Relatório do trabalho de Estrutura de Dados I

André Neves Prestes e Victor David Lima

T01 – Prof. Vinícius Fernandes Soares

## Sumário

Visão geral ----------------------------------------------------------------------------------------------------- 2

Estratégias implementadas ------------------------------------------------------------------------------- 15

Bibliotecas ----------------------------------------------------------------------------------------------------- 19

Custo computacional --------------------------------------------------------------------------------------- 19

# Visão geral

A implementação do trabalho foi realizada de modo que facilitasse seu uso e manutenção e entendimento do código. Para atender ao primeiro ponto, foi tomada a decisão de uma interface gráfica, buscando a maior semelhança possível com a realidade, dentro das limitações dos caracteres da tabela ASCII, pois deve rodar em um terminal. Essa interface fornece ao usuário suas cartas, as que estão na mesa, animações para comprar e jogar cartas e os demais dados, deixando o jogo o mais claro e orgânico possível. Também foram tomados os cuidados de “atrasar” algumas operações, como a de distribuição de cartas e as jogadas do “CPU”, tornando o jogo mais orgânico e dando o tempo necessário para que o jogador acompanhe tudo que acontece a cada rodada. Para realizar esses atrasos, foram utilizadas as funções “sleep” e “usleep”, em segundos e microsegundos, respectivamente.

Com o objetivo de atender ao segundo ponto, foi dada prioridade à clareza do código, com funções bem comentadas, com nomes intuitivos e utilizando instruções mais simples, claro que sem deixar a otimização de lado. Além disso, a divisão do código foi feita em duas bibliotecas, “Jogo” e “Cartas”, separando bem as funções e sua área de atuação. Todas as funções foram testadas incansavelmente, de modo que funcionassem sem erro algum e atendessem do melhor modo possível ao que foi requisitado, logo, o trabalho funciona perfeitamente e não apresenta vazamentos de memória, com consumo de 5.456 bytes (quando apenas uma partida é jogada).

A seguir serão citadas as funções e explicadas a lógica utilizada para criá-las, conforme requisitado.

### Structs:

Para a implementação do Jogo proposto no trabalho foi necessária a criação de quatro structs, sendo elas:

typedef struct

{

char id;

char naipe;

int valor;

int posicao;

} TipoCarta;

typedef struct TipoCelula

{

TipoCarta Carta;

TipoApontador Prox;

} TipoCelula;

typedef struct TipoCelula \*TipoApontador;

typedef struct {

TipoApontador Primeiro, Ultimo;

} TipoBaralho;

typedef struct

{

TipoCarta cartasJogador[maxTamMao];

int Primeiro, ultimo;

} TipoMao;

1. **TipoCarta**: Criada para representar todas as cartas que aparecerão no jogo.

* **id**: Representa os “números” das cartas (‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘J’, ‘Q’, ‘K’, ‘A’), sendo seu tipo char pois entre “números” também há letras, como mostrado acima, assim facilitando na hora de imprimi-los nas cartas.
* **naipe**: Variável do tipo char que representa os naipes das cartas (Ouro, Espadas, Copas e Paus). Mais a frente no script as letras para simbolizar cada um desses naipes são ‘o’ - Ouro, ‘e’ - Espadas, ‘c’ - Copas e ‘p’ - Paus.
* **valor**: Variável do tipo int com os valores que cada carta tem de acordo com as regras da bisca ( de ‘2’ a ‘6’ - 0 pontos, ‘J’ - 2 pontos, ‘Q’ - 3 pontos, ‘K’ - 4 pontos, ‘7’ - 10 pontos e ‘A’ - 11 pontos).
* **posição**: Variável do tipo int que apresenta as posições relativas a ordem das cartas em bisca ( ‘2’ = 0, ‘3’ = 1, ‘4’ = 2, ‘5’ = 3, ‘6’ = 4, ‘J’ = 5, ‘Q’ = 6, ‘K’ = 7, ‘7’ = 8 e ‘A’ = 9), facilitando na hora de verificar quem ganhou a rodada, sendo que quanto maior a posição maior é a probabilidade desta carta ganhar.

1. **TipoCelula:** Criado para representar uma carta no baralho, posteriormente usado para fazer a lista encadeada que será o baralho.

* **Carta**: Variável do TipoCarta que apresentará as informações de uma carta individual.
* **Prox**: Variável do TipoApontador (é o nome do TipoCarta após ser renomeado e transformado em um ponteiro usando o typedef struct TipoCelula \*TipoApontador) que apontará para a próxima carta do baralho.

1. **TipoBaralho:** struct criado para apontar para a primeira e última carta do baralho (sentinelas), para facilitar o manuseio das cartas do baralho.

**Primeiro**: Variável do TipoApontador que aponta para a primeira carta.

* **Ultimo**: Variável do TipoApontador que aponta para a última carta.

1. **TipoMao:** struct criada para representar a mão tanto do jogador quanto dos CPU’s. Usada como arranjo devido à facilidade de implementação e uso, como o tamanho das mãos é pequeno, não afeta tanto o uso de memória.

* **cartasJogador[maxTamMao]**: Array do TipoCarta que tem tamanho maxTamMao, definido no script como três pois é o número máximo possível de cartas na mão, que recebe as cartas do baralho.
* **Primeiro**: Variável de tipo int que tem sempre como valor a posição 1 do array.
* **Ultimo**: Variável do tipo int que tem o valor sempre igual ao número de cartas na mão mais um, sendo que se este valor é igual ao Primeiro quer dizer que o array está vazio.

### Funções:

* Para viabilizar a interface gráfica, foi criada a função gotoxy, que recebe um valor x e outro y, e desloca o “cursor” do terminal para a posição (x,y) da janela, permitindo editar exatamente o caracter que será exibido nessa posição.

As outras funções que printam as demais informações na tela usam a gotoxy, demonstrando sua importância para o funcionamento adequado do jogo.

void gotoxy (int x, int y)

{

printf(“\033[%d;%dH”, y+1, x+1);

}

* Visando inserir uma carta no baralho, tomamos a decisão de inseri-la no final, pois facilitaria na implementação de funções como criaBaralho, citada abaixo, e a corta, que retira a carta da metade do baralho e insere o monte com posição < 20 após o outro.

Para atribuir as cartas ao baralho, ao invés de inicializar cada posição do baralho, criamos a função criaBaralho, para fazer isso quando necessário:

void criaBaralho (TipoBaralho \*Baralho)

{

//Fazendo o Baralho vazio

FLVazia(Baralho);

//Variável usada para o switch, visando atribuir as características às cartas

int i;

//Variável usada para percorrer a lista de naipes, avançando a cada 10 cartas para inserir o naipe correspondente

int posNaipes;

//Vetores contendo os naipes, ids e valores das cartas

char naipes[] = {'o', 'e', 'c', 'p'};

char id[] = {'2', '3', '4', '5', '6', 'J', 'Q', 'K', '7', 'A'};

int valores[] = { 0, 0, 0, 0, 0, 2, 3, 4, 10, 11};

int posicao[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

//Criando TipoCarta auxiliar, para atribuir as cartas à lista

TipoCarta aux;

for (posNaipes = 0; posNaipes < 4; posNaipes++)

{

for (i = 0; i < 10; i++)

{

aux.id = id[i];

aux.naipe = naipes[posNaipes];

aux.valor = valores[i];

aux.posicao = posicao[i];

InsereBaralho(aux, Baralho);

}

}

}

* Com o intuito de retirar uma carta do baralho, criamos a função retiraCarta, que devido à lista ter sido implementada com sentinelas, tivemos que dar um funcionamento diferente em duas situações, quando a carta retirada é a primeira, e para as demais cartas:

//Retira uma carta da posição "posicao" do baralho, retorna tal carta e da free na original do baralho

TipoCarta retiraBaralho (TipoBaralho \*Baralho, int posicao){

//Declaração de variáveis auxiliares para percorrer e realizar substituições na lista encadeada

int i = 0;

TipoApontador p, aux;

TipoCarta CartaAux;

//Quando se deseja retirar a primeira Carta

if (posicao == 0)

{

aux = Baralho -> Primeiro;

Baralho -> Primeiro = aux -> Prox;

CartaAux = aux -> Carta;

free (aux);

return (CartaAux);

}

else

{

//Retirando as Cartas das demais posições

for (p = Baralho -> Primeiro; p -> Prox != NULL; p = p -> Prox)

{

if (posicao == i + 1)

{

aux = p -> Prox;

CartaAux = aux -> Carta;

p -> Prox = aux -> Prox;

free (aux);

return (CartaAux);

}

//Incrementando o valor de i para encontrar a posição desejada

i++;

}

}

}

* Objetivando embaralhar as cartas, decidimos por utilizar a função srand, para garantir que os números gerados sejam sempre aleatórios, provendo ao jogador uma partida completamente diferente a cada execução do programa. Essa decisão fez necessária a inclusão da biblioteca time.h, para o correto funcionamento da srand. Optamos por percorrer todas as cartas do baralho para garantir que seriam trocadas na maior parte dos casos.

//Embaralha o baralho usando a funçao srand, assim pegando uma carta com a função retiraBaralho na posição dada por rand e inserindo na ultima posição com a função InsereBaralho

void embaralha (TipoBaralho \*Baralho)

{

//Usando srand, para garantir a aleatoriedade a cada execução

srand(time(NULL));

//Declaração das variáveis auxiliares para realizar as trocas no Baralho

TipoCarta aux;

int posicao;

for (int i = 0; i < 40; i++)

{

posicao = rand() % 40;

aux = retiraBaralho(Baralho, posicao);

InsereBaralho( aux, Baralho);

}

}

* Para gerenciar a distribuição de cartas na última rodada, criamos a função NumeroCartasComecoEFimDois, que verifica se a quantidade de cartas do baralho é igual a 0, sendo que, se isso for verdade, incrementa a variável aposZero, que será utilizada em outras funções para verificar se é a última rodada, caso seja, habilita a comparação para verificar quem ganhou e printar o resultado e desabilita as funções de dar cartas. A função NumeroCartasComecoEFimQuatro segue a mesma lógica, porém para quatro jogadores.

//Verifica se o numero de cartas no baralho é igual a zero, se ele for igual muda a variavel numeroDeCartasMao assim desabilitando a função de dar cartas e verifica se quem ganhou

void NumeroCartasComecoEFimDeJogoDois(int numeroDeCartasBaralho, int \*numeroDeCartasMao, int \*aposZero, int pontosCPU, int pontosJogador, int \*fim, TipoBaralho \*Baralho){

if (numeroDeCartasBaralho == 0)

{

switch (\*aposZero)

{

case 0:

\*aposZero = (\*aposZero) + 1;

break;

default:

\*numeroDeCartasMao = (\*numeroDeCartasMao) - 1;

\*aposZero = (\*aposZero) + 1;

break;

}

}

if (pontosCPU >= 60 || pontosJogador >= 60)

{

DeletaIntervalo(20, 5, 130, 30);

if(pontosCPU > pontosJogador)

{

PrintYOULOSE(fim, Baralho);

}

else

{

PrintYOUWIN(fim, Baralho);

}

}

if (\*aposZero == 4)

{

DeletaIntervalo(20, 5, 130, 30);

if(pontosCPU > pontosJogador)

{

PrintYOULOSE(fim, Baralho);

}

else

{

PrintYOUWIN(fim, Baralho);

}

}

}

* A fim de controlar a distribuição a cada rodada, foi criada a função ProximaRodadaDois, em que é verificado quem começa na próxima rodada, assim, distribuindo as cartas por tal jogador, e após isso é utilizada a função NumeroCartasComecoEFimDois para verificar se a função de distribuição da próxima rodada será desabilitada ou não. A função ProximaRodadaQuatro segue a mesma lógica, porém aplicada a quatro jogadores.

//Começo da proxima rodada

void ProximaRodadaDois(char \*comando, char modoDebug, TipoBaralho \*Baralho, TipoMao \*MaoJogador, TipoMao \*MaoCPU2, int comeca, int \*numeroDeCartasBaralho, int \*numeroDeCartasMao, int \*aposZero, int pontosJogador, int pontosCPU, int \*fim){

if(comeca == 0){

DarCartaPeloJogadorDois(MaoJogador, MaoCPU2, Baralho, numeroDeCartasBaralho, \*aposZero);

if(modoDebug == 'q'){

DeletaIntervalo(0, 3, 14, 40);

imprimeBaralho (\*Baralho, 0, 3);

DeletaIntervalo(14, 0, 43, 2);

imprimeMao(\*MaoCPU2, 20, 0);

}

NumeroCartasComecoEFimDeJogoDois(\*numeroDeCartasBaralho, numeroDeCartasMao, aposZero, pontosCPU, pontosJogador, fim, Baralho);

gotoxy(20, 45);

gotoxy(55, 39);

printf("Sua vez! ");

DeletaCaracter(60, 42);

gotoxy(60, 42);

scanf(" %c", comando);

} else{

DarCartaPeloCPUDois(MaoJogador, MaoCPU2, Baralho, numeroDeCartasBaralho, \*aposZero);

if(modoDebug == 'q'){

DeletaIntervalo(0, 3, 14, 40);

imprimeBaralho (\*Baralho, 0, 3);

DeletaIntervalo(14, 0, 43, 2);

imprimeMao(\*MaoCPU2, 20, 0);

}

NumeroCartasComecoEFimDeJogoDois(\*numeroDeCartasBaralho, numeroDeCartasMao, aposZero, pontosCPU, pontosJogador, fim, Baralho);

gotoxy(20, 45);

gotoxy(55, 39);

printf("Vez do CPU!");

sleep(1);

}

}

* A função criada para gerenciar a partida de dois jogadores foi a LogicaMesaDois, que apura quem ganhou a rodada, quem começa a próxima, se o jogo terminou ou não, em qual posição da listaMesa é inserida cada carta, o tempo necessário para as animações acontecerem e se o modo Debug foi acionado ou não. A função LogicaMesaQuatro segue o mesmo raciocínio, porém implementado para quatro jogadores.

//Logica de jogo para mesa de dois jogadores

void LogicaMesaDois(char modoDeJogo, char \*comando, TipoBaralho \*Baralho, char modoDebug){

int pontosJogador = 0, pontosCPU = 0, numeroDeCartasBaralho = 40, numeroDeCartasMao = 3, comeca = 0, monteJ = 0, monteCPU2 = 0, mesaCPU2, aposZero = 0, fim = 0, Jogando = 0;

TipoCarta trunfo, listaMesa[2];

TipoMao MaoJogador, MaoCPU2;

PrimeiraMesaDois(&trunfo, &MaoJogador, &MaoCPU2, Baralho, &numeroDeCartasBaralho);

if(modoDebug == 'q'){

imprimeBaralho (\*Baralho, 0, 3);

imprimeMao(MaoCPU2, 20, 0);

}

gotoxy(60, 42);

scanf(" %c", comando);

while (fim != 1 && \*comando != 'v'){

switch(comeca){

case 0:

defaultMesa(comando, numeroDeCartasMao);

Jogando = 0;

listaMesa[0] = SeletorMesa(&MaoJogador, comando, numeroDeCartasMao, 54, 22, Baralho);

gotoxy(55, 39);

printf("Vez do CPU!");

sleep(1);

Jogando = 1;

ModoDeEscolhaCPU(&mesaCPU2, modoDeJogo, MaoCPU2, listaMesa, trunfo, comeca, Jogando, numeroDeCartasMao);

listaMesa[1] = TiraCartaEPrinta(&MaoCPU2, mesaCPU2, 59, 22, numeroDeCartasMao, 2);

funcComparaDois(listaMesa, trunfo, &comeca, &pontosJogador, &pontosCPU, &monteJ, &monteCPU2);

sleep(1);

MudaValoresDois(numeroDeCartasBaralho, pontosJogador, pontosCPU, monteJ, monteCPU2);

sleep(1);

ProximaRodadaDois(comando, modoDebug, Baralho, &MaoJogador, &MaoCPU2, comeca, &numeroDeCartasBaralho, &numeroDeCartasMao, &aposZero, pontosJogador, pontosCPU, &fim);

break;

case 1:

Jogando = 1;

ModoDeEscolhaCPU(&mesaCPU2, modoDeJogo, MaoCPU2, listaMesa, trunfo, comeca, Jogando, numeroDeCartasMao);

listaMesa[1] = TiraCartaEPrinta(&MaoCPU2, mesaCPU2, 54, 22, numeroDeCartasMao, 2);

gotoxy(55, 39);

printf("Sua vez! ");

DeletaCaracter(60, 42);

gotoxy(60, 42);

scanf(" %c", comando);

defaultMesa(comando, numeroDeCartasMao);

if(\*comando == 'v'){

freeBaralho (Baralho);

criaBaralho (Baralho);

break;

}

Jogando = 0;

listaMesa[0] = SeletorMesa(&MaoJogador, comando, numeroDeCartasMao, 59, 22, Baralho);

funcComparaDois(listaMesa, trunfo, &comeca, &pontosJogador, &pontosCPU, &monteJ, &monteCPU2);

sleep(1);

MudaValoresDois(numeroDeCartasBaralho, pontosJogador, pontosCPU, monteJ, monteCPU2);

sleep(1);

ProximaRodadaDois(comando, modoDebug, Baralho, &MaoJogador, &MaoCPU2, comeca, &numeroDeCartasBaralho, &numeroDeCartasMao, &aposZero, pontosJogador, pontosCPU, &fim);

break;

}

}

if(fim == 1){

gotoxy(40, 42);

printf("Escolha 'v' para ir para o menu : ");

scanf(" %c", comando);

}

}

* Com o objetivo de realizar a comparação entre as cartas da mesa, para decidir quem ganhou a rodada, foram criadas 3 funções secundárias e uma função primária, de modo que as funções secundárias foram:

1. **maior:** que percorre a listaMesa e encontra a carta com maior número na variável posição do struct TipoCarta, e retorna sua posição na listaMesa.
2. **achaTrunfosECompara:** que percorre a listaMesa, comparando todos os naipes das cartas com o naipe do trunfo, e se houver ao menos um, retorna a posição na listaMesa do maior.
3. **naipesIguaisAoPrimeiro**: que percorre a listaMesa, comparando todos os naipes das cartas com o naipe da primeira carta jogada, e se houver ao menos um, retorna a posição na listaMesa do maior.

Já a função primária, funcComparaDois, apenas ordena as funções secundárias seguindo as prioridades de comparação descritas nas regras do jogo, sendo elas:

maior < naipesIguaisAoPrimeiro < achaTrunfosECompara

Vale ressaltar que a funcComparaQuatro segue o mesmo raciocínio, porém adaptado para quatro jogadores.

//Compara duas cartas, primeiro verifica se existe algum trunfo, se existir um ele ganha, dois ele compara e maior ganha, se nao existir ele ve que jogou primeiro e se o naipe

//do primeiro for igual ao do segundo entao ele verifica o maior, se nao for igual o primeiro ganha

void funcComparaDois(TipoCarta \*listaMesa, TipoCarta trunfo, int \*comeca, int \*pontosJogador, int \*pontosCPU, int \*monteJ, int \*monteCPU)

{

int ganhou;

//Procura e compara as cartas com naipe igual ao do trunfo

ganhou = achaTrunfosECompara(listaMesa, trunfo, 2);

//Se nenhuma carta de naipe igual ao do trunfo foi encontrada (ganhou = -1)

if (ganhou == -1)

{

//Procura a maior carta de naipe igual à primeira jogada

ganhou = naipesIguaisAoPrimeiro(listaMesa, 2, \*comeca);

}

switch(ganhou)

{

/\*Incrementa o número de cartas do monte do vencedor em 2, sinalizando que duas cartas foram obtidas

Define quem começa a próxima rodada baseado em quem ganhou

Incrementa os pontos de acordo com o valor das cartas obtidas para o monte de cada jogador\*/

//Caso o jogador ganhe

case 0:

\*pontosJogador = (\*pontosJogador) + (listaMesa[0].valor) + (listaMesa[1].valor);

\*comeca = 0;

\*monteJ = (\*monteJ) + 2;

break;

//Caso o CPU ganhe

case 1:

\*pontosCPU = (\*pontosCPU) + (listaMesa[0].valor) + (listaMesa[1].valor);

\*comeca = 1;

\*monteCPU = (\*monteCPU) + 2;

break;

}

}

* Para o menu principal, criamos a função SeletorMenuPrincipal, que, com base no comando inserido pelo usuário, chama as funções relacionadas a cada função, contidas nas bibliotecas jogo e carta.

//Switch do menu principal

void SeletorMenuPrincipal(){

TipoBaralho Baralho;

TipoCarta trunfo1;

char modoDebug, escolha, \*comando;

comando = &escolha;

PrintMenuFora();

criaBaralho (&Baralho);

PrintMenuConteudoPrincipal();

gotoxy(40, 39);

printf("Escolha sua opcao : ");

scanf(" %c", comando);

while(\*comando != 's'){

switch (\*comando){

//Mostrar Cartas

case 'a':

DeletaIntervalo(26, 20, 132, 36);

MostrarBaralho(26, 20, Baralho);

PrintVoltarEDeletaCaracter(comando);

while (\*comando != 'v' && \*comando != 's'){

defaultMenu( comando);

}

break;

//Funcao Embaralhar

case 'b':

DeletaIntervalo(26, 20, 132, 36);

MostrarBaralho(26, 20, Baralho);

sleep(3);

DeletaIntervalo(18, 20, 137, 36);

sleep(2);

embaralha (&Baralho);

MostrarBaralho(26, 20, Baralho);

PrintVoltarEDeletaCaracter(comando);

while (\*comando != 'v' && \*comando != 's'){

defaultMenu( comando);

}

break;

//Funcao Cortar

case 'c':

DeletaIntervalo(26, 20, 132, 36);

MostrarBaralho(26, 20, Baralho);

sleep(3);

DeletaIntervalo(18, 20, 137, 36);

sleep(2);

corta(&Baralho, &trunfo1);

MostrarBaralho(26, 20, Baralho);

PrintVoltarEDeletaCaracter(comando);

while (\*comando != 'v' && \*comando != 's'){

defaultMenu( comando);

}

break;

//Modo de Jogo

case 'd':

DeletaIntervalo(26, 20, 132, 36);

gotoxy(40,23);

printf(" Modo Debug ? (q/n)\n ");

PrintVoltarEDeletaCaracter(&modoDebug);

if(modoDebug == 'v'){

break;

}

DeletaIntervalo(26, 20, 132, 36);

gotoxy(40,23);

printf(" e \u25BA Burrice Artificial\n ");

gotoxy(40,24);

printf(" h \u25BA Inteligencia Artificial\n ");

PrintVoltarEDeletaCaracter(comando);

while (\*comando != 'v' && \*comando != 's'){

switch (\*comando){

//Nivel Facil

case 'e':

menuQuantidadeDeJogadores(comando);

while(\*comando != 's' && \*comando != 'v'){

switch (\*comando){

//Dois Jogadores

case 'o':

LogicaMesaDois('e', comando, &Baralho, modoDebug);

DeletaIntervalo(0, 0, 250, 50);

if(\*comando == 'v'){

break;

}

break;

//Quatro Jogadores

case 'p':

LogicaMesaQuatro('e', comando, &Baralho, modoDebug);

DeletaIntervalo(0, 0, 250, 50);

if(\*comando == 'v'){

break;

}

break;

default:

defaultMenu( comando);

break;

}

}

break;

//Nivel Dificil

case 'h':

menuQuantidadeDeJogadores(comando);

while(\*comando != 's' && \*comando != 'v'){

switch (\*comando){

//Dois Jogadores

case 'o':

LogicaMesaDois('h', comando, &Baralho, modoDebug);

DeletaIntervalo(0, 0, 250, 50);

if(\*comando == 'v'){

break;

}

break;

//Quatro Jogadores

case 'p':

LogicaMesaQuatro('h', comando, &Baralho, modoDebug);

DeletaIntervalo(0, 0, 250, 50);

if(\*comando == 'v'){

break;

}

break;

default:

defaultMenu( comando);

break;

}

}

break;

default:

defaultMenu( comando);

break;

}

}

break;

//Voltar

case 'v':

DeletaLinha(26, 38, 36);

DeletaIntervalo(18, 20, 137, 36);

PrintMenuFora();

PrintMenuConteudoPrincipal();

DeletaCaracter(60, 39);

gotoxy(40, 39);

printf("Escolha sua opcao : ");

scanf(" %c", comando);

break;

//Sair

case 's':

freeBaralho (&Baralho);

break;

default:

defaultMenu( comando);

break;

}

}

freeBaralho (&Baralho);

}

**Estratégias de Implementação**

Devido à necessidade de se ter pelo menos 2 modos diferentes de dificuldade, escolhemos criar o modo fácil, que decide aleatoriamente qual carta jogar, usando a função rand, e um modo difícil, com implementação mais complexa, que opta qual carta jogar com base nas regras do jogo, cartas da mesa e de sua mão, para oferecer um certo desafio ao jogador.

//Modo de jogo fácil, o cpu escolhe as cartas com a função rand

void ModoFacil(int \*mesaCPU, int numeroDeCartasMao)

{

srand(time(NULL));

\*mesaCPU = rand() % numeroDeCartasMao;

}

//Modo de jogo difícil

void ModoDificil(int \*mesaCPU, TipoMao MaoCPU, TipoCarta \*listaMesa, TipoCarta trunfo, int comeca, int Jogando, int numeroDeCartasMao)

{

int i, tamanho, maiorCarta;

\*mesaCPU = 0;

//Se tiver apenas uma carta na mão

if (numeroDeCartasMao == 1)

{

\*mesaCPU = 0;

}

//Caso seja a primeira jogada da rodada, joga sua menor carta

else if (comeca == Jogando)

{

MenorMao(MaoCPU, trunfo, mesaCPU, numeroDeCartasMao);

}

else

{

//Descobre quantas pessoas jogaram

if (comeca < Jogando)

{

tamanho = Jogando - comeca;

}

else if (comeca == (Jogando + 1))

{

tamanho = 3;

}

else if (comeca == (Jogando + 2))

{

tamanho = 2;

}

int Ordem[tamanho], t, j = 0, posiTrunfo[tamanho], posiNaipe[tamanho], MaiorMesa = 0, ganhou;

t = comeca;

//Ordena um vetor na ordem em que as cartas foram jogadas

for (i = 0; i < tamanho; i++)

{

if((t + i) > 3)

{

Ordem[i] = j;

j++;

}

else

{

Ordem[i] = t + i;

}

}

t = 0;

for (i = 0; i < tamanho; i++)

{

//Procura os trunfos na mesa

if(listaMesa[Ordem[i]].naipe == trunfo.naipe)

{

posiTrunfo[t] = Ordem[i];

t++;

}

}

switch(t)

{

//Nenhuma carta com naipe igual ao trunfo

case 0:

ganhou = -1;

break;

//Apenas uma carta com naipe igual ao trunfo

case 1:

ganhou = comeca;

break;

//Mais de uma carta com naipe igual ao trunfo

default:

ganhou = maior(listaMesa, t, posiTrunfo);

break;

}

//Se não há nenhuma carta com naipe igual ao trunfo

if (ganhou < 0)

{

t = 0;

//Procura as cartas da mesa com naipe igual à primeira jogada

for (i = 0; i < tamanho; i++)

{

if (listaMesa[Ordem[i]].naipe == listaMesa[Ordem[0]].naipe)

{

posiNaipe[t] = Ordem[i];

t++;

}

}

switch(t)

{

//Caso haja apenas uma carta na mesa igual ao naipe da primeira jogada

case 1:

ganhou = posiNaipe[0];

break;

//Mais de uma carta na mesa igual ao naipe da primeira jogada

default:

ganhou = maior(listaMesa, t, posiNaipe);

break;

}

MaiorMesa = ganhou;

//Encontra a maior carta da mão

MaiorMao(MaoCPU, trunfo, listaMesa, comeca, &maiorCarta, numeroDeCartasMao);

//Verifica se o naipe é igual ao do trunfo

if (MaoCPU.cartasJogador[maiorCarta].naipe == trunfo.naipe)

{

//Caso seja, joga a carta

\*mesaCPU = maiorCarta;

}

//Verifica se o naipe da maior carta encontrada é igual ao da primeira carta jogada

else if (MaoCPU.cartasJogador[maiorCarta].naipe == listaMesa[MaiorMesa].naipe)

{

//Caso o valor seja maior, joga a carta

if (MaoCPU.cartasJogador[maiorCarta].posicao > listaMesa[MaiorMesa].posicao)

{

\*mesaCPU = maiorCarta;

}

//Do contrário, joga a menor carta da mão

else

{

MenorMao(MaoCPU, trunfo, mesaCPU, numeroDeCartasMao);

}

}

//Procura a primeira cartas com naipe igual à primeira jogada e com valor maior

else

{

for (i = 0; i < numeroDeCartasMao; i++)

{

if (MaoCPU.cartasJogador[i].naipe == listaMesa[MaiorMesa].naipe)

{

if (MaoCPU.cartasJogador[i].posicao > listaMesa[MaiorMesa].posicao)

{

\*mesaCPU = i;

i = numeroDeCartasMao;

}

//Caso não encontre, joga a menor carta

else if (i == numeroDeCartasMao - 1)

{

MenorMao(MaoCPU, trunfo, mesaCPU, numeroDeCartasMao);

}

}

}

}

}

//Caso a maior carta da mesa tenha o naipe igual ao trunfo

else

{

MaiorMesa = ganhou;

//Encontra a maior carta da mão

MaiorMao(MaoCPU, trunfo, listaMesa, comeca, &maiorCarta, numeroDeCartasMao);

//Verifica se a maior carta tem naipe igual ao do trunfo

if (MaoCPU.cartasJogador[maiorCarta].naipe == trunfo.naipe)

{

//Se a carta com naipe igual ao do trunfo tiver valor maior que a maior da mesa

if (MaoCPU.cartasJogador[maiorCarta].posicao > listaMesa[MaiorMesa].posicao)

{

//Joga a carta

\*mesaCPU = maiorCarta;

}

//Do contrário, joga a menor carta da mão

else

{

MenorMao(MaoCPU, trunfo, mesaCPU, numeroDeCartasMao);

}

}

//Do contrário, joga a menor carta da mão

else

{

MenorMao(MaoCPU, trunfo, mesaCPU, numeroDeCartasMao);

}

}

}

}

# Bibliotecas

* **Cartas:** Biblioteca criada com todas as funções relacionadas ao manuseio das cartas, sendo ela subdividida em funções relacionadas à mão, ao baralho, deletar linhas, caracteres e intervalos, funções de comparação e de impreção.
* **Jogo:** Biblioteca criada para conter toda a lógica por trás do jogo de bisca assim conectando as funções criadas na biblioteca cartas com as suas devidas posições no decorrer do programa.

# Custo computacional

O big O é uma notação assintótica para analisar o tempo de execução de uma função ou programa, no qual ele é o limite superior. É calculado somando ou multiplicando as execuções das linhas de código. Por exemplo, numa soma seria O(1), já num “for” ou “while” de soma seria as n execuções multiplicadas pelo O(1), ficando O(n). Logo:

* **Função gotoxy:** O(1), pois apresenta apenas um print, na posição (x,y).
* **Função criaBaralho:** Como nesta função está presente a função FLVazia, que tem custo O(3), e um “for” que executa 4 vezes, e dentro deste existe outro “for”, que executa 10 vezes e realiza 4 atribuições mais a função insereBaralho, que tem O(4). Logo, temos que criaBaralho tem custo total de O(3 + (4\*10\*8)) = O(323), porém, como é constante é igual a O(1).
* **Função retiraBaralho:** Como nesta função temos um “if” com uma comparação e 5 atribuições, assim, O(6), e um “for” que vai até a posição ‘n‘, que no pior caso é 40, seu custo é O(6n). Logo, temos que o custo da função retiraBaralho é O(6+6n), portanto O(n), sendo n a posição onde é retirada a carta.
* **Função embaralha:** Como nesta função temos um srand, com O(1), um “for” que vai até a posição 40 e dentro deste, ocorre uma atribuição com rand, assim, O(2), outra atribuição com retiraBaralho, portanto O(n), e a função insereBaralho, que possui O(4). Logo, temos que o custo da função embaralha é O(1+40\*(2+n+4)), portanto O(n), onde n é a posição onde é retirada a carta.
* **Função NumeroCartasComecoEFimDeJogoDois:** Sabendo que DeletaIntervalo tem custo O((y2+y1)\*(x2+x1+4)+3), e que PrintYOULOSE e PrintYOUWIN são O(1), e considerando os “if’s” e “switch”, a função NumeroCartasComecoEFimDeJogoDois tem O((y2+y1)\*(x2+x1)), onde (x1,y1) e (x2,y2), são coordenadas passadas para a função.
* **Função ProximaRodadaDois:** Levando em consideração as funções acima, e as condições presentes na função, temos que a função ProximaRodadaDois tem custo igual a 3 vezes o custo da função NumeroCartasComecoEFimDeJogoDois, logo, O(3\*(y2+y1)\*(x2+x1)), assim, O((y2+y1)\*(x2+x1)), onde (x1,y1) e (x2,y2), são coordenadas passadas para a função.
* **Função LogicaMesaDois:** Como a função depende de outra que requer um comando específico do usuário para sair do loop, o custo é O(∞), pois há a possibilidade dele não sair do loop.
* **Função funcComparaDois:** Sabendo que as funções maior, achaTrunfosECompara e naipesIguaisAoPrimeiro são O(n), como a função funcComparaDois utiliza-os, seu custo é O(n), sendo n o número de jogadores.
* **Função SeletorMesa:** Como a função depende de outra que requer um comando específico do usuário para sair do loop, o custo é O(∞), pois há a possibilidade dele não sair do loop.
* **Função ModoFacil:** Como a função modoFácil usa a srand e realiza uma atribuição após escolher um valor aleatório com a rand, temos que seu custo é O(3), logo O(1).
* **Função ModoDificil:** Considerando os custos da maiorMao, que é O(n+y), e da menorMao, que é O(2n+m), e dois “for” até a posição t, e um “for” até a posição n, além dos “if’s” e “switchs”, temos que o custo da função ModoDificil é O(n+m+y+t), onde n é o numeroDeCartasMao, t é o tamanho decidido na função, m é o tamanho decidido em menorMao e y é o tamanho decidido em maiorMao.