```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <stdbool.h>
#include "logic.h"
#include "list.h"
#include "file.h"
#include "sdl.h"
#include "ea.h"
#include "error.h"

// Definição das operações válidas nas pilhas de cartas
void stack_push(Stack **sp, Card *card)
{
    Stack *old_sp = *sp;
    Stack *new = (Stack *) ecalloc((size_t) 1, sizeof(Stack));

    new->card = card;

    new->next = old_sp;
    new->prev = NULL;
    if (old_sp != NULL)
        old_sp->prev = new;
    else {
        // estamos a puxar o primeiro elemento, sp tava a NULL
    }

    *sp = new;
}

Card *stack_pop(Stack **sp)
{
    if (*sp == NULL) {
        fprintf(stderr, "Erro: tentou-se fazer pop numa stack vazia.\n");
        exit(EXIT_FAILURE);
    }

    Stack *pop = *sp;
    Card *card = pop->card;
    *sp = pop->next;
    if (pop->next != NULL)
        pop->next->prev = NULL;

    free(pop);

    return card;
}

/*
 * Ler a estrutura de configuração vinda de file.c:read_config()
 * e inicializar os jogadores de acordo.
 */
int init_game(Config *config, List *players)
{
    Player *new_player = NULL;
    const int num_decks = config->num_decks;

    for (int i = 0; i < MAX_PLAYERS; i++) {
        new_player = (Player *) ecalloc((size_t) 1, sizeof(Player));
        if (i + 1 <= config->num_players) {
            // Jogadores efetivos
            new_player->type = config->player_type[i];
            new_player->ingame = true;
            strcpy(new_player->name, config->player_names[i]);
            new_player->money = config->money[i];
            new_player->orig_bet = config->bets[i];
            new_player->bet = new_player->orig_bet;
        }
        else {
```

```
71                  // Lugar não especificado na configuração.
72                  strcpy(new_player->name, "Empty");
73                  new_player->type = VA;
74                  new_player->ingame = false;
75              }
76              new_player->playing = false;
77              list_append(players, new_player);
78          }
79
80          free(config);
81
82          return num_decks;
83      }
84
85      /* Wrapper para dar uma carta a um jogador.
86       * Atualiza cards_left no Megadeck e insere novos baralhos quando
87       * este acaba.
88       */
89      void give_card(Player *player, Megadeck *megadeck)
90      {
91          int random = 0;
92
93          if (megadeck->cards_left == 0)
94              megadeck->cards_left = create_megadeck(megadeck);
95
96          // random: 1 - cards_left
97          /* random é o número de nós a seguir na lista, por isso,
98           * tem de ser pelo menos 1 (dummy head node),
99           * ou no máximo o número de nós (se seguirmos *cards_left
100          * nós a partir do dummy head node, chegamos à tail) */
101         random = rand() % megadeck->cards_left + 1;
102         List *random_node = megadeck->deck;
103         for (int i = 0; i < random; i++) {
104             if (random_node->next != NULL)
105                 random_node = random_node->next;
106             else {
107                 fprintf(stderr, "Erro: tentou-se dar uma carta não existente.\n");
108                 exit(EXIT_FAILURE);
109             }
110         }
111
112         count_cards((Card *) random_node->payload, megadeck);
113
114         stack_push(&(player->cards), random_node->payload);
115         player->num_cards++;
116         list_remove(random_node);
117         megadeck->cards_left--;
118     }
119
120     int create_megadeck(Megadeck *megadeck)
121     {
122         int total_cards = 0;
123         Card *cur_card = NULL;
124
125         for (int i = 0; i < megadeck->num_decks; i++)
126             for (int j = 0; j < 4; j++)
127                 for (int k = 0; k < SUIT_SIZE; k++) {
128                     cur_card = (Card *) ecalloc(1, sizeof(Card));
129                     cur_card->suit = j;
130                     cur_card->id = k;
131                     list_append(megadeck->deck, cur_card);
132                 }
133
134         total_cards = megadeck->num_decks * DECK_SIZE;
135         return total_cards;
136     }
137
138     void new_game(List *players, Player *house, Megadeck *megadeck)
139     {
140         // só fazer new_game quando já toda a gente jogou
```

```
141         if (find_active_player(players) != NULL) {
142             return;
143         }
144 
145         // só fazer new_game quando houver jogadores para jogar
146         if (find_ingame_player(players) == NULL) {
147             return;
148         }
149 
150         // atualizar as contagens das EAs com os valores da ronda anterior,
151         // antes de qualquer carta ser distribuída na nova ronda.
152         update_count(players, megadeck);
153 
154         // Limpar cartas e retirar apostas
155         clear_cards_take_bet(players, house, megadeck);
156         // Dar cartas
157         distribute_cards(players, house, megadeck);
158         // Encontrar qual o jogador que começa a jogar
159         find_playing(players, house);
160 
161         // Se a casa tiver blackjack...
162         if (house->status == BJ) {
163             // ...a ronda acaba logo.
164             pay_bets(players, house);
165             return;
166         }
167     }
168 
169     // limpar cartas e retirar apostas aos jogadores que possam jogar
170     void clear_cards_take_bet(List *players, Player *house, Megadeck *megadeck)
171     {
172         List *aux = players->next;
173         Player *cur_player = NULL;
174         while (aux != NULL) {
175             cur_player = (Player *) aux->payload;
176             if (cur_player->money < cur_player->bet)
177                 cur_player->ingame = false;
178 
179             // Se o jogador jogou na ronda antes
180             if (cur_player->ingame) {
181                 // Limpar cartas antigas
182                 destroy_stack(&cur_player->cards);
183                 cur_player->num_cards = 0;
184 
185                 // Verificar se o jogador pode jogar outra vez e
186                 // retirar as apostas a todos os jogadores
187                 // (apenas fazemos o cálculo dos dinheiros no final da ronda!)
188                 if (cur_player->type == HU) {
189                     if (cur_player->money < cur_player->bet)
190                         cur_player->ingame = false;
191                     else
192                         cur_player->money -= cur_player->bet;
193                 }
194                 else if (cur_player->type == EA) {
195                     // O jogador é EA, chamar hi_lo para modificar
196                     // a sua aposta antes de esta ser retirada,
197                     // de acordo com a estratégia hi-lo.
198                     #ifdef HI_LO
199                     hi_lo(cur_player, megadeck);
200                     #endif
201                     cur_player->money -= cur_player->bet;
202                 }
203             }
204             aux = aux->next;
205         }
206         destroy_stack(&house->cards);
207     }
208 
209     void distribute_cards(List *players, Player *house, Megadeck *megadeck)
210     {
```

```
211      List *aux = players->next;
212      Player *cur_player = NULL;
213      int public_house_points = 0;
214
215      // Distribuir cartas realisticamente
216      for (int i = 0; i < 2; i++) {
217          aux = players->next;
218          cur_player = NULL;
219          while (aux != NULL) {
220              cur_player = (Player *) aux->payload;
221              // se puder jogar...
222              if (cur_player->ingame)
223                  // dar uma carta
224                  give_card(cur_player, megadeck);
225              aux = aux->next;
226          }
227          if (i == 1)
228              // segunda vez, mostrar apenas os pontos da casa até agora
229              public_house_points = count_points(house);
230          give_card(house, megadeck);
231      }
232
233      house->num_cards = 1; // desenhar só uma carta
234      count_points(house);
235      if (house->points == 21)
236          house->status = BJ;
237      else {
238          house->points = public_house_points;
239          house->status = WW;
240      }
241  }
242
243  /*
244   * Encontrar o primeiro jogador a jogar a seguir ao new_game()
245   */
246  void find_playing(List *players, Player *house)
247  {
248      bool found = false;
249      List *aux = players->next;
250      Player *cur_player = NULL;
251      while (aux != NULL) {
252          cur_player = (Player *) aux->payload;
253          if (cur_player->ingame) {
254              // Colocar status a Waiting
255              cur_player->status = WW;
256
257              // Contar pontos e verificar se tem blackjack
258              count_points(cur_player);
259              if (cur_player->points == 21)
260                  cur_player->status = BJ;
261
262              // Dar a vez ao primeiro jogador sem blackjack
263              // E não dar a vez a ninguém se a casa tiver blackjack
264              if (!(house->status == BJ) && !(cur_player->status == BJ) && !foun
     d) {
265                  cur_player->playing = true;
266                  found = true;
267              }
268              else
269                  cur_player->playing = false;
270
271          }
272          aux = aux->next;
273      }
274  }
275
276  /*
277   * Série de três funções auxiliares para encontrar certos jogadores na lista.
278   */
279  List *find_ingame_player(List *players)
```

```
280 {
281     List *aux = players->next; // dummy head
282     Player *cur_player = NULL;
283     while (aux != NULL) {
284         // iterar até ao jogador que está a jogar
285         cur_player = (Player *) aux->payload;
286         if (cur_player->ingame)
287             break;
288         else
289             aux = aux->next;
290     }
291 
292     return aux;
293 }
294 
295 List *find_active_player(List *players)
296 {
297     List *aux = players->next; // dummy head
298     Player *cur_player = NULL;
299     while (aux != NULL) {
300         // iterar até ao jogador que está a jogar
301         cur_player = (Player *) aux->payload;
302         if (cur_player->playing)
303             break;
304         else
305             aux = aux->next;
306     }
307 
308     return aux;
309 }
310 
311 /*
312  * Esta função é chamada na main apenas para
313  * registar as teclas de hit, stand, etc. se o jogador
314  * for humano (ou seja para as teclas não afetarem as
315  * jogadas dos jogadores EA mesmo se primidas)
316  */
317 List *find_active_human_player(List *players)
318 {
319     List *aux = players->next; // dummy head
320     Player *cur_player = NULL;
321     while (aux != NULL) {
322         // iterar até ao jogador que está a jogar
323         cur_player = (Player *) aux->payload;
324         if (cur_player->playing && cur_player->type == HU)
325             break;
326         else
327             aux = aux->next;
328     }
329 
330     return aux;
331 }
332 
333 void surrender(List *players, Player *house, Megadeck *megadeck)
334 {
335     List *aux = find_active_player(players);
336     if (aux == NULL) {
337         return;
338     }
339 
340     Player *cur_player = (Player *) aux->payload;
341 
342     cur_player->status = SU;
343     stand(players, house, megadeck);
344 }
345 
346 bool double_bet(List *players, Player *house, Megadeck *megadeck)
347 {
348     List *aux = find_active_player(players);
349     // não fazer nada se não for a vez dum jogador
```

```c
    if (aux == NULL) {
        return false;
    }

    Player *cur_player = (Player *) aux->payload;

    // não fazer nada se o jogador não pode fazer double
    if (cur_player->money < cur_player->bet || cur_player->num_cards != 2) {
        return false;
    }

    cur_player->money -= cur_player->bet;
    cur_player->bet += cur_player->bet;

    player_hit(players, house, megadeck);
    if (!(cur_player->status == BU)) {
        stand(players, house, megadeck);
    }

    return true;
}

void bet(List *players)
{
    List *aux = find_active_player(players);
    if (aux != NULL) {
        return;
    }

    get_new_bet(players);
}

AddPlayerError add_player(List *players, List *old_players, Megadeck *megadeck
, SDL_Window *window)
{
    int pos = get_clicked_player();

    if (pos == 0) {
        // Sinalizar que o utilizador clicou fora da área de jogadores
        return OUT;
    }

    List *aux = list_follow(players, pos);
    Player *old_player = (Player *) aux->payload;
    if (old_player->ingame) {
        // Sinalizar que o utilizador clicou num jogador que ainda está a joga
r
        return NOTEMPTY;
    }

    show_add_player_input_message(window);

    Player *new_player = get_new_player(pos);
    // subtrair a count da ronda para, quando somarmos em new_game(),
    // ficar a zero.
    if (new_player->type == EA)
        new_player->count -= megadeck->round_count;

    old_player = (Player *) list_remove_pos(players, pos);
    list_append(old_players, old_player);
    list_insert_pos(players, pos, new_player);

    return OK;
}

/*
 * Esta função faz o stand dos jogadores, mas serve para
 * encontrar o jogador que joga a seguir, por isso é chamada sempre
 * no final da vez do jogador.
 */
```

```
418  void stand(List *players, Player *house, Megadeck *megadeck)
419  {
420      List *aux = find_active_player(players);
421      Player *cur_player = NULL;
422      bool end_of_round = false;
423
424      // Se não encontrarmos um jogador a jogar...
425      if (aux == NULL) {
426          // não fazer nada
427          return;
428      }
429
430      // se encontrarmos, fazer-lhe stand, se este não tiver outros atributos
431      cur_player = (Player *) aux->payload;
432      if (cur_player->status == WW)
433          cur_player->status = ST;
434      cur_player->playing = false;
435
436      // passar ao próximo jogador
437      aux = aux->next;
438
439      if (aux != NULL) {
440          // se este próximo jogador existir,
441          // procurar o próximo jogador válido a seguir
442          while (aux != NULL) {
443              cur_player = (Player *) aux->payload;
444              if (cur_player->ingame && !(cur_player->status == BJ))
445                  break;
446              else
447                  aux = aux->next;
448          }
449          // se ele existir, dar-lhe a vez
450          if (aux != NULL) {
451              cur_player->playing = true;
452          }
453          else {
454              // não existe um próximo jogador válido para jogar
455              end_of_round = true;
456          }
457      }
458      else {
459          // se não existir um próximo jogador, fizemos stand do último jogador
460          end_of_round = true;
461      }
462
463      if (end_of_round) {
464          // A ronda acabou, fazer o hit da casa e distribuir o dinheiro
465          house_hit(house, megadeck);
466          pay_bets(players, house);
467      }
468  }
469
470  void player_hit(List *players, Player *house, Megadeck *megadeck)
471  {
472      List *aux = find_active_player(players);
473      Player *cur_player = NULL;
474
475      if (aux != NULL) {
476          cur_player = (Player *) aux->payload;
477      }
478      else {
479          return;
480      }
481
482      give_card(cur_player, megadeck);
483      count_points(cur_player);
484
485      if (cur_player->points > 21) {
486          cur_player->status = BU;
487      }
```

```
488
489     if (cur_player->points >= 21) {
490         stand(players, house, megadeck);
491     }
492 }
493
494 void house_hit(Player *house, Megadeck *megadeck)
495 {
496     house->num_cards = 2;
497
498     while (house->points <= 16) {
499         give_card(house, megadeck);
500         count_points(house);
501     }
502
503     if (house->points > 21) {
504         house->status = BU;
505     }
506 }
507
508
509 /*
510  * Esta função distribui o dinheiro no final da ronda e é
511  * chamada sempre nessa altura.
512  */
513 void pay_bets(List *players, Player *house)
514 {
515     List *aux = players->next;
516     Player *cur_player = NULL;
517     while (aux != NULL) {
518         cur_player = ((Player *) aux->payload);
519
520         // not playing
521         if (!cur_player->ingame) {
522             // skip this player
523             aux = aux->next;
524             continue;
525         }
526
527         // surrender
528         if (cur_player->status == SU) {
529             house->money -= cur_player->bet / 2;
530             cur_player->money += cur_player->bet / 2;
531             cur_player->losses++;
532         }
533         // blackjack casa e do jogador: tie
534         else if (cur_player->status == BJ && house->status == BJ) {
535             cur_player->money += cur_player->bet;
536             cur_player->ties++;
537         }
538         // blackjack do jogador: win
539         else if (cur_player->status == BJ && !(house->status == BJ)) {
540             cur_player->money += 2*cur_player->bet + cur_player->bet/2;
541             house->money -= cur_player->bet + cur_player->bet/2;
542             cur_player->wins++;
543         }
544         // blackjack da casa: loss
545         else if (!(cur_player->status == BJ) && house->status == BJ) {
546             house->money += cur_player->bet;
547             cur_player->losses++;
548         }
549         // bust da casa e do jogador: loss
550         else if (cur_player->status == BU) {
551             house->money += cur_player->bet;
552             cur_player->losses++;
553         }
554         // bust da casa: win
555         else if (!(cur_player->status == BU) && house->status == BU) {
556             cur_player->money += 2*cur_player->bet;
557             house->money -= cur_player->bet;
```

```
558             cur_player->wins++;
559         }
560         // empate mesmos pontos: tie
561         else if (cur_player->points == house->points) {
562             cur_player->money += cur_player->bet;
563             cur_player->ties++;
564         }
565         // jogador ganha com mais pontos: win
566         else if (cur_player->points > house->points) {
567             cur_player->money += 2*cur_player->bet;
568             house->money -= cur_player->bet;
569             cur_player->wins++;
570         }
571         // house ganha com mais pontos: loss
572         else if (cur_player->points < house->points) {
573             house->money += cur_player->bet;
574             cur_player->losses++;
575         }
576         else {
577             // isto nunca pode acontecer
578             fprintf(stderr, "Erro: estado de jogador desconhecido.\n");
579             exit(EXIT_FAILURE);
580         }
581         aux = aux->next;
582     }
583 }
584
585
586 /*
587  * Conta os pontos do jogador.
588  * No primeiro loop atribui a todos os ases 11 pontos.
589  * No segundo loop se pontos > 21 remove 10 pontos ate não haverem ases.
590  */
591 int count_points(Player *player)
592 {
593     Stack *cards = player->cards;
594     int num_ace = 0;
595
596     player->points = 0;
597     while (cards != NULL) {
598         player->points += point_index(cards->card->id);
599         if (cards->card->id == 12)
600             num_ace++;
601         cards = cards->next;
602     }
603
604     while (player->points > 21 && num_ace > 0) {
605         player->points -= 10;
606         --num_ace;
607     }
608
609     return player->points;
610 }
611
612
613 // Conversão do id da carta para pontos
614 int point_index(int id)
615 {
616     int points = 0;
617     id %= 13;
618     if (id == 12)
619         points = 11;
620     else if (id <12 && id>8)
621         points = 10;
622     else
623         points = id + 2;
624     return points;
625 }
626
627 /*
```

```c
 * Série de três funções que libertam a diversa memória alocada
 * ao longo do programa, como as pilhas dos jogadores e a lista dos jogadores
 */
void destroy_list(List *head)
{
    List *aux = head->next; // dummy head
    List *tmp = NULL;
    while (aux != NULL) {
        tmp = aux;
        aux = tmp->next;
        free(tmp->payload);
        free(tmp);
    }
    free(head);
}

void destroy_stack(Stack **cards)
{
    while (*cards != NULL)
        free(stack_pop(cards));
}

void destroy_players_list(List *players)
{
    List *aux = players->next;
    Player *cur_player = NULL;
    while (aux != NULL) {
        cur_player = (Player *) aux->payload;
        destroy_stack(&cur_player->cards);
        aux = aux->next;
    }
    destroy_list(players);
}
```