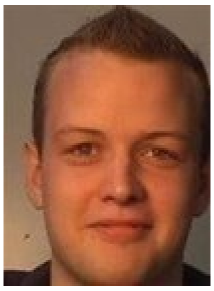
Technical University of Denmark

02322 - Machine Oriented Programming F21

11/05-2021 Machine Oriented Programming Project 2

Group 22



Christian Kyed
s184210, SWT

Sander Eg Albeck Johansen
s195453, SWT

Mikkel Hillebrandt Thorsager
s195470, SWT

Important links:

GitHub: https://github.com/CKyed/MaskinaerAflevering2

Youtube part 1: https://youtu.be/Z-b7PeXsZng

Youtube part 2 :https://youtu.be/gah7kHE6K2s

Requirements specifications

We have been assigned a project where we have to implement the card game “Yukon” in C using linked list. The game is played with a deck of cards, on a game board with 7 columns and 4 “foundations”, the objective of the game is to move the cards into the four foundations arranged by suit and in order from Ace - King.

We will also be making some “support” commands that will help support the game.

Analysis

For us to find out what exactly to implement in our “Yukon” game, we will go into detail and analyse the Yukon rules, the layout of the “board”, and cards in general.

The game is played with a single pack of 52 playing cards. with the suits hearth, spades

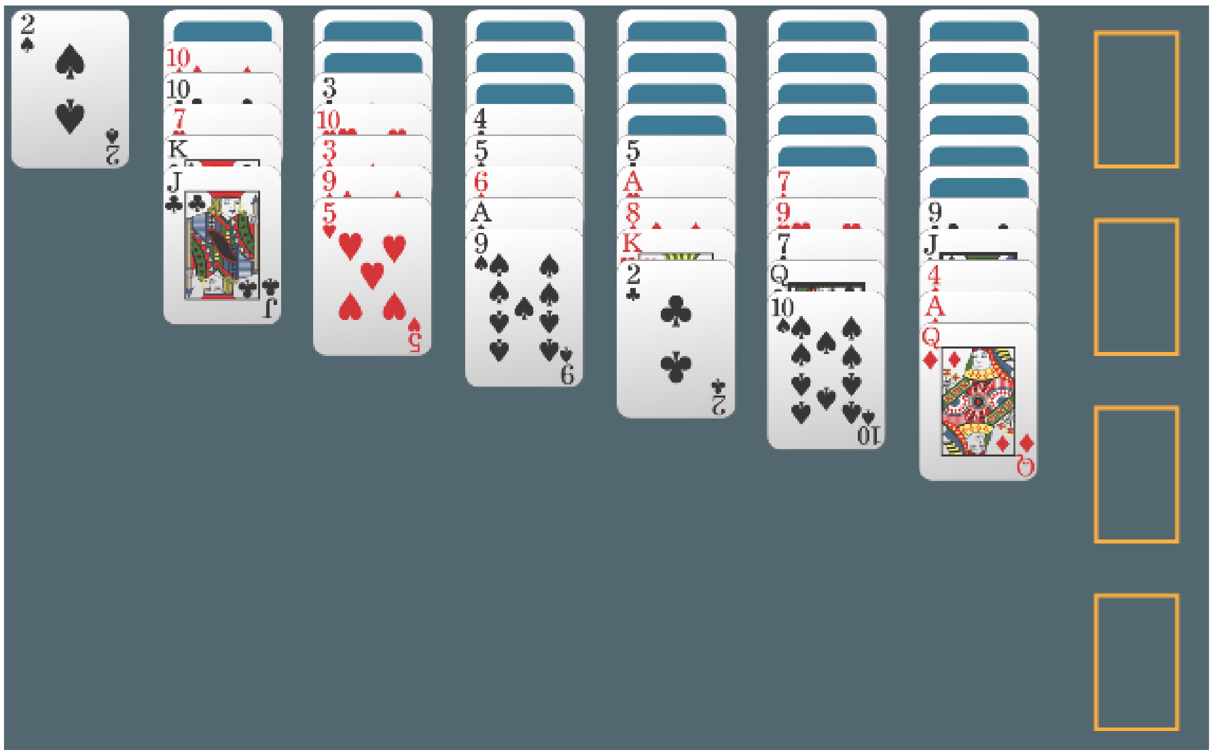


Figure 1: Example Yukon board

To play Yukon Solitaire, the entire deck is dealt into 7 columns of 1, 6, 7, 8, 9, 10, and 11 cards from left to right. 21 of these cards are dealt face down and the rest are dealt face up as shown in Figure 1. Cards must be moved into the four foundations arranged by suit and in order from Ace - King. Cards can be moved between the columns by placing a card (and the eventual cards below it) from a column to a new column, below the bottom card from that column, if that card that has a value one greater than the card that we want to move but is not of the same suit (it can also have the same color but from a different suit). If there are no cards below a hidden card on a column, then the card becomes visible. Once all cards are moved to the foundations, the player wins.

Functional requirements

Basic/support functions

- R1: LD. Load file, this function will allow us to specify a filename, and if the file exists, return the file, which contains a deck of cards that is saved in that file.
    - R1a: If the specified filename does not have a corresponding file/deck saved, create, give an error message that the file does not exist
    - R1b: If no filename is specified, create a new unshuffled deck of cards(first we get all the clubs from Ace (A) to King (K), then we get all the diamonds from A to K, then all the hearts from A to K and finally all the spades from A to K), and the return message should be OK.

- R2: SW. It shows all the cards in the order they are placed in the deck, they are placed on the gameboard like you would read a book from left to right, and then after placing a card in C7, will iterate back to C1, until all 52 cards have been placed.
    - R2a: if there is no deck loaded, return an error message.

- R3:SL. This command shuffles the cards in an interleaved manner. It first splits the deck in 2 piles of cards
    - R3a: If the optional parameter <split> is specified, then we put the number of cards from the top of the deck, specified by this parameter, into one pile and the rest into the second pile. The parameter <split> should be a positive integer smaller than the number of cards in the deck.

- R6: SR. This command shuffles the cards in a random mannerWe remove the top card from the unshuffled pile, and we add it in a random position in a new pile called the shuffled pile. We keep doing this until we finish all the cards from our pile. The shuffled pile becomes our current card deck.

- R7 SD:Saves the cards from the current card deck to a file specified by the parameter <filename>. If parameter <filename> is not specified, then use the default filename “cards.txt”. The file stores the cards, one card per line, so there will be 52 lines.

- R8 QQ: Command that exits the program.

- R9 P: This command starts a game using the current card deck. It puts the game in the PLAY phase. The cards are dealt from left to right (from column 1 to column 7), one row at a time. All the previous commands that are available for the STARTUP phase are not available in the PLAY phase and the user should get an error message "Command not available in the PLAY phase".

- R10 Q: Command that quits the current game and goes back to the STARTUP phase. The memory still contains the deck of cards used to play the game that we are quitting. So, if we use the command P again after Q, we basically restart the last game.

Cards/deck

- R11: Single deck of cards consisting of cards in the range, Ace, 2,3,4,5,6,7,8,9,10, Jack, Queen, king with the suits spade, diamond, clubs and hearts. Which will give us a combined playing card deck with 52 cards.

- R12: When we are making an unshuffled deck the order of the suits should be Clubs, Diamonds, hearts, and finally all the spades.

Board

- R13: The game board should appear before displaying the cards, there should be raw columns with names from C1-C7, and foundations are labeled from F1-F4. There is example of this in appendix 1.

- R14: In the bottom of our gameboard, we will have 3 lines: Last Command (Displays the last command that was inserted at the input prompt), Message(The message we've got from running the last command. It could be an error message if there was a problem running the last command or OK if the command was executed without error.), and Input(prompt called "Input >" where we can input new commands)

- R15: Cards should be shown when they are face up, with the first character being its value (2-9, 10, jack, queen, and king), and the second character being the suit(C- Clubs, H-hearts, D-diamonds and S-spades)

- R16: If the card is not visible, then we just use parenthesis: []

Game

- R17 Game Moves/Game: This requirement has been removed since we filled 3 pages on its own, and we did not implement it or any of its sub-requirements.

Implemented and requirements

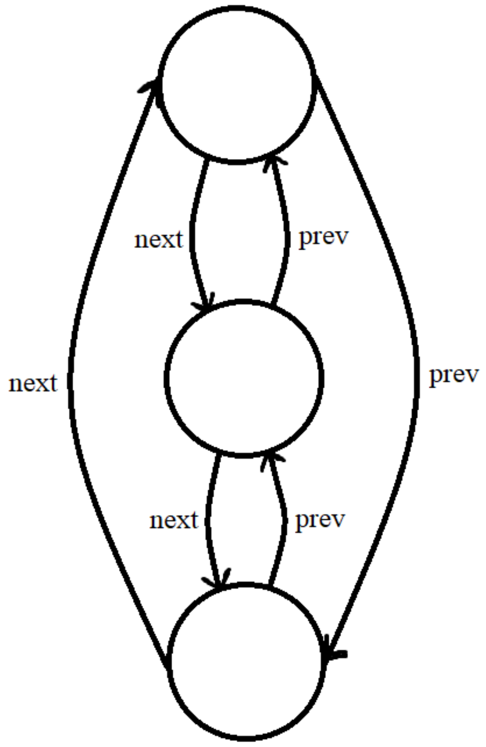
	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17
Implemented		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
Not implemented																	X
Implemented but missing sub requirements	X																

- R1b: Has not been implemented
- R17: Has not been implemented

Design

Data Structure:

The linklist is a core part of our program since it is used in various functions.
We chose to make double linked-lists for our nodes. Which means that the list is not linear, but rather circular. The first nodes' previous pointer points to the last node, and the last nodes' next pointer points to the first node, illustrated below:

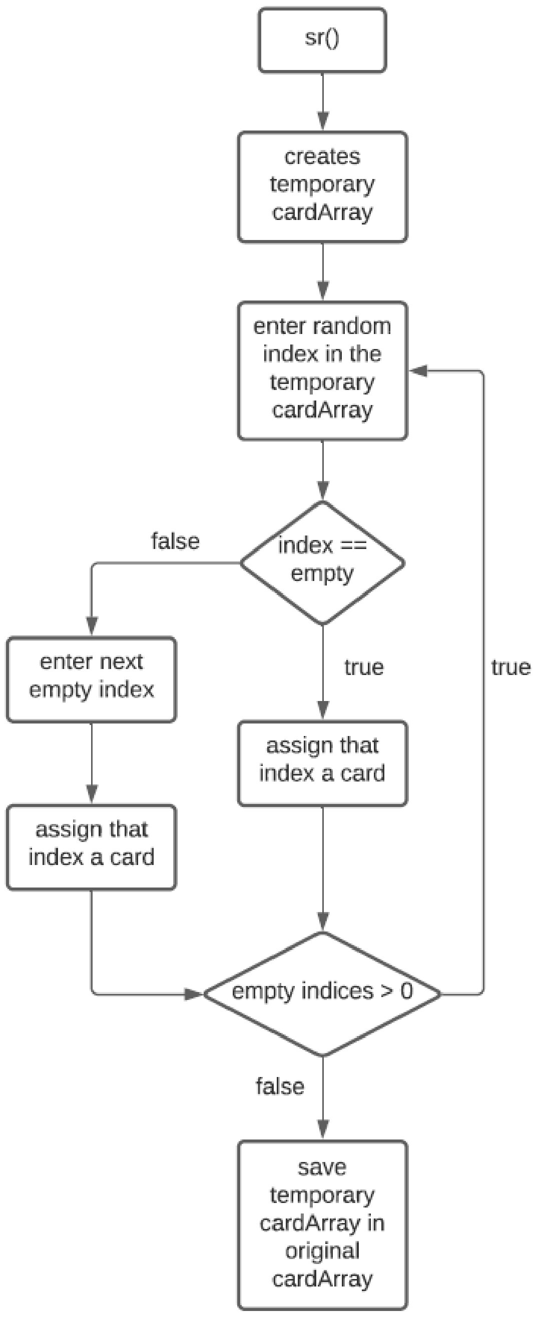


Every node has a card connected to it, also the head node. The head nodes card values are special since they don't repræsent a real card, but instead have values that can be used by the program to determine if the node it is working on is the head node or not.
We use an array of nodes to store the head nodes in. Every head node in the array repræsent a column in the card game. Since we use a circular double linked list it is fast and easy to find the last card in the column. The last node in a column is the headNodes previous node.

We have implemented various functions to help manage the linkedlist. In those where it is relevant we use malloc for dynamic memory assignment. Therefore when a node is deleted the node will also be removed from memory

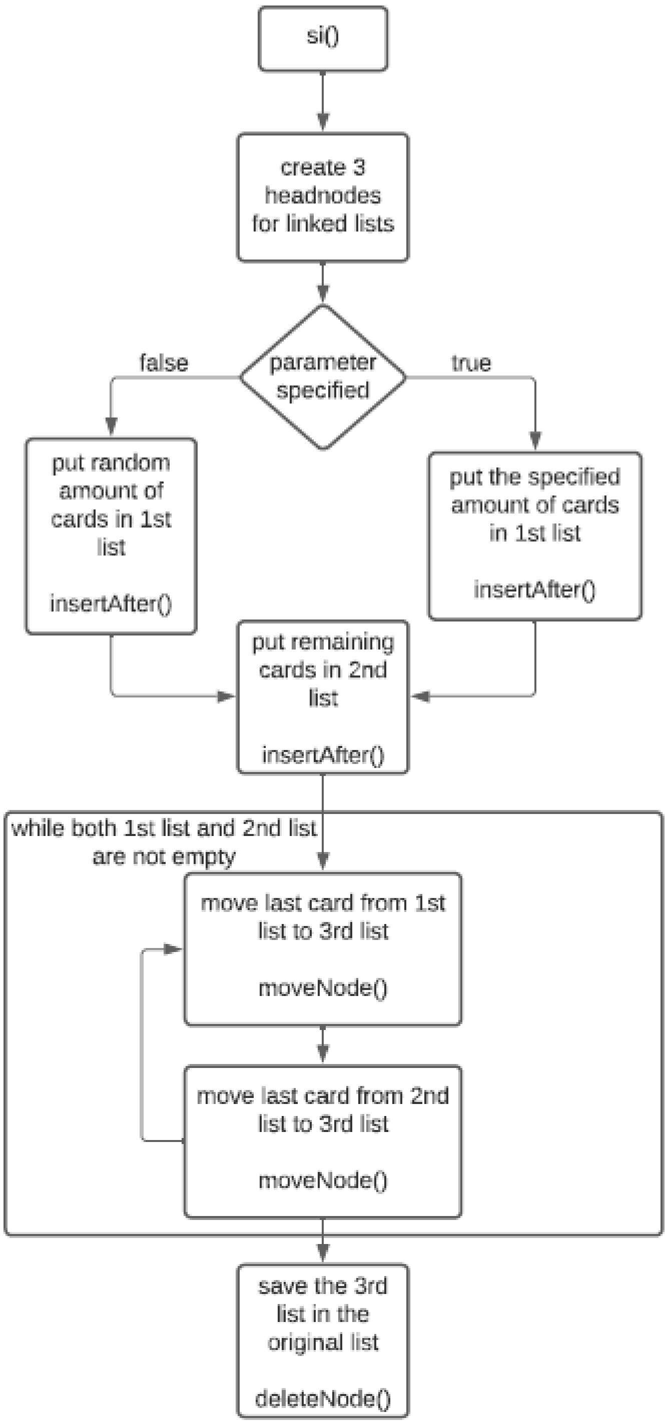
SR-function:

The SR-function shuffles the card array randomly.
It does this by first creating a new temporary and empty card array. It then enters a random (using rand() to choose the random number) index in the temporary card array, and assigns that index the first card from the original card array. It then enters another random index, and assigns the second card to that index. If the index entered is not empty, it just iterates to the next empty index in the array. When all the empty indices have been assigned a card, the temporary array is saved in the original array.



SI-function:

The SI-function splits the card deck into 2, and then shuffles them together. Where the deck is split depends on what parameter the user enters. First we create 3 head nodes so we can make 3 linked lists for our 3 piles of cards. Then, if the user enters a parameter lower than 53, we put that amount of cards into the 1st linked list with an insertAfter()-operation. If the user enters a parameter too high, or no parameter at all, we put a random amount of cards into the 1st linked with an insertAfter()-operation. Then the remaining cards are put into the 2nd linked list, also with insertAfter(). We then proceed to alternately move the last card from the 1st list into the 3rd list, and then the last card from the 2nd list into the 3rd list. We do this in a while-loop, which breaks once both 1st and 2nd lists are empty for cards. Then we copy the 3rd list (which is now shuffled) to the original list of cards, and delete the nodes that we have no need for anymore.


si()
create 3 headnodes for linked lists
parameter specified
false
true
put random amount of cards in 1st list
insertAfter()
put the specified amount of cards in 1st list
insertAfter()
put remaining cards in 2nd list
insertAfter()
while both 1st list and 2nd list are not empty
move last card from 1st list to 3rd list
moveNode()
move last card from 2nd list to 3rd list
moveNode()
save the 3rd list in the original list
deleteNode()

Implementation

Before we can talk about the implementation of our “support” functions, that we can use in our “start up phase”, we have to show how our program works in general

While in the startup phase, you should be able to use all the support functions, except for the Q function, which can only be used during the play phase.

Our startup phase is run within a Whileloop, that stops at the GETS(str) on line 98, and waits for the players input. The input is stored in str, which will

afterwards it runs through a switch-case like function, a example of this can be seen in codesnippet 2 on line 112 where the strncmp compares our variable str.

Implementation af SW

The SW function is used for showing the card deck that is currently loaded. A deck is loaded through the LD(load) function. So for the SW function to work, you first have to call the LD function in input, if you haven't done this, then you will receive an error, telling you “No cards in deck”.
The if statement checks if there is a cardDeck available by making a simple check to the first cards rank. If this statement is true. The showCardDeckFunction will be called first

```
int sw(){
   if(cardArray[0].rank>0){
       //Hvis der er kort i bunken
       createShowCardDeck();
       printCardDeck();
       return 1;
   }else{
       //Hvis der ikke er nogle kort i bunken
       return 0;
   }
}
```

The createShowCardDeck function is a simple function which first clears the linked list in headArray with the emptyLinkedList() function.
In each iteration we make sure that the card we are working with is set to faceUp so it will be visible later. After that we use the insertAfter function to create a new node and insert it in the correct column. We use the x value to make sure that iterate through the columns.

```
void createShowCardDeck(){
       //Sørger for at linkedlisterne er tomme
       emptyLinkedList();
       for (int i = 0; i < 52; ++i) {
           int x = i%7;
           cardArray[i].isFaceUp=true;

insertAfter(headArray[x].previousPointer,&cardArray[i]);
       }
}
```

PrinCardDeck()

The printCardDeck() function is a function which can be used by multiple functions in the program to print out the columns. It prints out the cards the way they are stored in the 7 linked list from the headArray.
The function is long but a part of the it has been illustrated below.
This part of the function makes sure that if the card is faceUp, the correct rank is printed out, either a number if its between 2 and 9 or a letter if its 1 or between 10 and 13.
If the card isn't visible it will print out the two brackets, []

```
if(tempCard.isFaceUp){
   //Hvis kortet ikke er skjult
   if(tempCard.rank==1){
       printf("A%c",tempCard.suit);
   }else if(tempCard.rank==10){
       printf("T%c",tempCard.suit);
   }else if(tempCard.rank==11){
     printf("J%c",tempCard.suit);
   }else if(tempCard.rank==12){
       printf("Q%c",tempCard.suit);
   }else if(tempCard.rank==13){
       printf("K%c",tempCard.suit);
   }else{
         printf("%d%c",tempCard.rank,tempCard.suit);
   }
}else{
   //Hvis kortet er skjult
   printf("[]");
}
```

Implementation af double linked list

We have implemented a double linked list in our c program which we use as the main data structure for the card game. We have 7 linked lists which are static and are used for the 7 columns in the card game. Every one of those linked lists has a head node with a node card. The card for the head node has the rank of -1 which is used in different places in the program to determine the head node, when it's looped through the list.
With the headArray it's easy to always find the last node in the linked list, since we can always call headArray[x].previousPointer and by that way get the last node in the list.

To make it easier to work with the linked list we have implemented 3 different functions which can either insert a new node, move a node, delete a node or empty a linked list.

Implementation af insertAfter()

This is our insertAfter function. It takes a previous node and inserts a new node after that.
Since our double linked list is circular every node will always have a previous pointer and a next pointer.
In this function we use malloc when we create a new code, so we later can delete it from memory. A node is also given an address where the pointer of the nodeCard should point to.
This node makes sure that all the necessary pointers are changed and we have tested it thoroughly since it's a big part of our program

```
void insertAfter(struct node *prevNode,struct card*
newCardPointer){
      if(prevNode==NULL){
           printf("Tidligere node er NULL");
           return;
       }
      //Opretter en ny node og allokere plads i memory til den
      struct node* newNode=(struct node*)malloc(sizeof (struct
node));

      newNode->nodeCard=newCardPointer;
      newNode->nextPointer=prevNode->nextPointer;
      //nextPointer i noden før, peger nu hen til den nye nuværende
node
      prevNode->nextPointer=newNode;
      //I den nye, nuværende node, peger prevNode nu hen til noden
før
      newNode->previousPointer=prevNode;
      if(newNode->nextPointer!=NULL){
           newNode->nextPointer->previousPointer=newNode;
      }
}
```

Implementation af deleteNode()

Delete node will, as the name says, delete a node from the linked list it is connected to.
It will change the previous node's nextpointer and the next node's previous pointer so the linked list still is complete and circular.
It also uses the free() method which makes sure that the node is freed from memorry so the program doesn't use unnecessary space.

```
void deleteNode(struct node* deleteNode){
   //Sletter en node og samtidig sørger
   //Sørger for at den næste node får ny korrekt previous pointer

deleteNode->nextPointer->previousPointer=deleteNode->previousPointer
;
   //Sørger for at den tidligere node får ny korrekt next pointer
   deleteNode->previousPointer->nextPointer=deleteNode->nextPointer;
   //Frigiver noden fra hukommelsen (malloc)
   free(deleteNode);
}
```

Implementation af moveNode()

Move nodes take two nodes as input.
The currentNode input is the node we want to move.
The function moves all the relevant pointers from the previous node and the next node so they now point to each other.
The node that are moved will be moved to the placement after the previousNode argument. The function moves all the relevant points in the linked list the node are moved to.

```
void moveNode(struct node* currentNode, struct node*
previousNode){
       //Flytter en node

currentNode->previousPointer->nextPointer=currentNode->nextPointer
;

currentNode->nextPointer->previousPointer=currentNode->previousPoi
nter;

       currentNode->previousPointer=previousNode;
       currentNode->nextPointer=previousNode->nextPointer;

       previousNode->nextPointer=currentNode;

       currentNode->nextPointer->previousPointer=currentNode;
}
```

Implementation af emptyLinkedList()

The function emptyLinkedList empties all the linked lists in the headArray.
To empty a linked list it uses the deleteNode function. The if statement checks if the node it wants to delete is the headNode. If it is the headNode but instead increment the counter by one. Everytime the while loop starts the counter is set to 0. At some point the counter will be 7, which is the number of linked lists in headArray, and the loop will then end, and also the function.

```
void emptyLinkedList(){
   int counter =0;
       while (counter!=7){
           counter=0;
           for (int i = 0; i < 7; ++i) {

if(headArray[i].previousPointer->nodeCard->rank!=-1){
                   deleteNode(headArray[i].previousPointer);
               }else{
                   counter=counter+1;
               }
           }
       }
}
```

Implementation af gameCardDeck()

The gameCardDeck function is used to create the linked lists used for when the game is played. The function is called when the game enters the playing mode.
The code below is not the whole function but only the most important part of it.
First it will empty the linked list. If no nodes are in the linked list,except the head node, nothing will happen.
The function makes sure that the card that has to be invisible to the players(face down) is set to that. It builds up the linked list with the insertAfter function. We used loops for some of this function, but other places it was easier just to input each node manually. This is the part of the code that is not shown. In the end it makes no difference to either runtime or functionality of the code.

```
void createGameCardDeck(){
//Tømmer linked listerne:
emptyLinkedList();
//Laver linked listen der skal bruges til spillet;
//Det første element tiløjes
insertAfter(headArray[0].previousPointer,&cardArray[0]);
   int tempI;

   //Hvis der tages udgangspunkt i et ublandet kortdæk(Den vidst i
opgave beskrivelsen) tilføjes kortene op til QH.
   //De kort der skal være faceDown bliver sat til det
   for (int i = 1; i < 7; ++i) {
       tempI=i;
       for (int j = 0; j < 6; ++j) {
           //Hvis kortet ikke er faceUp bliver den sat til false
           if(j<i){
               cardArray[tempI].isFaceUp=false;
           }
           //Tilføjer kortene til vores linked list

insertAfter(headArray[i].previousPointer,&cardArray[tempI]);
           tempI=tempI+6;
       }
   }

…..*MORE CODE*....
}
```

Tests

Test case	Expected result	Actual result	Remarks
Load a deck with a specified filename	That we get a deck with the specified filename	We get the specified card deck, but you can't just use the specified file name	You have to use the entire path to the file, to get it.
Load a deck without a specified filename	We get an unshuffled deck of cards	We get an unshuffled deck of cards	
Use the Show deck(SW) with an unshuffled deck	It will show a unshuffled deck in the right order	It showed a unshuffled deck in the right order	
Use the split function to shuffle the deck without specifying the parameter	We tested the random part separately to check if every number was different. We printed out the randnumber and did the split shuffle in hand	it did split just as we expected from the random number.	
Shuffle (SR) the deck in a random manner.	Will shuffle the deck	The deck is shuffled	
Save your current deck (SD) with a specified parameter	Save your current deck with a specified parameter	The deck is saved, but you can’t just write the specification, you need the file path too, for where you want to save it	We can find this file, and see that is store a card per line, aswell
Save your current deck (SD) without a specified parameter	Saved in a file with the name cards.txt	It is saved in a file called cards.txt	We don't know where it is saved in the file structure, so we cant open in it from the file explorer and check it
Use command QQ to quit the program	The program will be exited	We exit the program just as expected	
Use command P to enter The play phase, and use a Startup phase command	You will enter play mode	You enter play mode, and cant use any functions from the startup phase	The game should deal cards to you, and start the

			game, but we haven't implemented the game functions
Use the Q command, and while in startup-phase	It should do nothing	It does nothing, expect say it’s a unknown command	
Use the Q command while in playing-phase, afterwards check if you can use the startup commands	It should put you back in startup-mode, and allow you to use startup commands again	it does as expected	

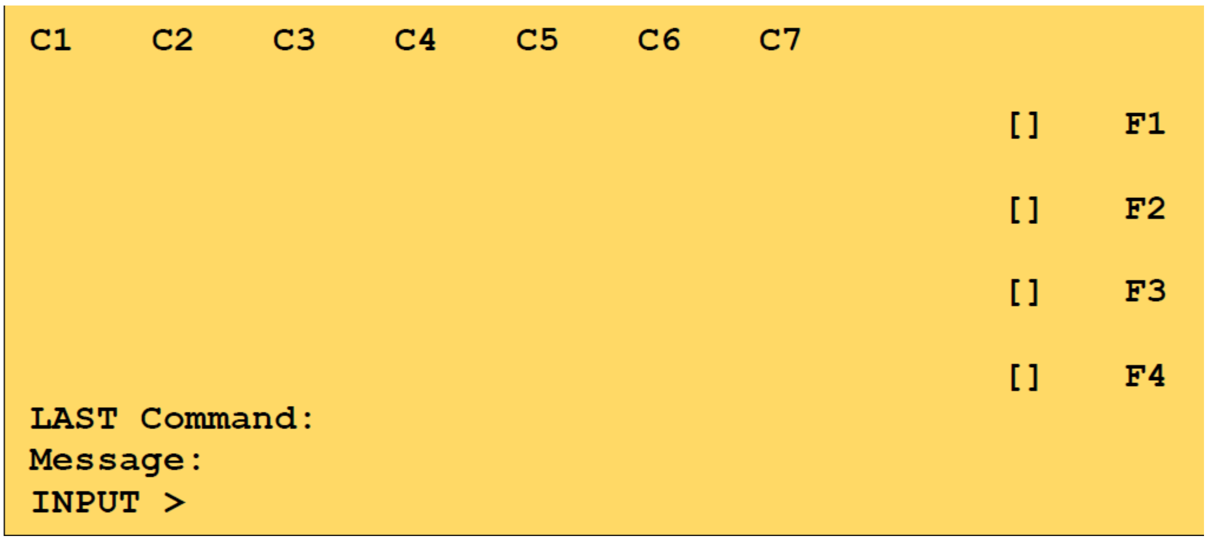
Conclusion

We have implemented most of the required functions for the program. We have made a good and robust data structure, using a double linked list where it is appropriate and makes sense to the given problem that is being solved. All the necessary functions for a double linked list have been implemented and are used for various functions. The user interface works as intended without any unexpected crashes. Further development of the game would be easy and fast to implement because of the base functionality with the linked list. Implementing card moves in the game would only require implementing input from the user and using the already made move function. Our conclusion is therefore that even if we didn’t have the time to fully implement the game, our program is in a state where it is robust with lots of functionality. Lots of functions are already fully implemented and most of the functions needed for the final game are already implemented, especially the difficult parts.

Contributions

	Sander Eg Albeck Johansen	Mikkel Hillebrandt Thorsager	Christian Kyed
Kode: Implementation af LD og SD	X		X
Kode: Implementation af SI		X	X
Rapport: Analyse	X	X	
Rapport: Krav	X	X	
Rapport Design		X	X
Rapport: Implementering	X		X
Rapport: Test	X		
Kode: Implementation af SW, SR			X
Kode: Implementation af UI			X
Kode: Implementation af datastruktur (Double linked list)			X
Kode: Implementation af gameCardDeck			X
Video			X

Appendix



```
C1    C2    C3    C4    C5    C6    C7

                                             []    F1

                                             []    F2

                                             []    F3

                                             []    F4
LAST Command:
Message:
INPUT >
```

Appendix 1: UI example