Projeto ATAD

GeoCaching

Alunos:

- Gabriel Ambrósio, nº 160221013

Turma: Lab 6ª 16H30

Docente: Prof. Aníbal Ponte

Indice

TAD

Para este projeto, de GeoCaching, usei duas TAD's como pedido no enunciado uma TAD SetList, uma variação da TAD List, que não permite duplicados. Para implementar esta TAD usei a estrutura de dados Lista Ligada:

```c
struct node;
typedef struct node* PtNode;

typedef struct node {
    ListElem elem;
    PtNode next;
    PtNode prev;
} Node;


typedef struct listImpl {
    PtNode header;
    PtNode trailer;
    unsigned int size;

} ListImpl;
```

Para garantir que não são adicionados Caches duplicadas do ficheiro, criei uma função equalCache(Cache c1, Cache c2) no ficheiro cache.c:

```c
int equalCache(Cache c1, Cache c2) {
    if ((strcmp(c1.code, c2.code) == 0) && (strcmp(c1.name, c2.name) == 0) && (strcmp(c1.state, c2.state) == 0) && (strcmp(c1.owner, c2.owner) == 0) &&
        return 1;
    }
    return -1;
}
```

Esta função recebe duas caches, e vai comparar todos os atributos de cada cache e verificar se todos são iguais, se forem retorna 1, se não retorna -1;

Depois, alterei ligeiramente a função listAdd(…), antes de criar o node e adicionar o elemento, iterei a lista e usei o método anterior, para verificar se a Cache que estamos a adicionar já existe na lista, se a função retornar 1 então o elemento já existe e não é adicionado.

```c
int listAdd(PtList list, int rank, ListElem elem) {
    if (list == NULL) return LIST_NULL;
    if (rank < 0 || rank > list->size) return LIST_INVALID_RANK;

    //Descobrir que NO ocupa o RANK e inserir o NOVO no ants de NO.
    PtNode nodeAt = nodeAtRank(list, rank);

    for (int i = 0; i < list->size; i++) {
        Cache aux;
        listGet(list, i, &aux);
        if (equalCache(aux, elem) == 1) {
            return -1;
        }
    }

    PtNode newNode = (PtNode)malloc(sizeof(Node));
    if (newNode == NULL) {
        return LIST_NO_MEMORY;
    }

    newNode->elem = elem;
    newNode->next = nodeAt;
    newNode->prev = nodeAt->prev;

    nodeAt->prev->next = newNode;
    nodeAt->prev = newNode;

    list->size++;

    return LIST_OK;
}
```

O resto dos métodos ficaram inalterados, sendo iguais aos usados nas aulas teóricas e laboratoriais.

Função	ED	Complexidade
listCreate()	Lista Ligada	O(1)
listDestroy()	Lista Ligada	O(1)
listAdd()	Lista Ligada	O(n)
listRemove()	Lista Ligada	O(n)
listGet()	Lista Ligada	O(n)
listSet()	Lista Ligada	O(n)
listClear()	Lista Ligada	O(n)
nodeAtRank()	Lista Ligada	O(n)

A segunda TAD pedida foi um Map, que guardava o código da Cache como String (key), e a cache respetiva(value). Para a sua implementação usei uma tabela de dispersão como pedido:

```c
#define MULTIPLIER 7
#define HASHTABLE_SIZE 1951

typedef struct keyValue {
    MapKey key;
    MapValue value;
    int inUse; //para saber se o bucket que a key aponta estaa  ser usado (se sim avança para o prox) (1 - usado, 0 - livre)
} KeyValue;


typedef struct mapImpl {
    KeyValue *elements;
    unsigned int size;
    unsigned int capacity;
} MapImpl;
```

Usei a hash function para Strings disponibilizada nos slides das aulas.

```c
int hashFunction(PtMap map, MapKey key) {

    int length = strlen(key);
    int A = 31415;
    int B = 27183;
    int total = 0;
    for (int i = 0; i < length; i++) {
        total = (total * A + key[i]) % map->capacity;
        A = A * B % (map->capacity - 1);
    }

    return total;
```

Adicionei também uma nova variável chamada inUse à KeyValue, que usei para saber se o bucket para qual a chave aponta, está ou não a ser usado (1 - usado, 0 - livre), se tentar adicionar uma Cache a um bucket e este tiver a variável a 1, existe uma colisão e esta é tratada. Usei a técnica Open Addressing para tratar das colisões:

```c
int mapPut(PtMap map, MapKey key, MapValue value) {
    if (map == NULL) return MAP_NULL;
    if (map->size == map->capacity) return MAP_FULL;


    int bucket = hashFunction(map, key);
    while (map->elements[bucket].inUse == 1) {
        bucket++;
    }

    KeyValue entry;
    entry.inUse = 1;
    strcpy_s(entry.key, sizeof(entry.key), key);
    entry.value = value;

    map->elements[bucket] = entry;

    map->size++;

    return MAP_OK;
}
```

Na função mapPut(…), primeiro é usada a hash function para determinar o bucket em que vai ficar o KeyValue, mas antes de adicionar, verifica se este já esta a ser usado, e avança de bucket até encontrar um vazio, por fim adiciona o KeyValue ao Map.

```
120  int mapRemove(PtMap map, MapKey key, MapValue *ptValue) {
121      if (map == NULL) return MAP_NULL;
122      if (map->size == 0) return MAP_EMPTY;
123
124      int index = hashFunction(map, key);
125
126      for (unsigned int i = index; i < map->capacity; i++) {
127          if (strcmp(map->elements[index].key, key) == 0) {
128              *ptValue = map->elements[index].value;
129              map->elements[index].inUse = 0;
130
131              map->size--;
132              return MAP_OK;
133          }
134          index++;
135      }
136
137      return MAP_UNKNOWN_KEY;
138  }
```

A função mapRemove(…), foi também alterada, primeiro descobrimos qual o bucket em que o keyValue, supostamente está, se a key no bucket for igual a recebida por parâmetro é porque o KeyValue foi encontrado e pode ser removido, senão avança de bucket até o encontrar, pois este foi colocado num dos buckets seguintes (Open Addressing).

```
152  int mapGet(PtMap map, MapKey key, MapValue *ptValue) {
153      if (map == NULL) return MAP_NULL;
154      if (map->size == 0) return MAP_EMPTY;
155
156      int bucket = hashFunction(map, key);
157
158      for (unsigned int i = bucket; i < map->capacity; i++) {
159          if (map->elements[bucket].inUse == 1) {
160              if (strcmp(map->elements[bucket].key, key) == 0) {
161                  *ptValue = map->elements[bucket].value;
162                  return MAP_OK;
163              }
164              bucket++;
165          }
166          else {
167              return MAP_UNKNOWN_KEY;
168          }
169      }
170
171      return MAP_UNKNOWN_KEY;
172  }
```

A função mapGet(…) funciona de maneira semelhante à função anterior, mas não remove o elemento, e apenas o devolve por parâmetro.

Função	ED	Complexidade
mapCreate()	Tabela de Dispersão	O(1)
mapDestroy()	Tabela de Dispersão	O(1)
mapPut()	Tabela de Dispersão	O(n)
mapRemove()	Tabela de Dispersão	O(n)
mapGet()	Tabela de Dispersão	O(n)
mapContains()	Tabela de Dispersão	O(n)
mapClear()	Tabela de Dispersão	O(n)
hashFunction()	Tabela de Dispersão	O(n)

Comandos

O primeiro comando pedido para explicar no enunciado é o **clear(PtList list, PtMap map)**, que limpa uma List e um Map recebidos por parâmetro, o código é bastante simples, se a List e o Map tiverem conteúdo, são chamadas as funções clear de cada TAD:

```
310  void clear(PtList list, PtMap map) {
311      if (listIsEmpty(list) == 1 || mapIsEmpty(map) == 1) {
312          printf("NAO EXISTEM CACHES PARA LIMPAR!");
313          return;
314      }
315      listClear(list);
316      mapClear(map);
317      printf("CACHES IMPORTADAS LIMPAS");
318  }
```

Depois a função **foundr(PtList list)**, que pretende mostrar a percentagem de vezes que cada Cache foi encontrada. Primeiro verifico se a list passada por parâmetro não esta vazia. Depois vou percorre-la e contar o número total de vezes que foram encontradas caches. Por fim percorro novamente a list e vou calcular a percentagem, usando uma regra 3 simples e represento na consola.

```
320  void foundr(PtList list) {
321      if (listIsEmpty(list) == 1) {
322          printf("LIST VAZIA!");
323          return;
324      }
325
326      int size;
327      listSize(list, &size);
328      int totalFounds = 0;
329
330      for (int i = 0; i < size; i++){
331          Cache cache;
332          listGet(list, i, &cache);
333          totalFounds += cache.founds;
334      }
335
336      for (int i = 0; i < size; i++) {
337          Cache cache;
338          listGet(list, i, &cache);
339
340          printCache(&cache);
341          double percentage = (cache.founds * 100.0) / totalFounds;
342          printf("ENCONTRADA %d VEZES\nPERCENTAGEM: %.2f %%\n", cache.founds, percentage);
343      }
344  }
```

Complexidade algorítmica – O(n)

Em terceiro a função **center(PtList list)**, que pede para mostrar 4 estatísticas , a média e desvio padrão da latitude e longitude das caches.

```
346  void center(PtList list) {
347      if (listIsEmpty(list) == 1) {
348          printf("LISTA VAZIA!");
349          return;
350      }
351
352      int size;
353      listSize(list, &size);
354
355      double latitude = 0, longitude = 0, latitudeDP = 0, longitudeDP = 0;
356
357      latitudeLongitudeMean(list, &latitude, &longitude);
358
359      dpMean(list, latitude, longitude, &latitudeDP, &longitudeDP);
360
361      printf("Media/Desvio Latitude: %f / %.2f\nMedia/Desvio Longitude: %f / %.2f", latitude, latitudeDP, longitude, longitudeDP);
362  }
```

Complexidade algorítmica – O(n)

Para a conseguir concluir esta função, criei duas outras, a latitudeLongitudeMean(…) e dpMean(…) para calcular respetivamente as médias e os desvios.

```
void latitudeLongitudeMean(PtList list, double *latitude, double *longitude) {
    int size;
    listSize(list, &size);

    double latitudeMean = 0, longitudeMean = 0;

    for (int i = 0; i < size; i++) {
        Cache cache;
        listGet(list, i, &cache);
        latitudeMean += cache.latitude;
        longitudeMean += cache.longitude;
    }

    *latitude = latitudeMean / size;
    *longitude = longitudeMean / size;
}
```

Complexidade algorítmica – O(n)

Calculado as medias das latitudes e longitudes, e passadas por parâmetro.

```
void dpMean(PtList list, double latitude, double longitude, double *latitudeDp, double *longitudeDp) {
    int size;
    listSize(list, &size);

    double latitudeDP = 0, longitudeDP = 0;

    for (int i = 0; i < size; i++) {
        Cache cache;
        listGet(list, i, &cache);
        latitudeDP += pow((cache.latitude - latitude), 2);
        longitudeDP += pow((cache.longitude - longitude), 2);
    }
    *latitudeDp = sqrt((latitudeDP / size));
    *longitudeDp = sqrt((longitudeDP / size));
}
```

Complexidade algorítmica – O(n)

Calcular os desvios, usando as medias calculadas anteriormente. Usei a fórmula do desvio padrão:

$$DP = \sqrt{\frac{\sum_{i=1}^{n}(x_i - M_A)^2}{n}}$$

De seguida a função **sort(PtList list)**, que vai ordenar a list recebida por parâmetro, de uma de três formas disponíveis, escolhida pelo utilizador. Ordenar por dono (A-Z) e desempatar pelo número de favoritos, por altitude (decrescente) ou pela distância euclidiana relativa à média da latitude e longitude.

```
void sort(PtList list) {
    if (listIsEmpty(list) == 1) {
        printf("LISTA VAZIA!");
        return;
    }

    int order = -1;
    PtList ordered = listCreate();

    int size;
    listSize(list, &size);

    for (int i = 0; i < size; i ++) {
        Cache c;
        listGet(list, i, &c);
        listAdd(ordered, i, c);
    }

    while (order != 1 && order != 2 && order != 3) {
        printf("Ordem:\n1 - Dono\n2 - Altitude\n3 - ");
        printf("Option> ");
        scanf_s("%d", &order);
    }
    if (order == 1) {
        sortOwner(ordered);

        listPrint(ordered);
    }
    else if (order == 2) {
        sortAltitude(ordered);

        int ordSize;
        listSize(ordered, &ordSize);

        for (int i = ordSize; i >= 0; i--) {
            Cache c;
            listGet(ordered, i, &c);

            if (c.altitude < 0) {
                listRemove(ordered, i, &c);
            }
        }

        listPrint(ordered);
    }
    else if(order == 3){
        double latitudeMean = 0, longitudeMean = 0;

        latitudeLongitudeMean(ordered, &latitudeMean, &longitudeMean);
        sortDistancia(ordered, latitudeMean, longitudeMean);

        listPrint(ordered);
    }
    else {
        printf("Invalid Input...");
        return;
    }
}
```

Complexidade algorítmica – $O(n^2)$

Criei 3 funções para ajudar a estruturação da função principal:

- A primeira sortOwner(…):

```
175  void sortOwner(PtList orderedList) {//ordenar por dono
176      int size;
177      listSize(orderedList, &size);
178      for (int i = 0; i < size ; i++) {
179
180          int indexMin = i;
181
182          for (int j = i; j < size; j++) {
183
184              Cache a, indexMinC;
185              listGet(orderedList, j, &a);
186              listGet(orderedList, indexMin, &indexMinC);
187
188              if (strcmp(a.owner, indexMinC.owner) == 0) {//desempate por favoritos
189                  if (a.favourites <  indexMinC.favourites) {
190                      indexMin = j;
191                  }
192              }else if (strcmp(a.owner, indexMinC.owner) < 0) {
193                  indexMin = j;
194              }
195          }
196          swap(orderedList, i, indexMin);
197      }
198  }
```

Complexidade algorítmica – $O(n^2)$

Que usa o princípio de selection sort, vai percorrer a list e organiza-la alfabeticamente no atributo owner, se existir o mesmo dono mais do que uma vez, o desempate é feito pelos favoritos.

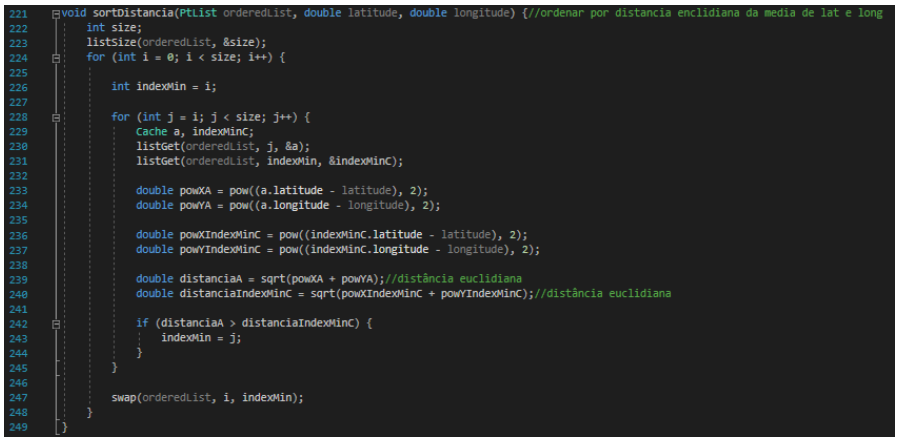
- sortAltitude(…):

```
200  void sortAltitude(PtList orderedList) {//ordenar por altitude
201      int size;
202      listSize(orderedList, &size);
203      for (int i = 0; i < size; i++) {
204
205          int indexMin = i;
206
207          for (int j = i; j < size; j++) {
208              Cache a, indexMinC;
209              listGet(orderedList, j, &a);
210              listGet(orderedList, indexMin, &indexMinC);
211
212              if (a.altitude > indexMinC.altitude) {
213                  indexMin = j;
214              }
215          }
216
217          swap(orderedList, i, indexMin);
218      }
219  }
```

Complexidade algorítmica – $O(n^2)$

Usa o mesmo princípio da anterior, no enunciado é pedido para ignorar as Caches que tem uma altitude desconhecida, eu decidi retirar as altitudes negativas (desconhecidas) da lista após a ordenar na função principal.

- sortDistancia(…)



```
void sortDistancia(PtList orderedList, double latitude, double longitude) {//ordenar por distancia enclidiana da media de lat e long
    int size;
    listSize(orderedList, &size);
    for (int i = 0; i < size; i++) {

        int indexMin = i;

        for (int j = i; j < size; j++) {
            Cache a, indexMinC;
            listGet(orderedList, j, &a);
            listGet(orderedList, indexMin, &indexMinC);

            double powXA = pow((a.latitude - latitude), 2);
            double powYA = pow((a.longitude - longitude), 2);

            double powXIndexMinC = pow((indexMinC.latitude - latitude), 2);
            double powYIndexMinC = pow((indexMinC.longitude - longitude), 2);

            double distanciaA = sqrt(powXA + powYA);//distância euclidiana
            double distanciaIndexMinC = sqrt(powXIndexMinC + powYIndexMinC);//distância euclidiana

            if (distanciaA > distanciaIndexMinC) {
                indexMin = j;
            }
        }

        swap(orderedList, i, indexMin);
    }
}
```

Complexidade algorítmica – $O(n^2)$

Esta função vai calcular a distância euclidiana de cada cache às médias e ordena-las decrescentemente. Usei a fórmula para a distância bidimensional:

$$\sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.$$

Depois a função **dates(PtList list)** que mostra na consola a cache mais antiga e mais recente, e a diferença em meses entre elas:

```
void dates(PtList list) {
    if (listIsEmpty(list) == 1) {
        printf("LISTA VAZIA!");
        return;
    }

    int size;
    listSize(list, &size);

    Cache oldest, newest;
    listGet(list, 0, &oldest);
    listGet(list, 0, &newest);

    for (int i = 0; i < size; i++) {
        Cache aux;
        listGet(list, i, &aux);
        if (aux.hidden_date.year > newest.hidden_date.year) {//verifica o ano
            newest = aux;
        }
        else if (aux.hidden_date.year = newest.hidden_date.year) {//se for o mesmo
            if (aux.hidden_date.month > newest.hidden_date.month) {//verifica o mes
                newest = aux;
            }
        }
        if (aux.hidden_date.year < oldest.hidden_date.year) {//verifica o ano
            oldest = aux;
        }
        else if (aux.hidden_date.year = oldest.hidden_date.year) {//se for o mesmo
            if (aux.hidden_date.year < oldest.hidden_date.year) {//verifica o mes
                oldest = aux;
            }
        }
    }
    printf("Cache Mais Antiga: %d/%d/%d\n", oldest.hidden_date.day, oldest.hidden_date.month, oldest.hidden_date.year);
    printf("Cache Mais Nova: %d/%d/%d\n", newest.hidden_date.day, newest.hidden_date.month, newest.hidden_date.year);
    printf("Diferenca Em Meses: %d\n", ((newest.hidden_date.year - oldest.hidden_date.year) * 12) + newest.hidden_date.month - oldest.hidden_date.month);
}
```

Complexidade algorítmica – O(n)

Nesta função percorro a list e guardo ao mesmo tempo a cache mais antiga e a mais recente, a medida que percorro a lista. Depois de ter as duas, cálculo a diferença entre as duas e apresento na consola.

Outra função pedida é a **sizes(Ptlist list)** que pede o número de caches que tenham o mesmo tamanho, sem poder assumir que apenas existem os tamanhos no ficheiro disponibilizado:

```
461  void sizes(PtList list) {
462      if (listIsEmpty(list) == 1) {
463          printf("LISTA VAZIA!");
464          return;
465      }
466
467      int size;
468      listSize(list, &size);
469
470      char **sizes;
471      sizes = (char**)malloc(sizeof(char*) *  1000);
472      int sizesSize = 0;
473
474      for (int i = 0; i < size; i++) {
475
476          int number = 0;
477          Cache cache;
478          listGet(list, i, &cache);
479
480          if (checkSize(sizes, cache.size, sizesSize) == 1) {//verifica se o size que vamos contar, ja foi contado, se foi passa a prox
481              continue;
482          }
483
484          for (int f = 0; f < size; f++) {
485              Cache aux;
486              listGet(list, f, &aux);
487              if (strcmp(cache.size, aux.size) == 0) {
488                  number++;
489              }
490          }
491
492          int strLen = strlen(cache.size);
493          sizes[sizesSize] = (char*)malloc(sizeof(char) * (strLen + 1));
494          strcpy_s(sizes[sizesSize], (strLen + 1), cache.size);
495
496          sizesSize++;
497          printf("Number Of %s Size Caches: %d\n", cache.size, number);
498      }
499
500  }
```

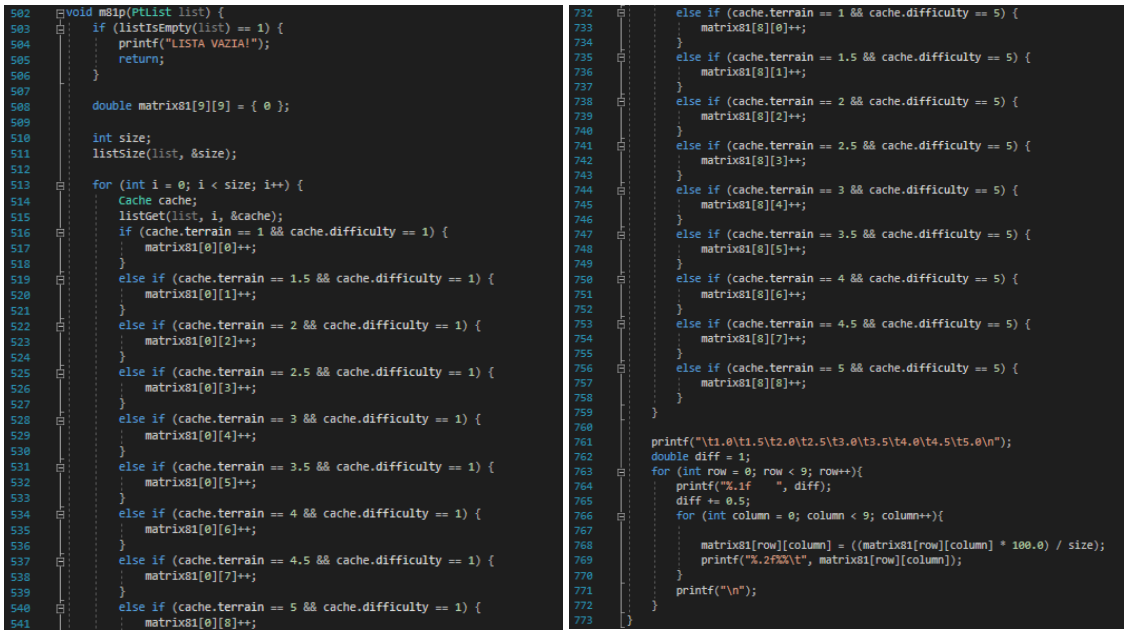
Complexidade algorítmica – O(n)

Para resolver este problema, criei dentro da função um array dinâmico de Strings que vai armazenar o nome de um tamanho após o contar, antes de o fazer verifica se o tamanho já está no array, se já estiver é porque já foi contado e salta para a próxima cache. Assim a função consegue contar qualquer tamanho que apareça no ficheiro uma só vez. Criei a função checkSIzes(…) que vai fazer a verificação.

```
251  int checkSize(char **sizes, char *size, int arraySize) {//verifica se a string size esta no array sizes
252      if (arraySize == 0) return -1;
253      for (int i = 0; i < arraySize; i++) {
254          if (strcmp(sizes[i], size) == 0) {
255              return 1;
256          }
257      }
258      return -1;
259  }
```

Complexidade algorítmica – O(n)

A penúltima função é a **m81p(PtList list)**, que pretende mostrar uma matriz 9x9 com o numero de caches para cada combinação terreno/dificuldade.



```
void m81p(PtList list) {
    if (listIsEmpty(list) == 1) {
        printf("LISTA VAZIA!");
        return;
    }

    double matrix81[9][9] = { 0 };

    int size;
    listSize(list, &size);

    for (int i = 0; i < size; i++) {
        Cache cache;
        listGet(list, i, &cache);
        if (cache.terrain == 1 && cache.difficulty == 1) {
            matrix81[0][0]++;
        }
        else if (cache.terrain == 1.5 && cache.difficulty == 1) {
            matrix81[0][1]++;
        }
        else if (cache.terrain == 2 && cache.difficulty == 1) {
            matrix81[0][2]++;
        }
        else if (cache.terrain == 2.5 && cache.difficulty == 1) {
            matrix81[0][3]++;
        }
        else if (cache.terrain == 3 && cache.difficulty == 1) {
            matrix81[0][4]++;
        }
        else if (cache.terrain == 3.5 && cache.difficulty == 1) {
            matrix81[0][5]++;
        }
        else if (cache.terrain == 4 && cache.difficulty == 1) {
            matrix81[0][6]++;
        }
        else if (cache.terrain == 4.5 && cache.difficulty == 1) {
            matrix81[0][7]++;
        }
        else if (cache.terrain == 5 && cache.difficulty == 1) {
            matrix81[0][8]++;
        }
        else if (cache.terrain == 1 && cache.difficulty == 5) {
            matrix81[8][0]++;
        }
        else if (cache.terrain == 1.5 && cache.difficulty == 5) {
            matrix81[8][1]++;
        }
        else if (cache.terrain == 2 && cache.difficulty == 5) {
            matrix81[8][2]++;
        }
        else if (cache.terrain == 2.5 && cache.difficulty == 5) {
            matrix81[8][3]++;
        }
        else if (cache.terrain == 3 && cache.difficulty == 5) {
            matrix81[8][4]++;
        }
        else if (cache.terrain == 3.5 && cache.difficulty == 5) {
            matrix81[8][5]++;
        }
        else if (cache.terrain == 4 && cache.difficulty == 5) {
            matrix81[8][6]++;
        }
        else if (cache.terrain == 4.5 && cache.difficulty == 5) {
            matrix81[8][7]++;
        }
        else if (cache.terrain == 5 && cache.difficulty == 5) {
            matrix81[8][8]++;
        }
    }

    printf("\t1.0\t1.5\t2.0\t2.5\t3.0\t3.5\t4.0\t4.5\t5.0\n");
    double diff = 1;
    for (int row = 0; row < 9; row++){
        printf("%.1f    ", diff);
        diff += 0.5;
        for (int column = 0; column < 9; column++){

            matrix81[row][column] = ((matrix81[row][column] * 100.0) / size);
            printf("%.2f%%\t", matrix81[row][column]);
        }
        printf("\n");
    }
}
```

Complexidade algorítmica – O(n)

Este método é bastante grande (muitas linhas de código), mas funciona como pretendido.

Percorre a list e verifica qual são os valores do terreno e dificuldade, dependendo de quais forem adiciona um valor à posição correspondente da matriz, depois quando é mostrado na consola, é feita a percentagem.

O ultimo método pedido é o **search(PtMap map)** que possibilita ao utilizador pesquisar uma cache mediante de uma key, este método é também bastante simples. Começa por pedir ao utilizador a chave, e usa a função mapGet(…), se esta existir mostra a na consola, se não diz que não existe.

```
void search(PtMap map) {
    if (mapIsEmpty(map) == 1) {
        printf("MAP VAZIO!");
        return;
    }

    char key[11];
    printf("Code Of Cache:\n");
    printf("Option>");
    scanf_s("%[^\n]", key, 11);
    printf("\n");
    Cache c;
    if (mapGet(map, key, &c) == 5) {
        printf("Cache nao encontrada!");
    }
    else {
        printCache(&c);
    }
}
```

Complexidade algorítmica – O(n)

Limitações

No desenvolvimento deste projeto, não tive muitas dificuldades, desde a criação de estruturas e as suas funções até à implementação da TAD SetList, tive certas dificuldades quando comecei a implementar a TAD Map, usando tabelas de dispersão, mas acabei por conseguir implementa-la de forma a que funcionasse, em todo o projeto, apenas não consigo utilizar a função mapDestroy() (apesar de ter utilizado a implementação fornecida e não a ter alterado) quando o utilizador decide fechar a aplicação. No debugging, percebi que este funcionava, quando não fazia a incrementação do bucket no método mapPut() (tem um comentário a mostrar o que faz com que não funcione), isto pode estar a alterar o valor do ponteiro, e quando tento fazer free() este rebenta, mas não consegui resolver este pequeno problema. De resto o projeto está a funcionar como pretendido.

Conclusões

Para concluir, o projeto foi feito sem problemas muito grandes, pois foi feito sobre a matéria estudada durante o semestre, e foi também, disponibilizado muitas ferramentas para o poder fazer, deste a implementação de TADs e estruturas, à leitura de ficheiros, à complexidade algorítmica das funções. Como já foi referido o projeto está de acordo com o pedido no enunciado, foram usadas duas TADs (SetList e Map) implementadas como pedido (tabela de dispersão), e os comandos principais foram também implementados na sua plenitude, funcionando como esperado.