SSC0600 e SSC0601 - Trabalho 05

Prof. Adenilso da Silva Simão
Estagiários PAE: Jorge Francisco Cutigi, Leo Natan Paschoal e Samuel Lopes

Alunos:	N°USP:
Roberto Kazushi Yuuki Junior	11395815
Tsuyoshi Sonobe	10739246

Qwirkle

Links

Link para o vídeo de resolução/discussão do trabalho:

https://drive.google.com/file/d/1Fa5bo4KFfDkhP-DHB5o2YvkwLtOvFj9L/view?usp=sharing

Link para o vídeo de teste:

https://drive.google.com/file/d/1fLC2qKubvcrbWRLgDz78KqtZI6_MGmIQ/view?usp=sharing

Descrição da solução

Para melhor organizar a implementação do jogo Qwirkle, dividimos as funções utilizadas em 3 arquivos *.c* com seus respectivo *.h*: *deck.c*, *jogo.c* e *jogadores.c*. No arquivo *deck.c* estão as funções relacionadas às peças do jogo. No *jogo.c* estão as funções relacionadas ao tabuleiro do jogo. Já no arquivo *jogadores.c* estão as funções para distribuição das peças entre os jogadores, ações dos jogadores (trocar, passar, jogar), verificação das regras do jogo e contagem dos pontos. Além disso, temos na *main.c* a organização do andamento do jogo pela chamada das funções.

Deck.c

Funções:

- **(pontDeque) "gerarMonte()":** retorna um ponteiro para uma estrutura deque, essa estrutura contém 3 cartas de cada uma das 36 combinações possíveis de formato e cores das peças na ordem crescente de todas as cores possíveis de cada formato. Primeiro declara-se dois vetores que armazenam os formatos e cores possíveis e logo

após aloca-se memória para o deque. Por meio de 3 estruturas de repetição do tipo *for* gera-se todas as 108 cartas com as informações das 108 peças do jogo, o primeiro for controla a repetição das cartas, o segundo controla os formatos possíveis e o mais interno controla as cores possíveis. Por fim, retorna-se o ponteiro para essa estrutura deque com as todas as 108 cartas organizadas.

- **(pontDeque) "iniciarDeque()":** essa função inicia o deque e aloca todas as 108 peças nela. Inicialmente na função, é alocado dinamicamente a estrutura do deque em *aux*. Em seguida chama a função *gerarMonte* para receber todas as peças existentes no jogo e gera números aleatórios que representam quantas peças serão passadas até que a peça seja a sorteada. Então a peça sorteada é retirada do monte e adicionada ao início da estrutura do deque. Ponteiros foram utilizados para que sempre que uma peça fosse retirada do monte, o ponteiro da peça anterior apontasse para a peça apontada pela peça retirada a fim de não perder os endereços e informações das peças seguintes. Esse processo é feito dentro de um laço while com condição do monte com todas as peças do jogo ainda ter peças, ou seja, até que todas as peças sejam sorteadas de maneira aleatória.
- **(void) "resetarDeque(pontDeque deque):** essa função recebe um deque e libera a memória alocada para o deque em si e todas as cartas contidas nele. Para isso, percorre-se todas as cartas do deque enquanto houver cartas e a função *free* é utilizada em cada ponteiro para carta.

Jogadores.c

Funções:

- **(void) "iniciarPiecesJog(pontDeque monte, pontJogadores jog):** função que recebe como parâmetros um ponteiro para struct do tipo *deque* e um ponteiro para struct do tipo *jogadores*. Inicialmente é declarada uma variável *temp* do tipo *pontCarta* que é um ponteiro para a struct *aux*. Então dentro de um for de 0 a 6, copia as 6 primeiras peças do monte do jogo (todas as peças disponíveis já "embaralhadas") para as peças do jogador e as remove do monte. Por fim atribui o valor 6 ao número de peças do jogador.

- **(void) "trocarPiecesJog(pontDeque monte, pontJogadores jog, int pos):** função que recebe como parâmetros um ponteiro para struct do tipo *deque*, um ponteiro para struct do tipo *jogadores* e uma variável do tipo int que indica a posição da peça a ser trocada no deque do jogador. Na função, primeiramente é alocado dinamicamente, na

variável *temp* do tipo *pontCarta*, memória para uma struct do tipo *aux* que contém o formato e cor de uma peça. Após isso, através do remanejamento de ponteiros, o formato e cor da peça do deque do jogador a ser trocada é copiada na variável *temp* (ponteiro para struct do tipo *aux*), então a próxima peça do monte de peças do jogo é copiado para o deque do jogador na posição passada por parâmetro. Por fim remove-se a primeira posição do monte e o ponteiro de início aponta para a próxima posição.

- **(void) "printarJog(pontJogadores jog)":** a função recebe como parâmetro um ponteiro para struct do tipo *jogadores.* Tal função imprime o nome do jogador, em seguida, através de um for, percorre todas as peças do jogador imprimindo cada uma.

- **(int) "verificarJogada(pont board, char peca, char numero, int linha, int coluna, int *n_jogada, int *todasJogadas)":** essa função recebe como parâmetros: um ponteiro para struct do tipo *board*, utilizada para acessar as posições do tabuleiro, duas variáveis do tipo char, que representam o formato e cor da peça, duas do tipo int, que indicam a posição no tabuleiro em que a peça será jogada, um ponteiro do tipo int que indica o número da jogada atual do jogador e um ponteiro, também do tipo int, que aponta para um vetor responsável por armazenar as jogadas anteriores do jogador. Dentro da função são utilizadas 3 variáveis principais do tipo int: *verifica*, *posAdj* e *pecaIgual.* A variável *verifica* foi inicializada com 1 e é a responsável por armazenar se a posição é válida (valor igual a 1) ou se é inválida (valor igual a 0) e será retorna ao final da função. Já a variável *posAdj* é utilizada para indicar qual a posição da peça adjacente em relação à peça jogada (0 = acima, 1 = esquerda, 2 = direita, 3 = abaixo). A variável *pecaIgual* indica qual atributo da peça jogada é igual à sua peça vizinha (1 = formato, 2 = cor).

  Como utilizamos as regras apenas a partir da segunda jogada, ou seja, quando o tabuleiro é diferente de 1 por 1, então as regras são feitas dentro de um if com tal condição. Para iniciar a verificação das regras do jogo para a colocação de uma peça no tabuleiro, primeiramente foi analisado se a peça vai ser jogada em uma posição que possua alguma peça vizinha, caso não haja, é atribuído o valor 0 a *verifica*, caso haja, verifica se o formato ou a cor das peças são iguais, sendo que se tanto formato quanto cor forem iguais, também é uma jogada inválida (*verifica* = 0), se apenas um atributo for igual, então são armazenados valores a posição da peça adjacente e o atributo em comum nas variáveis *posAdj* e *pecaIgual.* Esse processo é feito para cada

um dos casos: linha e coluna iguais a 0, linha igual a 0 e coluna no tamanho máximo, apenas linha igual a 0, linha no tamanho máximo e coluna igual a 0, linha e coluna iguais ao tamanho máximo, apenas linha no tamanho máximo, apenas coluna igual a 0, apenas coluna com tamanho máximo e posição não está em nenhuma borda do tabuleiro. Esses casos foram utilizados para que na verificação das posições adjacentes a função não tentasse acessar posições inexistentes, provocando um erro de *segmentation fault.*

Após essa verificação em relação às casas adjacentes, caso a jogada ainda seja válida (*verifica == 1*), é analisado se a posição em que a peça será jogada está em uma linha ou coluna já ocupada por um formato ou cor, para isso, são utilizadas as variáveis *posAdj* e *pecaIgual.* A posição da peça adjacente é utilizada para saber se é necessário analisar a linha, para frente ou para trás, ou a coluna, para cima ou para baixo, o atributo em comum é utilizado para saber se deve-se manter a linha ou coluna com a mesma cor ou com o mesmo formato. Dessa forma, para cada um dos casos de posição adjacente é analisado se o atributo igual a da peça vizinha é igual a todas as peças na linha, até encontrar uma posição vazia. Caso seja diferente, a jogada é inválida.

Por fim, após as duas verificações amteriores, se a jogada ainda seja válida, é analisada a última regra, a partir da terceira jogada do jogador, o jogador deve sempre jogar na mesma linha ou mesma coluna das jogadas anteriores. Para fazer isso, foi usado o ponteiro *todasJogadas* passado como parâmetro e que aponta para um vetor com todas as jogadas do jogador (linha e coluna de cada jogada), compara-se então se o jogador jogou em uma mesma linha (posições 0 e 2 do vetor devem ser iguais) ou coluna (posições 1 e 3 do vetor devem ser iguais), caso seja uma linha, a posição em que peça será jogada deverá se manter nessa linha, caso não esteja, a jogada é inválida e *verifica = 1*, o mesmo acontece com o caso da coluna. Então adiciona-se a última jogada ao vetor. Por último, a função retorna o valor de *verifica*, sendo 1 uma jogada válida e 0 uma jogada inválida.

- **(void) "joga(pont board, pontDeque monte, pontJogadores jog, char peca, char numero, int linha, int coluna, int *n_jogada, int *todasJogadas)":** essa função do tipo void recebe como parâmetros: um ponteiro para struct do tipo *board*, um ponteiro para struct do tipo *deque*, um ponteiro para struct do tipo *jogadores*, um char que representa o formato da peça a ser jogada, um char para representar a cor (representado por números) da peça, dois inteiros indicando a posição no tabuleiro em

que a peça será jogada, um ponteiro do tipo int que indica o número da jogada atual do jogador e um ponteiro, também do tipo int, que aponta para um vetor responsável por armazenar as jogadas anteriores do jogador.

Primeiramente na função, inicializa-se a variável *aux* do tipo int que indicará se a posição em que se quer jogar a peça é válida (será igual a 1 caso seja inválida e 0 caso seja válida). Então verifica-se se a posição passada como parâmetro existe no tabuleiro e se está vazia, caso as condições não forem atendidas, a posição é inválida, então o valor de aux é 1, caso as condições forem atendidas, verifica-se então se a peça a ser jogada está disponível dentre as peças do deque do jogador, caso esteja, armazena a posição da peça no deque e atribui 0 ao valor de *aux*, caso contrário, atribui 1 a *aux*. Após isso, é criada uma variável *verifica* para armazenar o valor retornado pela função *verificarJogada.* Caso *verifica* seja igual a 1 e *aux* também, a peça pode ser jogada, então é atribuído o valor da peça (seu formato e sua cor) à posição indicada, o tabuleiro é realocado pela função *reallocBoard*, a peça é retirada do deque do jogador e o número da jogada é atualizado. Caso contrário, a mensagem de jogada inválida é imprimida na tela.

- **(void) "cheatMode(pont board, pontDeque monte, pontJogadores jog, char peca, char numero, int linha, int coluna, int *n_jogada, int *todasJogadas)":** essa função é muito parecida com a função *joga,* pois também verifica se a posição em que a peça será jogada é válida, por isso, recebe os mesmos parâmetros. Portanto a função age da mesma maneira, porém no momento de verificar se a peça está no deque do jogador, caso não esteja, também procura no monte do jogo (restante das peças disponíveis), possibilitando assim que o jogador consiga jogar qualquer peça disponível.

- **(void) "fgetss(char str[], int n, char tipoComando[])":** função feita para leitura de strings sem que haja problemas de buffer ou erro, caso passe do tamanho. Recebe como parâmetros um vetor de tipo char, no qual a string será armazenada, um inteiro que indica a última posição do vetor e um vetor do tipo char. Dentro da função é lido caractere por caractere e armazenado no vetor *str* até que se encontre uma quebra de linha, após isso, caso ainda leia algo diferente de *'\n'* encerra a string com *'\0'* e imprime uma mensagem informando que os caracteres após o limite de tamanho da string serão desconsiderados.

- **(void) “toUpper(char str[])”:** função que recebe uma string como parâmetro e a percorre, se o caractere for uma letra minúscula, então será alterado para uma letra maiúscula. Isso é feito a partir subtraindo 32 do caractere em questão, o que de acordo com a tabela ASCII é o seu correspondente em letra maiúscula.

- **(void) “reporPiecesJog(pontJogadores jog, pontDeque monte)”:** função que recebe como parâmetros um ponteiro para struct do tipo *jogadores,* um ponteiro para struct do tipo *deque*. Na função é utilizado um laço while com condição do monte de peças do jogo ainda ter peças e a quantidade de peças do jogador ser menor do 6, dentro desse laço, adiciona o formato e cor da primeira peça na sequencia do monte às peças do jogador e acrescenta um ao número de peças dele. Em seguida, através de um ponteiro, aponta para a primeira posição do monte de peças do jogo e o desaloca, também remaneja o ponteiro de maneira que passe a apontar para a segunda posição do monte e depois diminui em um o valor do número de peças no monte.

- **(void) “leituraComandos(pont board, pontDeque monte, pontJogadores jog, char cheat)”:** essa função lê os comandos dados pelo jogador e direciona para as funções que desempenham esses comandos. Declara-se 3 *strings* que são iguais aos comandos possíveis: trocar, jogar e passar. Declara-se também os inteiros *aux*, *num_jogada* e o vetor alocado dinamicamente *todasJogadas* que representam, respectivamente, o controle para determinar qual interface de comandos será mostrada, a jogada atual e todas as posições nas quais o jogador colocou uma peça. Entra-se num *while* enquanto o jogador não acionar a função trocar ou passar. Dentro desse *while*, é mostrado o tabuleiro na tela, as informações do jogador e, dependendo do valor de *aux*, os comandos possíveis. Logo após, é realizada a leitura do comando do jogador e separa-se a função do comando digitado. Essa função é armazenada na *string funcao* e comparada com as *strings* declaradas no início. Caso a função seja a de trocar, verifica-se se as peças digitadas são válidas, armazenando-as em um vetor de peças e, caso as peças sejam válidas, aciona-se a função *verificaTroca* que realizará a troca se todas as peças digitadas estejam no vetor de peças do jogador. Caso a função seja a de jogar, as coordenadas dadas são buscadas na *string* digitada pelo jogador, converte-se os caracteres para inteiros e, dependendo do modo *cheat mode* estar ativo, direciona para uma das funções de jogada, a *joga* e *cheatMode*. Caso a função seja a de passar, muda-se o *aux* para sair na próxima verificação do *while*, a função de contar pontos e de repor peças do jogador é acionada se houver

uma jogada antes junto com a mudança do *aux*. Pode ocorrer o caso em que o comando não é igual a nenhuma das *strings* declaradas, o valor de aux é mantido e um aviso é mostrado na tela nesse caso. Ao final da função, imprime-se a quantidade total de pontos do jogador e a quantidade acumulada nessa rodada.

- **(int) "verificaPeca(char formato, char cor)":** essa função verifica se um formato e cor é compatível com alguma das 36 possibilidades de peça. Primeiro, declara-se dois vetores que contém os formatos e cores possíveis. Por meio de duas estruturas do tipo *for*, procura-se o formato e a cor passadas como parâmetro. Por fim, a função retorna 1 se a cor e o formato foi encontrado e 0 caso contrário.

- **(int) "verificaTroca(pontDeque monte, pontJogadores jog, piece pecas[], int qtdPecasTroca)":** a função realiza as trocas se todas as peças contidas no vetor de peças *pecas* também estão presentes no vetor de peças do jogador. Por meio de duas estruturas do tipo *for*, é realizada conta-se a quantidade de peças no vetor *pecas* presentes no vetor de peças do jogador. Se esse valor é igual quantidade peças a serem trocadas dada pelo inteiro *qtdPecasTroca*, imprime-se as peças a serem trocadas e as trocas são feitas por meio da função *trocarPiecesJog*. Caso o valor não seja igual, uma mensagem mostra que as peças não são válidas.

- **(pontJogadores) "iniciarJogs(pontDeque monte, int qtdJogs)":** essa função é responsável por alocar memória para a quantidade de jogadores passada como parâmetro, distribuir as peças, zerar os pontos e ligar os jogadores através dos ponteiros. Em um *while* que é repetido a quantidade de jogadores dada, aloca-se memória para um jogador, a função *iniciarPiecesJog* é acionada para esse jogador e os pontos são zerados. Dependendo do jogador, mudam-se os ponteiros presentes no jogador para que o anterior aponte para o atual. Retorna-se, por fim, o ponteiro para o primeiro jogador.

- **(void) "selecionarVencedor(pontJogadores inicio)":** a função seleciona o vencedor ou vencedores de acordo com os seguintes critérios em ordem de importância: maior quantidade de pontos e menor quantidade de peças do jogador. Declara-se os inteiros *maiorPontos*, *qtdPiecesVencedo e qtdVencedores*. Por meio de um *while*, percorre-se todos os jogadores e compara-se os dados do jogador atual com os valores armazenados nos inteiros seguindo os critérios para seleção do vencedor. Caso a

quantidade de pontos seja maior que *maiorPontos*, são atualizados o valor dos 3 inteiros declarados inicialmente. Caso haja um empate de pontos, se quantidade de peças do jogador atual os mesmos passos do caso anterior são realizados e é incrementado a quantidade de vencedores se houver outro empate. Por fim, o vencedor ou os vencedores são mostrados na tela com as suas quantidades de pontos e de peças.

- **(void) "contarPontos(pontJogadores jog, int *todasJogadas, int tam, pont board)":** essa função adiciona os pontos acumulados em uma rodada por um jogador baseando-se nas coordenadas das jogadas armazenadas no vetor *todasJogadas*. Caso seja realizada pelo menos 2 jogadas, calcula-se a diferença entre as linhas e as colunas das últimas duas coordenadas do vetor, se algum deles for 0 significa que as jogadas foram em linha ou em coluna. Baseado na última informação, calcula-se a quantidade de pontos considerando apenas as coordenadas no vetor por meio de uma das duas funções de contar posições(*contarPosicoesHorizontal ou contarPosicoesVertical)*. Depois, conta-se os pontos acumulados na direção oposta em cada uma das coordenadas do vetor. De forma semelhante, conta-se os pontos para o caso de apenas uma coordenada no vetor. Caso haja *qwirkle,* os pontos são dobrados em uma linha ou coluna, os pontos das mesma são dobrados.

- **(int) "contarPosicoesHorizontal(pont board, int posHInicial, int posVInivial)":** essa função conta as peças conectadas na direção horizontal à peça na coordenada passada como parâmetro. Primeiro, incrementa-se a quantidade de posições até que uma posição vazia seja atingida à direita e depois o mesmo é feito para posições à esquerda. Isso é feito por duas estruturas do tipo *while* que, enquanto o formato da posição atual for diferente de um espaço, incrementa ou decrementa o índice da coluna. Ao final, essa quantidade armazenada no inteiro *pontos* é retornada.

- **(int) "contarPosicoesHorizontal(pont board, int posHInicial, int posVInivial)":** assim como na função anterior, conta-se as posições conectadas à posição passada como parâmetro na vertical, primeiro calculando as posições acima e depois as posições abaixo e, por fim, retornando o valor encontrado. A maneira como essa função funciona é análoga à função anterior, variando o índice das linhas até que posições vazias sejam encontradas tanto acima quanto abaixo.

Jogo.c

Funções:

- **(pont) "startBoard()":** essa função aloca memória para uma estrutura do tipo *board* com apenas uma posição vazia, isto é, 1 linha e 1 coluna e retorna o ponteiro para essa estrutura. Primeiro, o *malloc* é utilizado para alocar a memória para a estrutura do tipo *board*, o número de linhas e colunas são definidos como 1, aloca-se memória para um ponteiro para ponteiro da estrutura *piece* e aloca-se memória para uma estrutura piece. Os campos de cor e formato dessa única posição é preenchida com espaços e, por fim, ponteiro para o *board* é retornado.

- **(void) "reallocBoard(pont board, int linha, int coluna, int tam, int *todasJogadas)":** essa função é responsável por realocar memória para o tabuleiro toda vez que a última posição em que uma peça foi colocada estiver localizada em uma das quatro bordas do tabuleiro. Caso a última peça for adicionada na linha 0 do tabuleiro, aumenta-se a quantidade de linhas por meio de um *realloc* e um *malloc*, o primeiro para aumentar a quantidade de ponteiros para ponteiros do tipo *piece* e o segundo para alocar memória para a nova linha em si. Como o *realloc* adiciona mais uma posição no final do vetor de ponteiros para ponteiros, um *for* é necessário para mudar todas as linhas para uma linha abaixo. Por fim, preenche-se essas novas posições com espaços. De forma análoga, o mesmo ocorre quando a última peça for adicionada na coluna 0 do tabuleiro. Para os casos em que a peça foi colocada no extremo direito ou/e inferior, não é necessário mudar as posições das peças do tabuleiro. Nos casos em que é necessário mudar as posições das peças também é necessário atualizar as coordenadas contidas no vetor *todasJogadas*.

- **(void) "resetBoard(pont board)":** essa função percorre todas as posições do vetor de ponteiros para ponteiros de peças e libera a memória alocada para cada uma delas, por fim libera a memória dedicada ao vetor em si e, no final, realiza o *free* no ponteiro para o *board*.

- **(void) "printBoard(pont board)":** essa função imprime todas as informações de cada peça posicionada no tabuleiro. Por meio de um *for,* a informação das coordenadas das colunas são colocadas na tela, logo após, duas estruturas do mesmo tipo encadeadas, percorrem todas as peças e as informações de formato e cor de cada peça é colocada na tela.

Main.c

No arquivo *main.c* foi feita a organização do andamento do jogo através da chamada das funções dos outros arquivos já apresentados. Para isso, primeiramente foi definida uma variável *controleJogar* do tipo int inicialmente com o valor 1 que serve para controlar o while. O while foi utilizado para dar a opção do jogador poder jogar novamente, dentro desse laço inicia-se o tabuleiro e o monte, após isso faz a leitura da quantidade de jogadores, com uma mensagem caso o número de jogadores seja inválido, em seguida são iniciados os jogadores e se faz a leitura do nome de cada um desses jogadores. Logo depois, é verificado se o usuário deseja jogar em cheat mode ou não, também com tratamento de possíveis erros de entrada inválida. Então inicia-se um while que se repetirá enquanto houver cartas no monte, dentro desse laço o jogo de fato acontecerá, a partir da chamada da função *leituraComandos* para cada jogador e em seguida passando para o próximo jogador. Por fim, após sair do laço while, é chamada a função para verificar o vencedor do jogo e é perguntado se o usuário deseja jogar novamente, caso deseje, continua no laço while, caso contrário, sai do laço e o programa se encerra.