Universidade Federal do Espírito Santo

Ezequiel Reinholtz Schneider
Henrique Coutinho Layber

Implementação de bisca em C
Usando estruturas encadeadas de dados

Professor Dr. Vinícius Fernandes Soares Mota
Universidade Federal do Espírito Santo
Ciência da Computação
Vitória, Espírito Santo, 2019

Sumário

Visão geral

O objetivo do trabalho é fazer uso das listas encadeadas, evitando o uso extra de memória e estruturas estáticas para armazenar dados.

Um jogo de bisca convencional, embora simples e intuitivo para humanos, tem operações razoavelmente complexas a nível de programação, como por exemplo o ato de comprar do topo do baralho. Embora estejamos acostumados naturalmente, é uma operação que transfere uma célula do baralho para a mão, carregando informações como valor, naipe e também expondo a próxima carta para ser comprada. Este é só um exemplo mas tem operações mais complicadas como saber quem ganhou (encartou) a o turno jogado.

Tipos Abstratos de Dados (TADs)

TADs são tipos de dados (structs) que definimos o comportamento. São maneiras diferentes de se apresentar e organizar os dados. Grande parte do desafio pode ser resolvido logo de cara ao formar TADs bem definidos e do tipo correto. Foram criados 3 TADs para a implementação da bisca. São eles:

Struct tCarta

A definição do struct:

```
/*define o numero da carta e o naipe;
char naipe, valor;*/
typedef struct structCarta{
    char naipe, valor;
}tCarta;
```

Note que são declarados os campos como `char`, ao invés do habitual `int`, pelo simples motivo de que o `int`, na maior parte das máquinas, ocupa 8 bytes enquanto o char ocupa 1 byte. Não será armazenada informações maiores que 255 $(11111111)_2$.

Struct tDeck

A definição dos structs:

```c
/*Define os items que serão contidos no deck.
(lista encadeada de cartas com sentinela)*/
typedef struct tCelula tCelula;
typedef struct tCelula{
    tCarta carta;
    tCelula *proximo;
} tCelula;

/*Define as sentinelas e auxiliador do deck.
(lista encadeada de cartas com sentinela)*/
typedef struct tDeck{
    tCelula *primeiro, *ultimo;
    int quantidade;
} tDeck;
```

É, como diz na definição, uma lista encadeada de cartas com sentinela. Ou seja, cada célula contém uma carta e aponta para a próxima célula. Ser uma lista encadeada ajuda no sentido de transferência de cartas. Só é necessário alocar o espaço das cartas uma única vez ao iniciar o programa. A opção por armazenar a também a quantidade de cartas no struct vai facilitar o acesso (quando precisar dessa informação) e a checagem para saber se o deck está vazio (basta ver de quantidade é `0`).

Struct tPlayer

A definição do struct:

```c
//  Define os jogadores em lista circular
typedef struct Player tPlayer;

typedef struct Player{
    char points, Id;
    tDeck *mao;
    tPlayer *proximo;
}tPlayer;
```

Com o uso de listas circulares, resolvemos intrinsecamente a questão de ordem nos turnos (quem começa, quem compra, encartes, etc.). É definido também uma importante variável `Id`, e `points`.

O `Id` é definido da seguinte maneira: `Id 0` é o usuário, e os outros Ids são bots. O número do `Id` define o nome do bot.

`points` é atualizada ao fim de cada rodada, ao saber quem ganhou o monte. Novamente, não precisaremos de valores acima de 255.

Estratégias utilizadas

Representação das informações

Além de usar TADs estratégicamente definidos, usar representações de informações bem definidas também resolve uma série de problemas. Por exemplo definimos a ordem de valores das cartas como a ordem de encarte, então, a função de printar teria que adaptar para mostrar a carta que é, e não a “importância de encarte”. Com isso poupamos de tratar de casos e comparações, é só tratar a saída para printar para o usuário, o que teria de ser feito de qualquer forma. Por termos feito a lista do baralho de forma separada, podemos reutilizá-la sem problemas para montar a mãos dos jogadores e monte usando a mesma definição.

Embaralhar

Para a função de embaralhar, não houve preocupação com probabilidades e estatísticas, mas em fazer uma função simples e eficiente, sem alocar mais memória. A nossa estratégia foi trocar o conteúdo de duas cartas (assim evitando problemas com continuidade da lista encadeada) n vezes.

Turnos e jogadas

Quando o jogo é iniciado e a quantidade de jogadores é definida, já começamos comprando 3 cartas para cada um, uma carta por vez, em ordem. O jogo acontece até que a mão do último jogador esteja vazia. Em um turno, o jogador joga uma carta da mão (ou o usuário pode ativar ações especiais como mostrar o baralho, mas ainda deve jogar uma carta) no monte. Os pontos do monte são contados e atribuídos para o jogador que ganhou a rodada, o monte é esvaziado e a memória ocupada pelas cartas nele é liberada. A lista circular de jogadores então é “rotacionada” para apontar para o ganhador do

turno. O próximo turno então começa no ganhador no turno anterior. O usuário sempre começa a partida.

IA

O jogo conta com umas regras para jogar as cartas de maneira que façam um pouco mais de sentido. É um exagero chamar de inteligência artificial, mas é feito para manter simplicidade e “umas regras para jogar as cartas de maneira que façam um pouco mais de sentido”, querendo ou não, no resultado (saída) se assemelha uma IA.

Para a IA de nível fácil, ela é essencialmente desligada, então ela joga sempre a primeira carta da mão.

Para a IA de nível difícil, ela joga a menor carta da mão se há trunfo do monte, caso não tenha, ela joga a maior carta se tiver pontos no monte e caso não haja nem trunfo ou pontos, ela joga a menor carta da mão.

Custos computacionais e notação Big O

Ao analisar, o baralho tem 40 cartas, pode não ser muito mas se for feito funções descuidadas, podemos acabar iterando pelo baralho centenas de vezes. Então, é sempre melhor criar funções que dependam o mínimo possível do tamanho do alvo, ainda mantendo elas genéricas e reutilizáveis. São as funções mais custosas do trabalho:

Embaralha

A estratégia para embaralhar é trocar o conteúdo da primeira carta com o conteúdo de uma carta aleatória, n vezes. Isso implica então que no pior caso, o número gerado será sempre o maior possível, e a entrada é n, o custo computacional dele será

`(n*quantidade de cartas)`, normalmente 40, então `embaralha` é uma função $O(n^2)$.

Corta pq o prof pediu no PDF

A função corta é muito simples. Ela pega uma carta aleatória no baralho e faz ela se tornar a última. No pior dos casos, a carta cortada é última e então a função é $O(n)$ onde n é o número de cartas.

Jogada

A função jogada depende apenas da dificuldade e se o jogador a jogar é um bot. Para o usuário, a função é $O(1)$.

Quando o bot é fácil, a função também é $O(1)$ mas quando não é, como o bot verifica várias coisas no monte todo, o custo computacional acaba resultando em $3n$, sendo n o número de cartas no monte. Logo, `jogada` é $O(n)$ também.

GanhadorMonte

É uma função que retorna qual índice na lista encadeada do monte o ganhador está. É extremamente importante para o ponteiro de jogadores caminhar e iniciar o próximo turno no jogador que ganhou. Ele verifica todo o monte, então o custo computacional é fixo em n, sendo n o número de cartas no monte, logo esta função também é $O(n)$.

Turno

A função `turno` é mais complicada de se analisar o custo computacional. Tem muitas variáveis como tamanho da mão dos jogadores, número de jogadores e qual a dificuldade selecionada. É necessário analisar o custo computacional de cada função chamada por ela antes de ter a resposta:

`todosCompram`: Para todos comprarem juntos e em ordem. Mantém a mãos dos jogadores com 3 cartas se for possível comprar. `O(n)`, `n` = número de jogadores.

`jogada`: `O(n)`, `n` = número de cartas no monte.

`imprimeDeck`: `O(n)`, `n` = número de cartas no monte.

`ganhadorMonte`: `O(n)`, `n` = número de cartas no monte.

`resgataTrunfo`: `O(1)`.

`esvaziaDeck`: `O(n)`, `n` = número de cartas no monte.

Então, o custo computacional de `turno` é a soma do custo computacional das funções que ele chama, ou seja: `4n`. `turno` é uma função `O(n)`.

Jogo

É a função-mestra. Faz as outras chamadas de funções. Assim como `turno`, chama algumas funções e é mais fácil analisar elas antes de analisar a função `jogo`.

`preenche`: `O(n)`, `n` = número de cartas.

`embaralha`: `O(n2)`, `n` = número de cartas

`corta`: `O(1)`.

`quantosJogadores`: `O(1)`.

`dificuldadeMenu`: `O(1)`.

`iniciaNPlayers`: `O(n)`, `n` = número de jogadores.

`todosCompram`: `O(n)`, `n` = número de jogadores.

`turno`, executado número de cartas vezes: `O(n)`, `n` = número de jogadores.

Portanto, o custo computacional de `jogo`, tal como `turno`, é soma das funções que ele chama,

Conclusão

Ressaltando o objetivo de simplicidade e economia de memória a bisca foi desenvolvida com estruturas modularizadas e totalmente focadas em suprir as exigências de utilizar somente listas encadeadas, como também ressaltar um modo intuitivo e fácil de jogar bisca. Implementando também um tratamento menos “inocente” de jogo executado pelo adversário (computador) que também não fosse impossível de se vencer. Ademais foi realizado cálculos de complexidade em funções repetitivas para refinamento do código e rapidez no processo.

Anexo 1

```
/*
    OBJETIVO: Traduzir a saída de números para cartas.
    ENTRADAS: A carta.
    SAIDA: -
    PRE-CONDICAO: Ser um naipe e um valor valido dentro do jogo de bisca.
    POS-CONDICAO: O valor "traduzido" foi impresso na tela e nada foi alterado.
*/
void filtrAEPrinta(tCarta *carta){
    if(carta == NULL)
        return;

    switch(carta->valor){
        case 0: printf("2 "); break;
        case 1: printf("3 "); break;
        case 2: printf("4 "); break;
        case 3: printf("5 "); break;
        case 4: printf("6 "); break;
        case 5: printf("Q "); break;
        case 6: printf("J "); break;
        case 7: printf("K "); break;
        case 8: printf("7 "); break;
        case 9: printf("A "); break;
        default: puts("###### DEU ALGO ERRADO COM O VALOR MEU CONSAGRATED ######");
    }

    printf("de ");

    switch(carta->naipe){
        case 0: printf("Ouros "); break;
        case 1: printf("Paus "); break;
        case 2: printf("Copas "); break;
        case 3: printf("Espadas "); break;
        default: puts("###### DEU ALGO ERRADO COM O NAIPE MEU PROGRAMADO ######");
    }
    puts("");
}
```

Função para printar carta

Anexo 2

```
/*
    OBJETIVO: Funcao que joga a carta no modo facil;
    ENTRADAS: Ponteiro tPlayer* para 'player',
        ponteiro para o monte e ponteiro para o trunfo.
    SAIDA: Ponteiro tCarta* para 'carta'.
    PRE-CONDICAO: -
    POS-CONDICAO: A carta escolhida foi jogada em 'monte'.
*/
void jogadaHard(tPlayer *player, tDeck *monte, tCarta *trunfo){
    //printf("Jogada feita pelo bot %d = ", pGetId(player));

    // compara se a maior carta do monte eh trunfo
    if(getNaipe(maiorCarta(monte, trunfo)) == getNaipe(trunfo)){
        tCarta* menor = menorCarta(pGetMao(player), trunfo);
        jogaCarta(player, monte, buscaCarta(menor, pGetMao(player)));
    }
    else{
        if(contaPontos(monte) > 0){
            tCarta* maior = maiorCarta(pGetMao(player), trunfo);
            jogaCarta(player, monte, buscaCarta(maior, pGetMao(player)));
        }
        else{
            tCarta* menor = menorCarta(pGetMao(player), trunfo);
            jogaCarta(player, monte, buscaCarta(menor, pGetMao(player)));
        }
    }
}
```

Função para jogar em modo difícil

Anexo 3

```
/*
    OBJETIVO: Embaralhar 'deck'.
    ENTRADAS: Ponteiro para 'deck'.
    SAIDA: -
    PRE-CONDICAO: 'deck' existe e está alocado corretamente.
    POS-CONDICAO: 'deck' está embaralhado.
*/
void embaralha(tDeck *deck, int passes){
    if(vazio(deck))
        return;

    tCelula *aux = primeiro(deck);
    srand(time(NULL));

    int iteracao = rand() % getQuantidade(deck);
    for(int i = 0; i < passes; i++){
        for(int n = 0; n < iteracao; n++){
            if(proximo(aux) == NULL)
                break;
            aux = proximo(aux);
        }

        swap2Celulas(primeiro(deck), aux);

        iteracao = rand() % getQuantidade(deck);
        aux = primeiro(deck);
    }
}
```

Função embaralha

Anexo 4

```
/*
    OBJETIVO: Funcao que realiza a jogada, distribui os pontos ao ganhador da rodada e retorna o jogador vencedor
    do turno.
    ENTRADAS: Ponteiro tPlayer* para 'Lista de jogadores', Ponteiro tDeck* para 'baralho', Ponteiro tDeck* para
    'monte', Ponteiro tCarta para 'trunfo', Variavel char para 'Dificuldade do jogo', Variavel int para 'numero
    de jogadores'.
    SAIDA: Ponteiro tPlayer* para 'Jogador vencedor da rodada'.
    PRE-CONDICAO: Haver cartas para os jogadores comprar.
    POS-CONDICAO: Todos os jogadores terem feitos uma jogada, proximo a jogar definido.
*/
tPlayer* turno(tPlayer *player, tDeck *baralho, tDeck *monte, tCelula *trunfo, char dificuldade, int nJogadores){

    // if(player != NULL)
    //     imprimeDeck(pGetMao(player));
    if(getQuantidade(pGetMao(player)) < 3)
        todosCompram(player, baralho, nJogadores);

    jogada(player, monte, getCarta(trunfo), dificuldade, nJogadores, baralho);

    system("clear");
    puts("#MONTE NO FIM DO TURNO#");
    imprimeDeck(monte);
    //indicar quem ganhou
    tPlayer *qmGanhou = ganhadorMonte(player, monte, getCarta(trunfo));

    if(possui(monte, getCarta(trunfo)))
        resgataTrunfo(monte, trunfo);
    esvaziaDeck(monte);
    puts("-------------------------");
    return qmGanhou;
}
```

Função turno

Anexo 5

```
/*
    OBJETIVO: Iniciar 'n' quantidades de tPlayers vazios corretamente.
    ENTRADAS: A quantidade de players a ser inicializada.
    SAIDA: Ponteiro para tPlayer, apontando para o jogador criado.
    PRE-CONDICAO: n > 0.
    POS-CONDICAO: Inicializados, porém sem elementos, ou zerados.
*/
void jogo(tDeck *baralho){
    preenche(baralho);
    embaralha(baralho, 150000);
    corta(baralho);

    system("clear");
    int nJogadores = quantosJogadores();
    int dificuldade = dificuldadeMenu();
    if(nJogadores > getQuantidade(baralho)) return;

    tPlayer *players = iniciaNPlayers(nJogadores);

    for(int i = 0; i < 3; i++)   //jogares compram 3 cartas
        todosCompram(players, baralho, nJogadores);

    tCelula *trunfo = ultimo(baralho);
    tDeck *monte = iniciaVazio();
    system("clear");
    while(!vazio(pGetMao(players)))
        players = turno(players, baralho, monte, trunfo, dificuldade, nJogadores);
    free(trunfo);
    destroiDeck(monte);
    mostraPontuacaoEQuemGanhou(players);

    destroiPlayers(players);
}
```

Função jogo