



**Engenharia Eletrotécnica e de Computadores**

# Trabalho Prático

Estruturas de Dados Avançadas

2021-2022

Instituto Politécnico do Cávado e do Ave  
João Sampaio 18611



INSTITUTO POLITÉCNICO  
DO CÁVADO E DO AVE  
ESCOLA SUPERIOR DE TECNOLOGIA



# Índice

Introdução .....	5
1. Propósitos e Objetivos - Fase 1 .....	6
2. Estruturas de Dados.....	7
2.1. Ficheiro (leitura/armazenamento).....	10
2.2. Ficheiro (Alocação de memória).....	11
2.3. Criação de um nodo/operação e a sua inserção na lista ligada.....	12
2.4. Apagar um nodo/operação da lista .....	13
2.5. Modificar um elemento da lista .....	15
2.6. Cálculo do tempo de operação .....	18
3. Testes Realizados .....	20
4. Propósitos e Objetivos - Fase 2 .....	24
5. Estruturas de Dados.....	25
5.1. Ficheiro (leitura/armazenamento).....	27
6. Proposta de escalonamento .....	28
6.1. Processo de escalonamento .....	29
1.1. Ficheiro FJSSP.....	32
2. Testes Realizados .....	33
Conclusão .....	37
Bibliografia .....	38

# Glossário

Node – elemento da lista ligada correspondente a uma operação.

EOF (end of file) – referência representativa do final de um ficheiro .txt.

*Flexible Job Shop Problem* (FJSSP) - extensão do problema clássico de escalonamento de job shop, que permite que uma operação seja processada por qualquer máquina de um determinado conjunto.

*user input* – valores inseridos pelo utilizador.

*Job* – processo.

*Process Plan* – plano de processos.

*makespan* – tempo necessário para realizar o produto.

# Introdução

O presente relatório tem a finalidade de demonstrar o funcionamento e lógica referentes ao trabalho prático proposto como avaliação da unidade curricular de Estruturas de Dados Avançadas.

O trabalho prático tem como objetivo sedimentar os conhecimentos relativos a definição e manipulação de estruturas de dados dinâmicas, na linguagem de programação C.

O objetivo principal consiste no desenvolvimento de uma solução digital para o problema de escalonamento denominado *Flexible Job Shop Problem* (FJSSP).

A implementação permitirá gerar uma proposta de escalonamento para a produção de um produto, dividindo-o em diferentes processos ou *jobs*, constituídos por diversas operações, que por sua vez, são constituídas por diferentes máquinas.

Cada máquina apresenta um tempo de operação distinto, onde a implementação deverá de ser capaz de avaliar a melhor opção, de maneira a minimizar o tempo necessário na produção do produto (*makespan*).

O trabalho pratico é dividido em duas fazes distintas.

# 1. Propósitos e Objetivos - Fase 1

Descrição Fase 1:

- Definição de uma estrutura de dados dinâmica para a representação de um *job* com um conjunto finito de  $n$  operações;
- Armazenamento/leitura de ficheiro de texto com representação de um *job*;
- Inserção de uma nova operação;
- Remoção de uma determinada operação;
- Alteração de uma determinada operação;
- Determinação da quantidade mínima de unidades de tempo necessárias para completar o *job* e listagem das respetivas operações;
- Determinação da quantidade máxima de unidades de tempo necessárias para completar o *job* e listagem das respetivas operações;
- Determinação da quantidade média de unidades de tempo necessárias para completar uma operação, considerando todas as alternativas possíveis;

Na alteração/modificação de uma operação, o programa deve ser capaz de:

- Modificar o valor do tempo de operação de uma máquina á escolha, numa operação á escolha;
- Adicionar uma máquina numa operação;
- Remover uma máquina numa operação;
- Ser capaz de permitir modificar apenas máquinas que estão na operação;
- Não permitir o uso de máquinas repetidas na mesma operação;
- Não permitir que o tempo de operação de uma máquina seja nulo.
- Não permitir escolher uma máquina que não exista na operação.

## 2. Estruturas de Dados

Para a solução a implementar optei por uma lista ligada simples (estrutura de dados), de modo a criar uma lista de todas as operações inseridas em cada *job*.

Cada uma dessas operações tem como conteúdo as máquinas utilizadas e o seu tempo de operação, sendo permitido a modificação de todos os parâmetros.

A implementação permite:

- Criar, eliminar ou modificar toda a operação, desde as máquinas que podem ser utilizadas e os seus respetivos tempos de operação.  
Limitações são aplicadas nos processos supracitados, de modo, por exemplo, a não permitir duas máquinas iguais na mesma operação.
- Ficheiro .txt usado para armazenar todas as alterações realizadas na lista, e também carregar o conteúdo no início do programa.
- Cálculo do tempo máximo, mínimo e médio necessário para concluir o *job*.

A seguinte imagem exemplifica a disposição da lista ligada.

Em cada operação existem dados referentes às máquinas e os seus tempos de operação, incluindo também o respetivo endereço da operação seguinte, de forma a criar a conexão entre a lista.

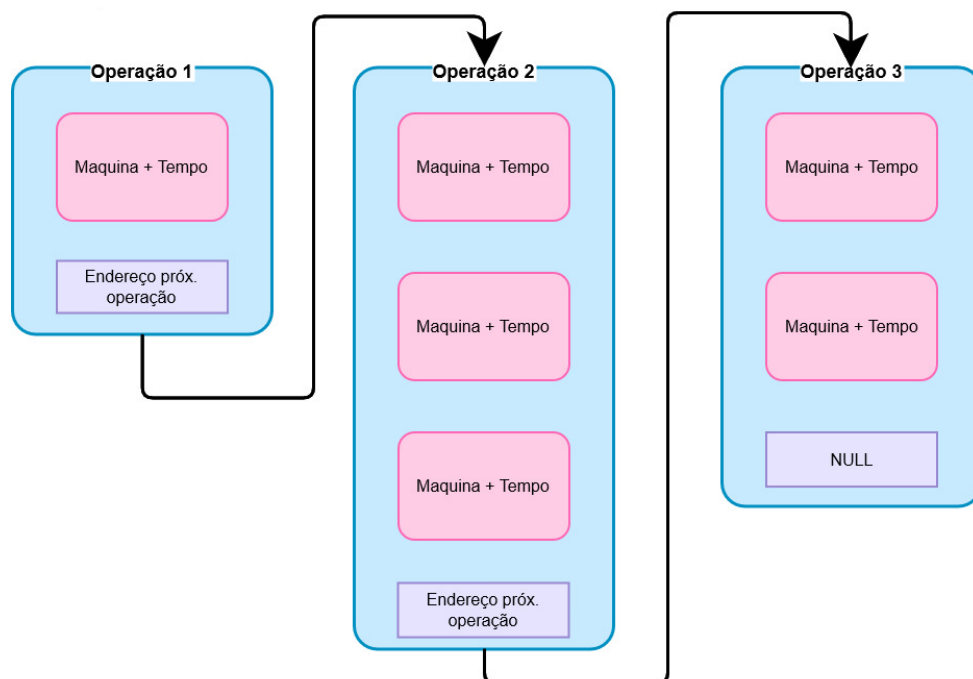


Figura 1 - Disposição da Lista Ligada

Os diferentes elementos de cada operação foram implementados através de apontadores usando alocação de memória dinâmica.

Esta abordagem permite ter em cada *node* (elemento da lista/operação), vários endereços para várias máquinas/tempo de operação, sem a necessidade de recorrer a memória estática como *arrays*, que iriam limitar o programa.

Ao último endereço da lista é atribuído o “valor” *NULL*, de forma a simbolizar o final da lista ligada.



Transcrevendo o diagrama anterior para código em linguagem C, construí a seguinte estrutura, que permitirá a criação das operações.

```
4
5  typedef struct operation
6  {
7      int counter; // to count the number of machines inside an operation
8      int* machineOperationTime;
9      int* machineNumber;
10     struct operation* next; // next position on the list
11 } operation;
```

Figura 2 - Estrutura representativa de uma operação

***machineOperationTime*** – representa o tempo de operação.

***machineNumber*** – representa a máquina.

***next*** – representa o endereço do próximo *node* na lista ligada.

***counter*** – variável auxiliar que permite contar a quantidade de máquinas, facilitando diversos processos como a leitura do ficheiro.

## 2.1. Ficheiro (leitura/armazenamento)

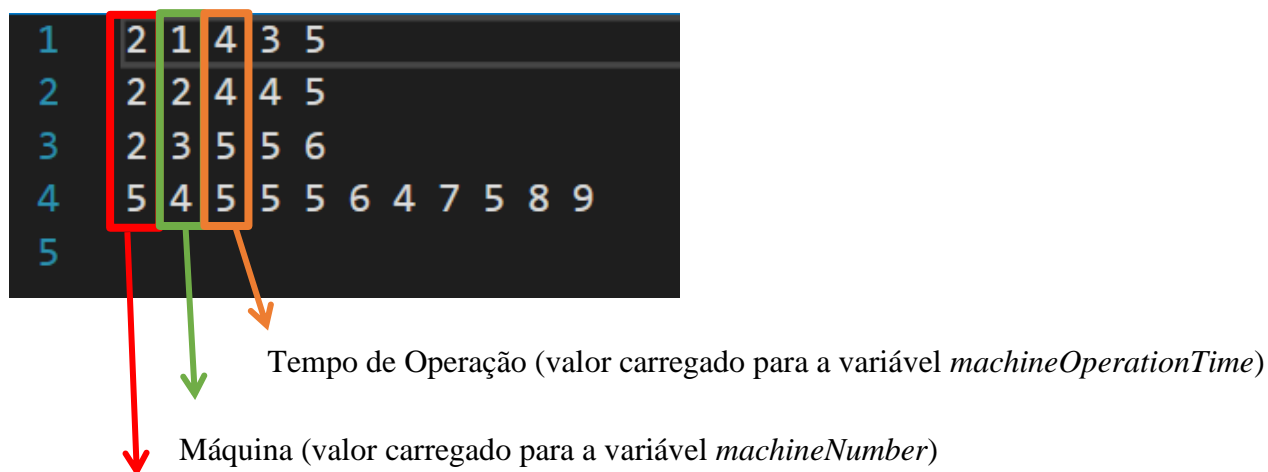
O processo de leitura do ficheiro e criação da lista ligada, no início da execução do programa, demonstrou-se um dos maiores desafios desta implementação. No entanto fui capaz de resolver o problema recorrendo a um método de leitura simplificado e funcional.

Sem comprometer a eficácia da implementação, organizei da maneira mais funcional possível o ficheiro, tendo como primeiro elemento da linha a quantidade de máquinas que fazem parte da operação.

Este valor inicialmente lido do ficheiro permite ao programa saber a quantidade de máquina que constituem a operação.

O valor será carregado na variável *counter*, incluída na estrutura de uma operação, permitindo recorrer a essa variável através de ciclos.

Em cada repetição do ciclo, é realizada uma alocação de memória para cada máquina e o seu respetivo tempo de operação.



Contador de máquinas dentro da operação (valor carregado para a variável *counter*)

Figura 3 - Disposição do ficheiro de leitura/armazenamento da lista

## 2.2. Ficheiro (Alocação de memória)

De forma a conseguir incluir mais que uma máquina e o seu respetivo tempo de operação, numa única operação, realizei uma alocação de memória dinâmica conforme o que fora gravado no ficheiro.

```
temporary->valueReadMachine = (int*)malloc(sizeof(int) * saveValue);
temporary->valueReadOpTime = (int*)malloc(sizeof(int) * saveValue);
```

Figura 4 - Alocação de memória com malloc()

Inicialmente existe uma declaração das variáveis e a sua respetiva alocação de espaço. Ao alocar espaço é necessário ter em conta o tipo de variável, neste caso, apontador para inteiro, e o seu tamanho, descrito na variável *saveValue*, que corresponde ao valor lido e carregado na variável *counter*.

De seguida, através de um ciclo, atribuímos os valores inseridos, ou extraídos do ficheiro, e inserimos nas respetivas variáveis. O ciclo irá possibilitar a resolução do objetivo que é colocar várias máquinas na mesma operação.

```
for (int i = 0; i < saveValue; i++) { // loops the amount of times needed to fill the operation
    fscanf(file, "%d ", &temporary->valueReadMachine[i]);
    fscanf(file, "%d ", &temporary->valueReadOpTime[i]);
}
// calls the function to create the list, with all the values and the counter
createOp = createNodeFile(head, temporary->valueReadMachine, temporary->valueReadOpTime, saveValue);
*head = insertAtTail(head, createOp, NULL);
createOp->next = NULL;
```

Figura 5 Ciclo de leitura do ficheiro

Após o ciclo da operação estar terminado, a função de criação do *node* é invocada. Após a criação da operação, a função retorna o *node* que foi criado, sendo recebido na variável *createOp* do tipo *operation\** (tipo igual á estrutura).

Esse *node* é enviado como parâmetro na invocação da função de inserção na lista ligada, inserindo sempre no final da lista. Após finalizar sem erros, a cabeça da lista é retornada, permitindo a sua atualização. O endereço seguinte da lista será nulo, até outro nodo ser criado.

O processo decorre até terminar o ficheiro, ou seja, até EOF (*end of file*).

Na gravação do ficheiro apenas é necessário percorrer toda a lista ligada, escrevendo cada valor.

### 2.3. Criação de um nodo/operação e a sua inserção na lista ligada

Duas funções distintas foram implementadas de modo a criar um *node* na lista ligada.

Uma das funções apenas será utilizada quando é necessário realizar uma leitura do ficheiro, ou seja, no início da execução do programa (como fora explicado no tópico anterior). A outra permite a criação de *nodes* novos (operações novas) durante a execução do programa.

A decisão de separar em duas funções distintas tem o intuito de simplificar ao máximo o programa, de maneira a criar uma implementação de fácil compressão e sem comprometer a sua funcionalidade.

Novamente nesta segunda função, é utilizada uma variável que indicará qual é a quantidade de máquinas que irão fazer parte da operação, no entanto, desta vez, essa variável e todos os valores das máquinas e os seus respetivos tempos de operação, serão obtidos através de *user input*.

```

}else{
    printf("Choose the time of operation of machine: %d\n", inputMachine);
    scanf("%d", &inputOpTime);
    // adds the new node to the linked list
    node->machineNumber[i] = inputMachine; // adds a machine
    node->machineOperationTime[i] = inputOpTime; //adds a operation time fo
    counter++;
    node->counter = counter;
    previousMachine = inputMachine;
}

```

Figura 6 - Criação de um node

Após a leitura ou criação dos *nodes* pelo utilizador, o respetivo *node* criado não consta na lista ligada, ainda é necessário realizar a sua inserção e criar a ligação com o próximo elemento.

O programa tem a capacidade de realizar a inserção em qualquer posição, desde que seja especificado como parâmetro de entrada, ou seja escolhido através da interface gráfica com o utilizador. Por predefinição a inserção é realizada no final da lista ligada (exemplos no tópico Testes Realizados).

## 2.4. Apagar um nodo/operação da lista

De forma a eliminar um *node*/operação da lista, apenas é necessário chamar a função correspondente, enviando como parâmetro de entrada a cabeça da lista e o nodo a ser removido, que é posteriormente selecionado pelo utilizador (variável *option\_2*), e encontrado na lista através da função de procura de *nodes* (*find\_node()*).

```

if (scanf("%d", &option_2) > 0)
    if (option_2 > 0) {
        deleteNode(&head, find_node(head, option_2));
    }
    else break;

```

Figura 7 - Chamada da função que eliminará o node escolhido

Na função *deleteNode()*, inicialmente temos uma avaliação de qual posição é a escolhida. Se o nodo corresponder á cabeça da lista, a função está preparada para identificar a situação e eliminar a cabeça da lista. No entanto, não é possível simplesmente eliminar esse nodo, senão criaria problemas na lista ligada, faltando a ligação para as restantes posições.

De forma a colmatar esse problema, a cabeça da lista é transferida para a segunda posição, criando assim uma nova cabeça e atualizando o restante. Após essa atualização, o *node* que posteriormente estava na cabeça, é eliminado através da função *free()*.

Se o *node* escolhido não for o que está na cabeça da lista, o programa irá percorrer toda a lista ligada até encontrar o *node* em questão, e através da função *free()*, esse *node* será libertado deixando de ocupar espaço na memória.

```
if ((*head)->machineNumber == value->machineNumber)
{
    temporary = *head;    //backup to free the memory
    *head = (*head)->next;
    free(temporary);
}
```

Figura 8 - Remoção do node correspondente á cabeça da lista ligada

A função *free()* é sempre utilizada em casos que existe uma alocação de memória dinâmica e é necessário libertá-la.

## 2.5. Modificar um elemento da lista

Como operações de modificação, optei pelas seguintes:

- Modificar o valor do tempo de operação de uma máquina á escolha numa operação á escolha.
- Adicionar uma máquina numa operação.
- Remover uma máquina numa operação.

Adicionar uma nova máquina:

```
case 1: // add new item
    //adds the new machine and operation time, to the last position inside
    nodeToModify->machineNumber[nodeToModify->counter] = addMachine;
    nodeToModify->machineOperationTime[nodeToModify->counter] = addOpTime;
    nodeToModify->counter++;
    break;
```

*Figura 9 - Adição de uma máquina numa operação existente*

Na seguinte opção, o *node* enviado como parâmetro de entrada da função, que foi posteriormente escolhido pelo utilizador, é encontrado através da função *find\_node()*.

Se uma operação tiver 2 elementos, ou seja, duas máquinas e os seus respetivos tempos de operação, a variável *counter* terá o valor 2 guardado. No entanto, as posições ocupadas nessa operação para as máquinas e tempos são [0] e [1], alocados dinamicamente.

Por essa razão apenas necessito de colocar como parâmetro dentro dos parenteses retos, a variável *counter*, que irá sempre corresponder ao espaço seguinte, neste caso [2]. O valor guardado em *counter* é atualizado após inseridos os valores escolhidos.

O que está descrito na imagem anterior, apenas será realizado na eventualidade da máquina que está a ser inserida, já não constar na operação. Se já existir, o programa não permitirá a sua colocação.

### Remover uma máquina existente:

```

if (scanf("%d", &option) > 0) {
    while (counter < nodeToModify->counter) { // save the node in a buffer /
        bufferMachine[counter] = nodeToModify->machineNumber[counter];
        bufferOpTime[counter] = nodeToModify->machineOperationTime[counter];
        counter++;
        if (nodeToModify->machineNumber[counter] == option) { // saves the p
            pos = counter;
        }
    }
    // starts on the position to delete, and reorganizes the arrays
    for (counter = pos; counter < nodeToModify->counter; counter++) {
        bufferMachine[counter] = bufferMachine[counter + 1];
        bufferOpTime[counter] = bufferOpTime[counter + 1];
    }
}
nodeToModify->counter = counter - 1; // decrease one number on the operation
counter = 0;
while (counter < nodeToModify->counter) { // fill the node again without the
    nodeToModify->machineNumber[counter] = bufferMachine[counter];
    nodeToModify->machineOperationTime[counter] = bufferOpTime[counter];
    counter++;
}

```

Figura 10 Processo de eliminação da máquina

O utilizador escolhe inicialmente uma máquina que irá ser identificada dentro da operação, e o seu valor juntamente com o valor tempo de operação, serão copiados para dois buffers (*arrays*).

Após a finalização dessa etapa, executo uma operação que irá retirar o valor da posição onde se encontra a máquina a ser removida, substituindo o valor presente nessa posição, pelo valor na posição seguinte do *array*.

O valor na variável *counter* irá ser decrementado, devido á eliminação de uma máquina, e todos os elementos do array são copiados novamente para o *node*.

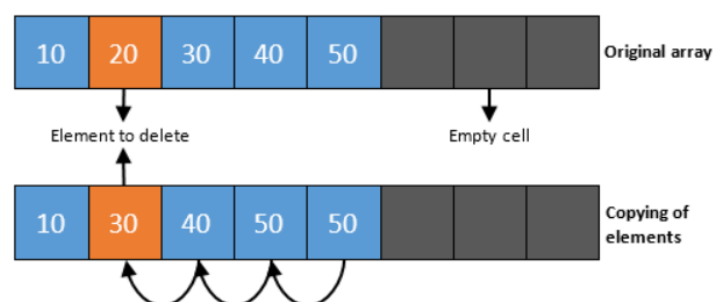


Figura 11 - Explicação gráfica do procedimento



Como característica adicional, o programa é capaz de identificar quando a operação ficou sem máquinas disponíveis, resultando na eliminação por completo da operação.

```
else { // delete the operation since there is no
    printf("Last machine available deleted.\n");
    printf("Operation deleted.\n");
    deleteNode(head, nodeToModify); // function c
}
```

*Figura 12 - Eliminação total da operação*

#### Modificar o tempo de operação:

É escolhida uma máquina existente na operação, sendo o valor do seu tempo de operação atualizado, conforme o que for escolhido pelo utilizador.

```
if (nodeToModify->machineNumber[i] == option) { // identify the machine chosen
    printf("\nPrevious operation time: %d\n", nodeToModify->machineOperationTime[i]);
    printf("New operation time: ");
    if(scanf("%d", &option) > 0) // reutilizing the option variable
        nodeToModify->machineOperationTime[i] = option; // update the operation time of sa
}
```

*Figura 13 - Modificação do tempo de operação*

## 2.6. Cálculo do tempo de operação

De forma a realizar o cálculo do tempo de operação no *job*, utilizei uma abordagem simples.

Para realizar o cálculo do valor máximo e mínimo apenas recorri ao uso de ciclos. Um ciclo para percorrer toda a lista e um ciclo para poder atualizar uma variável.

Sempre que o valor máximo é encontrado na operação, ele é guardado e mais tarde somado (ao valor da operação anterior), de forma a obter o valor máximo de tempo que teria de decorrer até terminar o *job*.

Isto aconteceria se o sistema optasse pelas máquinas com mais tempo de operação por cada operação.

Para o tempo mínimo, o contrário é aplicado ao código representado na imagem seguinte, atualizando sempre que encontrar o valor mínimo na operação.

```
while (temporary != NULL) {  
    for (int i = 0; i < temporary->counter; i++) {  
        if (i == 0) {  
            max = temporary->machineOperationTime[i];  
        }  
        else if (max <= temporary->machineOperationTime[i]) {  
            max = temporary->machineOperationTime[i];  
        }  
    }  
    sum += max;  
    temporary = temporary->next;  
}
```

Figura 14 - Cálculo do valor máximo de tempo de operação até concluir o *job*

Para o cálculo da média de valores apenas é necessário realizar um somatório de todos os tempos de operação em todas as operações, e dividir pela quantidade de máquinas no *job* inteiro.

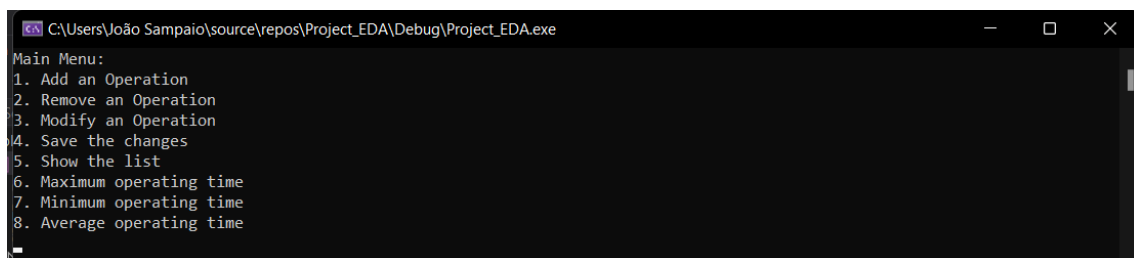
```
while (temporary != NULL) {  
    for (int i = 0; i < temporary->counter; i++) {  
        sum += temporary->machineOperationTime[i];  
    }  
    count += temporary->counter;  
    temporary = temporary->next;  
}
```

*Figura 15 - Cálculo do valor máximo de tempo de operação até concluir o job*

### 3. Testes Realizados

De modo a facilitar a compreensão do funcionamento da solução implementada, apresento os seguintes exemplos:

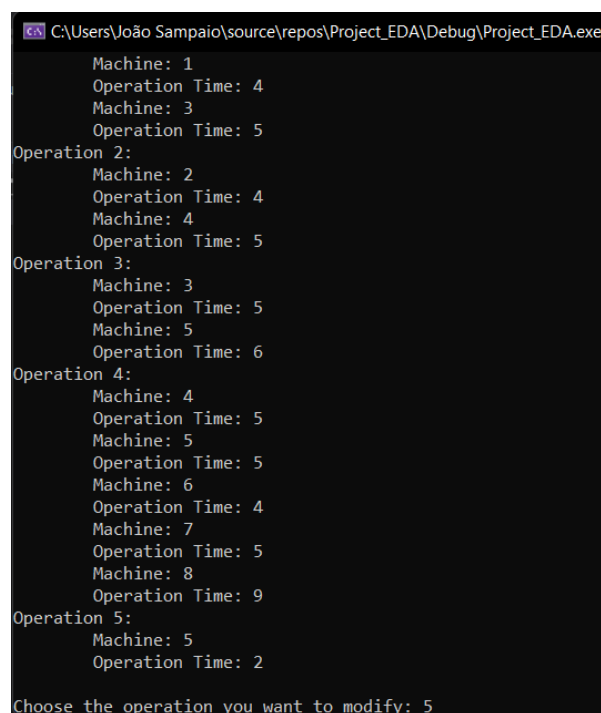
O menu principal apresenta todas as possibilidades de operações a realizar, de modo a criar um *job*. Para aceder aos submenus de cada operação, é necessário fornecer ao programa o *input* do número referente à operação desejada.

A screenshot of a Windows command prompt window titled "C:\Users\João Sampaio\source\repos\Project\_EDA\Debug\Project\_EDA.exe". The window displays the "Main Menu:" with a list of eight options: 1. Add an Operation, 2. Remove an Operation, 3. Modify an Operation, 4. Save the changes, 5. Show the list, 6. Maximum operating time, 7. Minimum operating time, and 8. Average operating time. A cursor is visible at the bottom of the list.

```
C:\Users\João Sampaio\source\repos\Project_EDA\Debug\Project_EDA.exe
Main Menu:
1. Add an Operation
2. Remove an Operation
3. Modify an Operation
4. Save the changes
5. Show the list
6. Maximum operating time
7. Minimum operating time
8. Average operating time
```

Figura 16 - Menu Principal

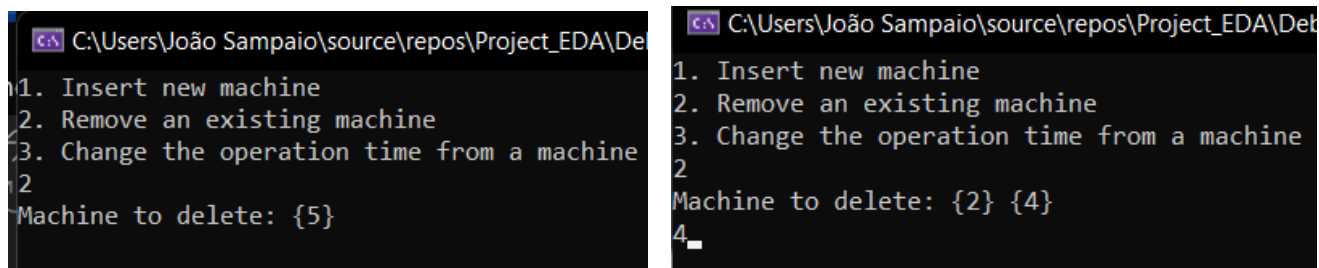
Escolhendo a modificação de uma operação, acedemos ao menu de modificação, onde toda a lista é apresentada e como *input* o utilizador escolherá a operação que pretende modificar.

A screenshot of the same Windows command prompt window, now showing the "Operation 1:" through "Operation 5:" menu. Each operation lists its machine and operation time. At the bottom, it prompts the user to "Choose the operation you want to modify: 5".

```
C:\Users\João Sampaio\source\repos\Project_EDA\Debug\Project_EDA.exe
Machine: 1
Operation Time: 4
Machine: 3
Operation Time: 5
Operation 2:
Machine: 2
Operation Time: 4
Machine: 4
Operation Time: 5
Operation 3:
Machine: 3
Operation Time: 5
Machine: 5
Operation Time: 6
Operation 4:
Machine: 4
Operation Time: 5
Machine: 5
Operation Time: 5
Machine: 6
Operation Time: 4
Machine: 7
Operation Time: 5
Machine: 8
Operation Time: 9
Operation 5:
Machine: 5
Operation Time: 2
Choose the operation you want to modify: 5
```

Figura 17 - Escolha da operação a modificar

Como exemplo, irei executar um processo de remoção de uma máquina na operação 2 e na operação 5, removendo a máquina 4 e a 5, respetivamente.

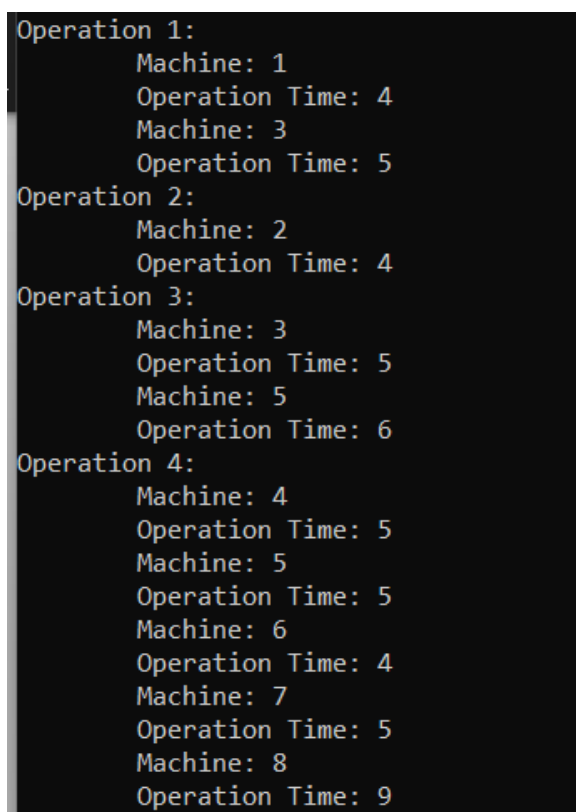


The figure consists of two side-by-side terminal windows. The left window shows a list of three operations: '1. Insert new machine', '2. Remove an existing machine', and '3. Change the operation time from a machine'. Below the list, the user has entered '2' and 'Machine to delete: {5}'. The right window shows the same list of operations. Below the list, the user has entered '2' and 'Machine to delete: {2} {4}', followed by a cursor on the next line.

```
C:\Users\João Sampaio\source\repos\Project_EDA\De
1. Insert new machine
2. Remove an existing machine
3. Change the operation time from a machine
2
Machine to delete: {5}

C:\Users\João Sampaio\source\repos\Project_EDA\De
1. Insert new machine
2. Remove an existing machine
3. Change the operation time from a machine
2
Machine to delete: {2} {4}
4
```

Figura 18 - Processo de remoção máquina 5 e máquina 4



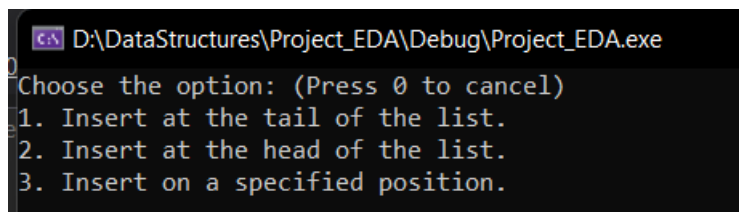
The terminal window displays a list of operations grouped by their type. Operation 1 has two entries for Machine 1 and Machine 3. Operation 2 has one entry for Machine 2. Operation 3 has two entries for Machine 3 and Machine 5. Operation 4 has five entries for Machine 4, Machine 5, Machine 6, Machine 7, and Machine 8.

```
Operation 1:
  Machine: 1
  Operation Time: 4
  Machine: 3
  Operation Time: 5
Operation 2:
  Machine: 2
  Operation Time: 4
Operation 3:
  Machine: 3
  Operation Time: 5
  Machine: 5
  Operation Time: 6
Operation 4:
  Machine: 4
  Operation Time: 5
  Machine: 5
  Operation Time: 5
  Machine: 6
  Operation Time: 4
  Machine: 7
  Operation Time: 5
  Machine: 8
  Operation Time: 9
```

Figura 19 - Lista final após a modificação

Como podemos verificar, as máquinas já não fazem parte da lista de operações.

### Exemplo de criação de um node:

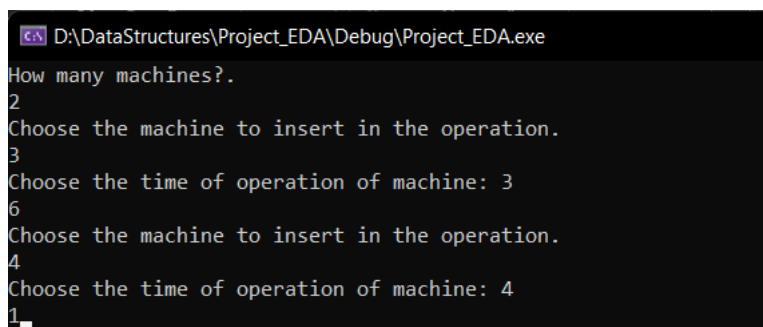


```
D:\DataStructures\Project_EDA\Debug\Project_EDA.exe
Choose the option: (Press 0 to cancel)
1. Insert at the tail of the list.
2. Insert at the head of the list.
3. Insert on a specified position.
```

Figura 20 - Submenu da criação de uma operação

Irei colocar a nova operação na 3<sup>o</sup> posição da lista.

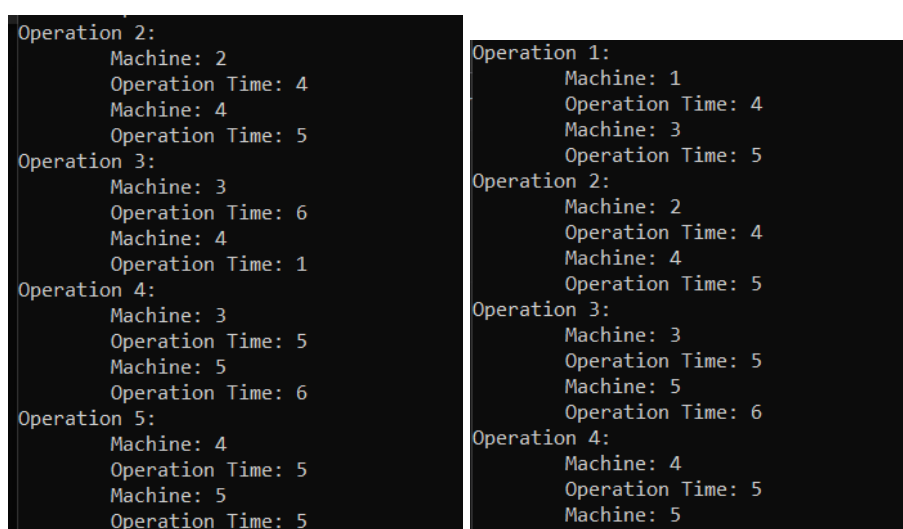
Após escolher a 3<sup>o</sup> posição, o programa necessita que o utilizador indique quantas máquinas serão utilizadas, o seu valor e o tempo de operação.



```
D:\DataStructures\Project_EDA\Debug\Project_EDA.exe
How many machines?..
2
Choose the machine to insert in the operation.
3
Choose the time of operation of machine: 3
6
Choose the machine to insert in the operation.
4
Choose the time of operation of machine: 4
1
```

Figura 21 - Escolha das máquinas/tempo de operação

Como podemos comparar, o novo *node* foi adicionado na 3<sup>a</sup> posição, reordenando a lista.



```
Operation 2:
Machine: 2
Operation Time: 4
Machine: 4
Operation Time: 5
Operation 3:
Machine: 3
Operation Time: 6
Machine: 4
Operation Time: 1
Operation 4:
Machine: 3
Operation Time: 5
Machine: 5
Operation Time: 6
Operation 5:
Machine: 4
Operation Time: 5
Machine: 5
Operation Time: 5

Operation 1:
Machine: 1
Operation Time: 4
Machine: 3
Operation Time: 5
Operation 2:
Machine: 2
Operation Time: 4
Machine: 4
Operation Time: 5
Operation 3:
Machine: 3
Operation Time: 5
Machine: 5
Operation Time: 6
Operation 4:
Machine: 4
Operation Time: 5
Machine: 5
Operation Time: 5
```

Figura 22 - Antes e depois da lista ligada

### Exemplo de remoção de uma operação:

Utilizando a operação criada anteriormente, procedemos á sua eliminação.

```
Operation Time: 9
Choose the operation to delete.(Press 0 to go back)
3
```

*Figura 23 - Escolha da operação a apagar*

Após a seleção, o *node* adicionado á lista no exemplo anterior foi removido, e a lista voltou a ser reordenada.

```
Operation Time: 5
Operation 2:
Machine: 2
Operation Time: 4
Machine: 4
Operation Time: 5
Operation 3:
Machine: 3
Operation Time: 5
Machine: 5
Operation Time: 6
Operation 4:
Machine: 4
Operation Time: 5
Machine: 5
Operation Time: 5
Machine: 6
Operation Time: 4
Machine: 7
```

*Figura 24 - Lista após a remoção*

Para todos os exemplos anteriores, considere como lista ligada, a tabela apresentada no enunciado do trabalho pratico.

## 4. Propósitos e Objetivos - Fase 2

### Descrição Fase 2:

- Definição de uma estrutura de dados dinâmica para representação de um conjunto finito de  $m$  jobs
- associando a cada job um determinado conjunto finito de operações;
- Armazenamento/leitura de ficheiro de texto com representação de um process plan (considerar obrigatoriamente para efeito de teste o process plan da Tabela 1);
- Inserção de um novo job;
- Remoção de um job;
- Inserção de uma nova operação num job;
- Remoção de uma determinada operação de um job;
- Edição das operações associadas a um job;
- Cálculo de uma proposta de escalonamento para o problema FJSSP (obrigatoriamente limitado a um tempo máximo de processamento configurável), apresentando a distribuição das operações pelas várias máquinas, minimizando o makespan (unidades de tempo necessárias para a realização de todos os jobs). A proposta de escalonamento deverá ser exportada para um ficheiro de texto possibilitando uma interpretação intuitiva (utilizar por exemplo um formato tabular ou representação gráfica html, ou outra);
- Representação de diferentes process plan (variando a quantidade de máquinas disponíveis, quantidade de job, e sequência de operações, etc) associando as respetivas propostas de escalonamento.



## 5. Estruturas de Dados

Nesta segunda fase inicialmente procedi á modificação da estrutura de dados anterior (Fase 1), mudando para uma lista duplamente ligada, visando explorar a vantagem de percorrer essa lista em duas direções distintas. Também usei o mesmo tipo de lista para esta segunda fase.

De modo a construir uma tabela como a de exemplo no enunciado do trabalho prático, ou seja, 8x7 (jobs x operações), foi necessário incluir em cada registo da lista de jobs uma declaração de uma lista de operações, como está representado na imagem seguinte.

```
typedef struct operation
{
    // used on the file reading
    int* valueReadMachine;
    int* valueReadOpTime;
    int counter; // to count th
    // actual machines and oper
    int* machineOperationTime;
    int* machineNumber;
    struct operation* next; //
    struct operation* previous;
} operation;

typedef struct job
{
    operation* operation; // li
    struct job* next; // next p
    struct job* previous; // pr
} job;
```

Figura 25 – Novas estruturas

struct job\* next – guarda o endereço do próximo registo na lista de jobs.

struct job\* previous – guarda o endereço do registo anterior na lista de jobs.

operation\* operation – declaração de uma lista de operações em cada registo da lista de jobs.

Na imagem seguinte está representada a tabela utilizada para criar o plano de processo.

Process Plan	Operation						
	0 1	0 2	0 3	0 4	0 5	0 6	0 7
pr <sub>1,2</sub>	(1,3) [4,5]	(2,4) [4,5]	(3,5) [5,6]	(4,5,6,7,8) [5,5,4,5,9]			
pr <sub>2,2</sub>	(1,3,5) [1,5,7]	(4,8) [5,4]	(4,6) [1,6]	(4,7,8) [4,4,7]	(4,6) [1,2]	(1,6,8) [5,6,4]	(4) [4]
pr <sub>3,3</sub>	(2,3,8) [7,6,8]	(4,8) [7,7]	(3,5,7) [7,8,7]	(4,6) [7,8]	(1,2) [1,4]		
pr <sub>4,2</sub>	(1,3,5) [4,3,7]	(2,8) [4,4]	(3,4,6,7) [4,5,6,7]	(5,6,8) [3,5,5]			
pr <sub>5,1</sub>	(1) [3]	(2,4) [4,5]	(3,8) [4,4]	(5,6,8) [3,3,3]	(4,6) [5,4]		
pr <sub>6,3</sub>	(1,2,3) [3,5,6]	(4,5) [7,8]	(3,6) [9,8]				
pr <sub>7,2</sub>	(3,5,6) [4,5,4]	(4,7,8) [4,6,4]	(1,3,4,5) [3,3,4,5]	(4,6,8) [4,6,5]	(1,3) [3,3]		
pr <sub>8,1</sub>	(1,2,6) [3,4,4]	(4,5,8) [6,5,4]	(3,7) [4,5]	(4,6) [4,6]	(7,8) [1,2]		

Figura 26 – Tabela com o process plan usado na implementação

As funções respetivas às operações são iguais, visto que o processo de criação da lista ligada é o mesmo.

Funções essas de criação de um nodo, remoção de um nodo, mostrar a lista, guardar as alterações nos ficheiros e a proposta de escalonamento.

No entanto, agora ao criar um novo nodo, há necessidade de declarar uma lista ligada de operações, onde lhe é atribuída o valor NULL inicialmente.

Após isso realizo chamadas às funções da fase 1, de modo a criar as operações com as respetivas máquinas e tempos de operação.

Ao escolher a operação modificar e escolhendo o job onde serão aplicadas as modificações, o programa acede ao menu anterior da fase 1 com todas as operações.

## 5.1. Ficheiro (leitura/armazenamento)

O processo de leitura do ficheiro foi modificado, de maneira a simplificar a sua compreensão.

Para tal foram criados dois ficheiros distintos, um contendo a informação das operações, e outro contendo a quantidade de operações por cada job e a respetiva quantidade de jobs.

1	4	
2	7	
3	5	
4	4	
5	5	
6	3	
7	5	
8	5	
9		

Job 1 com 4 operações

Job 2 com 7 operações

São 8 jobs no total

1	2	1	4	3	5					
2	2	2	4	4	5					
3	2	3	5	5	6					
4	5	4	5	5	5	6	4	7	5	8
5	3	1	1	3	5	5	7			
6	2	4	5	8	4					
7	2	4	1	6	6					
8	3	4	4	7	4	8	7			
9	2	4	1	6	2					
10	3	1	5	6	6	8	4			
11	1	4	4							

Job 1 com 4 operações

Job 2 com 7 operações

No segundo ficheiro repete-se o esquema apresentado na fase 1 (tópico 2.1), onde a primeira coluna representa a quantidade de máquinas para cada operação, seguida pelo valor da primeira máquina e depois o seu tempo de operação.

## 6. Proposta de escalonamento

O objetivo principal consiste no desenvolvimento de uma solução digital para o problema de escalonamento denominado *Flexible Job Shop Problem* (FJSSP).

Cada máquina apresenta um tempo de operação distinto, onde a implementação deverá de ser capaz de avaliar a melhor opção, de maneira a minimizar o tempo necessário na produção do produto (*makespan*).

Como solução criei um algoritmo que percorre toda lista de jobs avaliando cada operação individualmente.

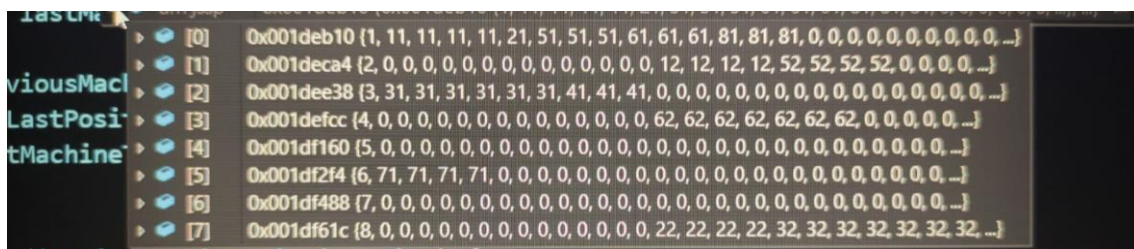
A avaliação de cada operação trata-se de uma comparação do tempo de operação de todas as máquinas que a constituem, escolhendo aquela com menos unidades de tempo.

Após ser escolhida, essa informação é guardada num *array* bidimensional de inteiros `arrFjssp[8][100]`, com todas as suas posições a 0 através da função `memset` (permite realizar comparações).

Cada posição [0] é ocupada pelo valor de uma máquina, existindo no máximo 8 máquinas, e 100 unidades de tempo máximas para a realização do *process plan*. Estes valores são configuráveis para cada *process plan* diferente.

Ao percorrer cada operação, localizando o valor mínimo de tempo de operação, a máquina onde consta esse tempo será escolhida e guardada na sua respetiva linha no *array*. Conforme o tempo de operação, por exemplo 4 unidades de tempo, 4 posições do *array* serão ocupadas de forma a simbolizar as 4 unidades de tempo necessárias até á conclusão.

No *array* são guardados o valor do job e da operação, através de uma multiplicação x10, permitindo ter como primeiro dígito o job e como segundo dígito a operação. No processo de armazenamento do *Process Plan*, separo os dígitos através do módulo %10.



Index	Machine ID (Hex)	Job 0	Job 1	Job 2	Job 3	Job 4	Job 5	Job 6	Job 7
[0]	0x001deb10	11	11	11	11	21	51	51	51
[1]	0x001deca4	2	0	0	0	0	0	0	0
[2]	0x001dee38	3	31	31	31	31	31	41	41
[3]	0x001defcc	4	0	0	0	0	0	0	0
[4]	0x001df160	5	0	0	0	0	0	0	0
[5]	0x001df2f4	6	71	71	71	71	0	0	0
[6]	0x001df488	7	0	0	0	0	0	0	0
[7]	0x001df61c	8	0	0	0	0	0	0	0

Figura 27 – Array que guarda a proposta de escalonamento

## 6.1. Processo de escalonamento

Realização do processo de proposta de escalonamento.

Inicialmente coloco todos os valores das posições a 0. Esta abordagem permite realizar comparações, como por exemplo na escrita do ficheiro, para colocar três pontos em todas as posições desocupadas.

Após isso realizo um preenchimento das posições iniciais com os valores das máquinas, tanto no *array* principal como no auxiliar.

```
memset(arrFjssp, 0, sizeof(arrFjssp));
memset(arrLastPosition, 0, sizeof(arrLastPosition));

// every line is a machine
for (int i = 0; i < 8; i++) {
    arrFjssp[i][0] = i + 1; // put all possible machi
    arrLastPosition[i][0] = i + 1;
}
```

Figura 27 – Inicialização dos arrays

De seguida, o programa realiza a gravação no *array* de todos os endereços de memória de cada operação.

```
for (int j = 0; j < 8; j++) {
    // this condition prevents the program from trigge
    if (temporary != NULL) {
        auxiliar = temporary->operation;
        for (int i = 0; i < 8; i++) {
            if (auxiliar != NULL) { // prevents the pr
                arrAdressOperation[j][i] = auxiliar;
                auxiliar = auxiliar->next;
            }
        }
        temporary = temporary->next;
    }
}
```

Figura 28 – Organização dos endereços no array

No processo de seleção da máquina reutilizo o algoritmo na função do tempo mínimo da fase 1, ligeiramente modificada para o novo propósito.

O endereço é atualizado a cada mudança de operação, e carregado na variável do mesmo tipo, auxiliar.

```
for (int j = 0; j < 8; j++) {
    jobCounter++;
    auxiliar = arrAddressOperation[j][operationCounter]; // access to the address
    if (auxiliar != NULL) {
        minTime = 0; // reset minimum time info variable
        for (int i = 0; i < auxiliar->counter; i++) {
            if (minTime >= auxiliar->machineOperationTime[i]) {
                // if there is a lower value it will update the variable min
                minTime = auxiliar->machineOperationTime[i];
                saveMachine = auxiliar->machineNumber[i];
            }
            else if (minTime == 0) {
                // first time updating the min value / creating the reference
                minTime = auxiliar->machineOperationTime[i];
                saveMachine = auxiliar->machineNumber[i];
            }
        }
    }
}
```

Figura 29 – Algoritmo de seleção de máquinas

Se for a primeira vez que uma máquina foi selecionada, o algoritmo irá proceder à colocação dos dados na respectiva posição no array, e à atualização do array de posições, com as últimas posições.

```
if (arrLastPosition[saveMachine - 1][1] == 0) {
    for (int i = 1; i < minTime + 1; i++) {
        // multiplying by 10 allows me to store both job and operation indicators
        arrFjssp[saveMachine - 1][arrLastPosition[saveMachine - 1][1] + i] = (jobCounter * 10) + counterOp;
        lastMachineTime++;
    }
    previousMachine = saveMachine;
    arrLastPosition[saveMachine - 1][1] = lastMachineTime + 1;
    lastMachineTime = 0;
}
```

Figura 30 – Primeira vez que a máquina foi usada

Se for a segunda vez que a máquina foi escolhida executa um código ligeiramente diferente, onde o array de posições é atualizado com a posição já guardada mais a posição nova.

Esse processo permite evitar o reescrever dos dados nas posições já ocupadas.

```
else {
    for (int i = 0; i < minTime; i++) {
        // multiplying by 10 allows me to store both job and operation indicators
        arrFjssp[saveMachine - 1][arrLastPosition[saveMachine - 1][1] + i] = (jobCounter * 10) + counterOp;
        lastMachineTime++;
    }
    previousMachine = saveMachine;
    arrLastPosition[saveMachine - 1][1] = lastMachineTime + arrLastPosition[saveMachine - 1][1];
    lastMachineTime = 0;
}
```

Figura 31 - Segunda vez que a máquina foi usada

Ao mudar de operação, a variável contadora de operações irá tornar uma condição verdadeira (na segunda linha da imagem).

Este processo permite atualizar a referência (inicialmente 0 quando não há valores) para uma nova, que é indicada pela última máquina escolhida. Sendo assim, impossibilitamos que o programa coloque os jobs a realizar mais que uma operação ao mesmo tempo.

```
if (block == 0) {
    if (counterOp > 1) { // only true on the second operation and so on, but only on the first
        // this will update the last position array, to define a new
        // reference / prevents the program from doing more than one operation at a time
        for (int i = 0; i < 8; i++) {
            saveValue = arrLastPosition[i][1];
            if (previousValue == 0) {
                previousValue = saveValue;
            }
            else if (previousValue < saveValue) {
                previousValue = saveValue;
            }
        }
        for (int i = 0; i < 8; i++) { // update all the positions with the new reference
            // the new reference is equal to the last position of the last machine used
            arrLastPosition[i][1] = previousValue;
        }
        block = 1; // blocks the entry until a new operation is being done
    }
}
```

Figura 32 – Atualização da referência no array com as últimas posições

## 6.2. Ficheiro FJSSP

Recorro a um ficheiro `.txt` possibilitando uma interpretação intuitiva da proposta de escalonamento.

O ficheiro está organizado á semelhança do *array*, constituído pelas 8 máquinas e 100 unidades de tempo máximas.

Em cada linha é escrito todas as informações referentes ao que está guardado *array*, conforme a quantidade de unidades de tempo de operação.

No exemplo da máquina 4 (representado na caixa amarela), temos 4 vezes (tempo de operação é 4) o job e operação onde a máquina foi escolhida.

	T75	T76	T77	T78	T79	T80	T81
	...	...	...	...	...	...	...
	...	...	...	...	...	...	...
	...	...	...	...	...	...	...
	...	J20p7	J20p7	J20p7	J20p7	...	...
	...	...	...	...	...	...	...
	...	...	...	...	...	...	...
	...	...	...	...	...	...	...
96	J20p6	...	...	...	...	...	...

Figura 33 – Ficheiro .txt com a proposta de escalonamento



## 7. Testes Realizados

Para realizar o processo de construção da proposta de escalonamento, é necessário selecionar a respetiva opção no menu.

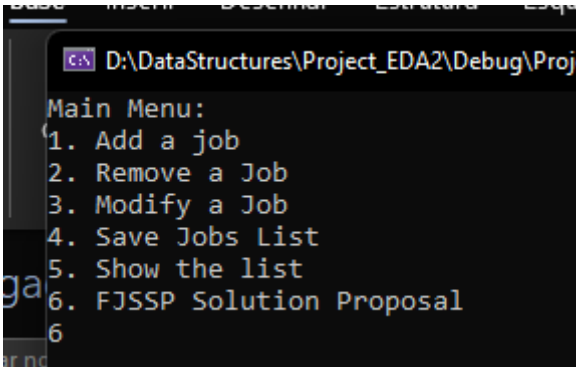


Figura 34 – Menu jobs

Após isso a função *fjssp()* irá ser chamada, enviando como parâmetro a cabeça da lista de *jobs*.

Aquando da finalização da criação da lista, uma mensagem será apresentada na consola, confirmando que o ficheiro foi atualizado conforme o *Process Plan* atual.

Conforme a tabela apresentada no enunciado do trabalho prático, apresento uma solução que concluirá esse *Process Plan* em 80 unidades de tempo.

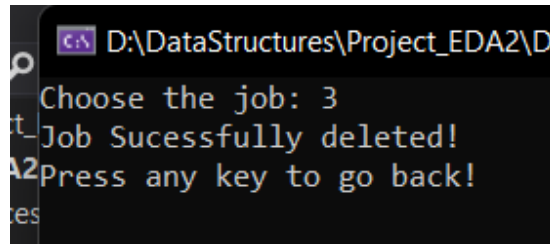
T76	T77	T78	T79	T80	T81	
...	...	...	...	...	...	07
...	...	...	...	...	...	(4)
...	...	...	...	...	...	[4]
...	...	...	...	...	...	
...	J20p7	J20p7	J20p7	J20p7	...	
...	...	...	...	...	...	
...	...	...	...	...	...	
...	...	...	...	...	...	
J20p6	...	...	...	...	...	

Figura 35 – Process Plan base

## 7.1. Testes com outros Process Plan

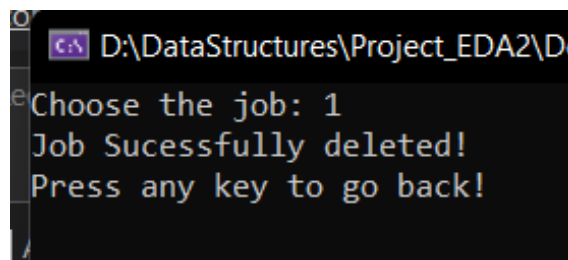
Neste exemplo de Process Plan, procedi á eliminação do job 1 e 3, sendo a lista atualizada conforme.

**Atenção:** O job 3 e 1 são os iniciais, por isso é necessário apagar primeiro o job 3 e depois o job 1, caso contrário, o job 3, seria o job 4 inicial, visto que a lista reorganiza ao eliminar.



```
C:\> D:\DataStructures\Project_EDA2\De
Choose the job: 3
Job Sucessfully deleted!
Press any key to go back!
```

Figura 36 – Eliminação do job 3



```
C:\> D:\DataStructures\Project_EDA2\De
Choose the job: 1
Job Sucessfully deleted!
Press any key to go back!
```

Figura 37 - Eliminação do job 1

Após a eliminação, gerei o novo *Process Plan*.

O *Process Plan* encontra-se na pasta do mesmo nome, no ficheiro .txt ProcessPlan2.

Ao eliminar o job 1 e 3, os jobs abaixo irão mudar de posição duas vezes (menos o job 2).

Verifiquei que o job 2 assumiu a posição de job 1, e a máquina 1 da operação 1 foi usada, com 1 tempo de operação. Verifiquei também que o job 4 ocupa a posição de job 2 no novo plano, sendo a máquina 3 escolhida com 3 unidades de tempo de operação.

	T1	T2	T3	T4	T5	T6	T7
1							
2							
3	M1	J10p1	J30p1	J30p1	J30p1	J40p1	J40p1
4							
5	M2	...	...	...	...	...	...
6							
7	M3	J20p1	J20p1	J20p1	...	...	...
8							
9	M4	...	...	...	...	...	...
10							
11	M5	...	...	...	...	...	...
12							
13	M6	J50p1	J50p1	J50p1	J50p1	...	...
14							
15	M7	...	...	...	...	...	...
16							
17	M8	...	...	...	...	...	...

		(2,4)	(3,5)	(4,5,6,7,8)			
pr <sub>1,2</sub>		[1,3]	[5,6]	[5,6,7,8]			
pr <sub>2,2</sub>	(1,3,5)	(4,8)	(4,6)	(4,7,8)	(4,6)	(1,6,8)	(4)
	[1,5,7]	[5,4]	[1,6]	[4,4,7]	[1,2]	[5,6,4]	[4]
pr <sub>3,3</sub>		(4,8)	(3,5,7)	(4,6)			
		[7,7]	[7,8,7]	[7,8]			
pr <sub>4,2</sub>	(1,3,5)	(2,8)	(3,4,6,7)	(5,6,8)			
	[4,3,7]	[4,4]	[4,5,6,7]	[3,5,5]			
pr <sub>5,1</sub>	(1)	(2,4)	(3,8)	(5,6,8)	(4,6)		
	[3]	[4,5]	[4,4]	[3,3,3]	[5,4]		
pr <sub>6,1</sub>	(1,2,3)	(4,5)	(3,6)				
	[3,5,6]	[7,8]	[9,8]				
pr <sub>7,2</sub>	(3,5,6)	(4,7,8)	(1,3,4,5)	(4,6,8) [4,6,5]	(1,3)		
	[4,5,4]	[4,6,4]	[3,3,4,5]		[3,3]		
pr <sub>8,1</sub>	(1,2,6)	(4,5,8)	(3,7) [4,5]	(4,6) [4,6]	(7,8)		
	[3,4,4]	[6,5,4]			[1,2]		

Este exemplo concluirá o *Process Plan* em 57 unidades de tempo.

Neste outro exemplo de Process Plan, repeti o processo de eliminação do job 3 e 1, porém retirei a máquina 1 do novo job 1 (a máquina escolhida no exemplo anterior), e também eliminei por completo a operação delineada a roxo no exemplo anterior.

Para eliminar essa operação, implementei duas soluções diferentes (fase1). Uma delas é eliminar diretamente o *node* correspondente à operação, através do processo de eliminação na lista ligada, e outro é eliminar todas as máquinas constituintes, resultando também na eliminação da operação.

	T1	T2	T3	T4	T5	T6	T7	T8
M1	J30p1	J30p1	J30p1	J40p1	J40p1	J40p1	J60p1	J60p1
M2	...	...	...	...	...	...	...	...
M3	J10p1	J10p1	J10p1	J10p1	J10p1	...	...	...
M4	...	...	...	...	...	...	...	...
M5	...	...	...	...	...	...	...	...
M6	J50p1	J50p1	J50p1	J50p1	...	...	...	...
M7	...	...	...	...	...	...	...	...
M8	J20p1	J20p1	J20p1	J20p1	...	...	...	...

pf <sub>1,2</sub>	(2,4)	(3,5)	(4,5,6,7,8)				
pf <sub>2,2</sub>	(3,5) (5,7)	(4,8)	(4,6)	(4,7,8)	(4,6)	(1,6,8)	(4)
pf <sub>3,3</sub>	(4,8)	(3,5,7)	(4,6)				
pf <sub>4,4</sub>	(2,8)	(3,4,6,7)	(5,6,8)				
pf <sub>5,5</sub>	(4,3,7)	(4,4)	(4,5,6,7)	(3,5,5)			
pf <sub>6,6</sub>	(1)	(2,4)	(3,8)	(5,6,8)	(4,6)		
	(3)	(4,5)	(4,4)	(3,3,3)	(5,4)		
pf <sub>7,7</sub>	(1,2,3)	(4,5)	(3,6)				
	(3,5,6)	(7,8)	(9,8)				
pf <sub>7,2</sub>	(3,5,6)	(4,7,8)	(1,3,4,5)	(4,6,8) [4,6,5]	(1,3)		
	(4,5,4)	(4,6,4)	(3,3,4,5)		(3,3)		
pf <sub>8,1</sub>	(1,2,6)	(4,5,8)	(3,7) [4,5]	(4,6) [4,6]	(7,8)		
	(3,4,4)	(6,5,4)			(1,2)		

Ao eliminar uma operação, a lista de operações está implementada de forma a reorganizar-se, conforme o que está indicado na imagem. Como se pode verificar a máquina 1 já não é opção, por isso a máquina com o menor tempo que agora é a máquina 3, foi escolhida. O job 4 que passou para job 2, agora tem como operação 1 a antiga operação 2, sendo a máquina 8 escolhida com 4 tempos de operação. O *Process Plan* termina após 49 unidades de tempo.

## Conclusão

O trabalho prático foi bastante enriquecedor, visto que possibilitou colocar em prática os conhecimentos obtidos ao longo das aulas da unidade curricular de Estruturas de Dados Avançadas, e também consolidar os conhecimentos adquiridos em Programação Imperativa.

Fui capaz de aprimorar as minhas capacidades já adquiridas ao nível da programação em linguagem C de modo geral. Clarifiquei certos pontos, como por exemplo apontadores, e consegui chegar mais além, colocando em prática métodos de criação de listas dinâmicas de dados e alocação de memória.

Devido a falta de tempo, não fui capaz de implementar uma solução mais eficiente para a redução do *makespan*, no entanto, apresento a seguinte proposta como opção:

- Um *array* que guarda as utilizações das máquinas, e sempre que tem de escolher uma, percorre o *array* verificando as menos utilizadas, entre as que são constituintes da operação que se encontra sobre avaliação no momento.

# Bibliografia

Link repositório GitHub:

- Fase 1:

[https://github.com/sampaio00joao/Project\\_EDA\\_18611](https://github.com/sampaio00joao/Project_EDA_18611) - atualizado pela última vez a 18/04/2022

- Fase 2:

[https://github.com/sampaio00joao/Project\\_EDA\\_Fase2](https://github.com/sampaio00joao/Project_EDA_Fase2) - atualizado pela última vez a 04/06/2022