COP3502C Computer Science 1
Dr. Andrew Steinberg
Spring 2022

Programming Assignment 3

The Card Game of War

Max Points 100

Due: 3/14/2022 at 11:59pm

Background Story of this Assignment

You have just learned your first elementary data structure (woohoo!). Something that I stressed in lecture that you will see throughout your CS classes is that many data structures can be derived from the linked list. Therefore, I believe the first data structure programming assignment should revolve around the linked list for many reasons. One reason is that you will get the chance to apply the linked list in an application scenario. Another reason and perhaps one of the most important ones is that you will get a fundamental understanding of implementing data structures in C by coding up linked lists.

Have fun and start early (seriously start early)! Make sure to see the TAs and ULAs for help!

Assignment Details

For this programming assignment, you will get to simulate the classic card game war using a singly linked list. Rules for the game can be found at this website: https://gamerules.com/rules/war-card-game/

In this program we will always assume only two players will play. This means each player will represent its own linked list (that's right two separate linked lists). As the rules state there are a total of 52 cards. The following number list shows each set of cards in the deck. Each type contains 4 (♠, ♣, ♥, and ♦) cards (hence $4 * 13 = 52$ cards total).

1. Ace
2. King
3. Queen
4. Jack
5. 10
6. 9
7. 8
8. 7
9. 6
10. 5
11. 4
12. 3
13. 2

Also, the order presented shows the order of card dominance (top being the highest and bottom being the lowest). This will determine which player wins after cards are drawn. For example, if Player A draws the card king of hearts and Player B draws the card 9 of clubs, then Player A wins that round. As a result, Player A takes its own card drawn and Player B's card and puts it in the back of the pile (sounds like inserting the back of a linked list 🤔). Now as the rules state, the objective of the game is to get the player to lose all their cards (this sounds like until the respective player's linked list is empty 🤔).

The Provided Skeleton File

You were provided with a skeleton C file that has the main function and function prototypes. This section will discuss the lines of code provided for you in the main function to assist you with understanding how the code will execute.

- Lines 6 – 8 shows the preprocessor directives.
- Lines 10 – 14 defines the `typedef` struct called `card_t` you will be utilizing in the assignment. The struct contains the following member components.
    - An integer called `rank`. This will keep track of the priority of the card. This is utilized in determining the winning round when cards are drawn.
    - A char pointer called `type`. This is a **dynamic** string. It will store the card name. Examples include "8 of Spades", "6 of Hearts", "Ace of Hearts", etc…
    - A struct pointer called `nextptr` that points to a `cart_t` struct. This is simply the node in the linked list.
- Line 33-37 sets the seed. At this point in the course you should know what that does. The test script uses the value 0 as the test seed.
- Line 38 is a call to a user-defined function called `rules`. This function has already been implemented for you since it is just `printf` statements 😊.
- Lines 40 and 41 are just variable declarations. Variable player represents the player you get to choose. You can either be player 1 or 2. This is done in lines 43 – 45. Variable `result`, will determine which player won in displaying the respective message. You will be playing against the computer.
- Lines 47 – 58 shows a for loop that simulates a game. It will keep track which game number the simulation is playing. It should only play up to 5 games total.
- Line 52 calls the function `playRound` which simulates an entire game. The function returns an integer that is stored in `result`. This value is used to determine to proper message to display to the terminal window based on the player you pick at the beginning of the simulation run.

The Function Prototypes

You were provided with a skeleton C file that has the main function and function prototypes. This section will discuss the lines of code provided for you in the main function to assist you with understanding how the code will execute. DO NOT CHANGE THE PROTOTYPES! Points will be deducted. You can add additional helper functions as long as they are not called in main and doesn't affect the overall output.

```
void rules();
```

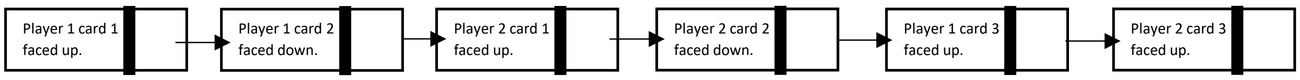
The `rules` function will display the rules of the game. It is already implemented for you so you do not need to change anything for this function.

```
int playRound();
```

The `playRound` function will simulate an ENTIRE round of the card game. This function will call the other functions (mentioned below) that are going to be mentioned in order to simulate the game. It takes no parameters and returns an integer value (1 or 2) that determines which player won the round. The part that determines which player won is already implemented for you in the `main` function.

IMPORTANT! Something important you will need to consider is the setup of the round. You will have a deck that is already setup as a linked list from the `openCardDeck` function, however it is not randomize (it is in top-down order of rankings). You will need to randomize the cards distributed to each player. Hint: Think about the `search`, `deckSize`, `insertBackDeck`, and `removeDeck` functions can properly pick a random card from the deck along with removing from the original setup deck.

ANOTHER SUPER IMPORTANT ITEM! When cards are drawn, and a winner is determined. Take the cards in the following order always when inserting back into a respective player's linked list. Player 1's card(s) go first and then Player 2's card(s) go last. The following figure shows the order to insert back into the winner's deck for the W-A-R scenario.


Player 1 card 1 faced up.
Player 1 card 2 faced down.
Player 2 card 1 faced up.
Player 2 card 2 faced down.
Player 1 card 3 faced up.
Player 2 card 3 faced up.

In this scenario, the winner of the first W-A-R gets a profit of 3 cards. If there is a tie again, then the profit is 5 cards (both players put 1 card faced down and the next one faced up to compare).

```
card_t * openCardDeck();
```

The `openCardDeck` function will setup the round by "opening the card deck pack". The function will open a text file (which is provided for you in the text file on Webcourses) called `deck.txt` and read the information to store in a linked list. You should see a user defined function called `insertBackSetup` being invoked. This function was provided for you already. There are no parameters however the function returns an address to a `typedef struct card_t`. This address is the head of a linked list. This linked list represents the whole deck you open from the container. It should contain 52 nodes. This function has been implemented for you. You do not need to make any changes or additions to this implementation. It is here to assist you with the game setup.

```
card_t * insertBackSetup(card_t *node, char *name, int cardrank);
```

The `insertBackSetup` function will insert a card to the back of the original deck for the setup purposes. The function takes three arguments. The first argument is an address to a `typedef struct card_t` which represents the head of the linked list that represents the original deck before being split between the players. The second argument is the name of the card (examples include "Ace of Hearts", "2 of Clubs", etc…). The last argument is the rank of card that will be used for the game to determine who wins each card draw. The function returns an address to a `typedef struct card_t` that represents the head of a linked list. You do not need to make any changes or additions to this implementation. It is here to assist you with the game setup.

```
int empty(card_t * node);
```

The `empty` function will determine if the linked list is empty. It takes a reference to a `typedef struct card_t` and returns an integer that determines if the linked list is empty or not. You do not need to make any changes or additions to this implementation. The function has already been implemented for you. It is here to assist you with the game implementation.

```
void cleanUp(card_t * head);
```

The `cleanUp` function will free the allocated memory used. This is important to implement to prevent memory leaks since you will be utilizing dynamic memory. It takes a reference to a head of a linked list and will free all the memory allocated in the program run contained in the linked list. The function does not return anything. Make sure you are properly freeing memory by utilizing valgrind.

```
int deckSize(card_t * head);
```

The `deckSize` function will count the number of cards in the linked list. The functions takes a reference to the head of the linked list and returns the number of cards in the linked list.

```
card_t * search(card_t * node, int spot);
```

The `search` function will traverse the linked list for a specific card in the pile. The function takes two arguments. The first argument is the reference to the linked list and an integer which is the location of the list called `spot`. The location starts with 0 (like indexing with an array) and goes up to the total number of nodes in the linked list minus 1.

```
card_t * copyCard(card_t * node);
```

The `copyCard` function will create a deep copy of a node in a linked list and return it. The parameter in the function header is the node that will be copied and returned.

```
card_t * removeCard(card_t * node, int spot);
```

The `removeCard` function will remove a node from the linked list. The function has two parameters. The first one is the reference to the linked list (head specifically) and an integer which represents the location (like indexing with an array). The function will deallocate the node in the respective `spot` from memory. Remember we are using dynamic allocation in this program. The function returns the head of the modified linked list.

```
card_t * insertBackDeck(card_t *head, card_t *node);
```

The `insertBackDeck` function will place a card to the back of the deck. In coding terms, place the node at the end of the linked list. There are two parameters in the function header. The first parameter is the reference to the head of the linked list and the second is the node that will be inserted to the end. This function simulates putting a card at the bottom of a pile.

```
int compareCard(card_t * cardp1, card_t * cardp2);
```

The `compareCard` function will compare the rankings of two cards. This simulates when players are drawing the cards from the respective piles. The function has two parameters. Both parameters are references to the `typedef struct card_t`. Each of these `typedef structs card_t` represents the head of each card from the respective player's pile. You will use the ranking system as described previously to determine who won. There are three scenarios to consider. One scenario is that player 1 wins. The second scenario is player 2 wins. The last scenario is that both players have a tie and will have to do the W-A-R draw. The function returns 3 possible numbers. The integer 1 returned represents that player 1 won. The integer 2 returned represents that player 2 won. The integer 0 returned means there was a tie. If the tie scenario occurs. Now if there is a tie, the rules state that the players will draw one card faced down and then place one more card face up and use that to compare for the W-A-R round. The winner of W-A-R round will take all 6 cards and place them at the end of their respective pile.

Another important scenario to consider for this is that you can possibly have consecutive ties and will have to repeat the action until a winner is declared. You will have to think about this implementation is done. This part is done in the `playRound` function. You do not put that in the `compareCard` function. `compareCard` just compares cards only and returns the outcome as described from above.

IMPORTANT! You can potentially run into a tie scenario where one of the players may not have the minimum cards (2 cards) to draw out for W-A-R. If that is the case, you will have the players draw the amount of cards the respective player has left and use the final card as the last card to flip and compare with the opponent. For example, if player 1 has 50 cards and player 2 has 2 cards, then each player will draw 2 cards instead of 3. This should be considered in the `playRound` function definition.

```
card_t * moveCardBack(card_t *head);
```

The `moveCardBack` function will take the card in front of the pile and place it in the back. In coding terms, you are taking the head of the linked list and moving it to the end. The function has one parameter which is the head of the linked list. After moving the card to the back, the function returns the new head of the linked list.

Requirements

Your program must follow these requirements.

- The output must match exactly (this includes case sensitivity, white space, and even new lines). Any differences in the output will cause the grader script to say the output is not correct. Test with the script provided in order to receive potential full credit. Points will be deducted!
- You may NOT use `realloc`, `calloc`, or any sort of other memory function except for `malloc` and `free`. If any other built in memory function is used then the ones stated will result in points being deducted!
- Make sure you do not have memory leaks! Those are bad to have. You can check for memory leaks using valgrind. In the provided file, you saw `malloc` being utilized, however it is not currently correlated with `free`. Think about how they would be used in the respective user defined functions. You will have to call them in your user defined functions. That is part of the challenge to think about with dynamic memory. You will have to fill that in your user defined functions.

- Do not change ANY content of the skeleton that was provided for you. Your code will be tested through a script that relies on this main function. Any changes to this will result in your program not working fully which will lead to point deductions that will not be fixed!
- Do not change the name of the text file provided for you. Use the original name that were provided for you.
- The text files must be in the same directory as your C file.
- Name your C file `program3_lastname_firstname.c` where `lastname` and `firstname` is your last and first name respectively. Please make sure it matches the spelling exactly how it is registered in Webcourses. Points will be deducted if the file is not named correctly.
- Make sure you test that `malloc` returns an actual heap address as discussed in lecture! If `malloc` isn't successful, then terminate the program using `exit(1)`. Make sure this statement doesn't get executed. Otherwise fix your code.
- Do not change the typedef struct that was provided for you.
- The dynamic strings allocated in your program will have no more than 20 characters.

Tips in Being Successful

Here are some tips and tricks that will help you with this assignment and make the experience enjoyable.

- **Draw! When coding with ADTs it is recommended you take a pencil and paper and draw your ADT after each line of code to see what you are doing in memory. This tedious and very smart method will go a super long way in coding the solution. I cannot emphasize this enough!!! It will save you from segmentation faults and losing information! It sounds tedious, but it will go a long way!!!**
- Do not try to write out all the code and build it at the end to find syntax errors. For each new line of code written (my rule of thumb is 2-3 lines), build it to see if it compiles successfully. **It will go a long way!**
- After any successful build, run the code to see what happens and what current state you are at with the program writing so you know what to do next! If the program performs what you expected, you can then move onto the next step of the code writing. If you try to write everything at once and build it successfully to find out it doesn't work properly, you will get frustrated trying find out the logical error in your code! **Remember, logical errors are the hardest to fix and identify in a program!**
- Start the assignment early! Do not wait last minute (the day of) to begin the assignment.
- Ask questions! It's ok to ask questions. If there are any clarifications needed, please ask TAs/ULAs and the Instructor! We are here to help!!! You can also utilize the discussion board on Webcourses to share a general question about the program as long as it doesn't violate the academic dishonesty policy.