Estruturas de Dados e Algoritmos II

Cestos de Compras

Elaborado por:

João Calhau nº31621

José Pimenta nº31677

Indice

1.Introdução

O trabalho de Estruturas de Dados e Algoritmos II tem como objectivo simular um sistema online de comércio eletrónico que suporte a noção de cesto de compras, que o cliente vai enchendo durante a sua visita à loja virtual.

Iremos assim implementar o dito sistema de modo a podermos realizar operações sobre os cestos, incluindo a criação e destruição, alteração de artigos nos mesmos e realizações de encomendas.

Para isso iremos implementar, através dos conhecimentos adquiridos na disiciplina de EDA2, algoritmos e estruturas de dados que sejam adequadas ao referido problema.

2.Estruturas de dados

2.1.Constituição das estruturas

Para realizar o trabalho criámos as seguintes estruturas de dados:

Items:

```
//Definir itens
typedef struct items {
    float price;                    //4 bytes
    char code[TAMANHO_ITEM+1];      //11 bytes
    bool active;                    //1 bytes
    short quant;                    //2 bytes
    char fill[2];                   //2 bytes
} item ;                            //total 20 bytes
```

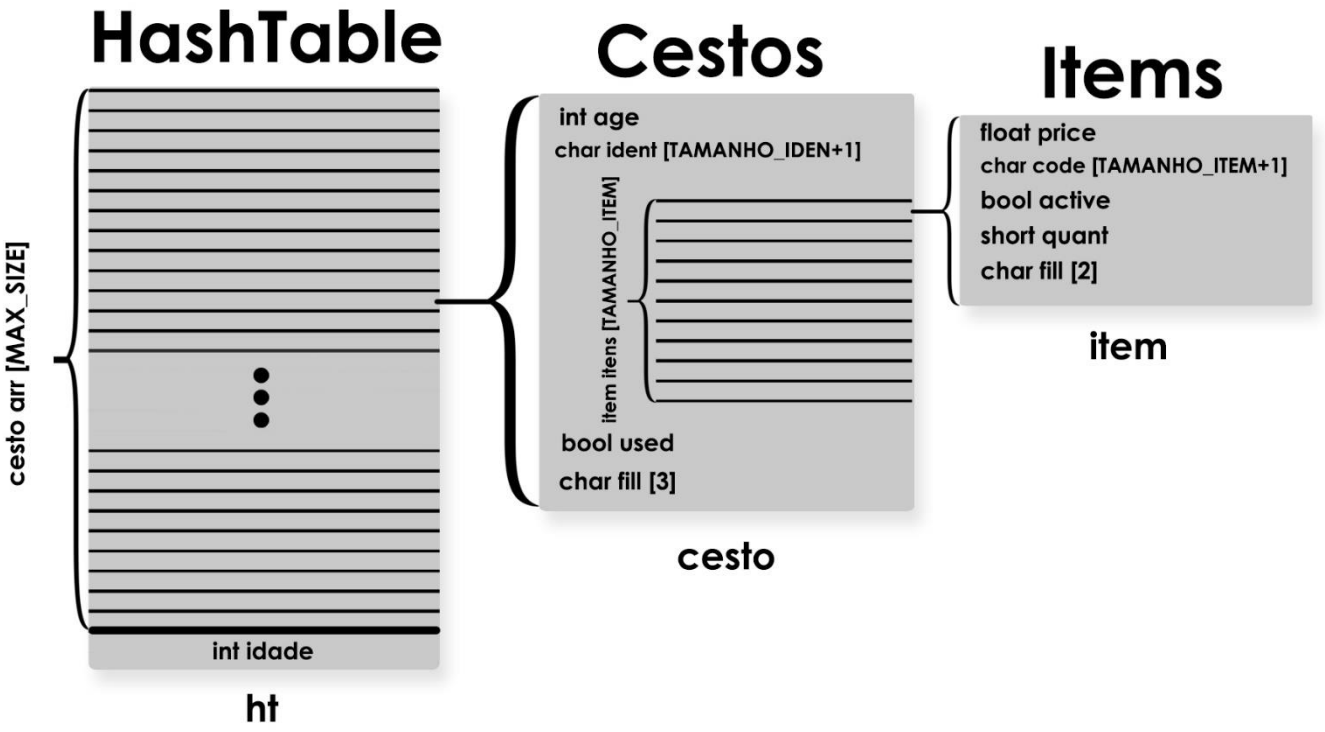
Cestos:

```
//Definir cestos de itens
typedef struct Cestos {
    int age;                        //4 bytes
    char ident[TAMANHO_IDEN+1];     //8 bytes
    item itens[TAMANHO_ITEM];       //200 bytes
    bool used;                      //1 bytes
    char fill[3];                   //3 bytes
} cesto ;                           //total 216 bytes
```

HashTable:

```
//Definir HashTable
typedef struct HashTable {
    cesto arr[MAX_SIZE];            //25 922 376 bytes
    int idade;                      //4 bytes
} ht ;                              //total 25 922 380 bytes (25,9 MB)
```

2.2.Diagrama e discrição das estruturas de dados usadas



Para a realização deste trabalho criamos as 3 estruturas referidas.

Na criação da HashTable incluímos uma variável, ***int idade***, que contem o tempo passado desde a criação da HashTable até ao momento atual. Para além disso a HashTable contem um array de cestos, ***cesto arr[MAX_SIZE],*** em que *MAX_SIZE* é a quantidade maxima de cestos possível na HashTable, sendo que o valor dessa variável, no nosso trabalho, é 200 003 pois é o primo mais próximo de 200 000, valor em que o factor de carga da HashTable seria 0,5 pois o máximo de cestos que podem ser inseridos é 100 000.

Na criação dos Cestos incluímos uma variável age, ***int age***, que guarda o instante em que ouve uma operação sobre cesto, incluímos uma variável ident, ***char ident[TAMANHO_IDENT+1],*** que tem o id do cesto e que tem espaço suficiente para 7 caracteres e +1 para o caracter nulo (‘\0’). Temos ainda uma variável used, ***bool used***, que serve para dizer se o cesto se encontra activo, ou não, na HashTable, uma variável fill, ***char fill[3],*** que só serve para uma organização de memória e um endereçamento corretos. Finalmete temos ainda um array de itens***, item itens[TAMANHO_ITEM],*** que tem espaço para 10 tipos de itens (visto que cada cesto só consegue ter 10 produtos diferentes).

Na criação dos Items incluímos uma variável price**, *float price***, que contem o preço unitário de cada item, uma variável code, ***char code[TAMANHO_ITEM+1]*****,** que contem o id do item de tamanho máximo 10 e +1 para o caracter nulo (‘\0’). Contem ainda uma variável active, ***bool active*****,** que informa se o item que está no cesto está activo ou não, uma variável quant**, *short quant***, que identifica a quantidade do item no cesto e uma variável fill, ***char fill[2],*** que, outra vez, só serve para uma organização de memória e um endereçamento corretos.

3.Formatos

Escolhemos guardar os dados num ficheiro binário porque esse tipo de ficheiro consegue guardar todo o tipo de dados, codificados em binário.

Quantidade de páginas totais para guardar em disco:

256MB = 1024 * 1024 * 256 = 268 435 456 Bytes

Páginas de 4096 Bytes

268 435 456 / 4096 = 65 536 Páginas

Quantidade de páginas utilizadas:

Tamanho da estrutura: 25 922 380 bytes

Páginas de 4096 bytes

25 922 380 / 4096 ≈ 6329 Páginas

Com esta estrutura de dados usamos apenas aproximadamente 1/10 das páginas totais.

4.Funcionamento do programa

No inicio do programa é criada uma tabela (é alocado espaço para a tabela, e são colocados todos os cestos, desativados, na tabela. O tempo da tabela é inicializado a 0).

O programa tenta ler de um ficheiro uma tabela já existente, caso não exista é utilizada a tabela criada anteriormente, senão é lida de disco e colocada no espaço alocado.

O programa vai receber os inputs do utilizador, e consoante estes vai escolher a operação correta a ser aplicada à tabela.

4.1.Criação de cesto de compras

Para criarmos um cesto de compras verificamos, a partir do nome do cesto, se este já se encontra na tabela, para isso fazemos hash do nome. Se na posição dada pelo hash se encontra um cesto ativo, verifica-se se o nome é igual. Em caso afirmativo o cesto já existe na tabela, caso contrário faz um novo hash. Se não se encontrar um cesto ativo é porque chegou a uma posição onde poderia estar, não estando insere-se nessa posição com uma idade igual ao tempo de inserção.

4.2.Introdução de artigos num cesto de compras

Para introduzir artigos num cesto, à semelhança da criação, vamos verificar se o cesto existe. Se não existe, não se insere nenhum item. Caso contrario, percorre os itens do cesto e vai guardando a ultima posição livre encontrada. Se encontrar o item, aumenta-lhe a quantidade, senão insere um item na última posição livre guardada anteriormente. É alterada ainda a idade do cesto para o tempo atual.

4.3.Retirada de artigos de um cesto de compras

Para retirar artigos de um cesto, à semelhança da criação, vamos verificar se o cesto existe. Se não existe, não se retira nada. Caso contrario, percorre os itens todos do cesto até encontrar o item desejado. Caso encontre, se a quantidade que se pretende remover for superior á que se encontra no cesto, não se remove, se for menor remove a quantidade desejada no cesto e se for igual remove o item do cesto. Caso não encontre o item no cesto, não se remove nada. É alterada ainda a idade do cesto para o tempo atual.

4.4.Visualização do conteúdo de um cesto de compras

Para visualizar os itens de um cesto, á semelhança da criação, vamos verificar se o cesto existe. Se não existe, não se faz nada. Caso contrario, percorre todos os itens do cesto e vai guardando os seus apontadores dos itens ativos para uma lista. Caso não encontre nenhum item ativo é porque o cesto está vazio, caso contrário ordena os itens da lista pela ordem crescente do código, percorre a lista, agora ordenada, e calcula o preço total de cada item e o preço total do cesto. É alterada ainda a idade do cesto para o tempo atual.

4.5.Realização da encomenda

Para realização da encomenda de um cesto, á semelhança da criação, vamos verificar se o cesto existe. Se não existe, não se faz nada. Caso contrário e se o cesto estiver vazio atualiza-se a idade e não se efetua mais nenhuma operação sobre o mesmo, se o cesto não estiver vazio, percorre todos os itens do cesto e "desativa-os", estando isto feito, "desativa-se" também o cesto e é efetuada a venda do mesmo.

4.6.Cancelamento de um cesto de compras

Para o cancelamento de um cesto, á semelhança da criação, vamos verificar se o cesto existe. Se não existe, não se faz nada. Caso contrário percorre todos os itens do cesto e "desativa-os", estando isto feito o cesto é "desativado" e assim cancelado.

4.7.Limpeza dos cestos de compras

Para fazer a limpeza dos cestos, através do tempo atual e da idade de remoção dada pelo utilizador, é calculado o tempo a partir do qual não são removidos cestos. Assim sabemos que cestos com tempo inferior a esse têm de ser removidos. São percorridos todos os cestos da tabela, e sempre que se encontre um cesto ativo e com tempo inferior ao calculado, é guardado o apontador para esse cesto numa lista e "desativa-se" o mesmo (O cesto está desativado, mas a informação da idade do mesmo ainda é acessível a partir dos apontadores, visto que os cestos nunca são apagados definitivamente). Após percorrer a tabela, ordena-se a lista por ordem decrescente da idade. Percorre-se a lista, agora ordenada, e imprime-se o nome dos cestos. Caso não encontre cestos, é mostrado que não existem cestos com idade superior à inserida pelo utilizador.

4.8.Fim de execução

Caso a instrução dada pelo utilizador não seja nenhuma das anteriores (XX), ou tenha chegado ao fim de ficheiro, é guardado o tempo atual na tabela e guardada a mesma em disco.

Finalmente o espaço previamente alocado para a tabela é libertado.

5.Justificação das escolhas feitas

5.1.Acessos a disco

No nosso programa só fazemos 2 acessos a disco, um no início para carregar a HashTable (ou não) para memória principal e um no fim para guardar a HashTable com as operações todas feitas sobre ela.

Os acessos ao disco são sempre constantes pois a quantidade de informação é sempre a mesma a ser guardada/lida, visto que a tabela tem sempre o mesmo tamanho (MAX_SIZE, neste caso 200 003).

Decidimos fazer apenas 2 acessos à memória secundária visto que como não usamos apontadores na estrutura da tabela podemos guardar a tabela toda em memória secundária e quando vamos lê-la do disco carregamos a tabela toda para memória principal.

5.2.Estrutura de Dados

5.2.1.HashTable

Escolhemos uma HashTable para fazer este trabalho porque de todas as estruturas de dados que demos até agora era a mais simples de implementar e parecia ser a estrutura melhor para este tipo de problema, visto que vamos ter um máximo de 100 000 cestos e como a HashTable tem um acesso constante (no melhor caso quando só se faz hash 1 vez, O(1)) se dermos um tamanho grande o suficiente (neste caso os 200 003 cestos), muito raramente vão haver colisões, o que evita o pior caso (O(n)) e o facto de termos de fazer rehash à tabela toda, visto que assim o factor de carga será sempre 0,5.

5.3.Algoritmos

5.3.1.Merge_Sort

Na função LC, inicialmente começámos por implementar o bubble sort ($O(n^2)$), mas rapidamente nos apercebemos que para uma quantidade de cestos muito grande um algoritmo com uma complexidade temporal menor seria melhor (claro que também não funcionou no Mooshak). Por essa razão mudámos o algoritmo de ordenação para o Merge sort que tem uma complexidade temporal de O(n log n) o que baixou bastante o tempo de execução (desta vez já funcionou no Mooshak).

Nota: n é tamanho do array a ser ordenado (neste caso o número de

5.3.2.Bubble_Sort_String

Na função VC, implementamos o bubble sort adaptado a comparar strings, que tem uma complexidade temporal de $O(n^2)$, porque é mais simples de implementar e o tamanho máximo de produtos que vamos ordenar é sempre 10. Para números pequenos o tempo de ordenação, entre o Merge Sort e o Bubble sort não diferencia muito.

Nota: n é o tamanho do array a ser ordenado

5.3.3.Hash

Algoritmo de hash Djb2. Este algoritmo baseia-se no número “mágico” 33, o porquê de ser esse o número nunca foi bem fundamentado

Fonte: http://www.cse.yorku.ca/~oz/hash.html

5.4.Funções

5.4.1.Insert_Item, Remove_Item, Visualize_Item e isEmpty

Complexidade O(m), devido ao ciclo for em que m é o tamanho do array de itens no cesto.

5.4.2.Create_HashTable

Complexidade O(n), devido ao ciclo for, em que n é o tamanho do array de cestos na HashTable.

5.4.3.Free_HashTable

Complexidade O(1), visto que é só um free da memória.

5.4.4.Get_Cesto

Complexidade, no pior dos casos O(n) (por causa do ciclo while).

5.4.5.Put_Cesto

Complexidade, no pior dos casos O(n+m) por causa do ciclo while e do ciclo for (em que n é o tamanho do array da tabela e m é o número máximo de itens).

5.4.6.Remove_Cesto

Complexidade, no pior dos casos O(n*m) por causa do ciclo for dentro do ciclo while (em que n é o tamanho do array da tabela e m é o número máximo de itens).

5.4.7.CC

Complexidade desta função é igual á do Put_Cesto, O(n+m).

5.4.8.IC

Complexidade desta função é O(n+m) (n+m porque há 2 chamadas de funções com com complexidades O(m) e O(n), explicadas previamente).

5.4.9.RC

Complexidade desta função é O(n+m) (n+m porque há 2 chamadas de funções com com complexidades O(m) e O(n), explicadas previamente).

5.4.10.VC

Complexidade desta função é O(n+m) (n+m porque há 2 chamadas de funções com com complexidades O(m) e O(n), explicadas previamente).

5.4.11.EC

Complexidade desta função é O(n+m+n*m) (porque há 3 chamadas de funções com complexidades O(n*m), O(n) e O(m), explicadas previamente).

5.4.12.XC

Complexidade desta função é O(n+n*m) (n+n*m porque há 2 chamadas de funções com com complexidades O(n*m) e O(n), explicadas previamente).

5.4.13.LC

Complexidade desta função é O(n^2*m+n*log n) (n^2*m+n*log n devido há chamada de uma função dentro de um ciclo for e há chamada de outra função, com complexidades O(n), O(n*m) e O(n log n)).

Nota: Todas complexidades temporais são para os piores casos, no nosso trabalho quase nunca, ou nunca, chega a esses casos devido à estrutura da HashTable explicada anteriormente.

6.Código do programa

```c
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define MAX_SIZE 200003 //Tamanho da HashTable
#define TAMANHO_IDEN 7  //Tamanho total nome de cestos (sem \0)
#define TAMANHO_ITEM 10 //Tamanho total nome dos itens (sem \0)
#define PRIME 199999    //Primo utilizado na funcao de hash2

//Definir itens
typedef struct items {
    float price;                //4 bytes
    char code[TAMANHO_ITEM+1];  //11 bytes
    bool active;                //1 bytes
    short quant;                //2 bytes
    char fill[2];               //2 bytes
} item ;                        //total 20 bytes


//Definir cestos de itens
typedef struct Cestos {
    int age;                    //4 bytes
    char ident[TAMANHO_IDEN+1]; //8 bytes
    item itens[TAMANHO_ITEM];   //200 bytes
    bool used;                  //1 bytes
    char fill[3];               //3 bytes
} cesto ;                       //total 216 bytes

//Definir HashTable
typedef struct HashTable {
    cesto arr[MAX_SIZE];        //25 922 376 bytes
    int idade;                  //4 bytes
} ht ;                          //total 25 922 380 bytes (25,9 MB)


void merge(cesto **Cesto, int n, int m) {
    int i, j, k;
    cesto **temp = malloc(n * sizeof(cesto));

    for (i = 0, j = m, k = 0; k < n; k++) {
        temp[k] = j == n                        ? Cesto[i++]
                : i == m                        ? Cesto[j++]
                : Cesto[j]->age < Cesto[i]->age ? Cesto[j++]
                :                                 Cesto[i++];
    }

    for (i = 0; i < n; i++) {
        Cesto[i] = temp[i];
    }
    free(temp);
}

```

```c
//merge sort das aulas e EDA1
//Ordena um array dividindo-o em partes mais pequenas
//até essas partes terem 1 elemento
//depois faz "merge" das partes ordenadas
void merge_sort(cesto **Cesto, int n) {
    if (n < 2)
        return;
    int m = n / 2;
    merge_sort(Cesto, m);
    merge_sort(Cesto + m, n - m);
    merge(Cesto, n, m);
}

//Bubble sort de EDA1 adaptado a strings
void bubble_sort_string(item *list[], short n) {
    short c, d;
    item *t;

    for (c = 0; c < (n-1); c++) {
        for (d = 0; d < n - c - 1; d++) {
            if (strncmp(list[d]->code, list[d+1]->code, TAMANHO_ITEM) > 0) {
                t = list[d];
                list[d] = list[d+1];
                list[d+1] = t;
            }
        }
    }
}

//Insere um item num cesto (ou incrementa esse item)
void Insert_Item(ht *HashTable, int hash, char codigo[], short quantidade, float preco) {
    int livre; //ultimo espaco livre a ser encontrado

    //Percorre todos os itens do cesto
    for (int i = 0; i < TAMANHO_ITEM; i++) {
        //caso o espaco esteja livre (active = false) mete a variavel livre nesse espaco
        if (HashTable->arr[hash].itens[i].active == false)
            livre = i;
        //caso o codigo seja igual, aumenta a quantidade desse item
        else if (strcmp(HashTable->arr[hash].itens[i].code, codigo) == 0) {
            HashTable->arr[hash].itens[i].quant += quantidade;
            return;
        }
    }
    //Caso seja um item novo, coloca-o na posicao referenciada pela variavel livre
    strncpy(HashTable->arr[hash].itens[livre].code, codigo, 11);
    HashTable->arr[hash].itens[livre].quant = quantidade;
    HashTable->arr[hash].itens[livre].price = preco;
    HashTable->arr[hash].itens[livre].active = true;
    return;
}
```

```c
//Remove um item de um cesto
void Remove_Item(ht *HashTable, int hash, char identificador[], char codigo[], short quantidade) {
    //Percorre todos os itens do cesto
    for (int i = 0; i < TAMANHO_ITEM; i++) {
        //Caso o item seja activo e o codigo seja igual
        if (HashTable->arr[hash].itens[i].active == true && strcmp(HashTable->arr[hash].itens[i].code, codigo) == 0) {
            //Se a quantidade for maior á que queremos remover, diminui a quantidade
            if (HashTable->arr[hash].itens[i].quant > quantidade) {
                HashTable->arr[hash].itens[i].quant -= quantidade;
                printf("+ %hd unidade(s) de %s retirada(s) do cesto %s\n", quantidade, codigo, identificador);
                return;
            }
            //Se a quantidade for igual, muda o active para false (item ja nao existe neste cesto)
            else if (HashTable->arr[hash].itens[i].quant == quantidade) {
                HashTable->arr[hash].itens[i].active = false;
                printf("+ %hd unidade(s) de %s retirada(s) do cesto %s\n", quantidade, codigo, identificador);
                return;
            }
            //Se a quantidade for maior do que a existente, nao modifica o cesto
            else {
                printf("+ quantidade de %s no cesto %s menor que %hd\n", codigo, identificador, quantidade);
                return;
            }
        }
    }
    //Caso o item nao exista no cesto
    printf("+ produto %s ausente do cesto %s\n", codigo, identificador);
}

//Ve todos os items de um cesto
void Visualize_Item(ht *HashTable, int hash, char identificador[]) {
    short x = 0; //Variavel para saber se o printf de VC <identificador> já foi feito
    float total = 0.0; //preco total de todos os itens do cesto (itens * quant)
    item *list[TAMANHO_ITEM]; //lista de apontadores para os itens de um cesto

    //Percorre todos os itens do cesto
    for (int i = 0; i < TAMANHO_ITEM; i++) {
        //Se o item tiver activo, adiciona um apontador para esse item á lista
        if (HashTable->arr[hash].itens[i].active == true) {
            list[x] = &HashTable->arr[hash].itens[i];
            x++;
            //Se o x tiver a 1, entao faz print a VC <identificador>
            //(so faz 1 vez, no primeiro item a ser adicionado)
            if (x == 1)
                printf("VC %s\n", identificador);
        }
    }
    //Se o x = 0, entao nao há nenhum item ativo
    if (x == 0)
        printf("+ cesto %s vazio\n", identificador);
    //Ordena os apontadores dos itens por ordem crescente do codigo
    else {
        bubble_sort_string(list, x);

        //percorre a lista, calcula o total e faz print dos itens por ordem
        for (int i = 0; i < x; i++) {
            float temp =list[i]->quant * list[i]->price;
            printf("%s %3hd %8.2f %8.2f\n", list[i]->code, list[i]->quant, list[i]->price, temp);
            total += temp;
        }
        //print do preco total
        printf("%.2f\n", total);
    }
}

//verifica se um cesto está vazio
bool isEmpty(ht *HashTable, int hash) {
    bool vazio = true;

    //percorre os itens de um cesto e verifica se esta vazio
    for (int i = 0; i < TAMANHO_ITEM; i++) {
        if (HashTable->arr[hash].itens[i].active == true) {
            vazio = false;
        }
    }
    return vazio;
}
```

```c
//Malloc necessario para por a HashTable em memoria
ht *Create_HashTable() {

    ht *HashTable = malloc(sizeof(ht));

    //inicializar todos os elementos da tabela a false
    for (int i = 0; i < MAX_SIZE; i++)
        HashTable->arr[i].used = false;

    //Inicializa a idade da HashTable a 0
    HashTable->idade = 0;

    return HashTable;
}

//Free necessario para tirar a HashTable de memoria
void Free_HashTable(ht *HashTable) {
    free(HashTable);
}

//funcao de hash djb2
unsigned long HashCode(char x[]) {
    unsigned long hash = 5831;
    int c;

    while ( (c = *x++) ) {
        hash = ((hash << 5) + hash) + c;
    }

    return hash;
}

//primeira funcao de duplo hashing de EDA1
int Hash1(char x[]) {
    unsigned long hashcode = HashCode(x);
    hashcode %= MAX_SIZE;

    int hash = (int) hashcode;

    if (hash < 0)
        hash += MAX_SIZE;

    return hash;
}

//segunda funcao de duplo hashing de EDA1
int Hash2(ht *HashTable, char x[]) {
    unsigned long hashcode = HashCode(x);
    hashcode %= MAX_SIZE;

    int hash = (int) hashcode;

    if (hash < 0)
        hash += MAX_SIZE;

    return PRIME - hash % PRIME;
}
```

```
243  //retorna um cesto de id = key
244  int Get_Cesto(ht *HashTable, char key[]) {
245      int hash1 = Hash1(key);
246      int hash2 = Hash2(HashTable, key);
247
248      //Procura o cesto enquanto o cesto estiver a ser usado
249      while (HashTable->arr[hash1].used == true) {
250          //Se o nome for igual retorna a posicao do cesto
251          if (strcmp(key, HashTable->arr[hash1].ident) == 0)
252              return hash1;
253          //Se o nome nao e igual, faz duplo hashing
254          else {
255              hash1 += hash2;
256              hash1 %= MAX_SIZE;
257          }
258      }
259      //Caso chegue a um cesto que nao esta a ser usado
260      //Logo o cesto nao existe no sistema
261      //E retorna -1
262      return -1;
263  }
264
265  //Insere um cesto na hashtable
266  bool Put_Cesto(ht *HashTable, int idade, char key[]) {
267      int hash1 = Hash1(key);
268      int hash2 = Hash2(HashTable, key);
269
270      //Procura o cesto enquanto o cesto estiver a ser usado
271      while(HashTable->arr[hash1].used == true) {
272          //Se o nome for igual retorna false
273          //(Cesto ja existe no sistema, logo nao inseriu)
274          if (strcmp(key, HashTable->arr[hash1].ident) == 0){
275              return false;
276          }
277          //Se o nome nao e igual, faz duplo hashing
278          else {
279              hash1 += hash2;
280              hash1 %= MAX_SIZE;
281          }
282      }
283
284      //Caso tenha encontrado uma posicao a false
285      //Insere um cesto (muda o used para true, muda a idade e o nome)
286      if (HashTable->arr[hash1].used == false) {
287          HashTable->arr[hash1].used = true;
288          strncpy(HashTable->arr[hash1].ident, key, 8);
289          HashTable->arr[hash1].age = idade;
290          //percorre os itens e mete-os todos a false
291          for (int i = 0; i < TAMANHO_ITEM; i++)
292              HashTable->arr[hash1].itens[i].active = false;
293      }
294      //Inseriu o cesto e retorna true
295      return true;
296  }
297
```

```c
//Remove um cesto da tabela
bool Remove_Cesto(ht *HashTable, char key[]) {
    int hash1 = Hash1(key);
    int hash2 = Hash2(HashTable, key);

    //Procura o cesto enquanto o cesto estiver a ser usado
    while(HashTable->arr[hash1].used == true) {
        //Se o nome for igual
        if (strcmp(key, HashTable->arr[hash1].ident) == 0) {
            //Percorre todos os itens e mete-os a false
            for (int i = 0; i < TAMANHO_ITEM; i++)
                HashTable->arr[hash1].itens[i].active = false;
            //Muda o cesto para false e retorna true (cesto removido)
            HashTable->arr[hash1].used = false;
            return true;
        }
        //Se o nome for diferente, faz duplo hashing
        else {
            hash1 += hash2;
            hash1 %= MAX_SIZE;
        }
    }
    //Encontrou um cesto que nao esta a ser usado (used = false)
    //Logo nao há mais cestos para a frente
    //Retorna false
    return false;

}
```

```c
//Insere um cesto na tabela
void CC(ht *HashTable, int idade) {
    char id[TAMANHO_IDEN+1]; //Char com o nome do cesto

    //Scan para ver o nome do cesto
    scanf("%s", id);
    bool teste = Put_Cesto(HashTable, idade, id);
    //Se teste == true, entao inseriu
    if (teste)
        printf("+ cesto %s criado\n", id);
    //Caso contrario, o cesto ja existia
    else
        printf("+ cesto %s existente\n", id);
}
```

```c
//Insere um item num cesto
void IC(ht *HashTable, int idade) {
    char id[TAMANHO_IDEN+1], cod[TAMANHO_ITEM+1]; //Chars com o nome do cesto e o codigo do item
    short quant; //Quantidade do item a ser inserido
    float preco; //Preco do item a ser inserido

    //Scan para ver o nome do cesto, codigo, quantidade e preco unitario do item
    scanf("%s %s %hd %f", id, cod, &quant, &preco);

    int teste = Get_Cesto(HashTable, id);
    //Se for <0 quer dizer que o cesto nao existe
    if (teste < 0)
        printf("+ cesto %s inexistente\n", id);
    //Caso contrario o cesto foi encontrado e é inserido um item na posicao teste
    else {
        Insert_Item(HashTable, teste, cod, quant, preco);
        HashTable->arr[teste].age = idade; //Idade atualizada
        printf("+ %hd unidade(s) de %s introduzida(s) no cesto %s\n", quant, cod, id);
    }
}
```

```c
//Remove um item de um cesto
void RC(ht *HashTable, int idade) {
    char id[TAMANHO_IDEN+1], cod[TAMANHO_ITEM+1]; //Chars com o nome do cesto e o codigo do item
    short quant; //Quantidade a ser removida do item

    //Scan para ver o nome do cesto, codigo e quantidade do item
    scanf("%s %s %hd", id, cod, &quant);

    int teste = Get_Cesto(HashTable, id);
    //Se for <0 quer dizer que o cesto nao existe
    if (teste < 0)
        printf("+ cesto %s inexistente\n", id);
    //Caso contrario o cesto foi encontrado e é removido (ou nao) o item
    else {
        Remove_Item(HashTable, teste, id, cod, quant);
        HashTable->arr[teste].age = idade; //Idade atualizada
    }
}

//Ve todos os itens num cesto
void VC(ht *HashTable, int idade) {
    char id[TAMANHO_IDEN+1]; //Char com o nome do cesto

    //Scan para ver o nome do cesto
    scanf("%s", id);

    int teste = Get_Cesto(HashTable, id);
    //Se for <0 quer dizer que o cesto nao existe
    if (teste < 0)
        printf("+ cesto %s inexistente\n", id);
    //Caso contrario o cesto foi encontrado e é feito um print de todos os itens (caso existam)
    else {
        Visualize_Item(HashTable, teste, id);
        HashTable->arr[teste].age = idade; //Idade atualizada
    }
}

//Efectua a venda de um cesto (caso nao esteja vazio)
void EC(ht *HashTable, int idade) {
    char id[TAMANHO_IDEN+1]; //char com o nome do cesto

    //Scan para ver o nome do cesto
    scanf("%s", id);

    int teste = Get_Cesto(HashTable, id);
    //Se for <0 quer dizer que o cesto nao existe
    if (teste < 0 )
        printf("+ cesto %s inexistente\n", id);
    //Caso contrario
    else {
        //Se o cesto nao estiver vazio, efectua-se a venda do cesto
        if (isEmpty(HashTable, teste) == false) {
            Remove_Cesto(HashTable, id);
            printf("+ efectuada venda do cesto %s\n", id);
        }
        //Se o cesto estiver vazio, atualiza-se a idade e nao se efectua a venda
        else {
            printf("+ cesto %s vazio\n", id);
            HashTable->arr[teste].age = idade;
        }
    }
}
```

```c
//cancela um cesto (caso exista)
void XC(ht *HashTable) {
    char id[TAMANHO_IDEN+1]; //Char com o nome do cesto

    //Scan para ver o nome do cesto
    scanf("%s", id);

    int teste = Get_Cesto(HashTable, id);
    //Se for <0 quer dizer que o cesto nao existe
    if (teste < 0)
        printf("+ cesto %s inexistente\n", id);
    //Caso contrario o cesto é removido do sistema
    else {
        Remove_Cesto(HashTable, id);
        printf("+ cesto %s cancelado\n", id);
    }
}

//remove todos os cestos cuja idade é maior que a idade passada como argumento
void LC(ht *HashTable, int idade) {
    int age; //Idade minima para ser removida
    int index = 0; //Tamanho da lista a ser ordenada
    bool lc = true; //Variavel para saber se o printf de LC <idade> ja foi realizado
    cesto *list[idade]; //Lista de apontadores para cestos a ser ordenada
    //A lista tem tamanho idade, porque o maior numero de cestos que se pode ter posto naquele momento
    //na hashtable seria igual á idade atual

    //Scan da idade minima para a remocao
    scanf("%d", &age);

    //Calculo do tempo ate que um cesto e removido
    int tempo = idade - age;
    //Percorre todos os cestos da hashtable
    for (int i = 0; i < MAX_SIZE; i++) {
        //Se estiver ativo e a idade de insercao for menor que o tempo (calculado anteriormente)
        //adiciona-se um apontador para o mesmo á lista e remove-se o cesto da hashtable
        //(Os valores de idade continuam acessiveis, visto que nao sao realmente removidos
        //apenas se muda o used para false)
        if (HashTable->arr[i].used == true && HashTable->arr[i].age < tempo) {
            //Se for o primeiro cesto, faz print a LC <idade> (so sera feito uma vez)
            if (lc) {
                printf("LC %d\n", age);
                lc = false;
            }
            list[index] = &HashTable->arr[i];
            index++;
            Remove_Cesto(HashTable, HashTable->arr[i].ident);
        }
    }
    //Se o lc for true (nao foi encontrado nenhum cesto com idade superior) faz print
    if (lc)
        printf("+ sem cestos com idade superior a %d\n", age);
    //Caso contrario, ordenam-se os cestos por idade decrescente
    else {
        merge_sort(list, index);
        //percorre a lista (agora ordenada) e faz print da mesma
        for (int i = 0; i < index; i++) {
            printf("%s\n", list[i]->ident);
        }
    }

}
```

```c
//Sabendo-se as letras das operacoes (passadas por char in[]) chama-se a funcao adequada
short Cestos_Virtuais(ht *HashTable, char in[], int idade) {
    //switch simples para saber qual é o input certo
    //Faz return 1 se a operacao nao é a ultima do input
    //Caso contrario faz return -1 (se no in[] está um XX ou se chegou ao fim de ficheiro)
    switch (in[0]) {
        case 'C' :
            CC(HashTable, idade);
            return 1;
        case 'I' :
            IC(HashTable, idade);
            return 1;
        case 'R' :
            RC(HashTable, idade);
            return 1;
        case 'V' :
            VC(HashTable, idade);
            return 1;
        case 'E' :
            EC(HashTable, idade);
            return 1;
        case 'X' :
            if (in[1] == 'C') {
                XC(HashTable);
                return 1;
            } else {
                return -1;
            }
        case 'L' :
            LC(HashTable, idade);
            return 1;
        default :
            return -1;
    }
}

int main() {
    ht *HashTable = Create_HashTable();

    FILE *file = fopen("ficheiro_tabela", "r"); //abrir ficheiro em modo read
    if (file != NULL) {
        fread(HashTable, sizeof(ht), 1, file);
        fclose(file); //fechar ficheiro enquanto nao é usado
    }

    char in[2]; //Char para input das operacoes

    int idade = HashTable->idade;
    int teste = 0;

    //Verificacao de fim de ficheiro
    while(scanf("%s", in) == 1 && teste != -1) {
        teste = Cestos_Virtuais(HashTable, in, idade);
        idade++;
    }

    HashTable->idade = idade; //actualiza a idade da HashTable

    file = fopen("ficheiro_tabela", "w"); //abrir ficheiro em modo write
    if (file != NULL) {
        fwrite(HashTable, sizeof(ht), 1, file);
        fclose(file); //fechar ficheiro já não é usado
    }

    Free_HashTable(HashTable);
    return 0;
}
```