Pontifícia Universidade Católica
do Rio de Janeiro

INF1301 - Programação Modular - 2016.1

Prof. Flavio Bevilacqua

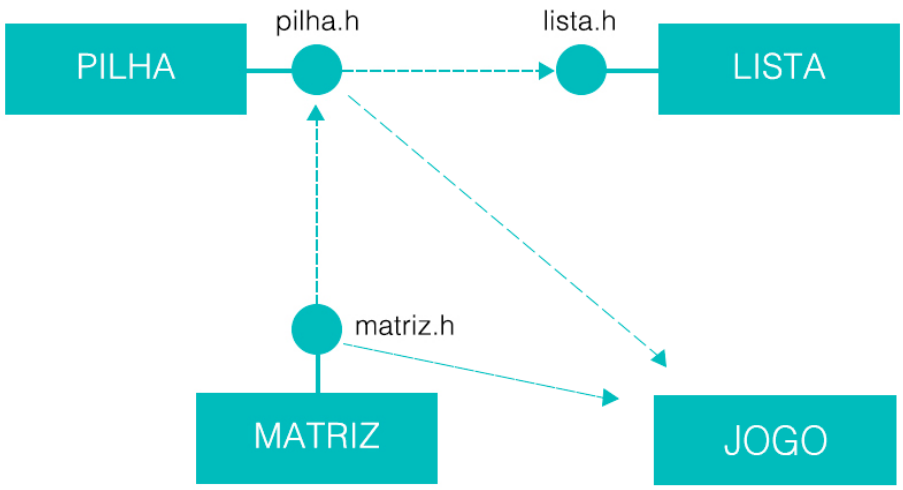
TRABALHO 3 - Arquitetura do Programa e Argumentação de Corretude

Gabriel Da Silva Gomes - 1412845

Gustavo Severo Barros - 1421713

João Pedro Masset Lacombe Dias Garcia - 1211768

1. Modelo de Arquitetura



Funções disponibilizadas em cada interface:

I. Lista - LIS

LIS_tppLista LIS_CriarLista (void (* ExcluirValor) (void * pDado));

Assertivas de Entrada:

- Deve existir um ponteiro para a função que processa a exclusão do valor referenciado pelo elemento a ser excluído;

Assertivas de Saída:

- Se existe memória disponível, a lista é criada e seu ponteiro aponta para NULL;

void LIS_DestruirLista(LIS_tppLista pLista);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista a ser destruida;

Assertivas de Saída:

- Se a lista existe, ela é esvaziada, destruida e seu ponteiro aponta para NULL;

void LIS_EsvaziarLista(LIS_tppLista pLista);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista a ser esvaziada;

Assertivas de Saída:

- Se a lista existe, ela é esvaziada;

LIS_tpCondRet LIS_InserirElementoAntes(LIS_tppLista pLista, void * pValor);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista a ser utilizada;
- Deve existir um valor a ser inserido na lista;

Assertivas de Saída:

- Se a lista existe e existe memória disponível, o valor é inserido antes do valor corrente;

LIS_tpCondRet LIS_InserirElementoApos(LIS_tppLista pLista, void * pValor);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista a ser utilizada;
- Deve existir um valor a ser inserido na lista;

Assertivas de Saída:

- Se a lista existe e existe memória disponível, o valor é inserido após o valor corrente;

LIS_tpCondRet LIS_ExcluirElemento(LIS_tppLista pLista);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista a ser utilizada;

Assertivas de Saída:

- Se a lista existe e ela não é vazia, o elemento corrente da lista é excluido;

char LIS_ObterValor(LIS_tppLista pLista);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista a ser utilizada;

Assertivas de Saída:

- Se a lista existe, se não for vazia, a referência para o elemento corrente da lista é retornado;

void IrInicioLista(LIS_tppLista pLista);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista a ser utilizada;

Assertivas de Saída:

- Se a lista existe, se não for vazia, torna corrente o primeiro elemento da lista;

void IrFinalLista(LIS_tppLista pLista);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista a ser utilizada;

Assertivas de Saída:

- Se a lista existe, se não for vazia, torna corrente o último elemento da lista;

LIS_tpCondRet LIS_AvancarElementoCorrente(LIS_tppLista pLista, int numElem);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista a ser utilizada;
- Deve existir um valor representando o número de elementos que o elemento corrente deve avançar;

Assertivas de Saída:

- Se a lista existe, se não for vazia e o valor for encontrado, a referência do valor é retornada;

LIS_tpCondRet LIS_ProcurarValor(LIS_tppLista pLista, void * pValor);

Assertivas de Entrada:

- Deve existir um ponteiro para a lista a ser utilizada;
- Deve existir um ponteiro para o valor a ser procurado;

Assertivas de Saída:

- Se a lista existe, se não for vazia, a referência do valor procurado é o elemento corrente;

II. Matriz - MAT

MAT_tpCondRet MAT_CriarMatriz(MAT_tpMatriz **pMatriz, int NumeroLinhas, int NumeroColunas);

Assertivas de Entrada:

- Deve existir um valor representando o número de linhas da matriz a ser criada;
- Deve existir um valor representando o número de colunas da matriz a ser criada;
- Deve existir um ponteiro por onde será passado, por referência, a matriz a ser criada;

Assertivas de Saída:

- Se o número de linhas e/ou colunas não for igual ou menor que 0 e exista memória disponível, pMatriz é atualizada com um ponteiro para matriz com o valor de linhas e colunas;

MAT_tpCondRet MAT_DestruirMatriz(MAT_tpMatriz ** pMatriz);

Assertivas de Entrada:

- Deve existir um ponteiro para a matriz a ser destruida;

Assertivas de Saída:

- Se a matriz existe, ela é destruida e seu ponteiro aponta para NULL;

MAT_tpCondRet MAT_InserirValor(MAT_tpMatriz *pMatriz, PIL_tppPilha pPilha);

Assertivas de Entrada:

- Deve existir um ponteiro para a matriz a ser utilizada;
- Deve existir um ponteiro para a pilha a ser inserida;

Assertivas de Saída:

- Se a matriz existe e seu elemento corrente existe, ponteiro para pilha é inserido na cabeça da pilha do elemento corrente da matriz;

MAT_tpCondRet MAT_AdicionarLinha(MAT_tpMatriz * pMatriz);

Assertivas de Entrada:

- Deve existir um ponteiro para a matriz a ser utilizada;

Assertivas de Saída:

- Se a matriz existe e existe memória disponível, um nova linha é adicionada a matriz e o número de linhas é incrementada;

MAT_tpCondRet MAT_AdicionarColuna(MAT_tpMatriz * pMatriz);

Assertivas de Entrada:

- Deve existir um ponteiro para a matriz a ser utilizada;

Assertivas de Saída:

- Se a matriz não é nula e existe memória disponível, um nova coluna é adicionada a matriz e o número de colunas é incrementada;

MAT_tpCondRet MAT_RemoverLinha(MAT_tpMatriz * pMatriz);

Assertivas de Entrada:

- Deve existir um ponteiro para a matriz a ser utilizada;

Assertivas de Saída:

- Se a matriz não é nula e seu elemento corrente existe, a última linha da matriz é removida e o número de linhas é decrementada;

MAT_tpCondRet MAT_RemoverColuna(MAT_tpMatriz * pMatriz);

Assertivas de Entrada:

- Deve existir um ponteiro para a matriz a ser utilizada;

Assertivas de Saída:

- Se a matriz não é nula e seu elemento corrente existe, a última coluna da matriz é removida e o número de linhas é decrementada;

MAT_tpCondRet MAT_IrParaCoordenada(MAT_tpMatriz * pMatriz, int linha, int coluna);

Assertivas de Entrada:

- Deve existir um ponteiro para a matriz a ser utilizada;
- Deve existir um valor representando o número da linha e da coluna da matriz para onde o nó corrente irá avançar;

Assertivas de Saída:

- Se a matriz não é nula, se seu elemento corrente existe e o número da linha e coluna não for igual ou menor a 0 e esteja dentro do limite do número de linhas e colunas da matriz, o nó corrente da matriz avança para o número da linha e da coluna;

MAT_tpCondRet MAT_IrPara(MAT_tpMatriz * pMatriz, MAT_tpCoord Coordenada);

Assertivas de Entrada:

- Deve existir um ponteiro para a matriz a ser utilizada;
- Deve existir o valor representando a coordenada para onde o nó corrente irá avançar;

Assertivas de Saída:

- Se a matriz não é nula, se seu elemento corrente existe e se sua coordena existe, o nó corrente da matriz avança para a coordenada;

MAT_tpCondRet MAT_ObterValorCorr(MAT_tpMatriz *pMatriz, PIL_tppPilha pPilha);

Assertivas de Entrada:

- Deve existir um ponteiro para a matriz a ser utilizada;

- Deve existir um ponteiro para a pulha a ser utilizada;

Assertivas de Saída:

- Se a matriz não é nula, se seu elemento corrente existe e se sua coordena existe, o ponteiro para pilha contido na cabeça da pilha do elemento corrente da matriz atualiza pPilha;

III. Pilha - PIL

PIL_tppPilha PIL_CriarPilha( );

Assertivas de Entrada:

- Não Possui;

Assertivas de Saída:

- Se existe memória disponível, retorna um ponteiro para cabeça da pilha criada;

PIL_tpCondRet PIL_DestruirPilha(PIL_tpPilha *pPilha);

Assertivas de Entrada:

- Deve existir um ponteiro para a pilha a ser destruida;

Assertivas de Saída:

- Se a pilha não é nula, ela é destruida e seu ponteiro aponta para NULL;

PIL_tpCondRet PIL_EmpilhaValor(PIL_tppPilha *pPilha, char ValorParm);

Assertivas de Entrada:

- Deve existir um ponteiro para a pilha a ser utilizada;
- Deve existir um valor a ser inserido na pilha;

Assertivas de Saída:

- Se existe memória disponível e a pilha não é nula, o valor a ser inserido é inserido no topo da pilha e o número de elementos da pilha é incrementado;

PIL_tpCondRet PIL_DesempilhaValor(PIL_tppPilha *pPilha);

Assertivas de Entrada:

- Deve existir um ponteiro para a pilha a ser utilizada;

Assertivas de Saída:

- Se a pilha não é nula e não é vazia, o valor do topo da pilha é removido e o número de elementos da pilha é decrementado;

PIL_tpCondRet PIL_ObterValorTopo(PIL_tppPilha *pPilha, char * ValorParm);

Assertivas de Entrada:

- Deve existir um ponteiro para a pilha a ser utilizada;
- Deve existir um ponteiro por onde será passado, por referência, o valor do topo da pilha;

Assertivas de Saída:

- Se a pilha não é nula e não é vazia, ValorParm é atualizado;

PIL_tpCondRet PIL_ObterTamanho(PIL_tppPilha *pPilha, int * pTamanhoPilha);

Assertivas de Entrada:

- Deve existir um ponteiro para a pilha a ser utilizada;
- Deve existir um ponteiro por onde será passado, por referência, o tamanho da pilha;

Assertivas de Saída:

- Se a pilha não é nula, pTamanhoPilha é atualizado;

IV. Jogo - JGO

JGO_tpCondRet CriarPilhas(PIL_tppPilha *pVetorPilhas);

JGO_tpCondRet GeraTabuleiro(MAT_tppMatriz *pMatriz, PIL_tppPilha *pVetorPilhas);

JGO_tpCondRet PreencheTabuleiro(MAT_tppMatriz matriz, char * pBaralho, int * pTamanhoBaralho);

JGO_tpCondRet ImprimeTabuleiro(MAT_tppMatriz matriz);

JGO_tpCondRet DestroiTabuleiro(MAT_tppMatriz *pMatriz);

void PreencheBaralho(char *pBaralho);

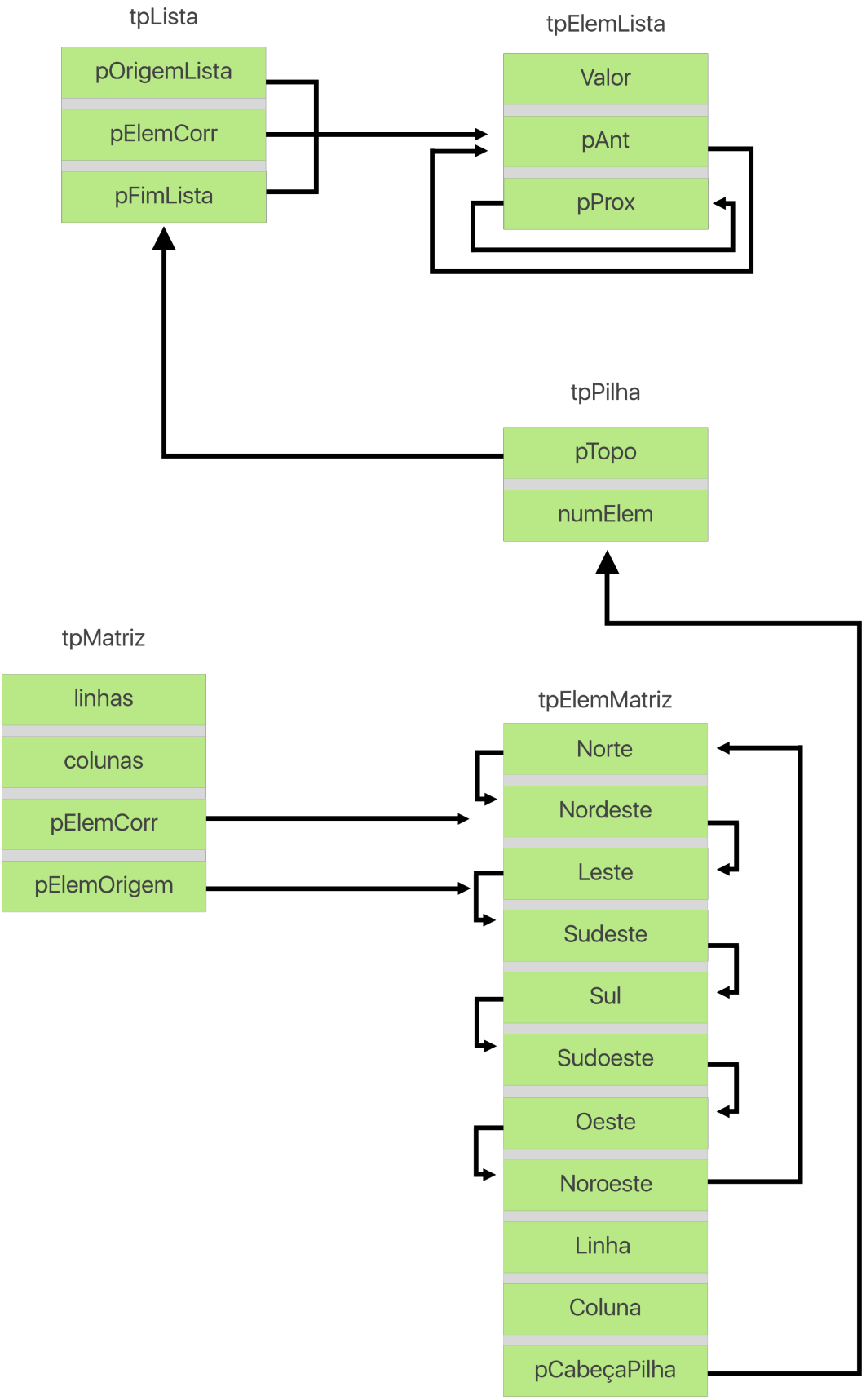
char SorteiaPeca(int i, int j, char *pBaralho, int *pTamanhoBaralho);

JGO_tpCondRet ChecaRetiravel (MAT_tppMatriz* pMatriz, int linha, int coluna, int *retorno);

JGO_tpCondRet RemovePecas (MAT_tppMatriz* pMatriz, int linhaPeca1, int colunaPeca1, int linhaPeca2, int colunaPeca2, int *retorno);

JGO_tpCondRet ChecaGameOver (MAT_tppMatriz* pMatriz, int *retorno);

2. Modelo Estrutural



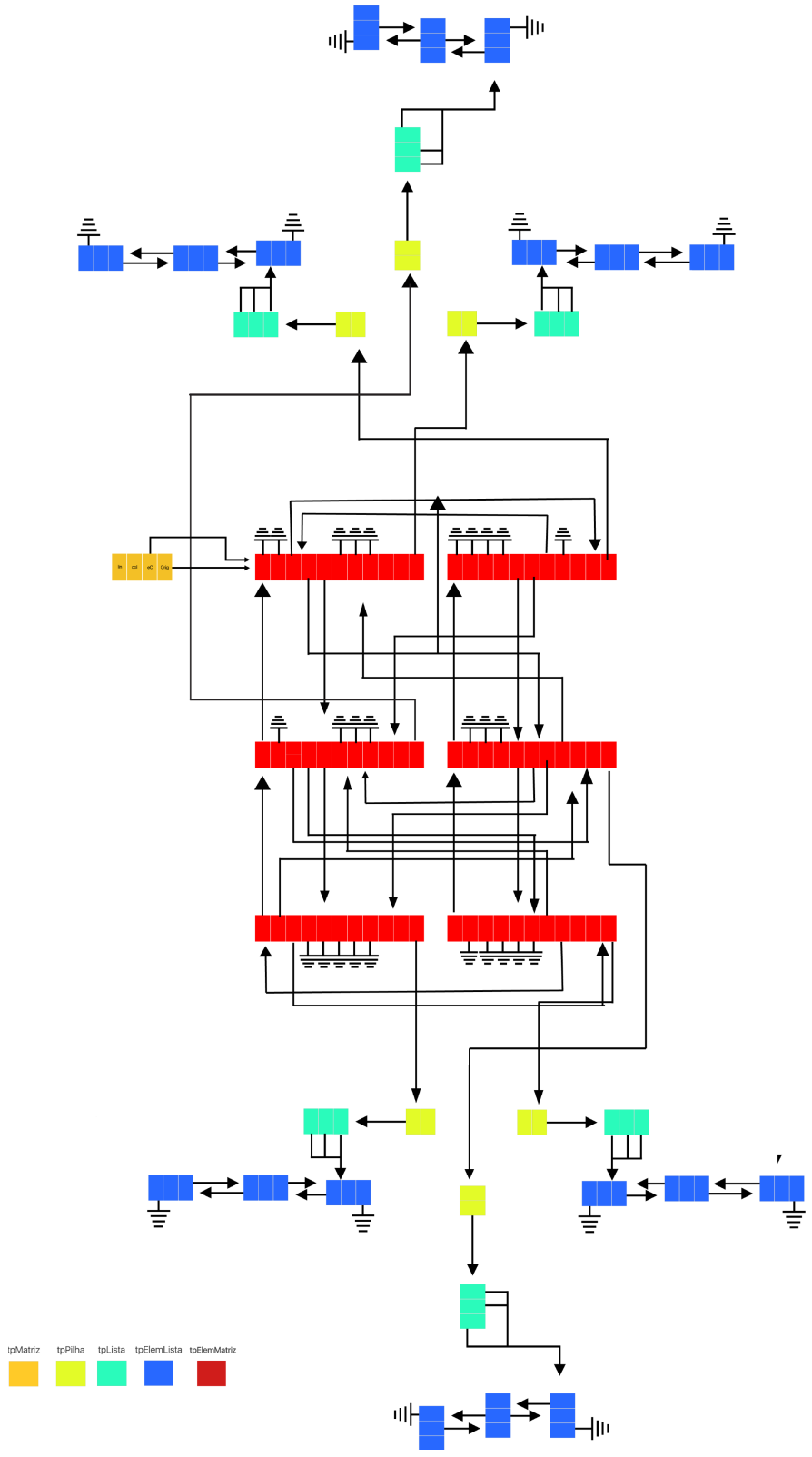
Assertivas Estruturais:

- **Matriz de Pilhas** -> Matriz N/M, com N*M elementos, cada um com um ponteiro para o elemento ao norte, nordeste, leste, sudeste, sul, sudoeste, oeste e noroeste, ponteiro para um tipo pilha e valor linha e coluna. Valem as assertivas estruturais de uma pilha com cabeça estruturada com uma lista duplamente encadeada com cabeça.

- **Pilha** -> Pilha com cabeça com ponteiro para um elemento lista e um valor com número de elementos da pilha. Valem as assertivas estruturais de uma lista duplamente encadeada com cabeça.

- **Lista** -> Valem as assertivas estruturais de uma lista duplamente encadeada com cabeça.

Exemplo



Argumentação de corretude:

- Criar Pilhas

AE

```
JGO_tpCondRet CriarPilhas(PIL_tppPilha *pVetorPilhas)
{
        int countPilhas;
        REP
        for(countPilhas = 0; countPilhas < TAMANHO; countPilhas++)
        {
                pVetorPilhas[countPilhas] = PIL_CriarPilha();
        }

        return JGO_CondRetOK;
} /* Fim função: Criar Pilhas */
```

AS

Repetição:

AE -> existe um ponteiro de pilha;

AS -> Caso a variável “TAMANHO” seja maior que 0, pVetorPilhas possui um vetor de pilhas vazias com o valor da variável “TAMANHO" como o número de pilhas;

AINV -> existem 2 conjuntos:
- a criar
- já criado

1) AE —> AINV: Pela AE, existe um ponteiro para um vetor de pilhas, porém suas pilhas ainda não foram criadas. Seus elementos então estão no conjunto a criar e vale AINV, porque existem 2 conjuntos.

2) AE && (C==F) —> AS: Pela AE, como (C==F), o vetor retornado é vazio. Vale a AS pois o vetor não é nulo pois a variável “TAMANHO" não é maior que 0.

3) AE && (C==T) + B —> AINV: Pela AE, como (C==T), o elemento entra no conjunto já criado e vale AINV.

4) AINV && (C==T) + B —> AINV: Para que AINV continue valendo, B tem que fazer com que o elemento em questão vá do conjunto a criar para já criado.

5) AINV && (C==F) —> AS: Com (C==F), as pilhas já foram criadas ou o “TAMANHO" não é maior que 0, valendo a AS.

6) Término: O conjunto a criar possui um número finito de elementos e a cada ciclo de repetição um deles passa para o conjunto já criado. A repetição só termina após um número finito de ciclos a serem dados.

- Gera Tabuleiro

AE

```
JGO_tpCondRet GeraTabuleiro(MAT_tppMatriz *pMatriz, PIL_tppPilha * pVetorPilhas)
{
	int contadorLinhas, contadorColunas;
	MAT_tpCondRet retornoMatriz;

	retornoMatriz = MAT_CriarMatriz(pMatriz, LINHAS, COLUNAS);

	if (retornoMatriz != MAT_CondRetOK)
	{
		return JGO_CondRetErroMatriz;
	}
```

REP

```
	for(contadorLinhas = 0; contadorLinhas < LINHAS; contadorLinhas++)
	{
		for(contadorColunas = 0; contadorColunas < COLUNAS; contadorColunas+
+)
		{
			retornoMatriz = MAT_IrParaCoordenada(*pMatriz, contadorLinhas,
contadorColunas);

			if (retornoMatriz != MAT_CondRetOK)
			{
				return JGO_CondRetErroMatriz;
			}

			retornoMatriz = MAT_InserirValor(*pMatriz, pVetorPilhas[LINHAS *
contadorLinhas + contadorColunas]);

			if (retornoMatriz != MAT_CondRetOK)
			{
				return JGO_CondRetErroMatriz;
			}
		}
	}

	return JGO_CondRetOK;
} /* Fim função: Gera Tabuleiro */
```

AS

Repetição :

AE -> existe um ponteiro para uma matriz e um ponteiro para pilha;

AS -> Caso a matriz a ser retornada por referência não seja nula, pMatriz recebe por referência uma matriz com pVetorPilhas em suas devidas linhas e colunas;

AINV -> existem 2 conjuntos:

- a criar
- já criado

1) AE —> AINV: Pela AE, existe um ponteiro para um vetor de pilhas e para uma matriz, porém a matriz ainda não foi criada. Seus elementos então estão no conjunto a criar e vale AINV, porque existem 2 conjuntos.

2) AE && (C==F) —> AS: Pela AE, como (C==F), a matriz retornada é vazia. Vale a AS pois a matriz não é nula e a variável “LINHAS" e “COLUNAS" não são maior que 0.

3) AE && (C==T) + B —> AINV: Pela AE, como (C==T), o elemento entra no conjunto já criado e vale AINV.

4) AINV && (C==T) + B —> AINV: Para que AINV continue valendo, B tem que fazer com que o elemento em questão vá do conjunto a criar para já criado.

5) AINV && (C==F) —> AS: Com (C==F), a matriz já foi criada ou as variáveis “LINHAS” e "COLUNAS" não são maior que 0, valendo a AS.

6) Término: O conjunto a criar possui um número finito de elementos e a cada ciclo de repetição um deles passa para o conjunto já criado. A repetição só termina após um número finito de ciclos a serem dados.

- Preenche Tabuleiro

AE

JGO_tpCondRet PreencheTabuleiro(MAT_tppMatriz matriz, char* pBaralho, int* pTamanhoBaralho)

{

```
int contadorLinhas, contadorColunas, contadorAltura;
PIL_tppPilha pilha = NULL;
MAT_tpCondRet retMatriz;
PIL_tpCondRet retPilha;
```

REP

```
	for(contadorAltura = 0; contadorAltura < LINHAS / 2; contadorAltura++)
	{
		for(contadorLinhas = contadorAltura; contadorLinhas < LINHAS -
contadorAltura; contadorLinhas++)

		{
			for(contadorColunas = contadorAltura; contadorColunas < COLUNAS
- contadorAltura; contadorColunas++)

			{
				char caracter;
				retMatriz = MAT_IrParaCoordenada(matriz, contadorLinhas,
contadorColunas);

				if(retMatriz != MAT_CondRetOK){
					return JGO_CondRetErroMatriz;
				}
				retMatriz = MAT_ObterValorCorr(matriz, &pilha);
				if(retMatriz != MAT_CondRetOK){
					return JGO_CondRetErroMatriz;
				}

				caracter = SorteiaPeca(contadorLinhas, contadorColunas,
pBaralho, pTamanhoBaralho);

				retPilha = PIL_EmpilhaValor(pilha, caracter);
				if(retPilha != PIL_CondRetOK){
					return JGO_CondRetErroPilha;
				}

			}
		}
	}

	return JGO_CondRetOK;
} /* Fim função: Preenche Tabuleiro */
```

AS

Repetição :

AE -> existe um ponteiro para uma matriz, um ponteiro para um baralho e um ponteiro para o tamanho do baralho;

AS -> Caso a matriz a ser retornada por referência não seja nula, pMatriz recebe por referência uma matriz com pVetorPilhas preenchido em sua respectiva coordenada na matriz que representa um tabuleiro;

AINV -> existem 2 conjuntos:

- a preencher
- já preenchido

1) AE —> AINV: Pela AE, existe um ponteiro para uma matriz para um baralho e para o tamanho do baralho, porém a matriz ainda não foi preenchida. Seus elementos então estão no conjunto a preencher e vale AINV, porque existem 2 conjuntos.

2) AE && (C==F) —> AS: Pela AE, como (C==F), a matriz retornada é vazia. Vale a AS pois a matriz não é nula e a variável “LINHAS" e “COLUNAS" não são maior que 0.

3) AE && (C==T) + B —> AINV: Pela AE, como (C==T), o elemento entra no conjunto já preenchido e vale AINV.

4) AINV && (C==T) + B —> AINV: Para que AINV continue valendo, B tem que fazer com que o elemento em questão vá do conjunto a preencher para já preenchido.

5) AINV && (C==F) —> AS: Com (C==F), o baralho já foi inserido ou as variáveis “LINHAS” e "COLUNAS" não são maior que 0, valendo a AS.

6) Término: O conjunto a criar possui um número finito de elementos e a cada ciclo de repetição um deles passa para o conjunto já criado. A repetição só termina após um número finito de ciclos a serem dados.

- Imprime Tabuleiro

AE

```
JGO_tpCondRet ImprimeTabuleiro(MAT_tppMatriz matriz)
{
        int contadorLinhas, contadorColunas, contadorAltura, tamanho;
        char conteudo;
        MAT_tpCondRet retMatriz;
        PIL_tpCondRet retPilha;
        PIL_tppPilha pilha;

        system("cls");

        printf("PACIENCIA MAH JONG:\n\n");
        printf("|  |  ");
```

REP

```
        for(contadorColunas = 0; contadorColunas < COLUNAS; contadorColunas++)
        {
                printf("|%d |", contadorColunas);
        }
```

```
        printf("\n");
        printf("\n");
```

REP

```
        for(contadorLinhas = 0; contadorLinhas < LINHAS; contadorLinhas++)
        {
                printf("| %d|  ", contadorLinhas);

                for(contadorColunas = 0; contadorColunas < COLUNAS; contadorColunas+
+)
                {
                        retMatriz = MAT_IrParaCoordenada(matriz, contadorLinhas,
contadorColunas);

                        if(retMatriz != MAT_CondRetOK){
                                return JGO_CondRetErroMatriz;
                        }
                        retMatriz = MAT_ObterValorCorr(matriz, &pilha);
                        if(retMatriz != MAT_CondRetOK){
                                return JGO_CondRetErroMatriz;
                        }

                        retPilha = PIL_ObterValorTopo(pilha, &conteudo);
                        retPilha = PIL_ObterTamanho(pilha, &tamanho);
                        if(retPilha != PIL_CondRetOK){
                                printf("Got here -2");
                                return JGO_CondRetErroPilha;
                        }

                        printf("|%d%c|", tamanho, conteudo);
                }

                printf("\n");
        }

        return JGO_CondRetOK;
} /* Fim função: Imprime Tabuleiro */
```

AS

Repetição :

AE -> existe um ponteiro para uma matriz;

AS -> Caso a matriz a ser retornada por referência não seja nula, certa coordenada de pMatriz é impressa na tela do usuário;

AINV -> existem 2 conjuntos:

- a imprimir
- já impresso

1) AE —> AINV: Pela AE, existe um ponteiro para uma matriz, porém a matriz ainda não foi impressa. Seus elementos então estão no conjunto a imprimir e vale AINV, porque existem 2 conjuntos.

2) AE && (C==F) —> AS: Pela AE, como (C==F), a matriz a ser impressa é vazia. Vale a AS pois a matriz não é nula e a variável “LINHAS" e “COLUNAS" não são maior que 0.

3) AE && (C==T) + B —> AINV: Pela AE, como (C==T), o elemento entra no conjunto já impresso e vale AINV.

4) AINV && (C==T) + B —> AINV: Para que AINV continue valendo, B tem que fazer com que o elemento em questão vá do conjunto a imprimir para já impresso.

5) AINV && (C==F) —> AS: Com (C==F), a matriz já foi impressa ou as variáveis “LINHAS” e "COLUNAS" não são maior que 0, valendo a AS.

6) Término: O conjunto a criar possui um número finito de elementos e a cada ciclo de repetição um deles passa para o conjunto já criado. A repetição só termina após um número finito de ciclos a serem dados.

- Destroi Tabuleiro

AE

```
JGO_tpCondRet DestroiTabuleiro(MAT_tppMatriz *pMatriz)
{
        int contadorLinhas, contadorColunas;
        MAT_tpCondRet retMatriz;
        PIL_tpCondRet retPilha;
        PIL_tppPilha pilha;

        REP
        for(contadorLinhas = 0; contadorLinhas < LINHAS; contadorLinhas++)
        {
                for(contadorColunas = 0; contadorColunas < COLUNAS; contadorColunas+
+)
                {
                        MAT_IrParaCoordenada(*pMatriz, contadorLinhas, contadorColunas);
                        if(retMatriz != MAT_CondRetOK){
                                return JGO_CondRetErroMatriz;
                        }
                        MAT_ObterValorCorr(*pMatriz, &pilha);
                        if(retMatriz != MAT_CondRetOK){
                                return JGO_CondRetErroMatriz;
                        }
```

```
                    if(pilha != NULL)
                    {
                        retPilha = PIL_DestruirPilha(pilha);
                        if(retPilha != PIL_CondRetOK){
                            return JGO_CondRetErroPilha;
                        }
                    }
                }
            }

            MAT_DestruirMatriz(pMatriz);

            return JGO_CondRetOK;
} /* Fim função: Destrói Tabuleiro */
```

AS

Repetição :

AE -> existe um ponteiro para uma matriz;

AS -> Caso a matriz não seja nula e seus valores localizados nas linhas e colunas não sejam nulos, a matriz é destruida;

AINV -> existem 2 conjuntos:

- a destruir
- já destruido

1) AE —> AINV: Pela AE, existe um ponteiro para uma matriz, as pilhas contidas na matriz ainda não foram destruidas. Seus elementos então estão no conjunto a destruir e vale AINV, porque existem 2 conjuntos.

2) AE && (C==F) —> AS: Pela AE, como (C==F), a matriz a ser destruida é vazia. Vale a AS pois a matriz não é nula e a variável “LINHAS" e “COLUNAS" não são maior que 0.

3) AE && (C==T) + B —> AINV: Pela AE, como (C==T), o elemento entra no conjunto já destruido e vale AINV.

4) AINV && (C==T) + B —> AINV: Para que AINV continue valendo, B tem que fazer com que o elemento em questão vá do conjunto a destruir para já destruido.

5) AINV && (C==F) —> AS: Com (C==F), a matriz já foi destruida ou as variáveis “LINHAS” e "COLUNAS" não são maior que 0, valendo a AS.

6) Término: O conjunto a criar possui um número finito de elementos e a cada ciclo de repetição um deles passa para o conjunto já criado. A repetição só termina após um número finito de ciclos a serem dados.

- Sorteia Peça

AE

```
char SorteiaPeca(int linha, int coluna, char *pBaralho, int *pTamanhoBaralho)
{
	int indicePecaSorteada, contadorPecas, l;
	char pecaSorteada;

	srand(time(NULL) * (linha + coluna + 1));
	indicePecaSorteada = rand() % (*pTamanhoBaralho);

	pecaSorteada = pBaralho[indicePecaSorteada];
	pBaralho[indicePecaSorteada] = 'Z';
```

REP

```
	for(contadorPecas = 0; contadorPecas < PECAS - 1; contadorPecas++)
	{
		if(pBaralho[contadorPecas] == 'Z')
		{
			pBaralho[contadorPecas] = pBaralho[contadorPecas + 1];
			pBaralho[contadorPecas + 1] = 'Z';
		}
	}

	(*pTamanhoBaralho)--;

	return pecaSorteada;
} /* Fim função: Sorteia Peça */
```

AS

Repetição :

AE -> existe um ponteiro para o baralho, para o tamanho do baralho e também existe um valor representando o número de linhas e o número de colunas;

AS -> Caso a variável “PECAS" seja maior que 0, a peça sorteada é retornada;

AINV -> existem 2 conjuntos:

- a sortear
- já sorteado

1) AE —> AINV: Pela AE, existe um ponteiro para o baralho, para o tamanho do baralho e também existe um valor representando o número de linhas e o número de colunas,

a peça a ser sorteada ainda não foi sorteada. Seus elementos então estão no conjunto a sortear e vale AINV, porque existem 2 conjuntos.

2) AE && (C==F) —> AS: Pela AE, como (C==F), a variável “PEÇAS" é menor que 0. Vale a AS pois “PEÇAS" não é maior que 0.

3) AE && (C==T) + B —> AINV: Pela AE, como (C==T), o elemento entra no conjunto já sorteado e vale AINV.

4) AINV && (C==T) + B —> AINV: Para que AINV continue valendo, B tem que fazer com que o elemento em questão vá do conjunto a sortear para já sorteado.

5) AINV && (C==F) —> AS: Com (C==F), a peça já foi sorteada ou a variável "PEÇAS" não é maior que 0, valendo a AS.

6) Término: O conjunto a criar possui um número finito de elementos e a cada ciclo de repetição um deles passa para o conjunto já criado. A repetição só termina após um número finito de ciclos a serem dados.

- Preenche Baralho

AE

```
void PreencheBaralho(char *pBaralho)
{
        int contador;
        REP
        for(contador = 0; contador < PECAS / 4; contador++)
        {
                pBaralho[contador] = 'A';
        }

        for(contador = PECAS / 4; contador < PECAS/ 2 ; contador++)
        {
                pBaralho[contador] = 'B';
        }

        for(contador = 2 * PECAS / 4; contador < 3 * PECAS / 4; contador++)
        {
                pBaralho[contador] = 'C';
        }

        for(contador = 3 * PECAS / 4; contador < PECAS; contador++)
        {
                pBaralho[contador] = 'D';
        }
} /* Fim função: Preenche Baralho */
```

AS

Repetição:

AE -> existe um ponteiro para um baralho;

AS -> Caso a variável “PECAS" seja maior que 0, o ponteiro do baralho é preenchido;

AINV -> existem 2 conjuntos:
- a preencher
- já preenchido

1) AE —> AINV: Pela AE, existe um ponteiro para o baralho, o baralho se encontra vazio e ainda não foi preenchido. Seus elementos então estão no conjunto a preencher e vale AINV, porque existem 2 conjuntos.

2) AE && (C==F) —> AS: Pela AE, como (C==F), a variável “PEÇAS" é menor que 0. Vale a AS pois “PEÇAS" não é maior que 0.

3) AE && (C==T) + B —> AINV: Pela AE, como (C==T), o elemento entra no conjunto já preenchido e vale AINV.

4) AINV && (C==T) + B —> AINV: Para que AINV continue valendo, B tem que fazer com que o elemento em questão vá do conjunto a preencher para já preenchido.

5) AINV && (C==F) —> AS: Com (C==F), o baralho já foi preenchido ou a variável "PEÇAS" não é maior que 0, valendo a AS.

6) Término: O conjunto a criar possui um número finito de elementos e a cada ciclo de repetição um deles passa para o conjunto já criado. A repetição só termina após um número finito de ciclos a serem dados.

- Checa Retiravel (Não possui repetição)
- Checa Game Over

AE

```
JGO_tpCondRet ChecaGameOver (MAT_tppMatriz* pMatriz, int *retorno)
{
  //Pilha utilizada na análise.
  PIL_tppPilha pilha = NULL;

  //Valores de pilha
  char pecaPilha = NULL;

  //Contadores
  int contadorPecasA = 0;
  int contadorPecasB = 0;
```

```
int contadorPecasC = 0;
int contadorPecasD = 0;
int contadorLinhas;
int contadorColunas;

//Variáveis de retorno.
MAT_tpCondRet retornoMatriz;
PIL_tpCondRet retornoPilha;
JGO_tpCondRet retornoJogo;
int retornoChecaRetiravel = 0;

//Valor default de retorno é 0.
(*retorno) = 0;
```

REP

```
//Percorre toda a Matriz e conta quantas pecas retiráveis existem de cada tipo.
for (contadorLinhas = 0; contadorLinhas < LINHAS; contadorLinhas++)
{
  for (contadorColunas = 0; contadorColunas < COLUNAS; contadorColunas++)
  {
    retornoMatriz = MAT_IrParaCoordenada(*pMatriz, contadorLinhas, contadorColunas);

    if (retornoMatriz != MAT_CondRetOK)
    {
      return JGO_CondRetErroMatriz;
    }

    retornoMatriz = MAT_ObterValorCorr(*pMatriz, &pilha);

    if (retornoMatriz != MAT_CondRetOK)
    {
      return JGO_CondRetErroMatriz;
    }

    retornoPilha = PIL_ObterValorTopo(pilha, &pecaPilha);

    if (retornoPilha != PIL_CondRetOK && retornoPilha != PIL_CondRetPilhaVazia)
    {
                printf("AQUI! 1\n");
      return JGO_CondRetErroPilha;
    }

    retornoJogo = ChecaRetiravel(pMatriz, contadorLinhas, contadorColunas,
&retornoChecaRetiravel);
```

```
      if (retornoJogo != JGO_CondRetOK)
      {
        return retornoJogo;
      }

      if (retornoChecaRetiravel == 1)
      {
        if (pecaPilha == 'A')
        {
          contadorPecasA++;
        }

        else if (pecaPilha == 'B')
        {
          contadorPecasB++;
        }

        else if (pecaPilha == 'C')
        {
          contadorPecasC++;
        }

        else if (pecaPilha == 'D')
        {
          contadorPecasD++;
        }
      }
    }

    //Verifica se ainda existe alguma jogada possível.
    if (contadorPecasA < 2 && contadorPecasB < 2 && contadorPecasC < 2 &&
contadorPecasD < 2)

    {
                    if (contadorPecasA == 0 && contadorPecasB == 0 && contadorPecasC == 0
&& contadorPecasD == 0){

                            *retorno = 2; /*Caso de vitoria*/
                    }
                    return JGO_CondRetOK;
    }

    *retorno = 1;
    return JGO_CondRetOK;
  }


} /* Fim função: Checa GameOver */
```

AS

Repetição :

AE -> existe um ponteiro para uma matriz e um ponteiro para o retorno;

AS -> Caso “LINHAS” e “COLUNAS" sejam maior que 0, não existam peças combináveis no tabuleiro ou não existam peças no tabuleiro, o programa é encerrado e é retornado o estado do jogo (vitória, não determinado ,derrota);

AINV -> existem 2 conjuntos:

- a contar
- já contado

1) AE —> AINV: Pela AE, existe um ponteiro para uma matriz e para o retorno, porém a matriz ainda não foi percorrida. Seus elementos então estão no conjunto a contar e vale AINV, porque existem 2 conjuntos.

2) AE && (C==F) —> AS: Pela AE, como (C==F), a matriz retornada é vazia. Vale a AS pois a matriz não é nula e a variável “LINHAS" e “COLUNAS" não são maior que 0.

3) AE && (C==T) + B —> AINV: Pela AE, como (C==T), o elemento entra no conjunto já contado e vale AINV.

4) AINV && (C==T) + B —> AINV: Para que AINV continue valendo, B tem que fazer com que o elemento em questão vá do conjunto a contado para já contado.

5) AINV && (C==F) —> AS: Com (C==F), a matriz já foi percorrida e suas linhas e colunas contadas ou as variáveis “LINHAS” e "COLUNAS" não são maior que 0, valendo a AS.

6) Término: O conjunto a criar possui um número finito de elementos e a cada ciclo de repetição um deles passa para o conjunto já criado. A repetição só termina após um número finito de ciclos a serem dados.