Métodos de Programação II

Departamento de Matemática
Faculdade de Ciências e Tecnologia
Universidade de Coimbra

Pedro Quaresma

2024/10/18 (v1777)

Bibliografia Principal

Metodologia Wirth, Niklaus. 1989. *Algoritmos e Estruturas de Dados*. Rio de Janeiro: Prentice-Hall do Brasil. — `68P/WIR.Alg`

C Kernighan, Brian, & Ritchie, Dennis. 1988. *The C Programming Language*. 2nd edn. Prentice Hall. — `68N/KER.C`

C de Sá, Joaquim P. Marques. 2004. *Fundamentos de Programação Usando C*. FCA, Lisboa, Portugal. — `68N/SA`

Conteúdo

1 Introdução **11**

2 Estruturas Dinâmicas **15**
2.1 Tipo Ponteiro 15
2.2 Estruturas de Dados Dinâmicas 18
2.2.1 Estruturas Dinâmicas na Dimensão 19
2.2.2 Exemplo: Pilha de Caracteres 25

3 Estruturas Lineares **27**
3.1 T.D.A. Pilhas 27
3.2 T.D.A. Filas 33
3.3 T.D.A. Listas 38

4 Estruturas Não Lineares **43**
4.1 T.D.A. Árvores 43
4.1.1 Travessias de Árvores Binárias 45
4.1.2 Árvores de Pesquisa 46
4.1.3 Árvores Binárias Balanceadas 47
4.2 T.D.A. Grafos 48
4.2.1 Formas de Representação de Grafos 50
4.2.2 Percursos em Grafos 51
4.2.3 Caminho Mais Curto 52

Referências **57**

A Construção de um Executável **59**
A.1 Compilador de C 59
A.1.1 Pré-processamento 60
A.1.2 Opções de Compilação 62
A.1.3 Utilizar uma Biblioteca 63
A.2 Makefile 64
A.2.1 O Programa `make` 65
A.2.2 O Ficheiro `Makefile` 65
A.3 Comandos para a Consola 68

B Ambientes Integrados de Desenvolvimento **69**
B.1 Editor de Textos Dedicado 69
B.2 Compilação & Makefiles 70
B.3 Depuração de Erros 70
B.3.1 Documentação 70
B.4 Alguns IDEs para o *C* 70

C Folhas práticas **73**
C.1 Comandos para a Consola 73
C.2 Projectos/Makefile 73
C.3 Gestão Dinâmica de Memória 74
C.4 Matrizes Diagonais 75
C.5 Estruturas Lineares – Listas 76
C.6 Tipos Abstractos de Dados 77
C.7 Tipos Abstractos de Dados Pilha 79
C.8 Tipos Abstractos de Dados Filas 79
C.9 Tipos Abstractos de Dados Listas 81
C.10 Tipos Abstractos de Dados Árvore Binária 83
C.11 Tipos Abstractos de Dados Grafo 87
C.12 Exercícios Diversos 88

Lista de Figuras

2.1 Variáveis Estáticas vs. Variáveis Dinâmicas 16
2.2 Mapa de Estradas (grafo) 19
2.3 Estruturas Dinâmicas na Dimensão 20
2.4 Árvores Binárias — Terminologia 23
2.5 Grafos — Terminologia 24

4.1 Árvores Binárias — Terminologia 43
4.2 Árvores Binárias — Exemplo 44
4.3 Árvores Binárias de Pesquisa 46
4.4 Árvores Binárias de Pesquisa — Não Balanceada 47
4.5 Árvores Binárias de Pesquisa — Balanceada 48
4.6 Grafo Não Dirigido 48
4.7 Grafo Dirigido, não Conexo 49
4.8 Grafos — Mapa de Estradas 54

C.1 Implementação de Estruturas Lineares 77
C.2 Árvore Binária — Percurso em Largura 86
C.3 Estrutura Hierárquica — Árvore Genérica 86

Lista de Tabelas

4.1 Grafos — Caminho mais curto 55

Lista de Algoritmos

3.1 Pilha de Inteiros (especificação) 29
3.2 Pilha de Inteiros (implementação) 30
3.3 Calculadora Polaca Inversa 31
3.4 Fila de Inteiros (especificação) 35
3.5 Fila de Inteiros (implementação) 36
4.1 Algoritmo Busca em Amplitude 51
4.2 Algoritmo Busca em Profundidade 52
4.3 Algoritmo de Dijsktra 53

Capítulo 1

Introdução

A linguagem *C* não permite a criação de novos tipos de dados. É no entanto possível usar as estruturas de dados pré-definidas para novas estruturas de dados (que não exactamente tipos).

Utilizando as estruturas: registos (`struct`), vectores e ponteiros é possível definir novas estruturas. Por outro lado, usando a metodologia de programação descendente, podemos associar as estas novas estruturas as operações necessárias para a definição de um novo tipo.

Os Tipos de Dados Abstractos (TDA, ou ADT em Inglês), são então a metodologia usada nas linguagens de programação para a especificação e implementação de novos tipos de dados.

Temos então que, para um novo tipo de dados, temos de definir como criar os novos elementos e, para esses novos elementos, as operações internas sobre eles.

O conjunto de operações deve ser completo, isto é devemos poder executar todas as operações necessárias para operacionalizar o novo tipo, e deve ser, tendencialmente, minimal, isto é, só se devem implementar as operações estritamente necessárias.

Por exemplo, suponhamos que queremos criar um novo tipo de dados, *pares de inteiros*, contendo as operações de soma e multiplicação.

Temos então: definição da nova estrutura; definição do conjunto completo, minimal, de operações internas.

$$\text{ParesInt} = (\mathbb{Z} \times \mathbb{Z}, \{\texttt{constroi}, \texttt{prim}, \texttt{seg}, 0, 1, +, \times\})$$

Estrutura de dados os novos elementos podem ser então definidos usando as estruturas pré-definidas, isto é, vectores ou registos ou ponteiros, de elementos de tipos previamente definidos.

Neste exemplo vai-se usar um registo:

```
// ParesInt = (int,int)
typedef struct pares {
    int prim,seg;
} ParesInt;
```

Operações internas além das operações internas que operacionalizem o novo tipo é necessário também ter as operações que permitem construir e desconstruir os novos elementos:

```
/*
 * Dados dois inteiros contrói o par correspondente
 * --> a,b : int
 * <-- r = (a,b) : ParesInt
 */
ParesInt controiPar(int, int);

/*
 * Constrói o par (0,0)
 * -->
 * <-- r = (0,0) : ParesInt
 */
ParesInt parZero();

/*
 * Constrói o par (1,1)
 * -->
 * <-- r = (1,1) : ParesInt
 */
ParesInt parUm();

/*
 * Obtém o primeiro elemento de um par
 * --> p = (a,b) : ParesInt
 * <-- a : int
 */
int primeiroDePar(ParesInt);

/*
 * Obtém o segundo elemento de um par
 * --> p = (a,b) : ParesInt
 * <-- b : int
 */
int segundoDePar(ParesInt);
```

Depois destas operações «constructivas» é necessário definir as outras operações. Neste exemplo vamos considerar que definimos operações de adição e multiplicação em pares:

```
/*
 * Soma de Pares de Inteiros: (a,b)+(c,d) = (a+c,b+d)
 * --> p,q : ParesInt
 * <-- r = p+q : ParesInt
 */
ParesInt somaPares(ParesInt, ParesInt)

/*
 * Multiplicação de Pares de Inteiros: (a,b)*(c,d) = (a*c,b*d)
 * --> p,q : ParesInt
 * <-- r = p*q : ParesInt
 */
ParesInt multiplicacao(ParesInt, ParesInt);
```

Vamos considerar que estas operações formam um conjunto minimal de operações. Por

razões de conveniência pode ser interessante implementar outras operações, por exemplo os operadores relacionais ($\{=, \neq, <, >, \geq, \leq\}$), assim como a leitura e/ou escrita de elementos do novo tipo.

A definição da nova estrutura assim como a declaração das funções seriam agrupadas num ficheiro `pares.h`, a implementação das funções seriam agrupadas num ficheiro `pares.c`.

Sempre que o novo tipo fosse necessário num dado programa ter-se-ia de incorporar a instrução de inclusão correspondente.

```
#include "pares.h"
```

No momento da compilação será necessário incluir também o ficheiro `pares.o`, na instrução de compilação. Considerando um dado ficheiro `Makefile`, temos o seguinte objectivo de compilação:

```
testaPares: testaPares.c pares.o
    ${CC} -o $@ $@.c pares.o
```

O programa `make` (programa que lê a `Makefile` e executa os comandos correspondentes) tem pré-definida a opção de compilar, automaticamente, o ficheiro em *C* (`pares.c`) de forma a obter o ficheiro em *linguagem máquina* (`pares.o`), para o mesmo poder ser integrado na compilação do programa principal.

A construção de um novo tipo de dados desta forma, como um tipo abstracto de dados, permite isolar esta nova definição do restante código. A reutilização do novo tipo de dados é possível e fácil.

Tentando resumir, quais são as possibilidade e virtualidades da definição de novos TDA?

Ao definir, deste modo, um novo tipo de dados podemos expandir os tipos pré-definidos do *C*, sendo que a sua utilização é bastante semelhante aos tipos pré-definidos.

Ao definir, deste modo, um novo tipo de dados, sendo que todas as manipulações são efectuadas através das operações definidas no TDA, permite:

- re-utilizar o novo tipo sempre que necessário (sem necessitar de saber os detalhes da implementação);
- re-implementar as operações, eventualmente optimizando o código, sem que o programa de chamada seja afectado.

Em *C* o carácter abstracto do novo tipo é metodológico, isto é o utilizador só deve aceder aos novos elementos através das operações definidas no novo tipo, no entanto pode-o fazer directamente a qualquer momento. Nas linguagens ditas *Orientadas aos Objectos*, a restrição é imposta pelos mecanismos da linguagem, tornando a definição de um TDA mais impositiva.

Capítulo 2

Estruturas Dinâmicas

Programas = Algoritmos + Estruturas de Dados

Niklaus Wirth

Do ponto de vista da programação procedimental um computador é uma máquina de estados. De um estado inicial, isto é, os dados de entrada, guardados num conjunto de variáveis que em conjunto definem o estado do programa, pretende-se atingir um dado estado final, definidor do resultado que se pretende obter com o programa.

Um programa é então uma sequência linear de transições de estado. O algoritmo define essa sequência, as estruturas de dados determinam o estado, o qual vai variando temporalmente à medida que se processam as diferentes instruções definidas no algoritmo.

A modelização da realidade que se quer automatizar é feita através da sua concretização em estruturas de dados, pré-definidas na linguagem, ou implementadas na forma de TDA.

2.1 Tipo Ponteiro

Além dos tipos pré-definidos "usuais" como sejam os inteiros (`int`) e os reais (`float, double`) o *C* disponibiliza o tipo ponteiro.

Tipo Ponteiro (`<tipo>*`,{`+,-,*,/,%`})—ponteiros para um dado tipo de dados. Neste caso não temos verdadeiramente um tipo de dados mas sim referências, isto é, o explicitar da ligação entre os identificadores (nomes das variáveis de um dado tipo) e os seus valores (células de memória aonde são guardados os valores). Os ponteiros não são mais do que valores naturais, $0 \leq n \leq maxRAM$, com $maxRAM$ o valor máximo da memória RAM que um dados sistema operativo suporta. Os referidos valores inteiros, não nulos, identificam as células de memória aonde os valores das variáveis são guardados (ver Figura 2.1)

- valores inteiros entre 0 e o valor máximo que o sistema operativo suporta (Linux 64bits, 128TiB ($= 2^{47}$).
- todas as operações com inteiros. No entanto dado se tratar de ponteiros, isto é referências a células de memória, a sua manipulação directa como valores inteiros deve ser feita com extremo cuidado.

Uma das possibilidade que este tipo de dados "não usual" abre é o da definição de estruturas de dados dinâmicas, tanto na dimensão como na forma, como se verá nos capítulos 3 e 4.

As variáveis estáticas (sem ponteiros) e as variáveis dinâmicas têm um comportamento diferente na forma como associam estas duas entidades, nomes e posições de memória. No primeiro caso as referências são geridas automaticamente pelo programa, no segundo essa é já parte das responsabilidades do programador.

Vejamos uma situação em concreto (ver Figura 2.1): a declaração de uma variável estática do tipo inteiro, `x`, e a sua posterior inicialização; e a declaração de uma variável dinâmica, um ponteiro para um inteiro, `z`, a inicialização do espaço e a posterior inicialização do valor apontado.

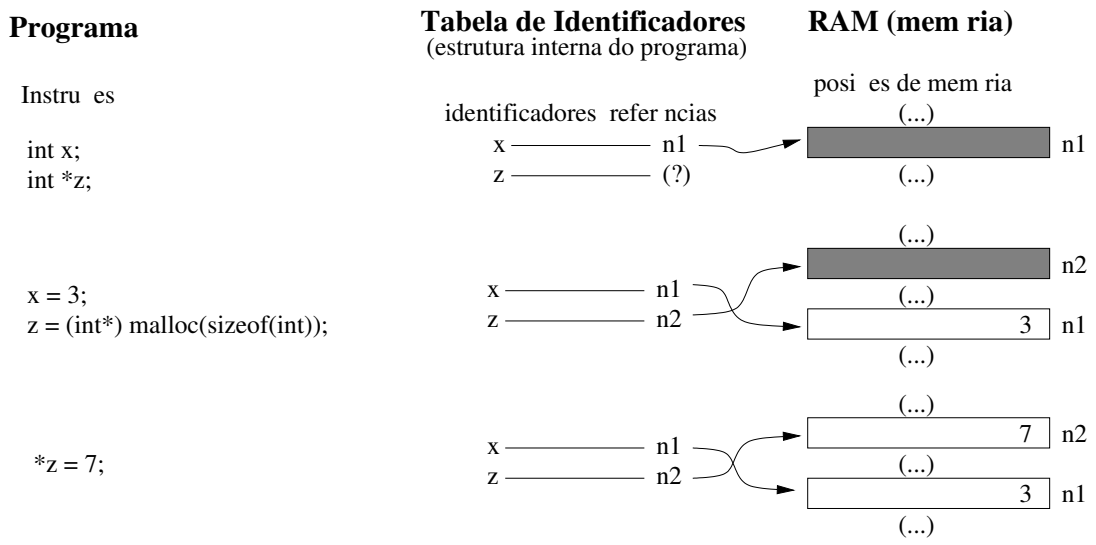


Figura 2.1: Variáveis Estáticas vs. Variáveis Dinâmicas

Aquando da declaração de um variável estática temos as seguintes acções (feitas automáticamente pelo compilador):

1. guardar o nome na tabela de identificadores;
2. reservar a memória necessária para guardar um valor do tipo associado ao identificador;
3. inicializar a referência associada (ponteiro, número natural referente a uma posição de memória) com o valor correspondente à posição de memória que foi anteriormente reservada.

o valor associado à variável começa por ser indefinido.

Aquando da declaração de uma variável dinâmica temos a seguinte acção:

1. guardar o nome na tabela de identificadores.

A referência associada a este nome começa por ser indefinida, qualquer tentativa de acesso ao valor (ainda inexistente) associado é um erro de acesso à memória.

Para uma variável dinâmica é necessário fazer uma reserva de memória de forma explícita (através da função `malloc`) e só depois é que é possível associar-lhe um valor.

Para uma variável do tipo estático é possível aceder à sua referência (ponteiro) de forma explícita utilizando o operador de referenciação "`&`" (dá-nos a referência):

declaração	identificador	referência	valor (referenciado)
<tipo> `x`	`x`	`&x`	`x`

Para uma variável do tipo dinâmico é possível aceder ao valor que lhe está associado através do operador referenciador «`*`» (dá-nos o valor referenciado):

declaração	identificador	valor (referenciado)
<tipo> `*x`	`x`	`*x`

Retomando o exemplo apresentado na figura 2.1 vejamos um exemplo da declaração, afectação e atribuição de valor a uma variável do tipo ponteiro.

Declaração começa-se por declarar uma variável do tipo ponteiro para um dado tipo específico, por exemplo ponteiro para um inteiro;

```
int *z;
```

neste momento ainda não é possível atribuir um valor (inteiro) a esta variável. Só o espaço para o ponteiro é que foi criado e nem sequer está inicializado. O tentar aceder ao valor de uma variável do tipo ponteiro nesta situação é considerado uma situação de erro (ver figura 2.1, primeira parte). Neste momento ainda nem sequer é possível saber quantos valores vão estar associados ao identificador `z`, tanto pode ser um só inteiro, como um vector de inteiros.

Afectação ou seja é necessário criar explicitamente o espaço de memória (para 1 inteiro) e inicializar o ponteiro para que este aponte para essa posição de memória.

```
z = (int *) malloc (1*sizeof(int));
```

a posição de memória em concreto não é inicializada (figura 2.1, parte do meio). Mais à frente falaremos com mais detalhe da funções de gestão de memória (`malloc` e `free`).

Atribuição Já tendo o espaço de memória assim como o ponteiro para essa posição, a atribuição e/ou utilização de valores é feita de forma (quase) normal, por exemplo:

```
*z = 7;    // atribuição de um valor, no espaço apontado por z
```

Como '`z`' é uma variável do tipo ponteiro o acesso ao valor que lhe está associado é feito através do operador `*`<*variável*> (figura 2.1, última parte).

Como é fácil de perceber a importância das variáveis do tipo ponteiro não está na sua utilização como alternativa às variáveis dos tipos simples. A importância dos ponteiros em *C* prende-se com a necessidade de fazer a passagem, por referência, de valores entre módulos distintos (funções), assim como com as estruturas de dados tanto as estáticas como as dinâmicas.

Tabelas Dinâmicas As tabelas também podem ser declaradas de forma dinâmica, por exemplo, `int *v`, pode ser vista como a declaração de uma tabela de elementos do tipo inteiro. À semelhança da definição, `int v[100]`.

A vantagem deste tipo de definição é que a estrutura assim criada pode depois ter uma dimensão ajustada às necessidades, em contraponto com o outro tipo de declaração que fixa a dimensão da tabela. A desvantagem é que obriga à gestão da afectação da memória por parte do utilizador.

2.2 Estruturas de Dados Dinâmicas

As estruturas de dados estáticas, por exemplo, `int v[100]`, levantam duas questões:

- têm uma dimensão (número de elementos) fixa;
- têm uma forma fixa.

Dimensão (número de elementos) Fixa Ao se definir uma tabela com um número de elementos fixo isso leva a que, aquando da utilização do programa, esse número máximo de elementos não pode ser ultrapassado, isto é se a utilização do programa revelar que a estimativa para a dimensão da estrutura está errada, a única forma de corrigir o problema é alterar a dimensão da estrutura e recompilar o programa. Por outro lado se a estimativa se revelar muito exagerada há um desperdício de memória, que é tento maior quantos menos elementos se utilizarem.

Por exemplo: numa dada empresa pretende-se guardar e manipular a informação referente aos seus empregados. Em vez de colocar a informação referente aos empregados em estruturas não relacionas entre si, é vantajoso agrupá-las todas numa só estrutura. Temos então a definição de um registo (`struct`) do tipo:

```
typedef struct empregado {
 char nome[60]; // nome do empregado
 int numero; // número de empregado
 int cc; // número de identificação (CC ou Passaporte ou ...)
 int tipoId; // tipo de identificação (CC=1, Passaporte=2, ...)
 int telefone;
} Empregado;
```

Esta declaração cria um novo tipo de dados, o tipo ≪`Empregado`≫. Caso se pretenda criar uma variável deste novo tipo podemos fazê-lo utilizando a sintaxe habitual:

```
Empregado ep1,ep2;
```

Como podemos ver por este exemplo, um registo pode conter uma (ou mais) tabela(s). O contrário também é verdade. Por exemplo:

```
Empregado listaEmpregado[100];
```

podia ser a estrutura aonde se guarda a informação referente à lista de empregados da empresa.

Esta implementação estática fixa o número máximo de empregados dos quais se pode guardar a informação, a empresa estará limitada a 100 empregados. Com esta definição não é possível guardar a informação para 101, ou mais, empregados. Por outro lado se se definir a

dimensão do vector de forma exagerada, e a empresa for pequena, há um possível desperdício de memória.

Por outro lado a definição do membro nome é uma componente com uma dimensão (fixa) apropriada para nomes portugueses (em geral muito grandes), já é totalmente exagerada para uma companhia que, por exemplo, tivesse a maior parte dos trabalhadores de nacionalidade inglesa (nomes em geral pequenos).

Este problema, número fixo de elementos de uma dada estrutura, pode ser resolvido utilizando uma declaração dinâmica para a dimensão da estrutura, definindo um ponteiro para a estrutura, por exemplo:

```
Empregado* listaEp;
```

A gestão da memória (e o acesso aos elementos) passa a ser responsabilidade do programador, aumentando um pouco a complexidade do programa.

Como contraponto, a dimensão do vector pode-se ajustar às necessidades da empresa. Não há limite máximo para o número de empregados da empresa (enquanto houver memória disponível). Por outro lado também não haverá desperdício da memória, a dimensão do vector vai-se ajustar ao número de empregados da empresa.

Forma Fixa Uma outra questão tem a ver com a implementação de estruturas cuja forma e dimensão são muito variáveis.

Por exemplo, como representar uma rede rodoviária, com as cidades e as estradas que as ligam? Uma resposta a esta questão é dada por uma estrutura dinâmica, tanto na forma como na dimensão, os grafos.

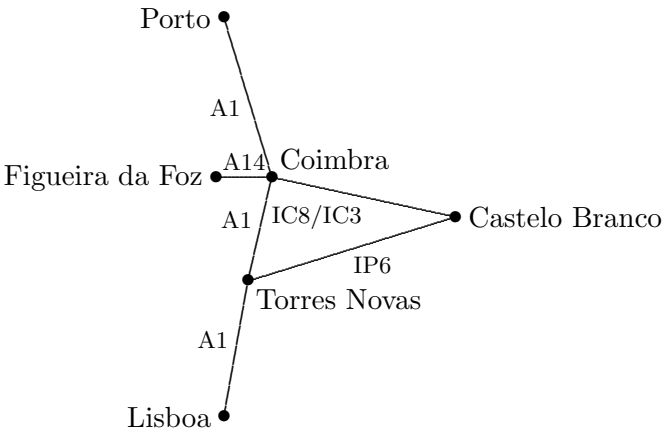


Figura 2.2: Mapa de Estradas (grafo)

Embora seja possível representar esta estrutura utilizando estruturas estáticas a utilização de estruturas dinâmicas capazes de se adaptarem tanto na forma como no número de elementos a guardar é muito vantajosa.

2.2.1 Estruturas Dinâmicas na Dimensão

Podemos definir tabelas com um número de elementos não explicitado à partida. As seguintes declarações são equivalentes:

```
int* tabela1;
int tabela2[];
```

ambas definem ponteiros para inteiros.

Em ambos os casos a possibilidade de associar um série de posições de memória contiguas ao ponteiro assim definido permite-nos criar uma tabela de elementos do tipo pretendido (neste caso o tipo `int`) em que a sua dimensão esteja apropriada às necessidade de utilização do programa.

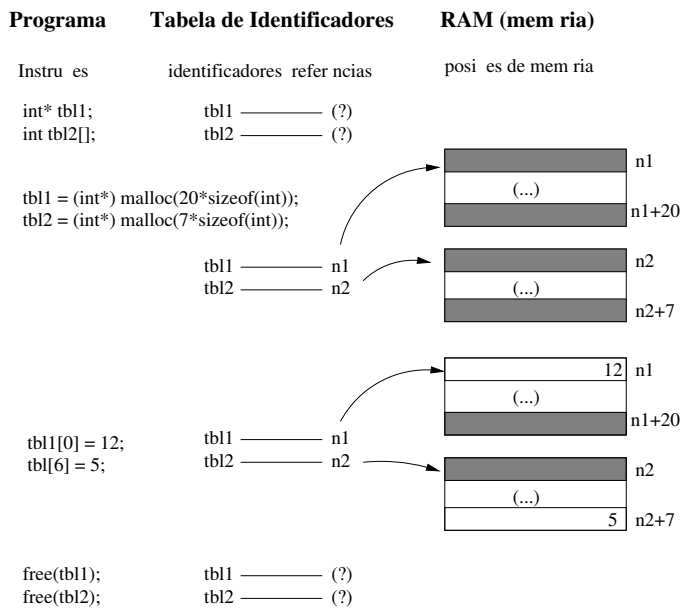


Figura 2.3: Estruturas Dinâmicas na Dimensão

Gestão de Memória a gestão de memória, a sua afectação e libertação está a cargo de duas funções específicas da linguagem *C*, as funções da biblioteca padrão `malloc` e `free`.

De forma a afectar memória usa-se a função `malloc`.

```
void *malloc(size_t tamanho)
```

A função `malloc` devolve um ponteiro para um objecto de dimensão (número de elementos), `tamanho`, ou o ponteiro nulo, `NULL`, caso o pedido de memória não possa ser satisfeito. O espaço (apontado) não é inicializado.

Para se poder saber o tamanho (número de células de memória) necessário aos valores que se quer guardar recorre-se à função da biblioteca padrão, `sizeof`. Esta função recebe como argumento um identificador de tipo e devolve o espaço que um valor desse tipo ocupa. Para guardar mais do que um valor, basta multiplicar o valor obtido pelo número de elementos que se pretendem guardar.

A função `malloc` devolve um ponteiro genérico, é necessário modificar esse ponteiro para o tipo específico de ponteiro que se pretende. Recorre-se então ao mecanismo de forçar o tipo de um elemento (*type cast*).

Temos então que uma definição como a seguinte forma:

```
v = (int *) malloc(20*sizeof(int));
```

iria criar o espaço para 20 valores do tipo inteiro sendo que a variável `v`, do tipo ponteiro para inteiros, estaria associada à referência para o primeiro elemento desse espaço.

Além da função `malloc` a afectação de memória conta com outra função, o `realloc`. Se a primeira se refere a ≪**m**emory **alloc**ation≫, isto é, afectação de memória, a segunda refere-se a ≪**realloc**ation≫, isto é, re-afectação da memória.

A função `realloc` usa-se sempre que seja necessário modificar a afectação de memória inicial, ou porque se sobre-dimensionou o espaço e quer-se diminuir o mesmo, ou porque se sub-dimensionou, e é necessário afectar mais memória:

```
void *realloc(void *p, size_t tamanho)
```

Temos então que, dado uma variável do tipo ponteiro já a apontar para um dado espaço de memória, é possível modificar o mesmo com a instrução:

```
v = (int *) realloc(v, 100*sizeof(int);
```

esta, modificava o espaço apontado por `v` de 20 para 100 posições de memória, as 20 primeiras posições ficam inalteradas, as restantes ficam por inicializar.

A função `realloc` é muito ineficiente dado que a sua implementação implica o criar de um espaço novo, o copiar dos elementos do espaço antigo para o novo e, finalmente, o libertar do espaço antigo. A sua utilização deve ser evitada sempre que possível.

Assim como a afectação, a libertação deste espaço de memória é também da responsabilidade do programador, o qual deve ter o cuidado de proceder à libertação do espaço logo que o mesmo deixe de ser necessário.

Para este efeito existe, na biblioteca padrão, a função `free` cujo formato é o seguinte:

```
void free(void *ponteiro)
```

A função `free` liberta o espaço apontado pelo seu argumento (ponteiro); não faz nada caso o ponteiro seja o ponteiro `NULL`. O ponteiro deve apontar para um espaço que tenha sido criado anteriormente pela função `malloc` (mesmo que depois modificado pela função `realloc`).

Em conclusão, no caso das tabelas podemos optar por uma declaração estática, ou por uma declaração dinâmica. Por exemplo:

```
char estatica[20];

char dinamica[];
dinamica = (char*) malloc(20*sizeof(char));
.
.
.
free(dinamica);
```

tendo afectado o mesmo número de posições de memória para as duas variáveis estas são equivalentes do ponto de vista de utilização (excepção feita para a possibilidade/necessidade de libertar o espaço afectado no segundo caso). No entanto a segunda declaração dá-nos a liberdade de só afectar o espaço no momento em que ele é necessário e (eventualmente) já com o conhecimento do espaço exacto que é necessário.

No primeiro caso o espaço a afectar tem de ser estimado aquando da programação, no segundo caso o espaço a afectar pode ser determinado aquando da utilização, ajustando-o às necessidades.

Estruturas Dinâmicas na Forma (e Dimensão) Além das vantagens que advêm do ajustar a dimensão às necessidade, a gestão dinâmica da memória dada pela utilização dos ponteiros permite também a definição de estruturas dinâmicas, não só na sua dimensão mas também na sua forma.

Como exemplos de estruturas dinâmicas muito usadas em programação temos:

Estruturas Lineares Pilhas, Filas e Listas.

Estruturas Não Lineares Árvores (binárias) e Grafos.

Nos capítulos seguintes iremos tratar de cada uma destas estruturas individualmente.

Estruturas Lineares Sequências lineares de elementos aliadas a uma disciplna de acesso a esses mesmos elementos.

Pilhas listas com acesso por um só ponto, o topo. Implementam a disciplina *Last In First Out (LIFO)*, o último a entrar é o primeiro a sair. Entre muitas utilizações temos o cálculo de expressões em notação Polaca inversa.

$$\text{Pilha} = (\{\text{pilhaVazia}, \text{elemento}:\text{Pilha}\}, \{\text{cria, push, pop, top, vazia?}\})$$

Filas listas com acesso apenas pelas extremidades, a entrada e a saída. Implementam a disciplina *First In First Out (FIFO)*, o primeiro a entrar é o primeiro a sair. A simulação de filas de espera, sejam elas caixas num dado hipermercado ou semáforos num dado cruzamento, podem ser modelizadas através deste tipo de estrutura.

$$\text{Fila} = (\{\text{filaVazia,elemento}:\text{Fila}\}, \{\text{cria, insere, retira, topo, vazia?}\})$$

Listas listas genéricas com acesso livre a uma qualquer posição. Podem ser usadas para modelizar todo o tipo de situações em que se pretende uma lista de elementos todos do mesmo tipo mas em que o número de elementos a considerar varia muito durante o correr do programa.

$$\text{Lista} = (\{\text{listaVazia}, \text{elemento}:\text{lista}\}, \{\text{cria, insereN, retiraN, veN, vazia?}\})$$

Estruturas Não Lineares

Árvores modeliza uma qualquer situação em que a informação siga uma estrutura hierárquica. Por exemplo, árvores genealógicas.

Árvores Genéricas A representação de uma árvores genérica é feita por uma «raiz» e por n «ramos». A dificuldade na sua representação passa pelo número indeterminado de ramos. Veja-se o caso das árvores genealógicas, dado uma raiz (os pais da família que se pretende representar), quantos ramos (os filhos) devemos considerar?

Felizmente é fácil de verificar que é possível representar qualquer tipo de árvore como uma árvore binária (uma raiz e dois ramos), sendo assim só em casos excepcionais se vai definir um tipo de árvore que não seja uma árvore binária.

Árvores Binárias uma raiz e duas sub-árvores. Este tipo de estrutura de dados, definida inductivamente, é usada para modelizar todas as situações em que se pretende representar uma estrutura hierárquica.

$$\mathrm{AB} = (\{\mathrm{Abvazia}, \mathrm{ABesq}:\mathrm{raiz}:\mathrm{ABdir}\}, \{\mathrm{criaVazia}, \mathrm{criaAB}, \mathrm{procuraElemento}, \mathrm{insereElemento}, \mathrm{retiraElemento}, \mathrm{percorreAB}, \mathrm{vazia?}\})$$

No caso deste tipo de dados há variantes tanto na forma de definir assim como nas operações de base (Main & Savitch, 1997; Semgupta & Korobkin, 1994; Weiss, 1997; Aho *et al.*, 1983).

No que se segue subentende-se árvore binária, sempre que se falar de árvore.

Terminologia usualmente usada para o tipo de dados árvore binária (ver Figura 2.4).

raiz: nó inicial da árvore.

folhas: nós terminais (sem sub-árvores).

ramos: uma das ramificações desde a raiz até a uma dada folha.

altura: número de nós do ramo com o maior número de nós.

nível: o conjunto de nós cuja distância (número de ramificações intermédias) à raiz é igual.

árvore completa: uma árvore contendo todos os nós (ramificações) possíveis, desde a raiz até às folhas.

árvore de pesquisa: uma árvore cujos nós estão organizados de acordo com uma dada função de ordenação.

árvore balanceada: uma árvore cuja altura entre todo e qualquer par de ramos seja sempre menor ou igual a 1.

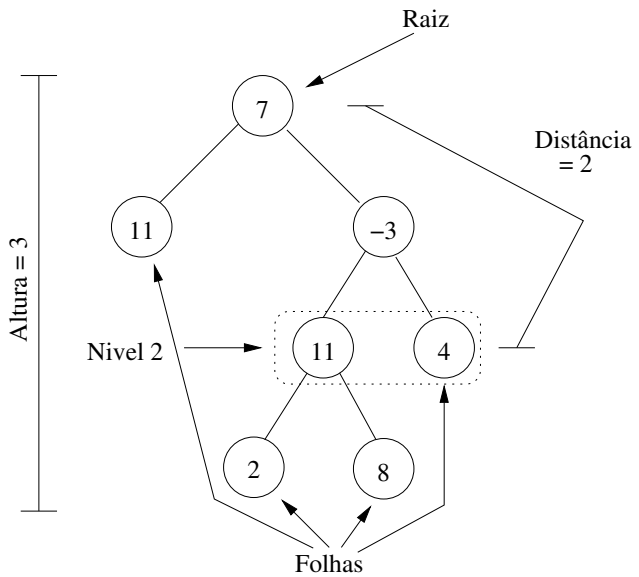


Figura 2.4: Árvores Binárias — Terminologia

Uma operação importante para este tipo de dados é a travessia da árvore, isto é, o «visitar» de todos os nós sem esquecimentos e sem repetições. Podemos ter travessias *em-ordem*, em *pré-ordem* e em *pós-ordem*, consoante a ordem pela qual se visita a raiz em relação aos outros dois elementos, a sub-árvore esquerda e a sub-árvore direita.

Grafos conjunto de nós e de arcos entre eles (ver figura 2.2). É uma estrutura muito importante quando se quer modelizar situações em rede, isto é, situações em que se tem «locais» e «ligações» entre eles.

Para este tipo de estrutura temos, a exemplo das árvores, diferentes representações. Talvez a mais usual é a representação através de um conjunto de nós e um conjunto de arcos (pares de nós: nó origem, nó destino).

As operações passam pela criação do grafo vazio, a inserção de um novo nó, a inserção de um novo arco, as travessias, e as operações de retirar nós e arcos.

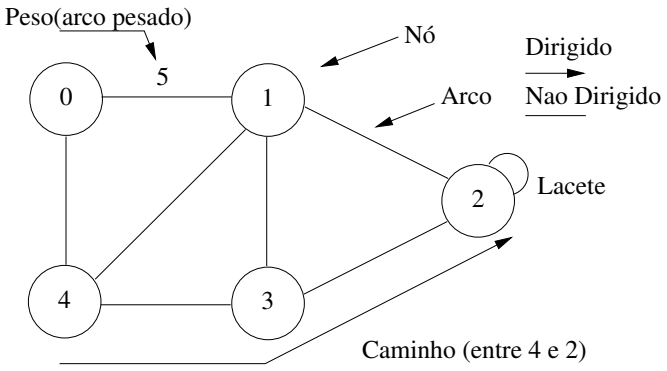


Figura 2.5: Grafos — Terminologia

Pelas definições destes tipos de dados acima apresentados é possível reparar que a definição dos elementos é feita de forma recorrente. Por exemplo para as *Pilhas*:

$$\text{Pilha} = \begin{cases} \text{Vazia,} & \text{caso de base} \\ \text{Elemento}:\text{Pilha} & \text{caso recorrente} \end{cases}$$

Este tipo de estrutura é possível de representar em *C* da seguinte forma:

```
typedef struct noPilha { // nó de um pilha
  int elems;             // caso de base (elemento)
  struct noPilha* prox;  // caso recorrente, ponteiro para o próximo elemento
} NoPilha;

typedef NoPilha* Pilha; // Pilha como o tipo ponteiro para Nó
```

Temos então a definição de uma pilha como uma lista de nós. A definição recorrente é possível dado que o elemento não se auto-referência, o que temos é que `prox` é do tipo ponteiro para `no` e não um elemento do tipo `no`.

Para a definição de uma árvore binária ter-se-ia algo de semelhante.

```
typedef struct noAB { // Nó árvore binária
  int elems;          // (caso de base) o elemento na «raiz» da árvore
  struct noAB* abesq; // (caso recorrente) ponteiro para a AB esquerda
  struct noAB* abdir; // (caso recorrente) ponteiro para a AB direita
} NoAB;

typedef NoAB* AB;  // Árvore Binária, ponteiro para Nó de árvore binária
```

2.2.2 Exemplo: Pilha de Caracteres

A título de exemplo apresenta-se uma possível implementação do tipo de dados abstracto *Pilha* em *C* (sem comentários no programa).

Ficheiro `pilhas.h`

```
/*********************************************/
/*  Pilhas Implementação Dinâmica            */
/*********************************************/
#if !defined PILHAS
#define PILHAS

/* Pilha de inteiros - lista simplesmente ligada de int
 */
typedef struct noPilha {
  char elems;
  struct noPilha* prox;
} NoPilha;

typedef NoPilha* Pilha;
/* Cria pilha - cria a pilha vazia
 * ⇒ p : Pilha* - ponteiro para Pilha
 * ⇐ void
 * o ponteiro *p é inicializado a NULL;
 */
void cria(Pilha*);

/* Push - insere um elemento no topo da pilha
 * ⇒ n : char - elemento a inserir
 * ⇒ p : Pilha* - ponteiro para Pilha
 * ⇐ void - a pilha é modificada
 */
void push(char, Pilha*);

/* top - vê o elemento no topo da pilha
 * ⇒ p : Pilha
 * ⇐ n : char - o elemento no topo da pilha
 * NOTA : não aplicável à pilha vazia
 */
char top(Pilha);

/*
```

```
 * pop - retira um elemento do topo da pilha
 * ⇒ p : Pilha* - ponteiro para Pilha
 * ⇐ void - a pilha é modificada
 * NOTA : não aplicável à pilha vazia
 */
void pop(Pilha*);

/*
 * vazia - verifica se a pilha é a pilha vazia
 * ⇒ p : Pilha
 * ⇐ b : int - 1, se é a pilha vazia; 0, caso contrário
 */
int vazia(Pilha);

#endif
```

Ficheiro `pilhas.c`

```
#include "pilha.h"
#include <stdlib.h>
#include <stdio.h>

void cria(Pilha* p){
  (*p) = NULL;
}

void push(char elem,Pilha* p){
  Pilha novo;

  novo = (Pilha) malloc(sizeof(NoPilha));
  novo->elems = elem;
  novo->prox = (*p);
  (*p) = novo;
}

void pop(Pilha* p){
  Pilha aux;

  aux = (*p);
  (*p) = (*p)->prox;
  free(aux);
}

char top(Pilha p){
  return(p->elems);
}

int vazia(Pilha p){
  return(p == NULL);
}
```

Capítulo 3

Estruturas Lineares

3.1 T.D.A. Pilhas

Sequência linear de elementos com acesso por um só ponto, o topo. Implementam a disciplina de acesso *Last In First Out (LIFO)*, o último a entrar é o primeiro a sair. Entre muitas utilizações temos o cálculo de expressões em notação Polaca inversa.

$$Pilha = (\{pilhaVazia, elemento\!:\!Pilha\}, \{\text{criar, push, pop, top, vazia?}\})$$

As pilhas são então definidas por recorrência.

$$\text{Pilha} = \begin{cases} \text{pilhaVazia,} & \text{Caso de base} \\ \text{Elemento:Pilha} & \text{Caso recorrente} \end{cases}$$

Este tipo de estrutura é possível de representar em *C* da seguinte forma:

```
typedef struct noPilha { // nó de um pilha
  int elems;          // caso de base (um inteiro)
  struct noPilha* prox;  // caso recorrente (ponteiro para o próximo elemento)
} NoPilha;

typedef NoPilha* Pilha;  // Pilha como o tipo ponteiro para Nó
```

Temos então a definição de uma pilha como uma lista de nós. A definição recorrente é possível dado que o elemento não se auto-referência, o que temos é que '`prox`' é do tipo ponteiro para '`no`' e não um elemento do tipo '`no`'. O caso de base é dado pelo ponteiro nulo, «`NULL`», definido pela linguagem *C*, como sendo um ponteiro nulo, para um qualquer tipo de dados.

Para uma correcta definição das operações internas é necessário explicitar o conjunto das pilhas não vazias $PilhaNãoVazia = Pilha \setminus \{pilhaVazia\}$.

Pode-se definir um conjunto completo e minimal de operações para o *TDA Pilhas*. Completo no sentido de se poderem com elas fazer todas as manipulações possíveis com pilhas, minimal no sentido de não ser possível prescindir de nenhuma delas.

As operações têm a seguinte especificação:

criar cria a pilha, sendo que o seu estado inicial é o da pilha vazia.

$$\begin{array}{rcccl} \text{criar} & : & \mathbf{1} & \longrightarrow & \mathit{Pilha} \\ & & * & \longmapsto & \mathit{pilhaVazia} \end{array}$$

Em linguagens procedimentais como o *C*, esta operação é separada em dois momentos: a criação do ponteiro através da definição de uma variável do tipo apropriado; a inicialização da variável anteriormente criada através da atribuição à mesmo do valor *ponteiro nulo* (≪`NULL`≫).

A função '`criar`', a ser implementada em *C*, vai corresponder a este segundo passo.

Temos então num primeiro momento (num dado programa que utilize as Pilhas).

```
Pilha p;
```

e num segundo momento a função de ≪criação≫ da pilha.

```
void criar(Pilha* p){
  (*p) = NULL;
}
```

push insere um elemento no topo da pilha.

$$\begin{array}{rcccl} \text{push} & : & \mathit{Pilha} \times \mathit{Elementos} & \longrightarrow & \mathit{Pilha} \\ & & (p, e) & \longmapsto & e : p \end{array}$$

Dado uma pilha e um elemento, coloca o elemento no topo da pilha. A pilha vai crescer em número de elementos, consoante a implementação usada será necessário acrescentar um novo nó à pilha, ou então, modificar a variável que guarda a dimensão da mesma.

```
void push(int elem, Pilha* p){
  Pilha novo;

  novo = (Pilha) malloc(sizeof(NoPilha));
  novo->elems=elem;
  novo->prox=(*p);
  (*p)=novo;
}
```

pop retira (sem o usar) o elemento que está no topo da pilha.

$$\begin{array}{rcccl} \text{pop} & : & \mathit{PilhaNãoVazia} & \longrightarrow & \mathit{Pilha} \\ & & e : p & \longmapsto & p \end{array}$$

É de notar que nesta operação nada é feito para aproveitar o elemento que se retira. Dado uma pilha, não vazia, o elemento do topo é retirado da pilha. A pilha diminui de número de elementos, consoante a implementação será necessário libertar o espaço de memória usado por esse elemento, ou então, modificar a variável que guarda a dimensão da pilha.

Uma nota importante: **esta operação só se aplica a pilhas não vazias**. Como proceder na situação contrária é difícil de explicitar numa linguagem como o *C*. Uma

das soluções possíveis passa por ter sempre um condicional que verifica essa condição a «envolver» a aplicação desta operação.

```
void pop(Pilha* p){
  Pilha aux;

  aux = (*p);
  (*p)=(*p)->prox;
  free(aux);
}
```

top verifica, sem modificar a pilha, o elemento do topo da pilha.

$$\begin{array}{rcccl} \text{top} & : & \mathit{PilhaNãoVazia} & \longrightarrow & \mathit{Elementos} \\ & & e : p & \longmapsto & e \end{array}$$

A pilha não é alterada por esta operação, só o elemento do topo da mesma é devolvido por esta operação.

Uma nota importante: **esta operação só se aplica a pilhas não vazias**. A forma de evitar este potencial problema será o mesmo que o usado para a operação `pop`.

```
int top(Pilha p){
  return(p->elems);
}
```

vazia? (vaziaPilha) Verifica se uma dada pilha está, ou não, vazia.

$$\begin{array}{rcccl} \text{vazia?} & : & \mathit{Pilha} & \longrightarrow & \mathbb{B} \\ & & p & \longmapsto & \begin{cases} \mathcal{V}, & \text{se p = pilhaVazia} \\ \mathcal{F}, & \text{se p} \neq \text{pilhaVazia} \end{cases} \end{array}$$

```
int vaziaPilha(Pilha p){
  return(p == NULL);
}
```

Exemplo: Calculadora em Notação Polaca Inversa

A título de exemplo apresenta-se uma possível implementação do tipo abstracto de dados *Pilha* (de inteiros) em *C*.

pilhaInt.h Definição da estrutura e das funções:

Algoritmo 3.1: Pilha de Inteiros (especificação)

```
/*************************************************/
/*  Pilhas (de inteiros) Implementação Ligada    */
/*************************************************/
#if !defined PILHASINT
#define PILHASINT
```

```
// definição de nó
typedef struct noPilha {
  int elems;
  struct noPilha* prox;
} NoPilha;

// uma pilha é um ponteiro para nó
typedef NoPilha* Pilha;

/* Cria a pilha vazia (ponteiro NULL)
 * --> p, ponteiro indefinido - modificado para  p=NULL
 * <-- void
 */
void criar(Pilha*);

/* Dado um elemento e uma pilha, coloca o elemento no topo da pilha
 * --> e : int
 * --> p : Pilha - modificado para e:p
 * <-- void
 */
void push(int, Pilha*);

/* Devolve o elemento no topo da pilha (não vazia)
 * --> e:p : Pilha (não vazia)
 * <-- e
 */
int top(Pilha);

/* Retira um elemento do topo da pilha (não vazia)
 * --> e:p : Pilha (não vazia) - a pilha passa a ser somente p
 * <-- void
 */
void pop(Pilha*);

/* Verifica se uma pilha está vazia (ou não)
 * --> p : pilha
 * <-- n : int - 0, a pilha não está vazia, 1 a pilha está vazia
 */
int vaziaPilha(Pilha);

#endif
```

pilhaInt.c Implementação das funções:

Algoritmo 3.2: Pilha de Inteiros (implementação)

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

#include "pilhaInt.h"

void criar(Pilha* p){
  (*p) = NULL;
```

```
}

void push(int elem, Pilha* p){
  Pilha novo;

  novo= (Pilha) malloc(sizeof(NoPilha));
  novo->elems=elem;
  novo->prox=(*p);
  (*p)=novo;
}

// erro se 'p' é a pilha vazia
void pop(Pilha* p){
  Pilha aux;

  aux=(*p);
  (*p)=(*p)->prox;
  free(aux);
}

// erro se 'p' é a pilha vazia
int top(Pilha p){
  return(p->elems);
}

int vaziaPilha(Pilha p){
  return(p==NULL);
}
```

calculadoraPolacaInversa.c Calculadora em Notação Polaca Inversa, calculadora com as operações aritméticas elementares (adição, subtracção, multiplicação e divisão). Utiliza uma pilha de inteiros como forma de estruturar o cálculo.

Algoritmo 3.3: Calculadora Polaca Inversa

```
/* Calculadora em Notação Polaca Inversa - calculadora com as
 * operações aritméticas elementares (adição, subtracção,
 * multiplicação e divisão).
 */
#include <string.h>
#include <stdlib.h> // atoi
#include <stdio.h>

#include "pilhaInt.h" // T.D.A. Pilha de Inteiros

int main() {
  Pilha pl;

  criar(&pl);

  char aux[10];
  int operando,op1,op2,res;
```

```
  printf("Introduza uma expressão em NPI\t");
  printf("('.' para terminar)\t");

  scanf("%s",aux);
  if (strcmp(aux,".")==0) {
    printf("Expressão vazia\n");
    return 1;
  }
  else {
    while (strcmp(aux,".")!=0) {
      // se é um operando
      if (strcmp(aux,"+")==0 || strcmp(aux,"*")==0 ||
          strcmp(aux,"-")==0 || strcmp(aux,"/")==0) {
        // retira da pilha os dois operandos
        op2 = top(pl);
        pop(&pl);
        op1 = top(pl);
        pop(&pl);
        // faz o cálculo
        if (strcmp(aux,"+")==0)
          res = op1+op2;
        if (strcmp(aux,"*")==0)
          res = op1*op2;
        if (strcmp(aux,"-")==0)
          res = op1-op2;
        if (strcmp(aux,"/")==0)
          res = op1/op2;
        // coloca o resultado na pilha
        push(res,&pl);
      }
      else { // é um operando
        // converte de string para inteiro
        operando = atoi(aux);
        // coloca-o na pilha
        push(operando,&pl);
      }
      // lê um novo elemento da expressão
      //     printf("introduza um elemento da expressão:\t");
      scanf("%s",aux);
    }
    // Após terminar o ciclo o resultado está no topo da pilha (que
    // aliás só contém esse elemento)
    res =  top(pl);
    printf("O resultado da expressão é:\t %d\n",res);
    return 0;
  }
}
```

3.2 T.D.A. Filas

Sequências de elementos com acesso apenas pelas extremidades, a entrada e a saída. Implementam a disciplina *First In First Out (FIFO)*, o primeiro a entrar é o primeiro a sair. A simulação de filas de espera, sejam elas caixas num dado hipermercado ou semáforos num dado cruzamento, podem ser modelizadas através deste tipo de estrutura.

$$Fila = (\{filaVazia, elemento{:}Fila\}, \{\text{criar, inserir, retirar, verTopo, vazia?}\})$$

A exemplo das pilhas, as filas são também definidas por recorrência. De igual modo é necessário explicitar o conjunto das filas não vazias: $FilaNãoVazia = Fila \setminus \{filaVazia\}$.

Em termos de implementação ir-se-á obter uma estrutura semelhante às pilhas.

```
typedef struct noFila {
    int elems;
    struct noFila* prox;
} NoFila;

typedef NoFila* Fila;
```

As operações têm a seguinte especificação:

criar cria a fila, o seu estado inicial é vazio, a *filaVazia*.

$$\begin{array}{rcccl} \text{criar} & : & \mathbf{1} & \longrightarrow & Fila \\ & & * & \longmapsto & filaVazia \end{array}$$

De forma semelhante ao caso das pilhas, também aqui, em linguagens procedimentais esta função não está definida de forma explícita. Trata-se de definir, conforme as implementações uma fila com zero elementos, uma fila em que tanto o topo como a base não definem (apontam para) nenhum elemento.

No caso de uma implementação (a exemplo da pilha) baseada numa lista simplesmente ligada tem-se:

No programa principal: a definição de um ponteiro do tipo `Fila`.

```
Fila fl;
```

No TDA Fila: a inicialização desse ponteiro como sendo o ponteiro nulo.

```
void criaFila(Fila *fl){
  (*fl) = NULL;
}
```

inserir insere um elemento na base (início) da fila.

$$\begin{array}{rcccl} \text{inserir} & : & Fila \times Elementos & \longrightarrow & Fila \\ & & (f, e) & \longmapsto & e : f \end{array}$$

A fila cresce de um elemento. Assim como para a pilha, pode ser necessário criar um novo nó à fila ou modificar a sua dimensão.

Para a implementação o elemento a inserir é feito no início da fila. A opção passa por ter, ou um inserir fácil de implementar, ou um retirar fácil de implementar. Aqui optou-se pela primeira opção.

```
void insereFila(int elem, Fila* fl){
  Fila novo;

  novo=afectaFila();
  novo->elems=elem;
  novo->prox=(*fl);
  (*fl)=novo;
}
```

retirar retira (sem o usar) um elemento do topo (fim) da fila.

$$\begin{array}{rcccl} \text{retirar} & : & FilaNãoVazia & \longrightarrow & Fila \\ & & f : e & \longmapsto & f \end{array}$$

É de notar que nesta operação nada é feito para aproveitar o elemento que se retira. Dado uma fila, não vazia, o elemento do fim da fila é retirado. A fila diminui de número de elementos, consoante a implementação será necessário libertar o espaço de memória usado por esse elemento, ou então, modificar a variável que guarda a dimensão da fila.

Uma nota importante: **esta operação só se aplica a fila não vazias**. Como proceder na situação contrária é difícil de explicitar numa linguagem como o *C*. Uma das soluções possíveis passa por ter sempre um condicional que verifica essa condição a «envolver» a aplicação desta operação.

```
void retiraFila(Fila* fl){
  NoFila *aux, *ant;

  // caso a fila não esteja vazia tem de ir até ao último elemento
  aux=(*fl);
  ant=aux;
  while (aux->prox != NULL) {
    ant=aux;
    aux=aux->prox;
  }
  if (ant == aux) { // o ciclo anterior é vazio
    (*fl) = NULL;
  }
  else {
    ant->prox=NULL;
    free(aux);
  }
}
```

verTopo verifica, sem modificar a fila o topo (fim) da fila.

$$\begin{array}{rcccl} \text{verTopo} & : & \mathit{FilaNãoVazia} & \longrightarrow & \mathit{Elementos} \\ & & f : e & \longmapsto & e \end{array}$$

A fila não é alterada por esta operação, só o elemento do fim da mesma é devolvido por esta operação.

Uma nota importante: **esta operação só se aplica a filas não vazias**. A forma de evitar este potencial problema será o mesmo que o usado para a operação `inserir`.

```
int topoFila(Fila fl){
  NoFila *aux,*ant;

  ant=NULL;
  aux=fl;
  while (aux!=NULL) {
    ant=aux;
    aux=aux->prox;
  }
  return(ant->elems);
}
```

vazia? (estaVazia) Verifica se uma dada fila está, ou não, vazia.

$$\begin{array}{rcccl} \text{vazia?} & : & \mathit{Fila} & \longrightarrow & \mathbb{B} \\ & & p & \longmapsto & \begin{cases} \mathcal{V}, & \text{se p} = \text{filaVazia} \\ \mathcal{F}, & \text{se p} \neq \text{filaVazia} \end{cases} \end{array}$$

```
int filaVazia(Fila fl){
  return(fl==NULL);
}
```

O conjunto de operações acima definido para o *TDA Filas* é completo e minimal.

Exemplo: Fila de Inteiros

A título de exemplo apresenta-se uma possível implementação do tipo abstracto de dados *Fila* (de inteiros) em *C*.

filaInt.h Definição da estrutura e das funções:

Algoritmo 3.4: Fila de Inteiros (especificação)

```
/************************/
/*  Filas Int Definição        */
/************************/
#ifndef FILAS
#define FILAS

// definição de nó
typedef struct noFila {
    int elems;
```

```
    struct noFila* prox;
} NoFila;

// uma fila é um ponteiro para nó
typedef NoFila* Fila;


/* Cria a fila vazia (ponteiro NULL)
 * --> f : Fila - ponteiro indefinido, modificado para  f=NULL
 * <-- void
 */
void criaFila(Fila *);


/* // Dado um elemento e uma fila, insere o elemento no início da fila
 * --> e : int
 * --> f : Fila - modificado para e:f
 * <-- void
 */
void insereFila(int, Fila *);

/* retira o elemento do fim da fila (não vazia)
 * --> f:e : Fila (não vazia) - a fila passa a ser somente f
 * <-- void
 */
void retiraFila(Fila *);

/* // devolve o elemento no fim da fila (não vazia)
 * --> f:e : Fila (não vazia)
 * <-- e
 */
int topoFila(Fila);

/* // verifica se uma fila está (ou não) vazia
 * --> f : Fila
 * <-- n : int - 0, a fila não está vazia, 1 a fila está vazia
 */
int filaVazia(Fila);

#endif
```

filaInt.c Implementação das funções:

Algoritmo 3.5: Fila de Inteiros (implementação)

```
/****************************/
/*  Filas Int Implementação */
/****************************/
#include <stdio.h> // necessário por causa do NULL
#include <stdlib.h>
#include <stddef.h>

#include "filaInt.h"
```

```
// Criar a fila vazia, o ponteiro NULL
void criaFila(Fila *fl){
  (*fl) = NULL;
}

// Dado um elemento e uma fila, coloca o elemento no início da fila
void insereFila(int elem, Fila* fl){
  Fila novo;

  novo= (Fila) malloc(sizeof(NoFila));
  novo->elems=elem;
  novo->prox=(*fl);
  (*fl)=novo;
}

// Retira um elemento do fim da fila (não vazia)
void retiraFila(Fila* fl){
  NoFila *aux, *ant;

  // caso a fila não esteja vazia tem de ir até ao último elemento
  aux=(*fl);
  ant=aux;
  while (aux->prox!=NULL) {
    ant=aux;
    aux=aux->prox;
  }
  if (ant==aux) {
    (*fl) = NULL;
  }
  else {
    ant->prox=NULL;
    free(aux);
  }
}

// Devolve o elemento no fim da fila (não vazia)
int topoFila(Fila fl){
  NoFila *aux,*ant;

  ant=NULL;
  aux=fl;
  while (aux!=NULL) {
    ant=aux;
    aux=aux->prox;
  }
  return(ant->elems);
}

// Verifica se uma fila está vazia (ou não)
int filaVazia(Fila fl){
  return(fl==NULL);
}
```

A exemplo das pilhas a representação de uma fila pode ser feita de diferentes formas. Seja na forma de lista simplesmente ligada, como no exemplo apresentado, seja na forma de uma lista contígua, etc.

3.3 T.D.A. Listas

Listas genéricas com acesso livre a uma qualquer posição. Podem ser usadas para modelizar todo o tipo de situações em que se pretende uma lista de elementos todos do mesmo tipo mas em que o número de elementos a considerar varia muito durante o correr do programa.

$$\text{Lista} = (\{listaVazia, elemento{:}lista\}, \{\text{criar, inserirN, retirarN, verN, vazia?}\})$$

A exemplo das pilhas e filas são também definidas inductivamente.
Pode-se definir um conjunto completo e minimal de operações para o *TDA Listas*.
As operações têm a seguinte especificação:

criar cria a lista, o seu estado inicial é o da lista vazia.

$$\begin{array}{rcccl} \text{criar} & : & \mathbf{1} & \longrightarrow & Lista \\ & & * & \longmapsto & listaVazia \end{array}$$

inserirN insere um elemento na n-ésima posição da lista.

$$\begin{array}{rcccl} \text{inserirN} & : & Lista \times Elementos \times \mathbb{N} & \longrightarrow & Lista \\ & & (l, e, i) & \longmapsto & l' = l_1 : \ldots : l_{i-1} : e : l_i : \ldots : l_n \end{array}$$

No caso em que o comprimento da lista é menor do que a posição em que se pretende efectuar a inserção pode-se optar por diferentes estratégias, para todo o $i >$ comprimento da lista:

- a inserção é feita logo após o último elemento da lista;
- a inserção não é feita, a lista fica inalterada (sem gerar um erro);
- a operação só está definida para inserções com $0 < i \leq$ comprimento da lista. A tentativa de inserção fora desses limite gera um erro.

A última das opções é aquela mais correcta do ponto de vista formal, no entanto é aquela que, por representar potenciais situações de erro, aquela que é mais complicada de implementar.

Note-se que aqui também pode acontecer a situação de erro dada por um valor de i negativo.

Em conclusão: um pouco à semelhança das operações de *pop* e *top* do tipo pilha a utilização desta operação deve ser salvaguarda através de um condicional apropriado.

retirarN retirar um elemento da lista.

$$\begin{array}{rcccl} \text{retirarN} & : & \mathit{Lista} \times \mathbb{N} & \longrightarrow & \mathit{Lista} \\ & & (l, i) & \longmapsto & l' = l_1 : \ldots : l_{i-1} : l_{i+1} : \ldots : l_n \end{array}$$

No caso em que o comprimento da lista é menor do que a posição em que se pretende efectuar a remoção esta não é efectuada. A outra opção possível é a de considerar que nesse caso o valor da função não está definido (situação de erro).

verN obtém o elemento na posição n.

$$\begin{array}{rcccl} \text{verN} & : & \mathit{Lista} \times \mathbb{N} & \longrightarrow & \mathit{Elementos} \\ & & (l, i) & \longmapsto & e\ (= l_i) \end{array}$$

No caso em que o comprimento da lista é menor do que a posição que se pretende visualizar tem-se uma situação de erro.

vazia? (estaVazia) verifica se a lista está vazia.

$$\begin{array}{rcccl} \text{vazia?} & : & \mathit{Lista} & \longrightarrow & \mathbb{B} \\ & & p & \longmapsto & \begin{cases} \mathcal{V}, & \text{se p = listaVazia} \\ \mathcal{F}, & \text{se p} \neq \text{listaVazia} \end{cases} \end{array}$$

T.D.A. Lista — Modelização

As listas podem ser modelizadas através de uma estrutura do tipo vectorial, dita representação contígua, ou como uma sequências de elementos, dita representação ligada (ver exercício C.5).

T.D. A. Lista — Representação contígua Como uma estrutura de armazenamento contígua o TDA Lista pode ser representado da seguinte forma:

```
typedef struct listaint {
  int dim;        // dimensão da lista (número de elementos)
  int * espaco;   // espaço contíguo para guardar elementos do tipo int
} ListaInt;
```

A vantagem deste tipo de estrutura é o imediato (e muito eficiente) acesso aos seus elementos. A desvantagem deste tipo de estrutura é a sua rigidez, uma vez definido o seu tamanho máximo não é fácil (de forma eficiente) alterar. Por outro lado no que diz respeito a sua utilização para a representação de uma lista, as inserções e remoções vão implicar o deslocar de elementos da lista, para «abrir» espaço, e/ou «fechar» espaço, e como tal não serão operações eficientes.

A conclusão que se pode tirar é que esta é uma boa representação em situações em que se saiba de antemão o comprimento da lista e em que esta seja alterada muito poucas vezes, mas acedida muitas vezes.

T.D.A. Lista — Representação ligada Como uma estrutura de armazenamento não contígua (dita, representação ligada) o TDA Lista pode ser representado da seguinte forma:

```
typedef struct no { // Nós
  int elems;
  struct no *prox;
} NoLista;

// Lista, ponteiro para um nó
typedef NoLista* ListaInt;
```

Esta é uma representação alternativa à representação contígua. É uma representação mais dispendiosa em termos de espaço, além dos elementos é necessário guardar os ponteiros entre eles. É uma representação menos eficiente no acesso, só o elemento inicial é que é acedido directamente, para todos os outros é necessário «percorrer» a lista. Mas, ao contrário da representação contígua, a dimensão da estrutura ajusta-se exactamente ao número de elementos da lista, O acesso aos elementos não é directo, mas o inserir e/ou retirar dos elementos é feito por uma simples manipulação dos ponteiros.

A conclusão que se pode tirar é que esta é uma boa representação em todas as situações em que a lista sofra muitas alterações, isto é, haja muitas inserções e remoções de elementos. Nestes casos, embora o acesso aos elementos seja menos eficiente, o eficiente manipular da estrutura, compensa.

T.D.A. Lista — Operações Como tipo que se pretende «abstracto», isto é o construir de um módulo que, independentemente da forma como é implementado, tem sempre uma dada funcionalidade, a especificação das operações são idênticas para ambos os casos.

```
/* Construir a lista vazia
 * —>
 * <— l : ListaInt - lista vazia, ponteiro nulo
 */
ListaInt criar();

/* Destruir a lista (libertar o espaço e defini-la como a lista vazia)
 * —> l : ListaInt*
 * <— l:  ListaInt*  - l=NULL e o espaço ocupado foi libertado
 */
void destruir(ListaInt*);

/* Precondição: posicao >= 0
 * Inserir, insere o elemento "elem" na posição "posicao"
 * —> l : ListaInt*
 * —> elem : int
 * —> posicao : int - posicao >= 0
 * <— l : ListaInt* - lista alterada, com um novo elemento na posição "posicao"
 * Notas: se posicao>dim, a inserção é feita no fim da lista.
 */
void inserirN(ListaInt *,int,int);

/* Precondição: posicao >=0
 * Retirar, remove o elemento da posição "posicao"
 * —> l : ListaInt*
```

```
 * —> posicao : int
 * <— l : ListaInt* - a lista 'l' com o elemento na "posicao" retirado
 * Nota: se posicao>dim não se retira nenhum elemento
 */
void retirarN(ListaInt *,int);

/* Precondição: 0 <= posicao < dim
 * Devolver, devolve o elemento numa dada posição
 * —> l : ListaInt
 * —> posicao : int - 0 <= posicao < dim
 * <— e : int - o elemento na posição "posicao"
 * Nota: se 0<= posicao < dim não se verifica, devolve zero
 */
int verN(ListaInt, int);

/* Vazia, verifica se a lista está vazia
 * —> l : ListaInt
 * <— 1 : int - lista vazia; 0 - lista não vazia
 */
int vazia(ListaInt);
```

A implementação das operações no TDA Lista é o objectivo (além da especificação) de exercício específico, na folha C.5.

Capítulo 4

Estruturas Não Lineares

4.1 T.D.A. Árvores

Árvores Binárias uma raiz e duas sub-árvores.

Dado que é possível representar qualquer tipo de árvore como uma árvore binária, este tipo de estrutura de dados é usada para modelizar todas as situações em que se pretende representar uma estrutura hierárquica.

$$\text{AB} = (\{Abvazia, ABesq\!:\!elemento\!:\!ABdir\}, \{\text{criar,destruir,insereRaiz,} \\ \text{insereFolha,retiraFolha,procuraElemento,vazia?}\})$$

No caso deste tipo de dados há variantes tanto na forma de definir assim como nas operações de base (Main & Savitch, 1997; Semgupta & Korobkin, 1994; Weiss, 1997; Aho *et al.*, 1983).

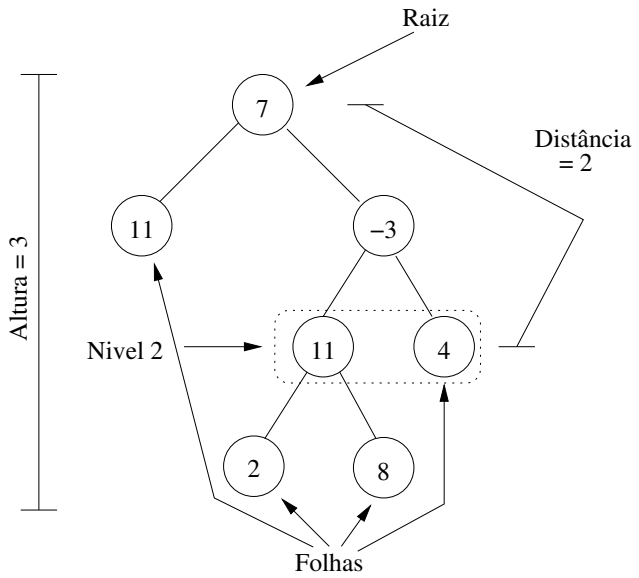


Figura 4.1: Árvores Binárias — Terminologia

Uma possível definição da estrutura é dada por um nó contendo a informação e ponteiros para outra (duas) árvores. Por exemplo:

```
typedef struct noAB { // Nó árvore binária
  int raiz;           // o elemento na «raiz» da árvore
  struct noAB* ABesq; // ponteiro para a AB esquerda
  struct noAB* ABdir; // ponteiro para a AB direita
} NoAB;

typedef NoAB* AB;
```

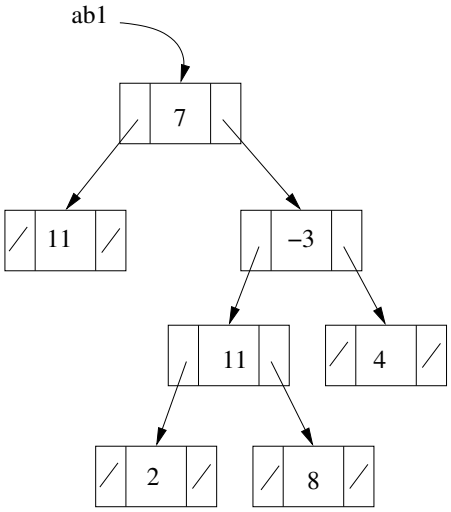


Figura 4.2: Árvores Binárias — Exemplo

Uma operação importante para este tipo de dados é a travessia da árvore, isto é, o «visitar» de todos os nós sem esquecimentos e sem repetições. Podemos ter travessias *em-ordem*, em *pré-ordem* e em *pós-ordem*, consoante a ordem pela qual se visita a raiz em relação aos outros dois elementos, a sub-árvore esquerda e a sub-árvore direita. De novo temos uma definição recorrente, neste caso com dois casos recorrentes e um caso de base.

Esta estrutura e a forma como é utilizada é muito mais complexa e variada do que os casos anteriores. Apresenta-se de seguida as definições das operações, é necessário ter em conta que neste caso não há uma uniformidade, estas definições representam uma possível definição para uma utilização genérica.

$$\begin{array}{rcl} \text{criaVazia} : \mathbf{1} & \longrightarrow & AB \\ * & \longmapsto & ABVazia \end{array}$$

$$\begin{array}{rcl} \text{criaAB} : AB \times Elementos \times AB & \longrightarrow & AB \\ (ab1, e, ab2) & \longmapsto & ab1 : e : ab2 \end{array}$$

Tanto $ab1$ como $ab2$ podem ser árvores binárias vazias. No caso extremo de ambas as sub-árvores serem vazias temos aqui uma operação que transforma um elemento numa árvore binária.

$$\begin{array}{rcl} \text{procuraElemento} : AB \times Elementos & \longrightarrow & \mathbb{B} \\ (ab, e) & \longmapsto & \begin{cases} \mathcal{V}, & e \in ab \\ \mathcal{F}, & e \notin ab \end{cases} \end{array}$$

$$\begin{array}{rcl} \text{insereElemento} : AB \times Elementos & \longrightarrow & AB \\ (ab, e) & \longmapsto & ab' \end{array}$$

A posição de inserção vai estar dependente da forma como os elementos estão organizados na árvore. No caso da inserção ser feito nos extremos (folhas) da árvore ao elemento serão «adicionadas» duas sub-árvores vazias.

$$\begin{array}{rcl} \text{retiraElemento} : AB \times Elementos & \longrightarrow & AB \\ (ab, e) & \longmapsto & ab' \end{array}$$

No caso do retirar de uma das folhas (elementos no extremo da árvore) é só uma questão de colocar no nó da árvore exactamente acima do elemento a retirar a árvore vazia. No caso de se pretender retirar um elemento do meio da árvore, isso vai implicar um re-ordenar da mesma. A forma exacta como isso é feito vai depender da forma como a árvore está organizada.

$$\begin{array}{rcl} \text{vazia?} : AB & \longrightarrow & \mathbb{B} \\ p & \longmapsto & \begin{cases} \mathcal{V}, & \text{se p = ABVazia} \\ \mathcal{F}, & \text{se p} \neq \text{ABVazia} \end{cases} \end{array}$$

4.1.1 Travessias de Árvores Binárias

Por travessia de uma árvore binária entende-se o «visitar» de todos os nós sem esquecimentos e sem repetições.

Podemos ter travessias *em-ordem*, em *pré-ordem* e em *pós-ordem*, consoante a ordem pela qual se visita a raiz em relação aos outros dois elementos, a sub-árvore esquerda e a sub-árvore direita.

Travessia em Pré-ordem

- «visita-se» (e.g. escreve-se o valor) a raiz;
- faz-se (de forma recorrente) a travessia da sub-árvore esquerda;
- faz-se (de forma recorrente) a travessia da sub-árvore direita.

Travessia em Ordem

- faz-se (de forma recorrente) a travessia da sub-árvore esquerda;
- «visita-se» (e.g. escreve-se o valor) a raiz;
- faz-se (de forma recorrente) a travessia da sub-árvore direita.

Travessia em Pós-ordem

- faz-se (de forma recorrente) a travessia da sub-árvore esquerda;
- faz-se (de forma recorrente) a travessia da sub-árvore direita;
- «visita-se» (e.g. escreve-se o valor) a raiz.

A escolha da ordem pela qual se faz as travessias das sub-árvores esquerda e direita é uma questão de escolha (é só importante manter a consistência).

O caso de base é a árvore binária vazia, caso em que nada é feito, mas que tem o efeito de terminar aí a chamada recorrente.

Para o exemplo apresentado acima (ver figura 4.2) ter-se-ia:

- Em pré-ordem: 7 11 -3 11 2 8 4
- Em ordem: 11 7 2 11 8 -3 4
- Em pós-ordem: 11 2 8 11 4 -3 7

Em qualquer dos casos não houve repetição, nem nenhum nó ficou por visitar. O tipo de travessia a escolher pode estar dependente da aplicação, isto é, da forma como a árvore binária foi construída.

4.1.2 Árvores de Pesquisa

Um tipo importante de árvore binária são as árvores de pesquisa, árvores cujos nós estão organizados de acordo com uma dada função de ordenação.

Por exemplo, dado uma árvore binária ela é considerada uma árvore binária de pesquisa para a função de ordenação, '$\leq$' (menor do que), aplicada a inteiros se: à esquerda da raiz (na sub-árvore esquerda), todos os elementos da árvore binária são menores, em valor, à raiz, e todos os elementos à direita da raiz (na sub-árvore direita) são maiores ou iguais do que a raiz.

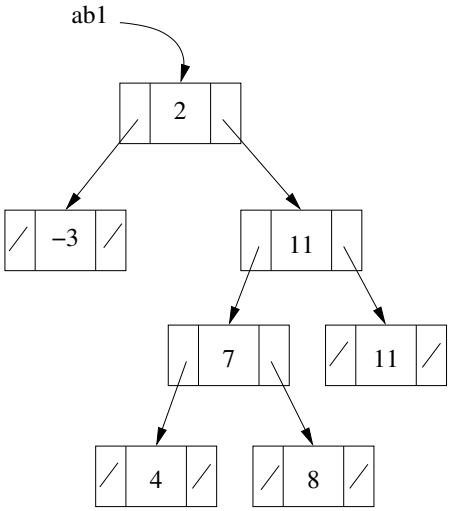


Figura 4.3: Árvores Binárias de Pesquisa

Este tipo de árvore binária é muito importante dado que podem ser consideradas como a implementação, numa estrutura de dados, do algoritmo de pesquisa binária.

Basicamente o processo da pesquisa binária segue a estrutura da árvore. O elemento a pesquisar é igual à raiz (caso de base, sucesso), ou é menor do que ele (recorrência na sub-árvores esquerda), ou é maior ou igual do que ele (recorrência na sub-árvore direita). A árvore vazia é também um caso de base neste caso de insucesso.

4.1.3 Árvores Binárias Balanceadas

A construção (simplista) de uma árvore binária de pesquisa pode levar a casos extremos de árvores totalmente assimétricas e que, em consequência, deixam de ser eficientes para efeitos de pesquisa.

Por exemplo, aplicando um algoritmo simples de colocação de elementos numa árvore binária de pesquisa:

- colocar na raiz (par uma árvore vazia), se é menor do que a raiz, colocar na sub-árvore esquerda, se é maior ou igual, colocar na sub-árvore direita
- à sequência de elementos -3 2 4 7 8 11 11

A árvore de pesquisa resultante, não seria mais do que uma lista, em forma de árvore (muito pouco) binária.

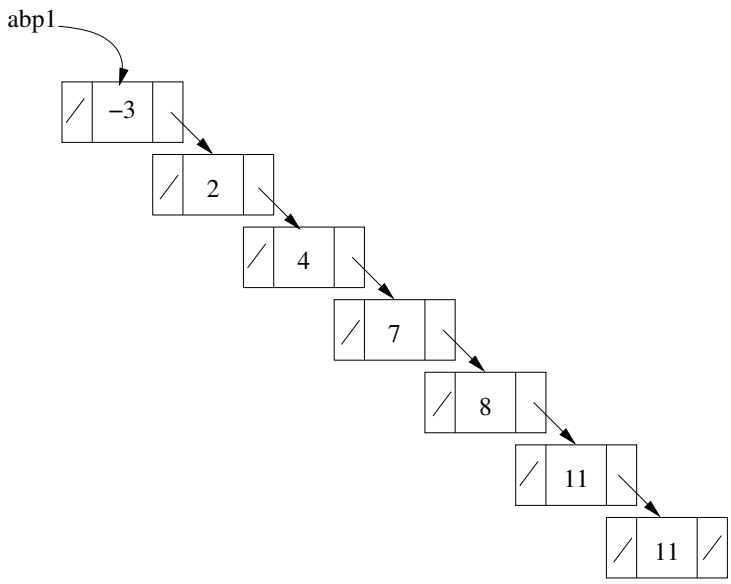


Figura 4.4: Árvores Binárias de Pesquisa — Não Balanceada

Numa árvore de pesquisa não balanceada (a figura 4.4 é um caso extremo) a procura de um elemento terá casos em que é menos eficiente do que se pretende.

Com vista a resolver este problema foram desenvolvidos algoritmos para garantir a construção balanceada de uma árvore. Aquilo que se perde na construção da árvore (em termos de tempo de execução), por se optar por um algoritmo mais complexo, ganha-se no momento da pesquisa.

Relembrando a definição: uma árvore balanceada é uma árvore binária cuja altura entre todo e qualquer par de ramos seja sempre menor ou igual a 1 (ver figura 4.5).

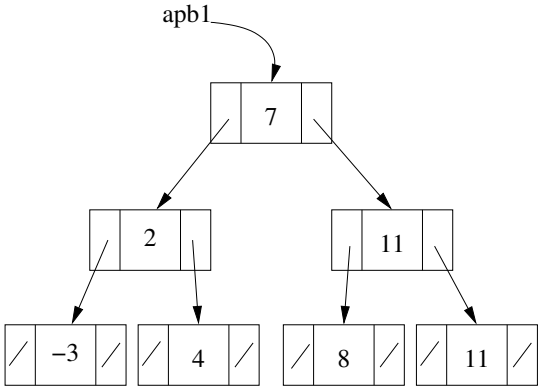


Figura 4.5: Árvores Binárias de Pesquisa — Balanceada

Um algoritmo muito conhecido para o balanceamento de árvores binárias é o algoritmo AVL (das iniciais dos seus autores Adelson-Velsky e Landis[1]).

Para árvores binárias de pesquisa balanceadas o algoritmo de pesquisa binária é o muito eficiente na pesquisa de um dado elemento entre uma «lista» de elementos do mesmo tipo.

4.2 T.D.A. Grafos

Conjunto de nós (ou vértices) e de arcos (ou arestas) entre eles (ver figura 4.6). É uma estrutura muito importante quando se quer modelizar situações em rede, isto é, situações em que se tem «locais» e «ligações» entre eles.

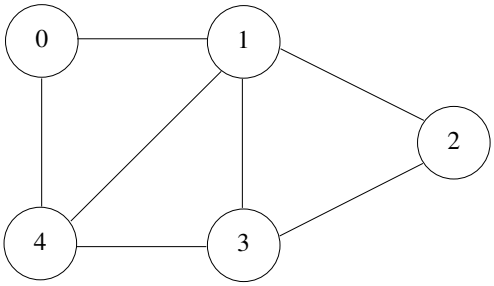


Figura 4.6: Grafo Não Dirigido

[1] Adelson-Velsky, Georgy; Landis, Evgenii (1962). "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences (in Russian). 146: 263–266.

Para este tipo de estrutura temos, a exemplo das árvores binárias, diferentes representações. Talvez a mais usual é a representação através de um conjunto de nós e um conjunto de arcos (pares de nós: nó origem, nó destino).

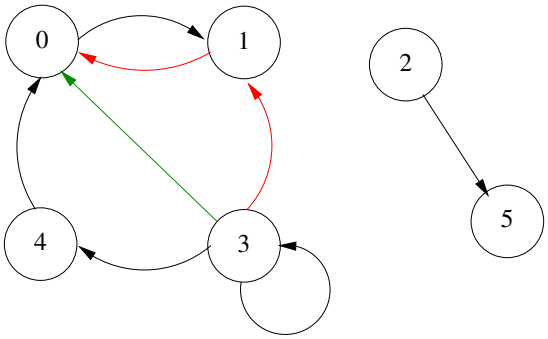


Figura 4.7: Grafo Dirigido, não Conexo

Temos que neste caso o grafo pode ser representado por o conjunto de nós (N):

$$N = \{0, 1, 2, 3, 4, 5\}$$

e pelo conjunto de arcos (A):

$$A = \{(0, 1), (1, 0), (2, 5), (3, 1), (3, 0), (3, 4), (3, 3), (4, 0)\}$$

As operações passam pela criação do grafo vazio, a inserção de um novo nó, a inserção de um novo arco, as operações de retirar nós e arcos e os caminhos e travessias.

Grafo = ({Nós = {Elementos}, Arcos= {(Elementos,Elementos)}},{criaGrafo, insereNo,retiraNo,insereArco,retiraArco,vazio?})

O grafo da figura 4.6 é **conexo**, isto é, tem um nó do qual é possível atingir todos os outros, aliás é **fortemente conexo** dado que de um qualquer nó pode-se atingir todos os demais. O grafo da figura 4.7 é não conexo.

Um **grafo completo** é um grafo que para cada nó existe um caminho para todos os outros nós. Um grafo completo é fortemente conexo.

O **Grafo Complementar** de um dado grafo G é um grafo H nos mesmos nós tais que dois nós de H são adjacentes se e somente se eles não são adjacentes em G.

Um grafo G é dito **grafo planar** se poder ser imerso no plano, isto é, os arcos não se cruzem entre si.

Um **caminho** é definido como uma sequência de um ou mais arcos entre dois dados nós. Por exemplo (ver figura 4.7)

$$< (3, 1), (1, 0) > \qquad < (3, 0) >$$

definem dois caminhos entre o nó 3 e o nó 0. O caminho $< (3, 0) >$ é o **caminho mais curto**.

4.2.1 Formas de Representação de Grafos

Matriz de adjacências Um grafo pode ser representado como uma matriz de adjacências, as linhas reference-se aos nós origem, as colunas ao nós destino. Por exemplo, para o grafo da figura 4.7, ter-se-ia:

	0	1	2	3	4	5
0	–	1	–	–	–	–
1	1	–	–	–	–	–
2	–	–	–	–	–	1
3	1	1	–	1	1	–
4	1	–	–	–	–	–
5	–	–	–	–	–	–

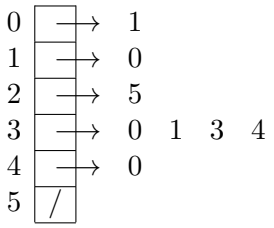
Esta representação, embora interessante pela facilidade da sua manipulação, só é apropriada para grafos completos ou quase completos. Para grafos como os apresentados (ver figuras 4.6 e 4.7) ter-se-ia uma matriz esparsa, isto é, uma matriz com muito poucas células significativas.

Em *C* ter-se-ia:

```
#define MAXNOS 1000;

// Grafo como Matriz de Adjacência
typedef struct grafo {
  int matAdj[MAXNOS][MAXNOS];
  int numNos;
} Grafo;
```

Lista de Adjacências Uma outra representação, mais compacta, mais apropriada à representação de grafos não completos, é a representação dada por um vector de nós e respectivas listas de adjacências. Por exemplo, para o grafo da figura 4.7, ter-se-ia:



Embora de manipulação mais complexa esta representação é muito comum dado se ajustar corretamente à dimensão do grafo a representar.

Em *C* ter-se-ia:

```
#define MAXNOS 1000;

// Lista de inteiros (adjacências)
// Nós
typedef struct no {
  int elem;
  struct no *prox;
} No;
```

```
// Lista, ponteiro para nó
typedef No* Lista;

// Grafo como vector de listas de adjacência
typedef struct grafo {
  Lista listAdj[MAXNOS];
  int numNos;
} Grafo;
```

4.2.2 Percursos em Grafos

À imagem das árvores binárias é também necessário ter critérios para percorrer um grafo, visitando os seus nós, totalmente e sem repetições.

No caso dos grafos a existência de possíveis ciclos (infinitos) torna a construção de percursos um pouco mais delicada.

Percurso em Amplitude Partindo-se de um dado nó (escolha aleatória) «visita-se» esse nó. O que se entende por «visita» vai depender do que se pretende obter ao percorrer o grafo, pode ser a soma dos seus valores, pode ser o colocar numa lista de todos os nós, etc.

Após se ter visitado o primeiro nó vai-se considerar cada um dos nós adjacentes. Para cada um desses nós temos que:

- visita-se o nó;
- coloca-se o nó em uma fila.

Ao terminar de visitar todos os nó adjacentes ao nó inicial, toma-se o nó que estiver no topo da fila, retira-se o nó da fila, e repete-se o processo usando sempre a fila para colocar novos nós a visitar.

Quando, no processo, se encontra um nó que já foi visitado, não se repete a visita, nem se coloca o mesmo na fila dos nós a visitar (ver Algoritmo 4.1).

```
funcao BuscaEmAmplitude(Grafo grafo)
   escolha um nó de partida s de G
   marque s
   insira s em F
   enquanto (F não está vazia) faz
      seja v o primeiro vértice de F
      retira v de F
      para (cada w em listaDeAdjacência de v) faz
         se (w não está marcado) entao
            visite aresta entre v e w
            marque w
            insira w em F
         fimse
      fimpara
   fimenquanto
fimfuncao
```

Algoritmo 4.1: Algoritmo Busca em Amplitude

Percurso em Profundidade Partindo-se de um dado nó (escolha aleatória) «visita-se» esse nó. De seguida vai-se escolher o primeiro nó adjacente ao nó inicial, desde que esse nó adjacente ainda não tenha sido visitado, fazendo-se:

- visita-se esse nó adjacente;
- coloca-se o nó inicial numa pilha;
- o nó adjacente passa a ser o nó inicial.

O processo repete-se até ao momento em que um dado nó «inicial» não tem nós adjacentes ao mesmo, ainda não visitados, então o novo nó inicial é aquele que estiver no topo da pilha, retira-se esse elemento da pilha e recomeça-se o processo até que a pilha esteja vazia.

```
funcao BuscaEmProfundidade(Grafo grafo)
    escolha um nó de partida v de G
    Pilha S
    S.push((v,v))
    enquanto (S não está vazia) faz
        (v,vl) = S.top()
        S.pop()
        visitar arco vl -> v
        se (v não está marcada) entao
            marcar v como já visitada
            para (todos os nós w na lista de adjacências de v) faz
                S.push((w,v))
            fimpara
        fimse
    fimenquanto
fimfuncao
```

Algoritmo 4.2: Algoritmo Busca em Profundidade

4.2.3 Caminho Mais Curto

Uma das operações mais importantes relacionadas com redes de transporte (modelizadas como grafos) é a procura por caminhos entre dois dados nós, nomeadamente a procura pelo caminho mais curto.

O caminho mais curto entre um nó de origem e um nó destino significa o encontrar de um caminho (conjunto de arcos) entre dois nós que apresente o «custo mínimo» para um dado critério, seja o custo (monetário), o tempo, a distância, ou um outro qualquer critério relevante para o problema em questão.

As diferentes situações que na prática requerem encontrar o caminho mais curto têm feito com que este problema tenha sido dividido em diferentes casos, a fim de facilitar seu estudo, pesquisas e desenvolvimento. Por exemplo:

- Determinar o caminho mais curto entre dois nós específicos de uma rede;
- Determinar o caminho mais curto entre todos os pares de nós da rede;
- Determinar o segundo, terceiro, quarto, etc., caminho mais curto;

- Encontrar o caminho mais rápido numa rede com tempos de viagem e dependendo de uma hora de saída;
- Encontrar o caminho mais curto entre dois nós específicos precisando-se atravessar obrigatoriamente determinados nós intermediários.

Um exemplo de uma aplicação usual é dada pelos aplicações para o cálculo de viagens que, além de possibilitarem a escolha dos critérios a adoptar (dinheiro, tempo, distância, etc.), dão em resposta mais do que uma possibilidade aos seus utilizadores.

Algoritmo de Dijsktra é um algoritmo para achar o caminho mais curto que foi concebido por Edsger Dijkstra em 1956 e publicado em 1959 (Dijkstra, 1959).

Dados dois nós, o nó inicial e o nó final o algoritmo atribuí valores arbitrários iniciais e depois vai melhorando a solução até encontrar aquela com o menor custo.

1. Começa-se por marcar todos os nós como **não visitados** e cria-se um conjunto com esses nós.
2. Atribui-se, para todos os nós, um custo inicial: zero para o nó inicial e infinito para todos os outros nós
3. Fixa-se o nó inicial como o nó actualmente em consideração, **nó corrente**.
4. Para o nó corrente, considera-se todos os nós adjacente ainda não visitados e calcula-se o custo do caminho entre eles. Compara-se o custo. Por exemplo, se o nó corrente A tem um custo de 6 e o arco entre ele e o adjacente B tem um custo de 2 então o custo de B, passando por A, é de 8 $(6 + 2)$. Se o nó B estava marcado com um custo maior do que 8, então actualiza-se, caso contrário mantêm-se o valor corrente.
5. Quando todos os nós adjacentes já foram considerados, marca-se o nó corrente como visitado e retira-se da lista do conjunto de nós não visitados. Um nó visitado não será mais considerado.
6. Se o nó destino já foi marcado como visitado, ou se o menor custo corrente entre nós não visitados é infinito (não existe um caminho entre os nós inicial e final) parar o ciclo. O algoritmo terminou.
7. Caso contrário, selecciona-se o nó não visitado que tem o menor custo corrente, fixa-se este como o novo nó corrente a recomeça-se o ciclo (passo 4).

```
funcao Dijkstra(Grafo grafo, no origem)
   criar o conjunto de nós Q

   para cada vertice v faz
       custo[v] := INFINITO
       noPrev[v] := NAODEFINIDO
       adicionar v a Q
   fimpara
   custo[origem] := 0
```

```
    enquanto Q is not empty faz
        noCorrente := nó em Q com o custo mínimo
        remover noCorrente de Q
        para cada v em Q, nó adjacente de noCorrente faz
            alt := custo[noCorrente] + custoArco(noCorrente,v)
            se alt < custo[v] entao
                custo[v] := alt
                noPrev[v] := noCorrente
            fimse
        fimpara
    fimenquanto
    return custo[], noPrev[]
fimfuncao
```

Algoritmo 4.3: Algoritmo de Dijsktra

Para implementar este algoritmo é necessário considerar conjuntos de nós, assim como listas de nós. O resultado é dado em termos de dois vectores, um com os custos relativos a cada um dos caminhos da origem para os diferentes nós, o outro com a informação referente ao nó prévio ao nó destino, esta última informação permite re-construir a sequência de nós que constituí o caminho da origem ao destino.

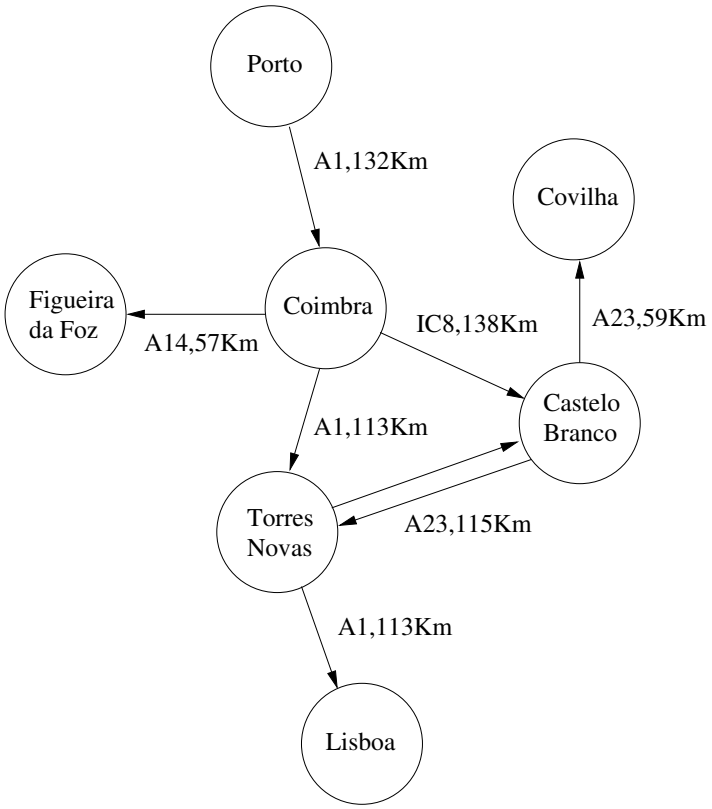


Figura 4.8: Grafos — Mapa de Estradas

Para o grafo representado um dado conjunto de ligações rodoviárias (ver figura 4.8) o resultado final seria o dado pela tabela abaixo (ver tabela 4.1). A primeira linha é dada pelos nós (destino) do grafo, a segunda linha dá-nos a distância (mínima) entre os nós origem e destino, a terceira linha dá-nos no nó anterior ao nó destino. A última linha permite reconstruir o caminho mais curto entre os nós origem e destino.

Porto	Coimbra	F. Foz	C. Branco	T. Novas	Lisboa	Covilhã
Indefinido	Porto	Coimbra	Coimbra	Coimbra	Torres Novas	Castelo Branco
0	132	189	270	245	358	329

Tabela 4.1: Grafos — Caminho mais curto

Que correspondem aos caminhos (origem → destino):

- Porto → Porto, 0Km;
- Porto → Coimbra, 132Km;
- Porto → Coimbra → Figueira da Foz, 189Km;
- Porto → Coimbra → Castelo Branco, 270Km;
- Porto → Coimbra → Torres Novas, 245Km;
- Porto → Coimbra → Torres Novas → Lisboa 358Km,
- Porto → Coimbra → Castelo Branco → Covilhã, 329Km.

Caso se pretenda obter somente o caminho mais curto referente a um dado destino basta parar o ciclo principal assim que o nó destino seja atingido.

Referências

Aho, Alfred, Hopcroft, John, & Ullman, Jeffrey. 1983. *Data Structures and Algorithms.* Addison-Wesley Publishing Company.

Darcano. 2007. *Tutorial: Aprenda a criar seu próprio Makefile.* Forum Ubuntu Linux - PT. `http://ubuntuforum-pt.org/index.php/topic,21155.0.html`.

de Sá, Joaquim P. Marques. 2004. *Fundamentos de Programação Usando C.* Tecnologias de Informação. FCA, Editora de Informática LDA. 68N/SA.

Dijkstra, E.W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, **1**, 269–271.

Kernighan, Brian, & Ritchie, Dennis. 1988. *The C Programming Language.* 2nd edn. Prentice Hall.

Knuth, Donald E. 1973. *The Art of Computer Programming.* 2nd edn. Vol. 1, Fundamental Algorithms. Reading, USA: Addisson-Wesley Publishing Company.

Knuth, Donald E. 1981. *The Art of Computer Programming.* 2nd edn. Vol. 2, Seminumerical algorithms. Reading, USA: Addisson-Wesley Publishing Company.

Main, Michael, & Savitch, Walter. 1997. *Data Structures and other Objects Using C++.* Addison-Wesley. DMAT 68P/MAI.

Mecklenburg, Robert. 2004. *Managing Projects with GNU Make.* 3rd edn. O'Reilly Media.

Semgupta, Saumyendra, & Korobkin, Carl Philip. 1994. *C++, Object-Oriented Data Structures.* New-York: Springer-Verlag. DMAT 68N/SEN.

Stroustrup, Bjarne. 1997. *The C++ Programming Language.* Addison Wesley Longman, Inc.

Weiss, Mark Allen. 1997. *Data Structures and Algorithm Analysis in C.* Menlo Park, California: Addison-Wesley.

Wirth, Niklaus. 1976. *Algorithms + Data Structures = Programs.* Automatic Computation. Englewood Clifs, New Jersey: Prentice-Hall, Inc.

Wirth, Niklaus. 1986. *Algorithms & Data Structures.* Englewood Cliffs, New Jersey, USA: Prentice-Hall.

WIRTH, NIKLAUS. 1989. *Algoritmos e Estruturas de Dados*. Rio de Janeiro: Prentice-Hall do Brasil.

Apêndice A

Construção de um Executável

As linguagens tais como *C* são ditas linguagens compiláveis. Quer se com isto dizer que, através da utilização de um programa próprio, o compilador, o programa escrito na linguagem *C* vai ser transformado num programa em código máquina e autónomo. Isto é o compilador vai transformar o programa em *C* num executável.

A forma como essa transformação é feita, e os programas que podem ser usados para nos ajudar nesse processo são o objecto deste capítulo.

A.1 Compilador de C

A transformação de um programa em *C* num programa executável é feito pelo compilador respectivo. É possível individualizar três fases distintas no processo de compilação: o pré-processamento; a fase de análise e de geração do código máquina (programa na linguagem da máquina); e agregação e finalização (algumas vezes designado por «linkagem», um aportuguesamento do termo inglês *linking* (agregando/juntando)). Vejamos com um pouco mais de detalhes estas diferentes fases.

pré-processamento: o ficheiro contendo o programa é analisado por completo sendo que as directivas de pré-processamento são cumpridas. É também nesta fase do processamento que todos os comentários são descartados. O resultado desta fase é um programa contendo exclusivamente código *C*. Num processo de compilação normal esta fase não produz nenhum ficheiro de saída.

análise: análise léxica (palavras), sintáctica (frases) e semântica (tipos de dados) do código resultante da fase anterior para avaliar se o mesmo está de acordo com a regras da linguagem *C*.

Se não houver erros, o resultado final desta fase é dado pela conversão do código *C* em código máquina. Nem sempre o resultado desta fase resulta na produção de um ficheiro de saída (ver a secção A.1.2).

agregação (*linkagem*): no caso de se tratar do ficheiro contendo a função `main` («ponto de partida» do programa) o compilador pode proceder à fase final da transformação, agregando todo o código máquina produzido nas fases anteriores, juntamente com as bibliotecas necessárias (incluídas através de directivas de pré-processamento), assim

como o código necessário à execução do código pelo sistema operativo (*runtime routines*), num ≪bloco≫ único que é o programa executável.

O resultado desta fase, para o caso em que não há erros, é um ficheiro contendo o executável respeitante ao nosso programa.

É de notar que actualmente é normal o ficheiro resultante do processo de compilação não ser verdadeiramente auto-contido, isto é, o ficheiro não tem em si tudo o que é necessário para o seu funcionamento.

O facto de cada vez mais se usar bibliotecas externas para complementar os programas faz com que a proporção entre o código próprio do programa e o código externo (bibliotecas) seja em muitos casos muito desproporcionado em favor das bibliotecas. Este facto é ainda mais visível numa linguagem como o *C* dado que a sua modularidade leva a um uso intensivo de bibliotecas padrão (biblioteca padrão do *C*, bibliotecas gráficas, etc.).

A conclusão do que foi dito acima é que, em geral, a fase de agregação vai ≪deixar de fora≫ as bibliotecas (do sistemas) objecto das instruções de inclusão (`#include`), deixando só a indicação que elas são necessárias para o funcionamento do programa. Este processo, designado por agregação dinâmica (*dynamic linking*) tem como grande vantagem gerar programas executáveis com uma dimensão (em ≪bytes≫) que é proporcional ao código *C* escrito pelo programador. Tem como desvantagem o facto de que, na ausência de uma das bibliotecas necessárias ao seu funcionamento, o programa não executará.

É possível explicitar que se pretende a inclusão de tudo o que é necessário para a execução do programa obrigando o compilador a proceder a uma, assim designada, agregação estática (*Static Linking*). A desvantagem é que um qualquer programa, mesmo que pequeno, irá gerar um executável de dimensão considerável (vai depender das bibliotecas que se pretendeu incluir). A vantagem é que neste caso o programa será auto-contido, não necessitando de nenhum recurso adicional para o seu funcionamento (ver a secção A.1.2).

A.1.1 Pré-processamento

As directivas de pré-processamento passam pela definição de ≪macros≫, da declaração dos nomes dos ficheiros a incluir (bibliotecas e/ou programas auxiliares), assim como a compilação condicional (Kernighan & Ritchie, 1988; Stroustrup, 1997).

Qualquer linha iniciada por '`#`' (eventualmente com espaços em branco antes) será pré-processada. O efeitos das directivas é global para todo o ficheiro em que estão contidas. A declaração de um destes comandos ocorre sempre numa única linha, no entanto uma declaração muito longa pode ser dividida em várias linhas do ficheiro, basta para tal colocar '\' no fim de uma linha para que a próxima linha seja formalmente uma continuação da linha anterior, isto é, do ponto de vista da definição as duas linhas passam a contar como um linha única.

Definição de ≪Macros≫

A sintaxe deste tipo de definição é a seguinte:

```
#define   <identificador> <sequênciaPalavrasReservadas>
#define   <identificador> (<listaIdentificadores>) <sequênciaPalavrasReservadas>
```

A definição de «macros» serve para dar um nome a uma expressão (eventualmente) complexa. Os casos mais usuais prendem-se com a definição de valores constantes que, desta forma, ganham um nome, um significado, assim como uma maior facilidade na sua (eventual) modificação. Por exemplo:

```
#define PI 3.14
#define NUMMAXELEM 30
```

No segundo caso podia significar o número de elementos que se estava a considerar para um dado vector. A declaração, leitura, escrita e cálculo, com os seus correspondentes ciclos seriam escritos todos em função deste valor. A sua alteração seria muito fácil. Por exemplo ter-se-ia:

```
int tabela[NUMMAXELEM];
```

O considerar de um valor mais exacto para π seria também uma questão de alterar uma só linha.

Uma nota: é usual escrever os identificadores associados à definição de macros com todas as letras em maiúsculas. Esta é uma convenção informal na comunidade de programadores em *C*.

A definição:

```
#define VALORABS(a,b) ((a)>(b) ? (a)-(b) : (b)-(a))
```

define uma macro que devolve o valor o valor absoluto da diferença entre os seus argumentos.

Inclusão de Ficheiros

As inclusão de bibliotecas e/ou ficheiros auxiliares é definida do seguinte modo:

```
#include  <<nomeFicheiro>>
#include  "<nomeFicheiro>"
```

No primeiro caso trata-se de inclusão de bibliotecas do sistema, a sua localização concreta varia de acordo com o compilador e/ou sistema operativo usado. A sua utilização está dependente de uma correcta instalação da biblioteca no compilador/sistema operativo em causa.

O segundo caso é usualmente usado para a inclusão de programas auxiliares e/ou bibliotecas definidas pelo próprio programador. O nome do ficheiro pode incluir o «caminho» completo até ao ficheiro.

Ou seja, no primeiro caso a referência ao ficheiro é uma referência relativa, a conversão desta referência relativa para uma localização concreta do ficheiro em questão está a cargo do compilador. No segundo caso estamos a explicitar a localização concreta do ficheiro a incluir.

Compilação Condicional

As directivas de pré-processamento servem também para definir que certas componentes do programa podem ser, ou não, compiladas.

`#if`	<expressãoConstante>	início do condicional
`#ifdef`	<identificador>	se está definido
`#ifndef`	<identificador>	se não está definido
`#elif`	<expressãoConstante>	else if
`#else`	<expressãoConstante>	
`#endif`		fim do condicional

As linhas `if`, `elif` e `else` definem um conjunto de opções do qual uma (e só uma) será correcta e como tal processada, sendo os outros casos ignorados.

Uma das utilizações usuais para este tipo de directivas é dado pela inclusão de opções diferenciadas consoante o compilador e/ou sistema computacional em que a compilação está a ser efectuada.

Outra das utilizações é dada pela salvaguarda de uma tentativa (que falharia) de re-declaração de funções e/ou classes aquando da inclusão de um ficheiro auxiliar o qual, pela divisão «normal» de um programa em componentes, pode ser alvo de inclusão por mais do que um ficheiro e, por essa via, de uma dupla (ou mais) inclusão.

Esta situação é evitada através das seguintes directivas de pré-processamento. Por exemplo para uma dada especificação de Pilhas de Caracteres `pilhaChar.h`.

```
#ifndef PILHAS
#define PILHAS

void pop(pilha p);
char top(pilha p);
(...)

#endif
```

A macro `PILHAS` começa por não estar definida, mas aquando da primeira inclusão `PILHAS` passa a estar definida (o seu valor não é relevante). Numa inclusão posterior toda a definição da classe é ignorada não dado azo a erros de re-definição.

A.1.2 Opções de Compilação

Um qualquer compilador de *C* tem um conjunto extenso de opções que permitem modificar a forma como a compilação se vai processar. No que se segue vai-se apresentar algumas das opções mais usuais para o compilador *GNU* C *Compiler*[1].

[1] `gcc --help`

Opção	Significado
`-o <nomeFicheiro>`	atribuir o nome ao ficheiro executável.
`-c <nomeFicheiro.cpp>`	cria somente o código máquina, isto é, não faz os passos de agregação e criação de um executável.
`-Wall`	(*Warnings all*) Os avisos sobre construções que alguns utilizadores podem achar questionáveis e que são facilmente evitáveis, passam a ser emitidos. Um conjunto alargado de avisos passa a ser emitido.
`-I<directório>`	Adiciona o directório em questão à lista de directórios a serem pesquisados à procura de ficheiro de cabeçalhos (*header files*, `.hpp`).
`-l<biblioteca>`	Pesquisa, para agregação, uma dada biblioteca.
`-L<directório>`	Adiciona o directório em questão à lista de directórios a serem pesquisados à procura de bibliotecas (ver opção `-l`).

A.1.3 Utilizar uma Biblioteca

A crescente modularização na construção de programas, de forma quase natural surge a necessidade de construir não um programa, entidade fechada, mas uma biblioteca (ou pelo menos um nó, «livro» , de uma biblioteca). Designa-se por biblioteca (*library*) um, ou mais módulos, que implementam um dado conjunto de funcionalidades e que podem ser agregadas aos nossos programas. Além da biblioteca padrão do *C*, a *Standard Library* existem muitas outras à nossa disposição. Por exemplo:

GMP *The GNU Multiple Precision Arithmetic Library.* A GMP é uma biblioteca livre para a aritmética de precisão e gama de variação arbitrárias. Implementa inteiros com sinal, racionais, e reais. As principais aplicações da biblioteca GMP são: sistemas criptográficos; segurança da Rede; sistemas algébricos, entre outras.

`http://gmplib.org/`.

Boost A biblioteca *Boost* é um esforço colectivo para providenciar uma biblioteca livre de código aberto e verificada pela própria comunidade de utilizadores. Providencia um conjunto alargado de funcionalidades.

`http://www.boost.org/`.

NAG *Numerical Algorithms Group Library*, é uma biblioteca comercial com uma vasta gama de algoritmos matemáticos e estatísticos.

`www.nag.co.uk/numeric/CL/CLdescription.asp`.

GTK+ *the GIMP Toolkit*, é uma biblioteca, multi-plataforma para a criação de interfaces gráficos.

`http://www.gtk.org/`.

Além destas existem muitas outras com diferentes graus de aplicabilidade, funcionalidade e qualidade.

Como é que podemos então criar, e usar, a nossa própria biblioteca? Como foi dito acima uma biblioteca é algo que providencia um dado conjunto de funcionalidades e que podemos agregar aos nossos programas (a exemplo do que fazemos com a *STL*).

Como usar

Para podermos usar uma biblioteca temos de:

- ter acesso às declarações dos métodos contidos na biblioteca (*header files*);
- ter acesso ao código, na forma de um ficheiro já pré-compilado;
- adicionar as opções '`-l`', '`-L`' e '`-I`' ao comando de compilação. A primeira destas opções indica-nos o nome da biblioteca (sem o prefixo '`lib`'). A segunda opção indica a directoria aonde se encontra a biblioteca (se não estiver integrada no sistema). A última opção indica-nos aonde se encontram as *header files*.

A opção de compilação '`-l`' refere-se ao nome da biblioteca. É no entanto de notar que, por convenção, os nomes das bibliotecas começam todos por '`lib`', por exemplo `libBibMPI.so`, sendo que o prefixo '`lib`' não faz parte do nome da biblioteca, isto é, ter-se-ia `-lBibMPI`.

As bibliotecas podem ser de dois tipos no que diz respeito à sua utilização.

Bibliotecas Dinâmicas consistem de métodos que são pré-compilados e que são carregadas somente aquando da execução do programa. A sua extensão em sistemas Linux é a extensão '`so`', de objectos partilhados (*shared object*).

Bibliotecas Estáticas consistem de métodos que são pré-compilados e agregados ao programa aquando da compilação deste último. A sua extensão usual em sistemas Linux é a extensão '`a`', de arquivo (`archive`).

A grande vantagem das bibliotecas dinâmicas é a sua separação do programa compilado, só na altura da execução é que são chamadas a intervir. As vantagens desta aproximação são múltiplas:

- O ficheiro (executável) do programa não contém o código da biblioteca, em consequência é de menor dimensão do que no caso contrário (bibliotecas estáticas).
- A biblioteca pode sempre ser actualizada sem que isso signifique uma recompilação do programa que a usa.
- A biblioteca pode ser usada por muitos programas, inclusive simultaneamente.

A desvantagem advêm da dependência do programa (para a sua execução) da biblioteca. Num sistema em que a biblioteca dinâmica não esteja instalada os programas que dependem dela não funcionarão.

Isto é, se se quer providenciar uma solução auto-contida, um programa que para funcionar não necessita de mais nada do sistema. Então deve-se construir e usar bibliotecas estáticas. No caso contrário devem-se usar bibliotecas dinâmicas.

A.2 Makefile

O programa `make` é um utilitário presente em todos os sistemas do tipo *Unix* (*Linux, MacOS, etc.*) assim como em outros sistemas em que se instale um compilador de *C*. Este programa permite automatizar o processo de compilação, sendo possível especificar o que se

pretende através da construção de um ficheiro designado `Makefile`. De seguida apresenta-se, de forma sumária, o programa `make` e a forma como construir o ficheiro `Makefile`. Para uma exposição mais completa consulte (Mecklenburg, 2004; Darcano, 2007), ou aceda ao manual do programa[2].

A.2.1 O Programa `make`

O programa `make` é uma maneira muito conveniente de gerir grandes programas ou grupos de programas. Quando se começa a escrever programas cada vez maiores e visível a diferença de tempo necessário para recompilar esses programas em comparação com programas menores. Por outro lado, normalmente trabalha-se apenas em uma pequena parte do programa (tal como uma simples função que se está depurando), e grande parte do resto do programa permanece inalterada.

O programa `make` ajuda na manutenção desses programas observando quais partes do programa foram mudadas desde a última compilação e recompilando apenas essas partes.

Para isso, é necessário que se escreva uma ≪`Makefile`≫, que é um arquivo de texto responsável por dizer ao programa make "o que fazer" e contém o relacionamento entre os arquivos fonte, objecto e executáveis.

Outras informações úteis colocadas no `Makefile` são as ≪`flags`≫ que precisam ser passados para o compilador e o agregador (*linkador*), como directórios onde encontrar arquivos de cabeçalho (arquivos `.hpp`), com quais bibliotecas o programa deve ser agregado, etc. Isso evita que se precise escrever linhas de comando enormes incluindo essas informações para compilar o programa.

O programa `make` pode ser invocado de duas formas distintas (com '`$>`' a representar a linha de comandos do sistema):

- `$> make`, neste caso o primeiro dos objectivos contidos no ficheiro `Makefile` é executado.
- `$> make <objectivo>`, neste caso especifica-se qual o objectivo que se pretende executar.

Vejamos então como construir um ficheiro `Makefile`.

A.2.2 O Ficheiro `Makefile`

Uma `Makefile` contém essencialmente comentários, ≪macros≫ (regras de substituição) e objectivos (*targets*).

Comentários: Os comentários são delimitados pelo carácter "#" e pelo fim-de-linha respectivo. Por exemplo

```
# Projecto - Cálculo do Factorial
```

[2] `man make`; `http://www.gnu.org/software/make/manual/make.html`

Macros As macros são um simples mecanismo de substituição sintáctica. Permitem ajustar de uma forma simples o processo de compilação a diferentes ambientes de compilação. Permitem também a construção de um ficheiro mais fácil de ler e compreender. Por exemplo.

```
FLAGS  = -lm -Wall
CC     = gcc
```

As macros são especificadas da seguinte forma:

```
<nome> = <valor>
```

sendo que por convenção (usualmente aceite) os nomes das macros são escritos utilizando somente maiúsculas.

Na `Makefile`, expressões da forma `$(<nome>)` ou `${<nome>}` são automaticamente substituídas pelo valor correspondente.

Objectivos Os objectivos determinam a acção a ser efectuada quando se executa o programa `make`. Como foi dito acima se o «chamar» do programa for feito sem argumentos o primeiro dos objectivos é aquele que vai ser processado, caso contrário só o objectivo invocado é que é processado. No caso do objectivo invocado não estar especificado na `Makefile` ocorre uma situação de erro e nada é processado. Por exemplo:

```
all:    usaFactorial

clean:
        -rm usaFactorial *.o
```

o carácter '-' antes do comando `rm` (*remove*, apagar ficheiros) tem como finalidade o forçar a continuação do comando mesmo na eventualidade da ocorrência de um erro, por exemplo um dos ficheiros que se pretende apagar não existir.

Os objectivos são especificados da seguinte forma:

```
<objectivo1> <objectivo2> ... : <dependência1> <dependência2> ...
<espaço_tabular> <comando1>
<espaço_tabular> <comando2>
  ...
```

é de notar que o espaço tabular (tecla *Tab*) é obrigatório.

Caso uma das linhas respeitantes a um dos comandos seja muito comprida pode-se continuar a mesma (para efeitos de facilitar a sua leitura) por tantas linhas quanto o necessário. Neste casos é obrigatório colocar o carácter '\' (linha de continuação) entre no fim das linhas que tenham continuação.

Um tipo de objectivo especial é o assim designado, falsos objectivos (*phony target*). Este tipo de objectivos não está associado a um nome de ficheiro e é usualmente usado para evitar eventuais confusões entre nomes de ficheiros e nomes de objectivos. Por exemplo é usual ter-se:

```
.PHONY : clean all
```

Macros Especiais O programa `make` tem um conjunto muito grande de macros pré-definidas[3] das quais convêm destacar as seguintes:

- `$@`, é o nome do ficheiro a ser produzido (nome do objectivo);
- `$?`, são os nomes dos dependências que foram alteradas;
- `$<`, é o nome do ficheiro que causou a acção;
- `$*`, é o prefixo comum aos ficheiros resultantes e suas dependências.

Exemplos de `Makefiles` Um primeiro exemplo englobando os exemplos usados anteriormente:

```
#Compilador
CC = gcc
#Opções de Compilação
CFLAGS = -Wall

# Falsos objectivos, não estão ligados a nenhum ficheiro
.PHONY: all clean

all: usaFactorial

clean:
        -rm usaFactorial *.o

usaFactorial: usaFactorial.c factorial.o
        ${CC} -o $@ $@.c factorial.o ${CFLAGS}
```

esta `Makefile` refere-se à automatização da construção do projecto de exemplo *factorial*.

Um segundo exemplo pode vê-lo como uma `Makefile` para automatizar a compilação de todos os exercícios, acrescentando objectivos à medida que resolvemos novos exercícios.

```
CC      = gcc

.PHONY : clean all

all:     exer15 exer19

clean:
        -rm *.o exer15 exer19

exer15: exer15.o
        ${CC} -o $@ exer15.o

exer19: exer19.o
        ${CC} -o $@ exer19.o
```

[3]Para saber quais fazer `prompt make -p`

A.3 Comandos para a Consola

Na compilação e execução de programas é algumas vezes necessário recorrer à *consola*, local aonde é possível executar programas sem recurso a um interface gráfico. Eis alguns dos comandos necessários para "navegar" no sistema de ficheiros e colocar a executar os programas.[4]

ls — lista o conteúdo de um directório (*listing*).

ls -la — lista o conteúdo de um directório com um nível de detalhe mais elevado ('l') e mostrando os ficheiros escondidos ('a').

cd *dir* — muda para o directório *dir* (*change directory*).

cd — muda para o directório de base (*home*).

cd .. — muda para o directório acima do actual.

pwd — mostra o directório actual (*print working directory*).

more *ficheiro* — mostra o conteúdo de um ficheiro (de texto).

./executável — executa (coloca a funcionar) o *executável*.

[4]Os comandos descritos referem-se à consola *bash* do sistema operativo *Linux*. Comandos semelhantes encontram-se disponíveis noutras consolas/sistemas operativos.

Apêndice B

Ambientes Integrados de Desenvolvimento

Ambientes Integrados de Desenvolvimento, IDE, sigla formada pelas iniciais de *Integrated Development Environment* são sistemas que procuram responder a algumas das questões importantes que se colocam aquando do desenvolvimento de programas:

- a escrita dos programas através de um editor de textos dedicado;
- a compilação do programa, eventualmente envolvendo mais (muito mais) do que um ficheiro;
- a depuração dos erros;
- a documentação.

Vejamos isso ponto a ponto.

B.1 Editor de Textos Dedicado

Antes de mais é de esclarecer que estamos a falar de um editor de textos e não de um processadores de texto. Estes últimos manipulam o texto acrescentando muita informação necessária ao processamento do texto, nomeadamente todas as questões relacionadas com a formatação, o que é totalmente inapropriado quando se está a querer construir algo que vai ser submetido ao tratamento automático de um compilador.

Como primeira conclusão: não tente usar um processador de textos para a escrita de um qualquer programa. É necessário usar um editor de textos.

Um editor de texto dedicado é um editor (não acrescenta nada ao ficheiro contendo o código do programa) mas que conhece a linguagem que se está a usar. Isto é um editor deste tipo tem um conhecimento da linguagem que passa por conhecer a sua estrutura léxica e gramatical, o que leva a que possa:

- fazer a verificação sintáctica à medida que se escreve. Em geral isso reflecte-se num código de cores, isto é, a atribuição diferentes cores aos tipos de dados, aos identificadores da linguagem, aos comentários, às sequências de caracteres, etc;

- fazer a verificação gramatical à medida que se escreve, o que se reflecte na forma como a indentação (os diferentes níveis de alinhamento do texto) é feita;

Pode parecer pouco mas, dado que um programa não é mais do que um texto numa dada linguagem, texto esse que vai ser processada automaticamente por um programa, o qual é muito pouco, ou mesmo nada, tolerante aos erros sintácticos e gramaticais, é fácil de perceber que toda a ajuda é muito importante.

Quando algo não está na cor que é suposto estar... algo está sintacticamente errado, por exemplo um identificador da linguagem que está escrito de forma errada.

Quando algo não se ajusta (indenta) para a forma correcta... algo está gramaticalmente errado, por exemplo um esquecimento de um ";" no fim de uma instrução.

B.2 Compilação & Makefiles

Desde a compilação simples através da simples escolha de uma opção, até à construção e utilização, com diferentes graus de automatismo, de "makefiles" para a compilação de programas multi-ficheiros.

B.3 Depuração de Erros

Desde a indicação e posicionamento automático, da linha aonde os erros ocorrem, até à ligação com programas específicos para o depuramento de erros.

B.3.1 Documentação

No caso do *C* uma ajuda importante é o de mostrar a estrutura do programa com as suas diferentes funções. Em alguns casos (e em certas linguagens), é mesmo possível gerar parte da documentação de forma automática.

B.4 Alguns IDEs para o *C*

É importante que o ambiente de programação escolhido seja:

- Apropriado (talvez mesmo especializado) para o ambiente de trabalho (leia-se empresa) em que se está.
- Multi-plataforma, isto é, esteja disponível em diferentes plataformas computacionais computador/sistema operativo.
- Multi-linguagem, isto é, que seja capaz de se adaptar aos requisitos das diferentes linguagens de programação.

Obviamente que estes dois requisitos aplicam-se a duas situações bem distintas. O primeiro quando se está a trabalhar numa equipa com hábitos e objectivos bem definidos, aí é importante que, para uma maior eficiência, todos utilizem a mesma plataforma de desenvolvimento.

Quando se tem uma situação mais difusa é importante que utilizemos uma plataforma que melhor se adapte às mudanças de plataforma computacional usada e/ou de linguagem de programação usada.

Duas aproximações que se adaptam a estes requisitos são:

Emacs neste caso estamos perante um editor de textos não exactamente um IDE completo. Dado ser programável (em EmacsLisp) é altamente configurável, o seu modo específico para o *C* é muito bom. Dado não ser um IDE as tarefas de compilação e de depuração de erros não estão integradas. De aprendizagem algo difícil é no entanto um dos (se não o) editor de texto mais flexível e poderoso existente. Multi-plataforma, multi-linguagens de programação, código aberto. `http://www.gnu.org/software/emacs/`.

Geany é um IDE bastante completo, o seu editor, embora não tão poderoso como o (X)emacs é mesmo assim muito bom. A construção de Makefiles não é automática, a ligação a um depurador de erros é possível embora necessite configuração. Muito fácil de usar. Multi-plataforma, multi-linguagens de programação, código aberto. `http://www.geany.org/`.

Apêndice C

Folhas práticas

C.1 Comandos para a Consola

ls — lista o contendo de um directório (*listing*).

ls -la — lista o conteúdo de um directório com um nível de detalhe mais elevado ('l') e mostrando os ficheiros escondidos ('a').

cd *dir* — muda para o directório *dir* (*change directory*).

cd — muda para o directório de base (*home*).

cd .. — muda para o directório acima do actual.

pwd — mostra o directório actual (*print working directory*).

more *ficheiro* — mostra o conteúdo de um ficheiro (de texto).

./executável — executa (coloca a funcionar) o *executável*.

C.2 Projectos/Makefile

1 Operações com fracções — Organize o seu código em ficheiros e construa uma `Makefile` correspondente.

1. Declare fracções como estruturas do tipo `struct` cujos campos são do tipo `integer`;

2. Escreva sub-programas para operar com fracções (ler, escrever, simplificar, somar, multiplicar, dividir, subtrair, calcular potências de fracções).

3. Elabore um programa para escrever os primeiros n termos de uma sucessão associada à *série harmónica*:
$$H = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$$
sob a forma de fracção. Por exemplo, para $n = 4$, a saída do programa deverá ser:

```
1
3/2
11/6
25/12
```

2

Pretende-se implementar *Conjuntos* (de inteiros) através de uma representação como um vector (dinâmico) de inteiros.

Organize o seu código em ficheiros e construa uma `Makefile` correspondente.

1. Escreva sub-programas para:
   - Criar um conjunto;
   - Construir um conjunto (a partir de uma sequência de inteiros a ser «lida» do teclado);
   - Escrever um conjunto;
   - Calcular o cardinal de um conjunto;
   - Calcular o conjunto intersecção de dois conjuntos;
   - Calcular o conjunto reunião de dois conjuntos.
2. Construa um programa para testar os sub-programas implementados.

C.3 Gestão Dinâmica de Memória

3 Um polinómio em x de coeficientes reais pode ser representado por um vector de reais, a posição no vector representa o grau do monómio respectivo, o valor no vector, representa o coeficiente do monómio.

Por exemplo, o polinómio

$$x^8 + 5x^6 - 7x^5 + 6x + 1$$

pode ser representado por

1	6	0	0	0	−7	5	0	1

1. Defina a estrutura (dinâmica) vector de reais.
2. Construa as funções necessárias à manipulação de polinómios (leitura, escrita, inserir monómio, apagar monómio, somar e subtrair polinómios, multiplicação de um polinómio por um escalar).
3. Elabore um programa de chamada que permita testar as funções implementadas.

4 Pelo Teorema Fundamental da Aritmética qualquer número inteiro positivo pode ser decomposto como produto de potências não negativas de números primos, $n = p_1^{n_1} \times p_2^{n_2} \times \ldots \times p_k^{n_k}$. Por exemplo,

$$2268 = 2^2 \times 3^4 \times 7^1.$$

Considere que a representação canónica de um inteiro positivo é definida como uma sequência de pares (primo,expoente):

```
typedef struct parPrimoExpoente {
                    int primo, expoente;
} ParPrimoExpoente;

ParPrimoExpoente *repCanonica;
```

e elabore sub-programas para:

1. Obter a representação canónica de um dado inteiro positivo.
2. Verificar se um dado inteiro na forma canónica é, ou não, um número primo.
3. Calcular o produto de dois números inteiros na forma canónica.
4. Calcular o máximo divisor comum de dois inteiros na forma canónica.
5. Calcular o mínimo múltiplo comum de dois inteiros na forma canónica.

C.4 Matrizes Diagonais

Matrizes diagonais são aquelas em que elementos exteriores à diagonal principal são nulos. Por exemplo:

$$\begin{bmatrix} 3,5 & 0 & 0 \\ 0 & -1,6 & 0 \\ 0 & 0 & 2,5 \end{bmatrix} \quad \text{de um forma genérica} \quad \begin{bmatrix} a_{1,1} & & & \\ & a_{2,2} & & 0 \\ & & \ddots & \\ & 0 & & \ddots \\ & & & & a_{n,n} \end{bmatrix}$$

5 Sugira uma boa representação que tome em consideração que os únicos elementos não nulos são os elementos da diagonal principal.

6 Usando uma representação apropriada implemente sub-programas para fazer a sua soma e multiplicação.

Matrizes Triangulares

Matrizes triangulares são aquelas em que elementos acima ou abaixo da diagonal principal são zero. Matriz triangular superior, $\forall_{i>j}, a_{i,j} = 0$. Matriz triangular inferior, $\forall i < j, a_{ij} = 0$.

$$\begin{bmatrix} a_{1,1} & & & \\ a_{2,1} & a_{2,2} & & 0 \\ \vdots & \vdots & \ddots & \\ a_{n,1} & \dots & a_{n,n-1} & a_{n,n} \end{bmatrix} \quad \text{ou} \quad \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ & a_{2,2} & \dots & a_{2,n} \\ & & \ddots & \vdots \\ & 0 & & a_{n,n} \end{bmatrix}$$

7 Sugira uma boa representação que tome em consideração os elementos abaixo (acima) da diagonal principal são nulos.

8 Usando uma representação apropriada implemente sub-programas para fazer a sua soma e multiplicação.

Matrizes Simétrica e Anti-simétricas

Matrizes simétricas são aquelas coincidem com a sua transposta, ou seja, $A = A^T$. Doutro modo matrizes em que $\forall_{i,j}, a_{i,j} = a_{j,i}$. Matrizes anti-simétricas são aquelas coincidem com o oposto da sua transposta, ou seja, $A = -A^T$. Doutro modo matrizes em que $\forall_{i,j}, a_{i,j} = -a_{j,i}$.

Por exemplo:

$$\begin{bmatrix} a & d & e \\ d & b & f \\ e & f & c \end{bmatrix} \quad \text{e} \quad \begin{bmatrix} 0 & 1 & 2,3 \\ -1 & 0 & 4,2 \\ 2,3 & -4,2 & 0 \end{bmatrix}$$

9 Sugira uma boa representação que tome em consideração a simetria em relação à diagonal principal.

10 Usando uma representação apropriada implemente sub-programas para fazer a sua soma e multiplicação.

Matrizes Esparsas

Matrizes esparsas são aquelas que possuem uma grande quantidade de elementos que valem zero.

Podemos encarar a representação de uma matriz esparsa através de uma tabela de triplas, (linha,coluna,valor).

11 Usando a representação referida acima implemente sub-programas para fazer a sua soma, multiplicação e produto escalar.

C.5 Estruturas Lineares – Listas

12 Defina sequências (listas) de números inteiros como:

1. vectores (dinâmicos), ver Figura C.1—(1);
2. listas simplesmente ligadas, ver Figura C.1—(2);
3. listas duplamente ligadas, ver Figura C.1—(3);
4. listas duplamente ligadas circulares, ver Figura C.1—(4).

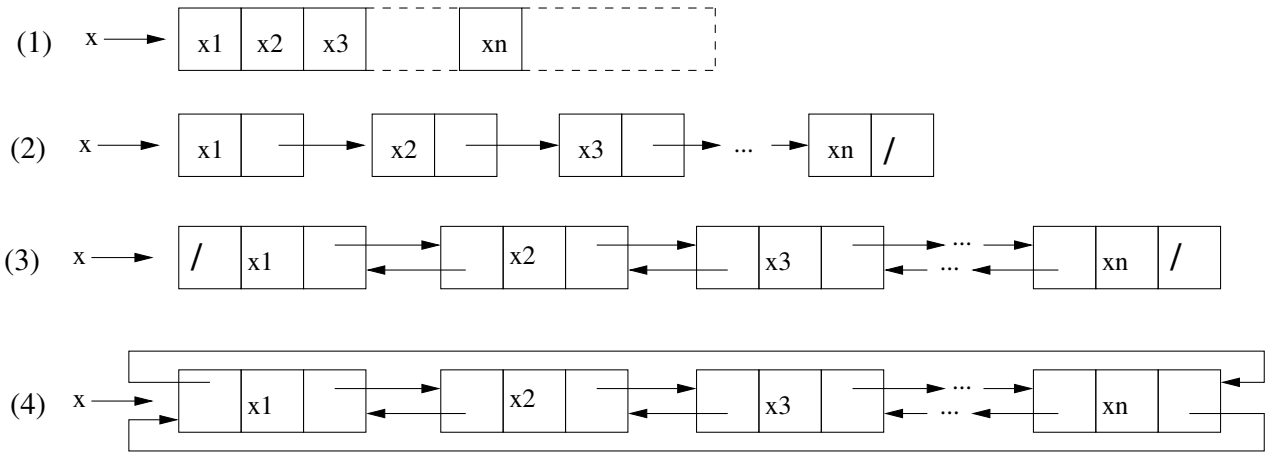


Figura C.1: Implementação de Estruturas Lineares

Implemente as operações seguintes indicando as vantagens / desvantagens de cada uma das representações.

(a) Criar uma lista vazia.

(b) Verificar se uma lista está vazia.

(c) Devolver o n-ésimo elemento, (com $1 \leq n \leq N$), de uma lista de N elementos.

(d) Inserir na n-ésima posição, (com $1 \leq n \leq N$), um dado elemento numa lista de N elementos.

(e) Remover o n-ésimo elemento, (com $1 \leq n \leq N$), de uma lista de N elementos.

(f) Procurar um dado elemento, na lista.

(g) Destruir uma lista.

C.6 Tipos Abstractos de Dados

Um *Tipo Abstracto de Dados* (TAD) é uma estrutura do tipo algébrico formada por: um conjunto de elementos e um conjunto de operações internas sobre esses elementos.

Por exemplo o tipo básico '`int`' pode ser caracterizado como o *TAD Inteiros* da seguinte forma:

Inteiros = (`int`, {zero, um, soma, multiplicação, diferença, divisão, simétrico})

13 Sequências de Caracteres ("Strings"). A linguagem *C* não possuí as "strings" como um tipo básico, elas são implementadas como vectores de caracteres terminadas pelo carácter especial '`\0`'.

Temos que, em *C*, '`x`', é uma constante do tipo `char`, "`x`", é uma constante do "tipo string", a qual é guardado internamente como | 'x' | '\0' |.

Implemente o *TAD Palavras* definido da seguinte forma:

Palavras = (`char*`,{vazia, ler, escrever, comprimento, atribuir, concatenar, sub-palavra, compara})

Em que: *sub-palavra* nos diz se uma dada palavra está, ou não, contida numa outra; *compara* deve ter como resultado, para duas dadas palavras, menor, igual ou maior.

14 TAD Fraccionais.

1. Construa o TAD Fraccionais:
   (a) Declare fracções como estruturas do tipo `struct` cujos campos são do tipo `int`;
   (b) Escreva sub-programas para operar com fracções (ler, escrever, simplificar, somar, multiplicar, dividir, subtrair, calcular potências de fracções).
2. Elabore uma calculadora de fraccionários.

15 TAD Complexos.

1. Construa o TAD Complexos:
   (a) Declare complexos como estruturas do tipo `struct` cujos campos são do tipo `double`.
   (b) Escreva sub-programas para operar com complexos (ler, escrever, conjugado, inverso multiplicativo, somar, multiplicar, dividir, subtrair).
2. Elabore uma calculadora de complexos.

16 TAD Conjuntos (de inteiros).

1. Construa o TAD Conjuntos:
   (a) Declare conjuntos como estruturas do tipo `struct` cujos campos são do tipo `int` (número de elementos) e `int *` (elementos).
   (b) Escreva sub-programas para: criar o conjunto vazio; criar um conjunto singular; a relação de pertença; cardinal de um conjunto; intersecção de conjuntos; reunião de conjuntos; a diferença de conjuntos; ler um conjunto; escrever um conjunto;
2. Construa um programa para testar os sub-programas implementados.

17 Construa o TAD Conjuntos, tal como na pergunta anterior, mas com uma modelização dos conjuntos através de uma lista simplemente ligada.

C.7 Tipos Abstractos de Dados Pilha

Pilhas são sequências lineares de elementos com acesso por um só ponto, o topo. Implementam a disciplina de acesso *Last In First Out (LIFO)*, o último a entrar é o primeiro a sair.

Considerando a operação de sequenciação, ':', colocar um elemento e um sequência, temos a seguinte definição para o TAD Pilha.

$$Pilha = (\{pilhaVazia, elemento\colon Pilha\}, \{\text{cria, push, pop, top, vazia?}\})$$

18 Implemente o TAD Pilha.

- Defina o tipo Pilha, baseando a implementação numa lista (simplesmente) ligada.
- Defina as operações internas como funções no novo tipo.

Escreva ficheiros de definição de funções (*header*, .h) e ficheiros de implementação das funções (.c).

19 Re-implemente o TAD Pilha baseando a implementação do novo tipo em vectores dinâmicos.

Conceba um conjunto de testes que permita avaliar as vantagens e desvantagens de cada uma das implementações.

20 Elabore um sub-programa que use uma estrutura ligada do tipo `Pilha` para ler uma palavra, terminada por um espaço, e escrever as suas letras por ordem inversa.

21 Ao contrário da notação usual (*infixa*), na notação polaca (*pós-fixa*) cada operador é colocado depois dos dois operandos correspondentes.

As expressões seguintes são exemplos de expressões equivalentes nas duas notações:

$2 * (30 + 4)$	$2\ 30\ 4\ +\ *$
$(1 + 5) * (8 - (4 - 1))$	$1\ 5\ +\ 8\ 4\ 1\ -\ -\ *$
$23 + 4 * 5$	$23\ 4\ 5\ *\ +$

Escreva um sub-programa que leia uma expressão (em números inteiros) em notação polaca e, utilizando uma `Pilha`, avalie-a.

C.8 Tipos Abstractos de Dados Filas

Filas são sequências de elementos com acesso apenas pelas extremidades, a entrada e a saída. Implementam a disciplina *First In First Out (FIFO)*, o primeiro a entrar é o primeiro a sair.

Considerando a operação de sequenciação, ':', colocar um elemento e um sequência, temos a seguinte definição para o TAD Fila.

$$Fila = (\{filaVazia, elemento\colon Fila\}, \{\text{cria, insere, retira, topo, vazia?}\})$$

22 Implemente o TAD Fila.

- Defina o tipo Fila, baseando a implementação numa lista (simplesmente) ligada.
- Defina as operações internas como uma funções no novo tipo.

Escreva ficheiros de definição de funções (*header*, .h) e ficheiros de implementação das funções (.c).

23 A estrutura fila é muito usadas na implementação de programas de simulação de filas de espera. Por exemplo, as caixas num supermercado, as filas de carros num semáforo, as filas numa cantina, etc.

Utilizando o TAD Fila de inteiros, escreva um programa que comece por preencher uma fila com 200 valores aleatórios (funções `srand` e `rand`) e que de seguida calcule o valor mínimo, o valor máximo e o valor médio dos elementos da fila.

24 Re-implemente o TAD Fila baseando a implementação do tipo *vectores dinâmicos.*

Conceba um conjunto de testes que permita avaliar as vantagens e desvantagens de cada uma das implementações.

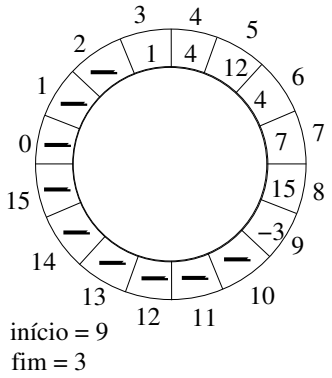
25 A implementação de *lista circular simplesmente ligada* é a mais apropriada para o TAD fila.

Re-implemente o TAD Fila baseando a implementação numa *lista circular simplesmente ligada.*

Conceba um conjunto de testes que permita avaliar as vantagens e desvantagens de cada uma das implementações.

26

É possivel implementar de forma eficiente o conceito de lista circular utilizando uma representação contígua.



Re-implemente o TAD Fila baseando a implementação nesta nova configuração.

Conceba um conjunto de testes que permita avaliar as vantagens e desvantagens de cada uma das implementações.

27 Deque (double ended queue) – é uma versão generalizada da estrutura de dados de fila que permite inserir e excluir em ambas as extremidades.

Operações do TAD Deque

- criar: criar um novo *deque*;
- inserirInicio: Adiciona um elemento ao início do *deque*;
- inserirFim: Adiciona um elemento no fim *deque*;
- apagarInicio: Apaga um elemento no início do *deque*;

- apagarFim: Apaga um elemento no fim do *deque*;
- verInicio: Obtém o elemento no início do *deque*;
- verFim: Obtém o último elemento do *deque*;
- estaVazia: Verifica se o *deque* está vazio, ou não.

1. Discuta qual será a melhor forma de representar esta nova estrutura. Implemente-a.
2. Implemente as operações que conjuntamente com a nova estrutura definem o TAD *deque*.

28 Considere um dado percurso circular, no qual existem n bombas de gasolina. Pretende-se encontrar o primeiro percurso circular, feito por um dado camião, que visita todas as bombas de gasolina.

Existem dois conjuntos de dados.

- A quantidade de gasolina que todas as bombas de gasolina têm.
- Distância da bomba de gasolina para a próxima bomba de gasolina.

Usando filas como estrutura de dados, calcule o primeiro ponto de onde um camião será capaz de completar o círculo (o camião vai parar em cada bomba de gasolina e tem capacidade infinita). Suponha que para 1 litro de gasolina, o camião pode ir 1 unidade de distância.

Por exemplo, considere que existem 4 bombas de gasolina. A quantidade de combustível e distância entre bombas de gasolina é dada pelos seguintes pares de valores $(4, 6), (6, 5), (7, 3)$ e $(4, 5)$. O primeiro ponto de onde o camião pode fazer um percurso circular é a segunda bomba de gasolina. A saída deve ser «Início = 2ª bomba de gasolina».

C.9 Tipos Abstractos de Dados Listas

Listas são sequências de elementos com acesso ilimitado, é possível aceder a cada um dos elementos de forma independente.

Considerando a operação de sequenciação, ':', colocar um elemento em uma sequência, temos a seguinte definição para o TAD Lista.

$$\text{Lista} = (\{listaVazia, elemento{:}lista\}, \{\text{cria, insereN, retiraN, veN, vazia?}\})$$

29 Implemente o TAD Lista.

- Defina o tipo Lista, baseando a implementação numa lista (simplesmente) ligada.
- Defina as operações internas como uma funções no novo tipo.

Escreva ficheiros de definição de funções (*header*, .h) e ficheiros de implementação das funções (.c).

30 Re-implemente o TAD Lista baseando a implementação do novo tipo em vectores dinâmicos.

Conceba um conjunto de testes que permita avaliar as vantagens e desvantagens de cada uma das implementações.

31 Re-implemente o TAD Lista baseando a implementação do novo tipo em listas duplamente ligadas.

Conceba um conjunto de testes que permita avaliar as vantagens e desvantagens de cada uma das implementações.

32 Escreva uma função que conte o número de vezes que um determinado elemento ocorre em uma lista.

33 Inverter a ordem de uma lista ligada.

Dado o ponteiro para o nó inicial de uma lista ligada, inverta a lista ligada, invertendo as ligações entre nós.

34 Adicione ao TAD Lista a implementação do algoritmo de *fusão*. Dados duas listas ordenadas, fundir as duas de modo ordenado.

Discuta as vantagens/desvantagens de uma implementação usando o TAD Lista versus uma implementação de listas, especifica para o problema em questão.

35 Implemente o método de ordenação *Borbulagem* (≪Bubble Sort≫).

- Percorra a lista trocando pares de elementos que estejam fora de ordem.
- Repita o processo até que, numa das passagens anteriores, não se proceda a nenhuma troca.

36 Implemente o método de ordenação *Selecção Linear* (≪Linear Sort≫).

- Encontre o maior dos n elementos;
- Coloque esse elemento na sua posição final trocando-o com o enésimo elemento;
- Recursivamente ordene $n - 1$ primeiras posições da lista.

37 Implemente o método de ordenação *Inserção Ordenada* (≪Insertion Sort≫).

- Consideramos a lista como a concatenação de duas sequências: uma ordenada, e a segunda não ordenada. No início a primeira sequência contém apenas um elemento.
- O primeiro elemento da sequência não ordenada é inserido na posição correcta da sequência ordenada.
- Esta operação aumentou a sequência ordenada de um elemento e retira esse elemento à sequência não ordenada.
- Repetimos até todos os elementos estarem na sequência ordenada.

38 Implemente o método de ordenação *Fusão* (≪Merge Sort≫).

Considerem-se duas listas ordenadas. A sua fusão numa só lista ordenada é possível seguindo os seguintes passos:

- Comparam-se os dois elementos iniciais das duas listas.
- Escolhe-se a lista que contêm o elemento menor e copiam-se os seus elementos para a lista final até que o elemento que se está a considerar já seja maior do que o elemento inicial da outra lista.
- troca-se de listas e repete-se o processo, até que se chega ao fim de uma das listas.
- copiam-se os restantes elementos da outra lista para a lista final.

O processo precisa sempre de uma lista auxiliar, sobre o qual se vai construir a lista ordenada.

Pode-se aplicar este método para ordenar uma lista genérica. Basta considerar a divisão recursiva em metades de uma dada lista, tendo como caso de base, as listas com um só elemento (uma lista com um só elemento está ordenada).

39 Implemente o método de ordenação (≪Quick Sort≫).

Dado uma lista de n inteiros e um inteiro a existente na lista pretende-se particionar essa lista em duas partes: os elementos inferiores a a e os elementos superiores a a.

- Começando do lado esquerdo da lista procurar um elemento que seja maior ou igual a a.
- Começando do lado direito da lista procurar um elemento que seja menor ou igual a a.
- Trocar estes dois elementos.
- Continuar até os dois processos de pesquisa se cruzarem.

40 Re-implemente o exercício C.3, referente aos polinómios em x, usando o TAD lista.

C.10 Tipos Abstractos de Dados Árvore Binária

41 Implementação do TAD Árvore Binária.

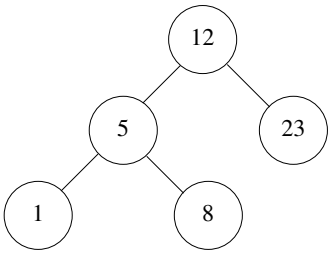
$$\mathrm{AB} = (\{Abvazia, ABesq\!:\!elemento\!:\!ABdir\}, \{\text{criar,destruir,inserir,} \\ \text{retirar,procuraElemento,vazia?}\})$$

considerando que a inserção é feita de forma a criar uma árvore binária de pesquisa, isto é, à esquerda do elemento na raiz, todos os elementos são menores do que a raiz e à direita, todos os elementos são maiores ou iguais à raiz. Implemente o TAD Árvore Binária (de pesquisa).

42 Travessias de árvores

Ao contrário das estruturas de dados lineares (vectores, listas, filas, pilhas, etc.), que possuem apenas uma maneira lógica de serem precorridas, as árvores podem ser percorridas de várias formas diferentes.

As formas mais usuais para travessias de árvores são: em-ordem, pré-ordem, pós-ordem, travessia em largura.



- Travessias em profundidade:
  - Em-ordem (Esquerda, Raiz, Direita): `1 5 8 12 23`;
  - Pré-ordem (Raiz, Esquerda, Direita): `12 5 1 8 23`;
  - Pós-ordem (esquerda, direita, raiz): `1 8 5 23 12`.
- Travessia na largura (por níveis): `12 5 23 1 8`.

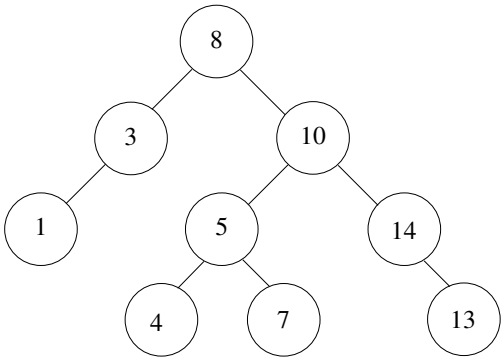
Implemente as diferentes travessias em profundidade.

43 Seja `ABI` o tipo abstracto de dados *Árvore Binária de Inteiros*, em cujos nós se armazena um número inteiro e assuma que os números inteiros não se encontram registados em nenhuma ordem particular.

Elabore sub-programas para:

1. Escrever os elementos registados numa árvore binária *em-ordem* (*pós-ordem* / *pré-ordem*).
2. Contar o número de nós de uma árvore binária.
3. Listar os elementos registados nas folhas de uma árvore.
4. Contar o número de folhas de uma árvore binária.
5. Contar os nós internos de uma árvore binária.
6. Obter um ponteiro para o nó que contém o maior elemento registado numa árvore.
7. Calcular a altura de uma árvore.
8. Construir uma árvore binária completa com profundidade n, sendo os nós gerados recursivamente utilizando pré-ordem.

44 Árvores de pesquisa é uma árvore binária em que, para cada nó, temos as seguintes propriedades:



- A sub-árvore esquerda de um nó contém apenas nós com valores menores que o valor do nó.
- A sub-árvore direita de um nó contém apenas nós com valores maiores ou iguais que o valor do nó.
- As sub-árvores esquerda e direita também devem ser uma árvore binária de pesquisa.

1. Implemente uma função de construção de uma árvore binária de pesquisa a partir de uma lista de elementos.
2. Implemente uma função de pesquisa binária utilizando a estrutura de árvore binária de pesquisa.

45 Seja `ABI` o tipo abstracto de dados *Árvore Binária de Inteiros*, em cujos nós se armazena um número inteiro e assuma que os números inteiros não se encontram registados em nenhuma ordem particular.

Elabore sub-programas para:

1. Dados uma árvore e dois inteiros, a, b, verificar se b é **descendente** de a, isto é, se b pertence a uma das sub-árvores do nó que contém a. NOTA: Deve ser contemplada a possibilidade de a e/ou b não pertencerem à árvore dada.
2. Dado uma árvore binária e um inteiro, escreva todos os nós descendentes (ascendentes) do nó correspondente ao inteiro dado.

46 O *percurso em largura* de uma árvore binária consiste em percorrer os nós de uma árvore por níveis, desde o mais alto (raiz) até à base e da esquerda para a direita.

Por exemplo, para a árvore da figura C.2 ter-se-ia `A B O D F G E M C`.

Elabore um sub-programa para fazer uma travessia em largura.

Sugestão: utilize uma lista linear para indicar as sub-árvores ainda a serem percorridas.

47 A igualdade entre árvores binárias pode ser entendida como uma igualdade entre listas de elementos após a travessia das duas árvores binárias (por exemplo, em ordem). Pode também ser entendidas como uma igualdade estrutural, caso em que se pretende que além de elementos iguais as árvores tenham uma estrutura igual.

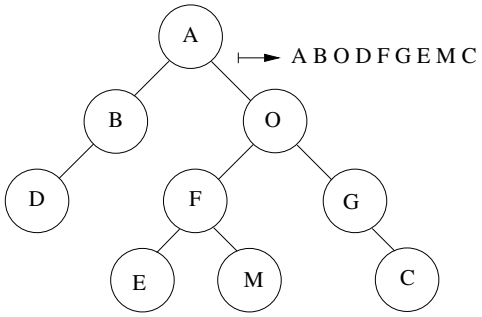


Figura C.2: Árvore Binária — Percurso em Largura

Temos então as seguintes definições:

Igualdade nos elementos – dados duas árvores binárias, ab_1 e ab_2 elas dizem-se iguais (nos elementos) se travessia(ab_1) = travessia(ab_2).

Equivalência – dados duas árvores binárias, ab_1 e ab_2 elas dizem-se equivalentes, eq(ab_1,ab_2), se: ambas as raízes forem vazias ou obtemRaiz(ab_1) = obtemRaiz(ab_2) e eq(obtemEsq(ab_1),obtemEsq(ab_2)) e eq(obtemDir(ab_1),obtemDir(ab_2)).

Elabore sub-programas para:

1. Determinar se duas árvores binárias são iguais (nos elementos).

2. Determinar se duas árvores binárias são equivalentes.

48 Dadas duas árvores binárias, ab_1 e ab_2 diz-se que ab_1 está contida em ab_2 se ab_1 é equivalente a ab_2 ou então existe pelo menos uma sub-árvores de ab_2 que seja equivalente a ab_1.

Elabore um sub-programa para:

1. Determinar se uma árvores binárias ab_1 está contida numa outra árvore binária ab_2.

49 Pretende-se guardar a informação sobre as diferentes unidades de ensino de uma dada universidade, preservando a estrutura hierárquica da mesma. Por exemplo, na figura C.3, tem-se uma representação parcial da estrutura da Universidade de Coimbra.

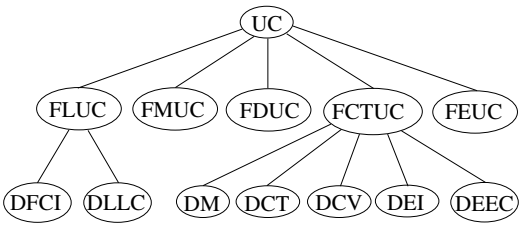


Figura C.3: Estrutura Hierárquica — Árvore Genérica

Elabore sub-programas para:

1. Inserir a informação, na forma de uma árvore binária (sem que haja perda de informação quanto à estrutura hierárquica).

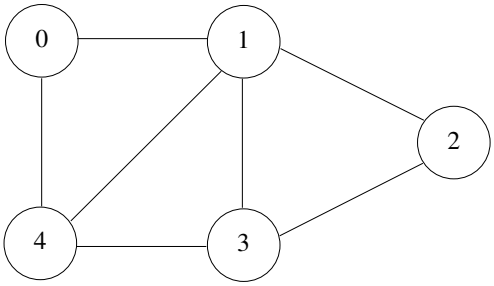
2. Que, para uma dada faculdade, liste todos os seus departamentos.

3. Que liste todas as faculdade, conjuntamente com os respectivos departamentos.

C.11 Tipos Abstractos de Dados Grafo

Um Grafo é uma estrutura de dados não linear que consiste em nós e arcos. Os nós são às vezes também referidos como vértices e as arcos, arestas. Os arcos ligam quaisquer dois nós no grafo.

Um grafo pode ser definido como: um conjunto finito de nós (vértices) e um conjunto finito de arcos (arestas) que ligam pares de nós.



No gráfico acima, o grafo é definido através dos conjuntos $N = \{0, 1, 2, 3, 4\}$, os nós do grafo, e o conjunto $E = \{(0, 1), (1, 2), (2, 3), (3, 4), (0, 4), (1, 4), (1, 3)\}$, arcos.

Os grafos são usados para resolver muitos problemas da vida real. Grafos são usados para representar redes. As redes podem incluir caminhos em uma cidade ou rede telefônica ou rede de circuitos. Os grafos são também usados em redes sociais como *linkedIn*, *Facebook*. Por exemplo, no *Facebook*, cada pessoa é representada com um nó. Cada nó é uma estrutura que contém a informação referente a cada pessoa.

50 T.A.D. Grafos = $\{(\{\text{Nós}\}, \{\text{Arcos}\}), (\{\text{criarGrafo, inserirNo, inserirArco, retirarNo, retirarArco}\})\}$

1. Implemente o T.A.D. Grafo usando para tal a representação matriz de adjacências.

2. Implemente o T.A.D. Grafo usando para tal a representação lista de adjacências.

51 Percursos completos em grafos.

1. Implemente o Percurso em Amplitude.

2. Implemente o Percurso em Profundidade.

52 Caminho Mais Curto

1. Implemente o algoritmo de Dijsktra para achar o caminho mais curto de uma dada rede de transportes.

C.12 Exercícios Diversos

53 Escreva sub-programas para retirar e devolver um ponteiro para o "maior" elemento de uma dada lista.

54 Escreva sub-programas para ordenar uma dada lista por ordem não decrescente,

1. usando selecção linear;
2. usando inserção linear.

55 Fazer a fusão ordenada de duas listas previamente ordenadas:

1. devolvendo uma nova lista;
2. devolvendo uma das listas dadas depois de actualizada.

56 Ao chegar à oficina de automóveis "AutoX" cada cliente dirige-se à recepção, onde preenche uma ficha, que além de conter a informação relativa ao automóvel serve para determinar a ordem de atendimento. Considere que cada cliente é definido pelos tipos de dados: nome; marca, modelo e matrícula do carro; tipo de avaria.

Com base numa estrutura ligada do tipo `Fila`, elabore sub-programas para realizar as seguintes operações:

1. Inserir um novo cliente na fila.
2. Atender um cliente na fila.