Relatório

Algoritmos e Estruturas de Dados 2

Aluno/os:

Henrique Monteiro Cartucho Nº21122
Roberto Filipe Manso Barreto Nº21123

Professores:

Alberto Simões

Óscar Ribeiro

Licenciatura em Engenharia de Sistemas Informáticos

Barcelos, abril, 2021

Índice

1. Introdução

1.1. Estrutura do documento

Este documento refere-se à descrição e documentação do trabalho prático realizado pelos alunos Roberto Filipe Manso Barreto e Henrique Monteiro Cartucho da turma de licenciatura de engenharia de sistemas informáticos. Este documento contém a descrição do pensamento e do desenvolvimento do problema e dificuldades do mesmo.

2. Resumo

O projeto desenvolvido consiste em um programa capaz de ler um conjunto de ficheiros que contêm informações sobre um DataSet de Lego, informações estas que estavam contidas em ficheiros do tipo tsv. O objetivo do trabalho é ler a informação, tratar e apresentar ao utilizador dando opções para o utilizador realizar no DataSet.

Para esta realização o desenvolvimento foi dividido em 3 grandes listas, a lista de sets(conjuntos), a lista de parts(peças) e a lista de relações (lista que contém as relações entre conjuntos e as peças dos mesmos), pois assim os 3 ficheiros ficam guardados em 3 listas diferentes para se conseguir manipular e se aceder aos dados.

Para além da divisão anteriormente mencionada, o código realizado foi também dividido em 4 secções levando assim a uma melhor organização do código, estas secções seriam todas as ações ligadas a parts divididas em 3 ficheiros que estão organizados dentro de um pasta, estes ficheiros contêm manipulação de listas, listagem das mesmas e pesquisa nas mesmas, o mesmo acontece para os sets e as relações, já para operações gerais de listas existem outros ficheiros de manipulação e definição de listas, assim como também o ficheiro que contém o código que efetua a leitura dos ficheiros e coloca as informações dos ficheiros nas listas.

3. Organização do trabalho em grupo

Para a organização do trabalho em grupo e para evitar problemas futuros, o grupo decidiu organizar o trabalho de forma a evitar problemas tais como perder trabalho, ficheiros corrompidos, etc….

Evitando então estes problemas o grupo criou um repositório no github permitindo assim ter o seu trabalho guardado sem ser em seus dispositivos, evitando também que quando uma alteração grande seja realizada a mesma não coloque em risco o restante do trabalho permitindo assim criar ramos diferentes não influenciando o ramo principal, outra grande valia desta ferramenta é a possibilidade de voltar com o próprio trabalho para trás no tempo acedendo ao trabalho colocado em vários pontos do tempo anteriores.

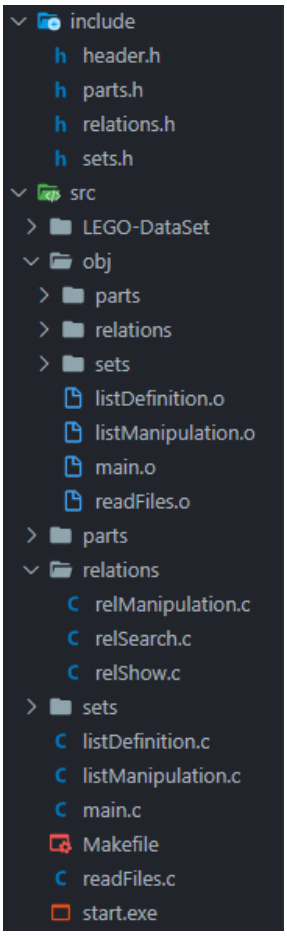
Para por fim organizar o relatório o grupo decidiu fazer uso da ferramenta google docs para assim ter o seu trabalho sempre guardado na cloud não correndo riscos de o perder, a principal mais valia do uso desta ferramenta é a possibilidade de os membros do grupo terem a possibilidade de ao mesmo tempo estar a trabalhar em regiões diferentes do relatório.

4. Desenvolvimento do trabalho

Para o desenvolvimento do trabalho foi primeiramente necessário perceber o que é um ficheiro .tsv foi então que se percebeu que um ficheiro .tsv é um ficheiro que tem os dados todos separados por tabs, logo foi então percebido como teria de ser lido um ficheiro, logo depois foi pensado que tipo de estrutura de dados utilizado, foi então que se decidiu utilizar listas simplesmente ligadas.

4.1. Makefile

Para auxiliar a organização do trabalho foi então criado um *makefile*, este *makefile* obriga a que exista uma estrutura específica de projeto. Esta estrutura seria, existem duas pastas principais, *include* e *src*, a pasta *include* contém todos os *headers* do programa e a pasta *src* é a pasta que contém os ficheiros com o código principal, dentro da mesma existe a pasta que guardar todos os ficheiros .o , a pasta dos ficheiros de dados e as pastas com os ficheiros das secções divididas.



4.2. Listas Criadas

Para guardar as informações a ser lidas dos ficheiros foram criadas 3 listas duplamente ligadas, a lista de PARTS que conteria as informações sobre as peças, informações essas como *part_num*, *name*, *class* e stock, já para conter as informações sobre sets foi criada a lista de SETS que contém as informações *set_num*, *name year* e *theme*, por fim para conter as informações de todas as peças necessárias para sets foi criada a lista RELATIONS pois na verdade seria a lista das relações entre SETS e PARTS, esta lista contém as informações de *set_num, part_num* e *quantity*.

4.3. Leitura dos Ficheiros

Para efetuar a leitura dos ficheiros foi preciso primeiramente perceber que *tsv* significa *tab separated values,* ou seja valores separados por *tabs*, este tipo de ficheiro facilita a leitura dos valores pois basta utilizar o caracter '\t' para efetuar ou ler um *tab*, foi então com essa ideia em mente que foram criadas as funções de leitura de ficheiros, nestas funções inicialmente são declaradas as variáveis que vão guardar as informações obtidas de cada linha do ficheiro e também o apontador do tipo FILE, de seguida é realizado a abertura do ficheiro dentro de um *if* para assim ser verificado se o ficheiro existe pois desta forma é possível avisar o utilizador caso o ficheiro não exista ou não seja encontrado, dentro do *if* é primeiro informado ao utilizador que se está a buscar as informações, de seguida é efetuado um *fseek* pois no começo de cada ficheiro existe um cabeçalho que indica o que é cada variável, com a função *fseek* foi possível passar a frente o numero de caracteres utilizando *nº de caractectes * sizeof(char)* a partir do começo do ficheiro, após ser saltado o cabeçalho é lida cada linha do ficheiro, para ler cada linha do ficheiro é utilizado o caracter '%[^\t]' para assim ler todos os caracteres até se encontrar um *tab* não incluindo o *tab* no valor lido e de seguida é possível efetuar um *tab* para ler o seguinte valor efetuando por exemplo '%[^\t]\t%[^\t]' seria possível ler dois valores separados por *tab*. Após a obtenção dos valores seria necessário inserir os mesmos e foram para isso criadas também funções de inserção para cada lista do programa, estas funções recebiam por parâmetro a lista na qual vai ser inserido o valor e os valores a inserir, na função é alocada memória para o novo node a ser criado e copiado para o mesmo todas as informações recebidas por parâmetro, de seguida é verificado se existe lista, se existir então o valor *next* do node seria a lista, caso contrário seria *null* porque seria o primeiro valor da lista, depois é verificado se existe valor seguinte, caso exista então a variável anterior do *node* seguinte seria igualada ao node a inserir, de seguida o node anterior ao node a inserir seria *null* pois não existe e seria retornado o node pois este já contém a ligação à lista, já na função de ler o ficheiro

a lista é igualada a esta função e por fim a abertura do ficheiro é fechada e a lista nova é retornada e no *main* igualada á respetiva lista, esta seria então a base de todas as funções de inserção nas listas e de leitura dos 3 ficheiros.

4.4. Realização dos exercícios

4.4.1. Quais os conjuntos de determinado tema (ordenados pelo ano)

Este exercício pedia que se obtenha os conjuntos de um determinado tema e que se ordene por ano os mesmos. Este problema foi então dividido em duas fases, primeiramente era necessário obter todos os conjuntos de um determinado tema, para isso foi criada a função *SearchSetbyTheme* esta função primeiramente cria uma nova lista *search* que vai ser utilizada para guardar todos os sets do tema pesquisado, depois é efetuado um ciclo for que itera pela lista de sets e compara o tema de cada com o tema a pesquisar com uma condição *if* utilizando também a função *LowerString* que vai comparar as strings em letras minúsculas assim não havendo problemas com esses tipos de escrita, sempre que esta condição se verifica, a mesma utilizando a função *InsertSet* insere o set verificado na lista de sets *search,* conseguindo assim uma lista de sets com o tema pesquisado pelo utilizador.

Para agora ordenar a lista foi primeiramente percebido em que consistia ordenar uma lista, para isto foi entendido primeiramente que seria necessário um ciclo que percorra a lista, para isso foi criado um ciclo *while* que verifica se existe algum set no set seguinte, sempre que o ano do set atual seja maior que o ano do set seguinte seria necessário efetuar a troca de lugar, para efetuar a troca foi criada uma função *swap* que recebe dois valores *a* e b e que troca de lugar um com o outro, por fim é passado para o set seguinte. Após isso foi reparado que mesmo assim a lista por vezes não ficava ordenada, então por isso foi criada a variável *swapped,* esta variável vai alterar entre 0 e 1, a função desta é avisar se algum numero foi alterado ou não, pois assim era possível criar um ciclo do while para sempre verifica de novo a lista enquanto algo for trocado conseguindo assim saber que pode ter mais algo para verificar, pois se a lista foi percorrida e nada foi alterado é porque a lista já se encontra ordenada.

Para a função *swap* são primeiramente declaradas variáveis para guardar as informações de um set, depois as informações do set a são passadas para as mesmas e o set a recebe as informações de b, depois o set b recebe as informações guardadas anteriormente do set a, conseguindo assim trocar as posições.

Para obter informações do tema a pesquisar é então pedido o mesmo ao utilizador e utilizando a função *ExistsTheme* verificado se o tema existe para pesquisar

4.4.2. As peças de determinado tipo em determinado conjunto

Neste exercício é pedido que se procure por peças de um determinado tipo em determinado conjunto, para isso foi então pensado primeiramente em obter as peças que pertencem ao conjunto, para isso seria necessário pesquisar na lista de relações por todas as relações associadas ao set pretendido e depois de obter todas essas relações seria necessário nas mesmas obter o *part_num* e pesquisar por esse mesmo *part_num* dentro da lista de *parts* e obter a *class* da peça associada a esse *part_num*, caso essa class seja a class que se procura então a *part* teria de ser adicionada a uma lista de *parts* que seria por fim retornada com todas as peças do tipo pesquisado.

Para colocar este pensamento em prática foi então primeiramente realizado o pedido da class pretendida ao utilizador e através da função *ExistsClass* é verificado se a class existe ou não, após isso é pedido o *set_num* a pesquisar, para isso é também utilizada a função *ExistsSet,* como já te tinha todas as informações necessárias foi então primeiro criada a função *SearchRelations.*

A função *SearchRelations* recebe por parâmetro a lista de relações de o *set_num* a pesquisar. Primeiramente é então declarada a variável *search,* variável esta cujo objetivo seria guardar em uma lista de pesquisa as relações encontradas, após a declaração é realizado um ciclo *for* que percorre a lista de relações verificando com uma condição *if,* se o *set_num* que se pesquisa é igual ao *set_num* que se está atualmente na lista, caso esta condição se verifique é então inserido na lista de pesquisa a relação da lista utilizando a função *InsertRelation,* por fim esta função retorna a lista de pesquisa.

Por fim era necessário realizar a função que percorre a lista devolvida pela função anteriormente falada e verificar peça a peça se é da *class* procurada, para isso foi então criada a função *PartsSearchByClassAndSet,* esta função recebe como parâmetro a lista de parts, a lista de relações obtida da função falada anteriormente e a *class* que o utilizador pretende pesquisar, primeiramente são declaradas duas variáveis, a search que é a lista que vai receber todas as partes com a *class* procurada e a variável *partSearch* que vai receber cada peça encontrada para ser comparada. Após a declaração das variáveis é efetuado um ciclo for que itera pela lista de relações, a cada iteração é pesquisado pela peça que tem o *part_num* da relação atual e a *class* que o utilizador pretende utilizando a função *SearchPartsByNumClass* que retorna a peça pesquisada e a mesma é guardada na variável *partSearch* que após ser verificada que realmente contém algo a mesma é inserida na lista search, por fim a mesma é retornada e no menu é apresentada ao utilizador por meio da função *ListParts*.

4.4.3. Quais as peças necessárias para construir um dado conjunto, indicando os dados de cada peça e respetiva quantidade

Este exercício pedia pelas peças necessárias para construir um dado conjunto indicando os dados de cada peça e a quantidade utilizada da mesma, para este exercício seria necessário então primeiramente obter a lista de relações com o *set_num* e de seguida pesquisar pelas *parts* existentes na lista de relações e listar as mesmas.

Para realizar este pensamento, foi então pedido ao utilizador o *set_num* a pesquisar, depois seria então pesquisado pelas relações do set_num, para isso foi então reutilizada a função *SearchRelations* cuja lista retornada é guardada numa variável, após essa função ser chamada foi então criada a função *PartsSearchBySet,* nesta função inicialmente são declaradas duas variáveis que são listas de parts, a lista search que vai guardar todas as partes encontradas na lista de relações e a lista *partSearch* que vai guardar cada peça pesquisada. Após a declaração de variáveis é realizado um ciclo for que itera pela lista de relações e chama a função *SearchPartsByNum* (explicada no parágrafo seguinte) passando por parâmetro a lista de parts e o part_num da relação atual, a mesma retorna a peça que se procura guardando-a na variável *partSearch* para depois ser inserida na lista search que vai depois por fim ser retornada para o menu, guardada em uma variável e listada para o utilizador através da função *ListPartsAndRelations*. A função *ListPartsAndRelations* recebe por parametro a lista de parts e de relações obtidas atrás, após isso é criada uma variável auxiliar que recebe a lista de parts e itera pela mesma utilizando um ciclo for, a cada iteração é realizado outro ciclo for que itera pela lista de *parts* e verifica a cada iteração se o part_num da *part* atual é o mesmo da relação e assim é realizado um print com as informações dos dois.

A função *SearchPartsByNum* recebe por parâmetro a lista de parts e o part_num a pesquisar, esta função primeiramente declara duas listas, a lista aux que recebe a lista parts e serve como auxiliar à lista parts e a lista *auxSearch* cujo objetivo é receber a part encontrada, de seguida à declaração das variáveis é realizado um ciclo for que itera pela lista auxiliar verificando se a *part_num* atual da lista é igual à *part_num* que se pretende pesquisar, caso seja a mesma peça é inserida na variável *auxSearch* retornada para assim não iterar pelo resto do ciclo desnecessariamente.

4.4.4. O total de peças em stock

Este exercício pedia para apresentar ao utilizador o total de peças em stock, para efetuar a contagem de peças em stock seria necessário somar o stock de cada peça obtendo assim o stock total.

Para colocar então em prática foi criada a função *StockParts* que recebe a lista de *parts*, nesta função é então inicializada uma variável do tipo inteiro chamada *counter* cujo objetivo é guardar o somatório do stock das peças, após a declaração desta variável é então criado um ciclo for cujo objetivo é iterar pela lista de *parts* e somar ao *counter* o stock de cada *part* a cada iteração retornando no fim do ciclo a variável *counter*, conseguindo assim através de um print mostrar o somatório do stock das peças ou seja o total de peças em stock.

4.4.5. O total de peças incluídas num determinado conjunto

Este exercício pede que se apresente ao utilizador a quantidade de peças incluídas em um determinado conjunto. Refletindo então sobre a questão, foi chegado à conclusão que o necessário seria obter na lista de relações, todas as relações de um certo conjunto e de seguida efetuar o somatório das quantidades das mesmas.

Para então aplicar a reflexão foi primeiramente pedido ao utilizador o *set_num* que deseja procurar, após essa obtenção que é verificada com uma função *ExistsSet* é então chamada a função *SearchRelations* atrás abordada e passado como parâmetros a lista de relações e o set_num que o utilizador deseja pesquisar, a lista de relações deste set devolvida pela função é então guardada em uma variável, variável essa que é utilizada para passar por parâmetro para a função *SetPartsQuantity*.

A função *SetPartsQuantity* primeiramente inicializa a variável counter do tipo inteiro a 0 e de seguida efetua um ciclo que *for* que itera pela lista de relações recebida e soma à variável counter a quantidade de cada peça que é necessária para construir o conjunto, por fim a variável counter é retornada e é apresentada ao utilizador através de um print à própria função.

4.4.6. A peça que é utilizada em mais conjuntos diferentes, independentemente da quantidade em cada um deles

Neste exercício é pedido para encontrar a peça mais utilizada em diferentes conjuntos, para isso foi pensado em vários algoritmos, desde percorrer a lista de peças e contar o numero de vezes que cada uma aparece na lista de relações e descobrir qual aparece mais vezes, até percorrer a lista de relações varias vezes ou seja percorrer o numero de peças ao quadrado, primeiramente esta função demorava horas a encontrar a peça, depois cerca de 20 minutos, de seguida através de alterar de forma recursiva para forma iterativa foi conseguido 5minutos mas mesmo assim este tempo seria tempo demais ainda para a quantidade de dados que se possui. Foi então que em uma aula da disciplina foi visualizado como o professor realizava, foi então que surgiu o a ideia de percorrer a lista de relações e a cada iteração guardar a *part_num* em uma lista criada para realizar a contagem na qual quando inserida uma nova *part* verifica se existe a parte e caso exista acrescenta +1 ao contador da peça, caso não exista cria mais um node na lista para colocar a peça nova, após se iterar pela lista de relações se se obter todas as contagens teria de ser verificado na lista de contagens qual é a peça com a contagem mais alta e seria então retornada essa peça para o utilizador.

Aprofundando então o ultimo algoritmo mencionado foi primeiramente criada a lista simplesmente ligada que iria guardar a contagem das peças, a lista *PARTCOUNTER*, esta lista contém com variáveis o *part_num* e o *counter* que guarda a contagem da peça, após isso foi criada a função *MoreUsedPart* que recebe por parâmetro a lista de relações, esta função inicialmente declara a lista de contagem de peças e depois através de um ciclo for itera pela lista de relações, a cada iteração é inserida na lista a peça da iteração através da função *InsertCounterSearch*. Esta função recebe como parâmetro a lista de contagem e o *part_num* a inserir, primeiramente a lista verifica se a posição atual não é nula ou seja se existe algo, caso não exista é então inserida a peça, pois apenas entra neste *if* quando for a primeira vez nesta função ou quando a peça a inserir não exista ainda na função logo acabaria no fim da função ou seja em posição com *null*, dentro desta condição é alocada memória para o novo node da lista e é copiado o *part_num* e o contador é inicializado a 1, como é um *head insert* o *next* seria a lista e a lista é igualada depois ao novo node, caso exista uma peça na posição, é então verificado se a *part_num* é a mesma que se pretende inserir, caso seja o contador dessa peça é incrementado por 1, caso não se verifique então é passado para a próxima posição da lista, por fim é retornada a lista. Após a inserção de todas as peças é chamada a função *MoreUsed* que recebe por parâmetro a lista de contagem, esta função primeiramente verifica se existe lista, caso exista é então criada uma variável do tipo *PARTCOUNTER* para guardar a peça com a contagem mais alta a qual inicialmente se torna a primeira peça da lista de contagem, de seguida é efetuado um

ciclo *while* para percorrer a lista verificando sempre se existe próximo na lista e que inicialmente passa a primeira posição à frente, no *while* é verificado a cada iteração se o contador atual é maior que o maior contador registado, caso seja essa *part_num* e contador são guardados na variável criada para guardar a peça com maior contagem, após o fim do ciclo é então criado um apontador para o *part_num* da peça com a maior contagem, é libertada a variável da peça com maior contagem e o apontador do *part_num* é retornado. Já na função *MoreUsedPart* é criado um apontador para o *part_num* retornado pela função *MoreUsed* o qual é retornado para o menu, no menu é pesquisada pela peça com o *part_num* obtido e as informações da mesma são listadas

4.4.7. A lista dos conjuntos que se conseguem construir com o stock existente

Neste exercício é pedido que se apresente todos os conjuntos que se conseguem construir, este exercício já teve várias abordagens mas nenhuma eficiente o suficiente pois demorava horas para listar todos os sets possíveis, o algoritmo inicial era então iterar pela lista de sets e verificar set a set se era possível construir o mesmo, após alguma revisão chegou-se à conclusão que o algoritmo mais eficiente que o anterior seria iterar pela lista de relações partindo sempre do principio que se pode construir todos os sets e verificar se a peça de cada relação tem stock suficiente para a quantidade necessária da peça, caso não tenha é então verificado que o set não se pode construir e para inserir o set na lista de sets teria de se verificar a cada iteração se o set atual é diferente do set anterior pois a lista de relações possui as relações todas de um set seguidas logo basta detetar quando se muda de set e se o set anterior se pode construir então o mesmo teria de ser inserido na lista de sets que é possível construir.

Aplicando então este pensamento a lógica foi criada a função *SearchSetCanBuild* que recebe por parâmetro a lista de sets, a lista de relações e a lista de *parts*, primeiramente são declaradas variáveis para a pesquisa de sets que se podem construir, *search*, para a pesquisa de sets individuais, set e para pesquisa de *parts* individuais, *part*, foi também criada a variável *canBuild* que é do tipo *bool* e é igualada a true pois parte-se sempre do principio que se pode construir um *set*. Após a declaração de variáveis é realizado um ciclo for que itera pela lista de sets, verificando primeiramente se existe um set anterior pois se for a primeira relação da lista não se pode verificar se é diferente do anterior, caso exista anterior é então verificado se o *set_num* da atual é diferente do anterior verificando então se se trocou de set, caso seja diferente é porque todas as relações do set anterior já foram passadas, é então verificado se a variável *canBuild* está a *true* significando que se pode construir o set anterior, caso esteja é pesquisado pelo set e o mesmo é inserido na lista de sets que se podem construir, depois fora do *if* é colocado a variável *canBuild* a *true* caso a mesma

esteja a false pois sempre que se inicia um set novo tem que se partir do principio que o set se pode construir, seguindo então para a verificação é verificado se a variável *canBuild* está a *true* pois não é necessário verificar as peças se o set já não se pode construir conseguindo assim poupar algum tempo de pesquisa de peças. Caso se possa construir o set ainda, é então primeiramente pesquisado pela peça, de seguida é verificado se a peça foi encontrada e caso tenha sido encontrada é verificado se o stock da peça é inferior à quantidade necessária, pois caso seja, não é possível construir o set tornando assim a variável *canBuild* false, caso a peça não seja encontrada quer dizer que o utilizador não possui a peça logo também não se pode construir tornando então a variável *canBuild* false também, ao fim do ciclo por completo é retornada a lista de pesquisa de todos os sets que podem ser construídos, a lista retornada é então no menu listada para o utilizador visualizar todos os sets que é possível construir.

4.4.8. Alterar o número de peças em stock

Neste exercício é pedido que se altere o número de peças em stock, ou seja com base em um identificador de peça alterar o stock da mesma, sendo que um número negativo retira ao stock e um número positivo adiciona ao stock. Aplicando a pensamento logico o necessário seria procurar a peça em questão na lista de peças e assim que se encontrar alterar o stock da mesma com base no stock que o utilizador deseja.

Para aplicar então a código foi primeiramente pedido ao utilizador o *part_num* verificando se o mesmo existe através da função *ExistsPart* e de seguida foi pedido a quantia a alterar, para alterar o stock foi então criada a função *EditPartsStock,* esta função recebe por parâmetro a lista de peças, o *part_num* da peça a procurar e a quantidade de stock a colocar, para procurar a peça é realizado um ciclo for que itera pela lista de peças e a cada iteração é verificado através de um *if* se o *part_num* da peça atual é o *part_num* que se pretende através de um *strcmp*, caso se verifique a condição significa então que a peça a alterar foi encontrada e é então testado através de um *if* se ao adicionar a quantia desejada pelo utilizador o stock fica negativo, caso o stock não fique negativo a quantia é então adicionada, caso contrário o utilizador é avisado que não é possível efetuar a alteração porque o stock ficaria negativo.

4.4.9. A adição de stock com base no identificador de um conjunto

Neste exercício é pedido que se adicione ao stock com base em um identificador de conjunto, ou seja quando o utilizador introduz um *set_num* adicionar ao stock as peças que o mesmo necessita e as respetivas quantidades, pois é como se o utilizador tivesse comprado um set que já teria e quisesse adicionar as peças do mesmo ao stock. Inicialmente o exercício foi entendido de forma errada, pois tinha sido entendido que o utilizador poderia realmente comprar o set e adicionar as peças do mesmo mas para além de poder comprar um set repetido também poderia comprar um set novo, foi então que foi então realizado o seguinte pensamento, seria necessário perguntar ao utilizador o *set_num* e verificar se o mesmo já estava registado, caso este já estivesse registado, seria adicionado ao stock das peças, as peças do set com base na lista de relações do mesmo, caso contrário as informações do set seriam pedidas e um novo set seria inserido na lista de sets, após isso seria necessário pedir as peças do set caso o utilizador quisesse inserir um set novo, ou seja agora é pedido ao utilizador um *part_num* ao utilizador e verificado se o *part_num* existe, caso exista é então perguntado ao utilizador se pretende adicionar a peça que já existe e caso deseje então é pedido ao utilizador a quantidade da peça que é necessária, caso o utilizador coloque um *part_num* que não existe é pedido ao utilizador as informações da peça e a mesma é adicionada á lista de peças, por fim estas peças e os sets são inseridos na lista de relações, mas visto que o set pode ter varias peças é então necessário fazer um ciclo que sempre no fim do mesmo perguntar se o utilizador deseja adicionar mais peças ou não sendo esta a condição de saída do ciclo, por fim teria de ser pesquisado as relações do set que foi inserido e depois listadas as informações das relações e dos sets.

Para colocar a reflexão atrás falada em prática é pedido as informações do set e lidas para variáveis, a variável de *set_num* é testada com a função *ExistsSet* e caso exista já um set com o *set_num* colocado é perguntado ao utilizador se deseja adicionar ou não o set, com isso caso deseje adicionar é pesquisado pelas relações e com um ciclo for é iterado pela lista devolvida pela função atrás mencionada e utilizando a função *EditPartStock* é alterado o stock das *parts* de acordo com a quantidade que o set necessita de cada peça verificada a cada iteração. Caso o utilizador deseje adicionar um set novo são então pedidas as informações restantes do set e depois pedidas as peças, através de um ciclo *do while,* cuja a condição é o utilizador desejar inserir mais peças, é pedido o *part_num* e caso se verifique que existe a *part* utilizando a função *ExistsPart* é avisado que a *part* existe e se o utilizador pretende adicionar a mesma ou não, caso o utilizador deseje adicionar a mesma, é então pedido ao utilizador a quantidade da peça que o set contém, após isso é através da função *EditPartStock* é editado o stock da peça em questão adicionando ao stock a quantidade que o utilizador introduziu e posteriormente inserido na lista de relações a relação do set com a

peça introduzida. Caso o utilizador não deseje introduzir uma peça já existente é então pedido ao utilizador todas as informações sobre a peça e a mesma é inserida na lista de *parts* através da função *InsertPart* e depois introduzida também na lista de relações, a relação entre o set e a peça, após isso é perguntado se o utilizador deseja adicionar mais peças e caso sim o ciclo volta a repetir, caso não é então pesquisado pela relação e listado através da função *ListPartsAndRelations* a lista de *parts* associadas ao set em questão.

Para verificar se é necessário pedir mais informações ao utilizador é utilizada uma variável booleana que sempre que o utilizador insere algo já existente a mesma é tornada verdadeira e assim é possível verificar se o utilizador decidiu inserir algo já existente ou não

4.4.10. Remover todas as peças de determinada classe

Neste exercício é pedido para remover todas as peças de uma determinada classe, para isso seria necessário iterar pela lista de peças procurando por peças que tenham as classes em questão e removendo cada peça que tenha a classe em questão.

Para colocar então em prática é pedido ao utilizador a *class* que pretende remover, verificando que a *class* existe através da função *ExistsClass* que itera pela lista de *parts* verificando se a *part* tem como *class* a *class* desejada e caso encontre a mesma retorna verdadeiro, após as verificação é então prosseguido para a função *RemovePartsbyClass* que recebe por parâmetro a lista de *parts* e a *class* a pesquisar, primeiramente é declarada uma variável *auxliar* para a lista de *parts*, de seguida é criado um ciclo *while* cuja condição é existir lista na posição atual, dentro do mesmo foi então criado uma condição *if* que verifica se a *class* da *part* atual é a *class* que foi recebida por parâmetro, caso não seja é passado para a *part* seguinte, pois caso se elimine uma *part*, a *part* atual passa a ser a "*part* seguinte" tendo assim de esta ser verificada também, caso seja verdadeira a condição é verificado se existe *part* seguinte, caso exista a *part* anterior a parte seguinte seria a *part* anterior, de seguida é verificado se existe parte anterior, caso exista a *part* seguinte à parte anterior seria a parte seguinte à *part* atual fazendo assim as ligações entre a parte seguinte e a atual, caso esta ultima condição não se verifique é então igualada a lista de *parts* à *part* seguinte fazendo assim passar a frente a primeira *part* pois para entrar neste *else* seria necessário esta ser a primeira *part* da lista. Após isto é eliminada a *part* atual, para isso a lista auxiliar é igualada à função *DeletePartsNode* e é passado por parâmetro a lista auxiliar, esta função recebe então a *part* a eliminar, cria uma lista de *parts* e iguala a *part* recebida à lista e a *part* recebida é igualada à *part* seguinte da lista eliminando depois a *part* pretendida e retornando a lista de *parts* com a *part* eliminada. Já na função *RemovePartsbyClass* é no fim do ciclo retornada a lista de peças com as

peças eliminadas. As parts não são eliminadas da lista de relações pois o set continua a necessitar da peça, ou seja, as peças são apenas removidas da lista de peças e não da lista de relações.

4.4.11. Remover todos os sets de determinado tema

Neste exercício é pedido para remover todos os sets de um determinado tema, isto implica remover todos os sets da lista de sets que contenham o tema que o utilizador desejar e também da lista de relações.

Para colocar este pensamento em prática é pedido ao utilizador o tema a eliminar, é verificado se existe o tema utilizando a função *ExistsTheme* que itera pela lista de sets e assim q encontra um set com o tema a procurar o mesmo retorna *true* assim obtendo se o tema existe ou não, após a verificação de se o tema existe é chamada a função *RemoveSetsbyTheme* e passado por paramento a lista de sets, a lista de relações e o tema a procurar para remover, esta função é igual à função realizada no exercício anterior mas que agora procura por sets e por sets com temas iguais ao tema escolhido, já dentro da condição *if* ao fim de realizar as ligações todas é chamada a função *RemoveRelationsbyTheme* que é igual à função do exercício anterior mas adaptada para lista de relações e que procura por *set_num*, assim conseguindo remover todos os sets de um certo tema tanto da lista de sets como da lista de relações removendo então de todas as listas necessárias

5. Lista de complexidades de funções

- Parts
  - PARTS *InsertPart(PARTS *lst, const char *part_num, const char *name, const char *part_class, int stock) => $O(1)$
  - Int StockParts(PARTS *lsts) => $O(n)$
  - Int SetPartsQuantity(RELATIONS *lst) => $O(n)$
  - PARTS *DeletePartsNode(PARTS *node) => $O(1)$
  - PARTS *RemovePartsbyClass(PARTS *lst, const char *class) => $O(n)$
  - Void EditPartStock(PARTS *lst, char *part_num, int quantity) => $O(n)$
  - Void FreeParts(PARTS *lst) => $O(n)$
  - PARTS *SearchPartsByNumClass(PARTS *parts, char *part_num, char *class) => $O(n)$
  - PARTS *PartsSearchByClassAndSet(PARTS *parts, RELATIONS *rel, char *class) => $O(n \times m)$
  - PARTS *PartsSearchBySet(PARTS *parts, RELATIONS *rel) => $O(n \times m)$
  - PARTS *SearchPartsByNum(PARTS *parts, char *part_num) => $O(n)$
  - PARTS *SearchPartsByClass(PARTS *parts, char *class) => $O(n)$
  - bool ExistsPart(PARTS *lst, char *part_num) => $O(n)$
  - bool ExistsClass(PARTS *lst, char *class) => $O(n)$
  - void ListParts(PARTS *lst) => $O(n)$
  - void ListPartsAndRelations(PARTS *lst_parts, RELATIONS *lst_rel) => $O(n \times m)$

- Sets
    - SETS *InsertSets(SETS *lst, char *set_num, char *name, int year, char *theme) => $O(1)$
    - void OrderSetbyYear(SETS *lst) => $O(n^2)$
    - void swap(SETS *a, SETS *b) => $O(1)$
    - SETS *DeleteSetsNode(SETS *node) => $O(1)$
    - SETS *RemoveSetsbyTheme(SETS *lst, RELATIONS *rel_lst, const char *theme) => $O(n \times m)$
    - void FreeSets(SETS *lst) => $O(n)$
    - SETS *SetsSearchByNum(SETS *sets, char *set_num) => $O(n)$
    - SETS *SearchSetbyTheme(SETS *lst, char *theme) => $O(n)$
    - SETS *SearchSetCanBuild(SETS *sets_lst, RELATIONS *rel_lst, PARTS *parts_lst) => $O(n \times m \times o)$
    - bool ExistsSet(SETS *lst, char *set_num) => $O(n)$
    - bool ExistsTheme(SETS *lst, char *theme) => $O(n)$
    - void ListSets(SETS *lst) => $O(n)$
    - void ListSetsNTY(SETS *lst) => $O(n)$
- Relations
    - RELATIONS *InsertRelation(RELATIONS *lst, char *set_num, int quantity, char *part_num) => $O(1)$
    - RELATIONS *DeleteRelationsNode(RELATIONS *node) => $O(1)$
    - RELATIONS *RemoveRelationsbySet(RELATIONS *lst, const char *set_num) => $O(n)$
    - void FreeRelations(RELATIONS *lst) => $O(n)$
    - RELATIONS *SearchRelations(RELATIONS *rel, char *set_num) => $O(n)$
    - RELATIONS *SearchPartsRelations(RELATIONS *rel, char *part_num) => $O(n)$
    - PARTCOUNTER *InsertCounterSearch(PARTCOUNTER *lst, char *part_num) => $O(n)$
    - char *MoreUsed(PARTCOUNTER *lst) => $O(n)$
    - char *MoreUsedPart(RELATIONS *lst) => $O(n \times m^2)$
    - void ListRelations(RELATIONS *lst) => $O(n)$

- List Manipulation
    - void clean_stdin(void) => $O(n)$
    - char *LowerString(char *string) => $O(n)$
- Read Files
    - PARTS *OpenParts(PARTS *parts) => $O(n)$
    - SETS *OpenSets(SETS *sets) => $O(n)$
    - RELATIONS *OpenRelations(RELATIONS *relations) => $O(n)$

6. Conclusão

O desenvolvimento deste trabalho permitiu a consolidação dos conceitos abordados pela disciplina sobre o tema de listas em linguagem C, com este desenvolvimento foi também possível desenvolver métodos de pensamento diferentes para obter assim formas mais eficazes de realizar os métodos. Com este trabalho foi possível também explorar o como a diferença entre tipos de funções, recursiva e iterativa, e também a diferença entre os algoritmos diferentes podem afetar drasticamente a eficiência do programa. Foi possível também explorar ferramentas de máquinas virtuais para utilizar Linux para verificar a quantidade de memoria perdida e maltratada no programa. Estre trabalho permitiu também a exploração de graus de complexidade de funções, conceito que não teria sido abordado em programas anteriormente realizados que e permitiu uma suma para o aprendizado.

Em suma a consolidação dos conceitos e a exploração e visualização de áreas nunca exploradas antes permitiram uma evolução dos membros do grupo na área de programação conseguindo assim uma melhor qualidade de trabalho futuro.