Algoritmia

Aulas práticas

REQUISITOS **6**
Para a escola 6
Para o aluno 6
AULA 01 ..... **7**
Trabalhar com vectores 7
Sentinelas 8
Recursividade 9
AULA 02 ..... **10**
Bubble sort 10
Exercícios 10
AULA 03 ..... **11**
Apontadores para funções 11
Exercícios 11
AULA 04 ..... **12**
Tipos de dados genéricos 12
Exercícios simples 12
Exercício: *High Scores* 12
Exercício: Jogo de proximidade 13
AULA 05 ..... **15**
Shell sort e *Quick Sort* 15
Exercício: Comparação de velocidade 15
AULA 06 ..... **17**
Apontadores e listas 17
Exercício 17
AULA 07 ..... **18**
Pesquisas e inserção a meio 18
Remoção de elementos 19
Outras abordagens à remoção de elementos 20
Outras abordagens: apontadores para apontadores 21
Outras abordagens: recursividade 21
Exercícios 22
AULA 08 ..... **23**
Resolução de exercícios "ao vivo" 23
AULA 09 ..... **24**
Pesquisa binária 24
Exercícios 24
AULA 10 ..... **25**
Árvores 25
Inserir novos elementos numa árvore 26
Elementos iguais e árvores ternárias 27

Apagar elementos de uma árvore 27
Árvores degeneradas e equilibradas 29
Exercícios 29
AULA 11 30
Exercícios 30
AULA 12 31
Ficheiros 31
Binário vs. texto 31
Abordagem para escrita 32
Abordagem para leitura 33
Exercícios 34
AULA 13 35
Tipos de dados variáveis 35
Listas genéricas 37
Exemplo 38
Exercício 38
EXERCÍCIOS 39
BIBLIOGRAFIA 40

Requisitos

Para a escola

Requisitos para as salas das aulas práticas de Algoritmia:

- Windows ou Linux com editor, compilador e biblioteca padrão C instalados. Sugiro Windows com:
  - Dev-C++ existente em http://www.bloodshed.net/devcpp.html,
  - CodeBlocks existente em http://www.codeblocks.org/,
  - Eclipse com o CDT e um compilador de C (em http://www.eclipse.org/cdt/), ou
  - Visual C++ 2008 Express Edition da Microsoft em http://www.microsoft.com/express/vc/.
- Acesso à Internet com um *browser*.

Deve haver 1 PC por aluno.

Cada aula está programada para uma duração de 2h.

Para o aluno

Comparência nas aulas. Este guião tem propositadamente omissos certos elementos importantes para a compreensão total da matéria (notas históricas, relações entre partes diferentes da matéria, avisos sobre erros comuns, etc., ou seja, elementos para uma nota 20), embora seja suficiente para passar com nota bastante acima de 10.

Deves ter instalado o editor e compilador de C em computador próprio se quiseres acompanhar a matéria em casa. Não é no entanto de todo necessário que tenhas estes sistemas em casa para conseguires passar à cadeira (podes usá-los na escola).

Esta cadeira assume que já tens experiência no uso de computadores e sabes C.

Aula 01

Introdução e contextualização: algoritmos e estruturas de dados. O editor e compilador de C. Hiperligações.

Referências e avaliação.

Trabalhar com vectores

Vectores *(arrays*, em Inglês) são conjuntos de elementos que estão guardados sequencialmente em memória.

Descarrega os exemplos da aula de hoje (explicados na aula). Eles exemplificam os seguintes processos:

- Pesquisa: varres o vector do primeiro elemento ao último. Se no meio encontras o item que estavas à procura, paras o processo indicando sucesso. Só indicas que o item não existe no vector depois de varrer todos os elementos.
- Inverter: varres o vector do início até metade. A ideia é trocar cada elemento com o elemento no "lado" oposto do vector. Repara que para efectuar essa troca precisas de uma variável temporária.

Faz agora os seguintes exercícios:

- `procura-2.c` – Pesquisar por múltiplas ocorrências do mesmo item. Altera a pesquisa para mostrar a posição em que o item é encontrado. Se ele for encontrado em múltiplas posições, deves exibi-las todas. Se ele não for encontrado no vector, deve ser exibida uma mensagem especial a indicar essa situação.
- `inverte-2.c` – Faz outro programa de inversão que usa dois vectores, criando uma versão invertida do primeiro vector, no segundo. Como será que podes fazê-lo funcionar?*

* Varres o primeiro vector do inicio ao fim. Cada elemento do primeiro vector é copiado para o segundo, mas na posição oposta. A ultima pergunta é: como calculas a "posição oposta"?

- `mistura.c`. Cria dois vectores de 10 elementos e um terceiro de 20 elementos. Faz o teu programa preencher o 3º vector com os elementos intercalados dos dois primeiros vectores. Exemplo:

```
vector1[10] = { 100, 200, 300, ... };
vector2[10] = {  1,    2,   3, ... };

vector3[20] = { 100, 1, 200, 2, 300, 3, ... };
```

- `mistura-inversa.c`. Repete o exercício anterior, mas onde o 2º vector é misturado no 3º do fim para o início. Exemplo:

```
vector1[10] = { 100, 200, 300, ... };
vector2[10] = {                ... , 8, 9, 10 };

vector3[20] = { 100, 10, 200, 9, 300, 8, ... };
```

Tira as dúvidas com o professor, nas aulas.

Sentinelas

A pesquisa é um processo relativamente comum e que precisa de ser efectuado o mais rapidamente possível. A pesquisa sequencial em vectores que vimos até agora não tem remédio senão repetir-se até se encontrar o que procurávamos, ou repetir até ao fim do vector, dependendo da situação.

Mas podemos melhorar o que acontece em cada iteração (repetição) do ciclo interno da pesquisa.

Até agora precisamos de fazer duas comparações neste ciclo:

- Comparar o elemento do vector onde nos encontramos com o item que estamos à procura,
- Verificar se chegámos ao final do vector para pararmos.

Podemos eliminar uma destas operações (tornando o ciclo mais rápido) se soubermos que o item de que estamos à procura existe sempre. Isto pode-se garantir criando mais um elemento ao final do vector que preenchemos com o item que vamos procurar antes de começar a procura. Assim sabemos que vamos sempre encontrar o item, e podemos eliminar a segunda comparação. Chamamos a este elemento adicional, a "sentinela".

Ou seja passamos disto:

```
for( i=0; i<tam; i++ ) {
      if(vct[i]==n) return 1;
}
```

para:

```
for( i=0; vct[i]!=n; i++ );
```

Para saber se o item que estávamos à procura existia no vector original ou não, só temos de comparar a posição onde ele foi encontrado, com a posição da sentinela. Se for igual, encontrámos a sentinela e exibimos no ecrã "não encontrado".

Como exercício, altera a pesquisa original dos exemplos para funcionar com sentinela (`procura-sentinela.c`).

Recursividade

Recursividade é o que se chama a algoritmos que funcionam fazendo uma função chamar-se a si mesma. Em Matemática diz-se que uma definição é recursiva quando se define recorrendo a si mesma.

Exemplo:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \times (n-1)! & \text{caso contrário} \end{cases}$$

Ou em linguagem C:

```
int factorial( int n )
{
if( n == 0 )
      return 1;
return n * factorial( n-1 );
}
```

Exercícios:

- Cria uma função recursiva que calcula a sequência Fibonacci. A função `main()` deve exibir os primeiro 10 números de Fibonacci. Fibonacci de 0 e 1 são eles mesmos. Acima de 1, o número Fibonacci é a soma dos dois anteriores Fibonacci (`n-1` e `n-2`).
- Cria uma função recursiva que exibe um vector de teste no ecrã (do primeiro elemento para o ultimo)[*].
- Cria uma função recursiva que exibe um vector de teste pela ordem inversa (do ultimo elemento para o primeiro).

Tira as dúvidas com o professor, nas aulas.

[*] Dica: usa uma função `exibe(int vct[], int tam, int posicao_a_exibir)`.

Aula 02

Introdução às ordenações: *Bubble sort*.

Bubble sort

O *bubble sort* ("ordenação por bolhas") é uma forma simples de ordenar vectores que deves ter aprendido nas aulas teóricas.

O exemplo `bubblesort.c` será explicado na aula.

Exercícios

Completa os seguintes exercícios:

- `bubblesort-invertido.c` – Mostra que compreendeste bem como funciona a ordenação, alterando-a para que funcione ao contrário (i.e., que resulte em números ordenados do maior para o mais pequeno).
- Aplica estas duas funções de ordenação no programa `ordenar5.c` cujo esqueleto tens disponível. Testa-o e confirma que funcionam bem.
- Cria um `bubblesort-double.c` que em vez de inteiros, usa `doubles` (números reais) no vector.
- `bubblesort-intercalado.c` – Altera o `bubblesort.c` para só ordenar os elementos das posições pares (0, 2, 4, etc.) de um vector de 20 elementos. Os elementos das posições ímpares (1, 3, 5, etc.) devem ficar onde estão. Exemplo:

  ```
  v[20] = { 2,9,4,0,4,8,2,3,1,1,3,9,2,6,7,5,8,2,9,0 };
  ```

  Depois do `bubblesort_intercalado()`:

  ```
  v[20] = { 1,9,2,0,2,8,2,3,3,1,4,9,4,6,7,5,8,2,9,0 };
  ```

- Cria um programa de ordenação que ordena os elementos das posições pares de forma ascendente e os das posições ímpares de forma descendente.

Tira as dúvidas com o professor, nas aulas.

Aula 03

Apontadores. *Bubble sort* com apontadores.

Apontadores para funções

Vamos fazer a função `bubblesort()` da aula passada independente da função de ordenação. Isto permite implementar facilmente ordenações complexas que não são as meras "ascendente" e "descendente". Vê `bubblesort2-fora-de-ordem.c` que será explicado na aula.

Exercícios

Completa os seguintes exercícios:

- O `bubblesort()` actual já está completamente genérico? De que característica(s) do vector é que ele ainda depende?
- Cria um `bubblesort-double.c` que em vez de inteiros, usa `doubles` (números reais) no vector (isto repete o exercício da aula anterior, para relembrares estes processos num novo código).
- Cria um `bubblesort2-hexa.c` que usa inteiros hexadecimais no vector. Ele deve ordenar o vector unicamente com base no dígito hexadecimal mais à direita. Por exemplo:

  ```
  int v[] = { 0xA7, 0x80, 0x9E, 0x26 };
  ```

  deve ficar ordenado como:

  ```
  int v[] = { 0x80, 0x26, 0xA7, 0x9E };
  ```

  Lembra-te que a forma de "limpar" parte de um número inteiro é com um "e" (`AND`) *bitwise* que se escreve em C "&".
  Exibes hexadecimais no ecrã com `%X` no `printf()`.
- Cria um `bubblesort2-char.c` que em vez de inteiros, usa `chars` (caracteres) no vector. Cada caracter escreve-se em C desta forma: `'c'`. Quero uma ordenação especial: Maiúsculas devem vir primeiro (por ordem alfabética), depois minúsculas, depois números e por fim todos os outros símbolos.

Tira as dúvidas com o professor, nas aulas.

Aula 04

Bubble sort com estruturas e tipos de dados genéricos. Exercícios com integração de conhecimentos anteriores.

Tipos de dados genéricos

Vamos fazer a função `bubblesort()` de duas aulas atrás independente do tipo de dados do vector. Vê `bubblesort-struct.c` que será explicado na aula.

Exercícios simples

Tenta fazer isto:

- Baseado no `bubblesort2-fora-de-ordem.c` da aula anterior e tirando ideias do exemplo de hoje, cria um `bubblesort2-struct.c` que em vez de inteiros, usa a estrutura `struct { char nome[80]; double nota; }` no vector. A ordenação deve ser por nome (dica: a função C `strcmp()` faz comparação de *strings)*.
- Aplica o algoritmo *Shell sort* ao exemplo de hoje, em substituição do *Bubble sort*.

Exercício: *High Scores*

Cria um `highscores.c` que mostra uma tabela de altas pontuações de um jogo. Neste jogo, para cada jogador vais ter o nome, número de inimigos mortos (pontuação), número de poções mágicas bebidas e quantidade dragões seus aliados.

Os jogadores ordenam-se:

- Um dragão aliado vale 10 poções mágicas;
- Jogadores com mais dragões/poções devem ser exibidos antes dos jogadores com menos;
- Jogadores com quantidade idêntica de dragões/poções devem ser ordenados por quantidade de inimigos mortos;
- Jogadores com quantidade idêntica de dragões, poções e inimigos mortos, devem ser ordenados alfabeticamente (tabela ASCII).

Assume os seguintes dados de teste (nome, inimigos mortos, poções, dragões):

```
"Grey Lord", 435, 2, 0
"Zas", 6034, 15, 2
"Lady Dragon", 6856, 29, 1
"Dragon Charm", 15907, 4, 6
"Swordmaster", 37002, 0, 0
"Lady Marion", 6856, 19, 2
"Gandalf", 8906, 38, 0
```

Podes criar mais dados de teste, se preferires.

Exercício: Jogo de proximidade

Cria um `proximidade.c` que será um jogo de proximidade de número. Este jogo joga-se obrigatoriamente com 2 ou mais jogadores.

O teu programa deve ao início pedir o número de jogadores, e "calcular" um número aleatório entre 1 e 100. De seguida repete sempre:

- Pede um número a cada jogador;
- Depois de todos terem introduzido os seus números, deve dizer se alguém acertou, e nesse caso termina o programa;
- Se ninguém acertou deve em vez disso mostrar os jogadores por ordem crescente de proximidade e repetir o processo.

Este jogo torna-se mais interessante com mais jogadores porque se torna difícil lembrarmo-nos dos valores que cada jogador sugere.

Este exercício deve ajudar-te a exercitar:

- Varrer um vector a pedir dados para o mesmo (preencher o vector);
- Pesquisar por múltiplas ocorrências de um valor (o número aleatório gerado) dentro de um vector (vê aula anterior);
- *Bubble sort*.

Dica: geras um número inteiro aleatório de 1 a 100 com:

```
#include <time.h>
#include <stdlib.h>

srand( (unsigned int) time(NULL) );

#if RAND_MAX < 100
numero = 1 + rand()*99 / RAND_MAX;
#else
numero = 1 + (rand() % 100);
#endif
```

Dica: pedes um inteiro ao utilizador com:

```
int num;
scanf( "%d", &num );
```

Tira as dúvidas com o professor, nas aulas.

Aula 05

Shell sort e *Quick sort*. Comparação de velocidade dos três principais algoritmos de ordenação.

Shell sort e *Quick Sort*

Estes programas vão ser explicados na aula.

Exercício: Comparação de velocidade

Cria um programa que cria dois vectores `v1` e `v2` não inicializados com `MAX` inteiros. Depois deve preencher o `v1` com números aleatórios de `0` a `RAND_MAX`.

Em seguida deve copiar `v1` para `v2` e ordenar `v2` com o *Quick sort*. Deve medir o tempo que leva a ordenar. Isto faz-se com:

```
#include <string.h>
#include <time.h>

/* copia v1 para v2 */
memcpy( v2, v1, sizeof(v1) );

/* corre a ordenacao, medindo o tempo que demora */
t_inicial = time( NULL );
quicksort( ... );
t_final = time( NULL );
duracao_em_segundos = t_final - t_inicial + 1;
printf( "...", duracao_em_segundos );
```

Repete o processo com os algoritmos *Shell sort* e *Bubble sort*. Estes devem estar todos no mesmo programa e correr uns a seguir aos outros. Se algum demorar apenas 1 segundo, vai aumentando o `MAX` até que todos levem vários segundos e ser possível uma comparação razoável.

Espera-se um *output* semelhante a:

```
Tempo demorado com Quick sort:  2s
Tempo demorado com Shell sort:  2s
Tempo demorado com Bubble sort: 2s
```

Obviamente que os tempos aqui exibidos estão incorrectos.

Responde às seguintes questões:

- Se duplicar o tamanho do vector, o tempo que leva ao *Bubble sort* ordenar também duplica?
- O que dá mais ganhos a uma aplicação: algoritmos mais inteligentes ou pedir hardware mais rápido?

Aula 06

Exemplos e exercícios de apontadores e listas.

Apontadores e listas

Os apontadores são essenciais em C e são usados frequentemente dentro das recentes linguagens de programação (Java, C#, etc.), embora não estejam disponíveis para o programador.

Para os entenderes, presta atenção à explicação destes programas na aula, por esta ordem:

- `apontadores.c`
- `troca.c`
- `memoria.c`

Os apontadores são então usados para se criarem estruturas dinâmicas que crescem e encolhem de acordo com as necessidades do programa. As mais simples dessas estruturas chamam-se de "listas".

As linguagens mais recentes implementam as listas usando apontadores internos, da forma que vos vamos mostrar.

Presta agora atenção aos seguintes programas:

- `lista.c`
- `lista-cabeca.c`

Exercício

Cria um programa semelhante ao `lista-cabeca.c`, mas que adiciona os novos elementos à cauda (ao fim da lista).

Que tipo de estrutura, já mencionada em Arquitectura de Computadores, é implementada com o `lista-cabeca.c` original? E que tipo de estrutura achas que será implementada com este `lista-cauda.c`?

Acrescenta a funcionalidade de fazer uma pesquisa a um destes programas.

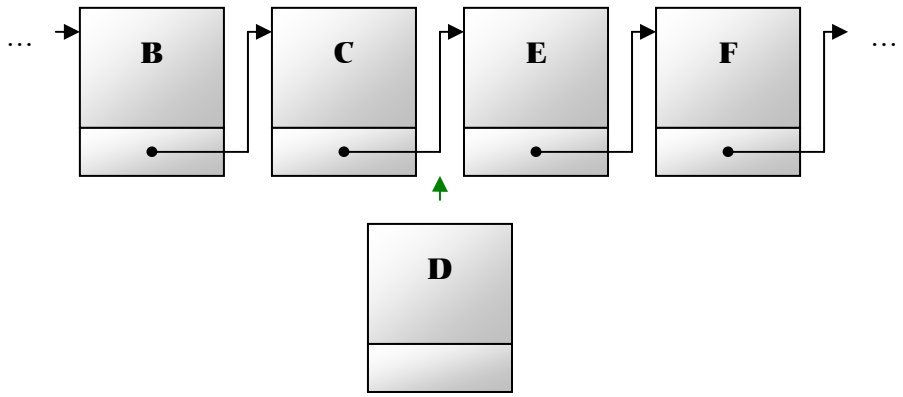
Tira todas as dúvidas com o professor. Vamos trabalhar com apontadores até ao final do semestre, com estruturas cada vez mais complexas. Não deixes nenhuma dúvida para trás!

Aula 07

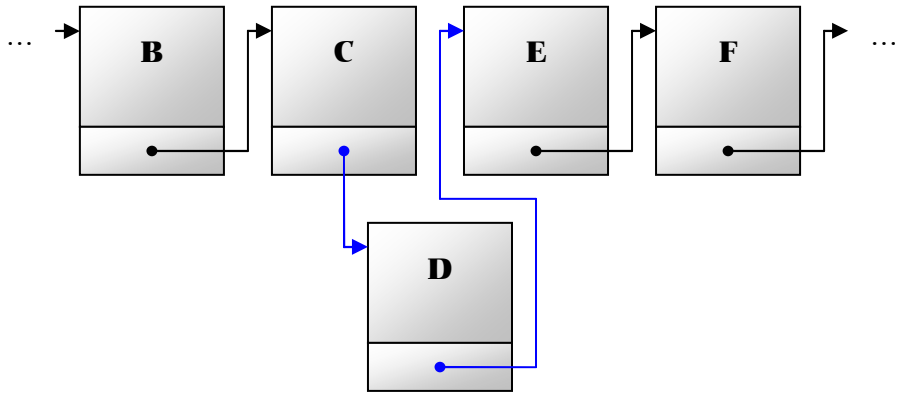
Inserção de elementos a meio de uma lista. Remoção de elementos.

Pesquisas e inserção a meio

Para inserir um elemento a meio de uma lista temos de resolver o seguinte problema:



Esquematicamente a solução é obvia:



onde as setas a azul representam as alterações à lista que temos de efectuar para inserir este elemento.

Isto significa que para a inserção a meio precisamos de ter algum tipo de código de pesquisa, de forma a encontrar o elemento **C** a seguir ao qual desejamos adicionar o novo elemento **D**.

Fazemos isto com o seguinte código:

```
elemento *p, *novo;
for( p = primeiro;  p != NULL;  p = p->proximo )
       {
       if( (*p)-é-o-que-quero )
             break;
       }
```

Este código tem como base os nomes das variáveis que vimos nos programas `lista.c` e `lista-cabeca.c` que vimos na aula anterior. Aqui, `(*p)-é-o-que-quero` é código que depende da aplicação e dos critérios de pesquisa que irás usar.

Se tudo correr bem, quando o ciclo terminar temos em `p` `NULL` (se não encontrámos o elemento), ou o endereço da primeira ocorrência desse elemento.

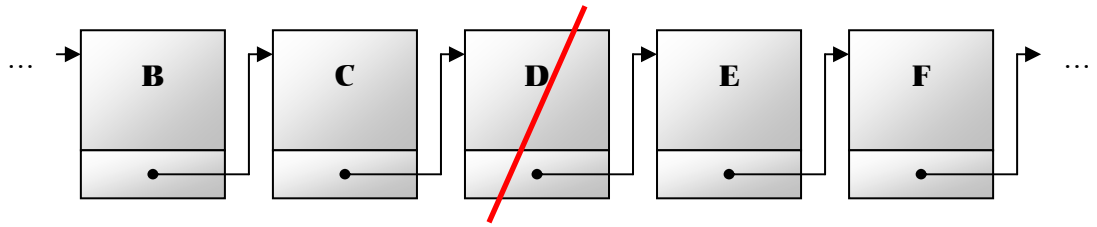
Neste ultimo caso, e assumindo que temos na variável `novo` o endereço do novo elemento a adicionar, fazemos simplesmente:

```
novo->proximo = p->proximo;
p->proximo = novo;
```

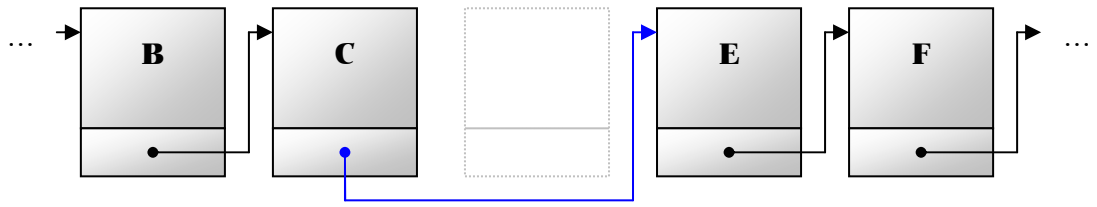
E fica adicionado.

Remoção de elementos

O problema é semelhante ao anterior, pretendendo-se agora a remoção de um elemento:



A solução é igualmente esquematicamente simples:



não esquecendo de libertar (`free()`) a memória ocupada pelo elemento **D**.

Em código vamos mais uma vez precisar de encontrar o elemento a apagar pelo que vamos necessitar de fazer uma pesquisa. Isto não seria necessário se estivéssemos a remover o primeiro ou último elemento da lista. No entanto desta vez temos de parar o ciclo com `p` a apontar para o elemento **anterior** ao elemento que queremos apagar. Por esta razão, temos de verificar o caso especial de não haver ninguém atrás do elemento que queremos apagar.

Ou seja:

```
elemento *p, *temp;
if( primeiro != NULL )
      {
      if( (*primeiro)-é-o-que-quero )
            apagar-à-cabeça;
      else
            {
            for( p = primeiro;  p->proximo != NULL;
                 p = p->proximo )
                 {
                 if( (*(p->proximo))-é-o-que-quero )
                       break;
                 }
            if( p->proximo != NULL )
                 trocar-apontadores;
            }
      }
```

O bloco `trocar-apontadores` terá o seguinte código:

```
temp = p->proximo;
p->proximo = p->proximo->proximo;
free( temp );
```

O bloco `apagar-à-cabeça` fará parte do exercício proposto ao final desta aula.

Outras abordagens à remoção de elementos

É perfeitamente possível resolver estes problemas com outras abordagens. Vamos ver a questão de remover um elemento a meio de uma lista que foi o mais complexo até agora.

O problema está em encontrar o elemento "anterior" ao que vou apagar. Porque não vou guardando esse elemento anterior numa variável temporária? Dessa forma a pesquisa seria mais simples porque preciso de encontrar o elemento que vou apagar, não o anterior a esse.

O código seria o seguinte:

```
elemento *p, *anterior;
if( primeiro != NULL )
      {
      anterior = NULL;
      for( p = primeiro;  p != NULL;  p = p->proximo )
            {
            if( (*p)-é-o-que-quero )
                  break;
            anterior = p;
            }
```

```
        if( p != NULL )
                {
                if( anterior == NULL )
                        apagar-à-cabeça;
                else
                        {
                        anterior->proximo = p->proximo;
                        free( p );
                        }
                }
        }
```

Outras abordagens: apontadores para apontadores

Podemos ainda pensar noutra simplificação. No código acima separámos o caso de apagar à cabeça de apagar no resto da lista, mas a única diferença desses casos é o apontador em que escrevemos (`primeiro` ou `...->proximo`).

Podemos então usar apontadores para apontadores para generalizar as coisas:

```
elemento *p, **apontador;
apontador = &primeiro;
for( p = primeiro;  p != NULL;  p = p->proximo )
        {
        if( (*p)-é-o-que-quero )
                break;
        apontador = &(p->proximo);
        }
if( p != NULL )
        {
        *apontador = p->proximo;
        free( p );
        }
```

Repara que todo o código passou também a funcionar mesmo com listas vazias.

Outras abordagens: recursividade

Há ainda outra solução, envolvendo funções recursivas. Ela **não é tão rápida** quanto a solução anterior, e **não funciona com listas grandes**, mas é elegante de se ler e serve para te mostrar ainda outra forma de se resolver as coisas.

Nas duas soluções anteriores estamos a guardar algum tipo de referência ao elemento "anterior" ao actual para o alterar e depois avançamos para o elemento seguinte à procura do elemento que queremos apagar.

Porque não pensamos de outra forma? Pensa só do ponto de vista de um apontador: devo chamar uma função que irá tratar do problema de apagar o elemento para onde ele aponta (se for o caso) e que me diz qual o novo valor que devo colocar neste apontador.

E se esse elemento não for o elemento a apagar? Não há problema. Repara que esse elemento terá um apontador para o elemento seguinte pelo que repito então com esse apontador o raciocínio já explicado.

O código fica assim:

```
elemento *apaga( elemento *p )
{
       elemento * temp;

       if( p == NULL )
            return NULL;
       if( (*p)-é-o-que-quero )
            {
            temp = p->proximo;
            free( p );
            return temp;
            }
       p->proximo = apaga( p->proximo );
       return p;
}

/* chamo esta funcao com: */
primeiro = apaga( primeiro );
```

Tira as dúvidas com o professor, nas aulas.

Exercícios

Resolve os seguintes problemas na aula:

1. Para que serve a variável `temp` nos exemplos acima?
2. Cria código que remove elementos à cabeça de uma lista e à cauda de uma lista. Altera o `lista-cabeca.c` da aula anterior para antes de terminar (após o utilizador escrever `0`), apagar o primeiro e o último elementos da lista, antes de exibir a lista.
3. Cria um programa que exibe um menu com várias opções para inserir elementos à cabeça e à cauda, para remover à cabeça e à cauda, para pesquisar elementos e para exibir a lista completa.

O ultimo programa será resolvido na aula seguinte.

Aula 08

Correcção dos exercícios da aula anterior. Em particular, o último exercício da aula anterior vai ser resolvido "ao vivo" durante esta aula.

Resolução de exercícios "ao vivo"

Já deverias ter os conhecimentos para resolver o último exercício da aula anterior. Ele é **muito simples** e é então **extremamente importante que não percas esta aula** se tiveste algum tipo de dificuldade a resolvê-lo!

Durante a aula, não percas tempo a passar o que vai sendo projectado/escrito no quadro. É mais importante prestares atenção. Os ficheiros que vão ser criados (um para cada turma) serão disponibilizados ao final da semana.

Aula 09

Pesquisa binária

Imagina que eu penso num número entre 1 e 100. Quero que adivinhes qual é, mas só te digo se as tuas tentativas estão acima ou abaixo do número em que pensei.

Qual seria a melhor estratégia para encontrar o número?

Seria obviamente começar pelo 50. A minha resposta ajuda a eliminar metade da gama da tua pesquisa. Se eu dissesse que tinha pensado num número inferior a 50, então ele só podia ser algo entre 1 e 50. Então voltas a dividir essa gama em 2 e testas o 25. Repetes o processo até esgotares os números ou encontrares o que procuravas.

Essa é a base da estratégia da pesquisa binária. Obviamente que isto só funciona se o critério dos elementos a pesquisar tiver alguma ordem para se poder dizer "maior" ou "menor". Essa é a desvantagem da pesquisa binária: só funciona com vectores ordenados.

Vê a explicação do programa `pesquisa_binaria.c` na aula.

Exercícios

Altera o programa `pesquisa_binaria.c` que faz parte da aula de hoje:

1. Aplica os conhecimentos das aulas 01 e 05 para fazer uma comparação de velocidade entre a pesquisa sequencial e a pesquisa binária. Repara que como as pesquisas são muito rápidas, só aumentar o tamanho do vector `v` não vai ser suficiente para teres tempos acima de 1s. Experimenta (também para garantir uma uniformização dos casos especiais de pesquisa) pôr ambos os algoritmos de pesquisa a pesquisar todos os inteiros de `1` a `val_max` e mede o tempo total.
2. Faz com que o programa da aula de hoje use uma lista pré-preenchida com *strings* (e.g.: usa os dados de teste do exercício *High Scores* da aula 04), e faça pesquisas no vector. Lembra-te da função `strcmp()` do C. Se isso funcionar bem, altera o vector para conter as estruturas completas do exercício *High Scores* da aula 04, pesquisando ainda apenas os nomes, mas exibindo toda a estrutura que é encontrada.

Tira as dúvidas com o professor.

Aula 10

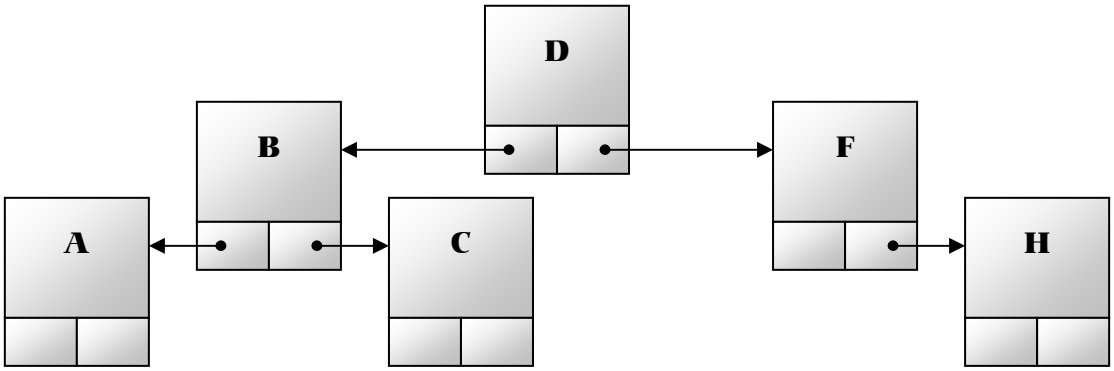
Árvores

Como é que implemento a pesquisa binária numa lista? Para ter acesso ao 50º elemento (por exemplo) tenho de percorrer todos os anteriores, um a um. Depois para aceder ao 25º terei de percorrer de novo os primeiros 25. Se tivesse feito uma pesquisa sequencial, já teria "por esta altura" visitado 75 elementos!

A pesquisa binária não é eficiente com listas. Então como podemos conjugar as vantagens de listas (dimensão dinâmica) com a velocidade da pesquisa binária?

Usamos estruturas novas chamadas de árvores. A árvore guarda os elementos pela mesma ordem em que seriam pesquisados com uma pesquisa binária.

Por exemplo, com 6 elementos, teríamos a seguinte árvore:



Repara como o primeiro elemento da árvore (o **D**, a sua "`raiz`") é o primeiro elemento que iríamos usar numa pesquisa binária. Ele aponta para dois elementos: um à sua esquerda (menor do que **D**) e outro à sua direita (maior do que **D**). Cada um destes repete o raciocínio, apontando para elementos menores à esquerda e maiores à direita.

Para que a raiz seja o ponto intermédio da pesquisa binária, todos os elementos a que temos acesso a partir do seu apontador esquerdo (e descendo através destes) são menores do que **D**. O inverso se pode dizer dos elementos à direita.

Em termos de código, a estrutura de cada elemento é muito semelhante à de uma lista:

```
struct s_elemento
       {
       ...
       struct s_elemento *esq;
       struct s_elemento *dir;
       };
```

```
typedef struct s_elemento elemento;

elemento *raiz = NULL;
```

Repara nos apontadores para a esquerda e direita (abreviados `esq` e `dir`) e no apontador inicial `raiz` que substitui o `primeiro` das listas. Mais uma vez, estes nomes são meros exemplos e podes encontrar muita literatura com nomes diferentes para estas mesmas coisas.

Inserir novos elementos numa árvore

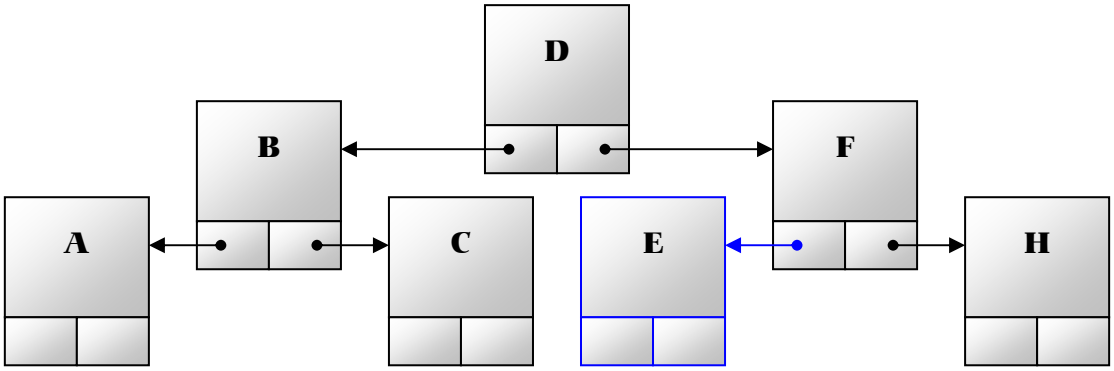
Cada novo elemento tem de ser inserido na posição adequada. E como cada elemento é maior ou menor do que outro elemento que já esteja na árvore, podemos sempre ir descendo a árvore até uma folha para inserir novos elementos. Os elementos já existentes não precisam de ser deslocados da sua posição actual.

O raciocínio é simples. Começamos na raiz. Se o novo elemento é menor, tentamos avançar pelo apontador esquerdo, Se for maior, tentamos avançar pelo apontador direito. Se não conseguimos avançar por esse apontador por ele ser `NULL`, então estamos numa folha e fazemos apontar esse apontador para o novo elemento, adicionando-o assim à árvore.

Por exemplo, vamos inserir o elemento **E** na árvore acima. Os passos são:

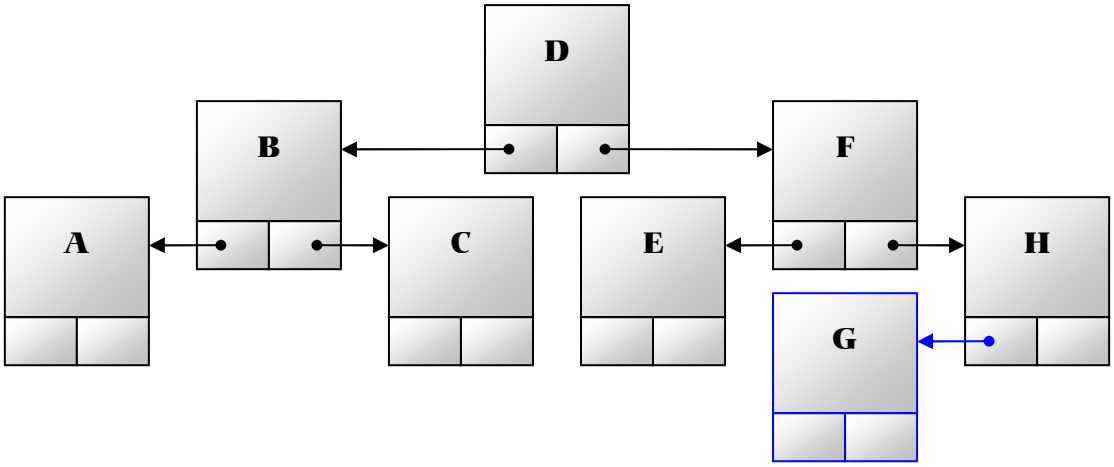
1. Começo na raiz, **D**
2. **E** é maior que **D** pelo que sigo pelo apontador direito, para **F**.
3. **E** é menor que **F** pelo que sigo pelo apontador esquerdo
4. Esse apontador era `NULL`, pelo que passa então a apontar para **E**

A árvore resultante é:



Muito bonito, mas já havia lugar para o **E**. E se eu quiser agora adicionar outro elemento, por exemplo **G**?

Repito o mesmo raciocínio. A árvore resultante será:



Repara que apesar do **G** vir imediatamente a seguir ao **F** no alfabeto, ficará depois do **H** na ordem de pesquisa. Isto é perfeitamente normal numa árvore: o **H** chegou antes do **G** à árvore mas as regras de ordenação da mesma mantêm-se.

Elementos iguais e árvores ternárias

Então e o que acontece se eu tentar adicionar outro elemento **igual** a um que já esteja na árvore?

Temos três soluções para o problema:

1. Colocamos os elementos iguais a outro elemento sempre à esquerda dele;
2. Colocamos os elementos iguais a outro elemento sempre à direita dele;
3. Criamos um novo apontador (e.g.: "`centro`") que guarda apenas os elementos iguais a este elemento.

As duas primeiras soluções servem perfeitamente, especialmente se esperamos ter poucos elementos idênticos entre si. Opto por uma das duas primeiras soluções e **uso sempre essa** no meu código, senão deixaria de encontrar elementos iguais.

A terceira solução é eficiente para o caso de poder ter muitos elementos iguais entre si. A esse tipo de árvores, com um terceiro apontador para elementos iguais, chamo de árvores ternárias. Não iremos dar mais detalhes sobre essas árvores.

Apagar elementos de uma árvore

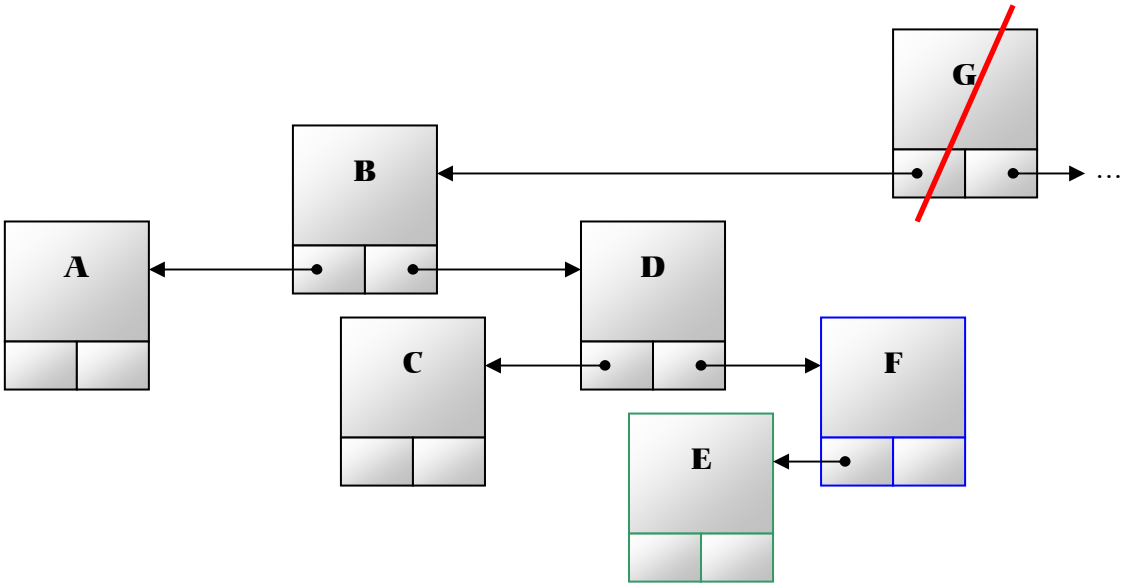
Apagar elementos de uma árvore é depende dos descendentes imediatos do elemento a apagar. Podem existir 3 casos:

1. O elemento a apagar não tem descendentes. Este caso é trivial e o elemento é simplesmente removido da árvore.
2. O elemento a apagar tem apenas descendentes de um dos lados (esquerda ou direita). Este caso também é bastante simples, pois o primeiro descendente (e a sua sub-árvore) substituem o elemento a apagar.

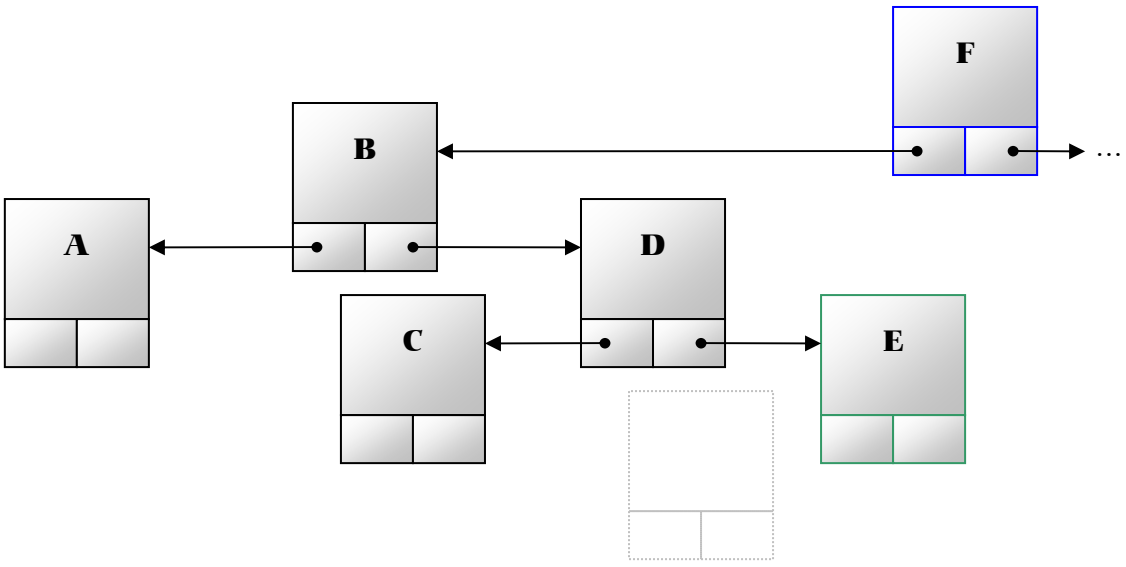
3. O elemento a apagar tem descendentes em ambos os lados (esquerda e direita).

O último caso é razoavelmente complexo. Eu vou substituir o elemento a apagar pelo seu sucessor ou predecessor imediato (em termos de ordenação). Se escolher o predecessor (por exemplo), encontro-o descendo pela esquerda do elemento a apagar, e a seguir sempre pela direita dos elementos seguintes até não haver direita. Esse elemento é o predecessor e substitui o elemento a apagar, e a sua sub-árvore esquerda (se existir) toma a posição antiga do predecessor.

Por exemplo, se eu quiser apagar o elemento **G** desta árvore:



Então desço pela esquerda de **G** (vou dar a **B**) e em seguida desço sempre pela direita até não haver direita (desço por **D** e depois **F**). **F** substitui **G** e a sua sub-árvore esquerda (neste caso apenas um elemento **E**) toma o lugar de **F**, ficando então a árvore:

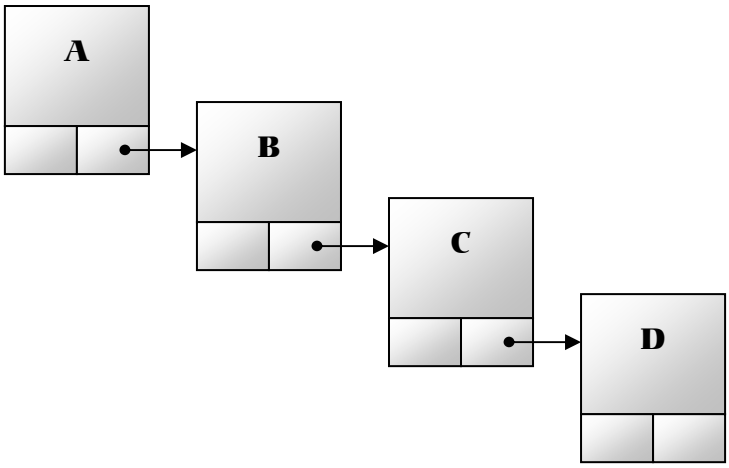


Usar sempre o mesmo critério (sucessor ou predecessor do elemento a apagar) pode levar a árvores desequilibradas (ver abaixo). Uma boa solução de apagar elementos numa árvore irá usar de forma semi-aleatória umas vezes o sucessor e outras vezes o predecessor.

Árvores degeneradas e equilibradas

Repara como é muito fácil a metade esquerda da árvore não ter tantos elementos quanto a metade direita. De facto imagina que vou criar uma árvore nova e lhe vou inserir os elementos **A**, **B**, **C**, **D** (por esta ordem).

A é o primeiro e fica na raiz. **B** é maior e fica à sua direita. E assim por diante até se ter a seguinte árvore:



Esta é uma propriedade das árvores, quando se lhe inserem elementos já ordenados: elas degeneram em listas.

Para isto temos duas soluções: equilíbrio pontual de árvores e árvores que se mantêm sempre equilibradas (aquelas onde em qualquer nó, o número de elementos à esquerda difere do número de elementos à direita no máximo por apenas um).

Exercícios

Cria um programa simples, parecido ao `lista.c` da aula 06, mas que cria uma árvore e lhe insere dois elementos. Quero a inserção a ser feita numa função independente.

Tenta também criar uma função para exibir todos os elementos da árvore. Tenta também fazer (outra?) versão que exibe todos os elementos da árvore, mas já ordenados. Dica: as soluções mais simples são recursivas.

Estas funções (e mais) serão dadas na próxima aula.

Aula 11

Exercícios sobre árvores.

Exercícios

Cria um programa que exibe um menu para gerir uma árvore que guarda nomes de pessoas. O menu deve ter várias opções para inserir elementos novos, para remover elementos, para pesquisar elementos e para exibir a lista completa, ordenada.

Se eu quisesse adicionar uma função para ordenar a árvore, como o faria?

Aula 12

Guardar e recuperar listas e árvores a partir de ficheiros.

Ficheiros

Listas e árvores são definidas por interligações dos seus elementos. Estas ligações são feitas com apontadores.

Já deves ter reparado que eu nunca dou um valor (endereço) a um apontador no meu código. Um apontador recebe indirectamente o endereço de outra variável do meu programa, ou de memória que reservei com `malloc()`.

Para gravar e ler estas estruturas de disco preciso então de ter em atenção duas coisas:

1. Os vários elementos destas estruturas não estão contíguos em memória, pelo que tenho de gravar cada elemento de cada vez (i.e., uma operação `fwrite()` ou `fprintf()` para cada elemento), e o mesmo se aplica à leitura;
2. Os endereços que estão nos apontadores que pertencem a cada elemento podem ser (ou não) escritos para o disco: a opção é tua. Tens é de te lembrar que se os leres de volta, **os valores desses apontadores são inválidos** porque se referiam a endereços de memória reservada noutra instância do teu programa.

A diferença entre o uso de `fwrite()` (e o correspondente `fread()`) ou `fprintf()` (e o correspondente `fscanf()`) tem a ver com o modo de escrita/leitura: respectivamente binário ou texto.

Binário vs. texto

Se escreves cada elemento de forma binária (`fopen(...,"wb")` e `fwrite()`) ou em texto (`fopen(...,"w")` e `fprintf()`) é contigo. Ambos servem para o propósito. Mas têm diferenças.

O modo binário é mais simples de usar. Funciona como o `malloc()`: não se importa com o que é que estar a ler/escrever. Importa-se só onde é que isso está em memória e qual é o seu tamanho.

Por exemplo, se `p` aponta para um `elemento`, eu escrevo o elemento em memória com um simples

```
fwrite( p, sizeof(elemento), 1, fp );
```

Isto é mais rápido e fácil de desenvolver do que a escrita em modo de texto descrita abaixo, mas tem problemas.

Os problemas têm a ver de alguns compiladores optimizarem as estruturas que usamos criando "buracos" nelas, e os computadores diferirem na forma como representam alguns tipos de dados (vírgula flutuante, complemento para dois, ou mais provável, *big-endian* vs. *little-endian)*. Mais sobre este problema na aula.

Isto significa que **o meu ficheiro só tem garantia de voltar a funcionar com o mesmo ficheiro executável do meu programa**, e não com recompilações do mesmo ou com outros programas e computadores.

O modo de texto resolve isto, em troca de complexidade. Tal como o nome indica, ele grava as coisas em texto o que significa que um inteiro de 32 bits que ocuparia no disco 4 bytes quando escrito em modo binário, ocupa agora tantos bytes quantos dígitos decimais (ou hexadecimais) ocupar a sua representação, já que vão ser gravados os caracteres que o representam para o ser humano, e não os bytes que o representam na memória do computador.

Por exemplo, a seguinte estrutura de um elemento:

```
struct  s_elemento
        {
        int numero;
        char tipo;
        struct s_elemento *proximo;
        };
```

Seria gravada em modo de texto com:

```
fprintf( fp, "%d %c", p->numero, p->tipo );
```

Nota como vamos precisar de distinguir um membro da estrutura de outro, precisamos de os separar. Como o `fscanf()` usa automaticamente *white space* (espaços, tabuladores ou quebras de linha) como separadores, esta é uma escolha típica.

Neste modo de escrita tens de ter cuidado como escolhes escrever *strings* devido à forma como `fscanf()` as lê. Mais detalhes na aula.

Abordagem para escrita

Não há grandes detalhes sobre a escrita de listas e árvores para além do que já foi dito, excepto que deves ter atenção da ordem em que escreves os elementos em relação à leitura.

No caso das listas a situação é trivial: dá mais jeito ler e escrever as listas da esquerda para a direita (do `primeiro` até ao último elementos) pelo que essa é exactamente a estratégia.

```
for( p = primeiro;  p != NULL;  p = p->proximo )
        fwrite( p, sizeof(elemento), 1, fp );
```

Nas árvores a situação é mais delicada. Se eu escrever os elementos de uma árvore "da esquerda para a direita" (e então por ordem), vou lê-los na mesma ordem. E

como já viste na aula anterior, as árvores não se dão bem com elementos a serem-lhe adicionados já em ordem!

Então nesse caso a melhor abordagem é a adição dos elementos pela mesma ordem em que estão em memória, algo parecido a isto:

```
void escreve( elemento *p, FILE *fp )
{
      if( p == NULL )
            return;
      /* escreve este proprio elemento */
      fwrite( p, sizeof(elemento), 1, fp );
      /* escreve os elementos por "debaixo" deste */
      escreve( p->esq, fp );
      escreve( p->dir, fp );
}
...
escreve( raiz, fp );
```

Tira as dúvidas com o professor, na aula.

Abordagem para leitura

Para além de agora estarmos a fazer o inverso da escrita, temos de ter o cuidado com a forma como a biblioteca `stdio.h` do C nos diz que chegámos ao final de um ficheiro.

Infelizmente, em C, `feof()` diz-nos se a **última** operação de leitura encontrou o fim do ficheiro *(end-of-file)* e não se ele se encontra agora "à nossa frente". Temos sempre de tentar ler, e se a leitura falhou podemos distinguir um erro do fim de ficheiro com `feof()`.

Isto cria um problema. Para ler um elemento preciso de espaço para ele. Então:

- Se fizer `malloc()` do elemento e ler para essa memória, então quando chegar ao fim do ficheiro tenho de fazer `free()` dessa memória (e ter cuidado com os apontadores);
- Posso criar uma variável temporária com um elemento. Leio para dentro dessa variável. Assim não preciso de fazer o `free()` (ao fim), mas tenho de copiar o conteúdo dessa variável para uma memória que reservo com `malloc()` em todos os outros elementos. Isto acaba por ser mais lento.

Para simplificar, vou neste exemplo ignorar a distinção e assumir sempre que se a leitura falhou, devo parar como se tivesse encontrado o fim do ficheiro.

Então, para ler uma lista, posso fazer algo parecido a:

```
elemento *novo, **pp;
...
for( pp = &primeiro;  ;  pp = &(novo->proximo) )
      {
      novo = malloc( sizeof(elemento) );
      if( novo == NULL )
            (erro: memória insuficiente);
```

```
        if( fread(novo, sizeof(elemento), 1, fp) != 1 )
              {
              *pp = NULL;
              free( novo );
              break;
              }
        *pp = novo;
        }
```

Repara como garantimos sempre que substituímos os apontadores "`proximo`" lidos por novos valores.

Para as árvores, como escrevemos os elementos sem indicação se pertenciam ao ramo esquerdo ou direito, ou a que nó, não conseguimos reconstruir a árvore enquanto a lemos, mas podemos recorrer à função `insere()` da aula anterior para ir inserindo os elementos que lemos. A árvore irá ficar mais ou menos tão equilibrada quanto a original porque tivemos cuidado na ordem de escrita dos elementos.

O código pode ser o seguinte:

```
elemento *novo;
...
for(;;)
        {
        novo = malloc( sizeof(elemento) );
        if( novo == NULL )
              (erro: memória insuficiente);
        if( fread(novo, sizeof(elemento), 1, fp) != 1 )
              {
              free( novo );
              break;
              }
        novo->esq = novo->dir = NULL;
        insere( &raiz, novo );
        }
```

Tira as dúvidas com o professor, na aula.

Exercícios

Pega no último exercício da aula 07 e nos exercícios da aula 10 e acrescenta a cada um a capacidade de escreverem para o disco a lista/árvore que são construídas em memória, e a capacidade para as lerem de volta (substituindo as estruturas que já estejam em memória).

Estas capacidades devem ser activadas mediante novas opções dos menus.

Aula 13

Trabalhar com listas de tipos de dados genéricos.

Tipos de dados variáveis

A linguagem C não nos permite ter variáveis cujo tipo é determinado em tempo real (i.e., durante a execução do programa).

Como resolvemos então isso? Em cada instante, fazemos `malloc()` do espaço necessário, e trabalhamos com esse espaço calculando de forma independente a posição de cada elemento dentro do espaço.

Por exemplo, imagina que quero criar num certo instante uma estrutura que tem:

- Um inteiro, seguido de
- Um vector de 20 caracteres, seguido de
- Um `double`.

O código que construiria esta estrutura seria:

```
struct
	{
	int membro1;
	char membro2[20];
	double membro3;
	};
```

Nota como isto é diferente de uma `union` com os mesmos membros. As `union` não nos resolvem o problema que temos nesta aula. Exercício: porquê? Porque não poderia eu criar uma `union` com todos os tipos de dados que poderia precisar e usar cada um deles conforme necessário?

No entanto eu não conseguia determinar que esta seria a estrutura pretendida quando estava a programar (devido a alguma exigência funcional do meu programa). Então como a construo?

Bem, primeiro reservo espaço para ela, guardando o endereço desse espaço num apontador genérico:

```
void *p;
...
p = malloc(sizeof(int)+sizeof(char[20])+sizeof(double));
```

Repara como usámos os nomes dos **tipos** de dados no `sizeof()`. Repara também como se especifica um vector em termos de tipo de dados. Com vectores também podemos determinar o seu tamanho com algo parecido a:

```
sizeof(char)*20
```

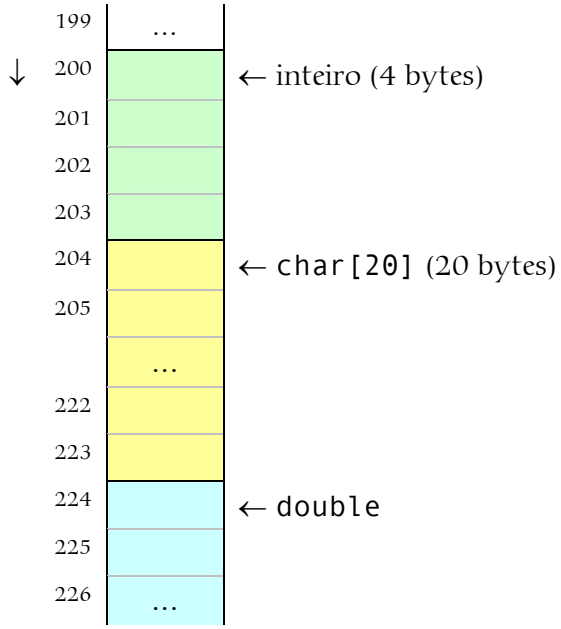
Se o inteiro é o primeiro membro desta estrutura, então o endereço dele dentro do espaço que reservei é simplesmente

```
p
```

O vector de caracteres vem depois do inteiro. Como a soma de um vector genérico com um inteiro representa um deslocamento em bytes (como num apontador para caracteres), então o endereço do vector de caracteres é:

```
p + sizeof(int)
```

Vamos ver isso em mais detalhe. Imagina que `p` ficou com o valor de 200. Então este é o aspecto da estrutura em memória:



Repara como o endereço inicial do vector de 20 caracteres seria 204, assumindo que um inteiro ocupa o tamanho típico de 4 bytes. Mas seja qual for o tamanho que ocupe, `sizeof(int)` dá-me sempre esse tamanho, pelo que `p+sizeof(int)` dá-me sempre o endereço inicial do vector.

Para exibir esse vector no ecrã eu poderia fazer (por exemplo):

```
printf( "%s", p+sizeof(int) );
```

Por fim e seguindo o mesmo raciocínio, o endereço do `double` é:

```
p + sizeof(int) + sizeof(char[20])
```

Posso fazer contas com o inteiro ou o `double` lendo os seus valores como com qualquer ponteiro. Embora um ponteiro genérico aponte para "vazio" (`void`), eu posso alterar o tipo de dados desse apontador temporariamente (com um *cast)* para outro de forma a poder ler o valor para onde ele aponta.

Por exemplo:

```
int i;
double d;
...
i = *((int*) p);
d = *((double*) p+sizeof(int)+sizeof(char[20]));
```

Esta é a técnica que as linguagens que permitem o uso de tipos de dados dinâmicos (determinados em tempo real) usam para implementar essa funcionalidade.

Listas genéricas

Um lista genérica precisa do apontador `proximo` para além dos dados que contém. Esses dados devem ser definidos da forma discutida acima, e podem fazer parte do `malloc()` do `proximo` em si:

```
struct s_elemento
       {
       struct s_elemento *proximo;
       };
typedef struct_s_elemento elemento;
...
void *p;
...
p = malloc( sizeof(elemento) + sizeof(...) + ... );
```

Ou simplesmente:

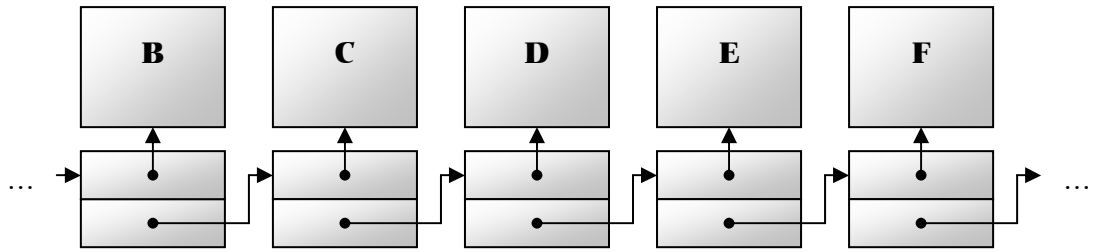
```
void *p;
...
p = malloc( sizeof(void*) + sizeof(...) + ... );
```

Esquematicamente é idêntico a qualquer diagrama de lista já exibido na aula 07.

Mas os dados também podem ser reservados independentemente:

```
struct s_elemento
       {
       void *dados;
       struct s_elemento *proximo;
       };
typedef struct_s_elemento elemento;
...
elemento *p;
...
p = malloc( sizeof(elemento) );
p->dados = malloc( sizeof(...) + sizeof(...) + ... );
p->proximo = NULL;
```

Esquematicamente seria algo parecido a:



A melhor solução depende da abstracção pretendida para as funções que adicionam novos elementos à lista.

Exemplo

O programa `dados-variaveis.c` está disponível já resolvido. Ele é a solução do enunciado que se segue.

Este programa deveria pedir ao utilizador 5 tipos de dados diferentes (suporta `int`, `double` e `char`), podendo cada um ser um vector.

Cria então um elemento destes em memória, pede os dados ao utilizador preenchendo o elemento, e volta a exibir os dados pela ordem inversa.

Exercício

Altera o exemplo anterior para:

- Adicionar o suporte ao tipo de dados `long` (inteiro longo);
- Reconhecer um vector de caracteres como um caso especial que deve ser lido e exibido como uma *string*.

Tira as dúvidas com o professor, nas aulas.

Exercícios

As aulas seguintes são de revisão e aprofundamento desta matéria e não estão detalhadas neste guião. Elas servem para te acompanhar na execução do projecto final da cadeira e para cimentar os conhecimentos aqui adquiridos.

Não deixes de ir às aulas.

Bibliografia

[C-LANG]
Brian W. Kernighan & Dennis Ritchie,
"The C Programming Language" (2ª edição), Prentice Hall.
Na Amazon do Reino Unido, em:
http://www.amazon.co.uk/C-Programming-Language-2nd/dp/0131103628/

[ART-PROG]
Donald E. Knuth,
"The Art of Computer Programming: Vol. 1-3" (2ª edição), Addison Wesley.
Na Amazon do Reino Unido, em:
http://www.amazon.co.uk/Art-Computer-Programming-Information-Processing/dp/0201485419/

(mais em breve)