Sebenta de Apoio às Aulas Teóricas de

Métodos de Programação II

Mestrado Integrado em

Engenharia de Telecomunicações e Informática

António J. Esteves

esteves@di.uminho.pt

Escola de Engenharia

Universidade do Minho

28 Março de 2016

Índice

1. Introdução

1.1 Objetivos da Unidade Curricular

Em conjunto com a Unidade Curricular (UC) de Métodos de Programação I, a presente UC tem por principal objetivo consolidar os conhecimentos sobre algoritmia e sua implementação na linguagem C. São também objetivos da UC: cobrir tópicos avançados da linguagem C, apresentar as estruturas de dados mais importantes, descrever e implementar algoritmos relevantes para a área das comunicações.

1.2 Resultados de Aprendizagem

Ao concluir com sucesso esta UC um aluno deve adquirir as seguintes competências:
- Ser capaz de elaborar algoritmos e de os implementar numa linguagem imperativa como o C.
- Conceber e escrever programas em C estruturados e modulares.
- Perceber que a linguagem C é uma linguagem de nível baixo/intermédio, e que por isso exige adquirir alguns conhecimentos sobre o *hardware* em que os programas escritos vão ser executados.
- Perceber estruturas de dados como listas, pilhas, filas, árvores e grafos, e implementar em C programas que utilizem essas estruturas.

1.3 Resumo das Caraterísticas Mais Relevantes da Linguagem C

Os aspetos fundamentais da linguagem C incluem (i) utiliza poucas palavras-chave, (ii) suporta tipos de dados compostos tais como *structs* ou *unions*, (iii) permite usar apontadores, ou seja acesso direto à memória, por exemplo para manipular *arrays* de tamanho variável, (iv) recorre a bibliotecas externas normalizadas de funções de entrada/saída, matemáticas, de acesso à memória ou outras facilidades, (v) quando é compilada gera código máquina nativo dum dado processador e (vi) é utilizada em conjunto com um pré-processador de diretivas.

O código C é rápido, compacto, de uso genérico, e é uma linguagem de programação "maioritariamente" independente da plataforma alvo que a vai executar. O C é utilizado para desenvolver sistemas tais como compiladores, interpretadores, sistemas operativos, sistemas de base de dados, ou microcontroladores. C é uma linguagem estática (compilada), tipada, estruturada e imperativa. Numa **linguagem tipada** todas as variáveis têm um tipo específico e os tipos são importantes para a linguagem. A **programação estruturada** procura melhorar a clareza, a qualidade e o tempo de desenvolvimento dos programas. Para isso, usa-se extensivamente subrotinas, blocos estruturados e ciclos. Isto contrasta com a utilização de testes simples e saltos com a instrução `goto`, que poderia levar a "código espaguete", que é difícil de perceber e de manter. Um **bloco** é uma seção de código agrupado. Exemplos de blocos estruturados em C são o `if..else` `if..else` e o `switch`. A **programação imperativa** é um paradigma que descreve a computação como instruções que mudam o estado (variáveis) dum programa. Analogamente ao comportamento imperativo das linguagens naturais que expressam ordens, os programas imperativos são uma sequência de comandos para o computador executar. Utilizando palavras de Dennis Ritchie, um dos criadores da linguagem, "o C é peculiar, tem limitações, mas é um enorme sucesso".

1.4 Paradigmas de Programação

1.4.1 Programação imperativa

A programação imperativa descreve a computação como uma **sequência** de instruções, ou comandos, a executar pelo computador. As instruções mudam o estado do programa. O **estado** dum programa é definido à custa dos valores das suas variáveis. A programação imperativa é análoga ao comportamento **imperativo** das linguagens naturais, em que se dão ordens: faz a tarefa X, a seguir faz a tarefa Y, depois faz a tarefa Z. Exemplos de linguagens que permitem programar de forma imperativa são o Fortran, Pascal e C.

1.4.2 Programação funcional

A programação funcional trata a computação como uma avaliação de **funções matemáticas**, em que os programas não possuem um estado gobal. É um paradigma declarativo, o que significa que a programação é feita com expressões. O valor de saída duma função depende apenas dos argumentos de entrada. Ou seja, chamar uma função com os mesmos valores produz sempre o mesmo resultado. Como não existem efeitos colaterais nas funções é mais fácil compreender e prever o comportamento dos programas. Um exemplo de linguagem funcional é o Haskell.

1.4.3 Programação lógica

A programação lógica especifica "o que computar" e não "como computar". Este tipo de programação é baseado em lógica formal. Um programa é um conjunto de afirmações na forma lógica, que expressam **factos** e **regras** sobre o problema. As **regras** são escritas na forma de **cláusulas**:

```
H :- B1, ..., Bn.
```

e são lidas como implicações lógicas:

```
H se B1 e ... Bn.
```

Em que `H` é a cabeça da regra e `B1, ..., Bn` são o corpo. Os **factos** são regras sem corpo e são escritos simplemente como:

```
H.
```

Prolog é um exemplo de linguagem lógica.

1.5 Áreas de Aplicação do C

O C é utilizado como linguagem de programação de sistemas bastante distintos no que toca a recursos de *hardware* disponíveis e ao ambiente de funcionamento. Podemos contudo enumerar as principais categorias de sistemas em que é muito utilizado: (i) sistemas operativos como o Linux ou UNIX, (ii) microcontroladores usados em automóveis, eletrónica de consumo, ou aviões, (iii) processadores embebidos em telemóveis, impressoras, ou dispositivos eletrónicos portáteis, (iv) processadores de sinal (DSPs) incluídos em aparelhos de áudio e TV digitais.

1.6 C vs. Linguagens Semelhantes

Vamos agora ver como se posiciona o C no contexto das linguagens similares. Entre este universo de linguagens encontramos linguagens derivadas do C mas que são mais recentes, como o C++, Objective C ou C#, e linguagens inspiradas no C mas que já são bastante diferentes, tais como o Java, Perl, ou Python. Entre as caraterísticas que é possível encontrar nestas linguagens e que o C não permite podemos mencionar: (i) fazer o tratamento de exceções de forma nativa, (ii) verificar se a posição de acesso a um *array* está dentro da gama válida (*range-checking*), (iii) eliminar o espaço de memória alocado dinamicamente sem ser com código incluído pelo programador nos programas (*garbage collection*), (iv) programação orientada aos objetos e (v) polimorfismo. **Polimorfismo** é uma característica duma linguagem de programação que permite que valores de diferentes tipos (de dados) sejam utilizados através duma interface uniforme. Um exemplo de polimorfismo é poder ter variantes duma função com o mesmo nome (por ex., `soma()`), em que cada variante opera sobre um tipo de dados diferente: `int soma(int op1, int op2)`, `float soma(float op1, float op2)` ou `double soma(double op1, double op2)`. Por ser uma linguagem de mais baixo nível, resulta que o código obtido a partir do C é normalmente mais rápido.

1.7 C é uma Linguagem de Baixo ou Médio Nível

Algumas das caraterísticas que fazem do C uma linguagem de baixo ou médio nível são: (i) é inerentemente não segura porque não possui *range checking*, (ii) só permite uma validação limitada do tipo de dados durante a compilação e (iii) não permite validação do tipo de dados durante a execução. Por estes motivos, deve utilizar-se o C com algum cuidado. Por exemplo, deve-se depurar os programas com uma ferramenta como o `gdb` e nunca correr os programas como `root` em Linux. Dois exemplos em que não é possível verificar se o tipo de dados corresponde ao espetável são: (i) quando uma função permite parâmetros em número e tipo variável (como o `printf`) e (ii) quando uma função permite parâmetros do tipo `void*`.

1.8 Funções de Entrada/Saída

A interação com o exterior dos programas escritos em C faz-se recorrendo a funções de bibliotecas normalizadas. Estas funções normalizadas acompanham qualquer compilador de uso genérico. A interação faz-se através de três canais de comunicação: um para entrada (`stdin`), um para saída (`stdout`) e outro para enviar mensagens de erro (`stderr`). Exemplos de dispositivos com os quais é comum os programas terem que comunicar é o teclado (entrada) e o ecrã (saída).

Apresenta-se a seguir uma lista com algumas das funções de entrada/saída utilizadas mais frequentemente:

- `puts(str)`: escreve o conteúdo da string `str` em `stdout`;
- `str = gets(str)`: lê uma linha a partir de `stdin` para a string `str`. A função `gets(str)` não deve ser usada pois é perigosa. Isto deve-se ao facto de não haver maneira de saber o espaço disponível na *string* `str` de modo a evitar escrever em zonas não alocadas a essa *string*;
- `putchar(ch)`: escreve o carater `ch` em `stdout`;
- `ch = getchar()`: devolve um carater lido de `stdin`;
- `printf(char format[], arg1, arg2, ...)`: escreve no ecrã de forma formatada;

- `sprintf(char str[], char format[], arg1, arg2, ...)`: semelhante a `printf()` mas escreve na *string* `str`;
- `fprintf(FILE* fp, char format[], arg1, arg2, ...)`: semelhante a `printf()` mas escreve no ficheiro identificado pelo apontador `fp`;
- `scanf(char format[], &arg1, &arg2,...)`: lê carateres do teclado, interpreta-os de acordo com o formato `format` especificado e guarda os valores lidos em `arg1`, `arg2`, etc;
- `sscanf(char str[], char format[], &arg1, &arg2,...)`: semelhante a `scanf()` mas lê da *string* `str`;
- `fscanf(FILE* fp, char format[], &arg1, &arg2,...)`: semelhante a `scanf()` mas lê do ficheiro identificado pelo apontador `fp`.

1.9 Macros e Diretivas para o Pré-processador

Em C é permitido incluir diretivas para que o pré-processador da compilação efetue tarefas antes da compilação propriamente dita. As duas diretivas mais comuns são `#include` e `#define`. `#include` indica ao pré-processador para utilizar o código duma biblioteca. `#define` define uma expressão através de macros.
Uma **macro** é uma regra que especifica como uma determinada sequência de entrada, muitas vezes uma sequência de carateres, deve ser mapeada numa sequência substituta. No contexto duma linguagem de programação, as macros são usadas para permitir que uma sequência de instruções possa ser utilizada pelo programador sob a forma de uma única instrução, tornando a tarefa de programar menos cansativa e menos suscetível a erros. Uma macro pode incluir argumentos e é tratada como uma função. Um exemplo de macro:

```
#define  add3(x,y,z)  ((x+y)+z)
```

Os parênteses garantem que a execução das operações é efetuada pela ordem correta. O compilador faz a substituição *inline. E*sta estratégia não é adequada para problemas recursivos. **Porquê?**
A opção `-Dnome=valor` do `gcc` equivale à diretiva `#define nome valor`. Deste modo, `nome` pode ser usado no código fonte em vez de `valor`.

Quando se declara uma função com a diretiva `inline` antes, instrói-se o compilador para substituir cada chamada a essa função pelo corpo completo dessa função. Esta modificação torna o código mais rápido ao eliminar o código necessário para gerir a chamada da função. Apresenta-se a seguir um exemplo da utilização da diretiva `inline`:

```
inline int maximo(int a, int b) {
return (a > b) ? a : b;
}
```

A seguinte instrução

```
z = maximo(x, y);
```

pode ser transformada em

```
z = (x > y) ? x : y;
```

1.9.1 Diretivas condicionais para o pré-processador

As diretivas #if, #ifdef, #ifndef, #else, #elif e #endif permitem controlar quais as linhas de código a compilar. Como já foi dito, estas diretivas são analisadas antes de o código propriamente dito ser compilado. As condições incluídas nestas diretivas devem ser expressões definidas com `define`'s ou literais. Um **literal** corresponde a um valor fixo no código fonte, tal como um inteiro, um número em vírgula flutuante, uma *string*, ou um valor booleano. Estas diretivas podem ser utilizadas num ficheiro *header* para garantir que as declarações nele contidas só são processadas uma vez.

A diretiva `#pragma` é a forma disponível em C para passar informação adicional ao compilador, além da que é possível especificar na própria linguagem C. Por seu lado, as diretivas `#error` e `#warning` instroem o compilador para enviar um erro e um aviso para a consola, respetivamente, no caso de ele passar por essa diretiva. Em caso de erro, a compilação é abortada. A diretiva `#undef msg` elimina a definição de `msg`, efetuada através dum `#define`. `#undef` é útil para evitar conflitos de definições. Finalmente, a diretiva `#pragma once` instrói o compilador para incluir um ficheiro *header* apenas uma vez, independentemente do número de vezes que surgir o `#include` desse ficheiro. Exemplos de utilização de diretivas condicionais:

```
#pragma once
/* código do ficheiro header */
```

Esta diretiva é equivalente a incluir uma guarda que evita que um ficheiro *header* seja processado/incluído múltiplas vezes, como se ilustra no código seguinte.

```
#ifndef _FILE_NAME_H_
#define _FILE_NAME_H_
/* código do ficheiro header */
#endif
```

O próximo exemplo mostra como se podem usar diretivas condicionais para verificar que compilador está a ser utilizado na compilação.

```
#ifdef  _MSVC_
// Faz algo específico do MSVC
#elif  _GCC_
// Faz algo específico do GCC
#else
#error  O tipo de compilador a usar não foi definido!
#endif
```

O exemplo a seguir ilustra como as diretivas condicionais permitem controlar o tamanho dum *array*, de modo a ele nunca ser inferior a 50 nem superior a 200.

```
#if     TABLE_SIZE>200
#undef  TABLE_SIZE
#define TABLE_SIZE 200
#elif   TABLE_SIZE<50
#undef  TABLE_SIZE
#define TABLE_SIZE 50
#endif
int table[TABLE_SIZE];
```

Exercício 1.9.1.1: Qual a diferença entre os dois fragmentos de código seguintes?

```
#ifdef  DEBUG
printf("Numero de iteracoes = %d\n",nit);
#endif
```

```
if(debug == 1){
  printf("Numero de iteracoes = %d\n",nit);
}
```

1.10 Precedência e Ordem de Avaliação dos Operadores

A prioridade dos operadores, influencia a ordem pela qual esses operadores são avaliados, podendo alterar a ordem de avaliação espectável e que, sem prioridades nem parênteses, seria da esquerda para a direita. Deste modo, é preciso ter cuidado quando se misturam operadores com prioridades distintas numa expressão, por forma a obter o resultado desejado.

- `++`, `--`, `(cast)`, `sizeof` são os operadores com prioridade mais alta
- Os operadores `*`, `/`, `%` têm prioridade maior do que `+`, `-`
- Os operadores ==, `!=`, >, >=, … têm prioridade maior do que `&&` e `||`
- O operador de atribuição "=" tem prioridade mais baixa.

Recomenda-se usar os parênteses curvos de forma generalizada para evitar ambiguidades e efeitos indesejados resultantes de uma análise incorreta da precedência dos operadores. Apresentam-se a seguir três exemplos da influência da precedência dos operadores no resultado gerado por uma expressão.

```
y = x*3+2;              /* o mesmo resultado que  y=(x*3)+2;              */
x !=0 && y==0           /* o mesmo resultado que  (x!=0) && (y==0)        */
d = c>='0' && c<='9';   /* o mesmo resultado que  d=((c>='0')&&(c<='9')); */
```

1.11 Apontadores e Endereços

Normalmente, um apontador representa o endereço da posição de memória onde está guardada uma variável. O endereço pode ser usado para aceder ou modificar essa variável a partir de qualquer local no código. Os apontadores são extremamente úteis, especialmente em estruturas de dados, mas tornam o código menos claro e mais vulnerável.

2. Tópicos Avançados em C

2.1 Funções: Como Dividir um Programa em Partes Menores

Nesta seção vamos ver como se utilizam funções para dividir um programa em partes menores, recorrendo para isso ao algoritmo de Euclides, que permite calcular o **m**áximo **d**ivisor **c**omum (MDC). O algoritmo baseia-se no seguinte princípio: `mdc(a,b) = mdc(b,a mod b)`, para números inteiros tais que `a > b`.

```
int mdc ( int a, int b) {
   if (a < b) {      /* se (a < b) trocar a com b */
      int t = a; a = b; b = t;
   }
   while (b != 0) { /* enquanto (b!=0) faz mdc(b,a mod b) */
      int temp = a%b;
      a = b;
      b = temp;
   }
   return a;
}
```

Esta implementação do algoritmo utiliza um ciclo em que repetidamente se substitui *a* por *b* e *b* por *a mod b*, até *b* ser 0. O algoritmo de Euclides estendido, apresentado a seguir, calcula os inteiros *x* e *y* tais que: *a*x + b*y = mdc(a, b)*.

Algoritmo de Euclides estendido:

Entradas: inteiros *a* e *b*
Saídas: inteiros, *x*, *y* e *d*, com *d=mdc(a,b)=a*x+b*y*
d0=a , x0=1 , y0=0
d1=b , x1=0 , y1=1
Enquanto(d1 != 0) fazer
  q = ⌊d0/d1⌋
  d2 = d1 , x2=x1 , y2=y1
  d1 = d0-q*d1
  x1 = x0-q*x1
  y1 = y0-q*y1
  d0 = d2 , x0=x2 , y0=y2
fimEnquanto
Devolve {d,x,y} = {d0,x0,y0}

2.1.1 Funções que devolvem múltiplos valores

O algoritmo de Euclides estendido devolve o valor de *mdc* e mais duas variáveis *x* e *y*. Mas como conseguir esse objetivo se as funções retornam no máximo um valor? Duas hipóteses de solução para este problema passam por (i) usar variáveis globais ou (ii) declarar variáveis para todas as saídas, exceto uma, fora da função. As variáveis declaradas fora das funções são globais, ou seja, o seu tempo de vida coincide com o do programa. As variáveis globais podem ser acedidas/alteradas em qualquer função. É por esta razão que a utilização de variáveis globais é completamente desaconselhada por violar as boas práticas de programação.

Considere que o algoritmo de *Euclides estendido* é implementado pela função `int ext_euclid(a,b)`, a qual devolve `d=mdc(a, b)`, x e y. Podemos usar variáveis globais para x e y, mas também podemos utilizar apontadores para devolver resultados múltiplos. Esta última alternativa é ilustrada na função seguinte.

```
int ext_euclid(int a, int b, int *x, int *y);
```

Ou seja, ao chamar `ext_euclid()` passamos-lhe apontadores para as variáveis que vão receber x e y:

```
int x, y, d;
d = ext _euclid(a, b, &x, &y);
```

O compilador pode emitir um aviso para sinalizar que x e y são utilizados antes de serem inicializados.

Quando temos que resolver um problema de grande dimensão, uma forma de lidar como complexidade inerente a essa dimensão é aplicar uma estratégia denominada por "dividir para reinar". A essência desta estratégia é dividir o problema em sub-problemas mais pequenos/simples. Se o problema consistir em escrever um programa em C, a implementação da estratégia "dividir para reinar" passa essencialmente pela utilização de funções. Além de utilizar funções, devem-se seguir as seguintes recomendações:

- Utilizar ciclos e recursividade, quando possível;
- Minimizar a transferência de informação entre a função que chama outra e a função que é chamada;
- Escrever pseudocódigo antes de codificar.

Pode-se colocar a questão "como se implementa `mdc(a,b)` recursivamente?". A resposta é: utilizando o princípio `mdc(a,b)=mdc(b,a mod b)`. O código seguinte mostra a versão recursiva da função `mdc()`.

```
int mdc_recursivo (int a, int b) {
   if ((a % b) == 0)
      return b;
   else
      return mdc(b, a % b);
}
```

2.2 Programação Modular

2.2.1 Utilização de ficheiros header

Os programas em C não têm, nem devem, ser monolíticos. Ou seja, o código não deve ser concentrado num único **módulo**. Embora o C não suporte explicitamente **programação modular**, o princípio base da programação modular em C consiste em agrupar as funções relacionadas num módulo. A divisão de programas em módulos é um bom princípio para a conceção de programas de grande dimensão. Um módulo possui interface e implementação. A **interface** pode estar contida num ficheiro *header* e a **implementação** em ficheiros fonte, *assembly* ou objeto (.c, .s, .o). Os módulos fornecem abstração, encapsulamento, e escondem informação, facilitando assim a organização de programas grandes.

2.2.2 Módulo para o algoritmo de Euclides

O algoritmo de Euclides é um método simples e eficiente de encontrar o máximo divisor comum entre dois números inteiros diferentes de zero. Este algoritmo é útil em vários contextos, logo seria interessante poder reutilizar a sua funcionalidade em diferentes programas. A solução passa por criar um módulo para os algoritmos de Euclides. Para isso é preciso criar um ficheiro *header* (.h) e um ficheiro fonte (.c). No ficheiro com o código fonte, por exemplo `euclid.c`, incluímos a implementação da função `mdc()` que se segue.

```
/* função mdc(): calcula o máximo divisor comum */
int mdc ( int a, int b) {
   if (a<b) { /* se (a < b) é preciso trocar a com b */
      int t = a; a = b; b = t;
   }
   while (b != 0) {
      int temp = a%b;
      a = b;
      b = temp;
   }
   return a;
}
```

Para podermos reutilizar o algoritmo de Euclides noutros ficheiros fonte, temos que dar conhecimento das funções e variáveis globais definidas em `euclid.c`. No que diz respeito às funções, colocamos a **assinatura** (ou **protótipo)** de cada função num ficheiro *header*. Quanto às *v*ariáveis, voltamos a declarar num ficheiro *header* as variáveis que forem globais, usando na declaração a palavra-chave `extern`. Ao aplicar `extern` na declaração duma variável informamos o compilador que a variável está definida algures noutro ficheiro fonte. Ao utilizar `extern` podemos aceder e alterar a variável global a partir de outros ficheiros fonte. Convém notar que em C **declarar** uma variável é diferente de **definir** uma variável. Exemplifica-se a seguir como se declara e define variáveis e funções.

Declarar e definir uma variável inteira `x`:

```
int x;
```

Declarar uma variável inteira x definida algures:

```
extern int x;
```

Declarar uma função maximo():

```
int maximo(int, int); /* assinatura ou protótipo da função */
```

Definir uma função maximo():

```
int maximo(int a, int b) {
   return (a>b)?a:b;
}
```

Questão: Qual é a assinatura duma função que soma dois `float`'s, passados como argumentos, e devolve o resultado dessa soma?

Após vermos o ficheiro com código fonte vamos mostrar o ficheiro *header*, por exemplo euclid.h, que vai conter os protótipos das funções `mdc()` e `ext_euclid()`.

```
/* Garantir que incluímos o ficheiro header apenas uma vez */
#ifndef __EUCLID_H__
#define __EUCLID_H__
/* Declarar as variáveis globais (definidas em euclid.c) */
extern int x, y;
/* Calcular o mdc */
int mdc ( int a, int b);
/* Calcular d = mdc(a,b) e resolver a equação ax+by=d */
int ext _euclid ( int a, int b, int *x, int *y);
#endif
```

Para utilizar o módulo Euclides vamos criar uma função `main()`, por exemplo no ficheiro main_euclid.c. Na função `main()` chamamos as funções `mdc()` e `ext_euclid()`, que estão definidas no ficheiro fonte `euclid.c`. Para isso, o `main()` executa os seguintes passos:

- Incluir o ficheiro *header* euclid.h:

  ```
  /* usa-se "" em vez de <> porque o ficheiro está em "." */
  #include "euclid.h"
  ```

- Ler da consola (*stdin*) os valores de a e b;
- Calcular d = mdc(a,b) com o algoritmo de Euclides:

  ```
  d = mdc(a, b);
  ```

- Apresentar na consola (*stdout*) o valor de d;
- Calcular d=mdc(a,b), x e y com algoritmo de Euclides estendido:

  ```
  d = ext_euclid(a, b, &x, &y);
  ```

  Os resultados ficam em d e nas variáveis x e y;
- Apresentar na consola os valores de d, x e y.

Para concluir o processo de criação dum módulo, vamos ver como compilar e depurar o módulo Euclides. Falta-nos implementar em C a função `main()`, com a funcionalidade que acabámos de descrever e guardar o código no ficheiro `main_euclid.c`. A seguir vamos compilar o ficheiro `main_euclid.c`. O resultado desta compilação será suficiente para obter um executável? A resposta é negativa dado que as funções `mdc()` e `ext_euclid()`, utilizadas no `main()`, estão definidas no ficheiro `euclid.c`. Logo, este ficheiro fonte também tem que ser compilado. Ao compilar vários ficheiros fonte, os vários ficheiros objeto gerados têm que ser ligados num único ficheiro executável. Este processo pode ser efetuado pelo `gcc` usando o seguinte comando:

```
gcc -g -O0 -Wall main_euclid.c euclid.c -o main_euclid
```

Por fim, podemos correr o executável com o comando `./main_euclid`

2.3 Alcance das Variáveis

Alcance é a zona do código em que a variável é válida. Em muitos casos, o alcance corresponde a um bloco com declaração de variáveis. As variáveis declaradas fora das funções possuem um alcance global. A definição duma função também possui um alcance. Podemos questionar-nos sobre qual é o alcance de cada variável no código seguinte.

```
int nmax = 20;
/* função main() */
int main(int argc , char **argv) { /* ponto de entrada */
   int a=0, b=1, c, n;
   printf("%2d: %d\n" ,1 ,a);
   printf("%2d: %d\n" ,2 ,b);
   for(n=3; n<nmax; n++) {
      c=a+b; a=b; b=c;
      printf("%3d: %d\n" ,n, c);
   }
   return 0; /* sucesso */
}
```

A variável `nmax` é válida em todo o módulo/ficheiro e as variáveis `a`, `b`, `c` e `n` são válidas na função `main()`. O programa escreve `20-1=19` linhas na consola. Quantas linhas são escritas na consola se o programa anterior for alterado para o código seguinte?

```
int nmax = 20;
int main ( int argc , char **argv) { /* ponto de entrada */
   int a=0, b=1, c, n, nmax =25;
   printf ( "%2d: %d\n" ,1 ,a );
   printf ( "%2d: %d\n" ,2 ,b );
   for (n=3; n<nmax; n++) {
      c=a+b;   a=b;   b=c;
      printf ( "%3d: %d\n" ,n, c );
   }
   return 0; /* sucesso */
}
```

Este programa escreve `25-1=24` linhas na consola, como por ser observado pela execução do Exercício **T02_ex1**.

Tendo em consideração o alcance das variáveis, o próximo programa apresenta um erro. Qual?

```
#include <stdio .h>
char *getMessage() {
  char msg[] = “Os apontadores sao divertidos mas ...”;
  return msg;
}
int main(void) {
   char *str = getMessage();
   puts(str);
   return 0;
}
```

O problema deste código reside no facto de o apontador devolvido pela função `getMessage()` ser inválido no `main()`, uma vez que ele aponta para uma variável `msg` fora de alcance.

2.4 Variáveis Estáticas

A palavra-chave `static` tem 2 significados, dependendo do local onde a variável estática for declarada. Fora duma função, variáveis e funções `static` apenas são visíveis dentro desse ficheiro, não têm alcance global. Ou seja, não se lhe pode aplicar a diretiva `extern` noutro ficheiro. Dentro duma função, as variáveis `static` são locais a essa função, são inicializadas apenas durante a inicialização do programa e não são reinicializadas cada vez que a função é chamada.

```
int funcao24() {
   static int umaVarPersistente=0; // só é inicializada a 0 uma vez
   ...
}
```

2.5 Variáveis em Registo

Durante a execução dos programas, os dados são processados colocando-os em registos do CPU. Isto acontece porque pelo menos 1 dos 2 operandos numa operação tem que estar em registo. Colocar explicitamente em registos os dados utilizados mais frequentemente minimiza o tempo extra associado com ler/guardar dados de/em memória. Podemos declarar explicitamente algumas variáveis em registo através da palavra-chave `register`, mas existem restrições à sua aplicação: (i) as variáveis a que se pode aplicar a diretiva `register` têm que ser dum tipo simples, sendo que o seu tamanho depende da arquitetura do CPU em causa; (ii) `register` aplica-se apenas a variáveis locais e argumentos de funções, e (iii) o compilador pode ignorar as declarações com `register` quando a declaração for num contexto não permitido ou quando houver demasiadas variáveis atribuídas a registos. Os registos não são memória, logo não é permitido usar um apontador para uma variável em registo. Apresenta-se agora o exemplo anterior, reformulado com variáveis em registo.

```
int main(register int argc , register char **argv) {
   register int a=0, b=1, c, n, nmax =20;
   printf("%3d: %d\n" ,1 ,a);
   printf("%3d: %d\n" ,2 ,b);
   for (n=3; n<=nmax; n++) {
      c=a+b; a=b; b=c;
      printf("%3d: %d\n" ,n, c);
   }
   return 0;
}
```

2.6 Programas com Parâmetros

Bibliografia: Seção 5.10 do livro *"The C Programming Language"*.

O C permite especificar que um programa ao ser executado aceita parâmetros. Para isso, deve-se definir a função `main()` com os seguintes parâmetros:

- `int argc`, que guarda o número de parâmetros com que se executou o programa (incluindo o nome do programa);
- `char **argv` ou `char *argv[]`, que é um apontador para um *array* de apontadores para *strings* de caracteres, as quais contêm os argumentos, um por *string*. `argv[0]` aponta para o nome do programa. Deste modo `argc` é pelo menos 1. `argv[1]`, `argv[2]`, até `argv[argc-1]` apontam para os verdadeiros, e opcionais, parâmetros com que se executou o programa, como se ilustra na figura 1.

Passar parâmetros a um programa apresenta vantagens:

- Permite executar o programa sobre dados variáveis sem o recompilar. Esta opção é vantajosa em relação a um programa que use `#define`'s.
- Permite executar o programa sobre dados variáveis sem os ler da consola. Esta opção revela-se vantajosa por exemplo quando não é conveniente/possível ter interação com humanos para introduzir dados. A execução de um programa com parâmetros é mais rápida e elegante do que um programa que lê dados da consola. Suponha que um programa é submetido, para execução, a uma máquina em que os programas podem ficar muito tempo em fila de espera a aguardar o início da execução. Neste cenário ler valores da consola não seria exequível.

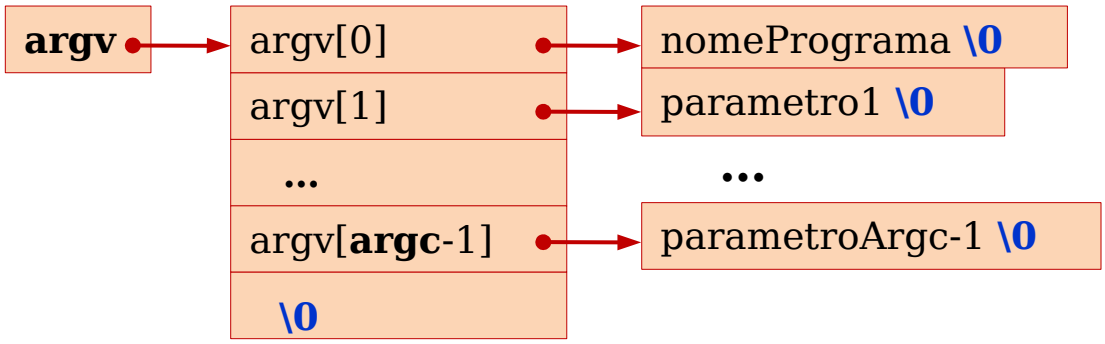


Figura 1 - Forma como são guardados os parâmetros passados quando se executa um programa.

Exemplo: Escrever um programa que faz o eco na consola dos parâmetros que lhe são passados. Ou seja, se o programa se chamar "`eco`", o comando "`eco Bom dia`" escreve na consola: `Bom dia`.

```c
/* faz o eco dos parâmetros passados ao programa */
#include  <stdio.h>
int main(int  argc,  char  *argv[]) {
    int  i;
    for (i=1; i<argc; i++) // os argumentos começam na posição 1
       printf("%s ", argv[i]);
    printf("\n");
    return  0;
}
```

2.7 Funções com uma Lista de Parâmetros Variável: Variádicas

Bibliografia: Seções B.7, A.7.3.2 e A.8.6.3 de *"The C Programming Language"*.

O ficheiro *header* `stdarg.h` permite definir **funções variádicas**, ou seja, funções que aceitam um número indefinido (variável) de argumentos. **`stdarg.h`** fornece as ferramentas necessárias para percorrer a lista de argumentos duma função cujo número e tipo são desconhecidos. Uma função variádica é declarada inserindo "..." depois dos argumentos conhecidos e tem que ter um ou mais argumentos conhecidos. Dada a seguinte assinatura para uma função variádica:

```c
int funcVar(int x, int y, ...);
```

Podemos mostrar dois exemplos de chamadas a essa função que são válidos e outro que é inválido:

```c
funcVar(3, 4);          // OK   - dois argumentos obrigatórios
funcVar(3, 4, 6.8);     // OK   - um argumento desconhecido (6.8)
funcVar(2);             // ERRO - falta y, um argumento obrigatório
```

A seguinte assinatura de função variádica é inválida:

```c
int errFuncVar(...);  // ERRO - a declaração deve ter pelo menos
                      //        um argumento obrigatório
```

Em `stdarg.h` é definido o tipo de dados `va_list` que pode ser usado para percorrer a lista de argumentos desconhecidos duma função variádica. O ficheiro *header* `stdarg.h` define também algumas macros para lidar com funções variádicas.

- `void va_start(va_list ap, argN)` inicializa a iteração sobre a lista de argumentos desconhecidos `ap`, para posterior utilização por parte de `va_arg()` e `va_end()`. `argN` é o nome do último argumento cujo tipo é conhecido;
- `type va_arg(va_list ap, type)` devolve um argumento da lista `ap`, com o tipo (modificado para) `type`. Em cada chamada de `va_arg()`, esta função modifica `ap` para que a próxima chamada devolva o argumento seguinte. Se não houver mais argumentos, ou se o tipo requerido (`type`) não for compatível com o tipo do próximo argumento na lista `ap`, ocorre um erro;

- `void va_end(va_list ap)` liberta a lista de argumentos `ap`. Uma chamada de `va_start()` deve ser acompanhada de uma chamada posterior a `va_end()`;
- `void va_copy(va_list dst, va_list src)` copia o conteúdo de uma lista de argumentos (`src`) para outra (`dst`).

Não há nenhum mecanismo pré-definido em C para determinar o número ou o tipo dos argumentos desconhecidos numa função variádica. Entre as convenções habitualmente usadas para ler corretamente os argumentos desconhecidos incluem-se: (i) usar uma *string* de formatação, com o estilo utilizado no `printf` e `scanf`, que especifica o tipo de cada argumento desconhecido; (ii) incluir um valor de sentinela/terminador no fim da lista de argumentos desconhecidos; e (iii) utilizar um argumento (o primeiro) para indicar o número de argumentos desconhecidos. A utilização de um valor de sentinela, ou terminador, permite detetar o fim dos dados, nos casos em que não há outro meio que indique explicitamente o tamanho desses dados. Por exemplo, em C usa-se o terminador '`\0`' apara saber onde acaba uma *string* de carateres.

Para aceder aos argumentos desconhecidos duma função variádica, deve declarar-se uma variável (`ap`) do tipo `va_list`:

```
va_list  ap;
```

Suponhamos que `lastArg` é o último argumento conhecido da função variádica. `ap` deve ser inicializada com *`va_start()`* antes de se aceder a qualquer argumento desconhecido:

```
va_start(ap,lastArg);
```

Cada chamada a `va_arg()` irá devolver o valor do próximo argumento desconhecido da lista `ap` e modificar `ap` de modo que a próxima chamada a `va_arg()` retorne o argumento seguinte:

```
valor = (type) va_arg(ap,type);
```

Após percorrer a lista de argumentos desconhecidos, e antes de sair da função variádica, deve libertar-se a lista `ap` chamando a função:

```
va_end(ap);
```

Apresenta-se a seguir um exemplo de função variádica.

```
#include <stdarg.h>
double  media(int count, ...) {
   va_list  ap;
   int      j;
   double   total = 0.0;
   // o 2º argumento de va_start é o nome do último argumento fixo
   va_start(ap, count);
   for(j=0; j<count; j++) {
      // é preciso indicar a va_arg o tipo (double) para efetuar cast.
      // va_arg  lê um argumento e avança ap para o próximo.
      total += (double)va_arg(ap, double); }
   va_end(ap);
   return total/count;
}
```

Esta função calcula a média dum número arbitrário de valores reais, passados como argumentos. A função não conhece o número de argumentos nem o seu tipo. Neste caso concreto, exige-se que o tipo dos argumentos seja `double`, e o número de argumentos seja passado no primeiro e único argumento com nome conhecido. Noutros casos, como por exemplo no `printf`, o número e o tipo dos argumentos é obtido a partir duma *string* de formatação. Em ambos os casos, o sucesso da estratégia usada está dependente de o programador fornecer a informação correta. Se forem passados menos argumentos do que a função está à espera, ou se o tipo dos argumentos passados for incorreto, isso pode implicar a leitura de endereços de memória inválidos e introduzir vulnerabilidades no código.

Exercício **T02_ex2**: Escrever uma função que calcula a média de um conjunto de valores `double`, com cardinalidade variável, passados como argumentos.

Exercício **T02_ex3**: Escrever uma função que implementa uma versão simplificada do `printf()`.

2.8 Estruturas

Uma **estrutura** é uma coleção de variáveis relacionadas, possivelmente de tipos diferentes, agrupadas sob um único nome. Este é um exemplo de **composição**, em que se constroem estruturas complexas a partir de outras mais simples. A próxima tabela mostra 2 exemplos de estruturas.

```
struct ponto {
   int  x ;
   int  y ;
   } ;
   /* convém notar o ";" no fim */
```

```
struct   empregado {
   char  pNome[100] ;
   char  uNome[100] ;
   int   idade ;
   } ;
   /* membros de tipos diferentes */
```

Utilizando a palavra chave `struct` define-se um tipo de dados novo. O nome da estrutura é opcional, tal como em:

```
struct {...} x,y,z;
```

As variáveis declaradas dentro da estrutura são os seus **membros**. *As v*ariáveis do tipo `struct` podem ser declaradas como qualquer outro tipo de dados nativo:

```
struct ponto ptA;
```

A inicialização faz-se especificando o valor de cada membro:

```
struct ponto ptA = {10,20};
```

A operação de atribuição (=) copia cada membro/valor da estrutura de origem (`{10,20}` neste caso) para a de destino (`ptA` neste caso). Ao copiar-se os valores de uma estrutura para outra, deve ter-se cuidado quando um membro for um apontador. Normalmente um apontador não deverá ser copiado, porque assim teríamos duas estruturas com um membro a apontar para a mesma posição de memória. O mais frequente nestas situações, é copiar todos os membros menos o apontador e colocar o apontador da estrutura de destino a apontar para uma posição de memória diferente da apontada pelo membro da estrutura de origem.

A tabela a seguir mostra mais 2 exemplos de estruturas.

```
struct triangulo {
   struct ponto  ptA;
   struct ponto  ptB;
   struct ponto  ptC;
};
/* membros podem ser estruturas */
```

```
struct elemento_lista {
  int  dados ;
  struct elemento_lista *prox;
  };
/* os membros podem ser do mesmo tipo */
/* da estrutura a que pertencem       */
```

Para aceder a um membro de uma estrutura utiliza-se o **operador '.'**. No próximo exemplo acede-se com o operador "." aos membros `x` e `y` da estrutura `pt`:

```
struct ponto pt={10,20};
int x=pt.x;
int y=pt.y;
```

Se uma estrutura estiver aninhada noutra(s), para aceder a um membro seu são necessários vários operadores '.'. Um exemplo:

```
struct retangulo {
   struct ponto se;  /* vértice superior esquerdo */
   struct ponto id;  /* vértice inferior direito  */
};
struct retangulo ret;
int x_se= ret.se.x;   /* estrutura aninhada exige 2 operadores '.' */
int y_se= ret.se.y;
```

2.9 Apontadores para Estruturas

Ao passar uma estrutura como argumento a uma função, a função recebe uma cópia completa dessa estrutura. Em estruturas de grande dimensão é mais eficiente passá-las por referência (apontador) às funções, como acontece no código em baixo em que se passa o endereço da estrutura `pt` (`&pt`) à função `funcT()`.

```
void funcT(struct ponto *);
struct ponto pt;
funcT(&pt);          /* Notar o operador & antes da variável pt */
```

Os membros duma estrutura, passada por referência, podem ser acedidos a partir do apontador usando o **operador '->'**.

Ao imlementar a funçao `funcT()` do exemplo anterior, acede-se aos campos `x` e `y` da estrutura `p`, usando para isso o apontador `pp` e o operador "`->`".

```
struct ponto p = {10,20};
void funcT(struct ponto *pp){
   pp->x = 10;       /* faz o mesmo que p.x=10 se p fosse passada por valor */
   int y = pp->y;    /* faz o mesmo que y=p.y  se p fosse passada por valor */
}
```

Questão: Haverá outra forma de aceder aos membros da estrutura `p` passada por referência à função `funcT()`?

Resposta: Sim, usando o conteúdo da posição apontada pelo argumento `pp` como início da estrutura. Ou seja, podemos aceder aos membros `x` e `y` através de `(*pp).x` ou `(*pp).y`, como se mostra no código seguinte.

```
struct ponto p = {10,20};
void funcT(struct ponto *pp){
   (*pp).x = 10;     /* faz o mesmo que p.x=10 se p fosse passada por valor */
   int y = (*pp).y; /* faz o mesmo que y=p.y  se p fosse passada por valor */
}
```

Questão: Porque são necessários os parênteses nas duas linhas de código do corpo da função `funcT()`?

Resposta: Os parênteses são necessários nas atribuições porque o operador '`.`' tem maior prioridade que o operador '`*`', logo "`*pp.x`" é diferente de "`(*pp).x`"

Questão: Sem os parênteses o código seria compilado com sucesso?

Resposta: Sem os parênteses o compilador assinalava erros porque, por exemplo, `*pp.x` que é equivalente a `*(pp.x)` estaria incorreto. `pp` é um apontador para uma estrutura e não uma estrutura, logo não se pode usar o operador '`.`' com um apontador para uma estrutura.

2.10 *Arrays* de Estruturas

Para declarar um *array* de `int`'s com 10 posições utiliza-se a seguinte linha de código:

```
int x[10];
```

Declarar um *array* de estruturas do tipo `struct ponto` com 10 posições é similar:

```
struct ponto p[10]; // ponto tem 2 membros int
```

Quando o número de posições dum *array*, por exemplo um *array* de inteiros, for reduzido podemos iniciá-lo logo na sua declaração:

```
int x[4]={0,20,10,2};
```

Inicializar um *array* de estruturas do tipo `struct ponto` é similar:

```
struct ponto p[3] = {0,1,10,20,30,12};
struct ponto p[3] = {{0,1},{10,20},{30,12}}; // inicialização equivalente
```

Neste exemplo, o valor **0** é atribuído ao membro que aparece em primeiro lugar na definição da estrutura `struct ponto` (membro `x`) e pertencente à primeira posição da estrutura `p[0]`. Ou seja `p[0].x=0`. O valor **1** é atribuído ao membro que aparece em segundo lugar na definição da estrutura `struct ponto` (membro `y`) e pertencente à primeira posição da estrutura `p[0]`. Ou seja `p[0].y=1`. Esta sequência de duas atribuições a `x` e `y` repete-se para os restantes elementos do *array*: `p[1]` e `p[2]`.

2.11 Tamanho das Estruturas

O tamanho de uma estrutura é maior ou igual à soma dos tamanhos dos seus membros. O tamanho pode ser maior quando usamos **alinhamento**. Considere a declaração da estrutura `ex1`:

```
struct ex1 {
   char c1;
   char c2; };
```

Cada membro de uma estrutura pode ser explicitamente alinhado num endereço múltiplo de `N` usando uma extensão do compilador, neste caso `__attribute__((aligned(N)))` do `gcc`. Na instrução seguinte, o membro `nomeVar` fica alinha num endereço múltiplo de `16`. O mesmo é dizer que `nomeVar` está armazenado a partir dum endereço múltiplo de `16`.

```
tipoV  nomeVar __attribute__((aligned(16)));   /* no gcc */
```

Questão: Se alinharmos cada membro da estrutura `ex1` num endereço múltiplo de 4, quantos bytes ocupa `ex1`? Isto é importante?
Resposta: O tamanho da estrutura `ex1` sem alinhamento é 2 bytes e com alinhamento passa a ser 8 bytes. Há vários casos em que interessa que todos os membros estejam armazenados a partir dum endereço múltiplo de 2^n, por exemplo 2^n=4. É o que ocorre quando queremos utilizar instruções vetoriais. Uma instruçao vetorial aplica a mesma operação a um conjunto de valores (2, 4, 8, ...) armazenados em posições de memória consecutivas. Consideremos que temos um ciclo em relação à variável `i`, que em cada iteração faz `z[i]=x[i]+y[i]`. Se os *arrays* `x`, `y` e `z` forem do tipo inteiro, ocupando cada elemento 4 bytes, mas não estiverem alinhados em endereços múltiplos de 4, o compilador não gera código vetorial para as instruções *assembly* que implementam a operação `z[i]=x[i]+y[i]`. Ao contrário, se os endereços dos *arrays* `x`, `y` e `z` estiverem alinhados em endereços múltiplos de 4, o compilador pode gerar instruções vetoriais que executam ao mesmo tempo 4 instruções, por exemplo 4 somas equivalentes a `x[i]+y[i]`, `x[i+1]+y[i+1]`, `x[i+2]+y[i+2]` e `x[i+3]+y[i+3]`. Neste caso, se o compilador vetorizar completamente o corpo do ciclo, irão ser necessárias 4 vezes menos iterações dado que em cada iteração faz 4 operações ao mesmo tempo. Logo o ciclo irá demorar a executar aproximadamente ¼ do tempo.

2.12 Definição de Tipos de Dados com `typedef`

Bibliografia: Seção 6.7 do livro "*The C Programming Language*".

Com `typedef` podemos criar novos (nomes para) tipos de dados existentes. Por exemplo podemos passar a designar o tipo `long long` por `LLong`.

```
typedef   long long   LLong;
```

Deste modo, `LLong` passa a ser um sinónimo de `long long` e pode ser usado em declarações, argumentos de funções, etc, do mesmo modo que um `int` pode. Exemplo:

```
LLong size, *sizes;
```

A próxima instrução torna *`String`* um sinónimo de `char*`:

```
typedef     char*     String;
```

Podemos agora ver três casos de utilização do novo tipo `String`: numa declaração de variáveis, num argumento de uma função e na modificação de um tipo de dados para outro tipo. Esta modificação de tipo é designada em C por ***cast***.

```
String p, linhas[MAX];
int strcmp(String s1, String s2);
p = (String)malloc(100); // aplicar (String) significa fazer cast
```

Sintaticamente, `typedef` é similar a `extern`, `static`, ou `register`.

2.12.1 Definição de tipos de dados e estruturas: um exemplo

Exercício: Escrever um programa que utilize a estrutura `struct empregado` e define um novo tipo de dados `Empregado` à custa de `struct empregado`. A função `main()` deve (i) declarar uma variável `e` do tipo `Empregado`, (ii) preencher os membros da estrutura `e` com valores pedidos ao utilizador e (iii) chamar a função `printDadosEmpregado(Empregado *e)`. Escrever a função `printDadosEmpregado()` que escreve na consola os dados do `Empregado e`, passado por referência à função.

```
struct empregado {
   char  pNome[100] ;
   char  uNome[100] ;
   int   idade ;
};
typedef  struct empregado  Empregado;
int main() {
   Empregado  e;
   printf("Primeiro nome do empregado: ");
   scanf("%s",e.pNome); // Porque não é preciso o '&' ?
   printf("Ultimo nome do empregado: ");
   scanf("%s",e.uNome);
```

```
    printf("Idade do empregado: ");
    scanf("%d",&(e.idade));
    printDadosEmpregado(&e);
    return 0;
}
void printDadosEmpregado (Empregado *emp) {
    printf("Nome: %s %s\n", emp->pNome, emp->uNome);
    printf("Idade: %d\n", emp->idade); }
```

2.13 Acesso a Ficheiros

Em C para aceder a ficheiros utilizamos funções da biblioteca `stdio.h`. Esta biblioteca dispõe de funções para abrir/iniciar o acesso a um ficheiro, ler do ficheiro, escrever no ficheiro, fechar/terminar o acesso ao ficheiro. O acesso pode ser feito em modo texto ou binário.

2.13.1 Acesso a ficheiros: abrir

A função declarada em `stdio.h` que é utilizada para abrir um ficheiro é `fopen()`.

```
FILE* fopen(char nome[], char modo[]);
```

`fopen()` devolve um apontador que funciona como canal de comunicação com o ficheiro. Se o ficheiro não existir devolve `NULL`. O apontador devolvido é do tipo `FILE*` e aponta para uma estrutura que contém informação sobre o ficheiro: localização, posição atual de acesso, modo de acesso atual, etc. O utilizador só tem que declarar uma variável do tipo `FILE*`:

```
FILE *fp;
```

A função `fopen()` possui 2 argumentos:

- `nome`, que é uma *string* de carateres contendo o nome do ficheiro a abrir;
- `modo`, uma *string* que indica o tipo de acesso que pretendemos ter ao ficheiro: "r" para leitura,"w" para escrita,"a" para escrever no fim do ficheiro, “r+”, “w+” e “a+”. Podemos acrescentar "b" nos tipos já mencionados para aceder a um ficheiro em modo binário. O modo “r+” abre o ficheiro para leitura e escrita, sendo a posição de acesso colocada no início do ficheiro. O modo “w+” abre o ficheiro para leitura e escrita. O ficheiro é criado se não existir, caso contrário será truncado. A posição de acesso é colocada no início do ficheiro. Por fim, o modo “a+” abre o ficheiro para ler e acrescentar (escrever no fim do ficheiro). O ficheiro é criado se não existir. A posição inicial de leitura é colocada no início do ficheiro e a escrita é sempre acrescentada no fim do ficheiro.

2.13.2 Acesso a ficheiros: fechar

A função `fclose()` fecha o canal de comunicação com um ficheiro aberto, libertando os recursos alocados ao acesso. `fclose()` é chamada automaticamente ao terminar um programa, para fechar o acesso a todos os ficheiros abertos pelo programa. Apesar disso, é recomendável o utilizador fechar explicitamente os ficheiros abertos, quando já não precisa de lhe aceder. A assinatura da função é:

```
int fclose (FILE* fp);
```

2.13.3 Acesso a ficheiros: leitura

Para ler de um ficheiro a biblioteca `stdio.h` disponibiliza alternativas, tais como: `fgetc()`, `fgets()` e `fscanf()`. A função `fgetc()` lê o próximo carater a partir do ficheiro apontado por `fp`. Em caso de erro, ou fim de ficheiro, devolve `EOF`. A assinatura da função é:

```
int getc(FILE* fp);
```

A função `fgets()` lê uma linha, até um máximo de `maxlen-1` carateres, a partir do ficheiro apontado por `fp` e guarda-a em `line[]`. Devolve um apontador para o *array* de carateres onde fica guardada a linha lida, ou então devolve `NULL` se chegar ao fim do ficheiro. A assinatura da função é:

```
char[] fgets(char line[], int maxlen, FILE* fp);
```

A função `fscanf()` permite ler dados de forma formatada, partir do ficheiro apontado por `fp`, guardando os dados lidos em `arg1`, `arg2`, etc. Esta função é similar a `scanf()` e `sscanf()`. Devolve o número de itens lidos corretamente. A assinatura de `fscanf()` é:

```
int fscanf(FILE* fp, char format[], arg1, arg2,...);
```

2.13.4 Acesso a ficheiros: escrita

Tal como acontece com a leitura, para escrever em ficheiro a biblioteca `stdio.h` também dispõe de alternativas, entre as quais se incluem: `fputc()`, `fputs()` e `fprintf()`. A função `fputc()` escreve o carater `c` no ficheiro apontado por `fp` e devolve o carater escrito ou `EOF` em caso de erro. A assinatura de `fputc()` é:

```
int fputc(int c, FILE* fp);
```

A função `fputs()` escreve a linha `line[]` no ficheiro apontado por `fp`. Em caso de sucesso devolve `zero`, senão devolve `EOF`. A assinatura da função é:

```
int fputs(char line[], FILE* fp);
```

A função `fprintf()` permite escrever os itens `arg1`, `arg2`,etc, de acordo com o formato especificado, no ficheiro apontado por `fp`. Esta função é similar a `printf()` e `sprintf()`.

```
int fprintf(FILE* fp, char format[], arg1, arg2, ...);
```

2.13.5 Acesso a ficheiros: `stdin`, `stdout`, `stderr`

Quando se executa um programa, o sistema operativo abre 3 ficheiros e disponibiliza apontadores para eles. Os ficheiros são: entrada *standard*, saída *standard* e erros *standard*. Os apontadores correspondentes são: `stdin`, `stdout` e `stderr`. Normalmente `stdin` está ligado ao teclado e `stdout` e `stderr` estão ligados à consola/terminal. Contudo, `stdin` e `stdout` podem ser redirecionados para ficheiros.

2.13.6 Acesso a ficheiros: exemplo

Exercício: Assumindo que dispomos de um ficheiro texto `lista.txt` que guarda os dados de `Empregado's`, um por linha, com o seguinte formato de linha: `PrimeiroNome SegundoNome Idade`. Pede-se para escrever um programa em C que lê cada linha do ficheiro para uma variável do tipo `Empregado`, a qual foi definida no exemplo anterior. A leitura das linhas deve ser feita com a função:

```
void LerEmpregados(FILE *fp);
```

Após ler cada linha do ficheiro, a função `LerEmpregados()` escreve no ecrã os dados do empregado lido. Pode reutilizar-se a função `printDadosEmpregado(Empregado *e)`, definida no exemplo anterior.

```
// Código reutilizado do exemplo anterior
#include <stdio.h>

struct empregado {
   char  pNome[100] ;
   char  uNome[100] ;
   int   idade ;
} ;

typedef  struct empregado  Empregado;

void printDadosEmpregado(Empregado *emp) {
   printf("Nome empregado: %s %s\n", emp->pNome, emp->uNome);
   printf("Idade empregado: %d\n", emp->idade);
}

int main(int argc, char *argv[]) { // argv[1] = ficheiro com empregados
   FILE *fp;
   if (argc < 2) {
      printf("Passe o nome do ficheiro como argumento!\n");
      return 1;
   }
   else if ((fp = fopen(argv[1], "r")) == NULL) {
      printf("Nao e' possivel abrir o ficheiro %s\n", argv[1]);
      return 2;
   }
   else {
      LerEmpregados(fp);
      fclose(fp);
      return 0;
   }
}
```

```
// Formato do ficheiro com empregados:
//      PrimeiroNome_1 UltimoNome_1 Idade_1
//      ...
//      PrimeiroNome_N UltimoNome_N Idade_N
//
void LerEmpregados(FILE *fp) {
   Empregado e ;
   int       n ;
   do {
      n=fscanf(fp,"%s %s %d", e.pNome, e.uNome, &(e.idade));
      if (n == 3)
         printDadosEmpregado(&e);
      else if (n==EOF)
         printf("*** Fim de ficheiro!\n");
      else // n<3 && n!=EOF
         printf("*** Linha incompleta!\n");
   } while ( n != EOF );
}
```

2.14 Alocação Dinâmica de Memória

Em C, quando precisamos de definir estruturas de dados cuja dimensão `DIM` só é conhecida em tempo de execução temos que declarar um apontador e reservar espaço em memória durante a execução (dinamicamente) para esses `DIM` elementos. Esta tarefa, considerada de baixo nível, é da responsabilidade do programador. Para a realizar podemos recorrer a uma das funções da biblioteca `stlib.h`: `malloc()`, `calloc()`, etc. A função comummente utilizada é o `malloc()`, a qual reserva um bloco de memória com tamanho n bytes. A assinatura da função é:

```
void* malloc(size_t n); // size_t <=> unsigned int
```

A função `malloc()` devolve um apontador para o bloco de memória reservado e por inicializar, em caso de sucesso, ou então devolve `NULL`, em caso de insucesso. O valor devolvido deverá ser convertido (*cast*) para o tipo de dados a guardar nessa memória usando `(tipo)` antes do nome da função. Exemplo em que se utiliza `malloc()` para reservar espaço para 100 inteiros:

```
int* ip = (int*)malloc(sizeof(int)*100);
```

Em alternativa, pode utilizar-se a função `calloc()` para reservar espaço em memória para um *array* com `n` elementos, cada um ocupando `size` bytes. A função inicializa a memória a `0`. A assinatura da função `calloc()` é:

```
void* calloc(size_t n, size_t size);
```

Exemplo de utilização de `calloc()`:

```
int* ip = (int*)calloc(100, sizeof(int));
```

Quando não precisamos mais da memória reservada com com `malloc()` ou `calloc()` devemos libertá-la, recorrendo para isso à função `free()` cuja assinatura é:

```
void free(void*);
```

Exemplo em que reservamos espaço para 100 `int`'s e no final da sua utilização libertamos esse espaço:

```
int* ip;
ip =(int*)malloc(sizeof(int)*100);
... // utilizar ip
free(ip);
```

Um erro comum quando se trabalha com alocação dinâmica de memória é aceder à memória depois de chamar `free()`.

2.15 Mais Informação sobre Apontadores

A forma nativa de implementar um ***array*** em C consiste em usar um apontador para um bloco de memória contígua. Consideremos um *array* de 8 inteiros:

```
int arr[8];
```

Para aceder a `arr` utilizamos o **operador de *array*** `[]`:

```
int a = arr[0];
```

`arr` funciona como apontador para o elemento 0 do *array*, como se percebe pela equivalência entre as duas instruções seguintes:

```
int *pa = arr;    <=>  int *pa = &arr[0];
```

Contudo convém notar a diferença entre `arr` não poder ser modificado (usando uma atribuição), enquanto o apontador `pa` pode.

Um **apontador** representa o endereço de memória onde está guardada uma variável. Alguns exemplos:

`int *pn;` → declara um apontador para um `int`

`struct div_t *pdiv;` → declara um apontador para uma estrutura do tipo `div_t`

Nas linhas de código que se seguem podemos ver como se faz o endereçamento e indireção recorrendo a apontadores.

```
double pi   = 3.14159;      // variável simples: pi
double *ppi = &pi;          // o apontador ppi fica com o endereço de pi: &pi
printf("pi = %lf\n", *ppi); // conteúdo da variável apontada por ppi: *ppi
```

Indireção consiste em aceder a uma variável através dum apontador, ou seja, consiste em aceder a uma variável de forma indireta. No exemplo anterior, acedemos ao conteúdo de `pi` (`*ppi`) através dum apontador `ppi` que contém o endereço de `pi`. Ou seja, `*ppi=pi`.

2.15.1 Aritmética dos apontadores

Dada a declaração `int *pa = arr;`, embora o apontador `pa` não seja um `int`, podemos lhe somar ou subtrair um valor `int`. Por exemplo, `pa+i` aponta para `arr[i]`. O valor do endereço resultante da soma `pa+i` é igual ao endereço de `pa` mais `i` vezes o tamanho dos dados para os quais `pa` aponta. No caso anterior o tipo=int, logo o tamanho=4. Suponhamos que `arr[0]` está no endereço de memória 100. Então `arr[3]` estará no endereço 100+3*4=112.

Questão: Dadas três linhas de código

```
int arr[8]; // arr[0] está no endereço de memória 100
int  *pa = arr;
char *pc = (char *)pa;
```

que valor de `i` satisfaz: `(int *)(pc+i) == pa+3` ?

Para responder a esta pergunta vamos recorrer à figura 2, a qual mostra a zona de memória correspondente ao *array* `arr`, armazenado a partir da posição de memória com endereço 100. Como colocamos dois apontadores de tipo diferente a apontar para o início do *array* `arr`, `pa` é um apontador para `int`'s e `pc` é um apontador para `char`'s, a memória por eles endereçada é vista de duas formas: como um *array* de elementos do tipo `int` (parte esquerda da figura) e como um *array* de elementos do tipo `char` (parte direita da figura). Como se pode ver pela figura, o valor de `i` que satisfaz a igualdade `pc+i==pa+3=112`, é `i=12`.

Solução: Exercício T04_ex0

2.15.2 Apontadores para apontadores

Em determinadas situações podemos precisar de dois níveis de indireção, ou seja, obter o conteúdo duma variável `n` recorrendo a dois apontadores: um apontador `pn` contém o endereço da variável `n`, e o segundo apontador `ppn` contém o endereço do primeiro apontador (`pn`). Neste caso, `ppn` é um apontador para outro apontador. Um exemplo:

```
int n   = 3;
int *pn  = &n;  // pn é um apontador para n (pn fica com o endereço de n)
int **ppn = &pn; // ppn é um apontador para o endereço de n
```

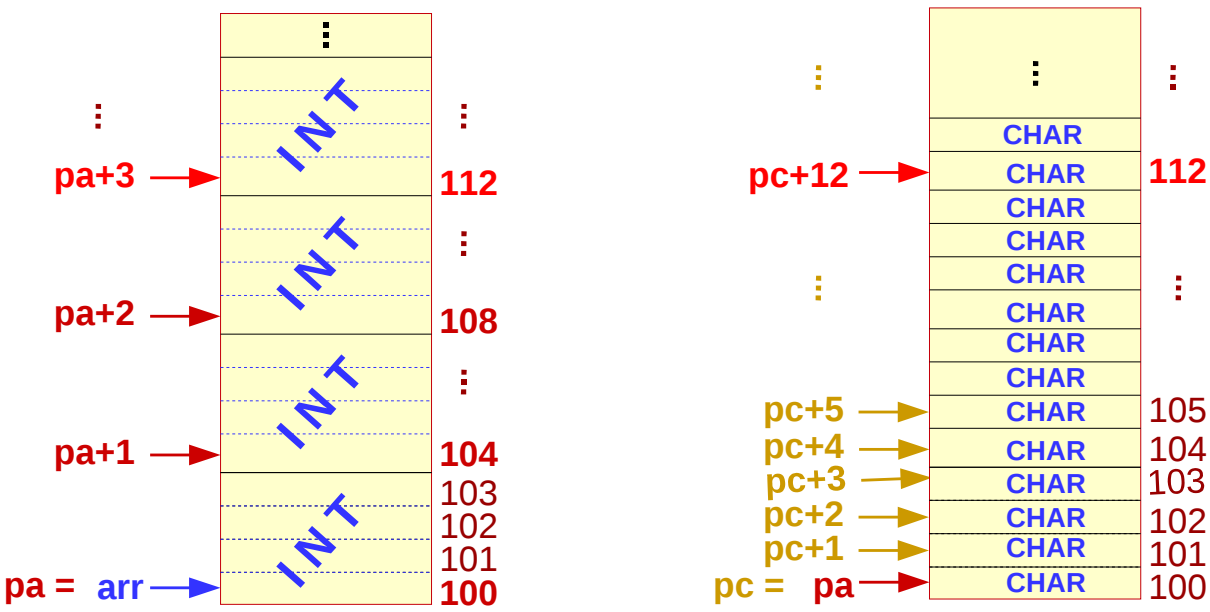


Figura 2 - Mapa de memória para identificar que valor de `i` *satisfaz a igualdade* `(int *)(pc+i)=pa+3`.

A figura 3 exemplifica graficamente a dupla indireção que permite chegar ao conteúdo da variável `n` passando por 2 apontadores: `ppn` e `pn`. Se a variável `n` contiver o valor `3` e estiver armazenada na posição `5070`, o apontador `pn` contém o endereço de `n` e fica com o valor `5070`. Se `pn` estiver armazenada na posição `5030`, o apontador `ppn` contém o endereço de `pn` e fica com o valor `5030`. Neste caso, o apontador para apontador `ppn` está armazenado na posição `5000`.

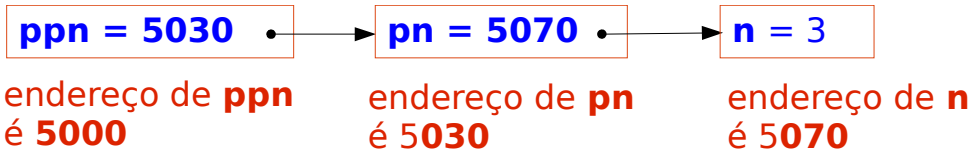


Figura 3 - Exemplo de dupla indireção.

Duas aplicações em C de apontadores para apontadores são os *arrays* de apontadores e os *arrays* de *strings*. Vamos ver agora um exemplo de aplicação de apontadores para apontadores.

Questão: O que faz a função `swap()` na sua forma habitual?

```
void swap (int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

Resposta: Como seria de esperar, `swap(&x,&y)` troca o conteúdo das variáveis `x` e `y`, recorrendo a uma variável auxiliar `temp` para efetuar a troca.

Questão: Qual a diferença de `swap1()` em relação a `swap()`?

```
void swap1 (int **a, int **b) {
   int *temp = *a;
   *a = *b;
   *b = temp;
}
```

Resposta: Neste caso, como a função `swap1()` recebe como argumentos apontadores para os apontadores que contêm o endereço das variáveis a trocar (`a` e `b`), em vez de se trocar o conteúdo das variáveis `a` e `b` troca-se o conteúdo dos apontadores para elas (`*a` e `*b`).

A resposta a esta questão pode ser confirmada no exercício **T04_ex1**.

2.15.3 *Arrays* de apontadores

Um *array* de apontadores é um vetor contendo apontadores. Ao declarar um *array* de apontadores temos que indicar o tipo elementos a incluir no *array,* por exemplo `tipoVar *`, e indicar que é um *array*, usando o operador "`[]`". Dois exemplos de *arrays* de apontadores:

```
int *arr[20];  // um array de apontadores para int's
char *arr[10]; // um array de apontadores para char's
```

Os apontadores guardados num *array* podem apontar para outros *arrays*. A próxima linha de código pode ser vista como a declaração de um *array* de apontadores para *arrays* de carateres, ou seja, a declaração de um *array* de apontadores para *strings*.

```
char *strs[10];
```

Exercício **T04_ex2**: Assuma que dispomos dum *array* `int arr[100]` que guarda números inteiros. O que se pretende com este exercício é obter uma versão ordenada do *array*, mas sem modificar o *array* original `arr`. Os valores devem ser ordenados de forma crescente, ou seja, do menor para o maior.

Para resolver o exercício vamos declarar um *array* de apontadores `int* array_ordenado[100]` para guardar o apontador para cada elemento de `arr` e ordenar os apontadores em vez dos próprios números. Esta abordagem é adequada para ordenar *arrays* contendo elementos de grande dimensão (como por exemplo *strings*). Na solução do exercício vamos utilizar um algoritmo de **ordenação por inserção**, ordenando os elementos do *array* por ordem crescente do valor.

```
// Percorrer o array até encontrar um elemento fora de ordem
// Deslocar esse elemento para a sua posição correta
// Continuar a percorrer o array
void ordenacao_insercao (void) {
   unsigned int  i, len;
   len = sizeof(arr)/sizeof(int); // número de elementos do array 'arr'
   for(i=1; i<len  i++)
      if(*array_ordenado[i] < *array_ordenado[i-1])
         deslocar_elemento(i);
}

// Deslocar o elemento da posição anterior do array para a atual (i),
// até se atingir o ponto de inserção: início do array (i=0) ou
// a posição em que o elemento anterior é menor que o elemento atual
void deslocar_elemento (unsigned int i) {
   int *end_i;
   for (end_i=array_ordenado[i];
        (i!=0) && (*array_ordenado[i-1] > *end_i); i--) {
      // Deslocar o elemento/apontador anterior para a posição atual
      array_ordenado[i] = array_ordenado[i-1];
   }
   // Inserir o elemento/apontador a ordenar na posição correta
   array_ordenado[i] = end_i;
}
```

2.15.4 *Arrays* de *strings*

Um *array* de *strings* guarda apontadores para *arrays* de `char`'s. Cada *string, ou array* de `char`'s, pode ter um tamanho diferente. Exemplos:

```
char str1[] = "bom dia";  /* tamanho = 8, contando com o terminador '\0' */
char str2[] = "ola";      /* tamanho = 4, contando com o terminador '\0' */
char str3[] = "adeus";    /* tamanho = 6, contando com o terminador '\0' */
char *strArray[] = {str1, str2, str3};
```

Convém notar que `strArray` guarda apenas os apontadores para as três *strings*, não os carateres das *strings* apontadas.

2.15.5 *Arrays* multidimensionais

A linguagem C também permite declarar *arrays* multidimensionais, especificados através da notação `[]...[]`. Um exemplo é:

```
int matriz[20][30];
// define um array 2D de int's com 20x30 elementos
```

Podem definir-se *arrays* com mais do que duas dimensões. Exemplo com 4 dimensões:

```
char matriz4D[5][6][10][4];
```

Questão: Qual é número de elementos do *array* `matriz4D`?

Os *arrays* multidimensionais são "retangulares", enquanto os *arrays* de apontadores permitem definir estruturas com uma forma irregular.

2.15.6 Apontadores `void`

O C não permite declarar nem utilizar variáveis **`void`**. O tipo `void` só pode ser usado como o tipo do valor devolvido pelas funções, como o tipo dos parâmetros das funções e na declaração de apontadores `void`.

Questão: Em que casos é pertinente passar apontadores `void` a uma função?

Os apontadores `void` apresentam uma vantagem: podem apontar para qualquer tipo de dados.

```
int   x;   void* p1=&x;     // p1 aponta para um int
float f;   void* p2=&f;     // p2 aponta para um float
```

Os apontadores `void` não podem ser dereferenciados. Ou seja, antes de aceder ao conteúdo do apontador, ele dever ser convertido (*cast*) para um tipo não `void`.

```
void* p1; printf ("%d",*p1);                    // acesso inválido a *p1
void* p2; int* px=(int*)p2; printf ("%d",*px); // acesso   válido a *p2
```

O operador de referência é o "`*`" que se coloca antes duma variável/parâmetro para indicar que essa variável/parâmetro passa a funcionar como um apontador, como acontece por exemplo em `int *p`. **Dereferenciar** um apontador significa obter o valor endereçado por esse apontador, como por exemplo ao fazer `x=*p`.

2.15.7 Apontadores para funções

Em algumas linguagens de programação, as funções são **entidades de primeira classe** porque podem ser usadas sem restrições: passadas como parâmetros, devolvidas como resultado de funções, etc. Diz-se que as funções são de primeira classe se puderem ser usadas como parâmetros e ser devolvidas como resultado em funções. Em C, as funções não são uma entidade de primeira classe, como as variáveis, mas é possível declarar apontadores para funções. O **apontador para uma função** é a declaração de um apontador que contém o endereço do início da função.

Vamos realçar a diferença entre duas declarações que são quase similares sintaticamente:

```
// Função que devolve um apontador para um int e tem com argumentos
// um int e um float
int *func1(int a, float b);

// Apontador para uma função que devolve um int e tem com argumentos
// um int e um float
int (*func2)(int a, float b); // notar os ()
```

Vamos analisar mais dois exemplos de declaração de um apontador para uma função. O primeiro exemplo declara um apontador para uma função que devolve um `int` e tem um argumento do tipo `int`. O segundo declara um apontador para uma função que devolve um `int` e tem dois argumentos do tipo `void*`.

```
int (*funcPtr) (int);

int (*funcPtr) (void*, void*);
```

Um apontador para função pode ser escrito, passado a funções, devolvido por funções e colocado em *arrays*.

Questão: Se quiséssemos que o algoritmo de ordenação por inserção, descrito atrás, funcionasse com um *array* de valores de qualquer tipo, como o implementávamos?

Resposta: A forma mais elegante consiste em utilizar um apontador para uma função de comparação que pode ser substituído por um apontador para a função adequada ao tipo de dados que se pretende comparar.

Exercício **T04_ex3**: Escrever um programa que chama uma função `funcS()`, que recebe dois argumentos do tipo `int` e escreve o valor dos argumentos no ecrã. A escrita no ecrã deve ser feita de duas maneiras: através do nome da função e através de um apontador para a função.

```
#include <stdio.h>
int funcS (int a, int b) {
   printf("A = %d\n",a);
   printf("B = %d\n",b);
   return 0;
}

int main(void) {
   int (*fptr)(int,int); // Declarar um apontador para função
   fptr = funcS;         // Atribuir o endereço da função ao apontador
   fptr(2,3);     // #1
   funcS(2,3);    // #2
   return 0;
}
```

Como o resultado escrito na consola é igual nos dois casos, **#1** e **#2**, concluímos que chamar a função através de um apontador (**#1**) produz o mesmo resultado que chamar a função através do nome (**#2**).
Quando é preciso usar uma função `funcX()` numa dada posição dum programa, basta colocar uma chamada a essa função nessa posição do código fonte. Mas o que fazer quando não sabemos, na fase de compilação, qual a função a chamar? Ou seja, o que fazer quando a decisão sobre qual a função a chamar só puder ser tomada em tempo de execução?

Nestes casos podemos usar um `switch` para selecionar uma função de entre um conjunto de funções. Mas existe outra forma: utilizar um apontador para função.

Exercício **T04_ex4**: Vamos revisitar o programa que realiza uma de 4 operações aritméticas: +, -, x, /. Vimos como o problema pode ser resolvido através de um `switch`. Vamos agora ver como o problema pode ser resolvido utilizando um apontador para função.

```
// Função para cada uma das 4 operações aritméticas.
// Uma destas funções será selecionada durante a execução.
float Somar(float a, float b) {
   return a+b;
}

float Subtrair(float a, float b) {
   return a-b;
}
float Multiplicar(float a, float b) {
   return a*b;
}

float Dividir(float a, float b) {
   return a/b;
}

// pt2Func é um apontador para função.
// A função apontada possui 2 argumentos do tipo float e devolve um float.
// Este apontador especifica qual a operação a executar.
float Calcula_ApontadorFuncao ( float a, float b,
      float (*pt2Func)(float, float) ) {
// Executar o cálculo usando o apontador para função
float resultado = pt2Func(a, b);
return resultado;
}
```

```
int main(void) {
   float a, b, res;
   char  op;
   a  = 2.5;
   b  = 5.3;
   op = '+';
   res = Calcula_ApontadorFuncao(a, b, Somar);
   printf("%f %c %f = %f\n", a, op, b, res);
   return 0;
}
```

Exercício **T04_ex5**: Utilizar a função `qsort()`, contida na biblioteca `stdlib.h`, para ordenar por ordem crescente ou decrescente um array de inteiros.

A função `qsort()` possui a seguinte assinatura:

```
void qsort(void* arrVals, int numVals, int sizeVal, int (*compare)(void *pa, void *pb));
```

`qsort()` permite ordenar uma *array* contendo qualquer tipo de dados.

Questão: Como se utiliza a função `qsort()`?

`qsort()` chama a função `compare()` sempre que é necessário efetuar a comparação de dois valores. A função `compare()` recebe dois parâmetros, `pa` e `pb`, e devolve um inteiro {<0, =0, >0}, dependendo da relação entre os dois parâmetros `pa` e `pb`.

Dado o seguinte *array* de inteiros:

```
int arr[]={10,9,8,1,2,3,5};
```

A função a utilizar no lugar do argumento `compare()` da função `qsort()`, de modo a ordenar o *array* de `int`'s por ordem crescente, é:

```
int crescenteInt(const void *pa, const void *pb) {
   return(*(int *)pa - *(int *)pb);
}
```

A função a utilizar no lugar do argumento `compare()` da função `qsort()`, de modo a ordenar o *array* de `int`'s por ordem decrescente, é:

```
int decrescenteInt(const void *pa, const void *pb) {
   return(*(int *)pb - *(int *)pa);
}
```

Para ordenar os valores de `arr` por ordem crescente podemos utilizar a seguinte instrução:

```
qsort(arr, sizeof(arr)/sizeof(int), sizeof(int), crescenteInt);
```

Para ordenar os valores de `arr` por ordem decrescente podemos utilizar a seguinte instrução:

```
qsort(arr, sizeof(arr)/sizeof(int), sizeof(int), decrescenteInt);
```

Os argumentos da função de comparação a utilizar em `qsort()` são qualificados com `const`. **`const`** é um qualificador que se aplica a uma entidade (variável, argumento, etc.) que não pode ser alterada durante a execução dum programa. Ou seja, a uma entidade qualificada como `const` não se podem fazer atribuições a não ser na inicialização.

2.15.8 *Array* de apontadores para funções

Para mostrar como se utiliza um *array* de apontadores para funções, vamos escrever uma calculadora que efetua uma de entre quatro operações, selecionada de acordo com o valor do argumento `op`. Começamos por mostrar a forma convencional em que se implementa a calculadora recorrendo a um `switch`. O `switch` contém quatro casos, cada um para chamar a função adequada à operação a realizar.

```
enum TIPO_OP{SOMA, SUB, MULT, DIV};
typedef enum TIPO_OP Tipo;
void calcula(float a, float b, Tipo op) {
   switch (op) {
      case SOMA:
         Somar(a,b); break;
```

```
        case SUB:
           Subtrair(a,b); break;
        case MULT:
           Multiplicar(a,b); break;
        case DIV:
           Dividir(a,b); break;
    }
}
```

Exercício **T04_ex6**: Implementar a mesma calculadora, mas utilizando um *array* de apontadores para funções em vez do `switch`.

```
enum TIPO_OP{SOMA=0, SUB=1, MULT=2, DIV=3};
typedef enum TIPO_OP Tipo;
typedef float (*fpCalcula)(float,float);

float calcula(float a,float b,Tipo op,fpCalcula fp[]) {
   return (fp[op])(a,b);
}

float Somar(...){...};      // funções iguais ao exercício T04_ex4
float Subtrair(...){...};
float Multiplicar(...){...};
float Dividir(...){...};

int main() {
    float       a, b, res;
    Tipo        op;
    fpCalcula   funcP[4] =
        { Somar,Subtrair,Multiplicar,Dividir };
    char        simboloOp[4] = { '+','-','x','/' };
    a  = 2.5;
    b  = 5.3;
    op = SOMA; // SOMA, SUB, MULT, DIV
    res = calcula(a, b, op, funcP);
    printf("%f %c %f = %f\n",a,simboloOp[op],b,res);
    return 0;
}
```

2.16 Bibliotecas Externas

Referências bibliográficas:

- www.gnu.org/software/libc/manual/
- crasseux.com/books/ctutorial/Libraries.html
- randu.org/tutorials/c/libraries.php
- www.techytalk.info/c-cplusplus-library-programming-on-linux-part-one-static-libraries/
- www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html
- www.theasciicode.com.ar

O núcleo da linguagem C é pequeno e simples, mas possui muitas funcionalidades fornecidas por **bibliotecas** externas de funções. As bibliotecas **normalizadas** tornam o código C portável. Uma biblioteca é um ficheiro de código compilado que o compilador junta (faz o *link*) com os nossos programas em C. Exemplos de funções fornecidas pelas bibliotecas são as matemáticas, de manipulação de *strings* e de entrada/saída. Também podemos criar as nossas próprias bibliotecas, mas como criar uma biblioteca num sistema GNU?

A maioria dos programas em C inclui pelo menos uma biblioteca. Assim, é necessário: ligar a biblioteca com o programa e incluir no programa os ficheiros *header* da biblioteca.

A biblioteca GNU normalizada para programação em C, `glibc`, é ligada automaticamente com todos os programas, mas os ficheiros *header* devem ser incluídos pelo programador. `glibc` inclui funções para entrada/saída, lidar com datas e horas, fazer cálculos, manusear *strings*, reservar memória, efetuar operações matemáticas, etc.

Exemplo: devemos incluir `stdio.h` num programa que use qualquer funcionalidade normalizada de entrada/saída do C, apesar de `glibc` ser ligada automaticamente. As outras bibliotecas não são ligadas automaticamente.

Exemplo: ligar um programa com a biblioteca `libm.so`

```
gcc -o nome_programa nome_programa.c -lm
```

A opção que indica ao compilador que se quer ligar com `libm.so` é `-lm`. Com a opção `-l` indicamos ao compilador qual a implementação duma função pretendemos usar, de entre as disponíveis em várias bibliotecas. Em conclusão, devemos sempre fazer o seguinte: (i) ligar a biblioteca com a opção `-l` no `gcc` (exeto no caso de `glibc`) e (ii) incluir os ficheiros *header* da biblioteca.

2.17 Ficheiros *Header*

Quando incluímos um ficheiro *header*, o compilador acrescenta os nomes das suas funções, os tipos de dados e outras declarações à lista de palavras e comandos reservados da linguagem. Depois disso, não podemos usar esses nomes para designar outra coisa que não seja aquilo que a biblioteca especifica, em qualquer ficheiro com código fonte que faça o "include" desse ficheiro *header*. O ficheiro *header* mais utilizado é aquele que contém as funções normalizadas de entrada/saída do `glibc`, ou seja, `stdio.h`. Um ficheiro *header* é incluído num ficheiro fonte `name1.c` utilizando a diretiva:

```
#include "name1.h"
```

Esta diretiva inclui o ficheiro `name1.h`, o qual deve estar localizado na pasta do ficheiro fonte `name1.c`. Já a diretiva:

```
#include <name2.h>
```

inclui o ficheiro `name2.h`, localizado numa pasta do sistema, como por exemplo /usr/include.

Exemplo: Utilizar a diretiva `#include` para incluir o ficheiro *header* normalizado `stdio.h` de modo a usar a função `printf`:

```
#include <stdio.h>
int main() {
  printf ("O ficheiro stdio.h foi incluido\n");
  return 0;
}
```

Se este código for guardado no ficheiro `test_io.c`, pode ser compilado com o comando:

```
gcc test_io.c -o test_io
```

A maioria das funções normalizadas do `glibc` pode ser incorporada num programa usando apenas a diretiva `#include` com os ficheiros *header* adequados. Consideremos agora que o código seguinte está guardado no ficheiro `test_math.c`.

```
#include <stdio.h>
#include <math.h>
int main () {
  double x, y;
  y = sin (x);
  printf ("Este exemplo exige incluir math.h\n");
  return 0;
}
```

Em programas que usem funções que não fazem parte de `glibc`, como por exemplo `sin()`, deve usar-se o `gcc` com a opção `-l` para fazer a ligação com as bibliotecas adequadas, `libm.so` neste exemplo. Se compilarmos o ficheiro `test_math.c` com o comando:

```
gcc -o test_math test_math.c -lm
```

A opção `-lm` indica ao `gcc` para efetuar a ligação com a biblioteca de funções matemáticas `libm.so`.

Para sabermos quais os ficheiros *header* a incluir num programa, e quais as bibliotecas com que se deve fazer a ligação, podemos recorrer ao comando "`man nomeFuncao`". Se o nosso programa utiliza a função `sin()` e não soubermos

qual a biblioteca que disponibiliza essa função, nem qual o ficheiro *header* a incluir, podemos recorrer ao comando `man` do Linux. O comando "`man sin`", quando executado na consola, produz o seguinte resultado:

```
NOME
       sin - função seno

SINOPSE
       #include <math.h>

       double sin(double x);

       Ligar com -lm.

DESCRIÇÃO
       A função sin() devolve o seno de x, estando x expresso em radianos.

VALOR DEVOLVIDO
       Em sucesso, esta função devolve o seno de x. (...)
```

2.18 Tipos de Bibliotecas

Existem 2 tipos de biblioteca: **estáticas** e **partilhadas**. As bibliotecas partilhadas também se podem chamar **dinâmicas**. Ao ligar com uma biblioteca estática, todo o código da biblioteca é fundido com o código objeto do nosso programa. Se fizermos a ligação com muitas bibliotecas estáticas, o executável terá um tamanho enorme. As bibliotecas partilhadas foram desenvolvidas para reduzir o tamanho do código dos programas. Ao ligar com uma biblioteca partilhada, o código da biblioteca não é fundido com código objeto do nosso programa. Em vez disso, um "esboço" (*stub*) do código da biblioteca é inserido no nosso código objeto. O código *stub* é muito pequeno e só chama as funções da biblioteca partilhada, o sistema operativo faz o resto. O tamanho dum executável criado com uma biblioteca partilhada é geralmente muito menor do que o criado com uma biblioteca estática.

As bibliotecas partilhadas também podem reduzir a quantidade de memória utilizada. Contudo, as bibliotecas estáticas são úteis quando se deseja distribuir um executável por pessoas cujos computadores não têm as bibliotecas partilhadas necessárias. Depois de ligar com bibliotecas estáticas, essas bibliotecas deixam de ser necessárias para executar o programa. Para não haver problemas ao fazer a ligação entre ficheiros objeto e bibliotecas, nos sistemas UNIX ou Linux o nome dos ficheiros das bibliotecas deve começar com `lib` e terminar com `.a` (estática) ou `.so` (partilhada).

Exemplo: `libm.a` é a versão estática da biblioteca de funções matemáticas do C e `libm.so` é a versão partilhada.

Quando quisermos ligar os nossos programas com uma dada biblioteca, devemos usar a opção `-l` seguida do nome dessa biblioteca, excluindo o prefixo `lib` e o sufixo `.a` ou `.so`.

Exemplo: O comando seguinte cria um programa executável com nome `ex1_math` a partir do ficheiro fonte `ex1_math.c` e da biblioteca `libm.so`.

```
gcc -o ex1_math ex1_math.c -lm
```

Por omissão, o compilador faz a ligação com a versão partilhada das bibliotecas. Se quisermos ligar com a versão estática da biblioteca, devemos executar o `gcc` com a opção `--static`. O comando seguinte liga o ficheiro objeto gerado a partir de `ex1_math.c` com `libm.a` em vez de `libm.so`:

```
gcc -o ex1_math ex1_math.c -lm --static
```

2.19 Comparação entre Bibliotecas Estáticas e Partilhadas

Para começar vamos sintetizar as diferenças de terminologia, relativa a bibliotecas, entre Linux e Windows. A terminologia do Linux é:

- As bibliotecas estáticas são ficheiros do tipo *archive* e possuem extensão ".a";
- As bibliotecas partilhadas são ficheiros do tipo *shared object* e possuem extensão ".so".

Na terminologia do Windows:

- As bibliotecas estáticas são ficheiros do tipo *library* e possuem extensão ".lib";
- As bibliotecas dinâmicas, ou partilhadas, são ficheiros do tipo *dynamic linked library* e possuem extensão ".dll".

Devemos ligar os nossos programas com bibliotecas partilhadas quando estas forem usadas por vários executáveis, nos outros casos devemos usar bibliotecas estáticas. Nas bibliotecas estáticas o seu código é incluído nos executáveis. Os executáveis ficam com um tamanho maior, mas não precisam das bibliotecas durante a execução. Por seu lado, o código das bibliotecas partilhadas não é incluído no executável. Em vez disso é carregado para memória durante a execução.

Deste modo, devem existir no sistema onde corremos o executável. Os executáveis estáticos carregam e executam mais rápido, porque não precisam de aceder aos símbolos (funções, etc.) de forma indireta.

As bibliotecas partilhadas ocupam menos memória, pois cada biblioteca só é carregada em memória apenas uma vez, e todos os programas usam a mesma cópia. Se removermos ou danificarmos uma biblioteca partilhada, qualquer programa que depender dela terá problemas. Em contraste com isto, os programas ligados estaticamente não precisam de bibliotecas para serem executados. Se precisarmos substituir uma biblioteca partilhada, a substituição é feita apenas uma vez e todos os programas afetados serão corrigidos. Em contraste com isto, se uma biblioteca estática for alterada temos que voltar a ligar com ela todos os programas estáticos que a utilizem. Se precisarmos de distribuir um executável, é mais seguro que ele seja estático, ou então exigimos que os utilizadores possuam as bibliotecas partilhadas necessárias.

2.20 Bibliotecas de Funções Normalizadas

Uma biblioteca contém um repositório de funções e macros normalizadas. Há muitos tipos de funções e macros numa biblioteca. Vamos apresentar a seguir alguns dos tipos de funções disponíveis, em conjunto com os ficheiros *header* que temos que incluir para as podermos usar:

- Funções para entrada/saída: `stdio.h`
- Manipulação de *strings*: `string.h` e `stdlib.h`
- Manipulação de carateres: `ctype.h`
- Funções matemáticas: `math.h`

Os ficheiros *header* encontram-se na pasta `/usr/include` e em algumas subpastas. Podemos confirmar este facto executando o comando:

```
ls -al /usr/include | more
```

2.20.1 Funções para entrada/saída

```
int     fclose(FILE *);
int     feof(FILE *);
int     fflush(FILE *);
int     fgetc(FILE *);
char    *fgets(char *, int, FILE *);
FILE    *fopen(const char *, const char *);
int     fprintf(FILE *, const char *, ...);
int     fputc(int, FILE *);
int     fputs(const char *, FILE *);
size_t  fread(void *, size_t, size_t, FILE *);
int     fscanf(FILE *, const char *, ...);
int     fseek(FILE *, long int, int);
int     fsetpos(FILE *, const fpos_t *);
size_t  fwrite(const void *, size_t, size_t, FILE *);
int     getc(FILE *);
int     getchar(void);
char    *gets(char *);
int     printf(const char *, ...);
int     putc(int, FILE *);
int     putchar(int);
int     puts(const char *);
int     putw(int, FILE *);
int     remove(const char *);
int     rename(const char *, const char *);
int     scanf(const char *, ...);
int     sprintf(char *, const char *, ...);
```

```
int     sscanf(const char *, const char *, int ...);
```

etc.

2.20.2 Funções para manipular *strings*

A biblioteca `glibc` disponibiliza várias funções para manusear *strings*. Apresentam-se a seguir as mais utilizadas. Para utilizar as funções cujo nome começa por `ato`, deve incluir-se o ficheiro *header* stdlib.h, enquanto para utilizar as funções cujo nome começa por `str`, deve incluir-se o ficheiro *header* string.h.

`atof` Converte uma string ASCII para um número real. Por exemplo, converte a string "-23.5" para o valor real -23.5. Exemplo:

```
double meu_valor;
char   minha_string[] = "+1776.23";
meu_valor = atof(minha_string);
printf("VALOR REAL = %f\n", meu_valor);
```

O resultado gerado pelo código anterior é: `VALOR REAL = 1776.230000`

`atoi` Converte uma string ASCII para o inteiro equivalente. Por exemplo, converte a string "-23.5" para o inteiro -23:

```
int  meu_valor;
char minha_string[] = "-23.5";
meu_valor = atoi(minha_string);
printf("VALOR INT = %d\n", meu_valor);
```

`atol` Converte uma string ASCII para um inteiro `long` equivalente. Por exemplo, converte a *string* "+2500000000" para o valor `long int` 2500000000:

```
long meu_valor;
char minha_string[] = "+2500000000";
meu_valor = atol(minha_string);
printf("%ld\n", meu_valor);
```

`strcat` Concatena/junta duas *strings* numa única. Exemplo:

```
char string1[50] = "Ola', ";
char string2[] = "mundo!\n";
strcat (string1, string2);
printf ("%s",string1);
```

Este exemplo junta o conteúdo de `string2` ao conteúdo existente em `string1`. O *array* `string1` fica com a string "`Ola', mundo!\n`".

`strcmp` Compara duas *strings* e devolve um valor que indica qual das *strings* surge primeiro por ordem alfabética. Se as duas *strings* forem iguais, `strcmp` devolve 0. Se a primeira *string* passada a `strcmp` surgir primeiro por ordem alfabética, `strcmp` devolve um valor menor que 0. Se a segunda *string* passada a `strcmp` surgir primeiro por ordem alfabética, `strcmp` devolve um valor maior que 0. Nota: os números possuem um código ASCII menor que as letras, e as letras maiúsculas possuem um código menor que as minúsculas.

Exemplo:

```
int  comparacao;
char string1[] = "alfa";
char string2[] = "beta";
comparacao = strcmp (string1, string2);
printf ("%d\n", comparacao); // escreve -1 (< 0)
comparacao = strcmp (string2, string1);
printf ("%d\n", comparacao); // escreve  1 (> 0)
comparacao = strcmp (string1, string1);
printf ("%d\n", comparacao); // escreve 0
```

`strcpy` Copia uma *string* para outra *string*. Exemplos:

```
char destino_string[50];
char origem_string[] = "Hoje e' segunda-feira?";
/* Exemplo 1 */
strcpy (destino_string, origem_string);
printf ("%s\n", destino_string);
/* Exemplo 2 */
strcpy (destino_string, "Estamos em aula agora?");
printf ("%s\n", destino_string);
```

`strlen` Devolve um inteiro correspondente ao tamanho duma *string*, em carateres, excluindo o terminador da *string* (`null='\0'`).

```
int  tamanho_string;
char minha_string[] = "norte";
tamanho_string = strlen (minha_string);
printf ("TAMANHO=%d\n", tamanho_string);
```

Este exemplo escreve `TAMANHO=5` no ecrã.

2.20.3 Funções para manipular carateres

Apresentamos agora algumas funções e macros para manusear carateres individualmente. Para as utilizar, devemos incluir o ficheiro *header* ctype.h. A biblioteca a usar é apenas `glibc`.

- `isalnum(int c)`: Devolve true se o argumento for um carater alfanumérico.
- `isalpha(int c)`: Devolve true se o argumento for uma letra do alfabeto, ou seja, um carater de 'A' a 'Z' ou de 'a' a 'z'.
- `isascii(int c)`: Devolve true se o argumento for um carater ASCII válido, isto é, se possui um valor inteiro no intervalo 0 a 127.
- `iscntrl(int c)`: Devolve true se o argumento for um carater de controlo. Estes carateres possuem normalmente um código no intervalo 0 a 31.
- `isdigit(int c)`: Devolve true se o argumento for um dígito no intervalo '0' a '9'.
- `isgraph(int c)`: Devolve true se o argumento for um carater gráfico: alfanumérico ou pontuação.
- `islower(int c)`: Devolve true se o argumento for uma letra minúscula do alfabeto.
- `isprint(int c)`: Devolve true se o argumento for um carater passível de ser escrito: gráfico ou espaço.
- `ispunct(int c)`: Devolve true se o argumento for um carater de pontuação.
- `isspace(int c)`: Devolve true se o argumento for um carater do tipo "espaço em branco", onde se incluem o espaço (`' '`), tabulação (`'\t'` e `'\v'`), mudança de linha (`'\n'`), `'\f'` e `'\r'`.
- `isupper(int c)`: Devolve true se o argumento for uma letra maiúscula do alfabeto.
- `isxdigit(int c)`: Devolve true se o argumento for um dígito hexadecimal válido, onde se incluem os dígitos decimais, as letras de 'a' a 'f' e as letras de 'A' a 'F'.
- `toascii(int c)`: Devolve o argumento sem o seu oitavo bit, ou seja, devolve um valor inteiro entre 0 e 127, que é assim um carater ASCII válido.
- `tolower(int c)`: Converte uma letra para a letra minúscula correspondente.
- `toupper(int c)`: Converte uma letra para a letra maiúscula correspondente.

2.20.4 Funções matemáticas

Vamos ver algumas funções e macros da biblioteca `libm`. Para as utilizar, poderemos ter que incluir os ficheiros *header* math.h, tgmath.h ou limits.h. As constantes a usar devem estar no formato vírgula flutuante. Deve usar-se, por exemplo, 7.0 em vez de 7.

- `int abs(int x)`: Macro que devolve o valor do argumento sem sinal.
- `double acos(double x)`: Devolve o arco-cosseno do argumento, cujo valor se deve situar no intervalo -1.0 a +1.0, inclusive. O resultado vem em radianos.

- `double asin(double x)`: Devolve o arco-seno do argumento em radianos, cujo valor se deve situar no intervalo -1.0 a +1.0, inclusive.
- `double atan(double x)`: Devolve o arco-tangente do argumento em radianos.
- `double atan2(double y, double x)`: Uma função especial que calcula o arco-tangente do primeiro argumento dividido pelo segundo (y/x). Exemplos:
  ```
  result = atan2(y, x);
  result = atan2(y, 3.14);
  ```
- `double ceil(double x)`: Devolve o menor inteiro não inferior ao argumento.
- `double cos(double x)`: Devolve o cosseno do argumento em radianos.
- `double cosh(double x)`: Devolve o cosseno hiperbólico do argumento.
- `double exp(double x)`: Devolve o valor de *e* elevado ao argumento, ou seja, e^x.
- `double fabs(double x)`: Função que devolve o valor absoluto do argumento.
- `double floor(double x)`: Devolve o maior inteiro não superior ao argumento.
- `double log(double x)`: Devolve o logaritmo natural do argumento x, o qual deve ser positivo.
- `double log10(double x)`: Devolve o logaritmo na base 10 do argumento x, o qual deve ser positivo.
- `double pow(double x, double y)`: Devolve o valor do primeiro argumento elevado ao segundo, ou seja, x^y. Exemplos:
  ```
  result = pow(x,y); /* x elevado a y */
  result = pow(x,2); /* x ao quadrado */
  ```
- `double sin(double x)`: Devolve o seno do argumento, que deve estar em radianos.
- `double sinh(double x)`: Devolve o seno hiperbólico do argumento.
- `double sqrt(double x)`: Devolve a raiz quadrada do argumento x, o qual deve ser positivo.
- `double tan(double x)`: Devolve a tangente do argumento, que deve estar em radianos.
- `double tanh(double x)`: Devolve a tangente hiperbólica do argumento.

2.21 Criar Bibliotecas

Se tivermos um conjunto de ficheiros C que contêm apenas funções, podemos transformá-los em bibliotecas passíveis de ser utilizadas, de forma estática ou dinâmica, por programas. Esta opção permite explorar a **programação modular** e **reutilização de código**: *escrever uma vez, usar muitas*.

Uma biblioteca é basicamente um arquivo de ficheiros objeto. A primeira coisa a fazer para criar uma biblioteca é dispor dos ficheiros C contendo todas as funções a disponibilizar. A biblioteca a criar pode conter vários ficheiros objeto. Depois de ter os ficheiros C, devemos compilá-los para ficheiros objeto.

2.21.1 Criar uma biblioteca estática

Para criar a biblioteca utilizamos o utilitário de arquivo `ar`:

```
ar rc libexemplo.a objfile1.o objfile2.o objfile3.o
```

Esta ação cria uma biblioteca estática com nome libexemplo.a. Num caso concreto, substituímos a parte "exemplo" do nome da biblioteca por algo que se adeque a esse caso. O comando seguinte cria um **índice** dentro da biblioteca:

```
ranlib libexemplo.a
```

O índice é uma lista dos símbolos definidos pelos ficheiros objeto arquivados. O comando seguinte lista o índice da biblioteca:

```
nm -s libexemplo.a
```

2.21.2 Utilizar uma biblioteca estática

Para podermos fazer a ligação de programas C com a biblioteca criada, sem erros, temos que disponibilizar um ficheiro *header* contendo o protótipo/assinatura das funções incluídas na biblioteca. Ao ligar um programa com a biblioteca, temos que indicar onde se encontra essa biblioteca. Um exemplo:

```
gcc -o progTest -L. -lexemplo --static progTest.o
```

A opção `-L.` no comando anterior indica ao `gcc` para procurar a biblioteca libexemplo.a na pasta do código fonte (“.”), para além de outras pastas pré-definidas contendo bibliotecas.

2.21.3 Criar uma biblioteca partilhada

Criar bibliotecas partilhadas, ou dinâmicas, é igualmente simples. Recorrendo ao exemplo anterior, para criar uma biblioteca partilhada começa-se por gerar os ficheiros objeto:

```
gcc -fPIC -c objfile1.c
gcc -fPIC -c objfile2.c
gcc -fPIC -c objfile3.c
gcc -shared -o libexemplo.so objfile1.o objfile2.o objfile3.o
```

A opção `-fPIC` indica ao compilador para gerar *Position Independent Code*. Deste modo, a biblioteca será criada utilizando endereços relativos em vez de absolutos, uma vez que a biblioteca poderá ser carregada múltiplas vezes em memória. A opção `-shared` indica ao compilador para criar uma biblioteca partilhada.

2.21.4 Utilizar uma biblioteca partilhada

O próximo passo será compilar o nosso programa e ligá-lo com a biblioteca partilhada libexemplo.so:

```
gcc -o progTest.o -c progTest.c
gcc -o progTest -L. -lexemplo progTest.o
```

Note-se que o comando que faz a ligação é semelhante ao que se usou na ligação com a biblioteca estática. Embora o processo de ligação seja semelhante, neste caso nenhum código da biblioteca é inserido no executável. Dai a razão do nome "biblioteca dinâmica/partilhada".

Dado que um programa que usa bibliotecas estáticas já tem o código das bibliotecas inserido nele, pode ser executado de forma autónoma. Um programa ligado a bibliotecas partilhadas precisa aceder às bibliotecas durante a execução, logo precisa saber onde se localizam essas bibliotecas partilhadas. A chave para que um programa funcione com bibliotecas partilhadas é usar a variável de ambiente LD_LIBRARY_PATH. Para ver o conteúdo desta variável, caso esteja definida, utiliza-se o comando:

```
echo $LD_LIBRARY_PATH
```

Para a definir a variável, de forma automática, podemos recorrer a um ficheiro de comandos (*script*). Ao definir, ou alterar, a variável `LD_LIBRARY_PATH` convém acrescentar o novo caminho sem apagar o seu conteúdo. Por exemplo, na consola `bash` usamos o comando seguinte quando a variável LD_LIBRARY_PATH já estiver definida:

```
export LD_LIBRARY_PATH = /caminho/para/minhaLib:$LD_LIBRARY_PATH
```

Se `LD_LIBRARY_PATH` ainda não estiver definida, usamos o comando:

```
export LD_LIBRARY_PATH=/caminho/para/minhaLib
```

Também podemos instalar a nossa biblioteca na pasta `/usr/local/lib` e acrescentar este caminho permanentemente a `LD_LIBRARY_PATH` incluindo o comando `export` no ficheiro `.bashrc`. Este ficheiro é executado automaticamente ao abrir uma consola `bash`.

3. Aspetos de Mais Baixo Nível do C

3.1 Acesso a Ficheiros de Texto e Binários

A informação é representada em computador através de *binary digits* (bits). Tendo por base o bit podemos representar informação de mais alto nível. Podemos por exemplo representar:

- **textos** (caracteres alfanuméricos) usando uma notação como Baudot, Braille, ASCII, Unicode, etc;
- **números inteiros** sem sinal, com sinal e módulo, em complemento para 1, complemento para 2, ou em excesso;
- **números reais** em vírgula flutuante, usando a norma IEEE 754;
- conteúdos **multimédia** como imagens fixas em formato BMP, JPG, GIF, PNG, etc, ou audiovisuais em formato MP3, MP4, AVI, MOV, WAV, FLV, etc.
- código para execução no computador escrito numa HLL (texto), em *assembly* (texto), ou objeto (binário).

A figura 4 mostra a tabela ASCII de 7 bits, que pode ser utilizada para representar texto.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Figura 4 - Tabela ASCII de 7 bits.

Questão: Colocar em ASCII o texto "`Ola Mundo.`"

Resposta:

O	l	a		M	u	n	d	o	.
4F	6C	61	20	4D	75	6E	64	6F	2E

Outra notação utilizada para representar texto é a codificação universal de texto Unicode (UTF-8 – Unicode Transformation Format-8). A figura 5 mostra a tabela com a notação UTF-8.

binary	hex	decimal	notes
00000000-01111111	00-7F	0-127	US-ASCII (single byte)
10000000-10111111	80-BF	128-191	Second, third, or fourth byte of a multi-byte sequence
11000000-11000001	C0-C1	192-193	Overlong encoding: start of a 2-byte sequence, but code point ≤ 127
11000010-11011111	C2-DF	194-223	Start of 2-byte sequence
11100000-11101111	E0-EF	224-239	Start of 3-byte sequence
11110000-11110100	F0-F4	240-244	Start of 4-byte sequence
11110101-11110111	F5-F7	245-247	Restricted by RFC 3629: start of 4-byte sequence for codepoint above 10FFFF
11111000-11111011	F8-FB	248-251	Restricted by RFC 3629: start of 5-byte sequence
11111100-11111101	FC-FD	252-253	Restricted by RFC 3629: start of 6-byte sequence
11111110-11111111	FE-FF	254-255	Invalid: not defined by original UTF-8 specification

Figura 5 - Codificação universal de texto UTF-8.

A tabela 1 mostra a representação de inteiros positivos e 5 representações de inteiros positivos e negativos (com 3 bits): sinal e módulo, complemento para 1, complemento para 2, excesso de 4 ($2^{n-1}=2^{3-1}$) e excesso de 3 ($2^{n-1}-1=2^{3-1}-1$).

$base_2$	$base_{10}$	S + M	Compl p/ 1	Compl p/ 2	Exc 2^{n-1}	Exc $2^{n-1}-1$
0 0 0	0	+0	+0	0	-4	-3
0 0 1	1	+1	+1	+1	-3	-2
0 1 0	2	+2	+2	+2	-2	-1
0 1 1	3	+3	+3	+3	-1	0
1 0 0	4	-0	-3	-4	0	+1
1 0 1	5	-1	-2	-3	1	+2
1 1 0	6	-2	-1	-2	2	+3
1 1 1	7	-3	-0	-1	3	+4

Tabela 1 - Várias representações de inteiros.

Para representar números em vírgula flutuante é frequente usar-se a norma IEEE 754, precisão simples (32 bits) ou dupla (64 bits). Esta norma inclui 3 campos: sinal (`S`), expoente (`Exp`) e mantissa (`F`). Com estes 3 campos o valor `V` de um número representado em vírgula flutuante por `|S|Exp|F|` é dado pela seguinte expressão:

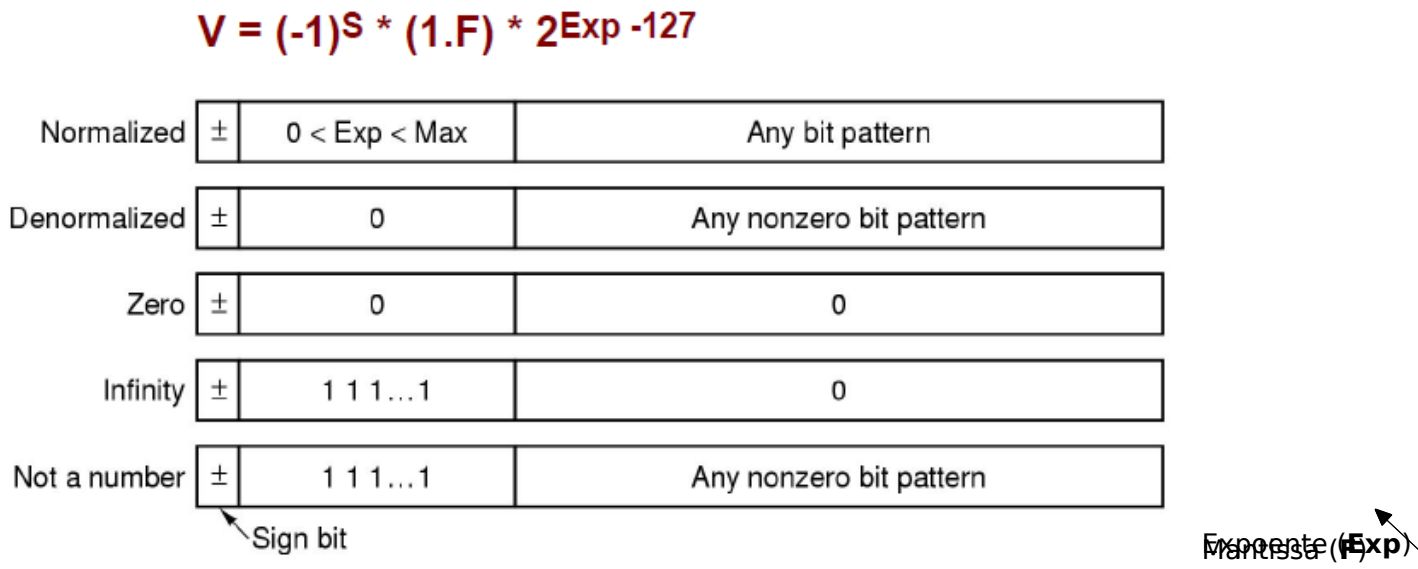


Figura 6 - Norma IEEE 754 para representar números em vírgula flutuante.

Como se pode observar pela figura 6, além dos números normalizados (números em que o primeiro bit da parte fracionária é "1"), a norma IEEE 754 permite representar números não-normalizados, zero, infinito e indicação de não-número.

Um **ficheiro de texto** contém linhas/registos de texto. Cada linha termina com um indicador de **fim de linha**, o qual é inserido automaticamente sempre que se indica um fim da linha.

Questão: Qual é o indicador de fim de linha?

Resposta:

> Nos sistemas operativos (SO) da Apple usa-se como indicador de fim de linha o carater '`\r`' (0x0D);
> Nos SO baseados em Unix usa-se o carater '`\n`' (0x0A);
> Nos Windows usa-se a combinação "`\r\n`" (0x0D 0x0A).

Quando lemos um ficheiro texto, o carater indicador de fim de linha (adequado ao SO) é convertido para o indicador de fim de *string*. Quando se escreve num ficheiro texto os carateres indicadores fim de linha adequados ao SO são escritos quando isso for ordenado. Isto torna a leitura e escrita de ficheiros texto muito mais fácil porque os indicadores de fim de linha são tratados pelo programador.

O conteúdo dum **ficheiro binário** está em bruto, ou seja, nada é convertido. Quando lemos um ficheiro binário, os indicadores de fim de linha são tratados como qualquer outro carater. Quando escrevemos num ficheiro binário os únicos indicadores de fim-de-linha escritos são aqueles que os dados a escrever contiverem. Um ficheiro binário pode conter texto, mas esse texto não é para ser dividido em linhas pelas ocorrências do indicador de fim da linha. Cabe ao programa que lê um ficheiro binário interpretar os dados nele contido. Por exemplo, pode ser uma tabela de valores, uma imagem, ou um vídeo.

Questão: Como ler/escrever ficheiros no modo binário em C?

Resposta:

Para **abrir** um ficheiro no modo binário usamos a função `fopen()` e um modo de acesso que inclua o carater "`b`":

```
fp=fopen(nomeFich, "rb");
fp=fopen(nomeFich, "wb");
```

Para **ler** ou **escrever** no modo binário recorremos às funções `fread()` ou `fwrite()`.

```
size_t fread(const void *Elemento, size_t sizeElemento, size_t numElementos,
             FILE *fp);
size_t fwrite(const void *Elemento, size_t sizeElemento, size_t numElementos,
             FILE *fp);
```

Explicação dos argumentos de `fread()` e `fwrite()`:

`Elemento` → *array* que guarda os valores a escrever/lidos;

`sizeElemento` → tamanho de cada valor em bytes;

`numElementos` → número de valores a escrever/ler;

`fp` → apontador para o ficheiro.

Exercício **T06_ex1**: Escrever em ficheiro um conjunto de valores que definem uma tabela. Os valores estão num *array* de estruturas do tipo `Aluno` que contém 3 campos: `nome` (*string*), `numero` (int) e `nota` (float). Utilize um *array* com 5 posições. Escrever 2 ficheiros: um no modo texto e outro no modo binário. Após executar o programa, visualize o conteúdo dos 2 ficheiros criados e tire conclusões sobre a diferença.

Conteúdo do ficheiro **texto**:

```
Anabela Maria 34111 11.5
   Marco Polo 35222 10.7
Paulo Paulino 36555 14.2
Joana Manuela 45333 9.8
 Manuel Lopes 46888 12.9
```

Conteúdo do ficheiro **binário**:

```
Anabela Maria\00\3F\85\00\00\00\008AMarco Polo\00
\96\89\00\0033+APaulo Paulino\00\CB\8E\00\0033cA
Joana Manuela\00\15\B1\00\00\CD\CC\1CAManuel Lopes
\00(\B7\00\00ffNA
```

3.2 Operações ao Nível do Bit e *Bit Fields*

Em C é possível efetuar operações ao nível do bit, nomeadamente operações lógicas e deslocamentos. Apresenta-se a seguir a sintaxe e o significado das diferentes operações ao nível do bit.

- `op1 & op2` → E lógico efetuado entre cada bit dum operando (`op1`) e o bit na mesma posição do outro operando (`op2`). O resultado do E lógico é '1' se ambos os operandos forem '1';
- `op1 | op2` → OU lógico efetuado entre cada bit dum operando e o bit na mesma posição do outro operando. O resultado do OU lógico é '1' desde que um operando seja '1';
- `op1 ^ op2` → OU lógico exclusivo efetuado entre cada bit dum operando e o bit na mesma posição do outro. O resultado do OU lógico exclusivo é '1' quando apenas um dos operandos for '1';
- `op << n` → deslocamento à esquerda do operando `op` em `n` posições; este deslocamento é equivalente a multiplicar `op` por 2^n;
- `op >> n` → deslocamento à direita do operando `op` em `n` posições; este deslocamento é equivalente a dividir `op` por 2^n.

Operador	Operação	Exemplos
&	E	`0x77 & 0x03;  /* resultado é 0x03 */` `0x77 & 0x00;  /* resultado é 0x00 */`
|	OU	`0x700 | 0x33; /* resultado é 0x733 */` `0x070 | 0;    /* resultado é 0x070 */`
^	OU EXCLUSIVO	`0x770 ^ 0x773; /* resultado é 0x003 */` `0x33  ^ 0x33;  /* resultado é 0x00 */`
<<	Deslocamento à esquerda	`0x01<<4;  /* resultado é 0x10 */` `1<<2;  /* resultado é 4 */`
>>	Deslocamento à direita	`0x010>>4;  /* resultado é 0x01 */` `4>>1;  /* resultado é 2 */`

Na tabela anterior são apresentados exemplos para cada uma das operações ao nível do bit.

***Bit fields* (conjuntos de bits)**

Um ***bit field*** é um conjunto de bits adjacentes pertencentes a uma palavra (*word*). Um *bit field* permite reduzir o espaço ocupado por variáveis quando os valores nelas guardados ocupam poucos bits, como é o caso de valores *booleanos*.

Exercício **T06_ex2**: Qual o espaço ocupado pelas estruturas `Flag` e `Flag_bf` descritas a seguir?

```
struct flag {
   unsigned int temCOR;
   unsigned int temSOM;
   unsigned int PAL;
} Flag;
```

```
struct flag_bf {
   unsigned int temCOR :1;
   unsigned int temSOM :1;
   unsigned int PAL    :1;
} Flag_bf;
```

O número que se segue aos ":" especifica a **largura** em bits. Cada variável deve ser declarada como `unsigned int` ou `int`.

Resposta: Assumindo que um `unsigned int` ocupa 4 bytes, `Flag` ocupa 4+4+4=**12** bytes e `Flag_bf` ocupa **4** bytes, dado que só são precisos 3 bits de um `unsigned int`.

Uma **máscara** pode ser vista como um valor binário/hexadecimal com todos os bits a '0' (ou '1'), exceto um sub-conjunto de bits que estão a '1' (ou '0'), que é utilizado em operações lógicas (E, OU, negação) com variáveis para (i) extrair apenas esse sub-conjunto de bits das variáveis, ou (ii) colocar esse sub-conjunto de bits das variáveis a '0' (ou a '1').

O exemplo incluído na próxima tabela ilustra outra vantagem dos *bit fields:* o código que utiliza *bit fields* é bem mais fácil de perceber e de elaborar (sem erros) do que o código que utiliza máscaras.

Código que utiliza máscaras	**Código que utiliza *bit fields***

`int f;`
`int temCOR=0x1; // bit 0='1'`
`int temSOM=0x2; // bit 1='1'`
`int PAL=0x4;  // bit 2='1'`
	`struct flag_bf f;`

`f |= temCOR;`
`f |= temSOM;`
`f |= PAL;`
	`f.temCOR = 1;`
	`f.temSOM = 1;`
	`f.PAL  = 1;`

`f &= ~temCOR;` `f &= ~temSOM;`	`f.temCOR = 0;` `f.temSOM = 0;`
`if (f & temCOR || f & PAL)` `{...}`	`if (f.temCOR || f.PAL)` `{...}`

3.3 Limites Impostos pela Representação dos Tipos de Dados Pré-definidos

Dependendo da precisão e da gama de valores desejadas, pode usar-se os tipos de dados incluídos na tabela seguinte.

	Com sinal	Sem sinal
`short`	`short int x;` `short y;`	`unsigned short x;` `unsigned short int y;`
`int`	`int x;`	`unsigned int x;`
`long`	`long x;`	`unsigned long x;`
`long long`	`long long x;`	`unsigned long long x;`
`float`	`float x;`	Não disponível
`double`	`double x;`	Não disponível
`char`	`char x; signed char x;`	`unsigned char x;`

Sobre a tabela anterior, convém dizer que os `char`'s com e sem sinal diferem apenas quando são usados em expressões aritméticas.

Exercício **T06_ex3**: Escrever código em C para saber qual o tamanho (em bytes) de cada tipo de dados escalar. Considera-se um tipo de dados **escalar** aquele que representa apenas um único valor. Exemplos de tipos de dados escalares em C são o `char`, `short`, `int`, `long`, `long long`, `float` e `double`. Na resolução do exercício utilize o operador `sizeof(tipo)`.

```
size = sizeof(char);           /* 1 */ (SO de 64-bit)
size = sizeof(short int);      /* 2 */ (SO de 64-bit)
size = sizeof(int);            /* 4 */ (SO de 64-bit)
size = sizeof(long int);       /* 8 */ (SO de 64-bit)
size = sizeof(long long int); /* 8 */ (SO de 64-bit)
size = sizeof(float);          /* 4 */ (SO de 64-bit)
size = sizeof(double);         /* 8 */ (SO de 64-bit)
```

A **gama de representação** de inteiros com `N` (8*size) bits é:

- Com sinal (em complemento para 2) → $-2^{N-1} : +2^{N-1} -1$
- Sem sinal → $0 : 2^{N} -1$

A tabela em baixo mostra a gama de representação dos tipos de dados `char`, `short int`, `int` e `long int`, com e sem sinal.

	bytes	bits	Gama (com sinal)	Gama (sem sinal)
char	1	8	-128 : +127	0 : 255
short	2	16	-32768 : +32767	0 : 65535
int	4	32	-2147483648 : +2147483647	0 : 4294967295
long	8	64	$-2^{63} : +2^{63} -1$	$0 : 2^{64} -1$

Um ***overflow*** (transbordo) com inteiros ocorre quando um número inteiro é aumentado acima do seu valor máximo ou diminuído abaixo do seu valor mínimo. A ocorrência de *overflows* com inteiros está fortemente ligada com o tipo de representação de inteiros em utilização: se tem sinal ou não, e se é um `short`, um `int`, um `long` ou um `long long`. Os *overflows* podem estar associados a inteiros com sinal ou sem sinal. Um *overflow* com sinal ocorre quando a representação do módulo do valor transborda para o bit de sinal. Um *overflow* sem sinal ocorre quando a representação subjacente já não consegue representar o valor. É impossível determinar se um *overflow* representa uma situação de erro, sem perceber o contexto em que ocorre.

Exercício **T06_ex4**: Escrever código em C para verificar a ocorrência de situações de *overflow* em inteiros com e sem sinal.

Resolução:

```
#include <limits.h>
int i;
unsigned int j;
i = INT_MAX;          // i = 2.147.483.647
i++;                  //
printf("i=%d\n",i);   // -2147483648 (overflow com sinal)

j = UINT_MAX;         // j = 4.294.967.295
j++;                  //
printf("j=%u\n", j);  // 0 ( overflow sem sinal)

i = INT_MIN;          // i = -2147483648
i--;                  //
printf("i=%d\n",i);   // 2147483647 ( overflow com sinal)

j = 0;                // j = 0
j--;                  //
printf("j=%u\n", j);  // 4294967295 ( overflow sem sinal)
```

Erros no sinal

Erros de sinal ocorrem quando se convertem inteiros com sinal para inteiros sem sinal. Quando um inteiro com sinal é convertido para um inteiro sem sinal do mesmo tamanho, o padrão de bits do inteiro original é preservado. Quando um inteiro com sinal é convertido para um inteiro sem sinal de maior tamanho, primeiro estende-se o sinal do valor original para os bits adicionais e só depois o valor é convertido. Em ambos os casos, o bit mais significativo (msb) perde a sua função de bit de sinal. Se o inteiro com sinal não for negativo, o valor não é alterado, mas quando o valor do inteiro com sinal é negativo, o resultado será erradamente um valor positivo.

Exercício **T06_ex4** (continuação): Escrever código em C para verificar a ocorrência de uma situação de erro no sinal quando se converte entre inteiros com e sem sinal.

Resolução:

```
int i = -3;
unsigned short u;
u = i;
printf(u = %hu\n", u); // u = +65533 (ERRO NO SINAL)
```

Erros de truncatura

Os erros de truncatura ocorrem quando um inteiro é convertido para um tipo de inteiro com tamanho menor e o valor do inteiro original está fora da gama do tipo menor (ERRO #1 no exemplo seguinte). Normalmente, os bits menos significativos do valor original são preservados e os bits mais significativos são perdidos. Quando um valor sem sinal é convertido para um valor com sinal do mesmo tamanho, o padrão de bits é preservado. Como consequência (i) o bit mais significativo passa a ser o bit de sinal e (ii) os inteiros sem sinal acima do maior valor permitido pelo tipo de inteiro com sinal usado como alvo da conversão são convertidos para valores negativos (ERRO #2 no exemplo seguinte).

Exercício **T06_ex4** (continuação): Escrever código em C para verificar a ocorrência de uma situação de erro de truncatura quando se fazem conversões entre inteiros de tamanho diferente.

```
int i = 100000;
unsigned short int us = 32768;
short int s;
s = i;
printf("s = %hd\n", s); // s = -31072 (ERRO #1)
s = us;
printf("s = %hd\n", s); // s = -32768 (ERRO #2)
us = 65535;
s = us;
printf("s = %hd\n", s); // s = -1     (ERRO #2)
```

3.4 Conversões entre Tipos de Dados: Implícitas e Explícitas

Quando uma expressão mistura vários tipos de dados, um ou mais operandos envolvidos nessa expressão serão automaticamente promovidos pelo compilador para o tipo de dados com maior precisão presente na expressão. Nesta conversão os dados são preservados. A regra de **conversão** automática (**implícita**) é:

`short int → int → unsigned int → long int → unsigned long int → float → double → long double`

Exemplo:

```
char  c;
int   i;
float f;
f = i+3.14159; // i é promovido a float <=> f=(float)i+3.14159
```

Outro tipo de conversão implícita ocorre ao passar de `char→int`. Deste modo, podemos comparar e manipular variáveis do tipo `char`. Um exemplo:

```
// a variável c e as constantes são convertidas para int
int isupper = (c>='A' && c<='Z') ? 1 : 0;
if (! isupper)
  // a subtração é possível devido à conversão de c para int
  c = c - 'a' + 'A';
```

O **operador *cast*** força a **conversão explícita** dum operando escalar para outro tipo de dados escalar especificado. O operador *cast* consiste de um "`nome-do-tipo`", entre parênteses, antes duma expressão:

```
(nome-do-tipo) expressao
```

A `expressao` é convertida para o tipo `nome-do-tipo`. Exemplo:

```
(int) x;
```

O `nome-do-tipo` pode ser um apontador para uma estrutura/união, mas não uma estrutura/união, porque estas não são um tipo escalar. Por exemplo:

```
int x;
(struct teste *)x   // é permitido
(struct teste)x     // não é permitido
```

Um **apontador** pode ser convertido para qualquer tipo de dados que tenha o mesmo tamanho, e convertido de volta para apontador. Quando se converte o tipo "`unsigned long`" para um tipo de **inteiro** de menor tamanho, os bits mais significativos do valor original são descartados. Quando se converte dum tipo de **inteiro** com tamanho inferior para outro tipo de `int` de maior tamanho, os bits adicionais mais significativos são preenchidos com o bit de sinal do valor original. Quando um operando dum tipo de **vírgula flutuante** é convertido para um tipo de **inteiro**, a parte fracionária é descartada. Pode acontecer que o resultado não seja representável no tipo inteiro. Quando um tipo de inteiro é convertido para um tipo de vírgula flutuante, e o valor estiver na gama de valores representáveis mas não exatamente, o resultado é o valor imediatamente superior ou imediatamente inferior.

3.5 Memória Física e Virtual

A **memória física** consiste nos recursos físicos onde se pode armazenar dados e aos quais o computador tem acesso. Exemplos destes recursos são os registos do CPU, a memória cache, a memória RAM, os discos rígidos e os dispositivos de armazenamento amovíveis.

A **memória virtual** é uma abstração fornecida pelo SO. Constitui-se como o espaço, com endereços próprios, "visível" por cada programa em execução.

A gestão da memória física é uma das principais funções do SO. A utilização da memória deve ser otimizada. Para isso, o CPU procura garantir que o código do(s) utilizador(es) usa a memória física disponível da melhor forma. Durante a execução dum programa, o SO pode deslocar os dados de um local da memória física para outro. Processadores embebidos, tais como os controladores industriais e domésticos, ou os processadores usados em dispositivos móveis e impressoras, apresentam recursos de armazenamento mais limitados do que os processadores de uso genérico.

Estrutura da memória virtual dum programa

Um programa em execução no Linux apresenta uma **estrutura de memória** semelhante à da figura 7. O segmento (seção) com o código e as constantes começa sempre no endereço 0x08048000. O próximo segmento da memória é

reservado para as variáveis globais e começa num endereço alinhado a 4 KB. Segue-se a ***heap***, a partir do próximo endereço alinhado com 4 KB. A *heap* cresce quando se chama a função `malloc()`. O segmento que começa no endereço 0x40000000 é reservado para **bibliotecas partilhadas**. A **pilha** começa sempre no endereço 0xBFFFFFFF e cresce em direção aos endereços de memória menores. O segmento que começa no endereço 0xC0000000 é reservado para código e dados do SO (**kernel**).

3.5.1 Diferença entre pilha e *heap*

Pilha é a zona de memória onde se guarda o ***activation record*** associado com a chamada de funções. ***Heap*** é a zona de memória onde se situa o espaço alocado dinamicamente às variáveis, por exemplo usando a função `malloc`.

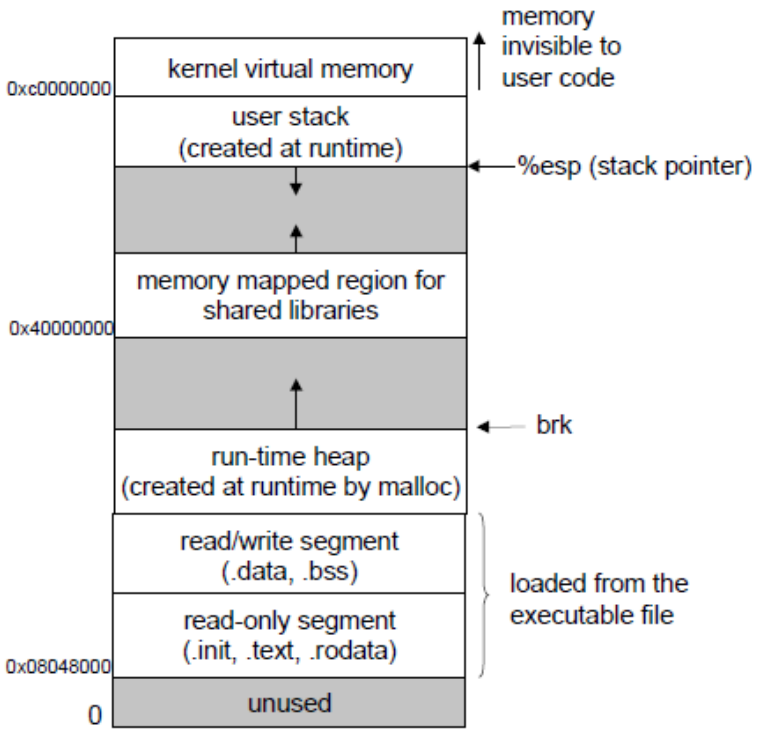


Figura 7 - Memória virtual dum programa em execução no Linux.

3.5.2 Hierarquia de memória

Cada nível da hierarquia de memória, ilustrado na figura 8, possui um tamanho, tempo de acesso e custo diferentes. Quanto mais próximo do CPU menor o tamanho, menor o tempo de acesso e maior o custo por byte.

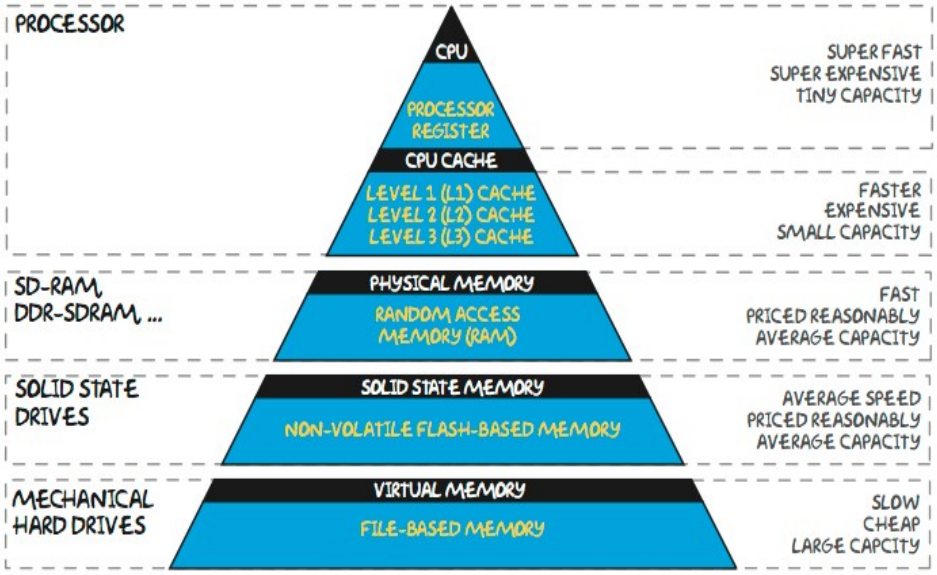


Figura 8 - Hierarquia de memória.

Questão**:** Quanta memória física possui o meu computador?

Em Linux temos várias formas de responder a esta pergunta. Apresentamos a seguir algumas:

- `cat /proc/meminfo` → mostra a informação sobre a memória, guardada no ficheiro `/proc/meminfo`;
- `sudo lshw` → mostra informação sobre o *hardware* do computador: CPUs, hierarquia memória, periféricos, etc;
- `sudo lstopo` → mostra a configuração, ou topologia, do computador: CPUs, caches, periféricos;
- `sudo cfdisk` → mostra informação sobre o disco rígido.

Resposta para uma máquina com um Core i3 330M:

```
Cache L1:      32 KB (instruções) + 32 KB (dados) [por core]
Cache L2:      256 KB  [por core]
Cache L3:      3 MB
RAM:           4 GB
Disco rígido: 640 GB
```

Questão: Que espaço de *swap* possui o meu computador?

Resposta utilizando o comando `top -n1 | grep used`:

A linha que começa com "`KiB Mem`" informa sobre a memória física: total, usada, livre, usada em *buffers* (um tipo de *cache*). A linha que começa com "`KiB Swap`" informa sobre o espaço de *swap*: total, usado, livre, usado como *cache*.

```
KiB Mem:  3909096 total, 2293456 used, 1615640 free, 119576 buffers
KiB Swap: 5996540 total,       0 used, 5996540 free, 871948 cached
```

Nota: 1 KiB = 1024 bytes

Resposta usando o comando `free`:

```
             total       used       free     shared    buffers     cached
Mem:       3909096    2287984    1621112          0     119652     871948
-/+ buffers/cache:    1296384    2612712
Swap:      5996540          0    5996540
```

Resposta usando o comando `vmstat -s`:

```
 3909096 K total memory
 2288364 K used memory
 1345276 K active memory
  661912 K inactive memory
 1620732 K free memory
  119668 K buffer memory
  871948 K swap cache
 5996540 K total swap
       0 K used swap
 5996540 K free swap
     ...   ...
```

Resposta obtida com os 3 comandos:

Memória física = 3.909.096 KB = 3.90 GB

Espaço de *swap* = 5.996.540 KB = 5.99 GB

3.5.3 Utilização da pilha em funções recursivas

O ***activation record*** é uma zona em memória, mais concretamente na pilha, associada com cada uma das chamadas de uma função. A estrutura é criada e mantida ao chamar a função e deve ser descartada quando a função termina a execução. Os elementos incluídos no *activation record*, do endereço maior para o menor, são: os argumentos da função, o endereço da instrução que se segue à instrução `CALL` que chamou a função, o valor do registo `EBP` antes da função ser chamada, os registos que a função atual decide salvaguardar na pilha (para preservar o seu valor) e as variáveis locais à função. A figura 9 ilustra um *activation record* genérico.

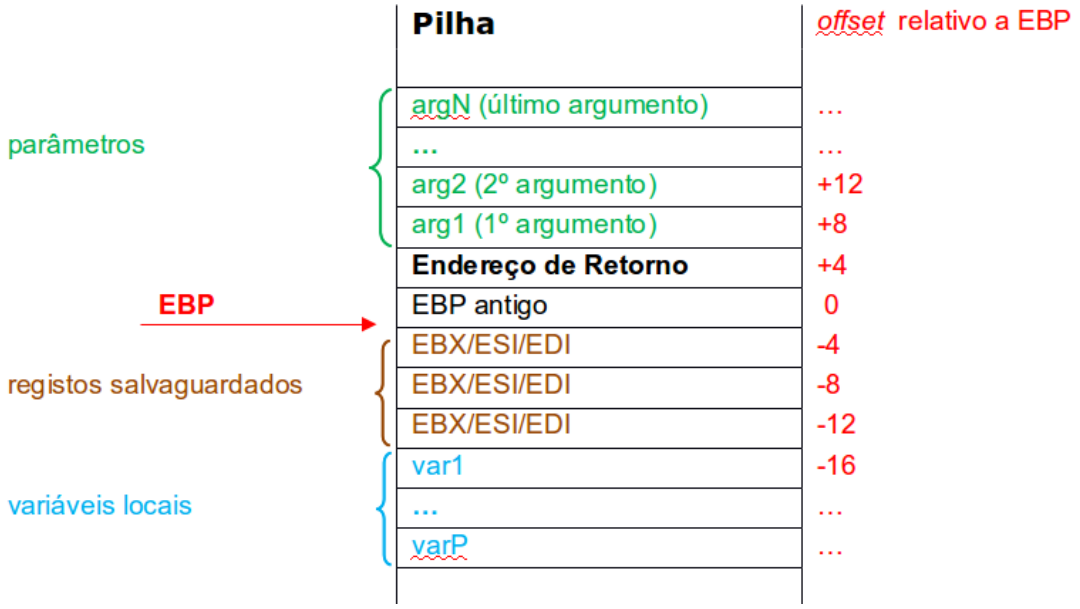


Figura 9 - Estrutura do activation record.

Questão: Porque pode correr mal a execução duma função recursiva?

Resposta: Sempre que a função recursiva se chama a si própria é criado um *activation record* na pilha. O *activation record* duma dada instância da função só é libertado quando essa instância da função terminar. Ou seja, nas sucessivas vezes que a função vai sendo chamada, a pilha vai acumulando *activation records*. Se a função for chamada um número de vezes muito elevado pode acontecer que a pilha encha e a execução do programa termine de forma anormal.

3.6 Armazenamento *Big Endian* vs. *Little Endian*

O tamanho dos diferentes tipos de dados depende da máquina e do compilador em utilização. No entanto, é sempre garantido o seguinte:

```
sizeof(char) < sizeof(short) <= sizeof(int) <= sizeof(long)    e
sizeof(char) < sizeof(short) <= sizeof(float) <= sizeof(double)
```

Nos tipos de dados numéricos que ocupam múltiplos bytes, a **ordenação dos bytes** é importante. Dependendo da arquitetura do computador, existem duas alternativas: *big endian* e *little endian*. Na ordenação ***little endian*** os bits **menos** significativos (LSBs) ocupam os endereços menores. Esta representação é usada em todos os processadores **x86**. Na ordenação ***big endian*** os bits **mais** significativos (MSBs) ocupam os endereços menores. Esta representação é usada no processador **PowerPC**. As redes costumam usar a ordenação *big-endian* e por isso esta ordenação também é designada de ***network order***.

Questão: Como se armazena em memória o valor hexadecimal `0A0B0C0D` usando as ordenações *little-endian* e *big-endian*? A resposta é apresentada na figura 10.

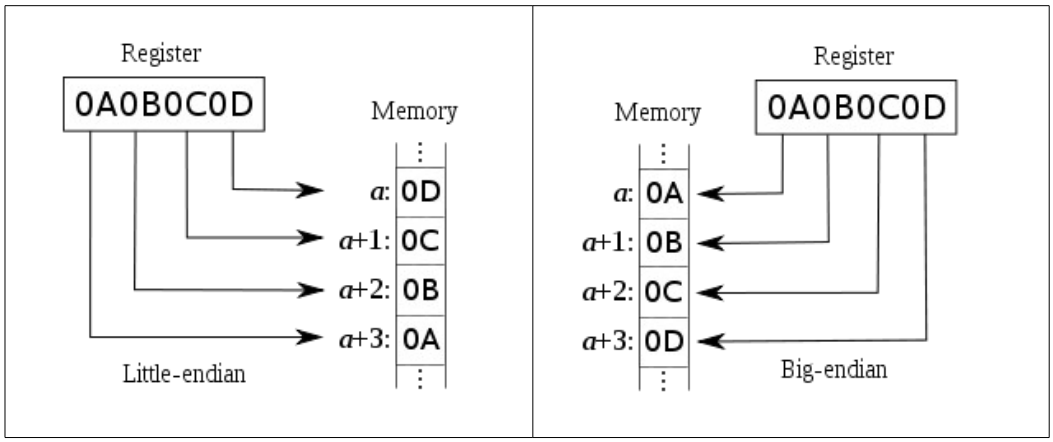


Figura 10 - Ordenação dos bytes em memória.

4. Estruturas de Dados Avançadas e Respetiva Implementação em C

Bibliografia: *Introduction to Algorithms*, capítulo 10.

Um **conjunto** é uma coleção de objetos ou elementos. A maioria dos conjuntos manipulados por algoritmos são **dinâmicos** porque podem crescer ou diminuir, ou seja, podem variar ao longo do tempo. Exemplos de estruturas de dados que implementam conjuntos dinâmicos são as **listas ligadas,** as **filas**, as **pilhas**, as **árvores** e as **tabelas de *hash***. Para cada tipo de conjunto e de algoritmo, pode ser necessário dispor de diferentes **operações** a efetuar sobre esse conjunto: inserir elemento(s), remover elemento(s), procurar elemento(s), etc.

4.1 Listas Ligadas

Uma lista ligada é uma estrutura de dados na qual os elementos são organizados linearmente. Num *array*, a ordem linear é determinada pelo índice dos elementos, enquanto na lista ligada a ordem é determinada por um apontador contido em cada elemento.

Definição: Uma **lista ligada** é uma estrutura de dados dinâmica, composta por uma **sequência** de elementos ou nodos, onde cada elemento contém uma ligação para o elemento seguinte na sequência.

As listas ligadas podem ser **ligadas** num único sentido, **duplamente ligadas, circulares**, **ordenadas** ou **não ordenadas**.

Listas Duplamente Ligadas

Cada elemento de uma lista duplamente ligada *L* contém **dados**, que podem funcionar como **chave**, e dois apontadores: *prox* e *prev* (figura 11). Dado um elemento *x* na lista, *x.prox* aponta para o seu **sucessor** na lista e *x.prev* aponta para o seu **antecessor**. Se *x.prev=NULL*, o elemento *x* não tem antecessor e é a **cabeça da lista.** Se *x.prox=NULL*, o elemento *x* não tem sucessor e é a **cauda da lista**. A variável *cabeça[L]* aponta para o primeiro elemento da lista. Se *cabeça[L]=NULL*, a lista está vazia.

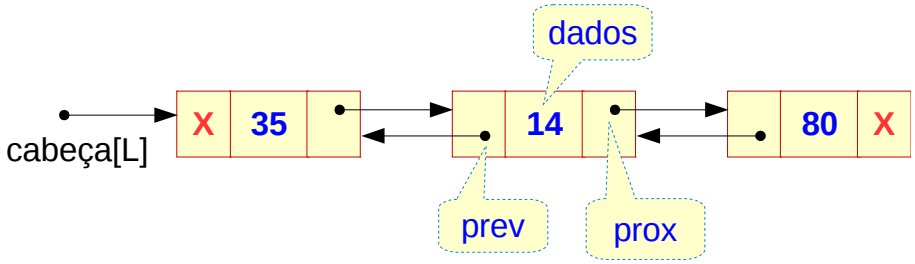


Figura 11 - Lista duplamente ligada.

Listas Ligadas

Tendo por base o exemplo da lista **duplamente ligada**, se a lista é **ligada** num único sentido, os seus elementos não possuem o apontador *prev*. Se uma lista é **ordenada**, os elementos são inseridos na lista de maneira a que os dados guardados nesses elementos formem uma sequência de "valores" ordenada: o elemento mínimo é a cabeça da lista e o elemento máximo é a cauda. Se a lista **não é ordenada**, os elementos podem aparecer por qualquer ordem.

Numa **lista circular**, o apontador *prev* da cabeça da lista aponta para a cauda, e o apontador *prox* da cauda da lista aponta para a cabeça (ver figura 12).

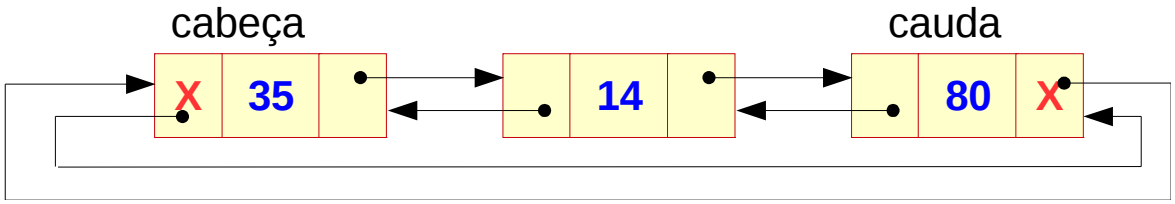


Figura 12 - Lista circular.

Vamos agora focar-nos nas listas ligadas num único sentido. Numa lista ligada cada elemento tem dados úteis e uma ligação ao elemento seguinte (figura 13). A cabeça da lista L é guardada numa variável separada. O fim da lista é indicado por NULL, representado por um 'X' nas figuras desta seção.

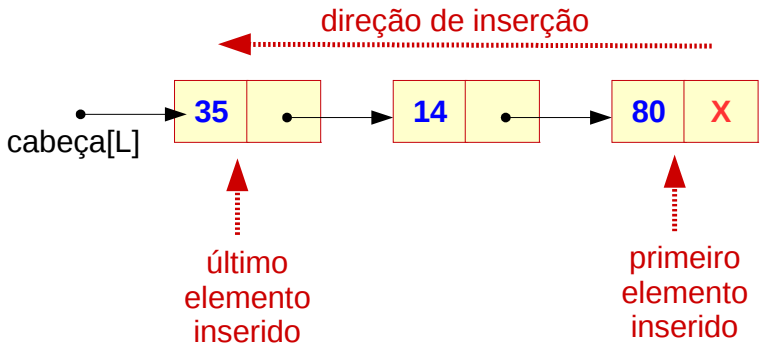


Figura 13 - Lista ligada.

A implementação em C do nodo e da cabeça duma lista ligada é:

```
struct nodo {
   int          dados;    // dados úteis
   struct nodo* prox;
};
struct nodo* head;         // cabeça da lista
```

Comecemos por fazer uma comparação entre as listas ligadas e os *arrays*, em termos do tamanho ocupado em memória e da complexidade (ou tempo de execução) das operações de acesso, inserção e eliminação de elementos.

	Lista ligada	***Array***	***Array* dinâmico**
Tamanho da estrutura	dinâmico	fixo	alterável [Δ]
Aceder a um elemento	O(n)	O(1)	O(1)
Inserir um elemento no meio	O(1)	ND	O(n) [Δ]
Remover um elemento do meio	**O(1) , O(n)** [*]	ND	O(n) [Δ]

[*] O(1) - Lista duplamente ligada e O(n) – lista ligada.

[Δ] Exige libertar e realocar memória.

O código da função que **cria um novo elemento** para inserir na lista ligada é**:**

```
struct nodo *criarElemento (int infoUtil) {
     struct nodo *p = (struct nodo *) malloc(sizeof(struct nodo));
     if (p != NULL) {
          p->dados = infoUtil;
          p->prox  = NULL;
     }
     return p;
}
```

A função devolve um apontador para o elemento criado ou `NULL`. O elemento é criado, mas não é inserido na lista.

Código da função que **insere um elemento** na cabeça da lista *L*:

```
// L=cabeça da lista, k=dados do elemento a inserir
struct nodo *adicionar(struct nodo *L, int k) {
   struct nodo *p = criarElemento(k); // Primeiro cria-se o novo elemento
   if(p == NULL)
      return L;
   p->prox = L;
   return p;
}
```

A função devolve um apontador para a cabeça da lista: a nova (em sucesso) ou a antiga (em insucesso).

Para **percorrer** uma lista ligada é comum usar-se um ciclo do tipo:

```
for (p=inicioLista; p!=NULL; p=p->prox) {
    ... /* fazer algo com o elemento apontado por p */
}
```

ou então um ciclo do tipo:

```
for(p=inicioLista; p->prox!=NULL; p=p->prox) {
    ... /* fazer algo com o elemento apontado por p */
}
```

Procurar um elemento *k* na lista *L*

A função `procurar()` procura o primeiro elemento da lista *L* que tem os dados iguais a *k* e devolve um apontador para esse elemento. Se nenhum elemento contiver os dados iguais a *k*, devolve NULL. *L* deve apontar para a cabeça da lista.

```
struct nodo *procurar(struct nodo *L, int k) {
    struct nodo *p = L;
    while (p!=NULL && p->dados!=k)
        p = p->prox;
    return p;
}
```

Remover um elemento *r* na lista *L*

Como a lista é ligada num único sentido, para encontrar um elemento a remover é preciso um ciclo que percorre a lista. Daqui resulta que a complexidade da operação remover é *O(n)*. A função `remover()` que vamos apresentar remove o elemento *r* da lista *L*. O argumento *L* deve apontar para a cabeça da lista.

A figura 14 mostra como se processa a eliminação do elemento `r` de uma lista ligada. Partindo da situação inicial representada em (a), a função percorre a lista desde a sua cabeça *L* até encontrar o elemento *r* a remover. O passo seguinte é ilustrado em (b), e consiste em colocar o elemento anterior a *r* a apontar para o elemento que se segue a *r* na lista. Este passo é realizado pela instrução:

```
prev.prox = r.prox;
```

No terceiro passo eliminamos o elemento *r* da memória e obtemos a configuração da lista representada em (c). Para isso basta chamar a função `free()`:

```
free(r);
```

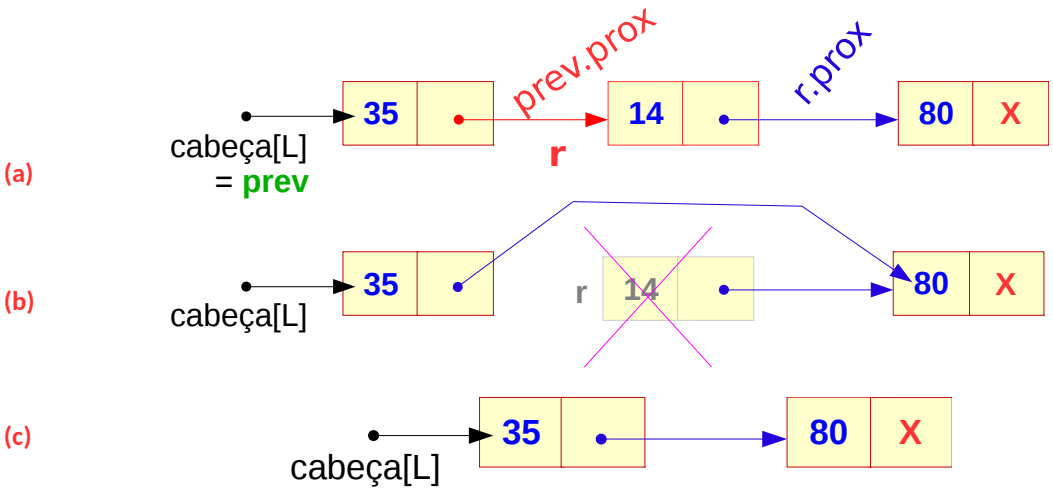


Figura 14 - Remover um elemento duma lista ligada.

Apresenta-se a seguir a implementação da função que remove o elemento *r* na lista *L*:

```
struct nodo *remover(struct nodo *L,struct nodo *r){
    struct nodo *p, *prev=NULL;
    for (p=L; p!=NULL && p!=r; p=p->prox) prev=p;
    if (p != NULL){         // r existe na lista
        if (prev != NULL) // r não é cabeça da lista
            prev->prox =  p->prox;
        else
            L = p->prox;  // nova cabeça da lista
        free(p);          // liberta a memória do elemento r
    }
    return L;             // devolve a cabeça da lista atualizada
}
```

4.2 Listas Duplamente Ligadas

A implementação em C do nodo e da cabeça duma lista duplamente ligada é:

```
struct nodoDL {
    int            dados; // dados úteis
    struct nodoDL  *prev;
    struct nodoDL  *prox;
};
struct nodoDL *head;     // cabeça da lista
```

Remover um elemento *r* da lista duplamente ligada *L*

A função `removerDL()` remove o elemento *r* da lista *L*, em que *L* deve apontar para a cabeça da lista. Como a lista é duplamente ligada, conhecendo o apontador para o elemento *r* a remover passamos a conhecer o elemento anterior (`r->prev`) e o seguinte (`r->prox`). Assim, não é necessário um ciclo para percorrer a lista à procura do apontador para o elemento anterior, como acontece na lista ligada num único sentido, e a complexidade da operação remover é *O(1)*.

```
struct nodo *removerDL(struct nodoDL *L, struct nodoDL *r) {
   if (r->prev != NULL) // r não é a cabeça da lista
      (r->prev).prox = r->prox;
   else                 // r é a cabeça da lista
      L = r->prox;
   if (r->prox != NULL) // r não é a cauda da lista
      (r->prox).prev = r->prev;
   free(r);
}
```

4.3 Pilha

Pilhas e filas podem ser implementadas com listas ligadas ou com *arrays*. Uma **pilha** é um tipo especial de lista em que o último elemento inserido (*push*) é o primeiro a ser retirado (*pop*) . Ou seja, a pilha é uma estrutura do tipo LIFO. A operação de remoção lê (POP) do fim da pilha e a operação de inserção também escreve (PUSH) no fim da pilha.

Como já vimos atrás, a estrutura de memória associada com a execução dum programa inclui uma pilha para guardar os dados associados com a chamada de funções (*activation record*). Estes dados incluem os argumentos da função, o endereço de retorno, registos e as variáveis locais da função.

A figura 15 mostra como se comporta a pilha quando se insere (a) e quando se remove um elemento (b).

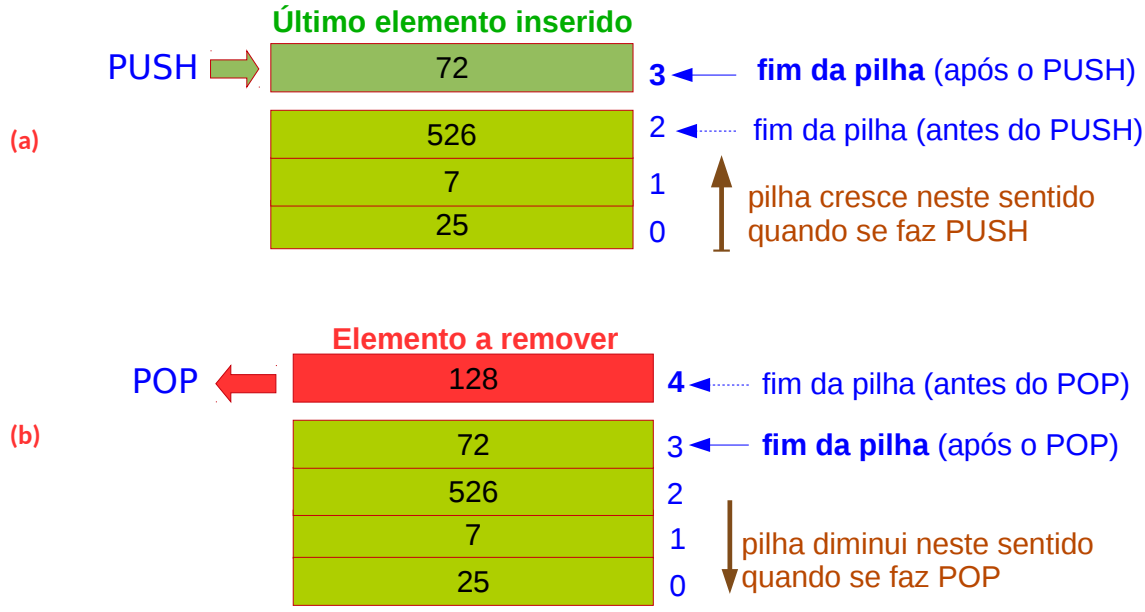


Figura 15 - Funcionamento duma pilha ao inserir (a) e ao remover um elemento (b).

4.3.1 Pilha implementada com um *array*

Pode usar-se um *array*, alocado de forma estática ou dinâmica, para armazenar os dados duma pilha. Uma pilha de inteiros é criada pela declaração:

```
int pilha[100];
```

Os elementos são inseridos e removidos no/do fim da zona ocupada do *array*. Logo, é preciso saber onde termina a ocupação atual do *array*. Para isso, basta apenas guardar a última posição do *array* ocupada, ou então a primeira livre. Exemplo:

```
/* topo/fim da pilha inicializado a zero <=> pilha vazia */
int fimPilha = 0;
```

Apresentamos agora a implementação da função `push()` que insere um elemento na pilha:

```
void push(int elemento) {
   pilha[fimPilha++] = elemento;
}
```

O código da função `pop()`, que retira um elemento da pilha, é:

```
int pop(void) {
   if (fimPilha > 0)
      return pilha[--fimPilha];
   else
      return 0 ; /* zero ou outro valor especial */
}
```

Há casos em que interessa implementar uma variante da função `pop()` que lê o elemento do fim da pilha sem o apagar.

4.3.2 Pilha implementada com uma lista ligada

Também podemos usar uma lista ligada, alocada de forma dinâmica, para armazenar os dados duma pilha. Considerando uma pilha de inteiros, a implementação em C do nodo (`pNodo`) e do topo da pilha (`pilha`) fica:

```
struct    pNodo {
   int          elemento;
   struct pNodo *pprox;
};
struct pNodo *pilha = NULL; // Indicar que a pilha está vazia
```

O topo da pilha é assim a cabeça da lista ligada, não sendo por isso necessária uma variável extra para guardar esse topo. O código seguinte implementa a função `push()` que insere um elemento na pilha:

```
void push (int elemento) {
   /* Alocar memória para um novo nodo */
   struct pNodo *novo_nodo = (struct pNodo *) malloc(sizeof(struct pNodo));
   novo_nodo->pprox    = pilha;    // elemento seguinte=antigo topo da pilha
   novo_nodo->elemento = elemento;
   pilha               = novo_nodo; // topo da pilha = novo nodo inserido
}
```

O campo `pprox` do nodo aponta para o elemento seguinte na pilha, que neste caso fica no sentido do início da pilha.

A implementação da função `pop()`, a qual retira um elemento da pilha, é:

```
int pop (void) {
   if (pilha != NULL) {            // Pilha não vazia
      struct pNodo *pelem = pilha; // Variável auxiliar
      int elem = pilha->elemento;
      pilha    = pelem->pprox;
      free(pelem); // Libertar a memória do nodo retirado
      return elem; // Devolve os dados úteis do elemento retirado da pilha
   }
   else         // Pilha vazia
      return 0; // zero ou outro valor especial
}
```

Pode interessar implementar uma variante da função `pop()` que lê o elemento do topo da pilha sem o apagar.

4.4 Fila

A **fila** tem um modo de funcionamento oposto ao da pilha: o primeiro elemento inserido será o primeiro a ser retirado. Ou seja, a fila é uma estrutura do tipo FIFO. Na fila lê-se e escreve-se em extremos opostos: do início e no fim, respetivamente. As filas são uma estrutura importante por exemplo em (i) interfaces gráficas de programas, para guardar os eventos/mensagens gerados pelo utilizador e (ii) em redes de computadores, para guardar os pacotes de dados recebidos ou a transmitir. Tal como na pilha, a posição de armazenamento dos elementos tem que seguir a ordem de inserção.

4.4.1 Fila implementada com um *array*

Tal como na pilha, os dados duma fila podem ser armazenados num *array*, alocado de forma estática ou dinâmica. Exemplo duma fila de `float`'s:

```
float fila[100];
```

Os elementos são inseridos no fim da fila e retirados do início. Nas filas é preciso guardar o seu **início** e o **fim** em variáveis à parte:

```
int inicio=0, fim=0;
```

Em alternativa, podemos guardar o início e o número de elementos da fila:

```
int inicio=0, ocupacao=0;
```

Aqui vamos adotar a segunda alternativa.

A implementação da função `inserirFila()`, que insere um elemento na fila, é:

```
void inserirFila(float elem) {
   if (ocupacao < 100) {
      fila[inicio+ocupacao] = elem;
      ocupacao++;
   }
}
```

Para retirar um elemento da fila usamos a função `removerFila()`:

```
float removerFila(void) {
   if (ocupacao > 0) {
      ocupacao--;
      return fila[inicio++];
   } else
      return 0; /* zero ou outro valor especial */
}
```

Vamos ver como este código é uma fraca implementação duma fila, recorrendo para isso um exemplo duma fila com capacidade 4. Consideremos que a fila tem 3 das 4 posições ocupadas, como se mostra na figura 16:

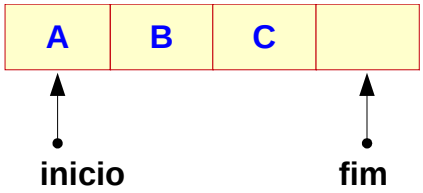


Figura 16 - Fila de tamanho 4 com 3 posições ocupadas.

Se inserirmos um 'D' no fim da fila ela fica cheia (figura 17).

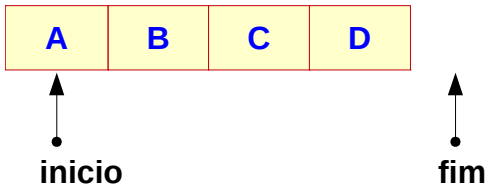


Figura 17 - Fila de tamanho 4 com as 4 posições ocupadas.

Vamos agora retirar o elemento inicial da fila, que neste caso é um 'A'. A posição `inicio` avança para a posição seguinte (figura 18).

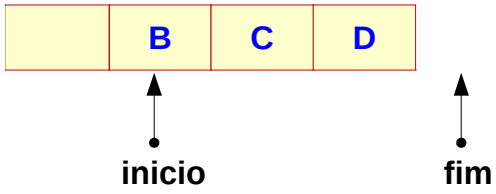


Figura 18 - Fila de tamanho 4 após remover o elemento inicial.

Questão: Se quisermos inserir um 'E' no fim da fila, onde o guardamos?
Solução: Com a fila implementada tal como vimos nesta seção, não conseguimos inserir o elemento 'E' no fim da fila. Para contornar a limitação de termos uma posição vazia e não a podermos ocupar, alteramos a fila para possuir uma estrutura circular. Neste caso o 'E' seria guardado no início do *array*. Para implementar a fila circular precisamos modificar as funções `inserirFila()` e `removerFila()` com as alterações assinaladas a vermelho. A nova versão de `inserirFila()` fica:

```
void inserirFila(float elem) {
   if(ocupacao < 100) {
      fila[(inicio+ocupacao)%100] = elem;
      ocupacao++;
   }
}
```

A nova versão da função `removerFila()` será:

```
float removerFila(void) {
   if(ocupacao > 0) {
      float elem = fila[inicio];
      ocupacao--;
      inicio++;
      if(inicio == 100)
         inicio = 0;
      return elem;
   }
   else
      return 0; /* zero ou outro valor especial */
}
```

Questão: Porque usamos as variáveis `inicio` e `ocupacao` em vez de tentar uma solução mais elegante?

4.4.2 Fila implementada com uma lista ligada

Vamos então ver como se pode guardar uma fila numa lista ligada alocada dinamicamente. A implementação do nodo da fila é:

```
struct fNodo {
   float elemento;
   struct fNodo *pprox;
};
struct fNodo *fila=NULL; // Indicar que a fila está vazia
```

O início da fila agora coincide com o início da lista ligada, logo não precisamos de o guardar numa variável separada. O fim da fila aponta para a última posição da lista ligada, então devemos guardá-lo numa variável separada:

```
struct fNodo *pFim = NULL;
```

A implementação da função `inserirFila()` que insere um elemento no fim da fila fica:

```
void inserirFila (float elem) {
   // Alocar espaço para o novo nodo da fila
   struct fNodo *nNodo = (struct fNodo *) malloc(sizeof(struct fNodo));
   nNodo->elemento = elem;
   nNodo->pprox = NULL; // O elemento inserido passa a ser o elemento final
   if(pFim != NULL)     // A fila não está vazia
      pFim->pprox = nNodo;
   else // Fila está vazia => elemento inserido passa a ser o início da fila
      fila = nNodo;
   pFim = nNodo;
}
```

Como se pode observar, inserir um elemento não afeta o início (apontador `fila`) desde que a fila não esteja vazia.

O código C da função `removerFila()`, que retira um elemento do início da fila, é:

```
float removerFila(void) {
   if (fila != NULL) { // A fila não está vazia
      struct fNodo *pelem = fila;
      float elem = fila->elemento;
      fila = pelem->pprox;
      if (pelem == pFim) // A fila vai ficar vazia
         pFim = NULL;
      free(pelem); // Libertar a memória do nodo retirado da fila
      return elem;
   }
   else
      return 0; // zero ou outro valor especial
}
```

Retirar um elemento não afeta o fim da lista a não ser quando a fila fica vazia.

4.5 Árvores Binárias e Árvores Binárias de Pesquisa

Bibliografia: *Introduction to Algorithms*, seções 12.2 e 12.3.

4.5.1 Introdução

Uma **árvore binária** é uma estrutura de dados dinâmica, onde cada nodo tem no máximo dois nodos filho e no máximo um pai. Uma **árvore binária de pesquisa** é uma árvore binária com ordenação entre os seus filhos. Uma árvore binária de pesquisa suporta várias **operações** sobre o conjunto dinâmico que representa: pesquisar, calcular o mínimo e o máximo, calcular o antecessor e o sucessor, inserir e remover. Assim, uma árvore binária de pesquisa pode ser utilizada como um dicionário ou como uma fila de prioridade.
Profundidade duma árvore é o caminho mais longo, em número de arcos, entre o nodo raiz e qualquer nodo terminal. As operações básicas sobre uma árvore binária de pesquisa demoram um tempo proporcional à profundidade da árvore. Numa árvore completamente binária, com n nodos, o **tempo de execução** destas operações é proporcional a $O(\log n)$, no pior caso.

Para o caso concreto duma árvore binária de pesquisa, vamos ver como:

- percorrer a árvore de modo a imprimir os seus valores de forma ordenada;
- procurar um valor na árvore;
- encontrar o elemento mínimo ou máximo da árvore;
- encontrar o antecessor ou sucessor de um elemento;
- inserir ou remover um elemento na/da árvore.

4.5.2 Árvores Binárias de Pesquisa

A árvore binária de pesquisa está organizada como uma árvore binária. Esta árvore pode ser implementada por uma lista ligada. Cada **nodo** contém uma chave, dados úteis (opcionalmente), um apontador para o nodo filho da esquerda, um apontador para o nodo filho da direita e um apontador para o nodo pai. Se um nodo filho ou pai não existe, o respetivo apontador é NULL. O **nodo raiz** é o único nodo na árvore cujo campo pai é NULL.

Implementação do nodo da árvore binária de pesquisa em C:

```
struct nodoA {
       int            chave;
       <tipo>         dadosUteis;
       struct nodoA* nodoPai;
       struct nodoA* nodoEsq;
       struct nodoA* nodoDir;
};
```

Para qualquer nodo x, as chaves na sub-árvore esquerda de x são no máximo iguais à chave de x, e as chaves na sub-árvore direita de x, são no mínimo iguais à chave de x. Pode usar-se árvores binárias de pesquisa diferentes para representar o mesmo conjunto de valores (figura 19). A figura 19 (a) representa uma árvore binária de pesquisa com 6 nodos e profundidade **2**. Enquanto a figura (b) representa um árvore binária de pesquisa com as mesmas chaves que (a), mas menos eficiente porque a profundidade é **4**.

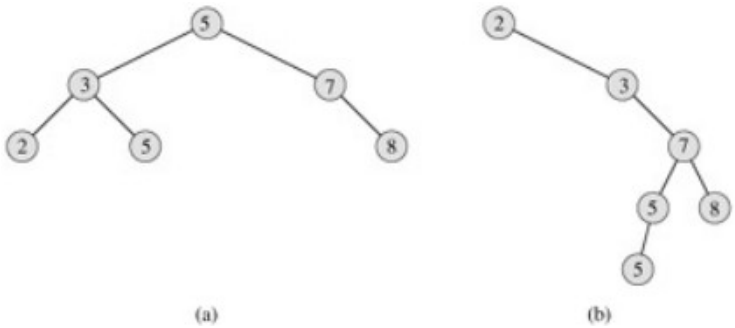


Figura 19 - Exemplos de árvores binárias de pesquisa.

As chaves de uma árvore binária de pesquisa são sempre armazenadas de modo a satisfazer a seguinte **propriedade**:

- Sendo x um nodo de uma árvore binária de pesquisa;
- Se y é um nodo na sub-árvore esquerda de x, então a chave de y é ≤ chave de x;

- E se z for um nodo na sub-árvore direita de x, então a chave de $x \leq$ chave de z.

Na figura 19 (a), a chave da raiz é 5, as chaves 2, 3 e 5 na sua sub-árvore esquerda não são maiores do que 5, e as chaves 7 e 8, na sua sub-árvore direita não são menores do que 5. A mesma propriedade aplica-se a cada nodo da árvore. Esta propriedade permite imprimir todas as chaves duma árvore de forma ordenada, usando um algoritmo recursivo simples: `Travessia_Ordenada_Arvore()`. Este algoritmo é chamado assim porque a chave da raiz de uma sub-árvore é impresso entre os valores da sua sub-árvore esquerda e os da sub-árvore direita. Para imprimir todos os elementos da árvore T usamos `Travessia_Ordenada_Arvore(T.raiz)`.

```
Travessia_Ordenada_Arvore(x)
  se x != NULL então
    Travessia_Ordenada_Arvore(x->nodoEsq)
    Imprimir a chave de x
    Travessia_Ordenada_Arvore(x->nodoDir)
  fse
```

O algoritmo imprime as chaves das duas árvores da figura anterior pela ordem correta: 2, 3, 5, 5, 7, 8. Com o algoritmo anterior, o tempo de travessia da árvore binária de pesquisa com n nodos é proporcional a n. Ou seja, a complexidade é $O(n)$.

Questão: Para o conjunto de chaves {1, 4, 5, 10, 16, 17, 21}, desenhar árvores binárias de pesquisa com profundidade 2, 3, 4, 5 e 6.

Resolução:

- A figura 20 mostra duas árvores com as chaves {1, 4, 5, 10, 16, 17, 21}, uma com profundidade **6** (a) e outra com profundidade **5** (b).

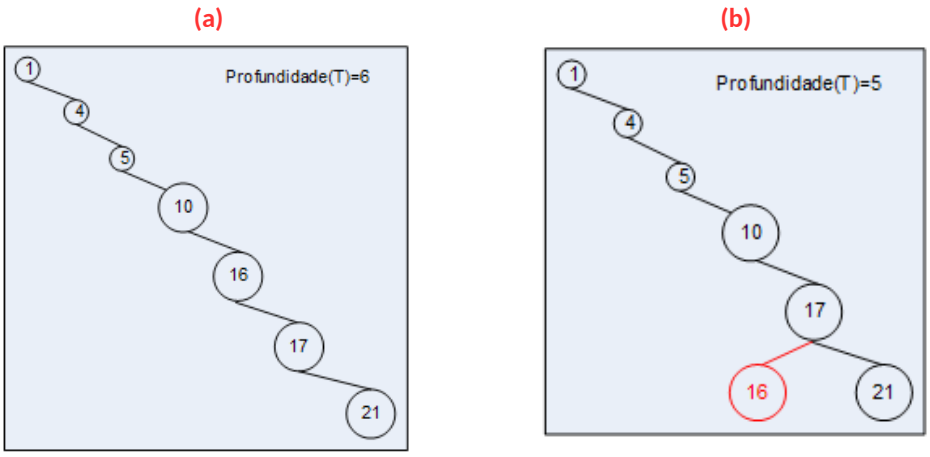


Figura 20 - Árvores com profundidade 6 (a) e 5 (b).

- A figura 21 mostra duas árvores com as chaves {1, 4, 5, 10, 16, 17, 21}, uma com profundidade **4** (a) e outra **3** (b).

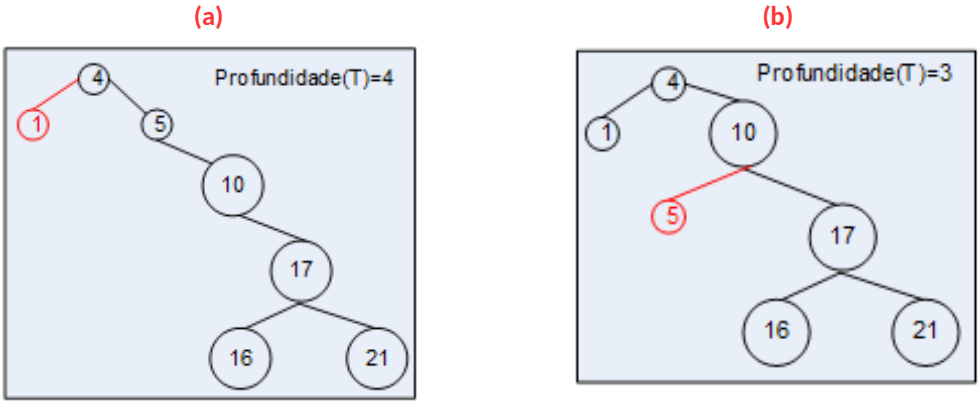


Figura 21 - Árvores com profundidade 4 (a) e 3 (b).

- A figura 22 mostra uma árvore com as chaves {1, 4, 5, 10, 16, 17, 21} e profundidade **2**.

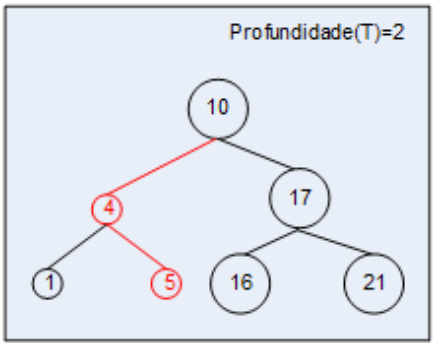


Figura 22 - Árvore com profundidade 2.

4.5.2.1 Árvores binárias de pesquisa: procurar o nodo com uma dada chave

O tempo de pesquisa de um nodo numa árvore é proporcional à sua profundidade, que no pior dos casos é igual ao número de nodos *n: O(n)*. A função `Pesquisa_Arvore()`, que vamos mostrar, procura o nodo com uma dada chave *k* numa árvore binária de pesquisa. Partindo do nodo raiz *x*, a função devolve o apontador para o nodo que tiver a chave igual a *k*, caso ele exista, ou NULL se não existir.

```
Pesquisa_Arvore (x, k)
   se x = NULL ou k = x->chave então
      devolve x
   fse
   se k < x->chave então
      devolve Pesquisa_Arvore(x->nodoEsq, k)
   senão
      devolve Pesquisa_Arvore(x->nodoDir, k)
   fse
```

O algoritmo começa a pesquisa na raiz *x* e segue uma trajetória descendente na árvore. Para cada nodo *x* encontrado, compara-se a chave *k* com a chave de *x*. Se as duas chaves são iguais, a pesquisa termina. Se *k* é menor do que a chave de *x* a pesquisa continua na sub-árvore esquerda de *x*. Senão, se *k* é maior do que a chave de *x*, a pesquisa continua na sub-árvore direita de *x*.

Exemplo de pesquisa de um nodo

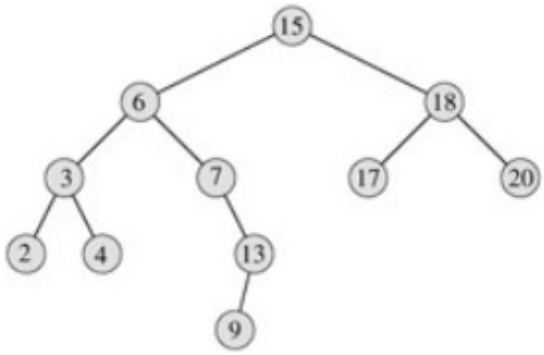


Figura 23 - Outro exemplo de árvore binária de pesquisa.

Para procurar a chave 13, seguimos o caminho **15** → esq de 15 → **6** → dir de 6 → **7** → dir de 7 → **13** a partir da raiz. A chave **mínima** da árvore é 2 e pode ser encontrada seguindo sempre o apontador da esquerda a partir da raiz. A chave **máxima** é 20 e é encontrada seguindo sempre o apontador da direita a partir da raiz.

4.5.2.2 Árvores binárias de pesquisa: calcular o nodo com chave mínima e máxima

O algoritmo seguinte devolve um apontador para o nodo com a chave mínima da sub-árvore cuja raiz é o nodo *x*.

```
Minimo_Arvore(x)
    enquanto x->nodoEsq ? NULL fazer
        x = x->nodoEsq
    fenquanto
devolver x
```

O algoritmo que procura o nodo com a chave máxima é semelhante.

```
Maximo_Arvore(x)
    enquanto x->nodoDir ? NULL fazer
        x = x->nodoDir
    fenquanto
devolver x
```

O tempo de pesquisa do máximo e do mínimo é proporcional à profundidade da árvore, que no pior dos caso é igual ao número de nodos *n*: *O(n)*.

4.5.2.3 Árvores Binárias de Pesquisa – calcular o nodo sucessor e antecessor

Dado um nodo duma árvore, por vezes é importante ser capaz de encontrar o seu sucessor, numa sequência ordenada que compõe um determinado caminho na árvore. Se todas as chaves forem distintas, o **sucessor** dum nodo *x* é o nodo com a menor das chaves que seja maior do que a chave de *x*. A estrutura duma árvore binária de pesquisa permite-nos obter o sucessor de um nodo sem fazer comparações de chaves.

O próximo algoritmo devolve o sucessor de um nodo *x* se ele existir, ou NULL se a chave de *x* for a maior da árvore.

```
Sucessor_Arvore(x)
    se x->nodoDir ? NULL então
        devolver Minimo_Arvore(x->nodoDir)
    fse
    y = x->nodoPai
    enquanto y ? NULL e x = y->nodoDir fazer
        x = y
        y = y->nodoPai
    fenquanto
    devolver y
```

O código do algoritmo `Sucessor_Arvore()` pode ser dividido em dois casos:

1. Se a sub-árvore direita que começa no nodo *x* não está vazia, então o sucessor de *x* é o nodo mais à esquerda (mínimo) da sub-árvore direita. Neste caso, o cálculo do sucessor é feito com a parte a verde de `Sucessor_Arvore()`.

Questão: Qual o sucessor do nodo com chave 15 na árvore da figura 23?

Resposta: É o nodo com chave 17.

2. Se a sub-árvore da direita de nodo *x* está vazia e *x* possui um sucessor *y*, então *y* é o menor antepassado de *x*, cujo filho esquerdo também é um antepassado de *x*. Neste caso, o cálculo do sucessor é feito com a parte vermelha de `Sucessor_Arvore()`.

Exemplo: Qual o sucessor do nodo com chave 13 na na árvore da figura 23?

Resposta: É o nodo com chave 15.

Exercício: Escrever o algoritmo `Antecessor_Arvore(x)`, que obtém o antecessor do nodo *x* numa árvore binária de pesquisa. O **antecessor** dum nodo *x* é o nodo com a maior das chaves que seja menor do que a chave de *x*.

4.5.2.4 Árvores binárias de pesquisa: inserir e remover um nodo

As operações de inserção e de eliminação dum nodo alteram o conjunto dinâmico representado por uma árvore binária de pesquisa. A estrutura de dados vai ser modificada para refletir esta alteração, mas tem que se manter válida a **propriedade** das chaves duma árvore binária de pesquisa. Modificar a árvore para inserir um novo elemento é relativamente simples, mas eliminar um nodo é um processo mais complicado.

Para **inserir** um elemento *z* numa árvore *T* usamos o algoritmo `Inserir_Arvore(T,z)`. O nodo *z* a inserir possui uma chave *v*, o *nodoPai*=NULL, o *nodoEsq*=NULL e o *nodoDir*=NULL. `Inserir_Arvore(T,z)` começa na raiz da árvore *T* e segue uma trajetória descendente. O apontador *x* define o caminho seguido e o apontador *y* segue-o, mantendo-se como o pai de *x*. No ciclo `enquanto`, os dois apontadores descem a árvore, indo para a esquerda ou para a direita dependendo da comparação da chave de *z* com a chave de *x*, até que *x* seja NULL. Quando *x* é NULL encontrámos a posição onde vamos inserir o elemento *z*.

```
Inserir_Arvore(T,z)
    y = NULL
    x = raiz de T
    enquanto x ? NULL fazer
         y = x
         se z->chave < x->chave então
              x = x->nodoEsq
         senão
              x = x->nodoDir
         fse
    fenquanto
    z->nodoPai = y
    se y = NULL então
         raiz de T = z
    senão se z->chave < y->chave então
         y->nodoEsq = z
    senão y->nodoDir = z
    fse
```

Exemplo: Inserir um elemento com chave 13 na árvore da figura 24. Os nodos a cinzento claro indicam o caminho seguido desde a raiz até à posição em que o elemento é inserido. A linha a tracejado indica a ligação que é criada ao inserir o elemento 13.

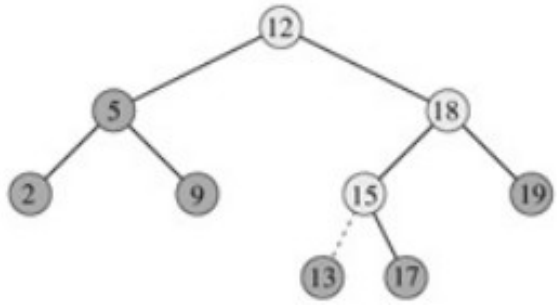
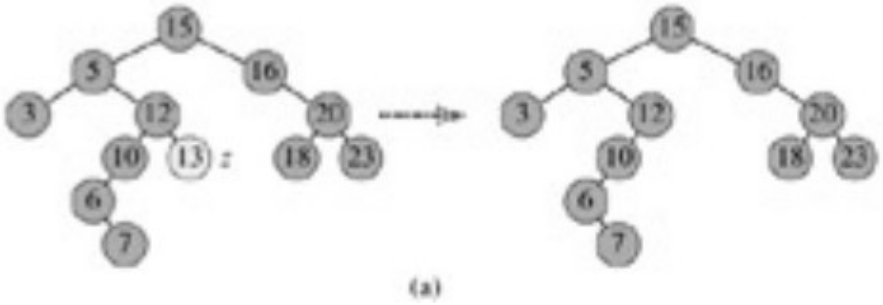


Figura 24 - Árvore binária de pesquisa para inserir um nodo.

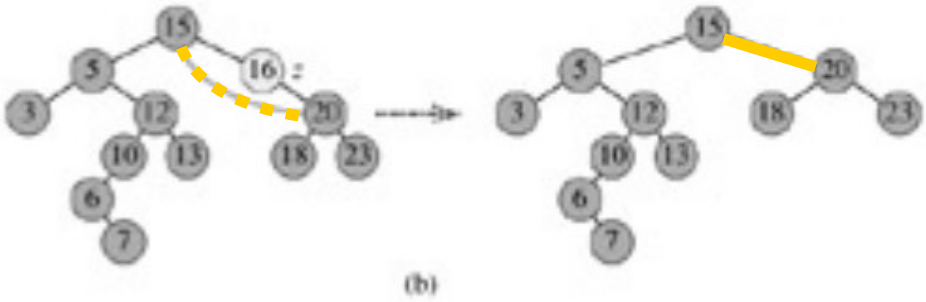
O algoritmo que vamos usar para **remover** um nodo *z* de uma árvore recebe como argumento o apontador para *z*. O algoritmo contempla 3 casos, ilustrados na figura 25:

1. Se *z* não tem filhos, modificamos o pai de *z* de modo a substituir o filho *z* por NULL;
2. Se o nodo *z* tem apenas um filho, fazemos uma nova ligação entre o filho de *z* e o pai de *z*;

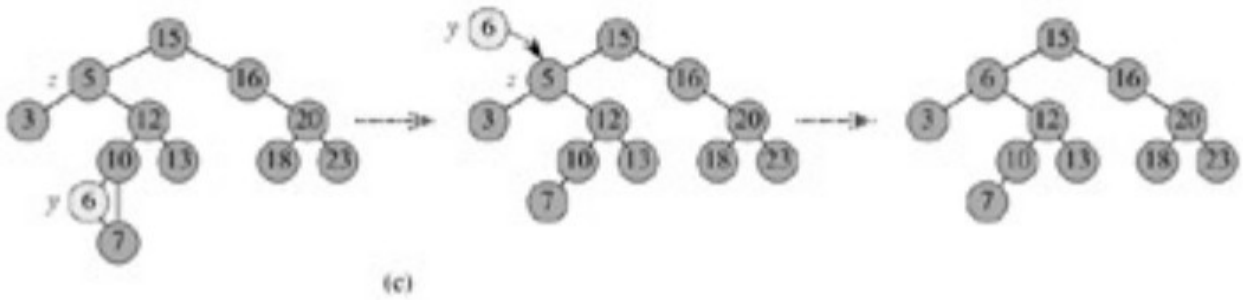
3. Se o nodo z tem 2 filhos, removemos o sucessor de z (y) do seu local e substituímos a chave e os dados úteis de z pelos respetivos valores do nodo y. Isto equivalente a substituir z pelo seu sucessor y.



(a) Remover o nodo z com chave **13**, o qual não tem filhos.



(b) Remover o nodo z com chave **16**, o qual tem um filho com chave 20.



(c) Remover o nodo z com chave **5**, o qual tem dois filhos com chaves 3 e 12.

Figura 25 - Árvore binária de pesquisa para remover um nodo.

Na árvore da esquerda na figura 25 (c), a ligação assinalada entre os nodos 10 e 7, é uma ligação que se vai estabelecer porque o nodo 6 (y) é retirado do seu lugar inicial. Na árvore central o nodo 6 (y), que é o sucessor de 5 (z), assume o lugar do nodo 5. Ainda na figura central, o nodo 7 assume o antigo lugar do nodo 6.

```
Remover_Arvore(T, z)
    se z->nodoEsq = NULL ou z->nodoDir = NULL então
       y = z
    senão
       y = Sucessor_Arvore(z)
    fse
    se y->nodoEsq ? NULL então
       x = y->nodoEsq
    senão
       x = y→nodoDir
    fse
```

Determinar o nodo a corrigir devido a eliminar-se $z \rightarrow y$

Determinar o filho não NULL de $y \rightarrow x$

```
se x ? NULL então
    x->nodoPai = y->nodoPai
fse
se y->nodoPai = NULL então
    raiz de T = x
senão se y = (y->nodoPai)->nodoEsq então
    (y->nodoPai)->nodoEsq = x
senão
    (y->nodoPai)->nodoDir = x
fse
se y ? z então
    chave de z = chave de y
    copiar os dados úteis de y para z
fse

devolver y   // y é o nodo a remover de      memória
```

Corrigir o nodo *x*

Corrigir o pai de *y*

Copiar os dados do sucessor de *z* (*y*) para *z*

4.6 Grafos

Bibliografia: *Introduction to Algorithms*, seções 22.1 e 22.2.

Um grafo G é composto por um conjunto de **arcos** E (***edges***) que ligam os **vértices** V desse grafo: $G = (V, E)$. Vamos começar por ver métodos para representar um grafo e para fazer pesquisas nesse grafo. Os dois tipos de representação de grafos mais usados são as listas de adjacência e a matriz de adjacência. Ambas as representações podem ser usadas com grafos direcionados e não direcionados. Num **grafo direcionado** os arcos possuem direção.

A representação com listas de adjacência é normalmente preferida porque proporciona uma forma compacta de representar grafos esparsos. Pode dizer-se que um **grafo é esparso** se $|E| << |V|^2$. A representação com matriz de adjacência é opção quando o **grafo é denso**, ou seja, quando $|E| \approx |V|^2$.

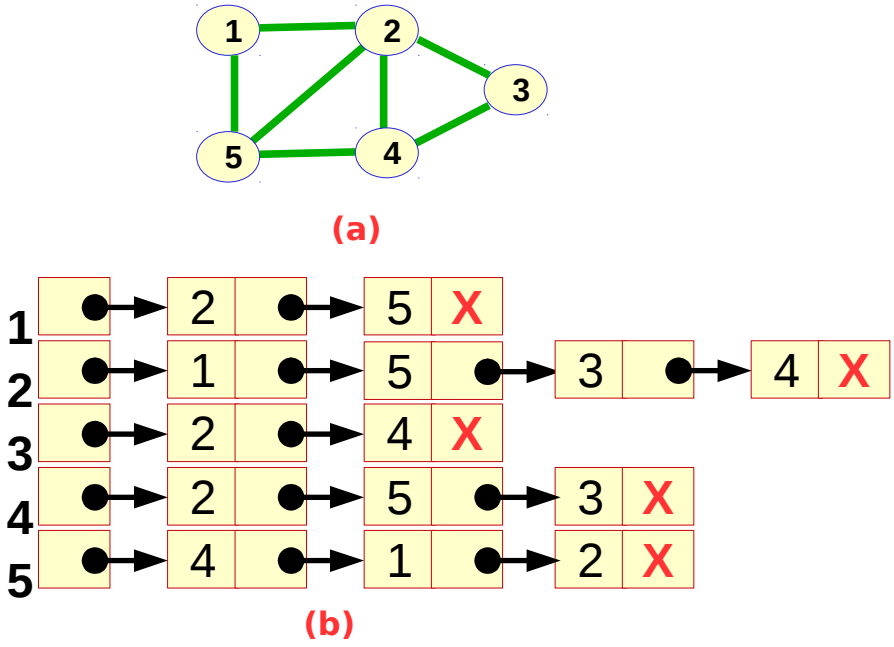


Figura 26 - Um grafo não direcionado (a) e representação com listas de adjacência (b).

4.6.1 Representação de grafos com listas de adjacência

A representação dum grafo $G=(V,E)$ através de **listas de adjacência** consiste num *array* Adj com $|V|$ listas, uma lista por cada vértice de V. Para cada vértice $u \in V$, a lista de adjacência $Adj[u]$ contém todos os vértices v para os quais existe uma arco $(u,v) \in E$. Dois vértices são **adjacentes** se existir um arco a ligá-los. Ou seja, a lista $Adj[u]$ inclui todos os vértices adjacentes de $u \in V$. Em alternativa a incluir vértices nas listas de adjacência podemos incluir apenas apontadores para esses vértices. Normalmente, os vértices de cada lista de adjacência são armazenados segundo uma ordem arbitrária.

A figura 26 mostra um grafo não direcionado e a sua representação com listas de adjacência. Por seu lado, a figura 27 apresenta um exemplo dum grafo direcionado e a sua representação com listas de adjacência.

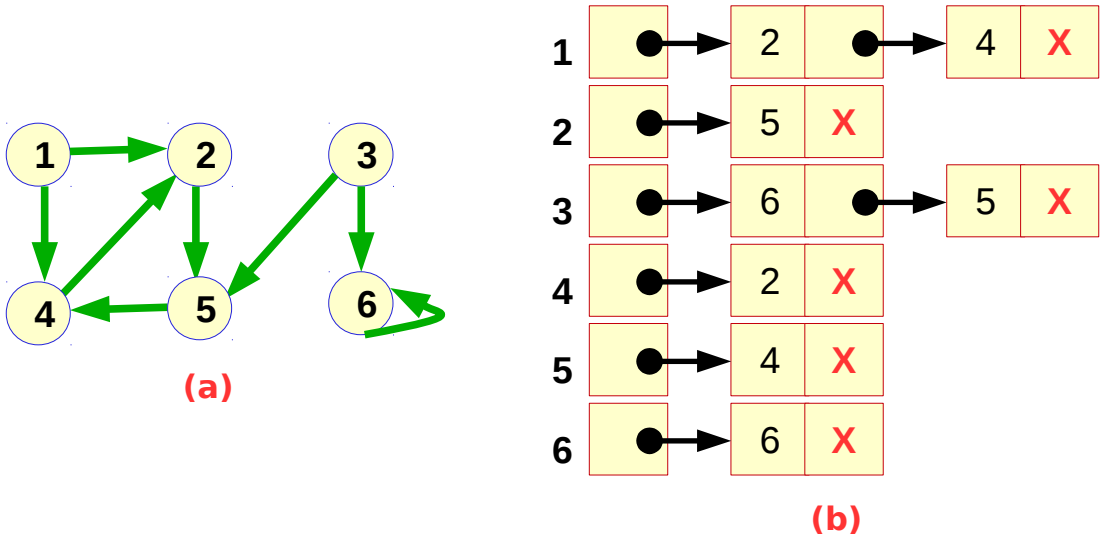


Figura 27 - Um grafo direcionado (a) e a representação com listas de adjacência (b).

Se G é um grafo direcionado, a soma dos tamanhos de todas as listas de adjacência é $|E|$. Se G é um grafo não direcionado, a soma dos tamanhos de todas as listas de adjacência é $2*|E|$. Para ambos os tipos de grafo, a representação com listas de adjacência requer uma quantidade de memória da ordem de grandeza de $|V|+|E|$, ou seja, $O(V+E)$.

As listas de adjacência podem facilmente ser adaptadas para representar grafos pesados. Num **grafo pesado** cada arco tem um peso associado. Por exemplo, se $G=(V,E)$ for um grafo pesado, o peso $w(u,v)$ do arco $(u,v) \in E$ é guardado junto com o vértice v na lista de adjacência de u.

Uma desvantagem das listas de adjacência resulta de a forma mais rápida de saber se um dado arco (u,v) existe num grafo ser procurar v na lista de adjacência $Adj[u]$. Esta desvantagem pode ser eliminada se representarmos o grafo com uma matriz de adjacência, mas isto implica utilizar mais memória.

4.6.2 Representação de grafos com matriz de adjacência

Ao representar um grafo $G=(V,E)$ com uma matriz de adjacência, assumimos que os vértices são numerados de 1 a $|V|$, de forma arbitrária. A representação do grafo G com matriz de adjacência consiste numa matriz $A=(a_{ij})$, com dimensão $|V|\times|V|$, em que:

$$a_{ij} = \begin{cases} 1 \rightarrow se\ o\ arco(i,j) \in E \\ 0 \rightarrow outros\ casos \end{cases}$$

A matriz de adjacência dum grafo com $|V|$ vértices ocupa um espaço em memória proporcional a $O(V^2)$, independentemente do número de arcos $|E|$ desse grafo.

A figura 28 mostra um exemplo de grafo não direcionado e a sua representação com matriz de adjacência.

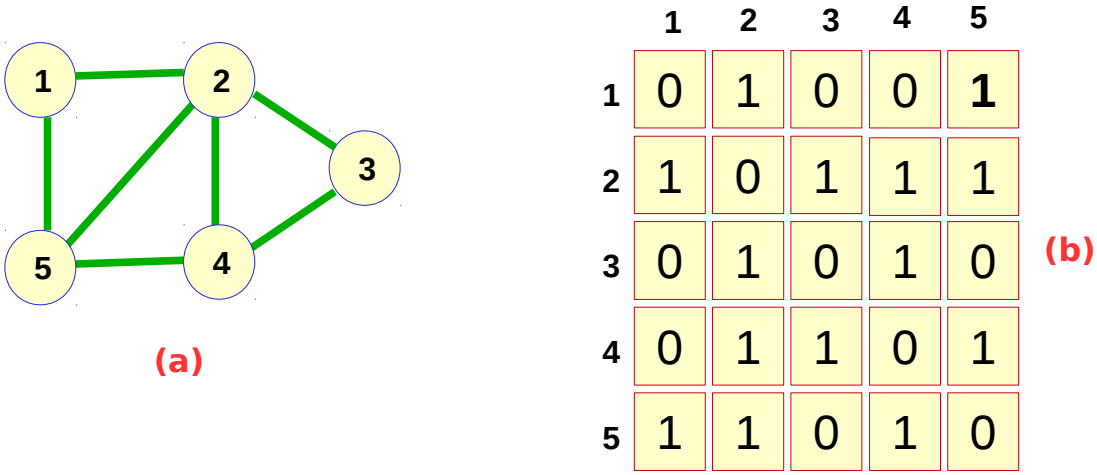


Figura 28 - Um grafo não direcionado (a) e a representação com matriz de adjacência (b).

A matriz de adjacência para um grafo não direcionado é simétrica. As posições `(u,v)` e `(v,u)` representam o mesmo arco. Assim, podemos guardar apenas as posições da diagonal e as posições localizadas acima dela. Deste modo, reduzimos a memória necessária para guardar a matriz para aproximadamente metade.

A figura 29 mostra um exemplo de grafo direcionado e a sua representação com matriz de adjacência.

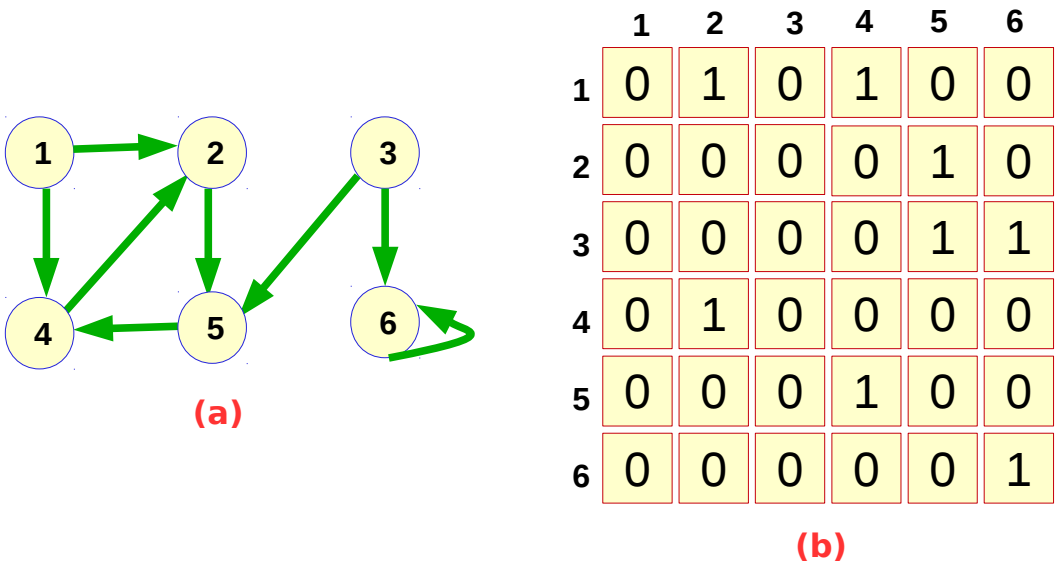


Figura 29 - Um grafo direcionado (a) e a representação com matriz de adjacência (b).

A matriz de adjacência também pode ser usada com grafos pesados. Por exemplo, se $G=(V, E)$ é um gráfico pesado, o peso $w(u,v)$ associado ao arco $(u,v) \in E$ é armazenado na linha u coluna v da matriz de adjacência. A simplicidade da matriz de adjacência torna-a atrativa quando os grafos são razoavelmente pequenos. Além disso, se o grafo não for do tipo pesado, a matriz de adjacência apresenta outra vantagem: em vez de se usar uma palavra de memória para guardar cada posição da matriz, podemos usar apenas um bit por posição.

Exercício: Representar uma árvore binária completa, contendo 7 nodos numerados de 1 a 7, usando (a) listas de adjacência e (b) uma matriz de adjacência.

4.6.3 Algoritmo de pesquisa em largura (BFS)

BFS é um dos algoritmos mais simples para efetuar pesquisas em grafos e é a base de muitos dos algoritmos utilizados com grafos. Dado um grafo $G=(V,E)$ e um vértice de origem s, o **algoritmo BFS** explora todos os arcos de G para encontrar todos os vértices acessíveis a partir de s. O algoritmo calcula o menor número de arcos (**distância**) desde s

até cada um dos vértices acessíveis. Também produz uma **árvore BF** com raiz em s que contém todos os vértices acessíveis a partir dele. Para qualquer vértice v acessível a partir de s, o caminho na árvore BF de s até v corresponde ao **caminho mais curto** entre s e v no grafo G. O algoritmo funciona com grafos direcionados e não direcionados.
O nome do algoritmo resulta de ele expandir a fronteira entre os vértices descobertos e por descobrir de forma uniforme em toda a largura (ou extensão) da fronteira. Ou seja, o algoritmo descobre todos os vértices à distância k de s antes de descobrir qualquer vértice à distância $k+1$. Para acompanhar o progresso do algoritmo, atribui-se uma de três cores a cada vértice: **`branco`**, **`cinzento`** ou **`preto`**. Todos os vértices começam a `branco`, depois passam a `cinzento` e depois a `preto`. Quando um vértice é descoberto pela primeira vez, deixa de ter a cor `branco`. Vértices com cor `cinzento` e `preto` já foram descobertos, mas a distinção entre eles serve para garantir que a pesquisa prossegue em toda a largura.

Para todos os arcos $(u,v) \in E$, em que o vértice u é `preto`, o vértice v é obrigatoriamente `cinzento` ou `preto`. Deste modo, todos os vértices adjacentes dum vértice `preto` já foram descobertos. Já os vértices `cinzentos` podem ter alguns vértices adjacentes a `branco`. Assim, a cor `cinzento` marca a fronteira entre os vértices descobertos e por descobrir.

Durante a execução do algoritmo é construída uma **árvore BF** com raiz no vértice de origem s. Sempre que um vértice **branco** v é descoberto, quando se está a pesquisar a lista de adjacência $Adj[u]$ dum vértice u já descoberto, o vértice v é adicionado à árvore BF e u passa a ser o **antecessor**/pai de v na árvore BF. As relações entre ascendentes e descendentes na árvore BF são definidas em relação à raiz s: se u está incluído num caminho entre a raiz s e o vértice v, então u é um **ascendente** de v e v é um **descendente** de u.

O algoritmo BFS que vamos apresentar assume que o grafo de entrada $G=(V,E)$ está representado com **listas de adjacência**. BFS usa as seguintes **estruturas de dados adicionais** para cada vértice do grafo:

- A cor de cada vértice $u \in V$ é armazenada na variável $cor[u]$
- O antecessor de u é armazenado na variável $p[u]$
- Se u não tem antecessor ($u=s$ ou u ainda não foi descoberto), então $p[u]=NULL$
- A distância entre a origem s e o vértice u é guardada em $d[u]$

O algoritmo usa ainda uma fila FIFO Q para gerir o conjunto de vértices com cor *CINZENTO* existentes em cada momento.

Apresenta-se agora o algoritmo de BFS de forma resumida:

```
BFS(G, s)
 1 para cada vértice u ∈ {V[G]-s} fazer
 2    cor[u] ← BRANCO
 3    d[u] ← ∞
 4    p[u] ← NULL ; fpara
 5 cor[s] ← CINZENTO
 6 d[s] ← 0
 7 p[s] ← NULL
 8 Q ← ∅
 9 InserirFila(Q, s)
10 enquanto Q?∅ fazer
11    u ← RemoverFila(Q)
12    para cada vértice v ∈ Adj[u] fazer
13       se cor[v]=BRANCO então
14          cor[v] ← CINZENTO
15          d[v] ← d[u]+1
16          p[v] ← u
17          InserirFila(Q, v)
18       fse
19    fpara
20    cor[u] ← PRETO
21 fenquanto
```

Vamos analisar o funcionamento do algoritmo. Para cada vértice u, as linhas 1-4 atribuem-lhe a cor *BRANCO*, definem $d[u]=infinito$, e definem o seu antecessor $p[u]$ *a NULL*. A linha 5 atribui a cor *CINZENTO* ao vértice de origem s, uma vez que se considera esse vértice descoberto no início do algoritmo. A linha 6 inicializa $d[s]$ a 0. A linha 7 define o antecessor do vértice de origem s a *NULL*. As linhas 8-9 inicializam a fila Q para conter apenas o vértice s. O ciclo `enquanto`, das linhas 10-21, itera enquanto existirem vértices com cor *CINZENTO*, ou seja, enquanto houver vértices

já descobertos que ainda não têm a sua lista de vértices adjacentes *(Adj)* totalmente analisada. Antes da primeira iteração, o único vértice *CINZENTO* e o único incluído na fila *Q* é o vértice de origem *s*. A linha 11 retira/remove o vértice *CINZENTO* mais antigo na fila *Q* e guarda-o em `u`. O ciclo `para` das linhas 12-19 percorre cada vértice *v* da lista de adjacência de *u (Adj[u]*). Se a cor de *v* for *BRANCO*, significa que *v* ainda não foi descoberto e o algoritmo descobre-o através das linhas 14-17:

- Muda-se a cor de *v* para *CINZENTO*;
- A distância de *v* até à origem *(d[v])* passa a ser *d[u]+1*;
- *u* é registado como antecessor de *v*;
- *v* é colocado na cauda da fila *Q*.

Quando todos os vértices da lista de adjacência de *u* estiverem examinados, *u* é colocado a *PRETO*.

Os resultados do algoritmo BFS dependem da ordem pela qual se visitam os vizinhos de cada vértice (*Adj[]*), na linha 12. Embora a árvore BF criada possa variar, as distâncias *d* não.

4.6.4 Exemplo de aplicação do algoritmo BFS

Nas figuras que mostram a evolução do algoritmo BFS, apresentadas à frente, um arco é mostrado a sombreado quando o algoritmo estabeleceu uma relação antecessor/sucessor entre os vértices ligados por esse arco. São estas relações que definem a árvore BF. Dentro de cada vértice *u* será mostrada a distância *d[u]* à origem. Mostra-se o estado da fila *Q* em cada iteração do ciclo `enquanto` (linhas 10-21). A distância *d[u]* dum vértice *u* à origem é mostrada ao lado dos vértices incluídos na fila *Q*.

Para começar, a figura 30 mostra o grafo (a) e sua representação com listas de adjacência (b).

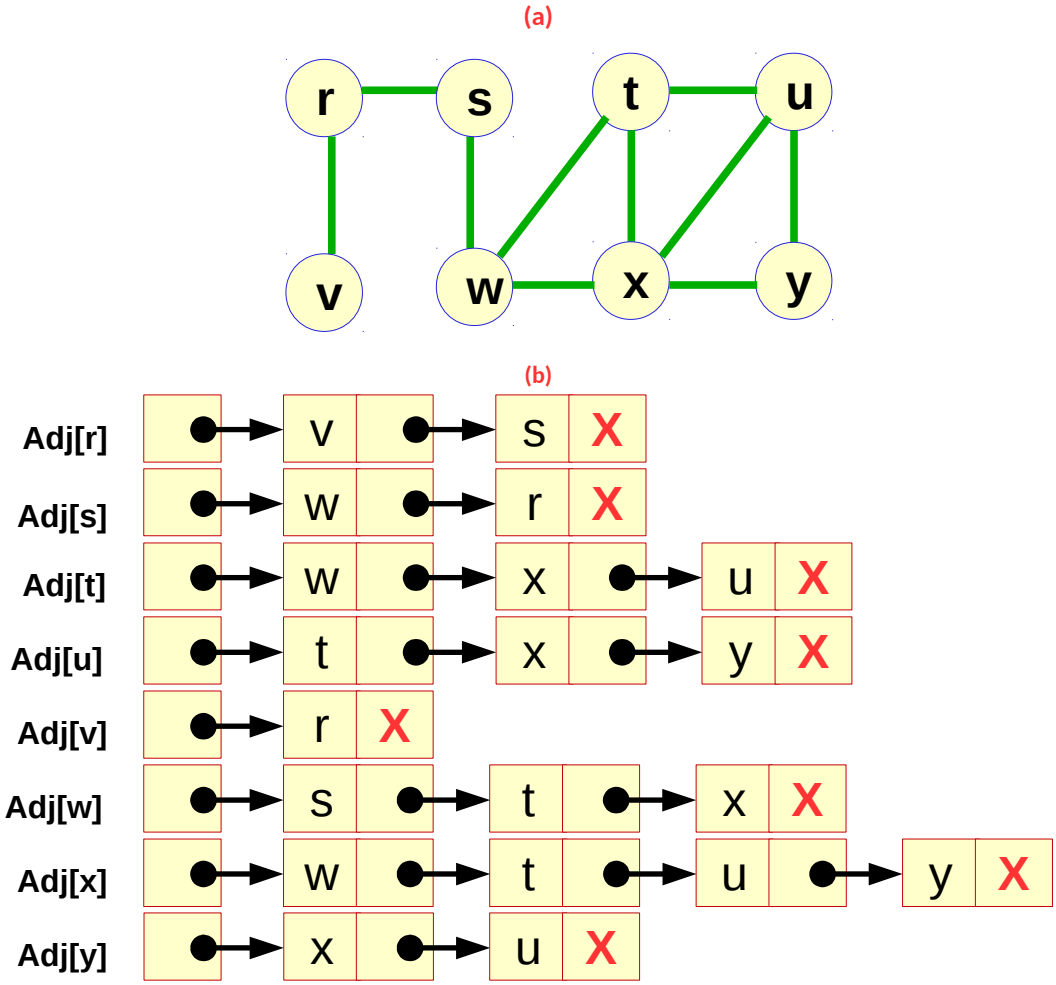


Figura 30 - Exemplo de grafo (a), e sua representação (b), utilizado para aplicar o algoritmo BFS.

A figura 31 mostra a evolução do algoritmo BFS, desde o estado inicial até terminar a sua execução, mostrando o estado de algumas variáveis (`d`, `cor` e `Q`) no fim de cada iteração do ciclo `enquanto`.

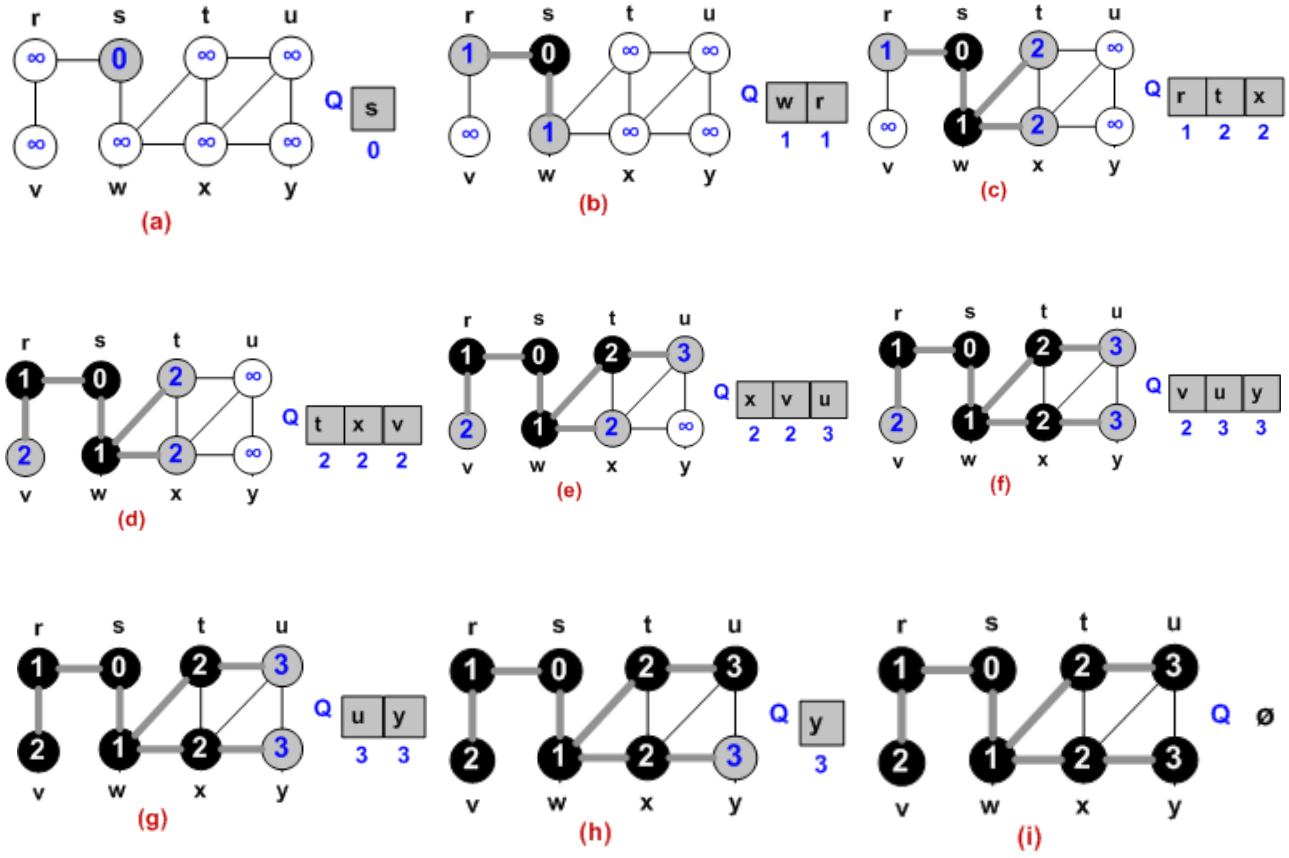


Figura 31 - Evolução do algoritmo BFS quando aplicado ao grafo da figura 30.

Caminhos mais curtos

O algoritmo BFS calcula a distância, a que se encontra da origem s, cada vértice v acessível. Definimos a distância do **caminho mais curto** $\delta(s,v)$ entre s e v como sendo o "número mínimo de arcos em qualquer dos caminhos possíveis entre o vértice s e o vértice v". Se não existir nenhum caminho entre s e v então $\delta(s,v)=\infty$. Para o exemplo anterior, a distância da origem s a cada vértice é:

$d[s]=0$	$d[v]=2$	$d[u]=3$
$d[r]=1$	$d[t]=2$	$d[y]=3$
$d[w]=1$	$d[x]=2$	

Árvore BF

O algoritmo constrói, de forma implícita, a árvore BF ao mesmo tempo que pesquisa o grafo. A árvore é construída usando a informação contida na variável p de cada vértice. Uma árvore BF é um sub-grafo $G_p=(V_p,E_p)$ formado pelos vértices acessíveis a partir de s e, para todos os vértices $v \in V_p$ existe um único caminho de s a $v \in G_p$, o qual é também um caminho mais curto entre s e $v \in G$. Para o exemplo anterior, a árvore BF obtida a partir do conteúdo do *array* $p[]$ é apresentada na figura 32.

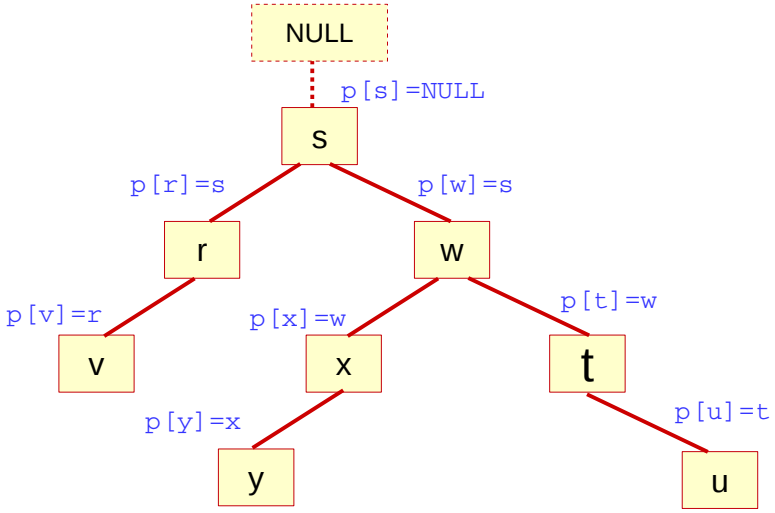


Figura 32 - Árvore BF resultante da aplicação do algoritmo BFS ao grafo da figura 30.

O algoritmo seguinte mostra no ecrã os vértices dum caminho mais curto entre *s* e *v*, assumindo que o algoritmo BFS foi previamente executado e gerou a lista *p[]* com os antecessores de cada vértice.

```
EscreveCaminho(G, s, v)
1 se v=s então
2    Escreve no ecrã s
3 senão se p[v]=NULL então
4    Escreve no ecrã "Não há caminho de " s " para " v
5 senão
6     EscreveCaminho(G, s, p[v])
7     Escreve no ecrã v
8 fse
9 Devolve
```

Para o exemplo anterior, o resultado de chamar a função `EscreveCaminho(G,'s','y')` é `s → w → x → y`.

4.7 Tabelas de *Hash*

Bibliografia: *Introduction to Algorithms*, seções 11.1, 11.2 e 11.3.

Muitas aplicações utilizam conjuntos dinâmicos que funcionam como **dicionários**, necessitando apenas das operações de inserção, pesquisa e eliminação. Por exemplo, o compilador duma linguagem de programação utiliza uma tabela de símbolos, em que a **chave** associada a cada entrada da tabela é uma *string* de carateres correspondente a uma palavra-chave dessa linguagem.

Uma **tabela *hash*** é uma estrutura de dados eficiente para implementar um dicionário. A pesquisa de um elemento numa tabela de *hash* pode demorar tanto tempo como pesquisar um elemento numa lista ligada. Deste modo, o tempo que demora uma pesquisa tem uma ordem de grandeza $\Theta(n)$, no pior caso. Assumindo certos pressupostos, o **tempo de pesquisa** dum elemento numa tabela de *hash* pode baixar para $\Theta(1)$.

Uma tabela de *hash* é uma generalização de um *array*. Endereçar diretamente um *array* tira partido da facilidade de aceder a qualquer elemento do *array* num tempo $\Theta(1)$. Quando o número de chaves armazenado é pequeno em comparação com o número de chaves possíveis, a tabela de *hash* torna-se uma alternativa eficaz aos *arrays* com endereçamento direto. A razão para a tabela de *hash* ser uma alternativa aos *arrays* com endereçamento direto reside no facto de ela utilizar um *array* com tamanho proporcional ao número de chaves armazenado. Na tabela de *hash* em vez de se usar diretamente a chave como índice do *array*, o índice do *array* é calculado a partir da chave.

4.7.1 Tabelas com endereçamento direto

Endereçamento direto é uma técnica simples que funciona bem quando o **universo de chaves** U é razoavelmente pequeno. Suponhamos que precisamos de um conjunto dinâmico em que cada elemento tem uma chave retirada do universo $U = \{0, 1, ..., m-1\}$, e em que m não é muito grande. Vamos supor que não existem elementos com a mesma

chave. Para representar o conjunto dinâmico, usamos um *array,* ou uma tabela com endereçamento direto, designado por *T[0:m-1]*, em que cada posição corresponde a uma chave no universo *U*. A figura 33 ilustra esta abordagem.

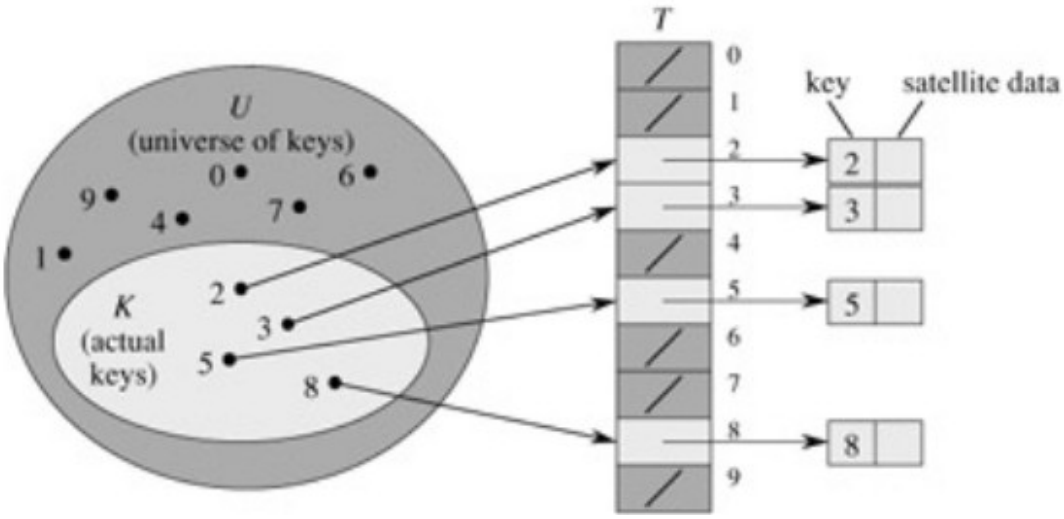


Figura 33 - Uma tabela com endereçamento direto.

Nesta figura mostra-se como se implementa um conjunto dinâmico com uma tabela de endereçamento direto *T*. Cada chave de *U*={0, 1, ..., 9} corresponde a um índice na tabela. O conjunto de chaves *k={2, 3, 5, 8}* representa as entradas da tabela que contêm apontadores para elementos válidos. As outras entradas (assinaladas com "/") estão a NULL.

As operações que suportam o dicionário, implementado com a tabela de endereçamento direto *T*, são apenas três: pesquisar, inserir e remover. Estas operações são facilmente implementadas, como se mostra no código seguinte.

```
Pesquisar_TabAcessoDireto(T, k) // procurar o elemento com chave k
    devolve T[k]

Inserir_TabAcessoDireto(T, x)   // inserir o elemento x
    k = chave de x
    T[k] = x

Remover_TabAcessoDireto(T, x)   // remover o elemento x
    k = chave de x
    Libertar x da memória       // opcional
    T[k] = NULL
```

Qualquer das operações é rápida, requerendo apenas um tempo da ordem de grandeza *Θ(1).*

Em alguns casos, os elementos do conjunto dinâmico podem ser armazenados diretamente na tabela. Nestes casos, em vez de se armazenar a chave e os dados úteis de cada elemento numa entidade externa à tabela, e depois incluir apenas um apontador na entrada adequada da tabela para essa entidade, armazenamos o próprio elemento nessa entrada da tabela. Também pode acontecer não ser necessário guardar a chave dos elementos. Neste caso o próprio índice dum elemento coincide com a sua chave. Se as chaves não são guardadas, é necessário ter uma forma de saber se as entradas da tabela estão vazias, ou seja, se não apontam para um elemento válido.

Questão: Suponha que tempos um conjunto dinâmico *S* representado por uma tabela de endereçamento direto *T* com tamanho *m*. Escreva uma função que encontra o elemento com o valor útil máximo em *S*.

Resposta:

```
Maximo_TabAcessoDireto(T) // procurar o elemento com valor útil máximo
    max = -∞
    para i=0 até m-1 fazer
        se (T[i] ? NULL) então
            x = T[i]
            val = valor útil de x
            se (val > max) então
                max = val
            fse
        fse
    fpara
    Devolver max
```

4.7.2 Tabelas de *hash*

O endereçamento direto apresenta os seguintes problemas:

- Se o universo de chaves U é grande, guardar uma tabela T com tamanho $|U|$ pode ser impraticável ou mesmo impossível;
- Se o conjunto de chaves K realmente guardadas for pequeno quando comparado com U, a maior parte do espaço de T está desperdiçado.

Quando o conjunto de chaves guardadas numa tabela (K) é muito menor do que o universo de todas as chaves possíveis (U), uma tabela *hash* requer muito menos espaço do que uma tabela de endereçamento direto. Ou seja, a ordem de grandeza do espaço ocupado é $\Theta(|K|)$. Por seu lado, o tempo de busca médio continua a ser $\Theta(1)$, mas o tempo de busca no pior caso aumenta.

Com **endereçamento direto**, um elemento com chave k é armazenado na entrada k. Com uma **tabela de *hash*** este elemento é armazenado na entrada $h(k)$. Para calcular a entrada correspondente à chave k usamos uma **função de *hash*** h. A função h mapeia o universo U das chaves nas entradas da tabela de *hash* $T[0:m-1]$:

$h: U \rightarrow \{0, 1, ..., m-1\}$

Dizemos "o elemento com chave k mapeia na entrada $h(k)$" ou "$h(k)$ é o valor de *hash* da chave k". O objetivo da função de *hash* é reduzir a gama dos índices da tabela/*array* que guarda as chaves. Em vez de $|U|$ valores, precisamos apenas de m valores. O espaço de armazenamento é reduzido proporcionalmente.

Vamos ver um exemplo sobre como a função de *hash* h mapeia as chaves em entradas da tabela T (figura 34).

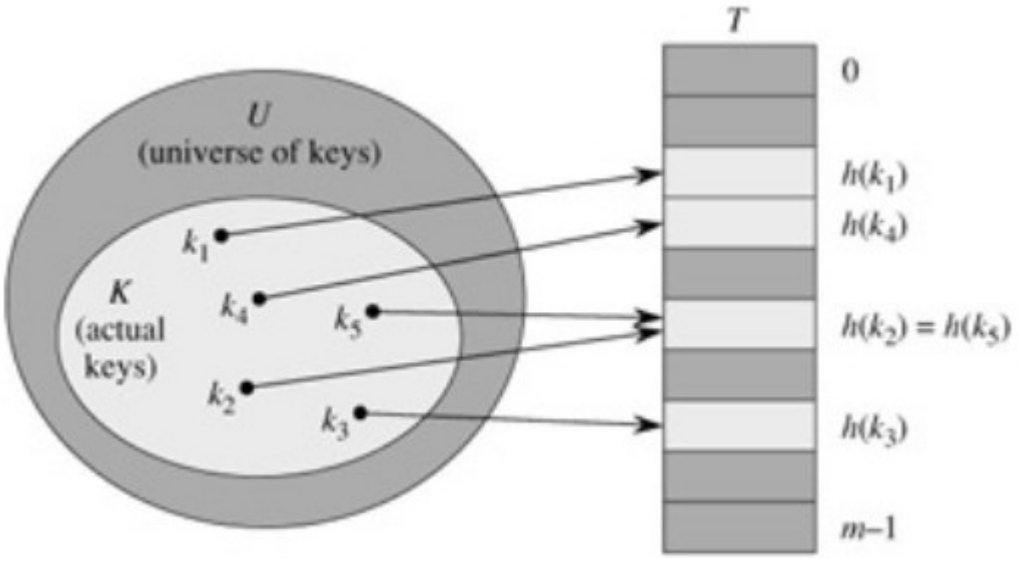


Figura 34 - Funcionamento da função de hash.

As chaves k2 e k5 mapeiam na mesma entrada, ocorrendo por isso uma **colisão**. Existem técnicas para resolver os conflitos criados pelas colisões, mas a solução ideal seria evitar as colisões. Uma aproximação a esse objetivo consiste em escolher uma função de *hash* *h* parecida com uma função aleatória, reduzindo assim as colisões ao mínimo. Contudo, a função de *hash* *h* tem que ser determinística: a uma entrada *k* corresponde sempre o mesmo resultado *h(k)*.

Como o tamanho do universo de chaves é maior que o tamanho da tabela de *hash*, $|U|>m$, haverá sempre pelo menos 2 chaves que têm o mesmo valor de *hash*, logo haverá colisões. Uma função de *hash* bem concebida pode minimizar as colisões, mas mesmo assim precisamos de um método para resolver as colisões.

4.7.2.1 Resolução de colisões por encadeamento

No método de resolução de colisões por encadeamento colocamos todos os elementos que mapeiam na mesma entrada da tabela *T* numa lista ligada. A entrada *j* da tabela *T* contém um apontador para a cabeça da lista que contém todos os elementos que mapeiam em *j*, ou NULL ("/") se a lista estiver vazia. A figura 35 ilustra a aplicação deste método.

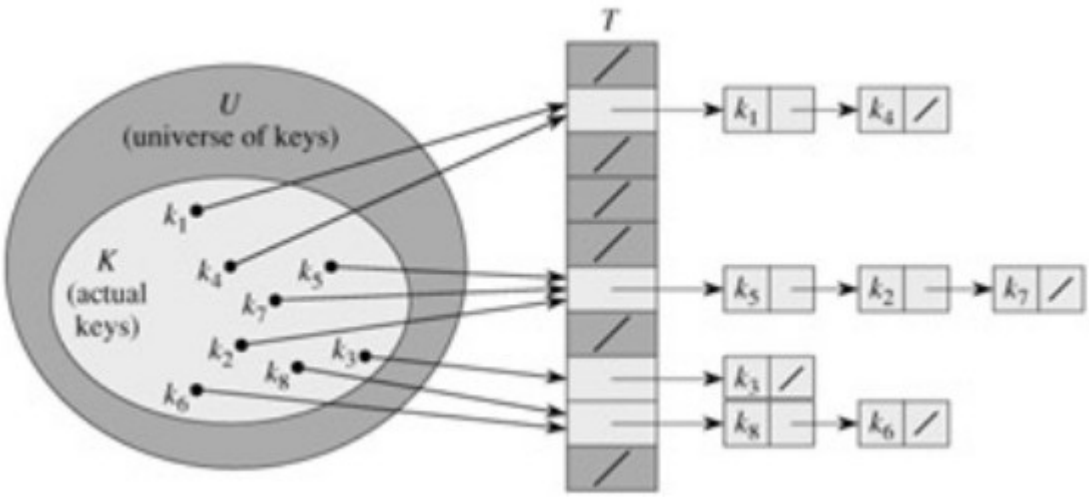


Figura 35 - Resolução de colisões pelo método de encadeamento.

Neste exemplo temos as seguintes colisões:

```
h(k1) = h(k4)
h(k2) = h(k5) = h(k7)
h(k6) = h(k8)
```

As **operações** de suporte a uma tabela de *hash* *T*, em que as colisões são resolvidas por encadeamento, são fáceis de implementar, como se pode observar pelo seguinte código para as operações pesquisar, inserir e remover.

```
Pesquisar_TabHashEncadeada(T, k)
    Procurar um elemento com chave k na lista ligada T[h(k)]

Inserir_TabHashEncadeada(T, x)
    Inserir x na cabeça da lista ligada T[h(chave de x)]

Remover_TabHashEncadeada(T, x)
    Eliminar x da lista ligada T[h(chave de x)]
```

No pior caso, o de **tempo de execução** da inserção é *Θ(1)*. A função de inserção é rápida, em parte porque se assume que o elemento *x* a inserir não existe na tabela. Esta hipótese pode ser verificada, se necessário, realizando uma pesquisa antes da inserção. No pior caso, o tempo de execução duma pesquisa é proporcional ao tamanho da lista (como veremos mais à frente). A eliminação dum elemento *x* pode ser realizada num tempo *Θ(1)* se as listas forem duplamente ligadas. Como a função de remover recebe um elemento *x* e não a sua chave *k*, não tem que efetuar qualquer pesquisa. Se as listas forem simplesmente ligadas, receber como entrada o elemento *x* em vez da chave *k*, não tem qualquer vantagem. Neste caso, para remover *x* será preciso percorrer a lista para se saber qual é o seu antecessor. Ou seja, a eliminação e a pesquisa demoram essencialmente o mesmo tempo.

4.7.2.2 Análise das tabelas de *hash* com encadeamento

Questão: Como se comportam as tabelas de *hash* com encadeamento?

Questão: Quanto tempo demora a pesquisar um elemento com uma determinada chave?

Para uma tabela de *hash* T, com m entradas e armazenando n elementos, o **fator de carga** de T é definido como $\alpha=n/m$. Podemos dizer que α é o número médio de elementos armazenados por cada lista ligada, em que α pode ser menor, igual, ou maior do que 1. O pior cenário ocorre quando todas as n chaves são mapeadas na mesma entrada de T, criando-se uma lista de tamanho n. Assim, o tempo de pesquisa neste cenário é $\Theta(n)$ mais o tempo para calcular a função de *hash*. Convém notar que as tabelas de *hash* não são utilizadas a pensar no desempenho no pior caso. O **desempenho** (médio) das tableas de *hash* depende da forma como a função de *hash* h distribui o conjunto de chaves disponíveis pelas m entradas. Por agora, vamos assumir que qualquer elemento tem a mesma probabilidade de ser mapeado em qualquer das m entradas, ou seja, o *hash* é **uniforme**.

Para j=0, 1, ..., m-1, se n_j for o tamanho da lista $T[j]$, o número total de elementos armazenados na tabela é $n = n_0 + n_1 + ... + n_{m-1}$ e o valor médio dos n_j é $E[n_j]=\alpha=n/m$. Assumindo que o valor de *hash* $h(k)$ pode ser calculado num tempo $\Theta(1)$, o tempo necessário para pesquisar um elemento com a chave k depende linearmente do tamanho $n_{h(k)}$ da lista $T[h(k)]$. Para estimar o **tempo de pesquisa** dum elemento devemos considerar dois casos:

- a pesquisa é mal sucedida porque nenhum elemento da tabela possui a chave k;
- a pesquisa encontra com sucesso um elemento com chave k.

Numa tabela de *hash* em que as colisões são resolvidas por encadeamento e assumindo a hipótese de *hash* uniforme, tanto nas pesquisas com ou sem sucesso, uma pesquisa demora em média o tempo esperado $\Theta(1+\alpha)$.

Questão: Que conclusões tirar da análise apresentada?

Se o número de entradas m da tabela *hash* for pelo menos proporcional ao número n de elementos guardados na tabela, temos que: $n=\Theta(m)$, logo $\alpha=n/m=\Theta(m)/m=\Theta(1)$. Deste modo, uma pesquisa demora um tempo médio constante $\Theta(1)$, assumindo que $\Theta(1+\alpha)=\Theta(2)=\Theta(1)$.

Como a **inserção** demora no pior caso um tempo $\Theta(1)$ e, quando as listas são duplamente ligadas, a eliminação também demora no pior caso um tempo $\Theta(1)$, então todas as operações sobre uma tabela de *hash* demoram em média um tempo $\Theta(1)$.

4.7.2.3 Funções de *hash*

Considera-se que uma função de *hash* é boa se satisfizer aproximadamente o mapeamento uniforme: uma chave tem a mesma probabilidade de ser mapeada em qualquer das m entradas, independentemente de onde estiverem mapeadas as outras chaves. Infelizmente, é pouco frequente podermos garantir que esta condição se verifica, uma vez que não conhecemos a probabilidade com que ocorre cada chave e as chaves não podem ser utilizadas de forma independente. Uma boa estratégia é calcular o valor de *hash* para que ele seja o mais independente possível de qualquer padrão que possa existir nas chaves. A maioria das funções de *hash* assume que o universo das chaves é o conjunto dos números naturais N={0, 1, 2, …}. Se as chaves não forem números naturais, deve encontrar-se uma forma de as interpretar como números naturais.

4.7.2.4 Funções de *hash* por divisão

Vamos analisar três tipos de função de *hash*: *hash* por divisão, *hash* por multiplicação e *hash* universal. No método de criação de funções de *hash* por divisão, mapeamos uma chave k na entrada $h(k)$, com $0 \le h(k) < m$, correspondente ao resto da divisão de k por m. Ou seja, a função hash é: $h(k) = k \% m$. Um exemplo: dada uma tabela de *hash* com tamanho $m=12$, a chave $k=100$ mapeia na entrada $h(k)$=100%12=4. Como a função requer apenas uma operação de divisão, o método da divisão é bastante **rápido**. No método da divisão, é comum evitar-se certos valores de m. Por exemplo, m não deve ser uma potência de 2, uma vez que se $m=2^p$, $h(k)$ coincide com os p bits menos significativos de k. Neste caso todas as chaves com os p bits menos significativos iguais mapeavam na mesma entrada da tabela. Um número primo não muito próximo duma potência de 2 é uma boa escolha para m.

Exemplo: Suponha que desejamos implementar uma tabela de *hash*, com colisões resolvidas por encadeamento, para guardar aproximadamente n=2000 *strings*. Assumindo que não nos importamos que em média 3 chaves sejam mapeadas na mesma entrada, então o tamanho m da tabela de *hash* será um número primo próximo de 2000/3≈666.66. O número *701* é escolhido para valor de m porque é um primo próximo de 2000/3 e afastado das potências de 2 mais

próximas: 512 e 1024. Se pretendermos que as chaves k sejam apenas números naturais, a função de *hash* será $h(k) = k \% 701$.

4.7.2.5 Funções de *hash* por multiplicação

O método de criação de funções de *hash* por multiplicação funciona em 2 passos:

1. Multiplica-se a chave k por uma constante A na gama $0<A<1$ e extrai-se a parte fracionária de $k*A$;
2. Depois, multiplica-se esse valor por m e aplica-se o operador *floor* "$\lfloor\ \rfloor$" ao resultado. $\lfloor x \rfloor$ é o maior inteiro não superior a x.

A função de *hash* resultante é $h(k) = \lfloor m*(k*A \% 1) \rfloor$, onde $k*A\%1$ representa a parte fracionária de $k*A$, que é igual a $k*A-\lfloor k*A \rfloor$.

Uma vantagem do método de multiplicação é o valor de m não ser crítico. Normalmente escolhe-se m igual a uma potência de 2 ($m=2^p$). Suponhamos que o tamanho da palavra do computador é w bits e que k cabe numa palavra. Forçamos A a ter um valor dado por $s/2^w$, em que s é um número inteiro na gama $0<s<2^w$. O **cálculo do valor de *hash*** é (ver a figura 36):

- Primeiro multiplicamos k pelo inteiro $s=A*2^w$ com w-bits;
- O resultado é um valor com $2*w$ bits: $r_1*2^w+r_0$, onde r_1 é a palavra/metade mais significativa do produto e r_0 é a metade menos significativa do produto.

O valor de *hash* desejado, com p bits, é dado pelos p bits mais significativos de r_0.

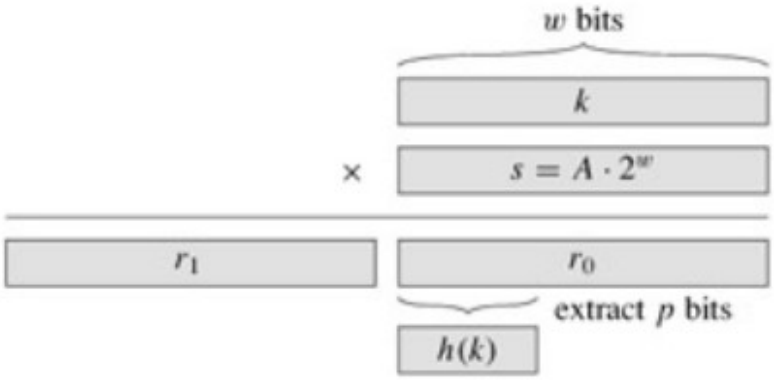


Figura 36 - Função de hash por multiplicação.

A representação da chave k (com w-bits) é multiplicada pelo valor $s=A*2^w$ (também com w-bits) Os p bits mais significativos da metade inferior do produto (r_0) formam o desejado valor de *hash:* $h(k)$.

Embora o método da multiplicação funcione com qualquer valor de A, a escolha ótima depende das características das chaves mapeadas na tabela. Um valor sugerido é: $A\approx(\sqrt{5}-1)/2\approx0.6180339887$.

Exemplo:

Suponhamos que k=123456, p=14, $m=2^{14}$=16384, e w=32

Usando a aproximação de A sugerida: $A\approx0.6180339=s/2^{32}$

Resulta que $s=0.6180339*2^{32}$, ou seja, s=2654435769

$k*s$ = 123456*2654435769=327706022297664

$k*s = (76300*2^{32}) + 17612864$

Logo, r_1=76300 e $r_0=17612864_{10}=010CC040_{16}$

Os 14 bits mais significativos de r_0 resultam no valor $h(k)=0000.0001.0000.1100.\ ...\ _2 = 67_{10}$

4.7.2.6 Funções de *hash* universais

Uma aplicação maliciosa pode escolher as chaves a mapear numa tabela de *hash*, usando uma função de *hash* fixa, de tal modo que todas sejam mapeadas na mesma entrada. O tempo médio de acesso aos dados da tabela degrada-se e passa a ser $\Theta(n)$. Qualquer função de *hash* fixa é vulnerável a este comportamento, dito de pior caso. A única forma eficaz de melhorar esta situação é usar uma função de *hash* que gera valores aleatórios e independentes das chaves a armazenar. Esta abordagem chama-se *hash* universal.

Começamos por escolher um número primo p suficientemente grande para que todas as chaves possíveis k estejam no intervalo [0:p-1].

Consideremos $Z_p=\{0, 1, ..., p-1\}$, e $Z^*_p=\{1, 2, ..., p-1\}$.

Como o tamanho do universo de chaves é maior que o número de entradas da tabela de *hash*: $p>m$.

Vamos agora definir a **função de *hash*** $h_{a,b}$, para qualquer $a \in Z^*_p$ e qualquer $b \in Z_p$, usando: (i) uma transformação linear envolvendo a e b, (ii) depois a aplicação do módulo p e (iii) por fim a aplicação do módulo m. A função de *hash* obtida é:

$$h_{a,b}(k)=((a*k+b)\%p)\%m$$

Por exemplo se p=17, m=6, k=8, a=3 e b=4, obtemos $h_{3,4}(8)=5$.

A família de todas as funções de *hash* é:

$$H_{p,m} = \{h_{a,b} : a \in Z^*_p \text{ e } b \in Z_p\}$$

Cada função de *hash* $h_{a,b}$ mapeia Z_p em $Z_m=\{0, 1, ..., m-1\}$.

Esta classe de funções de *hash* tem a propriedade de que o tamanho m da tabela gerada é arbitrário: não é necessariamente um número primo. Como existem $p-1$ valores possíveis para a e p valores possíveis para b, existem $p*(p-1)$ funções de *hash* no conjunto $H_{p,m}$.

5. Estruturação e Implementação de Algoritmos Importantes

5.1 Algoritmos para Encontrar os Caminhos Mais Curtos em Grafos

Bibliografia: *Introduction to Algorithms*, seções 24.1, 24.2 e 24.3.

Suponhamos que um condutor pretende encontrar o trajeto mais curto entre uma cidade A (Braga) e outra cidade B (Lisboa).

Questão: A partir dum mapa de Portugal, que disponibiliza a distância entre cruzamentos adjacentes, como podemos encontrar o caminho mais curto entre A e B?

Uma possibilidade consiste em identificar todos os caminhos entre Braga (A) e Lisboa (B), somar as distâncias parciais de cada caminho, e selecionar o caminho com menor distância global. É fácil perceber, que mesmo que não consideremos caminhos contendo ciclos, há imensas possibilidades, a maioria das quais nem vale a pena considerar. Por exemplo, um trajeto entre Braga e Lisboa, passando por Madrid, é obviamente uma má opção, porque a cidade intermédia (Madrid) fica completamente "fora da rota" entre a origem e o destino. Nesta seção, vamos ver como resolver este tipo de problema de forma eficiente.

Para resolver o problema que consiste em encontrar o caminho mais curto, vamos assumir que disposmos de um grafo **direcionado** e **com pesos** $G=(V,E)$, em que a função $w: E \rightarrow R$ faz o mapeamento entre os arcos e valores reais para o seu peso. O **peso do caminho** $p=v_0,v_1,...,v_k$ representa a soma dos pesos das arcos que o compõem:

$$w(p)=\sum_{i=1}^{k} w(v_{i-1},v_i)$$

Definimos o **peso do caminho mais curto** entre u e v como:

$$\delta(u,v)=\begin{cases} min\{w(p): u \rightarrow v\} \rightarrow se\ há\ um\ caminho\ de\ u\ para\ v \\ \infty \rightarrow outros\ casos \end{cases}$$

Um caminho mais curto do vértice u para v é definido como qualquer caminho p com o peso $w(p)=\delta(u,v)$.

No exemplo Braga-Lisboa, podemos representar o mapa de estradas através dum **grafo**: os **vértices** representam cruzamentos, os **arcos** representam segmentos de estrada entre cruzamentos, e os **pesos** dos arcos representam distâncias. O nosso objetivo é encontrar um caminho mais curto desde um determinado cruzamento em Braga (por exemplo, entre a Av. da Liberdade e a Av. João XXI) até um determinado cruzamento em Lisboa (por exemplo, entre a Av. Marechal Gomes da Costa e a Av. Almirante Gago Coutinho).

O **peso dos arcos** pode representar outra métrica distinta de distância, como por exemplo o tempo, o custo, a penalização, a perda ou qualquer outra quantidade acumulável ao longo de um caminho, e que pretendemos minimizar. O algoritmo BFS, que vimos anteriormente, é um algoritmo de procura do caminho mais curto, mas funciona apenas com grafos sem pesos, isto é, grafos em que os arcos têm todos um peso unitário.

5.1.1 Variantes deste tipo de algoritmo

Nesta seção vamos focar-nos no problema de encontrar os caminhos mais curtos **a partir dum único ponto inicial**: dado um grafo $G=(V,E)$, queremos encontrar um caminho mais curto a partir dum dado vértice inicial $s \in V$ para cada um dos vértices $v \in V$. Este algoritmo consegue resolver outros problemas, nomeadamente as seguintes variantes:

- Encontrar um caminho mais curto para um único vértice de destino v a partir de qualquer vértice u;
- Encontrar um caminho mais curto de u para v, dado um par de vértices u e v fixos;
- Encontrar um caminho mais curto de u para v, para todos os pares de vértices u e v.

5.1.2 Estrutura dos caminhos parciais otimizada

Os algoritmos que calculam os caminhos mais curtos geralmente dependem da seguinte **propriedade**: um caminho mais curto entre dois vértices inclui sub-caminhos mais curtos dentro dele.

Dado um grafo direcionado com pesos $G=(V,E)$, em que a função que calcula os pesos é $w: E \rightarrow R$. Se $p=v1,v2,...,vk$ for um caminho mais curto do vértice $v1$ para vk e para quaisquer i e j tais que $1 \leq i \leq j \leq k$, consideremos $pij=vi,vi+1,...,vj$ o sub-caminho de p entre os vértice vi e vj. Então, pij é um caminho mais curto de vi para vj.

5.1.3 Arcos com peso negativo

Em alguns casos, pode haver arcos com **peso negativo**. Se o grafo $G=(V,E)$ não inclui ciclos com peso negativo que sejam acessíveis a partir do vértice inicial s, então, para qualquer $v \in V$, o peso do caminho mais curto $\delta(s,v)$ é **bem definido**. Se existir um ciclo com peso negativo acessível a partir de s, os pesos dos caminhos mais curtos **não são bem definidos**. Se existir um ciclo com peso negativo num caminho de s para v, então definimos $\delta(s,v)=-\infty$.

Com o grafo da figura 37 vamos observar o efeito dos arcos com peso negativo e dos ciclos com peso negativo sobre o peso do caminho mais curto.

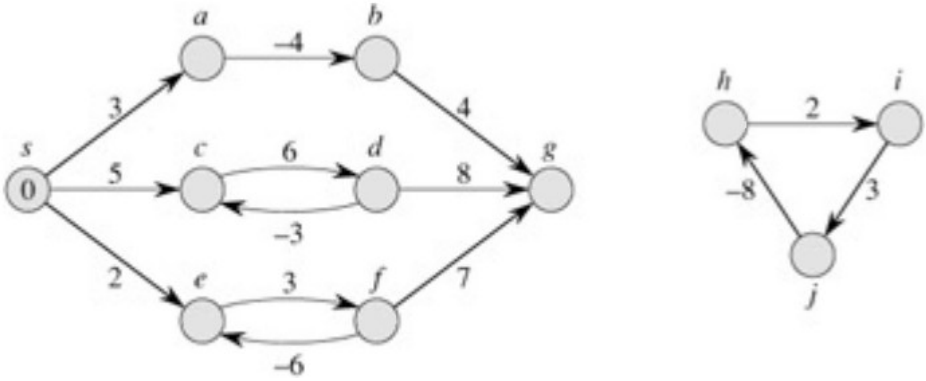


Figura 37 - Grafo com arcos e ciclos com peso negativo.

Questão: Qual o valor a incluir dentro de cada vértice, sabendo que ele corresponde ao peso do caminho mais curto desde a origem s até esse vértice?

Resposta: Existe apenas um caminho de s para a, logo o caminho é $\{s,a\}$ e o seu peso é $\delta(s,a)=w(s,a)=3$. Também existe apenas um caminho de s para b, logo $\delta(s,b)=w(s,a)+w(a,b)=3+(-4)=-1$. Existem inúmeros percursos de s para c: $\{s,c\}$, $\{s,c,d,c\}$, $\{s,c,d,c,d,c\}$, etc. Como o **ciclo** $\{c,d,c\}$ tem peso $6+(-3)=3>0$, o caminho mais curto de s para c é $\{s,c\}$ e tem peso $\delta(s,c)=5$. Do mesmo modo, o caminho mais curto de s para d é $\{s,c,d\}$, com peso $\delta(s,d)=w(s,c)+w(c,d)=11$. Analogamente, há inúmeros caminhos de s para e: $\{s,e\}$, $\{s,e,f,e\}$, $\{s,e,f,e,f,e\}$, etc. O ciclo $\{e,f,e\}$ tem um peso $3+(-6)=-3<0$, no entanto não existe um caminho mais curto de s para e. Se atravessarmos o ciclo com peso negativo $\{e,f,e\}$ repetidamente, podemos encontrar caminhos de s para e com peso sucessivamente mais negativo, ou seja, $\delta(s,e)=-\infty$. Do mesmo modo, $\delta(s,f)=-\infty$. Como g é acessível a partir de f, também podemos encontrar caminhos com pesos muito negativos de s para g e $\delta(s,g)=-\infty$. Os vértices h, i e j também formam um ciclo com peso negativo. Mas, como estes vértices não são acessíveis a partir de s, $\delta(s,h)=\delta(s,i)=\delta(s,j)=+\infty$.

O grafo anterior, depois de anotar cada vértice com o peso do caminho mais curto desde a origem até esse vértice, fica como se mostra na figura 38.

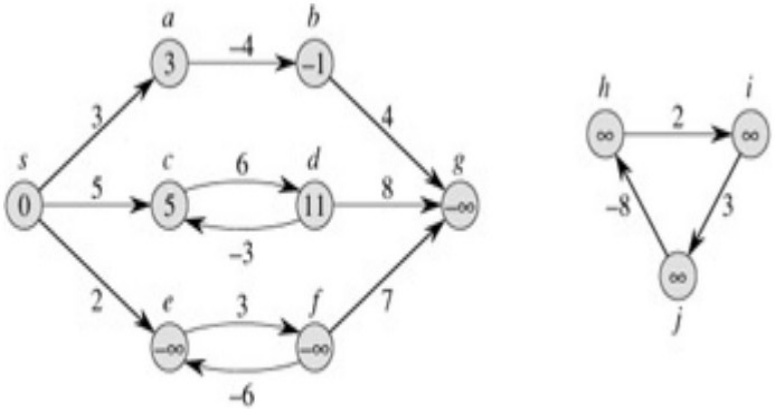


Figura 38 - Grafo da figura 37 anotado com os pesos do caminho mais curto.

5.1.4 Ciclos

Questão: Um caminho mais curto pode incluir ciclos?

Resposta: Como vimos, um caminho mais curto não pode conter ciclos com peso negativo. Também não pode conter ciclos com peso positivo, porque se removermos um ciclo do caminho geramos um caminho com a mesma origem e destino, mas com um peso menor. Vamos então demonstrar a veracidade desta afirmação:

> Consideremos $p=\{v0,v1,...,vk\}$ um caminho. Consideremos também que $c=\{vi,vi+1,...,vj\}$ é um ciclo com peso positivo incluído em p, em que $vi=vj$ e $w(c) > 0$. Então o caminho $p'=\{v0,v1,...,vi,vj+1,vj+2,...,vk\}$ tem um peso $w(p') = w(p)-w(c) < w(p)$ Assim sendo p, um caminho com um ou mais ciclos com peso positivo, não pode ser um caminho mais curto de $v0$ para vk.

Falta analisar os ciclos com peso nulo. Podemos remover ciclos com peso nulo de qualquer caminho porque o caminho resultante tem o mesmo peso. Conclusão: os caminhos mais curtos não têm ciclos. Um caminho sem ciclos designa-se **acíclico** e no caso dum grafo $G=(V,E)$ contém no máximo $|V|$ vértices e $|V|-1$ arcos.

5.1.5 Representação de caminhos mais curtos

Muitas vezes queremos calcular não apenas os pesos do caminho mais curto, mas também os vértices desse caminho. A representação que usamos nos caminhos mais curtos é semelhante à que usamos nas árvores BF do algoritmo BFS: para cada vértice $v \in V$ guardamos o vértice antecessor $p[v]$ ou `NULL` se não existir antecessor. Os valores $p[]$ produzidos pelos algoritmos a apresentar nesta seção definem uma árvore com os caminhos mais curtos desde s até cada um dos vértices acessíveis a partir de s. Sendo $G=(V E)$ um grafo direcionado, com pesos e com função de peso $w: E \rightarrow R$, e assumindo que G não inclui ciclos com peso negativo acessíveis a partir do vértice inicial $s \in V$, então os caminhos mais curtos são bem definidos.

Uma **árvore com os caminhos mais curtos**, com raiz no vértice s, é um sub-grafo direcionado $G'=(V',E')$, onde $V' \in V$ e $E' \in E$, tal que:

1. V' é o sub-conjunto dos vértices de G acessíveis a partir de s;
2. G' é uma árvore com raiz em s;
3. Para qualquer $v \in V'$, o único caminho de s para v contido em G' é um caminho mais curto de s para v em G.

Nem os caminhos mais curtos nem as árvores com caminhos mais curtos, são necessariamente únicos.

A figura 39 (a) mostra um grafo direcionado com pesos e duas árvores distintas contendo os caminhos mais curtos: (b) e (c).

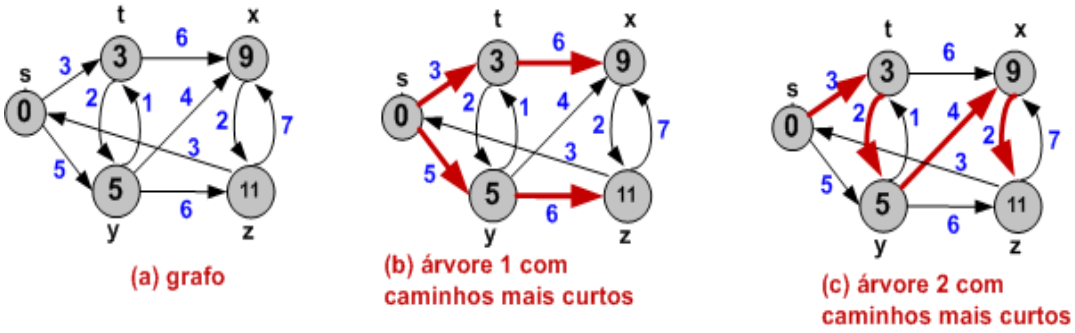


Figura 39 - Grafo com duas árvores distintas contendo os caminhos mais curtos.

Questão: Para o grafo (a), existe mais alguma árvore com caminhos mais curtos que seja distinta de (b) e (c)?

Resposta: Sim, existe mais uma árvore com caminhos mais curtos: $s \rightarrow t$, $s \rightarrow y \rightarrow x$, $s \rightarrow y \rightarrow z$.

5.1.6 Relaxamento

Os algoritmos a apresentar nesta seção utilizam a técnica de **relaxamento**. Para cada vértice $v \in V$, guardamos um atributo $d[v]$, que funciona como limite superior ou estimativa, para o peso do caminho mais curto de s para v. As estimativas do peso dos caminhos mais curtos $d[v]$ e os antecessores $p[v]$ são inicializados pela função:

```
INICIALIZAR-ORIGEM-UNICA(G, s)
1 para cada vértice v ∈ V fazer
2    d[v] ← ∞
3    p[v] ← NULL
4 fpara
5 d[s] = 0
```

Relaxar um arco *(u,v)* consiste em tentar melhorar o caminho mais curto, que termina em *v* e passa por *u*. Em caso afirmativo, diminui-se a estimativa do peso do caminho que termina em *v* *(d[v])* e atualiza-se o antecessor de *v: p[v]=u*.

O código seguinte efetua uma iteração do processo de relaxamento do arco *(u,v)*, em que *w* é a função de peso.

```
RELAX(u,v,w)
1 se d[v]>d[u]+w(u,v) então
2     d[v] ← d[u]+w(u,v)
3     p[v] ← u
4 fse
```

A figura 40 mostra dois exemplos de relaxamento dum arco *(u,v)*. Em (a) a estimativa do peso do caminho mais curto até *v* diminui e em (b) não há alteração da estimativa do peso do caminho mais curto.

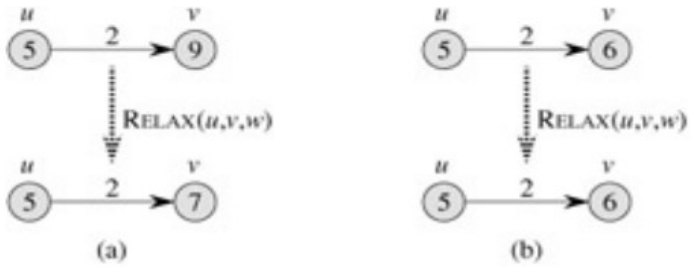


Figura 40 - Relaxamento de um arco dum grafo.

Cada algoritmo que vamos analisar chama a função `INICIALIZAR-ORIGEM-UNICA()` e depois relaxa repetidamente o peso dos arcos. O relaxamento é a única forma disponível para alterar a estimativa do peso dos caminhos mais curtos e os antecessores dos vértices. A diferença entre algoritmos reside no número de vezes que cada arco é relaxado e a ordem pela qual eles são relaxados. No algoritmo de Dijkstra e no algoritmo para caminhos mais curtos em grafos acíclicos direcionados, cada arco é relaxado apenas uma vez. No algoritmo Bellman-Ford, cada arco é relaxado várias vezes.

5.1.7 Algoritmo Bellman-Ford

O algoritmo Bellman-Ford soluciona o problema de encontrar os caminhos mais curtos, a partir dum único vértice inicial, no caso em que os pesos dos arcos podem ser negativos. Dado um grafo direcionado e com pesos *G=(V,E)*, com origem em *s* e função de peso *w:E→R*, o algoritmo devolve um valor booleano que indica se há (`FALSO`) ou não (`VERDADEIRO`) um ciclo com peso negativo acessível a partir de *s*. Se existir um ciclo deste tipo, o algoritmo indica que não existe solução. Caso contrário, o algoritmo gera os caminhos mais curtos e os seus pesos. O algoritmo usa relaxamento, para diminuir de forma progressiva a estimativa *d[v]* para o peso do caminho mais curto de *s* para qualquer vértice *v ∈ V*, até que este atinja o valor mínimo *δ(s,v)*.

```
BELLMAN-FORD(G,w,s)
1 INICIALIZAR-ORIGEM-UNICA(G,s)
2 para i = 1 até |V|-1 fazer
3    para cada arco (u,v) ∈ E fazer
4       RELAX(u, v, w)
5    fpara
6 fpara
```

```
 7 para cada arco (u,v) ∈ E fazer
 8    se d[v]>d[u]+w(u,v) então∞
 9       Devolver FALSO # indica que há ciclos com peso negativo
10    fse
11 fpara
12 Devolver VERDADEIRO  # indica que não há ciclos com peso negativo
```

Exemplo: Vamos analisar a aplicação do algoritmo Bellman-Ford a um grafo com 5 vértices (figura 41).

Através da linha 1, o algoritmo começa por inicializar os valores de d e de p para todos os vértices (figura 41 (a)). A seguir itera $|V|-1$ vezes sobre cada arco do grafo. Em cada iteração do ciclo exterior, linhas 2-6, relaxam-se todos os arcos do grafo, um em cada iteração do ciclo interior. A ordem pela qual se relaxam os arcos depende da ordem pela qual os vértices e seus vértices adjacentes estão armazenados. Assumindo que a ordem de armazenamento dos vértices segue o seu nome, ou seja $s \rightarrow t \rightarrow x \rightarrow y \rightarrow z$, então a ordem de relaxamento dos vértices é a seguinte:

$$(s,t) \rightarrow (s,y) \rightarrow (t,x) \rightarrow (t,y) \rightarrow (t,z) \rightarrow (x,t) \rightarrow (y,x) \rightarrow (y,z) \rightarrow (z,s) \rightarrow (z,x) \infty$$

Na figura 41 (b) a (e) mostra-se o estado do algoritmo após cada uma das $|V|-1=5-1=4$ iterações do ciclo exterior.

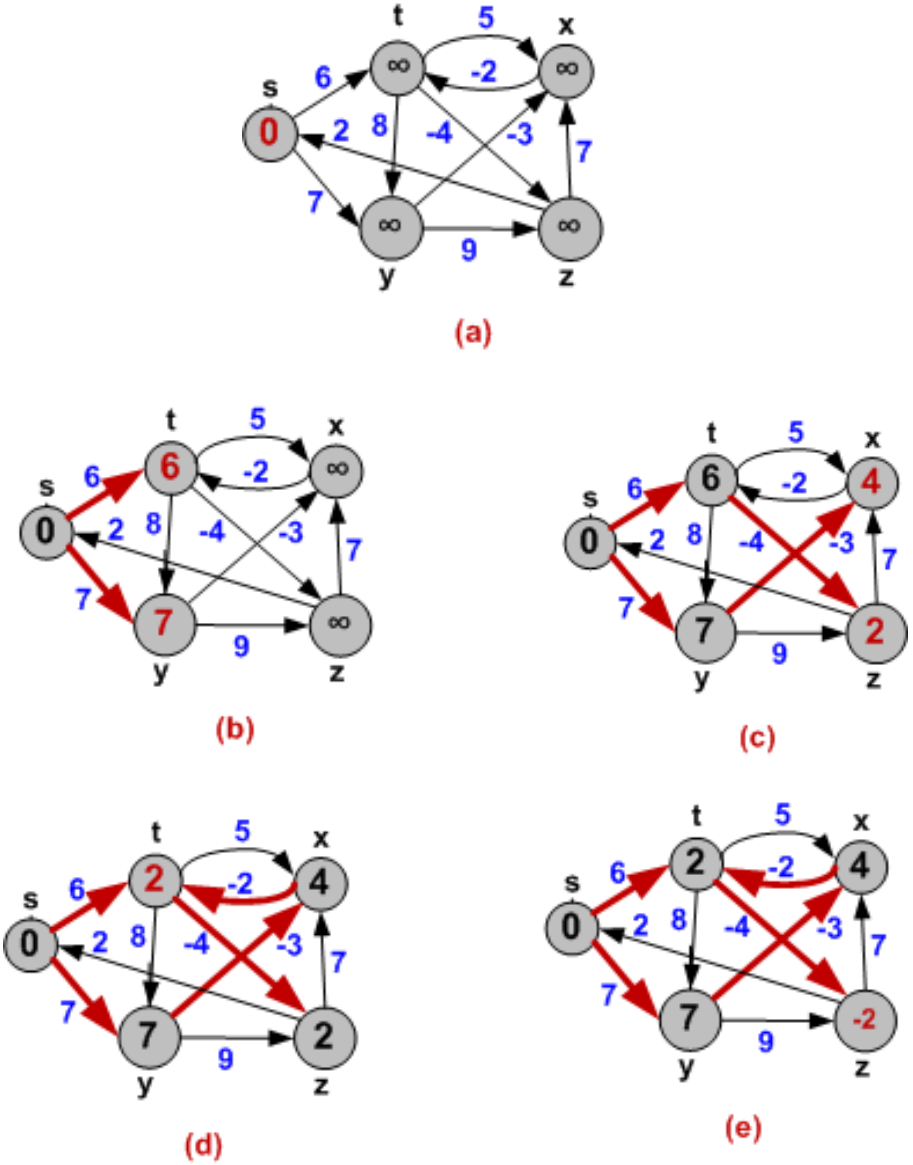


Figura 41 - Evolução da aplicação do algoritmo Bellman-Ford a um grafo.

Depois de efetuar as $|V|-1=4$ iterações anteriores, linhas 7-11, verifica-se se existem ciclos com peso negativo.

Questão: Quais os ciclos do grafo e qual o seu peso?

Resposta:

- $t \rightarrow x \rightarrow t$ Peso = 5-2= 3
- $s \rightarrow y \rightarrow z \rightarrow s$ Peso = 7+9+2 = 18
- $s \rightarrow t \rightarrow z \rightarrow s$ Peso = 6-4+2 = 4
- $t \rightarrow y \rightarrow x \rightarrow t$ Peso = 8-3-2 = 3
- $t \rightarrow z \rightarrow x \rightarrow t$ Peso = -4+7-2 = 1
- $t \rightarrow y \rightarrow z \rightarrow x \rightarrow t$ Peso = 8+9+7-2 = 22

Questão: Qual o valor devolvido pelo algoritmo?

Resposta: Como não há ciclos com peso negativo devolve `VERDADEIRO`.

Na figura 41, onde se mostra a evolução da execução do algoritmo Bellman-Ford, o valor de *d* (peso do caminho até um vértice) é mostrado dentro desse vértice. Se um arco *(u,v)* está **destacado**, então o vértice antecessor de *v* (no caminho mais curto) é *p[v]=u*. Os valores de *d* e *p*, na figura (e) são os valores finais resultantes da execução do algoritmo.

Questão: Quais os valores finais de *p[]*?

Resposta: Os valores finais de *p[]* são *p[s]* = `NULL`, *p[t]* = *s*, *p[x]* = *y*, *p[y]* = *s* e *p[z]* = *t*.

O algoritmo demora a executar um tempo $\Theta(V*E)$, dado que é o pior tempo de entre as 3 tarefas nele incluídas:

- A inicialização na linha 1 demora $\Theta(V)$;
- O ciclo das linhas 2-6 tem $|V|-1$ iterações e cada uma demora $\Theta(E)$. Logo este ciclo demora $\Theta(V*E)$;
- O ciclo das linhas 7-11 demora $\Theta(E)$.

5.1.8 Algoritmo para caminhos mais curtos em grafos acíclicos direcionados

Grafo **a**cíclico **d**irecionado (**DAG** em inglês) é um grafo direcionado sem ciclos. Ao relaxar os arcos dum DAG com pesos `G=(V,E)`, usando uma **ordenação linear** dos seus vértices, podemos calcular os caminhos mais curtos a partir dum único vértice inicial num tempo $\Theta(V+E)$. Num DAG, os caminhos mais curtos são sempre bem definidos, porque mesmo que existam arcos com peso negativo, não existem ciclos (com peso negativo).

O algoritmo começa por ordenar os vértices do DAG, obtendo-se um grafo com **topologia linear**. Se existir um caminho do vértice `u` para `v`, então `u` precede `v` na ordenação linear. Usando ordenação linear, o algoritmo faz apenas uma passagem por cada vértice. Ao processar um vértice relaxa todos os arcos que saem desse vértice. O algoritmo que calcula os caminhos mais curtos num DAG é apresentado a seguir:

```
CAMINHOS-MAIS-CURTOS-DAG(G, w, s)
1 Ordenar os vértices de G para obter uma topologia linear
2 INICIALIZAR-ORIGEM-UNICA(G, s)
3 para cada vértice u, obtido da topologia linear, fazer
4    para cada vértice v ∈ Adj[u] fazer
5       RELAX(u, v, w)
6    fpara
7 fpara
```

No algoritmo apresentado `Adj[u]` representa a lista ligada com os vértices adjacentes de `u`.

A figura 42 mostra a evolução da execução do algoritmo `CAMINHOS-MAIS-CURTOS-DAG()`. O algoritmo começa por ordenar os vértices de `G` para obter uma topologia linear e inicializa os valores de `d` e de `p` para todos os vértices (figura 42 (a)). Como se pode ver em (a), no início o valor do peso do caminho até cada um dos vértices (`d`) é infinito, exceto para a origem (`s`) que é zero. Em cada iteração do ciclo `para` das linhas 3-7, o algoritmo percorre cada um dos vértices pela ordem `r`→`s`→`t`→`x`→`y`→`z`. Para cada vértice `u` por onde o algoritmo passa, relaxam-se todos os arcos que partem de `u`, ou seja, relaxa-se cada um dos arcos `(u,v)`, em que `v` ∈`Adj[u]`. Após relaxar todos os arcos que

partem do vértice `u`, o vértice é colorido a preto na figura 42. O resultado final da execução do algoritmo é apresentado na figura 42 (g). Analisando (g), resulta que o caminho mais curto desde a origem `s` até um dado vertice `v` é constituído pelos arcos **destacados** que ligam `s` a `v`. Os caminhos mais curtos obtidos são: `s→x→y→z`, `s→t→y→z` e `s→t→z`. De `s` para `r` não existe caminho.

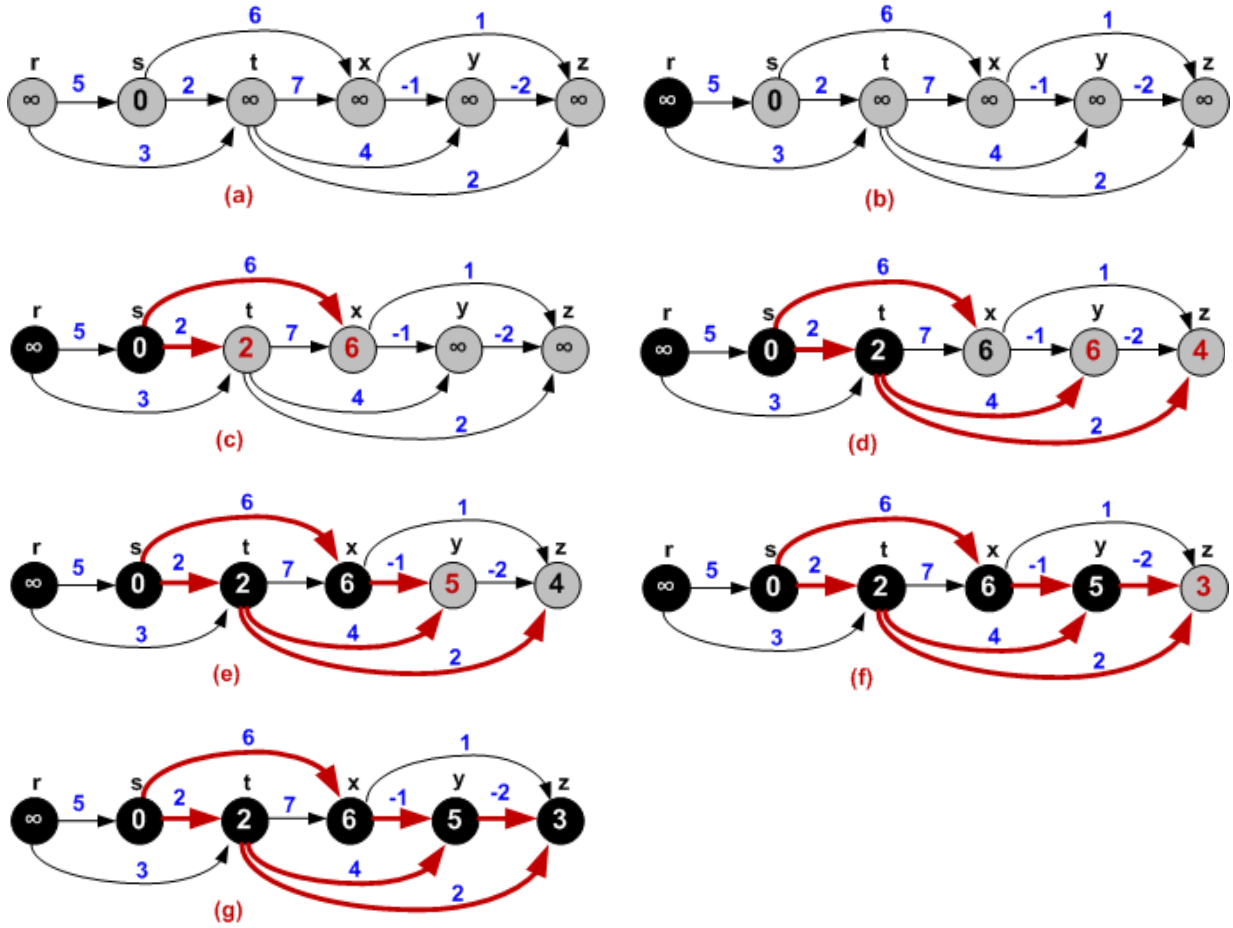


Figura 42 - Evolução da aplicação do algoritmo `CAMINHOS-MAIS-CURTOS-DAG()`.

O algoritmo `CAMINHOS-MAIS-CURTOS-DAG()` permite determinar os caminhos críticos, por exemplo em termos de duração das tarefas dum projeto, numa análise PERT (***p**rogram **e**valuation and **r**eview **t**echnique*). **PERT** é uma ferramenta utilizada para gerir projetos. Num **diagrama PERT** os **arcos** representam **tarefas** a realizar e os **pesos dos arcos** representam o **tempo** necessário para as executar. Se o arco `(u,v)` entra no vértice `v` e o arco `(v,x)` parte de `v`, então a tarefa `(u,v)` deve ser realizada antes da tarefa `(v,x)`. Um **caminho** representa uma sequência de tarefas a executar segundo uma certa ordem. Um **caminho crítico** é um dos caminhos mais longos do grafo. O peso dum caminho crítico é um limite inferior para o tempo total para executar todas as tarefas.

5.1.9 Algoritmo de Dijkstra

O algoritmo de Dijkstra resolve o problema de encontrar os caminhos mais curtos, a partir dum único vértice *s*, num grafo direcionado e com pesos *G=(V,E)*, no caso em que os pesos de todos os arcos são não negativos. Ou seja, quando $w(u,v) \geq 0$ para qualquer arco $(u,v) \in E$.

O algoritmo de Dijkstra mantém um conjunto *S* com os vértices cujos caminhos mais curtos a partir de *s* já foram calculados. O algoritmo seleciona repetidamente o vértice $u \in \{V-S\}$ com a menor estimativa para o caminho mais curto, acrescenta *u* a *S* e relaxa todos os arcos que partem de *u*.

O código seguinte, usa uma fila de prioridade-mínima *Q* contendo vértices. A chave associada a cada vértice de *Q* é o valor *d*, que representa a distância do caminho mais curto desde *s* até ao vértice em causa.

```
DIJKSTRA(G, w, s)
 1 INICIALIZAR-ORIGEM-UNICA(G, s)
 2 S ← ∅
 3 Q ← V
 4 enquanto Q ? ∅ fazer
 5    u ← EXTRAIR-MINIMO(Q)
 6    S ← S + {u}
 7    para cada vértice v ∈ Adj[u] fazer
 8       RELAX(u, v, w)
 9    fpara
10 fenquanto
```

O algoritmo relaxa os arcos como se mostra na figura 43. A linha 1 inicializa a distância à origem *d* e o antecessor *p*. A linha 2 inicializa o conjunto *S* como sendo um conjunto vazio. A linha 3 inicializa a fila de prioridade-mínima *Q* com todos os vértices de *V*. Em cada iteração do ciclo `enquanto`, linhas 4-10, um vértice *u* é retirado de *Q* e adicionado ao conjunto *S*. Na primeira iteração *u=s*. O vértice *u* tem sempre a menor estimativa para o caminho mais curto, de entre todos os vértices contidos em *{V-S}*. As linhas 7-9 relaxam cada arco *(u,v)* que parte de *u*, e atualizam a estimativa *d[v]* e o antecessor *p[v]* se o caminho mais curto até *v* pode ser melhorado ao passar por *u*.

Os vértices nunca são inseridos em *Q* após a linha 3. Cada vértice é retirado de *Q* e adicionado a *S* só uma vez. Deste modo, o ciclo `para` das linhas 7-9 é executado *|V|* vezes.

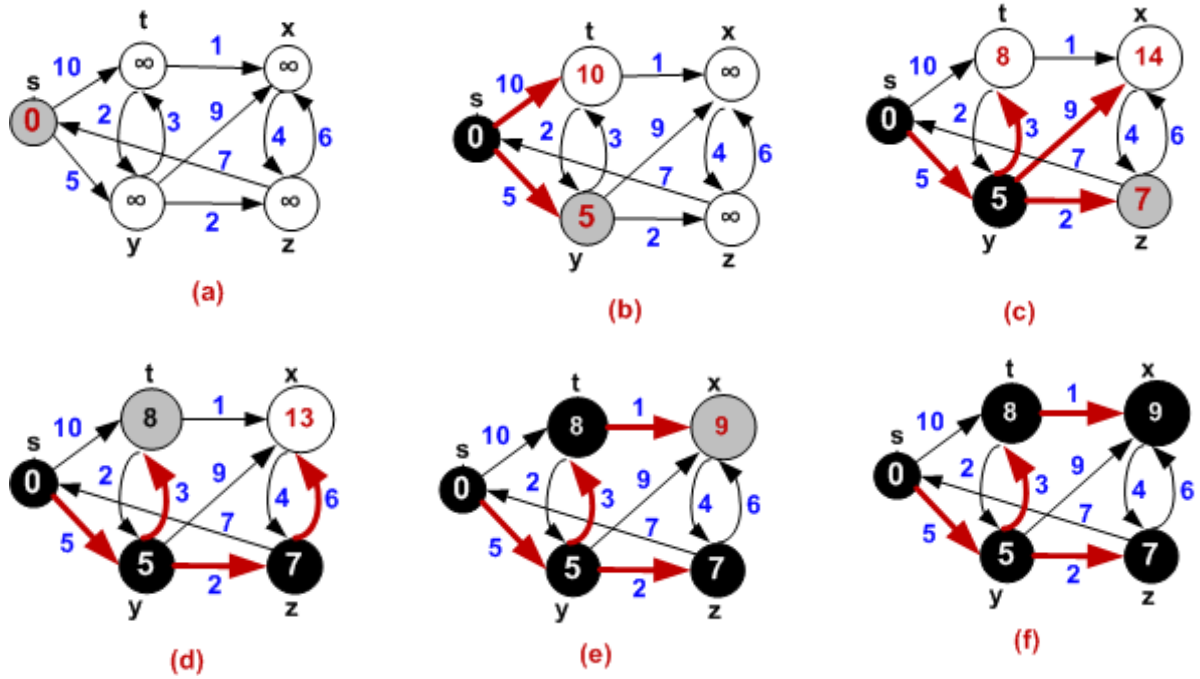


Figura 43 - Evolução da aplicação do algoritmo de Dijkstra.

Vamos ver agora vomo se processa a execução do algoritmo de Dijkstra. No grafo da figura 43, a origem *s* é o vértice mais à esquerda. As estimativas do caminho mais curto são mostradas dentro dos vértices e os arcos destacados indicam qual é o vértice antecessor de cada vértice. Vértices a `preto` já estão no conjunto *S* e os vértices a `branco` estão na fila de prioridade mínima *Q={V-S}*.

(a) é a situação antes da primeira iteração do ciclo `enquanto` das linhas 4-10. O vértice `cinzento` tem o valor mínimo de *d* e é escolhido como vértice *u* na linha 5.

(b) a (f) representam o estado do algoritmo depois de cada iteração sucessiva do ciclo `enquanto`. O vértice `cinzento` de cada figura é escolhido como vértice *u* na linha 5 da próxima iteração.

Os valores de *p* e *d* que se podem extrair da figura (f) são os valores finais da aplicação do algoritmo.

O tempo de execução do algoritmo depende da forma como a fila de prioridade mínima é implementada: um a*rray,* uma *heap* mínima binária ou uma *heap* de Fibonacci.

Bibliografia

[1] *The C Programming Language*. Brian Kernighan and Dennis Ritchie. 2nd Edition, Prentice-Hall, 1988. ISBN: 9780131103627.

[2] *Introduction to Algorithms*. Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. 2nd Edition, The MIT Press, 2001.

[3] *Algorithmics: The Spirit of Computing*. David Harel. 3rd Edition, Addison-Wesley, 2004.

[4] *An Introduction to GCC - for the GNU Compilers gcc and g++*. Brian Gough, 2004.

[5] *Understanding The Linux Virtual Memory Manager*. Mel Gorman, 2004.

[6] *Secure Coding in C and C++*. Robert C. Seacord, Addison-Wesley, 2006.

[7] www.gnu.org/software/libc/manual/

[8] crasseux.com/books/ctutorial/Libraries.html

[9] randu.org/tutorials/c/libraries.php

[10] www.techytalk.info/c-cplusplus-library-programming-on-linux-part-one-static-libraries/

[11] www.yolinux.com/TUTORIALS/LibraryArchives-StaticAndDynamic.html

[12] www.theasciicode.com.ar