INSTITUTO POLITÉCNICO DO CÁVADO E AVE

TRABALHO PRÁTICO 1

RELATÓRIO

JORGE MIGUEL AREZES NORO | 15705

ENGENHARIA E DESENVOLVIMENTO DE JOGOS DIGITAIS

PARADIGMAS DE PROGRAMAÇÃO II

DOCENTE: LUÍS GONZAGA MARTINS FERREIRA

2º SEMESTRE

1. INTRODUÇÃO

Este documento serve de relatório ao Trabalho Prático 1 da unidade curricular de Paradigmas de Programação II. É acompanhado de um projeto de Visual Studio (*pp2-tp1.sln*) dividido em 2 projetos: *Part1 – Static Data Structures* e *Part2 – Dynamic Data Structures*. Não existem dependências entre os dois projectos.

O projecto está disponível no GitHub, no seguinte URL:
https://github.com/mrMav/pp2-tp1

Na realização deste trabalho, houve como ideal apelar à parametrização do programa. Existem várias variáveis que podem ser ajustadas conforme as preferências do utilizador. Ficam abaixo ditas variáveis com a descrição do seu papel no programa:

```
/*
Number of columns in the Matrix Card
*/
#define COLUMNS_NUMBER 8

/*
Number of rows in the Matrix Card
*/
#define ROWS_NUMBER 8

/*
Number of digits in a Node sequence
*/
#define SEQUENCE_SIZE 3

/*
Number of times the user will be asked for Matrix Card input
*/
#define SECURITY_LEVEL 3
```

Code Snippet 1- Variáveis para parametrização do programa

2. Estruturas de Dados Estáticas

2.1 Alocação de memória

Para o desenvolvimento da parte 1 do trabalho prático, a estrutura de dados estática capaz de armazenar todos os dados do cartão matriz será um *array* de inteiros. Uma possível solução, a que foi implementada neste trabalho, passou por separar as células do cartão matriz em *structs* chamados de *Nodes.* Cada *Node* terá um *array* de inteiros, em que será armazenada a sequência de números.

Para construir a matriz, usamos então outro *struct* chamado de *Row.* Este *struct* armazena o número de *Nodes* igual ao número de colunas definidas na matriz (ver *Types.h)*. O *struct Matrix* tem como membro um *array* do tipo *Row*, em que armazena o número de linhas definidas na matriz. É possível ver na imagem abaixo a relação dos *structs* entre si:

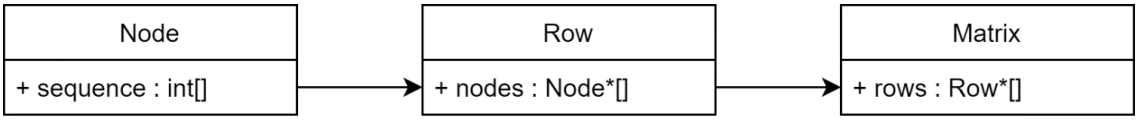

Figura 1-Relação das estruturas de dados.

A solução pode parecer complexa demais para a resolução deste problema simples, mas desta forma deixamos espaço para o crescimento da aplicação, como por exemplo, adicionar informação sobre determinado *Node* ou sobre determinada *Row*. Uma solução mais simples e básica podia ser implementar simplesmente um *array* tridimensional:

```
int matrix[ROWS_NUMBER][COLUMNS_NUMBER][SEQUENCE_SIZE];
```

Code Snippet 2- Possível implementação da matriz, usando um simples array tridimensional

As funções de alocação de memória destes objetos são encontradas no ficheiro *ObjectCreation.c*. Fica abaixo um exemplo (alocação em memória de uma matriz):

```
typedef struct {

	Row* rows[ROWS_NUMBER];

} Matrix;

Matrix* createMatrix() {

	Matrix* m = (Matrix*)malloc(sizeof(Matrix));

	for (int i = 0; i < ROWS_NUMBER; i++) {

		m->rows[i] = createRow();

	}

	return m;

};
```

Code Snippet 3- Alocação em memória de uma matriz.

O código chama a função *createRow()*, que por sua vez chama a função *createNode()*. Esta, vai popular o *Node* com uma sequência aleatória de números. Podemos ver essa função abaixo:

```
typedef struct {

	int sequence[SEQUENCE_SIZE];

} Node;

Node* createNode() {

	Node* n = (Node*)malloc(sizeof(Node));

	for (int i = 0; i < SEQUENCE_SIZE; i++) {

		n->sequence[i] = randomInt(0, 10);
	}

	return n;
};
```

Code Snippet 4- Alocação em memória de um Node

2.2 ALGORITMOS

No que diz respeito a algoritmos, esta primeira parte do trabalho é relativamente simples. Podemos analisar como exemplo, a função em que pedimos ao utilizador que valide uma operação. Fica abaixo o código em excerto (algumas partes foram retiradas para auxiliar a leitura neste relatório, ver ficheiro *UtilityFunctions.c* para analisar o código completo):

```
int validateOperation(Matrix* matrix) {
	// error handling
	(...)
	int isValid = 0;
	for (int i = 0; i < SECURITY_LEVEL; i++) {

		int row, col, digit, ans;
		char rowChar;

		row = randomInt(0, ROWS_NUMBER - 1);
		col = randomInt(0, COLUMNS_NUMBER - 1);
		digit = randomInt(0, SEQUENCE_SIZE - 1);
		rowChar = 65 + row;

		// output to inform user
		(...)
		scanf("%i", &ans);
		getchar();
		isValid = validatePosition(ans, row, col, digit, matrix);

		// break after one failed attempt
		if (isValid == 0)
			break;
	}
	return isValid;
};
```

Code Snippet 5- Função para validar uma operação

A função gera números aleatórios para a linha, coluna e posição do dígito pretendido. De seguida, mostra ao utilizador qual a célula e posição do dígito que se pretende e obtém a resposta. É chamada uma função que valida essa resposta. Vejamos abaixo:

```
int validatePosition(int ans, int row, int col, int digit, Matrix* matrix) {

	// error handling
	(...)

	// get the matrix digit in the given position
	int matrixDigit = matrix->rows[row]->nodes[col]->sequence[digit];

	if (matrixDigit == ans) {

		return 1;

	}
	else {

		return 0;

	}

};
```

Code Snippet 6- Função para validar escolha do utilizador

3. Estruturas de Dados Dinâmicas

3.1 Alocação de memória

Para a parte 2 do trabalho, optar-se-á por construir uma lista ligada. Esta lista irá ser composta por *Nodes*, em que cada *Node* representa uma célula do cartão matriz e tem como dados um *array* de inteiros (os números correspondentes a essa célula). A implementação do cartão matriz como lista ligada força o acesso às células de uma forma mais cuidadosa do que com uma estrutura de dados estática. A figura abaixo mostra a estrutura de um *Node* a implementar no código:

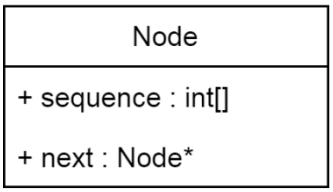

Figura 2- Estrutura de um Node da lista ligada

Para criar uma matriz, usa-se a função *buildMatrix()*, a qual retorna o *Node* inicial da matriz. Como não se está a usar um *array,* a função é simplesmente um ciclo que inicializa o número necessário de *Nodes.* A diferenciação de colunas e linhas da matriz, só será tida em conta nas funções de pesquisa e manipulação da matriz. Abaixo pode-se observar e analisar a implementação da função (ver *Matrix.c*) e a definição do tipo *Node* (ver *Types.c*):

```
typedef struct Node {

	int sequence[SEQUENCE_SIZE];
	struct Node* next;

} Node;

Node* buildMatrix() {

	Node* head = createNode();
	Node* tail = NULL;

	for (int i = 0; i < COLUMNS_NUMBER * ROWS_NUMBER - 1; i++) {

		tail = addNode(head, NULL);

	}

	return head;

};
```

Code Snippet 7-Alocação de memória de uma matriz, usando uma lista ligada.

A função *addNode(Node* head, Node* toAdd)* (ver *CreationFunctions.c*), é a responsável por alocar a memória de cada *Node* individualmente e contém dois parâmetros do tipo *Node**. *head* é o apontador para o *Node* inicial e *toAdd* é o apontador para o *Node* a ser inserido, que no caso de ser igual a *NULL*, é criado um novo *Node* com a sequência iniciada aleatoriamente. Esta função *addNode* é também utilizada quando o utilizador carrega os dados da matriz através de um ficheiro previamente guardado. Abaixo podemos analisar a função:

```
Node* addNode(Node* head, Node* toAdd) {

	Node* n = head;

	while (n->next != NULL) {

		n = n->next;
	}

	if (toAdd == NULL) {

		n->next = createNode();
	}
	else {

		n->next = toAdd;
	}

	return n->next;
}
```

Code Snippet 8- Inserção de um Node no final da lista

Para criar uma novo *Node* utiliza-se uma função *createNode()* bastante similar à usada no capitulo 1, Estruturas de Dados Estáticas. Pode-se ver essa função no ficheiro *CreationFunctions.c*.

3.2 ALGORITMOS

Nesta secção vamos analisar alguns algoritmos interessantes necessários para a manipulação de listas ligadas. Em primeiro, observemos como foi implementada uma função que realiza uma acção em cada *Node*. Embora não fosse realmente necessário, a implementação desta função foi um desafio pessoal. O objectivo era ter uma função similar ao que outras linguagens possuem, um *forEach()*.

É então implementada a função *forEachNode(Node* head, void(*callback)(Node* n).* A quem não esteja habituado, esta assinatura pode ser confusa. O primeiro parâmetro não é nada mais que o primeiro *Node* da lista. O segundo parâmetro é o apontador para uma outra função do tipo *void functionName(Node*)*. Ao saber que em C, todos os nomes das funções são apontadores para essas mesmas funções, o segundo parâmetro torna-se um pouco mais claro, pois sabemos que um asterisco antecedente de um apontador, desreferencia e retorna o valor apontado. A explicação pode suscitar dúvidas, portanto vamos analisar a implementação da função:

```
void forEachNode(Node* head, void(*callback)(Node* n)) {

	Node* n = head;

	int exit = 0;

	while (exit == 0) {

		(*callback)(n);

		n = n->next;

		if (n == NULL) {

			exit = 1;

		}
	}
};
```

Code Snippet 9- Implementação da função forEachNode

No corpo da função podemos observar um algoritmo que permite percorrer a lista ligada desde o *Node head* até ao final da lista (o final da lista é definido pelo *Node* em que a propriedade *next* devolve *NULL*). Dentro do ciclo, é invocada a função que é passada como argumento e é dado como argumento o *Node* que está no momento armazenado na variável *n*. De seguida, a variável *n* obtém o valor do próximo *Node* e assim sucessivamente.

Esta abordagem permite um acesso rápido a todos os *Nodes,* como podemos ver num exemplo de utilização em que pretendemos imprimir todos os *Nodes* no ecrã. Em primeiro lugar, definimos uma função que imprima um *Node*, sem nunca esquecer qual a assinatura modelo que temos de seguir (retornar *void* e ter um parâmetro *Node**):

```
void printNode(Node* n) {

	// error handling
	(…)

	for (int i = 0; i < SEQUENCE_SIZE; i++) {

		printf("%i", n->sequence[i]);

	}

};
```

Code Snippet 10-Função capaz de imprimir um Node no ecrã

Assim, podemos invocar a função *forEachNode* da seguinte forma:

```
forEachNode(head, printNode);
```

Code Snippet 11- Invocação da função forEachNode com callback

O próximo algoritmo que será analisado, é o implementado na função *validatePosition*. O algoritmo percorre a lista ligada a partir do *Node* passado como argumento à função (será sempre, em teoria, o inicio da matriz). Para determinar em que linha e coluna nos encontramos usamos um contador que armazena em que número de *Node* estamos. Esse contador determina a coluna ao realizar a operação aritmética módulo entre si e o número de colunas na matriz e guarda o esse resultado numa variável *cCol (current column)*. De igual modo determina a linha da matriz em que nos encontramos, mas em vez de usar a operação módulo, usa a divisão (entre inteiros).[1] Vejamos a implementação da função:

```
int validatePosition(int ans, int row, int col, int pos, Node* head) {
	// error handling
	(…)
	// current row being searched
	int cRow = 0;
	// current col being searched
	int cCol = 0;
	// count of searched nodes
	int i = 0;
	// loop exit flag
	int exit = 0;
	// current node
	Node* n = head;

	while (exit != 1) {
		// get the col and row arithmetically
		cCol = i % COLUMNS_NUMBER;
		cRow = i / COLUMNS_NUMBER;

		if (cRow == row && cCol == col) {
			// check pos in node and return result (0 or 1)
			if (n->sequence[pos] == ans) {
				return 1;
			}
			else {
				return 0;
			}
		}
		else {
			n = n->next;
			i++;
		}
		// break case.
		// should never be reached
		// (exceptions handled in error handling above)
		if (n == NULL) {
			exit = 1;
		}
	}
	return -1;
};
```

Code Snippet 12- Obter linha e coluna aritmeticamente, validatePosition em listas ligadas

[1] Esta forma de obter o número de linha e coluna é muito utilizada em desenvolvimento de jogos *tilebased*, em que a informação é guardada em *arrays* unidimensionais (vetores).

O último algoritmo que vamos rever é o implementado na função *freeMatrix(Node** head)*. Esta função é a responsável de devolver ao sistema operativo a memoria requisitada através das funções *malloc()*. A função tem como parâmetro um apontador para um apontador. Isto é necessário pois após a utilização da função de C *free(void* ptr)*, iremos alterar o valor do apontador para *NULL*. Vejamos:

```
void freeMatrix(Node** head) {

	if (*head == NULL) return;

	Node* current = *head;
	Node* next = NULL;

	int exit = 0;

	while (exit == 0) {

		next = current->next;

		free(current);

		current = next;

		if (current == NULL) {

			exit = 1;
		}
	}

	*head = NULL;
};
```

Code Snippet 13- Função para devolver memória da matriz ao sistema operativo.

Após verificarmos se o *Node* não é já igual a *NULL*, percorremos a árvore da mesma forma que previamente. Guardamos num *buffer* o apontador do *Node* seguinte e chamamos a função *free()* no actual *Node*. Após libertar a memória contida nesse apontador, atribuímos o seguinte *Node* ao actual e assim sucessivamente. No final, definimos o apontador inicial como *NULL*.

Após a análise destes algoritmos e para não alongar este relatório, deixam-se de parte os algoritmos para guardar a lista em ficheiro e recuperar a mesma. O algoritmo será o mesmo, pois o *core* da implementação de listas ligadas é o algoritmo para percorrer essa lista.