

Joana Martins e Juliane Ferreira

# **Relatório: Trabalho de Estrutura de Dados I**

Vitória, Brasil

14 de Maio de 2019

Joana Martins e Juliane Ferreira

## **Relatório: Trabalho de Estrutura de Dados I**

Relatório explicativo, na qual relata as atividades desenvolvidas para desenvolvimento de um jogo de busca utilizando a linguagem de programação C

Universidade Federal do Espírito Santo

Estruturas de Dados I

Centro Tecnológico

Vitória, Brasil

14 de Maio de 2019

# Lista de ilustrações

Figura 1 – Menu inicial . . . . .	6
Figura 2 – Menu quantidade de jogadores . . . . .	7
Figura 3 – Mao do jogador . . . . .	7
Figura 4 – Fim de rodada e menu do jogador . . . . .	8
Figura 5 – Final de uma partida . . . . .	8
Figura 6 – Menu de opções . . . . .	9
Figura 7 – Função Seta . . . . .	10
Figura 8 – Exemplo do switch com system("clear") no menu de dificuldades . . . .	10
Figura 9 – Struct utilizado no trabalho . . . . .	11
Figura 10 – Função para criar o baralho . . . . .	12
Figura 11 – Função para embaralhar auxiliar . . . . .	13
Figura 12 – Embaralhar . . . . .	14
Figura 13 – Exibição de cartas e baralho . . . . .	15
Figura 14 – Função que retira uma carta . . . . .	16
Figura 15 – Corta o trunfo . . . . .	19
Figura 16 – Funções que desalocam a memória . . . . .	20
Figura 17 – Memória final . . . . .	20
Figura 18 – Jogo . . . . .	21
Figura 19 – Função Tempo . . . . .	24
Figura 20 – Função Tempo . . . . .	24

# Sumário

1	INTRODUÇÃO . . . . .	4
2	JOGO DE BISCA . . . . .	5
3	COMO JOGAR? COMO O PROGRAMA EXECUTA? . . . . .	6
4	MENU DO JOGO . . . . .	9
5	FUNÇÕES BÁSICAS . . . . .	11
6	FUNÇÕES CRIAÇÃO E MANIPULAÇÃO . . . . .	12
7	FUNÇÕES DE EXIBIÇÃO . . . . .	15
8	FUNÇÕES INSERIR E REMOVER . . . . .	16
9	FUNÇÕES JOGO . . . . .	17
10	FUNÇÕES TRUNFO . . . . .	19
11	FUNÇÕES DESALOCAR . . . . .	20
12	MODOS JOGO . . . . .	21
13	DIFICULDADES . . . . .	22
14	TEMPO . . . . .	24
	Conclusão . . . . .	25

# 1 Introdução

O documento em questão trata primordialmente dos resultados obtidos pelo script de um jogo de busca desenvolvido em C.

Nele, estão as dificuldades e a linha de pensamento para resolvê-las e por fim o que aprendemos e quanto evoluímos durante todo o processo.

## 2 Jogo de Bisca

A Bisca é um jogo de cartas que utiliza-se do baralho com as cartas 2 a 7, dama, valete, rei e Ás, de todos os naipes, cujo principal objetivo é acumular mais pontos que o adversário. O número de participantes pode ser de 2 ou 4 jogadores.

Nesse trabalho estamos considerando as regras estabelecidas no documento do trabalho.

Assim nos baseando nisso para desenvolver toda a lógica por trás do funcionamento do programa.



Assim, isso levará para o menu de jogador, onde o usuário pode escolher contra quantos oponentes que jogar (sendo todos computadores). Usando assim da mesma lógica do menu anterior, apertando '1' para começar a jogar.



Figura 2 – Menu quantidade de jogadores

Ao começar o jogo, será visto sua mão inicial e o trunfo. Logo abaixo um local onde o usuário pode jogar a carta. Para jogar a carta é preciso digitar o valor da carta, ou seja, ás, 2, 7 e etc, junto com o naipe (o que seria C para Copas, E para Espadas, P para paus e O para ouro). Por fim, apertar Enter para a resposta registrar.

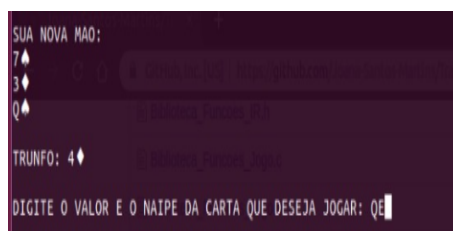
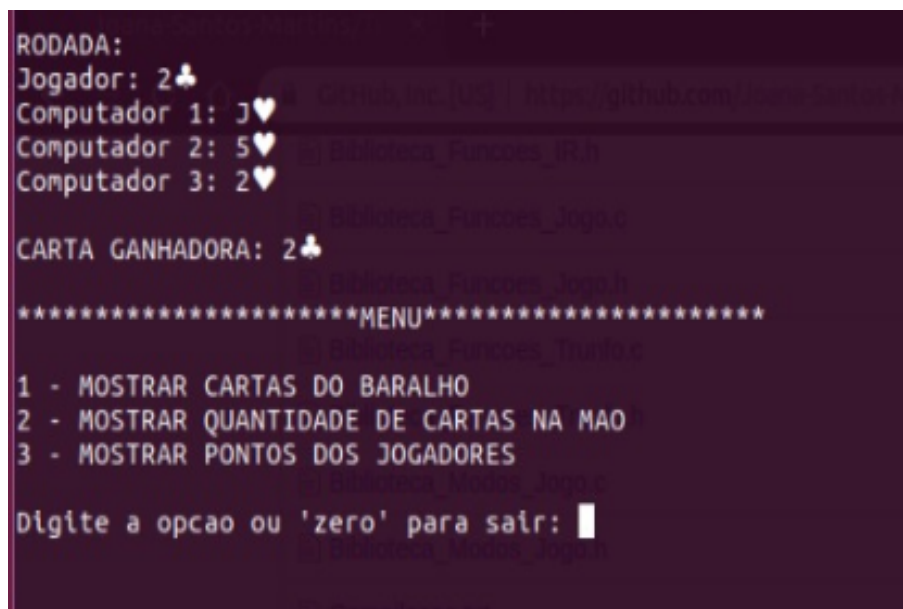


Figura 3 – Mao do jogador



Assim o programa irá calcular quem venceu a partida, o que afetará quem será o primeiro a jogar na próxima rodada. Seguinte a isso, será exibido um menu para o jogador ter opções do que fazer durante a partida, por exemplo, exibir a pontuação dos jogadores. Para sair e dar continuidade ao jogo basta apertar '0'.



```
RODADA:
Jogador: 2 ♣
Computador 1: 3 ♥
Computador 2: 5 ♥
Computador 3: 2 ♥

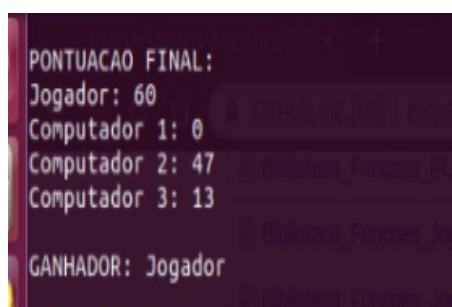
CARTA GANHADORA: 2 ♣

*****MENU*****
1 - MOSTRAR CARTAS DO BARALHO
2 - MOSTRAR QUANTIDADE DE CARTAS NA MAO
3 - MOSTRAR PONTOS DOS JOGADORES

Digite a opcao ou 'zero' para sair: 
```

Figura 4 – Fim de rodada e menu do jogador

Após fazer isso diversas vezes, uma hora o baralho acaba e alguém ganha o jogo. Assim, o programa mostra as pontuações finais e anunciar o ganhador.



```
PONTUACAO FINAL:
Jogador: 60
Computador 1: 0
Computador 2: 47
Computador 3: 13

GANHADOR: Jogador
```

Figura 5 – Final de uma partida

## 4 Menu do jogo

O menu do jogo serve para que o usuário possa escolher a dificuldade e a quantidade de jogadores com quais poderá jogar.



Figura 6 – Menu de opções

Isso foi feito primeiramente implementando uma função chamada `getch()` que consegue ler um caractere digitado pelo usuário sem precisar utilizar-se da tecla Enter. Sendo assim possível ler as setas do teclado para mudar as posições e ler a tecla Enter para selecionar uma opção sem sair do programa.

Assim começamos a implementação por uma função que vê qual a seta foi selecionada. Considerando que uma seta é composta com três caracteres que são `'33' ++ '[' ++ 'A'` (para seta para cima) ou `'B'` (para seta para baixo).

```
int seta(){
    int option;
    getch();
    switch(getch()){
        case 'A':
            option=0;
            break;

        case 'B':
            option=1;
            break;
    }
    return option;
}
```

Figura 7 – Função Seta

A partir disso foram sequências de switch cases para o computador identificar o que deve ser mostrado e no caso de uma seta ser apertada usar o comando `system("clear")` para limpar a tela e simular um menu gráfico. Além de retornar informações necessárias para rodar o programa, como dificuldade e o número de jogadores.

```
void dificuldade(int i){
    switch (i) {
        case 0:
            system("clear");
            titulo();
            printf("\n                SELECIONE A DIFICULDADE:\n\n");
            printf("                FACIL\n");
            printf("                DIFICIL\n");
            printf("\n                1 - PROX\n");
            printf("\n                0 - SAIR\n");
            break;
        case 1:
            system("clear");
            titulo();
            printf("\n                SELECIONE A DIFICULDADE:\n\n");
            printf("                -> FACIL\n");
            printf("                DIFICIL\n");
            printf("\n                1 - PROX\n");
            printf("\n                0 - SAIR\n");
            break;
        case 2:
            system("clear");
            titulo();
            printf("\n                SELECIONE A DIFICULDADE:\n\n");
            printf("                FACIL\n");
            printf("                -> DIFICIL\n");
            printf("\n                1 - PROX\n");
            printf("\n                0 - SAIR\n");
            break;
        default:
            break;
    }
}
```

Figura 8 – Exemplo do switch com `system("clear")` no menu de dificuldades

## 5 Funções Básicas

Biblioteca de funções básicas foi feita para definir a nossa struct, inicializar o baralho, contar a quantidade de itens da lista e comparar duas cartas (para facilitar a comparação no decorrer do processo).

```
/* STRUCTS */
typedef struct{
    char valor; //DETERMINA A PONTUACAO DA CARTA
    char naipe; //DETERMINA O NAIPE DA CARTA
} TipoCarta;

typedef struct TipoCelula *TipoApontador;

typedef struct TipoCelula {
    TipoCarta Item; //CARTA DE UMA CELULA
    TipoApontador Prox; //APONTADOR PARA A PROXIMA CELULA
} TipoCelula;

typedef struct {
    TipoApontador Primeiro, Ultimo;
} TipoLista;
```

Figura 9 – Struct utilizado no trabalho

Decidimos fazer desse jeito, pois tem todas as informações que precisamos para execuções futuras de maneira fácil de manipular. Ademais, nessa biblioteca está presente a alocação do baralho em si, o que permite o programa rodar.

## 6 Funções Criação e Manipulação

Essa biblioteca serve primordialmente para criar o baralho, de forma que atribui valores e o naipe a todas as cartas do baralho, de forma que crie todas as cartas necessárias.

```
void CriaBaralhoInicial(TipoLista* baralho){
    char naipes[4]={'P','O','E','C'};
    char valores[10]={'2','3','4','5','6','7','J','Q','K','A'};

    TipoCarta *carta=(TipoCarta*)malloc(40*sizeof(TipoCarta));
    int i=0;
    for(int v=0; v<10; v++){
        for(int n=0; n<4; n++){
            carta[i].naipe=naipes[n];
            carta[i].valor=valores[v];
            InsereCarta(carta[i],baralho);
            i++;
        }
    }
    free(carta);
}
```

Figura 10 – Função para criar o baralho

Apartir dessa lista será possível jogar o jogo em si.

- Embaralhar Auxiliar:

A função EmbaralhaBaralhoAuxiliar, que tem como entrada um TipoLista\* baralho, consiste em:

- 1- Dividir o baralho original dado como entrada ao meio, em dois baralhos auxiliares: o primeiro conterá suas 20 primeiras cartas, enquanto o segundo, suas 20 últimas;
- 2- Inserir, de forma intercalada, os elementos dos baralhos auxiliares no baralho original. Inicialmente, cria-se e aloca-se dois baralhos, baralhoAux1 e baralhoAux2, inicialmente vazios. Esses serão nossos baralhos auxiliares. Tem-se: a variável tamanhoBaralho, que é igual à quantidade de cartas do baralho original (no caso, ela será igual a 40); e a variável metadeBaralho, que é igual à metade da variável exposta anteriormente (assim, ela será igual a 20).

O primeiro loop ‘for’ é responsável pela inserção das cartas do baralho original no primeiro baralho auxiliar. Note que ele se repetirá de  $i=0$  à  $=20$ , pegando a primeira carta do baralho original, retirando-a e inserindo-a no baralhoAux1. Ao final deste loop, o baralhoAux1 conterá as 20 primeiras cartas do baralho original, enquanto o mesmo terá sido reduzido para suas 20 últimas cartas.

```

/* FUNCAO AUXILIAR DE EMBARALHAR UM BARALHO */
void EmbaralhaBaralhoAuxiliar(TipoLista* baralho){
    TipoLista *baralhoAux1=InicializaBaralho();
    ChecaBaralhoVazio(baralhoAux1);
    TipoLista *baralhoAux2=InicializaBaralho();
    ChecaBaralhoVazio(baralhoAux2);
    TipoCarta carta;
    int tamanhoBaralho=Quantidade(baralho);
    int metadeBaralho=tamanhoBaralho/2;
    for(int i=0;i<metadeBaralho;i++){
        carta=PegaPrimeiraCarta(baralho);
        InsereCarta(carta,baralhoAux1);
        RetiraPrimeiraCarta(baralho);
    }
    for(int i=metadeBaralho;i<tamanhoBaralho;i++){
        carta=PegaPrimeiraCarta(baralho);
        InsereCarta(carta,baralhoAux2);
        RetiraPrimeiraCarta(baralho);
    }
    while(!ChecaBaralhoVazio(baralhoAux1) && !ChecaBaralhoVazio(baralhoAux2)){
        carta=PegaPrimeiraCarta(baralhoAux1);
        InsereCarta(carta,baralho);
        RetiraPrimeiraCarta(baralhoAux1);
        carta=PegaPrimeiraCarta(baralhoAux2);
        InsereCarta(carta,baralho);
        RetiraPrimeiraCarta(baralhoAux2);
    }
    Desalocalista(baralhoAux1);
    Desalocalista(baralhoAux2);
}

```

Figura 11 – Função para embaralhar auxiliar

Já o segundo loop ‘for’ é responsável pela inserção das cartas remanescentes do baralho original no segundo baralho auxiliar. Note que ele se repetirá de  $i=20$  à  $i=40$ , pegando a primeira carta do baralho remanescente, retirando-a e inserindo-a no baralhoAux2. Ao final deste loop, o baralhoAux2 conterá as 20 cartas do baralho remanescente – as 20 últimas cartas do baralho original, enquanto o mesmo estará vazio.

Assim, o baralhoAux1 contém as 20 primeiras cartas do baralho original dado como entrada, enquanto o baralhoAux2, as 20 últimas. Montados os dois baralhos auxiliares, o próximo passo é inserir suas cartas no baralho original (agora vazio) de forma intercalada. Usando o ‘while’, enquanto ambos os baralhos auxiliares não forem vazios, insere-se a primeira carta do baralhoAux1 no baralho original, retirando-a do seu baralho de origem. Analogamente, insere-se a primeira carta do baralhoAux2 no baralho original, retirando-a do mesmo.

Por fim, basta desalocar os baralhos auxiliares alocados dinamicamente no começo da função.

Logo, ao final deste processo, ter-se-á um novo baralho embaralhado.

- Embaralhar

Note que a função anteriormente apresentada – a EmbaralhaBaralhoAuxiliar – como o nome sugere, é uma função auxiliar. Chamando-a por uma única vez, o baralho gerado não estará muito embaralhado. Foi-se observado, com uma bateria de testes, que o mesmo estaria mais bem embaralhado se a função auxiliar fosse chamada 4

vezes, diversificando a mão do jogador e do(s) computador(es). Este é o objetivo da função principal de embaralhar – `EmbaralhaBaralho`.

```
/* EMBARALHA UM BARALHO, CHAMANDO A FUNCAO DE EMBARALHAR AUXILIAR QUATRO VEZES */  
void EmbaralhaBaralho(TipoLista *baralho){  
    EmbaralhaBaralhoAuxiliar(baralho);  
    EmbaralhaBaralhoAuxiliar(baralho);  
    EmbaralhaBaralhoAuxiliar(baralho);  
    EmbaralhaBaralhoAuxiliar(baralho);  
}
```

Figura 12 – Embaralhar

Dado como entrada um baralho, a função chamará a `EmbaralhaBaralhoAuxiliar` 4 vezes, com o objetivo de deixar o baralho original mais embaralhado.

Note que, devido a forma de implementação da função auxiliar `EmbaralhaBaralhoAuxiliar`, o baralho embaralhado gerado sempre será o mesmo; conseqüentemente, as mãos do jogador e do(s) computador(es) sempre serão as mesmas no começo de cada partida. Contudo, tal fato não prejudica a jogabilidade do programa.

Por fim, com um baralho decididamente embaralhado é possível criar as mãos dos jogadores e por fim, dar-se início ao jogo.

## 7 Funções de Exibição

Essa biblioteca é usada em qualquer momento que as cartas tem q ser exibidas para o jogador. Sendo composta de duas funções, uma que exhibe cartas individualmente e uma que exhibe o baralho completo.

```
/* IMPRIME UMA DADA CARTA */
void MostraCarta(TipoCarta* carta){
    switch(carta->naipe){
        case 'P':
            printf("%c%s\n",carta->valor,CLUB);
            break;
        case 'E':
            printf("%c%s\n",carta->valor,SPADE);
            break;
        case 'C':
            printf("%c%s\n",carta->valor,HEART);
            break;
        case 'O':
            printf("%c%s\n",carta->valor,DIAMOND);
            break;
    }
}

/* IMPRIME UM DADO BARALHO */
void MostraCartasBaralho(TipoLista* baralho){
    TipoCelula* aux=baralho->Primeiro;
    while(aux!=NULL){
        MostraCarta(&aux->Item);
        aux=aux->Prox;
    }
    DesalocaCelula(aux);
}
```

Figura 13 – Exibição de cartas e baralho

Utilizamos unicode para printar os naipes das cartas, ficando mais bonito na hora de exibir para o usuário. O programa acima, checka qual é o naipe (Paus,Espadas,Ouro,Copas) e imprime o unicode correto.



## 8 Funções Inserir e Remover

Assim como o nome diz, essa biblioteca serve para inserir e retirar cartas do baralho.

Sendo assim é composta por função que checa se uma certa carta já existe, outra que insere uma carta no baralho e uma que insere uma carta no baralho. Já as de retirar consistem em uma que retira carta do baralho e uma que retira a primeira carta do baralho. Por fim, temos uma função retorna a primeira carta de um certo baralho.

```
void RetiraCarta(TipoCarta carta, Tipolista *baralho){
    TipoCelula* ant=NULL;
    TipoCelula* p=baralho->Primeiro;
    if(ChecaBaralhoVazio(baralho)){
        printf("\nBaralho vazio\n");
        return;
    }
    while(p!=NULL && (p->Item.naipes!=carta.naipes || p->Item.valor!=carta.valor)){
        ant=p;
        p=p->Prox;
    }
    if(p==NULL){
        printf("\nCarta nao existe. Nenhum elemento sera retirado.\n");
        DesalocaCelula(ant);
        DesalocaCelula(p);
        return;
    }
    if(p==baralho->Primeiro){
        baralho->Primeiro=p->Prox;
        DesalocaCelula(ant);
        DesalocaCelula(p);
        return;
    }
    else if(p==baralho->Ultimo){
        baralho->Ultimo=ant;
        ant->Prox=NULL;
        DesalocaCelula(p);
        return;
    }
    else{
        ant->Prox=p->Prox;
        DesalocaCelula(p);
        return;
    }
}
```

Figura 14 – Função que retira uma carta

Tudo isso, é manipulação de lista básica que aprendemos em sala. E através dessas funções é possível manipular os dados do jogo, de forma q seja possível fazer toda a lógica por trás dele.

## 9 Funções Jogo

Biblioteca composta pelas funções relacionadas à jogabilidade do programa.

- **Função PreparaBaralho:** Dado o endereço para um elemento TipoCarta, a função prepara o baralho para o jogo. Primeiro, inicializa-se e checa se o baralho inicializado está de fato vazio. Depois, atribui-se valores às suas cartas, usando a função CriaBaralhoInicial. Após devidamente criado, o mesmo é embaralhado, cortado e rearranjado, reposicionando o trunfo para seu final e salvando a carta no endereço fornecido como entrada.
- **PreparaMaos:** Informando o número de jogadores desejado, uma lista de listas é alocada dinamicamente, representando as mãos do jogador e do(s) computador(es). Em seguida, tais mãos são formadas, através da função FormarMao, que retira as cartas do baralho e as insere nas devidas mãos.
- **PreparaMontePontos:** Informando o número de jogadores desejado, uma lista de listas é alocada dinamicamente, representando o monte de pontos do jogador e do(s) computador(es).
- **ChecaExisteCarta:** A função permanece em um loop até que o valor e naipe da carta em que o jogador pretende jogar sejam válidos. Quando digitada uma carta válida e presente na mão do jogador, o valor será guardado no vetor carta (que possui as cartas jogadas pelo jogador e pelo(s) computador(es) em uma certa rodada) na posição JOGADOR, correspondendo, assim, à carta jogada pelo jogador.
- **MelhordaMao:** Retorna a carta de uma dada mão com maior número de pontos.
- **PiordaMao:** Retorna a carta de uma dada mão com menor número de pontos.
- **JogaComputador:** Escolhido o modo de jogo desejado (fácil ou difícil), a função realiza a jogada de um computador. Se o modo escolhido for ‘fácil’, o computador sempre jogará a primeira carta de sua mão. Caso o modo seja o ‘difícil’, ocorre: se a carta do jogador for do mesmo naipe do trunfo, o computador jogará sua pior carta (usando a função PiordaMao); caso contrário, ele jogará a sua melhor carta (usando a função MelhordaMao).
- **PontosCarta:** Dada uma carta como entrada, a função retorna a quantidade de pontos que ela oferece.
- **MelhorCarta:** Compara duas cartas dadas como entrada, tomando como base sempre a primeira. Se ambas forem do mesmo, a função retorna a de maior valor (usando

a função `PontosCarta`). Se forem de naipes diferentes, a primeira carta é retornada (vale regra do jogo).

- `CartaGanhadoraAux`: Compara duas cartas, considerando o trunfo da partida. Retorna a carta que tiver o mesmo naipe que o trunfo. Caso não ocorra, retorna a melhor carta (usando a função `MelhorCarta`).
- `RefazOrdemJogadores`: Com base no ganhador da rodada, refaz a circulação dos jogadores na mesa, alterando a ordem dos mesmos com base nas regras do jogo.
- `CartaGanhadora`: Com base na nova posição dos jogadores na mesa, retorna a carta ganhadora entre as cartas jogadas pelo jogador e pelo(s) computador(es), tomando como base o ganhador da rodada anterior.

; Em uma rodada, imprime as cartas que foram jogadas pelos participantes, imprimindo-a na ordem em que foram jogadas.

- `InserirMontePontos`: Sabendo o ganhador da rodada, a função insere as cartas dos demais participantes da mesa no monte de pontos do ganhador.
- `RefazMaoBaralho`: Após uma rodada, refaz as mãos dos jogadores, retirando ordenadamente uma carta do baralho e inserindo-na nas devidas mãos.
- `TotalPontos`: Dado um monte de pontos, retorna o total de pontos que ela possui.
- `ImprimePontos`: Em uma rodada, imprime os pontos até então acumulados por cada participante da mesa.
- `MaiorPonto`: Função que retorna o monte de pontos com maior número de pontos (usando a função `TotalPontos`).
- `Ganhador`: Imprime o ganhador do jogo com base no monte de pontos com maior pontos (usando a função `MaiorPonto`).
- `FazOrdemJogadores`: Ordena as jogadas dos participantes da mesa, começando com o ganhador da rodada anterior e seguindo de forma circular (vide regras do jogo).
- `GanhadorRodada`: Retorna o índice do participante que ganhou uma certa rodada.

## 10 Funções Trunfo

Nessa biblioteca estão as funções para cortar o baralho, ou seja, definir qual é o trunfo da partida.

```
TipoCarta CortaTrunfo(TipoLista* baralho){
    TipoCarta trunfo;
    if(ChecaBaralhoVazio(baralho)) printf("\nBaralho vazio!\n");
    else{
        srand(time(NULL));
        trunfo=baralho->Primeiro->Item;
        TipoCelula* aux=baralho->Primeiro;
        for(int i=2;aux!=NULL;i++){
            if(rand()%i==0)
                trunfo=aux->Item;
            aux=aux->Prox;
        }
        DesalocaCelula(aux);
    }
    return trunfo;
}
```

Figura 15 – Corta o trunfo

A lógica dessa função usa o `srand()` que gera números aleatórios com base no relógio. Assim igualamos a variável `trunfo` com o primeiro item do baralho, para depois fazer um `for` que gera um número aleatório para assim colocar dentro da carta `trunfo`, fazendo um `trunfo` diferente em cada partida e mudando toda a lógica do jogo.

# 11 Funções Desalocar

Nessa biblioteca tem as funções que tem apenas a utilidade de desalocar a memória que foi alocada durante a execução do programa. Já que a memória do computador utilizada tem que ser liberada, senão apenas gasta a memória do computador e fica informação lá que não é necessária depois que o programa acaba. Então para uma efetiva administração da memória, é necessário liberá-la.

Como existem muitas estruturas complexas e listas de listas no código do programa, foi necessário a implementação de funções que percorrem todas essas informações liberando, assim, a memória.

```
/* DESALOCA UMA DADA CELULA */
void DesalocaCelula(TipoCelula *celula){
    free(celula);
}

/* DESALOCA UMA DADA LISTA */
void Desalocalista(TipoLista *Lista){
    TipoCelula *p=Lista->Primeiro;
    TipoCelula *aux=p;
    while(p!=NULL){
        aux=p;
        p=p->Prox;
        free(aux);
    }
    free(Lista);
}

/* DESALOCA UMA DADA LISTA DE LISTAS */
void DesalocalistaDeLista(TipoLista **Lista, int numero_jogadores){
    for(int i=0;i<numero_jogadores;i++){
        Desalocalista(Lista[i]);
    }
    free(Lista);
}
```

Figura 16 – Funções que desalocam a memória

```
==6890==
==6890== HEAP SUMMARY:
==6890==   in use at exit: 0 bytes in 0 blocks
==6890== total heap usage: 533 allocs, 533 frees, 10,640 bytes allocated
==6890==
==6890== All heap blocks were freed -- no leaks are possible
==6890==
==6890== For lists of detected and suppressed errors, rerun with: -s
==6890== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 17 – Memória final

Podemos ver, então que o programa termina desalocando toda a memória anteriormente alocada e o debug(valgrind) não acusa nenhum erro de compilação nem de execução.

## 12 Modos Jogo

Nessa biblioteca é onde juntam todas as funções necessárias para o jogo, criando assim um loop quem recebe as entradas dada pelo usuário e as coloca de entrada nas funções de outras bibliotecas enquanto o baralho não tiver vazio. Assim que acabar as cartas do baralho essa função encerra o jogo para mostrar a pontuação final e o ganhador. Ademais, também há, dentro de outro loop, o menu do jogador que apresenta opções que possam ser úteis ao jogador durante o jogo e para ser moldado a sua estratégia.

```

* FUNCAO DE JOGABILIDADE DO BISCA */
void JogoBisca(int d){
    clock_t t0, tf;
    double tempo_gasto;
    t0 = clock();
    int numero_jogadores = modo_j(d);
    char modo_jogo = modo_d(d);
    int ganhou = 0;
    char opcao;
    TipoCarta trunfo;
    TipoCarta *cartaganhadora;
    TipoLista* baralho=PreparaBaralho(&trunfo);
    TipoLista** mao=PreparaMaos(baralho,numero_jogadores);
    TipoLista** pontos=PreparaMontePontos(baralho,numero_jogadores);
    TipoCarta carta[numero_jogadores];
    printf("SUA MAO INICIAL:\n");
    MostraCartasBaralho(mao[JOGADOR]);

    while(!ChecaBaralhoVazio(mao[JOGADOR])){
        printf("\nTRUNFO: ");
        MostraCarta(&trunfo);
        carta[JOGADOR]=ChecaExisteCarta(carta,mao,numero_jogadores);
        FazOrdemJogadores(ganhou,numero_jogadores,&trunfo,carta,mao,modo_jogo);
        cartaganhadora=CartaGanhadora(carta,&trunfo,numero_jogadores, ganhou);
        ImprimeCartasJogadas(carta,numero_jogadores,ganhou);
        ganhou=GanhadorRodada(numero_jogadores, carta, cartaganhadora);
        printf("\nCARTA GANHADORA: ");
        MostraCarta(CartaGanhadora(carta,&trunfo,numero_jogadores, ganhou));
        InsereMontePontos(carta,pontos,&trunfo,numero_jogadores, ganhou);
        RefazMaoBaralho(mao,baralho,numero_jogadores);
        do{
            printf("\n*****MENU*****\n");
            printf("\n1 - MOSTRAR CARTAS DO BARALHO");
            printf("\n2 - MOSTRAR QUANTIDADE DE CARTAS NA MAO");
            printf("\n3 - MOSTRAR PONTOS DOS JOGADORES");
            printf("\n\nDigite a opcao ou 'zero' para sair: ");
            scanf(" %c",&opcao);
            system("clear");

            switch(opcao){
                case '0':
                    break;
                case '1':
                    printf("\nIMPRIMINDO BARALHO\n");
                    MostraCartasBaralho(baralho);
                    break;
                case '2': printf("\nTOTAL DE CARTAS NA MAO: %d\n",Quantidade(mao[JOGADOR]));
                    break;
                case '3':
                    printf("\nPONTUACAO:");
                    ImprimePontos(pontos,numero_jogadores);
                    printf("\n");
                    break;
                default: printf("OPCAO INVALIDA. TENTE NOVAMENTE.\n");
                    break;
            }
        }while(opcao!="0");

        if(!ChecaBaralhoVazio(mao[JOGADOR])){
            printf("\nSUA NOVA MAO:\n");
            MostraCartasBaralho(mao[JOGADOR]);
        }
    }
    printf("\nPONTUACAO FINAL:");
    ImprimePontos(pontos,numero_jogadores);
    Ganhador(pontos,numero_jogadores);
}

```

Figura 18 – Jogo

## 13 Dificuldades

Como qualquer trabalho, sempre há erros quando se tenta programar algo um pouco mais complexo: falha de segmentação, ‘warnings’, ‘leak’ de memória, dentre outros. Neste, não foi diferente. Algumas das funções com as quais tivemos um certo nível de dificuldade foram:

- Desalocamento de memória

A Biliboteca Funcoes Desalocar é a responsável por desalocar os elementos que foram alocados dinamicamente na memória durante a execução do programa. Mas criá-la não foi de todo fácil.

Tratando-se de uma variável do tipo Struct TipoCelula (uma célula da lista encadeada), sua função de desalocamento de memória – a DesalocaCelula – é simples de se implementar, já que a célula é o elemento básico da lista. Por ser um ponteiro único, basta dar ‘free’ para liberá-la da memória. Para tal função, não houve dificuldades na implementação.

Para uma variável do tipo Struct TipoLista (o baralho, por exemplo), sua função de desalocamento – a DesalocaLista – ganha um pouco de complexidade. Ao final da execução do programa, sua falta era perceptível com o ‘leak’ que ela gerava. Mas, como visto e praticado em sala de aula, basta desalocar cada célula da lista separadamente. Para tal função, requeriu-se um pouco mais de planejamento para sua implementação.

A maior dificuldade ocorreu quando tratava-se de uma variável “Lista de Lista” (a TipoLista\*\* pontos, por exemplo). Tínhamos, assim, uma lista encadeada de listas encadeadas. O planejamento para sua função de desalocamento – a DesalocaListaDeLista – foi um tanto conturbada e demorada. Foi difícil compreender como acessar cada lista individualmente, bem como cada célula dessas listas. Ao final da execução do programa, sua falta gerava um ‘leak’ absurdo de memória, e consertá-lo não foi de tão imediato. Depois de bem estudar e analisar a estrutura em questão, conseguiu-se corrigir os erros apresentados.

- Embaralhar

Como apresentado anteriormente, a função de embaralhar é separada em duas partes: uma auxiliar que embaralha em si (EmbaralhaBaralhoAuxiliar) e outra que chama a auxiliar 4 vezes para deixar o baralho mais embaralhado (EmbaralhaBaralho).

Para essa função, uma das dificuldades foi relacionada ao planejamento da mesma. Existem outros modos de criar uma função que realiza o pedido, mas faltava talvez um pouco de conhecimento teórico para a sua implementação. Outra forma de embaralhamento foi pensada (usando, como apoio, as funções da Biblioteca Funcoes Trunfo), mas, ao analisá-la junto à função por nós apresentada, preferiu-se deixá-la de lado. Mas isso não significa que ela é melhor opção. A função faz o necessário, porém, com zero de variabilidade, entregando sempre o mesmo baralho embaralhado. Outro problema enfrentado foi o grande ‘leak’ que memória que essa função gerava. Como ela aloca dinamicamente outros dois baralhos auxiliares, o uso de memória aumentava consideravelmente. Esse problema, porém, foi contornado com a inclusão da Biblioteca Funcoes Desalocar, que desalocou tais baralhos corretamente.

- Ordenar as jogadas

No jogo bisca, como apresentado na descrição deste trabalho, existe uma ordenação nas jogadas: o ganhador de uma rodada começa a próxima. Contudo, no início da implementação do mesmo, tal regra não foi considerada, pois, no modo em que o jogo foi pensado, seu planejamento não era fácil. Os jogadores foram criados com valores pré-definidos, e não como uma struct particular ou como uma lista circular. Com isso, tinha-se um jogo em que o jogador era o que sempre começava as rodadas, seguido pelo(s) computador(es).

Para consertar tal erro, foram implementadas 2 funções, presentes na biblioteca Biblioteca Funcoes Jogo.h e anteriormente apresentadas: a RefazOrdemJogadores e a FazOrdemJogadores. Seus planejamentos foram um tanto extensos, demandando um pouco de tempo para consertar o erro. Foram criadas duas funções presentes na Biblioteca Funcoes Jogo.h, a FazOrdemJogadores e a RefazOrdemJogadores. Com elas o problema foi contornado, ajustando a jogabilidade corretamente.



## 14 Tempo

Nosso programa não gasta muito tempo para ser compilado, e nem para executar os comando enquanto ainda em execução. Por isso o consideramos bem efetivos nessa questão. Nós tentamos implementar as funções durante o código da maneira mais eficiente possível, ou seja, que faça o que queremos o mais rápido possível.

Assim, rodamos um script dentro do código que calcula o tempo de execução.

```
clock_t t0, tf;  
double tempo_gasto;  
t0 = clock();  
tf = clock();  
tempo_gasto = ( (double) (tf - t0) ) / CLOCKS_PER_SEC;
```

Figura 19 – Função Tempo

Assim, foi possível calcular o tempo da função principal. Que fica na Biblioteca Modos Jogo.

Como resultado:

A small screenshot of a terminal window with a dark background. It displays the text 'Tempo gasto: 0.284956 s' in a light-colored font. Below this line, there is a faint, partially visible line of text that appears to be '6000'.

Figura 20 – Função Tempo

# CONCLUSÃO

É imprescindível, para um aluno de computação, o entendimento de como se desenvolve jogos, fazer estruturas de dados mais complexas e resolver problemas diferentes. Além disso, com a experiência adquirida no decorrer do estudo, problemas posteriores encontrados durante a vida acadêmica serão mais facilmente resolvidos.

A partir do estudo, uma certeza maior ressoa: a de que o conhecimento gera sede por conhecimento, e que esse estudo é apenas o início de uma longa jornada no ramo computacional.