inputs include the characters l, L, r, R, m, and M.

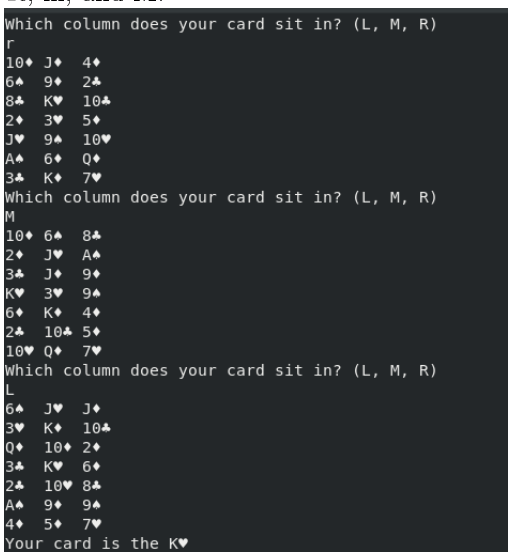
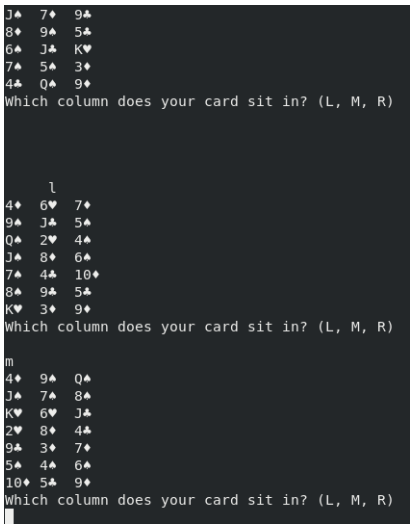
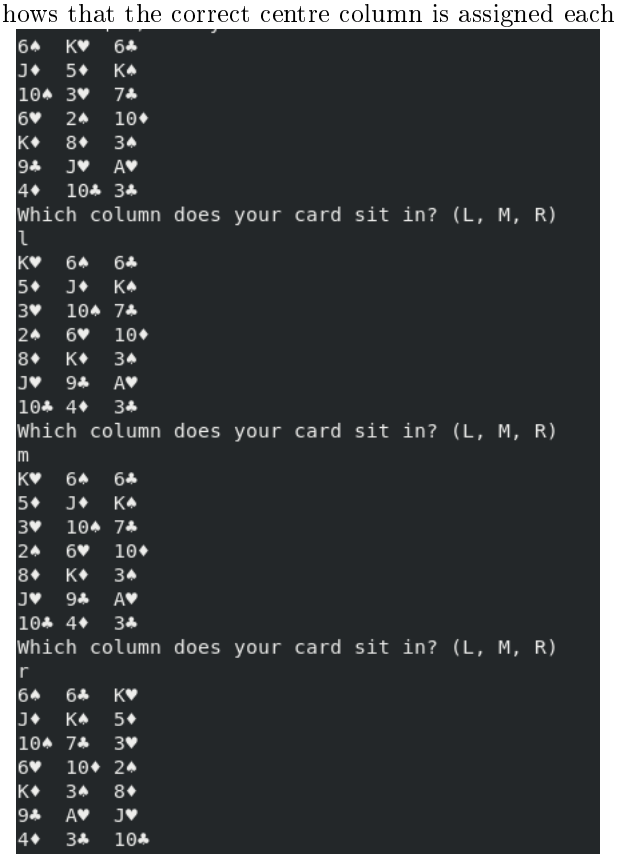


Figure 3: Example of valid inputs being accepted with preceding white space (white space and new line characters do not effect the single character input).



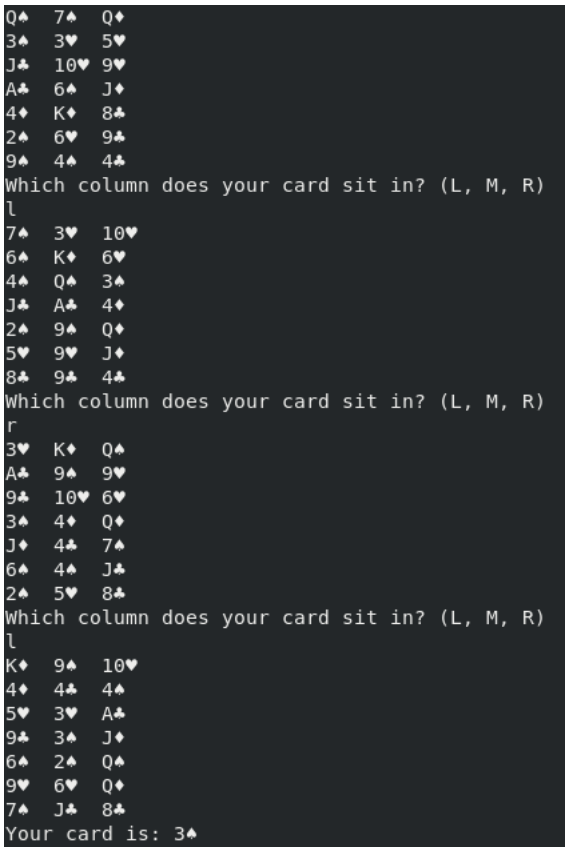
3.2 Gathering Columns

Figure 4: Output of the gather stage when the deal stage is commented out. The output shows that the correct centre column is assigned each time.



3.3 Dealing Cards

Figure 5: Output of multiple deal stages when the user's card is the 3 of spades. This output clearly shows that the cards are correctly dealt left-to-right, top-to-bottom.



4 Conclusion and Evaluation

In conclusion the requirements specified in the practical document to develop a card trick program have been fully implemented using both an index based implementation as well as non-global pointer based solution utilising a linked list structure. The solutions also both follow the constraints specified in regard to the allowed libraries as well as both only using local instances of data structures.

To improve upon my solution I would re-implement the way the newColumns struct is initialised in the pointer implementation when the cards are gathered and dealt. The current implementation of the program leaks 1.5MB of memory due to unused allocated memory that was once pointed to by a Columns struct. Without the frees at the end of the main function the program initially had over 3MB of leaked memory. I attempted to free the original columns struct when newColumns is assigned in both the gather and deal functions to reduce the amount from 1.7MB to 0MB however this resulted in a segmentation fault and so I decided to permit some memory not being freed (this would be more of a problem if the trick is repeated many times).

Figure 6: Output of 'valgrind ./readmymind'

```
==23702== HEAP SUMMARY:
==23702==     in use at exit: 1,728 bytes in 72 blocks
==23702==   total heap usage: 153 allocs, 81 frees, 5,612 bytes allocated
==23702==
==23702== LEAK SUMMARY:
==23702==    definitely lost: 216 bytes in 9 blocks
==23702==    indirectly lost: 1,512 bytes in 63 blocks
==23702==      possibly lost: 0 bytes in 0 blocks
==23702==    still reachable: 0 bytes in 0 blocks
==23702==         suppressed: 0 bytes in 0 blocks
==23702== Rerun with --leak-check=full to see details of leaked memory
==23702==
```

5 Extensions

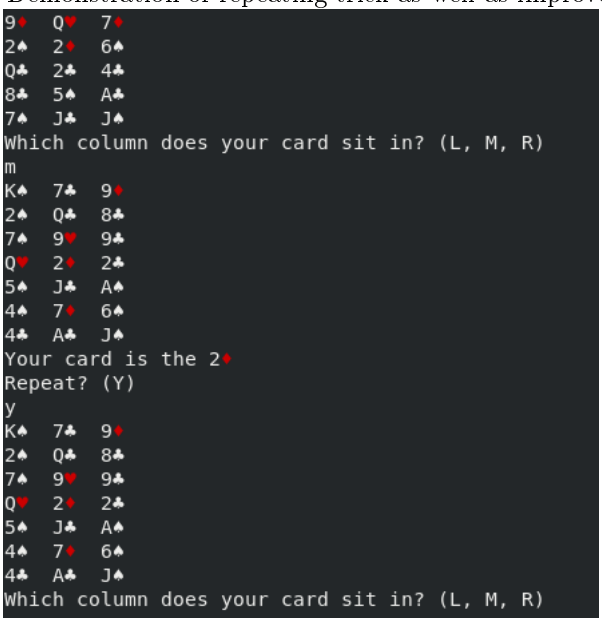
- **Allow Repetition of Trick -** In both implementations a while loop has been added to the main method to allow the user to repeat the card trick if they input the characters 'Y' or 'y' with anything else signifying an exit character. This has been achieved through the addition of a function getUserChoice in the io.c file which similarly to the getUserCol function gets the user's choice. A value of true is returned from a subsequent function isValidChoice if either character is entered.

- **Improve Ouput -** Improved output has been implemented with the letters A, J, Q, and K being output for the ranks 0, 10, 11 and 12 respectively and unicode card icons are output for the suits 0, 1, 2, and 3. For the suits 1 and 2 (hearts and diamonds) ANSI character codes have been used

to display the suit icons in the colour red.

- **Complexity Discussion** - It seems counter-intuitive to use pointers and a linked list for such an application due the natural way in which an index based solution can represent cards in columns. In addition to this, linked lists typically take O(n) time to traverse to access an element where as arrays are constant time, O(1), access using an index. The main argument for using linked lists in any application is the ability to dynamically reallocate the size of the data structure utilising it however in this application we know from the start what size data structures (decks and columns) we will be working with.

Figure 7: Demonstration of repeating trick as well as improved output



References

[1] Mohd Sanad, Zaki Rizvi, *"Singly Linked List Tutorial"*, HackerEarth, https://www.hackerearth.com/practice/data-structures/linked-list/singly-linked-list/tutorial/, (2016).