Aula prática 11

Esta aula tem como objetivo rever algumas das estruturas de dados apresentadas anteriormente, nomeadamente: filas de prioridade, árvore AVL e grafo.

1 Pretende-se implementar um programa que dado um dicionário de palavras e um texto identifique as palavras no texto que não constam do dicionário. Deverá implementar as funções indicadas nas alíneas seguintes no ficheiro `dict.c`.

1.1 Implemente uma função que leia um dicionário a partir de um ficheiro `f` e o represente através de uma árvore binária de pesquisa AVL `dict`:

```
int le_dicionario(FILE *f, arvore_avl *dict)
```

Todas as palavras lidas devem ser convertidas em minúsculas usando a função `downcase` disponibilizada no ficheiro `dict.c`. Considera-se que uma palavra é uma sequência de caracteres sem espaços. A função deve retornar o numero de palavras lidas.

1.2 Implemente uma função que ajude a percorrer uma *string* de palavra em palavra:

```
char * isola_palavra(char *str, int *len)
```

A função deverá receber uma *string* `str` (que deverá ficar intacta durante o processo) e retornar um apontador para o início da primeira palavra na *string* ou NULL caso não existam palavras na *string*. No caso de existir pelo menos uma palavra, a função deverá também retornar (através do parâmetro `len`) o tamanho da palavra encontrada.

1.3 Implemente uma função que procure numa *string* `str` a primeira palavra não contida em `dict` (i.e., uma palavra inválida).

```
char* verifica_texto(arvore_avl *dict, char *str, int *len)
```

A função deve retornar um apontador para a posição do início de uma palavra inválida em `str` ou NULL caso todas as palavras estejam contidas em `dict`. No caso de uma palavra inválida ser encontrada, a variável `len` deverá ser utilizada para retornar o tamanho da mesma.

Nota: A comparação das palavras deverá ser sempre feita com a versão em letras minúsculas de cada palavra (usando a função `downcase`). A *string* `str` deverá permanecer intacta durante todo o processo. Deverá utilizar a função desenvolvida na alínea 1.2 para percorrer a *string* `str`.

Exemplo:

```
while((str = isola_palavra(str, &comprimento)) != NULL) {
    ... código ...
    str += comprimento;
}
```

Após ter implementado as 3 funções anteriores, o resultado do programa deverá ser:

```
Lidas 52875 palavras
Palavra desconhecida: ola
Palavra desconhecida: adeus
Palavra desconhecida: foox
Palavra desconhecida: barz
Palavra desconhecida: bla
```

2 Pretende-se implementar um programa que faça o escalonamento de eventos com base no ficheiro `eventos.c`. Um evento é definido por uma *string* com o seguinte formato:

```
<Nome> <Duração> <Prioridade> <Tempo Chegada>
```

O processamento dos eventos consiste em 3 fases:

2.1 Leitura dos eventos. Nesta fase deverá ler todos os eventos (segundo o formato indicado anteriormente, um por linha) a partir de um ficheiro `f`, implementando a seguinte função:

```
int le_eventos(FILE *f,  heap *eventos_agendados)
```

Os eventos devem ser colocados numa fila de prioridade (`eventos_agendados`) por ordem de tempo de chegada. Para obter o tempo de chegada pode utilizar a função `pop_number` fornecida. Nesta fila de prioridade a *string* correspondente a cada evento terá o seguinte formato:

```
<Nome> <Duração> <Prioridade>
```

A função deve retornar o número de eventos lidos.

2.2 Processamento de eventos agendados. Nesta fase os eventos com tempo de chegada menor ou igual a "tempo" deverão ser retirados da fila de eventos agendados (`eventos_agendados`) para a fila de eventos em espera (`eventos_espera`), implementando a seguinte função:

```
int processa_tempo(int tempo, heap *eventos_agendados, heap
                                *eventos_espera)
```

Durante a mudança de filas, a prioridade dos eventos deve ser extraída das *strings* dos mesmos utilizando a função `pop_number`. A fila `eventos_espera` terá uma *string* correspondente a cada evento com o seguinte formato:

```
<Nome> <Duração>
```

A função deverá retornar o número de elementos processados.

Nota: Relembre que esta implementação da heap é uma min-heap no entanto pretende-se utilizar a mesma como uma max-heap. Para este efeito poderá utilizar o valor simétrico da prioridade (e.g., "prioridade = 4" passa a "prioridade = -4").

2.3 Execução de eventos em espera. Nesta fase os eventos em espera deverão ser executados de acordo com a sua prioridade , implementando a seguinte função:

```
char* proximo_evento(heap *eventos_espera, int *duracao)
```

Esta função deverá remover o evento com maior prioridade na fila e retornar o seu nome e duração (através do parâmetro `duracao`). Mais uma vez, a duração de cada evento deverá deve ser extraída da *string* utilizando a função `pop_number`.

Após ter implementado as 3 funções anteriores, o resultado do programa deverá ser:

```
Lidos 6 eventos
                              Fazer a ficha da AP11 180 1000  (14)
               Tomar banho 15 10  (0)
                              Ir ao ginasio 120 10  (20)
Fazer a cama 5 0  (0)
                              Ir dormir 480 100  (4000)
               Lavar o carro 30 5  (10)

@0: Fazer a cama  (duracao: 5)
@5: Tomar banho  (duracao: 15)
@20: Lavar o carro  (duracao: 30)
@50: Ir ao ginasio  (duracao: 120)
@170: Fazer a ficha da AP11  (duracao: 180)
@4000: Ir dormir  (duracao: 480)
```

3 Pretende-se implementar um programa com funções para analisar uma árvore AVL, que deverá ser implementado no ficheiro `arvore.c`.

3.1 Implemente a função `guarda_conteudo` que guarda numa lista todos os elementos de uma árvore AVL após uma visita em-ordem à árvore:

```
void guarda_conteudo (arvore_avl *arvore, lista *lst)
```

O primeiro parâmetro da função é o apontador para a árvore AVL e o segundo parâmetro é o apontador para a lista resultante. A árvore original não deve ser alterada. Sugestão: utilize uma função auxiliar recursiva.

Depois de implementada a função, o programa deverá apresentar:

```
Conteudo: dificil e' facil mt2 muito nada prog2 trabalho verdade zzzz...
```

3.2 Implemente a função `descobre_segredo` que retorna uma lista com as palavras que compõem o segredo contido numa árvore AVL. Cada nó da árvore contém uma palavra.

```
lista* descobre_segredo (arvore_avl *arvore, lista *indicacoes)
```

Os parâmetros da função são o apontador para a árvore AVL e para a lista contendo os caminhos a seguir na árvore. A lista `indicacoes` contém apenas as palavras "esquerda" e "direita", que indicam os ramos da árvore a percorrer a partir da raiz, que contém a primeira palavra. A árvore original não deve ser alterada.

Depois de implementada a função, o programa deverá apresentar:

```
Segredo: prog2 mt2 e' facil
```

4 Pretende-se analisar e obter informação sobre rotas de voos de uma determinada companhia aérea num programa a implementar no ficheiro `rotas.c`. O programa é composto por duas funções a implementar nas alíneas seguintes. Em ambas as alíneas é usada uma lista que contém os nomes de todos os aeroportos.

4.1 Implemente a função `proximas_n_chegadas`, que deve imprimir as primeiras n chegadas de aviões a um determinado aeroporto. A função deverá utilizar uma fila de prioridade baseada em ***heap*** para armazenar as chegadas.

```
int proximas_n_chegadas(lista *tempos, lista *origens, lista *aeroportos,
                                                        int n)
```

Os parâmetros da função incluem uma lista dos tempos de chegada e uma lista dos índices dos aeroportos de origem. Note-se que para cada i, na lista `tempos` é indicada a hora de chegada e na lista `origens` está guardado o índice (como *string*) do respetivo aeroporto de origem. São também passados por parâmetro a lista contendo os nomes dos aeroportos e ainda o número *n* de chegadas que se pretende imprimir. A função deve retornar 1 se for bem sucedida e 0 em caso contrário. Considere que no máximo existem 25 chegadas.

Depois de implementada a função, o programa deverá apresentar:

```
1: Faro (PT)
2: Bruxelas Charleroi (BE)
3: Frankfurt-Hahn (DE)
4: Lisboa (PT)
5: Madrid (ES)
```

4.2 Implemente a função `pesquisa_destinos` que determina todos os destinos alcançáveis a partir de um aeroporto com voos direto. Para tal utiliza-se a estrutura **grafo**, em que os vértices correspondem a aeroportos e as arestas a voos diretos entre dois aeroportos.

Note que existem sempre voos de ida e volta, pelo que a implementação se baseia num grafo não-direcionado. O resultado deve ser apresentado numa lista ordenada dos nomes dos aeroportos alcançáveis.

```
lista* pesquisa_destinos(grafo *rotas, lista *aeroportos, char *origem)
```

Os parâmetros da função são o apontador para o grafo e para a lista de aeroportos, bem como o nome do aeroporto de origem a partir do qual se pretende efetuar a pesquisa. Os parâmetros de entrada devem ser verificados.

Depois de implementada a função, o programa deverá apresentar:

```
Destinos diretos a partir de Lille (FR) = 2
Marselha (FR)
Porto (PT)
```