

Relatório do Trabalho Prático

Estruturas de Dados Avançadas

Docente: Luís Ferreira

Aluna:
Adriana Gomes a23151

Índice

1.	Introdução e Objetivos	5
2.	Estruturas de Dados e Funções desenvolvidas para gestão de <i>Jobs</i>	6
2.1.	<i>Jobs</i>	6
2.1.1.	Estruturas de registo para <i>jobs</i>	6
2.1.2.	Criar <i>Jobs</i>	6
2.1.3.	Inserir nodo na árvore binária	7
2.1.4.	Existe <i>Job</i>	7
2.1.5.	Ler de um ficheiro de texto	8
2.1.6.	Mostrar árvore	9
2.1.7.	Remover <i>Job</i> da árvore	9
2.1.8.	Guardar árvore em ficheiro	10
2.2.	Operações	12
2.2.1.	Estrutura de registo para operações	12
2.2.2.	Criar operações	12
2.2.3.	Inserir operações	12
2.2.4.	Remover operação	13
2.2.5.	Mostrar lista	14
2.3.	Máquinas	16
2.3.1.	Estrutura de registo para máquinas	16
2.3.2.	Criar máquinas	16
2.3.3.	Inserir máquinas nas operações	16
2.4.	Determinação do tempo necessário para completar um <i>job</i>	18
2.4.1.	Duração mínima e máxima do <i>job</i>	18
2.4.2.	Duração média do <i>job</i>	18
3.	Proposta de escalonamento	20
3.1.1.	Estruturas de registo	20
3.1.2.	Iniciar Array	20
3.1.3.	Preencher Célula	21
3.1.4.	Ocupar o Plano	22
4.	Testes ao projeto desenvolvido	23
4.1.1.	Lista de operações	23
4.1.2.	Inserir as máquinas nas operações	24
4.1.3.	Duração das operações	24
4.1.4.	Criação da árvore binária	25
5.	Conclusão	27
6.	Bibliografia	28

Índice de figuras

Figura 1 - Estrutura "Job" que contém informação sobre cada <i>job</i>	6
Figura 2 - Estrutura "JobInfoFile", armazena informação para ser guardada num ficheiro.	6
Figura 3 - Função desenvolvida para criar novos jobs.	6
Figura 4 - Função desenvolvida para inserir os jobs numa árvore binária.	7
Figura 5 - Função que verifica se um dado job já existe na árvore.	7
Figura 6 - Função que lê de um ficheiro e constrói a ABP.	8
Figura 7 - Função que imprime na consola os códigos dos <i>jobs</i>	9
Figura 8 - Função que devolve o nodo mais à esquerda do lado direito da árvore.	10
Figura 9 - Função para remover um <i>job</i> da árvore.	10
Figura 10 - Função desenvolvida para guardar os dados da AB num ficheiro.	11
Figura 11 - Função que permite percorrer a árvore, guardar toda a informação de cada nodo numa estrutura e escrever num ficheiro.	11
Figura 12 - Estrutura "Operacao" que contém informação sobre cada operação.	12
Figura 13 - Função desenvolvida para criar novas operações.	12
Figura 14 - Função para inserir as operações na lista.	13
Figura 15 - Função para verificar se a operação a inserir ainda não existe na lista.	13
Figura 16 - Função para remover operações da lista.	14
Figura 17 - Função desenvolvida para apresentar na consola as operações da lista.	14
Figura 19 - Função para mostrar as máquinas que realizam as operações.	15
Figura 20 - Estrutura "Maquina" que contém informação sobre cada operação.	16
Figura 21 - Função para criar máquinas.	16
Figura 22 - Função que permite adicionar uma lista de máquinas nas operações.	17
Figura 23 - Função para inserir uma máquina na lista de máquinas.	17
Figura 24 - Função desenvolvida para determinar o tempo mínimo para completar um <i>job</i>	18
Figura 25 - Função que permite determinar a duração média das operações e do <i>job</i>	19
Figura 26 - Estruturas desenvolvidas para o escalonamento.	20
Figura 27 - Funções para inicializar os arrays utilizados nesta proposta de escalonamento.	20
Figura 28 - Função que permite preencher os campos de "Celula" com a informação de cada nodo da árvore.	21
Figura 29 - Função "preencherArrayAuxiliar".	21
Figura 30 - Função desenvolvida para representar o plano nas condições definidas.	22
Figura 31 - Funções utilizadas na função <i>main</i> para criar a lista de operações.	23
Figura 32 - Resultado apresentado pela função "mostrarListaOperacao".	23
Figura 33 - Aplicação da função remover para a operação e listagem das operações.	23
Figura 34 - Conjunto de funções utilizadas para criar as máquinas e inserir nas operações.	24
Figura 35 - Resultado apresentado pela função "mostrarListaOperacaoMaquinas".	24
Figura 36 - Resultado das funções desenvolvidas para determinar o tempo necessário para completar as operações e o <i>job</i>	25
Figura 37 - Estrutura do ficheiro de texto utilizado.	25
Figura 38 - Funções utilizadas no <i>main</i> para apresentar a árvore.	26
Figura 39 - Resultado obtido pela função "mostrarArvore".	26
Figura 40 - Resultado da árvore após usar a função "removerJob" para remover o job 2.	26

1. Introdução e Objetivos

Este relatório foi desenvolvido no âmbito da unidade curricular de Estruturas de Dados Avançadas e tem como objetivo apresentar uma proposta de solução para o problema de escalonamento denominado *Flexible Job Shop Problem* (FJSSP) utilizando linguagem C. Num FJSSP existe um conjunto de *jobs* e cada *job* pode envolver várias operações. Por sua vez, as operações são processadas por um conjunto de máquinas.

O desenvolvimento deste projeto dividiu-se em duas fases. Numa primeira fase, foram aplicados os conhecimentos adquiridos sobre apontadores e listas ligadas, sendo considerado apenas um *job*. Na segunda fase, aplicaram-se mais estruturas, particularmente, árvore binária de procura (ABP) na gestão dos vários *jobs*. Numa ABP, cada nodo pode ter até 2 filhos e a raiz é menor do que todos os valores à direita e maior do que os nodos da esquerda. O elemento mais à esquerda é o menor da árvore e o mais à direita é o maior. Deste modo, comparativamente a uma lista ligada, a procura é mais rápida, dado que alguns nodos vão sendo excluídos.

2. Estruturas de Dados e Funções desenvolvidas para gestão de *Jobs*

2.1. *Jobs*

2.1.1. Estruturas de registo para *jobs*

Para a realização deste projeto foi definida uma estrutura de registo (figura 1) que armazena os dados dos *jobs*: código do job, um apontador para uma estrutura do tipo “Operacao” e dois apontadores para *jobs*, necessários para construir uma estrutura do tipo árvore binária.

```
typedef struct Job {  
    int codigoJob;  
    struct Operacao* listaOperacoes;  
    struct Job* left;  
    struct Job* right;  
} Job;
```

Figura 1 - Estrutura “Job” que contém informação sobre cada *job*.

Foi também definida uma estrutura designada “JobInfoFile” para armazenar toda a informação de um *job*, isto é, o seu código, as operações envolvidas no *job* e as máquinas que podem realizar cada operação. Esta estrutura é necessária para guardar a informação num ficheiro.

```
typedef struct JobInfoFile {  
    int codigoJob;  
    Operacao operacaoFile;  
    Maquina maquinaFile;  
} JobInfoFile;
```

Figura 2 - Estrutura “JobInfoFile”, armazena informação para ser guardada num ficheiro.

2.1.2. Criar *Jobs*

Para criar os *jobs* foi desenvolvida a função “criarNodoJob”, apresentado na figura 3. Esta função recebe como parâmetro o código do *job*. É utilizada a função *malloc* para alocar a memória necessária para um *job* e os campos da estrutura Job (figura 1) são definidos. Nesta fase, os apontadores não recebem nenhum endereço, ficando a NULL.

```
Job* criarNodoJob(int codigoJob) {  
  
    Job* novoJob = (Job*)malloc(sizeof(Job));  
    if (novoJob == NULL) {  
        return NULL;  
    }  
    novoJob->codigoJob = codigoJob;  
    novoJob->listaOperacoes = NULL;  
    novoJob->left = NULL;  
    novoJob->right = NULL;  
  
    return novoJob;  
}
```

Figura 3 - Função desenvolvida para criar novos jobs.

2.1.3. Inserir nodo na árvore binária

A função “inserirNodoJobArvore” (figura 4) permite inserir os *jobs* que foram criados a partir da função anterior numa árvore binária. Esta função começa por verificar se a árvore já contém algum elemento. Caso esteja vazia (NULL), o novo *job* passa a ser a raiz da árvore (*root*). Se a árvore incluir nodos, então, é verificado se o código do *job* a inserir é maior ou menor do que o da raiz. Esta verificação é feita recursivamente (*root* toma valor de cada nodo existente). Se o novo código for menor do que o da raiz, é inserido à esquerda, caso contrário é inserido à direita.

```
Job* inserirNodoJobArvore(Job* root, Job* novoJob) {  
    if (root == NULL) {  
        root = novoJob;  
    }  
    else {  
        if (root->codigoJob > novoJob->codigoJob) {  
            root->left = inserirNodoJobArvore(root->left, novoJob);  
        }  
        else  
            if (root->codigoJob < novoJob->codigoJob)  
                root->right = inserirNodoJobArvore(root->right, novoJob);  
    }  
    return root;  
}
```

Figura 4 - Função desenvolvida para inserir os jobs numa árvore binária.

2.1.4. Existe Job

A função “existeJob” foi desenvolvida para verificar se um dado *job* existe na árvore. Esta função recebe como parâmetros a raiz da árvore e o código do *job*, tal como se verifica na figura 5. De seguida, percorre-se a árvore verificando se o código dado é maior ou menor do que o código dos nodos. Esta verificação é feita de forma recursiva e caso o *job* exista, devolve a raiz, caso contrário devolve NULL.

```
Job* existeJob(Job* root, int codigoJob) {  
    if (root == NULL)  
        return NULL;  
    if (root->codigoJob == codigoJob)  
        return root;  
  
    if (root->codigoJob > codigoJob) {  
        return(existeJob(root->left, codigoJob));  
    }  
    else  
        if (root->codigoJob < codigoJob) {  
            return(existeJob(root->right, codigoJob));  
        }  
}
```

Figura 5 - Função que verifica se um dado job já existe na árvore.

2.1.5. Ler de um ficheiro de texto

Como neste trabalho é pedido para ler a informação a partir de um ficheiro de texto, foi desenvolvida a função “lerFicheirotexto” (figura 5). Esta função permite ler dados de um ficheiro, neste caso, um ficheiro designado “dadosIniciais.txt” e construir a árvore com toda a informação de *jobs*, operações e máquinas. O ficheiro contém o código do *job*, o código das operações, o ID das máquinas e o tempo para cada máquina.

É utilizada a função “fopen” para abrir o ficheiro e a função “fscanf” para guardar os dados nas respetivas variáveis. De seguida é chamada a função “existeJob” para verificar se o código já existe na árvore. Caso o job não exista o é criado e inserido na árvore. De seguida é criada uma variável do tipo “Operacao” que guarda o resultado da função “existeOperacaoPtr”. Esta função verifica se a operação já existe na lista de operações de um determinado *job* e devolve NULL se não existir. Neste caso, é criada a operação e inserida na lista. De seguida o valor de *auxOper* é atualizado e verifica-se se na lista de máquinas dessa operação existe a máquina que foi lida do ficheiro. Caso a máquina não exista, são chamadas as funções “criarMaquina” e “inserirMaquina” para criar e inserir a nova máquina na lista. Este processo está dentro de um ciclo *while* para ser realizado para todas as linhas do ficheiro.

```
Job* lerFicheiroTexto() {
    int job, operacao, tempo;
    char maquina[N];

    Job* novoJob = NULL;
    Job* arvore = NULL;

    FILE* fp;
    fp = fopen("dadosIniciais.txt", "rt");

    while (!feof(fp)) {

        fscanf(fp, "%d; %d; %d; %s;\n", &job, &operacao, &tempo, maquina);

        if (existeJob(arvore, job) == NULL)
        {
            novoJob = criarNodoJob(job);
            arvore = inserirNodoJobArvore(arvore, novoJob);
        }

        Operacao* auxOper = existeOperacaoPtr(novoJob->listaOperacoes, operacao);

        if (auxOper == NULL) {
            Operacao* novaOper = criarOperacao(operacao);
            novoJob->listaOperacoes = inserirOperacao(novoJob->listaOperacoes, novaOper);
        }

        if (auxOper == NULL)
            auxOper = existeOperacaoPtr(novoJob->listaOperacoes, operacao);

        if (existeMaquina(auxOper->listaMaquinas, maquina) == false) {

            Maquina* novaMaquina = criarMaquina(maquina, tempo);
            auxOper->listaMaquinas = inserirMaquina(auxOper->listaMaquinas, novaMaquina);
        }

    }

    fclose(fp);
    return arvore;
}
```

Figura 6 - Função que lê de um ficheiro e constrói a ABP.

2.1.6. Mostrar árvore

Para mostrar a árvore na consola foi desenvolvida a função apresentada na figura 7. Esta função mostra os códigos dos jobs inseridos por ordem (*in-order*). A função começa por analisar a raiz e, caso a raiz tenha elementos menores, a função é novamente invocada percorrendo o lado esquerdo. Quando chega ao final do lado esquerdo (NULL) é então escrito o valor mais baixo e assim sucessivamente.

```
void mostrarArvore(Job* root) {  
    if (root == NULL) return;  
    mostrarArvore(root->left);  
    printf("Root: %d \n", root->codigoJob);  
    mostrarArvore(root->right);  
}
```

Figura 7 - Função que imprime na consola os códigos dos jobs.

2.1.7. Remover Job da árvore

De forma a ser possível alterar a árvore, removendo *jobs*, foi desenvolvida a função “removerJob” (figura 8). Esta função recebe como parâmetros a árvore (raiz) e o código para identificar o *job* a eliminar. Inicialmente a árvore é percorrida até encontrar o *job*. Se esse nodo não tiver filhos é removido, caso tenha apenas um filho, é criada uma variável temporária guarda o nodo e este é substituído pelo nodo filho. De seguida a lista de operações do *job* é removida e o *job* é então removido através da função *free*. Caso o nodo a eliminar tenha dois filhos, este pode ser substituído pelo nodo mais à direita do seu lado esquerdo ou pelo nodo mais à esquerda do seu lado direito. Neste caso é criada uma variável auxiliar que guarda o apontador devolvido pela função “procurarMin” (figura 9), que procura o valor mais à esquerda (menor) do lado direito do nodo a remover. Por último, a função é chamada recursivamente para remover o nodo que substituiu o removido, ou seja, o que foi devolvido pela função “procurarMin”.

```

Job* removerJob(Job* root, int codigoJob) {
    if (root == NULL) return NULL;

    if (codigoJob < root->codigoJob)
        root->left = removerJob(root->left, codigoJob);
    else
        if (codigoJob > root->codigoJob)
            root->right = removerJob(root->right, codigoJob);

        else {
            if (root->left == NULL) {
                Job* temp = root;
                root = root->right;
                removerListaOperacoes(temp->listaOperacoes);
                free(temp);
                return root;
            }
            else if (root->right == NULL) {
                Job* temp = root;
                root = root->left;
                removerListaOperacoes(temp->listaOperacoes);
                free(temp);
                return root;
            }

            Job* temp = procurarMin(root->right);
            root = temp;
            root->right = removerJob(root->right, temp->codigoJob);
        }
    return root;
}

```

Figura 9 - Função para remover um *job* da árvore.

```

Job* procurarMin(Job* root) {
    if (root->left == NULL) return root;
    else
        return(procurarMin(root->left));
}

```

Figura 8 - Função que devolve o nodo mais à esquerda do lado direito da árvore.

2.1.8. Guardar árvore em ficheiro

De forma a guardar num ficheiro o resultado da árvore binária, com toda a informação sobre cada *job* foi criada a função “guardarArvoreFicheiro” (figura 10). Esta função recebe como parâmetros o apontador para o início da árvore e o nome do ficheiro. Utiliza-se a função *fopen* com o parâmetro “wb” para criar um ficheiro binário em modo escrita. De seguida é invocada a função “guardarArvore” e utiliza-se a função *fclose* para fechar o ficheiro.

A função “guardarArvore”, apresentada na figura 11, recebe como parâmetros a raiz da árvore e apontador para o ficheiro. Nesta função é definida uma variável auxiliar do tipo da estrutura que se pretende guardar no ficheiro. De seguida, os campos dessa estrutura são “preenchidos”, sendo que para cada máquina de cada operação é criada uma estrutura diferente. Inicialmente é atribuído o valor do código do *job*, posteriormente é percorrida a lista

de operações através de um ciclo *while* e para cada operação é percorrida a lista de máquinas. Utiliza-se a função *fwrite* para escrever no ficheiro (fp) os campos da estrutura. De seguida a função é chamada novamente para percorrer a árvore e escrever no ficheiro em *pre-order*.

```
bool guardarArvoreFicheiro(Job* root, char* fileName) {  
  
    FILE* fp;  
  
    if ((fp = fopen(fileName, "wb")) == NULL) return false;  
    guardarArvore(root, fp);  
  
    fclose(fp);  
    return true;  
}
```

Figura 10 - Função desenvolvida para guardar os dados da AB num ficheiro.

```
void guardarArvore(Job* root, FILE *fp) {  
  
    JobInfoFile auxFile;  
  
    if (root == NULL) return;  
  
    //Grava a root corrente  
    auxFile.codigoJob = root->codigoJob;  
    Operacao* auxOper = root->listaOperacoes;  
    while (auxOper) {  
        auxFile.operacaoFile.codigo= auxOper->codigo;  
  
        Maquina* auxMaq = auxOper->listaMaquinas;  
        while (auxMaq) {  
            auxFile.maquinaFile.tempo = auxMaq->tempo;  
            strcpy(auxFile.maquinaFile.id, auxMaq->id);  
            fwrite(&auxFile, sizeof(auxFile), 1, fp);  
            auxMaq = auxMaq->next;  
        }  
        auxOper = auxOper->next;  
    }  
    guardarArvore(root->left, fp);  
    guardarArvore(root->right, fp);  
}
```

Figura 11 - Função que permite percorrer a árvore, guardar toda a informação de cada nodo numa estrutura e escrever num ficheiro.

2.2. Operações

2.2.1. Estrutura de registo para operações

Para a realização deste projeto foi definida uma estrutura de registo (figura 12) que armazena os dados das operações: código da operação, um apontador para uma estrutura do tipo “Maquina” e outro apontador para a próxima operação.

```
typedef struct Operacao {  
    int codigo;  
    struct Maquina* ListaMaquinas;  
    struct Operacao* next;  
} Operacao;
```

Figura 12 - Estrutura “Operacao” que contém informação sobre cada operação.

2.2.2. Criar operações

As operações são criadas a partir da função “criarOperacao”, apresentada na figura 13. Esta função recebe como parâmetro o código da operação que se pretende adicionar à lista. Nesta função é utilizada a função *malloc* para alocar a memória necessária para uma operação. De seguida são inseridos os dados da operação. Nesta fase, os apontadores não recebem nenhum endereço, ficando a NULL.

```
Operacao* criarOperacao(int codigo) {  
  
    Operacao* novaOperacao = (Operacao*)malloc(sizeof(Operacao));  
    if (novaOperacao == NULL) {  
        return NULL;  
    }  
    novaOperacao->codigo = codigo;  
    novaOperacao->next = NULL;  
    novaOperacao->ListaMaquinas = NULL;  
    return novaOperacao;  
}
```

Figura 13 - Função desenvolvida para criar novas operações.

2.2.3. Inserir operações

A função “inserirOperacao” (figura 14) permite inserir as operações que foram criadas a partir da função anterior numa lista. Esta função começa por verificar se a operação a inserir já existe na lista e, se não existir, adiciona-a à lista. Caso a lista ainda não englobe nenhum elemento, então a nova operação é adicionada no início (*h* aponta para a primeira operação). Se a lista incluir elementos, então, através do ciclo *while*, é percorrida até o final, quando “aux->next = NULL”, e a nova operação é inserida. Logo, utilizando esta função, as novas operações serão sempre inseridas no final da lista existente.

```

Operacao* inserirOperacao(Operacao* h, Operacao* nova) {
    if (existeOperacao(h, nova->codigo)) return h;

    if (h == NULL) {
        h = nova;
        return (h);
    }
    else
    {
        Operacao* aux = h;
        while (aux->next != NULL) {
            aux = aux->next;
        }
        aux->next = nova;
    }
    return h;
}

```

Figura 14 - Função para inserir as operações na lista.

A função “existeOperacao” permite evitar que sejam inseridas operações iguais à lista. Esta função recebe como parâmetros o início da lista “Operacao” e o código da operação que se pretende inserir. No caso de a lista estar ainda vazia, a função retorna *false* e a operação é inserida. Se a lista não estiver vazia, vai ser percorrida, verificando se o campo código das operações existentes é igual ao novo. Se encontrar valores iguais, a função retorna *true*, e a operação não será inserida.

```

bool existeOperacao(Operacao* h, int codigo) {
    if (h == NULL) return false;
    Operacao* aux = h;
    while (aux != NULL) {
        if (aux->codigo == codigo) return true;
        aux = aux->next;
    }
    return false;
}

```

Figura 15 - Função para verificar se a operação a inserir ainda não existe na lista.

2.2.4. Remover operação

Para remover uma operação da lista através do seu código, foi criada a função designada “removerOperacao” (figura 16). Para remover a primeira operação da lista é necessário criar uma variável auxiliar com o endereço desta (Operacao* aux =h). De seguida define-se que o início da lista passa a ser a operação seguinte (h=h->next) e a função *free* permite libertar a memória ocupada pela primeira operação (apontada por aux). Se a operação a remover não for a da primeira posição da lista, é necessário criar mais uma variável auxiliar, “auxAnt” que aponta para a operação anterior à apontada por “aux”. A lista é percorrida e quando o código da operação que se pretende remover for encontrado, o campo *next* da operação anterior passa a apontar para a operação seguinte à que vai ser removida. Desta forma, uma operação que se encontre a meio da lista pode ser removida e a lista permanece ligada.

```

Operacao* removerOperacao(Operacao* h, int codigo) {
    if (h == NULL) return NULL;

    if (h->codigo == codigo) {
        Operacao* aux = h;
        h = h->next;
        free(aux);
    }
    else {
        Operacao* aux = h;
        Operacao* auxAnt = aux;
        while (aux && aux->codigo != codigo) {
            auxAnt = aux;
            aux = aux->next;
        }
        if (aux != NULL) {
            auxAnt->next = aux->next;
            free(aux);
        }
    }
    return h;
}

```

Figura 16 - Função para remover operações da lista.

2.2.5. Mostrar lista

De forma a poder verificar a inserção das operações na lista, foi criada a função “mostrarListaOperacao”. Esta função recebe por parâmetro o início da lista que é percorrida. Para cada operação é apresentado na consola o seu código.

```

void mostrarListaOperacao(Operacao* h) {
    Operacao* aux = h;
    while (aux != NULL){
        printf("Operacao %d; ", aux->codigo);
        aux = aux->next;
    } printf("\n");
}

```

Figura 17 - Função desenvolvida para apresentar na consola as operações da lista.

Posteriormente foi criada outra função que permite mostrar as máquinas ligadas a cada operação (figura 19). Neste caso foi necessária a utilização de dois ciclos *while*, um para percorrer a lista das operações e outro que percorre a lista de máquinas. Desta forma para cada operação são apresentadas as respetivas máquinas.

```
void mostrarListaOperacaoMaquinas(Operacao* h) {
    Operacao* aux = h;
    while (aux != NULL) {
        Maquina* auxmaq = aux->ListaMaquinas;
        while (auxmaq != NULL)
        {
            printf("Operacao %d: maquina %s\n", aux->codigo, auxmaq->id);
            auxmaq = auxmaq->next;
        }
        aux = aux->next;
    }
}
```

Figura 18 - Função para mostrar as máquinas que realizam as operações.

2.3. Máquinas

2.3.1. Estrutura de registo para máquinas

Para armazenar a informação sobre as máquinas que podem ser utilizadas em cada operação do *job* foi criada a estrutura da figura 20. Cada máquina tem a sua identificação (*id*), o tempo que demora a concretizar a operação e um apontador para a próxima máquina da lista.

```
typedef struct Maquina {  
    char id [N];  
    int tempo;  
    struct Maquina* next;  
} Maquina;
```

Figura 19 - Estrutura “Maquina” que contém informação sobre cada operação.

2.3.2. Criar máquinas

Esta função permite criar as máquinas e recebem por parâmetro os dados necessários para preencher a *struct*, ou seja, o *id* e o tempo (figura 21). À semelhança da função “*criarOperacao*”, neste caso é também necessário alocar memória para as máquinas.

```
Maquina* criarMaquina(char* id, int tempo) {  
    Maquina* novaMaquina = (Maquina*)malloc(sizeof(Maquina));  
    if (novaMaquina == NULL) return NULL;  
    strcpy(novaMaquina->id, id);  
    novaMaquina->tempo = tempo;  
    novaMaquina->next = NULL;  
    return novaMaquina;  
}
```

Figura 20 - Função para criar máquinas.

2.3.3. Inserir máquinas nas operações

A função “*inserirMaquinaOperacao*” (figura 22) foi desenvolvida para associar as máquinas às respetivas operações. Esta função recebe a lista de operações, o código de uma operação e a máquina a incluir na operação. O ciclo *while* permite percorrer a lista e procurar o código da operação à qual se pretende adicionar a máquina. Se o código existir, a máquina é inserida no campo “*ListaMaquinas*” da *struct* “*Operacao*” através da função “*inserirMaquina*” (figura 23). Esta função recebe por parâmetro o início na lista de máquinas e a máquina a adicionar. À semelhança da função “*inserirOperacao*”, as máquinas também são adicionadas no final da lista.


```

Operacao* inserirMaquinaOperacao(Operacao* h, int codOper, Maquina* m) {
    if (h == NULL) return NULL;
    else
    {
        Operacao* aux = h;
        while (aux != NULL && aux->codigo != codOper) {
            aux = aux->next;
        }
        if (aux != NULL) {
            aux->ListaMaquinas=inserirMaquina(aux->ListaMaquinas, m);
        }
    }
    return (h);
}

```

Figura 21 - Função que permite adicionar uma lista de máquinas nas operações.

```

Maquina* inserirMaquina(Maquina* h, Maquina* nova) {
    if (h == NULL) {
        h = nova;
        return (h);
    }
    else
    {
        Maquina* aux = h;
        while (aux->next != NULL) {
            aux = aux->next;
        }
        if (aux != NULL)
        {
            aux->next = nova;
        }
    }
    return h;
}

```

Figura 22 - Função para inserir uma máquina na lista de máquinas.

2.4. Determinação do tempo necessário para completar um *job*

2.4.1. Duração mínima e máxima do *job*

Para determinar o tempo mínimo necessário para completar um *job* foi definida a função “tempoMinimo” (figura 24). Esta função começa por percorrer a lista de operações e para cada operação, acede à lista de máquinas. O valor do campo “tempo” é comparado entre as máquinas e a variável “tempoMinimo” guarda o menor valor. Para determinar o tempo mínimo de realização do *job*, foi criada a variável “resultado” que soma o valor de tempo mínimo de cada operação.

Uma função semelhante foi também definida para apresentar a tempo máximo que cada operação demora e a respetiva máquina.

```
void tempoMinimo(Operacao* h) {
    Operacao* auxOp = h;
    Maquina* auxMq;
    int tempoMinimo = 0;
    int resultado = 0;

    while (auxOp != NULL) {
        tempoMinimo = 0;

        auxMq = auxOp->ListaMaquinas;

        while (auxMq != NULL) {
            if (auxMq->tempo < tempoMinimo) {
                tempoMinimo = auxMq->tempo;
            }
            else if (tempoMinimo == 0) {
                tempoMinimo = auxMq->tempo;
            }
            auxMq = auxMq->next;
        }
        resultado += tempoMinimo;

        printf("Operacao %d - Tempo mínimo: %d\n", auxOp->codigo, tempoMinimo);

        auxOp = auxOp->next;
    }
    printf("Resultado %d\n", resultado);
}
```

Figura 23 - Função desenvolvida para determinar o tempo mínimo para completar um *job*.

2.4.2. Duração média do *job*

O tempo médio de cada operação e o tempo médio necessário para completar um *job* podem ser apresentados através da função “tempoMedio” (figura 25). Para esta função, é necessária a criação de duas variáveis, uma (“soma”) para guardar a soma do tempo que as máquinas demoram a realizar a operação e outra (“contadorMq”) que guarda a quantidade de máquinas que podem estar envolvidas em cada operação. A média de tempo de cada operação corresponde à divisão entre estas variáveis (“resultadoMq”). O tempo médio do *job* corresponde à soma das médias obtidas para as operações. Esse valor é guardado pela variável “resultado”.

```

void tempoMedio(Operacao* h) {
    Operacao* auxOp = h;
    Maquina* auxMaq;
    float soma = 0;
    int contadorMaq = 0;
    float resultado = 0;
    float resultadoMaq = 0;

    while (auxOp != NULL) {
        soma = 0;
        contadorMaq = 0;

        auxMaq = auxOp->ListaMaquinas;

        while (auxMaq != NULL) {
            soma += auxMaq->tempo;
            contadorMaq++;
            auxMaq = auxMaq->next;
        }
        resultadoMaq = soma / contadorMaq;
        printf("Duracao media da Operacao %d: %.2f \n", auxOp->codigo, resultadoMaq);

        resultado += resultadoMaq;
        auxOp = auxOp->next;
    }

    printf("Duracao media: %.2f\n", resultado);
}

```

Figura 24 - Função que permite determinar a duração média das operações e do *job*.

3. Proposta de escalonamento

3.1.1. Estruturas de registo

Na figura 26 estão apresentadas as estruturas desenvolvidas para o escalonamento. O escalonamento é realizado com base na máquina mais rápida de cada operação. A estrutura “Celula” permite guardar essa informação juntamente com a operação e o *job* correspondente. O “tempoFinal” indica o momento em que a operação seguinte pode iniciar. Por sua vez, a estrutura designada “Planeamento” guarda apenas o Job e a Operação. Esta será utilizada para preencher *array* do planeamento.

```
typedef struct Celula {
    int codigoJob;
    int codigoOper;
    char idMaq [N];
    int tempo;
    int tempoFinal;
} Celula;

typedef struct Planeamento {
    int codigoJob;
    int codigoOper;
} Planeamento;
```

Figura 25 - Estruturas desenvolvidas para o escalonamento.

3.1.2. Iniciar Array

O *array* bidimensional “plano” é utilizado para representar a informação de *jobs* e operações (estrutura “Planeamento”) que utilizam determinada máquina e a duração de ocupação da máquina. Assim, as colunas serão as unidades de tempo e as linhas as máquinas. Foi também necessário um *array* auxiliar para guardar toda a informação dos *jobs*. Ambos são inicializados com o valor -1 (valorIni), através das funções apresentadas na figura 27.

```
void iniciaPlano(Planeamento plano[][T]) {
    for (int l = 0; l < M; l++) {
        for (int c = 0; c < T; c++) {
            plano[l][c].codigoJob = valorIni;
            plano[l][c].codigoOper = valorIni;
        }
    }
}

void iniciarArrayAuxiliar(Celula auxiliar[]) {
    int i;
    for (i = 0; i < T; i++) {
        auxiliar[i].codigoJob = valorIni;
        auxiliar[i].codigoOper = valorIni;
        strcpy(auxiliar[i].idMaq, "valorIni");
        auxiliar[i].tempo = valorIni;
        auxiliar[i].tempoFinal = valorIni;
    }
}
```

Figura 26 - Funções para inicializar os arrays utilizados nesta proposta de escalonamento.

3.1.3. Preencher Célula

A função “preencherCelula” (figura28) permite preencher todos os campos da estrutura “Celula” com a informação contida em cada nodo da árvore. O campo “tempoFinal” corresponde à última unidade de tempo ocupada pela última operação realizada de cada *job*. Estes dados são guardados num array auxiliar, através da função “preencheArrayAuxiliar” (figura 29).

```
void preencherCelula(Job* root) {
    int tempoMinimo;
    int maq;
    Celula auxCel;
    Celula auxiliar[T];

    auxCel.tempoFinal = 0;

    auxCel.codigoJob = root->codigoJob;

    Operacao* auxOper = root->listaOperacoes;
    while (auxOper) {
        auxCel.codigoOper = auxOper->codigo;

        Maquina* auxMaq = auxOper->listaMaquinas;
        tempoMinimo = auxMaq->tempo;
        strcpy(auxCel.idMaq, auxMaq->id);

        while (auxMaq) {
            if (auxMaq->tempo < tempoMinimo) {
                tempoMinimo = auxMaq->tempo;
                strcpy(auxCel.idMaq, auxMaq->id);
            }
            auxMaq = auxMaq->next;
        }
        auxCel.tempo = tempoMinimo;
        auxCel.tempoFinal += tempoMinimo;

        preencheArrayAuxiliar(auxiliar, &auxCel);

        auxOper = auxOper->next;
    }
}
```

Figura 27 - Função que permite preencher os campos de "Celula" com a informação de cada nodo da árvore.

```
void preencheArrayAuxiliar(Celula auxiliar[], Celula* nova) {
    int i;

    for (i = 0; i < T; i++) {
        while (auxiliar[i].codigoJob != valorIni)
            i++;

        auxiliar[i].codigoJob = nova->codigoJob;
        auxiliar[i].codigoOper = nova->codigoOper;
        strcpy(auxiliar[i].idMaq, nova->idMaq);
        auxiliar[i].tempo = nova->tempo;
        auxiliar[i].tempoFinal = nova->tempoFinal;
    }
}
```

Figura 28 - Função "preencherArrayAuxiliar".

3.1.4. Ocupar o Plano

Para preencher o *array* bidimensional “plano” que representa o resultado da proposta foi desenvolvida a função “ocupaPlano”. Esta função começa por encontrar a primeira posição livre na linha correspondente à máquina a ser ocupada. De seguida verifica se existe a quantidade de posições livres seguidas necessárias e quando encontrar preenche o *array*.

```
void ocupaPlano(Planeamento plano[][T], Celula* nova) {  
  
    int maqId;  
    int col = 0;  
    int colIni = 0;  
    int tempoTotal;  
    bool celulaLivre = false;  
  
    maqId = devolveMaquina(nova);  
  
    while (plano[maqId][col].codigoJob != valorIni)  
        col++;  
  
    tempoTotal = nova->tempo + col;  
  
    while ((col < T && celulaLivre == false))  
    {  
        colIni = col;  
        for (; col < tempoTotal; col++) {  
            if (plano[maqId][col].codigoJob != valorIni)  
            {  
                celulaLivre = false;  
                col++;  
                break;  
            }  
        }  
        if (col==tempoTotal)  
            celulaLivre = true;  
    }  
  
    if (celulaLivre) {  
        for (; colIni < tempoTotal; colIni++) {  
            plano[maqId][colIni].codigoJob = nova->codigoJob;  
            plano[maqId][colIni].codigoOper = nova->codigoOper;  
        }  
    }  
}
```

Figura 29 - Função desenvolvida para representar o plano nas condições definidas.

4. Testes ao projeto desenvolvido

4.1.1. Lista de operações

Para criar a lista de operações, na função *main* foi chamada a função “criarOperacao” para, numa primeira etapa, criar quatro operações. De seguida, para as operações serem inseridas numa lista foi chamada a função “inserirOperacao”, tal como se verifica na figura 31. Desta forma fica criada uma lista ligada com as 4 operações.

```
Operacao* op1 = criarOperacao(1);
Operacao* op2 = criarOperacao(2);
Operacao* op3 = criarOperacao(3);
Operacao* op4 = criarOperacao(4);

Operacao* inicio = NULL;

inicio = inserirOperacao(inicio, op1);
inicio = inserirOperacao(inicio, op2);
inicio = inserirOperacao(inicio, op3);
inicio = inserirOperacao(inicio, op4);
```

Figura 30 - Funções utilizadas na função *main* para criar a lista de operações.

Para mostrar a lista criada pelas funções anteriores, utiliza-se a função “mostrarListaOperacao”, obtendo-se o resultado da figura 32.

```
Operacao 1; Operacao 2; Operacao 3; Operacao 4;
```

Figura 31 - Resultado apresentado pela função “mostrarListaOperacao”.

De seguida foi testada a função para remover uma operação, neste caso a operação 2. Como se pode ver na figura 33, apresentando a lista após a aplicação da função, a operação 2 não aparece.

```
removerOperacao(inicio, 2);
mostrarListaOperacao(inicio);
Operacao 1; Operacao 3; Operacao 4;
```

Figura 32 - Aplicação da função *remover* para a operação e listagem das operações.

4.1.2. Inserir as máquinas nas operações

Para associar as máquinas às respectivas operações foram inicialmente criadas 8 máquinas, através da função “criarMaquina” (figura 34). A título de exemplo, a primeira máquina tem o *id* “M1” e demora 4 unidades de tempo a executar a tarefa. Cada máquina foi inserida na operação através da função “inserirMaquinaOperacao”. No primeiro caso, a função recebe a lista das operações, e vai inserir a máquina apontada por “m1” na operação com o código 1.

```
Maquina* m1 = criarMaquina("M1", 4);
Maquina* m2 = criarMaquina("M2", 2);
Maquina* m3 = criarMaquina("M3", 3);
Maquina* m4 = criarMaquina("M4", 5);
Maquina* m5 = criarMaquina("M5", 8);
Maquina* m6 = criarMaquina("M6", 7);
Maquina* m7 = criarMaquina("M4", 6);
Maquina* m8 = criarMaquina("M8", 2);

inicio = inserirMaquinaOperacao(inicio, 1, m1);
inicio = inserirMaquinaOperacao(inicio, 1, m2);
inicio = inserirMaquinaOperacao(inicio, 2, m3);
inicio = inserirMaquinaOperacao(inicio, 2, m4);
inicio = inserirMaquinaOperacao(inicio, 3, m5);
inicio = inserirMaquinaOperacao(inicio, 3, m6);
```

Figura 33 - Conjunto de funções utilizadas para criar as máquinas e inserir nas operações.

Para mostrar o resultado da aplicação das funções utiliza-se a função “mostrarListaOperacaoeMaquinas”. Como se pode observar pela figura 35, definiu-se que cada operação pode ser executada por duas máquinas. Por exemplo, a Operação 1 contém uma lista com duas máquinas, M1 e M2.

```
Operacao 1: maquina M1
Operacao 1: maquina M2
Operacao 2: maquina M3
Operacao 2: maquina M4
Operacao 3: maquina M5
Operacao 3: maquina M6
Operacao 4: maquina M4
Operacao 4: maquina M8
```

Figura 34 – Resultado apresentado pela função “mostrarListaOperacaoeMaquinas”.

4.1.3. Duração das operações

Para verificar a correta aplicação das funções desenvolvidas para determinar o tempo mínimo, máximo e médio do job, utilizaram-se as funções descritas no ponto 2.3., obtendo-se o resultado da figura 36. Tendo como exemplo a operação 2 – pode ser executada pelas máquinas M3 e M4 com 3 e 5 unidades de tempo, respetivamente – conclui-se que o tempo mínimo desta operação é 3, o máximo é 5 e a média é de 4 unidades de tempo, tal como se pode verificar na

figura. O tempo médio também se pode calcular pelo resultado do mínimo e do máximo, isto é, pela média entre 14 e 23.

```
Operacao 1 - Tempo minimo: 2
Operacao 2 - Tempo minimo: 3
Operacao 3 - Tempo minimo: 7
Operacao 4 - Tempo minimo: 2
Total Minimo: 14
Operacao 1 - Tempo maximo 4
Operacao 2 - Tempo maximo 5
Operacao 3 - Tempo maximo 8
Operacao 4 - Tempo maximo 6
Total maximo: 23
Media da Operacao 1: 3.00
Media da Operacao 2: 4.00
Media da Operacao 3: 7.50
Media da Operacao 4: 4.00
Duracao media: 18.50
```

Figura 35 - Resultado das funções desenvolvidas para determinar o tempo necessário para completar as operações e o *job*.

4.1.4. Criação da árvore binária

Para a criação de uma árvore construída a partir de um ficheiro de texto foi escrito um ficheiro com a seguinte estrutura: código do *job*; código da operação; tempo da máquina; identificador da máquina, como mostrado na figura seguinte.

```
4;3;5;M4
4;3;6;M6
4;3;7;M7
4;4;3;M5
4;4;5;M6
4;4;5;M8
1;1;4;M1
1;1;5;M3
1;2;4;M2
```

Figura 36 - Estrutura do ficheiro de texto utilizado.

Como se verifica pela figura 37, os *jobs* não se encontram ordenados. Esta foi a forma escolhida para a árvore ficar balanceada. De forma a visualizar a correta leitura e construção da árvore, foram utilizadas as funções da figura 38. O resultado está apresentado na figura 39, na qual se verifica uma leitura *in-order*, tal como pretendido.

```
Job* root = iniciarArvore();  
  
root=lerFicheiroTexto();  
mostrarArvore(root);
```

Figura 37 - Funções utilizadas no *main* para apresentar a árvore.

```
Root: 1  
Root: 2  
Root: 3  
Root: 4  
Root: 5  
Root: 6  
Root: 7  
Root: 8
```

Figura 38 - Resultado obtido pela função "mostrarArvore".

A função desenvolvida para remover um *job* também foi testada. Neste caso foi passado à função o "2", que corresponde ao job 2, tal como descrito no ponto 2.1.7. Usando novamente a função "mostrarArvore", obtém-se o relatório da figura 40. Pode-se verificar que o nodo com o código de *job* 2, foi removido.

```
Root: 1  
Root: 3  
Root: 4  
Root: 5  
Root: 6  
Root: 7  
Root: 8
```

Figura 39 - Resultado da árvore após usar a função "removerJob" para remover o job 2.

5. Conclusão

A realização deste projeto permitiu aplicar os conhecimentos adquiridos na UC de Estruturas de Dados Avançadas. Além disso, possibilitou desenvolver as capacidades de programação em linguagem C e um maior entendimento das suas funcionalidades.

Numa primeira fase de desenvolvimento do projeto, era pretendida a criação e manipulação de apenas um *job*, pelo que optei por criar uma solução simples que me permitisse consolidar os conceitos de apontadores e listas. O projeto final envolveu a manipulação de vários *jobs*, que foi elaborada com recurso a ABP. No final, foi desenvolvido algum código com o objetivo de criar uma solução de escalonamento para a produção de um artigo. Este código não ficou concluído. Pretende-se que a função “ocupaPlano” obtenha os dados do *array* auxiliar e que procure espaço (unidades de tempo) suficiente para inserir cada operação, sendo que cada operação só pode iniciar depois de a anterior terminar. Apenas após verificadas as condições, insere os dados da estrutura “Planeamento” no array “plano”.

6. Bibliografia

1. Damas, Luís (2007). *Linguagem C* (10ª ed.), LTC.
2. Ficheiros fornecidos pelo professor Luís Ferreira.