Relatório LI3

19 de Abril de 2020

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Sistema de Gestão de Vendas

Laboratórios de Informática 3

2º Ano, 2º Semestre, 2019/2020

Grupo 74

Francisco Peixoto A84668

Renato Gomes A84696

Introdução

Este relatório aborda a resolução do projeto prático em C de LI3. O projeto consiste, resumidamente, em construir um sistema de gestão de vendas, respondendo a queries de forma eficiente, aplicando conhecimentos de algoritmia e programação imperativa.

Ao longo deste relatório são abordadas as decisões tomadas na implementação do projeto, nomeadamente as estruturas utilizadas para criar cada um dos módulos, a razão das escolhas tomadas e as suas APIs.

Inicialmente, foi necessário fazer a validação dos ficheiros e foram criados diferentes módulos. Estes módulos são: a validação dos ficheiros, as estruturas de cada tipo de dados e o código auxiliar para executar corretamente as queries fornecidas pelos professores, e finalmente, a interface do sistema.

Depois dos ficheiros serem carregados, somos capazes de executar uma lista de queries disponibilizadas pela equipa docente. Para responder às diferentes queries são utilizadas as funções definidas nas API dos diferentes módulos anteriormente referidos.

Analise do Enunciado

Como referido na introdução, este projeto consiste na construção de um sistema de gestão de vendas.No enunciado é referido que temos de ler e validar os ficheiros fornecidos de clientes, produtos e vendas,e assim mostrar-nos como gerir e trabalhar com grandes quantidades de dados, para tal teríamos de arranjar uma solução que o fizesse de forma eficiente e rápida.

Fomos também introduzidos a novos princípios de programação, como a modularidade e o encapsulamento de dados, extremamente importantes quando se utiliza tamanha quantidade de dados.

A estrutura base do projeto que nos foi atribuída consiste em módulos distintos, para o catalogo de produtos, catalogo de clientes, faturação, filial, e outro para a leitura dos ficheiros.

O objetivo final seria correr as queries dadas no enunciado, apresentando-as de uma forma agradável e completa, para tal teríamos de criar um controlador, para que o utilizador consiga controlar o *output* do programa.

Descrição do Projecto

1 Validação

O projeto foi inicializado com a validação dos ficheiros de texto fornecidos de modo a garantir que não existiam clientes ou produtos com o código errado, ou vendas com informação errada. A validação dos produtos e dos clientes foi feita de modo semelhantes uma vez que só diferiam de um carácter. Analisando e separando os caracteres dos números para facilitar a organização por ordem alfabética.

Para a validação das vendas recorremos à função *tokens()* de modo a separar a informação de uma venda pelos campos necessários à resolução do trabalho.

Os produtos, clientes e vendas validas são todas contabilizadas quando são validadas, sendo apresentadas na altura da sua leitura na query 1.

O nosso programa efetua esta leitura e validação ,com o ficheiro de 1M de vendas, em cerca de 5 segundos.

2 Estruturas

Durante o desenvolvimento deste projeto, optamos pela criação de algumas estruturas que nos auxiliam em vários pontos.

```
struct arvore{
  int valor;              // valor para se poder gerir
    void* info;             // info que realmente se guarda
  struct arvore *esq, *dir;
  int altura;
};
```

Esta struct arvore AVL é a base do nosso projeto. É a maneira como organizamos os dados, uma vez que chegamos a conclusão que é uma maneira eficiente e rápida de mais tarde procurar esses mesmos dados.

```
struct lista_strings{
    int in_use;
    int size;
    int free_space;
    char** lista;
};
```

Esta struct é como o nome indica uma lista de strings. Usamos-la para guardar varias strings, como por exemplo códigos de produtos ou clientes para mais tarde serem usados no nosso navegador.

```
struct lista_ordenada{
    int in_use;
    int size;
    int free_space;
    char** lista;
    int* unidades;
};
```

Esta estrutura é também, como a estrutura acima, uma lista de strings usada para armazenar códigos de clientes e produtos, com a diferença que e como o nome diz usada de forma ordenada, utilizada por exemplo na query 10, onde precisa de uma lista que seja possível ordenar de acordo com o produto mais comprado de ordem decrescente.

```
struct lista_n{
    int N;
    int in_use;
    char** lista;
    int* unidades;
    int* n_clientes;
};
;
```

Esta estrutura por sua vez, é utilizada para armazenar os N maiores, ou seja para armazenar os N produtos mais vendido ou clientes mais compradores.

```
Lista_Ordenada iniciar_lista_ordenada(int size){
    Lista_Ordenada s = (Lista_Ordenada) malloc(sizeof(struct
    lista_ordenada));

    s->in_use = 0;
    s->size = size;
    s->free_space = size;
    s->lista = (char**) malloc(sizeof(char*) * size);
    s->unidades = (int*) malloc(sizeof(int) * size);

    return s;
}
```

Exemplo de uma função que inicializa uma lista ordenada.

3 AVL

Este ficheiro é o centro do nosso projeto, porque para além de conter as estruturas acima referidas, é também aqui que estão as funções de procura nessas listas e AVL, bem como funções que as inicializam e inserem elementos.

```
int search_tree(AVL a, int id){
  int r;
  if(a == NULL) r = 0;
  else{
    if(id == a->valor) r = 1;
    else{
      if(id < a->valor) r = search_tree(a->esq, id);
      else r = search_tree(a->dir, id);
    }
  }

  return r;
}
```

Por exemplo, esta função serve para verificar se um determinado elemento existe nessa AVL.

```
void add_lista(Lista_Strings s, char* c){
    is_full(s);
    s->lista[s->in_use++] = sdup(c);
}
```

Função que adiciona uma string a uma lista de strings.

Estes são só alguns exemplos, existem muitas outras funções com propósitos semelhantes, de organização e manipulação destas estruturas.

4 Registos de Clientes

Este ficheiro e extremamente importante no nosso projecto. É onde se encontram praticamente todas as funções de consulta de dados em relação aos clientes.

```
struct registos_cliente{
  AVL tabela_cliente[LETRAS][HASHNUMBER];
};
```

Esta estrutura é onde guardamos os clientes em forma de tabela e dividida por LETRAS (26) e um HASHNUMNER (599), numa estrutura AVL referida acima.

```
struct cliente{
    int comprou_in[5];  //se comprou em determinada filial , em
    todas [4] ou em nenhuma [0]
    int vezes_comprou[12][3];  //quantas vezes comprou por m s e
    por filial
    int size_comprados; //quantos comprou no total

    Lista_Ordenada P[12];// lista ordenada de produtos comprados
    por este cliente por m s
};
```

Esta estrutura define um cliente, guardando todas as informações relevantes para o restante projecto, como por exemplo se comprou em determinada filial, e quantas vezes comprou por mês e por filial, e a totalidade de produtos comprados.

Neste ficheiro temos também todas as funções de organização dos dados referentes aos clientes.

```
Cliente iniciar_cliente(){
  Cliente c = malloc(sizeof(struct cliente));

  for(int i = 0; i < 5; i++)
    c->comprou_in[i] = 0;

  for(int m = 0; m < 12; m++)
    for(int i = 0; i < 3; i++)
      c->vezes_comprou[m][i] = 0;

  c->size_comprados = 0;

  for(int m = 0; m < 12; m++)
    c->P[m] = NULL;

  return c;
}
```

Esta função inicia um cliente a 0 ou NULL.

```
RC iniciar_RC(){
  RC c = malloc(sizeof(struct registos_cliente));

  for (int i = 0; i < LETRAS; i++)
    for(int k = 0; k < HASHNUMBER; k++)
      c->tabela_cliente[i][k] = NULL;
```

```
7
8   return c;
9 }
```

Já esta fução inicializa o registo de clientes a NULL.

```
1 int c_vezes_comprou(RC c, char* cliente, int m, int f){
2   int nID = num_(cliente,1);
3   int index[3]; index[0] = 0, index[1] = 0, index[2] = 0;
4   hF_c(index,cliente);
5
6   Cliente c2 = search_info(c->tabela_cliente[index[0]][index[1]],
      nID);
7
8   return c2->vezes_comprou[m][f];
9 }
```

Esta função devolve quantas vezes um cliente comprou por filial e por mês. Estas são só algumas das funções que estão neste ficheiro.

5 Registos de Produtos

Semelhante ao ficheiro de Registos de Clientes, também os Registos de Produtos contêm quase todas as informações relativas ao produtos, bem como estruturas e funções que nos ajudam a consultar e guardar esses mesmos produtos.

```
1 struct registos_produto{
2   AVL tabela_produto[LETRAS][LETRAS][HASHNUMBER];
3 };
4
5
6 struct produto{
7     int vendido_in[5]; //se foi comprado em determinada filial, em
      todas [4] ou em nenhuma [0]
8     int vezes_comprado[12][3]; //vezes vendido por filial e por
      m s
9     int unidades_vendidas[3]; // numero de unidades vendidas por
      filial
10    int faturado_in[12][3][2]; //fatura  o por filial, m s e se
      estava em promo  o ou n o
11    int vezes_N[12][3];// vezes vendido com pre o normal por
      filial e por m s
12    int vezes_P[12][3];// vezes vendido em promo  o por filial e
      por m s
13    Lista_Strings N[4];
14    Lista_Strings P[4];
15 };
```

Esta é Estrutura que armazena os produtos numa AVL, e uma estrutura com todas as informações que achamos relevantes em relação a cada produto individualmente, muito semelhante ao que acontece nos clientes. Estas estruturas são muito importantes para conseguirmos aceder a sua informação necessária mais tarde nas queries e funções auxiliares a essas queries.

```
int p_vezes_comprado(RP p, char* produto, int m, int f){
  int nID = num_(produto, 2);
  int index[3]; index[0] = 0, index[1] = 0, index[2] = 0;

  hF_p(index, produto);

  Produto p2 = search_info(p->tabela_produto[index[0]][index[1]][
    index[2]], nID);

  return p2->vezes_comprado[m-1][f-1];
}

int p_faturado_in(RP p, char* produto, int m, int f, char NP){
  int nID = num_(produto,2), c = 42;
  int index[3]; index[0] = 0, index[1] = 0, index[2] = 0;
  hF_p(index,produto);

  Produto p2 = search_info(p->tabela_produto[index[0]][index[1]][
    index[2]], nID);

  if(NP == 'N')
    c = 0;
  if(NP == 'P')
    c = 1;

  return p2->faturado_in[m-1][f-1][c];
}
```

Estas funções acima representadas são apenas alguns exemplos de utilização das estruturas mostradas acima. Neste caso a função p-vezes-comprado é uma função que devolve o numero de vezes que um produto foi vendido num determinado mês numa determinada filial. Já a p-faturado-in devolve o total faturado de um produto, num determinado mês, filial e condição (N ou P).

6 Catalogo de Vendas

Este ficheiro é onde as vendas sâo lidas e validadas.

```
int validaVenda(char* linha, Produtos p, Clientes c){
  int r = 0, i = 0;
  char* tokens[7];

  i = toktok(linha, tokens);

  if(i == 7)
    if( atof(tokens[1]) <= 999.99 && atof(tokens[1]) >= 0.0 )
      if( atoi(tokens[2]) <= 200 && atoi(tokens[2]) >= 1 )
        if( tokens[3][0] == 'N' || tokens[3][0] == 'P')
          if( atoi(tokens[5]) <= 12 && atoi(tokens[5]) >= 1 )
            if( atoi(tokens[6]) <= 3 && atoi(tokens[6]) >= 1 )
              if( search_C(c, tokens[4]) ) // clientes
                if( search_P(p, tokens[0]) ) // produtos
                  r = 1;
  return r;
}

void load_vendas(char* path, Produtos p, Clientes c, Filial f1,
    Faturacao f2, int num[2]){
  char linha[32], *original = malloc(sizeof(char)*32);
  int i1 = 0, i2 = 0;
  char* tokens[7];

  FILE* file = fopen(path , "r");

  while( fgets(linha, 32, file) ){
    strcpy(original, linha);
    if(validaVenda(linha, p, c)){
      toktok(original, tokens);

      update_faturacao(f2, atoi(tokens[6]), atoi(tokens[5]), atof(
    tokens[1]), atoi(tokens[2]));

      update_cliente(get_clientes(f1), tokens[4], atoi(tokens[6]),
    atoi(tokens[5]), atof(tokens[1]), atoi(tokens[2]), tokens[0],
    tokens[3][0]);
      update_produto(get_produtos(f1), tokens[0], atoi(tokens[6]),
    atoi(tokens[5]), atof(tokens[1]), atoi(tokens[2]), tokens[4],
    tokens[3][0]);

      i1++;
    }
    i2++;
  }

  num[0] = i1;
  num[1] = i2;
  free(original);
  fclose(file);
}
```

Acima estão as funções de leitura e validação das vendas.

7 Catalogo de Clientes

De forma semelhante ao Catalogo de Vendas, também aqui é onde os clientes são lidos e validados bem como armazenados numa estrutura aqui criada.

```
struct clientes{
  AVL tabela_clientes[LETRAS][HASHNUMBER];
};
```

Usando a estrutura AVL criada acima, criamos uma tabela onde os clientes são armazenados.

```
void hF_clientes(int index[2], char value[]){
  int s = 0, c = 0;
  for(int i = 1; i < strlen(value); i++){
    s = value[i]; c+=s; c*=s;
  }

  c = abs(c % HASHNUMBER);

  index[0] = value[0] - 65;
  index[1] = c;
}
```

Função de procura na tabela.

8 Catalogo de Produtos

Este é também um ficheiro onde se lê, valida e armazena os produtos.

```
struct produtos{
  AVL tabela_produtos[LETRAS][LETRAS][HASHNUMBER];
};
```

Aqui também como no Catalogo de Clientes criamos uma estrutura com uma tabela de produtos.

```
int insert_produto(Produtos p, char* id){
  int index[3]; index[0] = 0, index[1] = 0, index[2] = 0;
  int nID = num_(id,2);
  hF_produtos(index, id);

  p->tabela_produtos[index[0]][index[1]][index[2]] = insert_tree(p
    ->tabela_produtos[index[0]][index[1]][index[2]] , nID, id, 'b')
    ;
  return index[0];
}
```

Função que insere um produto na tabela.

9 Faturação

É neste modulo que a faturação é feita e armazenada.É aqui também que estão todas as funções de manipulação da faturação.

```
struct faturacao{
  double tabela_faturacao[12][4];
  int tabela_vendas[12];
  double total_faturado;
};
```

Esta estrutura consiste numa tabela de faturação por mês e por filial bem como o seu total, tem também uma tabela de vendas dividida por mês, e o total faturado.

```
void update_faturacao(Faturacao f1, int filial, int mes, double
    preco, int unidades){
  f1->tabela_faturacao[mes-1][3] += preco*unidades;
  f1->tabela_faturacao[mes-1][filial-1] += preco*unidades;
  f1->tabela_vendas[mes-1]++;
  f1->total_faturado += preco*unidades;
}
```

Função que dado os parâmetros de uma venda dá update na estrutura faturação.

10 Filial

Neste modulo a filial é criada, através dos registos de clientes e produtos.

```
struct filial{
  RP produtos;
  RC clientes;
};
```

Estrutura da Filial usando os Registos de Clientes e Produtos criados nos módulos de mesmo nome.

11 Interface

É aqui que construímos as funções que dão origem ás queries que a equipa de docentes nos atribui, bem como a estrutura que da origem ao SGV. É por isso um modulo fulcral.

```
struct sgv {
  Clientes c;
  Produtos p;
  Filial f1;
  Faturacao f2;
};
```

Estrutura que reúne todos os módulos num único.

```
void query_10(SGV s, char* cliente, int mes, Lista_Ordenada P){

  RC clientes = get_clientes(s->f1);

  get_lista_P(clientes, cliente, mes, P);

  heapSort(P);
}
```

Função que realiza a query 10.

12 IO

Este é o modulo onde toda a interacção entre o utilizador e o programa acontece. É aqui que o menu para selecionar as queries e impresso, bem com qualquer outra informação relevante para o utilizador.

```
void menu(SGV s){
  char* r = malloc(sizeof(char*));
  printf("Pretende dar print ao menu? Yes: [y] No: [n]\n");

  if(scanf("%s", r)){
    if(r[0] == 'y' || r[0] == 'Y'){
      print_menu();
      load_query1(s);
      while(1){
        escolhe_query(s);
      }
    }

    if(r[0] == 'n' || r[0] == 'N'){
      load_query1(s);
      while(1){
        escolhe_query(s);
      }
    }
  }
  else {
    printf("\n\tO programa falhou na leitura da resposta. Yes: [y/Y
    ] No: [n/N]\n");
    menu(s);
  }

  free_sgv(s);
  free(r);
}
```

Função usada para imprimir o menu.

Teste de Desempenho: Tempo

Teste 1M	1	2	3	Media
Query_1	4.84s	4.81s	4.84s	4.83s
Query_2	0.015s	0.0s	0.0s	0.005s
Query_3	0.0s	0.0s	0.0s	0.0s
Query_4	0.031s	0.015s	0.015s	0.02s
Query_5	0.0s	0.0s	0.015s	0.005s
Query_6	0.015s	0.0s	0.015s	0.01s
Query_7	0.0s	0.0s	0.0s	0.0s
Query_8	0.0s	0.0s	0.0s	0.0s
Query_9	0.0s	0.0s	0.0s	0.0s
Query_10	0.0s	0.0s	0.0s	0.0s
Query_12	0.0s	0.0s	0.0s	0.0s

N	10	100	1000	10000
Query_11	0.17s	0.22s	0.73s	5.83s

Na tabela acima á esquerda testamos os tempos das queries(1 a 10 e 12) fornecidas pelos professores, sendo a query 1 o load dos ficheiros, neste caso de Vendas_1M. Já na tabela á direita testamos o tempo da queries 11 com os N primeiros elementos (de 10 a 10000), também com o ficheiro de Vendas_1M.
Como é possivel ver só a Query1 é que tem um tempo verdadeiro de espera, sendo que as outras queries são praticamente instantâneas. Isto deve-se ao facto de que o computador se limita a ler a informação que foi anteriormente carregada e armazenada na Query1.

Teste 3M	1	2	3	Media
Query_1	21.67s	22.04s	20.72s	12.95s

N	10	100	1000	10000
Query_11	0.17s	0.22s	0.92s	10s

Estes testes foram efetuados usando o ficheiro Vendas_3M. Como é possível ver os tempos da Query1 são superiores, como previsto, mas já o das queries 2 a 10 são essencialmente os mesmos, visto que é basicamente só leitura de dados. Já na Query 11 é notório o aumento de tempo principalmente com maiores quantidades de elementos.

Teste 5M	1	2	3	Media
Query_1	49.3s	49.51s	49.01s	23.65s

N	10	100	1000	10000
Query_11	0.16s	0.22s	1.06s	11s

De forma semelhante ao teste feito no ficheiro de 3M, também aqui só se nota um aumento de tempo na Query1 e na Query 11.

Teste de Desempenho: Memória

Memória	Teste 1M	Teste 3M	Teste 5M
Query_1	827.6MB	2049.2MB	3249.8MB

Nesta tabela é possível ver a quantidade de memória usada para realizar a Query1 com os diversos ficheiros de vendas.
Estes valores apesar de altos são muito melhores que os que tínhamos no inicio.

Este é um exemplo da quantidade de memória usada pelo programa antes de o tentarmos optimizar, como é visível chegou aos quase 16GB de memória, apesar de que estava a ser corrido com o programa valgrind, que por sua vez consome já ele uma quantidade de memória.

Memória	Teste 1M	Teste 3M	Teste 5M
Query_11	999.3MB	2318.7MB	3562.9MB

Acima está representado o uso de memória do programa quando executa a Query11 com 1000 elementos. Comparativamente á tabela acima existe um aumento ligeiro, isto deve se ao facto de que quando a Query11 é executada, esta aloca memória para criar a lista de produtos com o numero de unidades vendidas e quantidade de clientes que o compraram.

Conclusão

Em suma, o grupo pensa que o projeto foi bastante produtivo e aprendemos bastante, principalmente no que diz respeito a gestão de memória e como verdadeiramente estruturar um projeto de maior dimensão.

Chegámos também á conclusão de que ainda podemos melhorar bastante em todos os sentidos, nomeadamente na gestão de memória acima referida.