

Capítulo 17

Estruturas genéricas

Nos capítulos anteriores, apresentamos as estruturas de dados básicas utilizadas para a organização de informações na memória do computador. Na apresentação das estruturas e das principais funções de acesso, consideramos que a informação associada a cada nó era representada por um tipo simples. Desta forma, foi possível focar a discussão na estrutura em si, já que o tratamento do dado associado era trivial. Todas as estruturas que vimos e todas as funções que discutimos podem ser aplicadas aos casos em que precisamos armazenar informações mais complexas, representadas por estruturas. De fato, já foram apresentadas técnicas de programação que podem ser utilizadas para a representação de informações estruturadas, usando a própria estrutura como sendo a informação ou usando um ponteiro para a estrutura como sendo a informação.

No entanto, para cada novo tipo que quiséssemos tratar, tínhamos que reimplementar as funções que manipulam a estrutura. Como já discutimos, as funções mantêm-se praticamente idênticas, sendo necessário apenas modificar o tratamento das informações associadas a cada nó. Podemos então pensar em construir estruturas de dados genéricas, isto é, estruturas de dados capazes de armazenar qualquer tipo de informação. Para tanto, o tipo abstrato de dado (TAD) deve desconhecer a natureza da informação associada, sendo responsável apenas pela manutenção e organização na estrutura.

O cliente de um TAD de tipo genérico fica responsável por todas as operações que envolvam o acesso direto às informações. Internamente, o TAD guarda apenas um ponteiro para a informação. Esse ponteiro deve ser do tipo genérico, pois não se sabe, a princípio, o tipo da informação que será armazenada. Como já vimos, um ponteiro genérico em C é representado pelo tipo `void*`. Assim, o TAD, de posse de um ponteiro genérico, não pode acessar a memória por ele apontada, já que não conhece a informação armazenada. Por sua vez, o cliente pode converter esse ponteiro genérico no ponteiro específico para o tipo em questão e, então, acessar os dados do tipo.

17.1 Lista genérica

Vamos primeiro exemplificar a implementação de um TAD de tipo genérico, usando uma lista simplesmente encadeada. As mesmas técnicas de programação podem ser aplicadas às demais estruturas de dados.

A estrutura do nó de uma lista genérica tem que guardar o ponteiro para a informação e o ponteiro para o próximo nó. O código a seguir ilustra essa representação:

```

typedef struct listagen ListaGen;
typedef struct listagenno ListaGenNo;

struct listagenno {
    void* info;
    ListaGenNo* prox;
};

struct listagen {
    ListaGenNo* prim;
};

```

Funções que não manipulam a informação associada aos nós podem ser implementadas da forma como já vimos. Por exemplo, a função para criar uma lista vazia é naturalmente idêntica à função já apresentada para listas de valores inteiros.

Funções que manipulam a informação como um objeto opaco, isto é, funções que não precisam acessar as informações, não oferecem dificuldades para serem implementadas. Por exemplo, podemos implementar uma função que insere um novo nó no início da lista. O cliente é responsável por passar para a função o ponteiro da informação que será armazenada nesse novo nó. Portanto, a função para inserção não precisa acessar a informação, basta fazer o novo nó ter como informação o ponteiro passado pelo cliente. A implementação dessa função é similar ao caso da lista de inteiros, variando apenas o tipo do parâmetro passado (que agora é um ponteiro genérico).

```

void lgen_insere (ListaGen* l, void* p)
{
    ListaGenNo* n = (ListaGenNo*) malloc(sizeof(ListaGenNo));
    n->info = p;
    n->prox = l->prim;
    l->prim = n;
}

```

A dificuldade aparece quando temos que implementar as funções que precisam ter acesso às informações dos nós. Por exemplo, como podemos implementar uma função que libera a estrutura? Se quisermos liberar a estrutura da lista, também podemos querer liberar as informações associadas aos nós ou não. No entanto, como o TAD desconhece as informações, o cliente é quem deve ser responsável por decidir e, se for o caso, liberá-las. O TAD deve ficar responsável apenas por liberar a estrutura de dados em si.

Uma situação similar é observada na implementação da função que verifica se determinada informação está presente na lista. O TAD não é capaz de testar a igualdade entre duas informações. Apenas comparar os ponteiros não resolve, pois devemos comparar as informações propriamente ditas. O cliente pode, por exemplo, associar aos nós informações dos alunos de uma disciplina, e pode desejar fazer uma busca baseada apenas no nome de um aluno.

Se considerássemos a implementação de uma função para imprimir as informações da lista, teríamos o mesmo problema: apenas o cliente é capaz de acessar as informações e exibi-las na tela. Na verdade, como não sabemos *a priori* o tipo de informação que será associado aos nós, não podemos nem prever quais funções deverão ser oferecidas na interface do TAD. Cada cliente irá associar um tipo de informação diferente e precisará

de funções específicas para processar as informações armazenadas na estrutura. Por exemplo, um cliente que armazena pontos geométricos pode precisar de uma função para calcular o centro geométrico dos pontos armazenados. Outro cliente que armazena dados relativos aos alunos de uma disciplina pode precisar de uma função para calcular a média obtida numa prova.

Portanto, o TAD deve prover uma função genérica para percorrer todos os nós da estrutura. Para cada nó visitado, devemos implementar um mecanismo que nos permita chamar o cliente passando a informação associada. O cliente então processa a informação com finalidades específicas para cada situação.

17.2 Uso de callbacks

Nosso objetivo é implementar uma função que percorre todos os elementos armazenados na estrutura genérica. Para tanto, devemos separar a função que percorre os elementos da ação que será realizada em cada elemento. Assim, a função que percorre os elementos é única e pode ser usada para diversos fins. A ação que deve ser executada é passada como parâmetro. Essa ação é, geralmente, chamada de *callback*, pois é uma função do cliente (quem usa a função que percorre os elementos) que é “chamada de volta ao cliente” a cada elemento encontrado na estrutura de dados. Usualmente, essa função *callback* recebe como parâmetro a informação associada ao elemento encontrado na estrutura. No nosso exemplo, como temos uma lista genérica, a função recebe o ponteiro para cada informação encontrada na lista.

Para definir como parâmetro qual função *callback* deve ser chamada, temos que usar o conceito de *ponteiro para função*. O nome de uma função representa o endereço dessa função. A nossa função *callback* pode ter, por exemplo, a seguinte assinatura:

```
void callback (void* info);
```

Uma variável ponteiro para armazenar o endereço dessa função é declarada como:

```
void (*cb) (void*);
```

onde *cb* representa uma variável do tipo ponteiro para funções com a mesma assinatura da função *callback* anterior.

Assim, uma função genérica para percorrer os elementos da lista do nosso exemplo pode ser dada por:

```
void lgen_percorre (ListaGen* l, void (*cb)(void*)) {
    for (ListaGenNo* p=l->prim; p!=NULL; p=p->prox) {
        cb(p->info);
    }
}
```

Isto é, para cada elemento visitado, chama-se a função do cliente passando como parâmetro a informação associada.

17.2.1 Um exemplo de cliente

Para ilustrar a utilização da lista genérica, vamos considerar uma aplicação cliente que armazena pontos (x, y) na estrutura. O tipo `Ponto` pode ser definido por:

```
typedef struct ponto Ponto;
struct ponto {
    float x, y;
};
```

Para inserir pontos na lista genérica, o cliente aloca dinamicamente uma estrutura do tipo `Ponto` e passa seu ponteiro para a função de inserção. Para encapsular esse procedimento, o cliente pode implementar uma função auxiliar para inserir pontos (x, y) na estrutura da lista genérica.

```
static void insere_ponto (ListaGen* l, float x, float y)
{
    Ponto* p = (Ponto*) malloc(sizeof(Ponto));
    p->x = x;
    p->y = y;
    lgen_insere(l, p);
}
```

Uma vez construída a lista genérica, podemos pensar em imprimir os pontos armazenados na estrutura. Para tanto, vamos fazer uso da função `percorre` discutida. A cada elemento visitado, vamos imprimir as coordenadas do ponto associado. Nossa função *callback* então é responsável por converter o ponteiro genérico em um ponteiro para `Ponto` e imprimir a informação. Uma possível implementação dessa *callback* é mostrada a seguir. Devemos notar que o protótipo da função *callback* é fixo e independe da informação. A conversão para o tipo específico ocorre dentro da função.

```
static void imprime (void* info)
{
    Ponto* p = (Ponto*)info;
    printf("%f %f\n", p->x, p->y);
}
```

Com isso, para imprimir os elementos da lista bastaria chamar a função `percorre` com a ação `imprime` passada como parâmetro:

```
...
lgen_percorre(l, imprime);
...
```

Essa mesma função `percorre` pode ser usada para, por exemplo, calcular o centro geométrico dos pontos armazenados na lista. A ação associada aqui precisa apenas contar o número de elementos visitados e acumular os valores das coordenadas dos pontos visitados. Para tanto, podemos usar variáveis globais para representar o número de elementos e o somatório das coordenadas. A cada chamada da *callback*, os valores devem

ser atualizados. Assumindo que `NP` e `CG` são variáveis globais do tipo `int` e `Ponto`, respectivamente, a ação para acumular a soma das coordenadas dos elementos da lista pode ser simplesmente:

```
static void centro_geom (void* info)
{
    Ponto* p = (Ponto*)info;
    CG.x += p->x;
    CG.y += p->y;
    NP++;
}
```

De posse dessa *callback*, o cliente pode calcular o centro geométrico dos pontos:

```
...
NP = 0;
CG.x = CG.y = 0.0f;
lgen_percorre(l,centro_geom);
CG.x /= NP;
CG.y /= NP;
...
```

17.2.2 Passando dados para a *callback*

Já mencionamos que o uso de variáveis globais deve, sempre que possível, ser evitado, pois seu uso indiscriminado torna um programa ilegível e difícil de ser mantido. Para evitar o uso de variáveis globais, devemos criar um mecanismo para transferir um dado do cliente para a função *callback*. A função que percorre os elementos não manipula esse dado, apenas o transfere para a função *callback*. Como não sabemos *a priori* o tipo de dado necessário, definimos a *callback* recebendo dois parâmetros: a informação do elemento sendo visitado e um ponteiro genérico para um dado qualquer. O cliente chama a função que percorre os elementos passando como parâmetros a função *callback* e o ponteiro a ser repassado para essa mesma *callback* a cada elemento visitado.

Vamos exemplificar o uso dessa estratégia reimplementando a função que percorre os elementos.

```
void lgen_percorre(ListaGen* l, void(*cb)(void*,void*), void* dado)
{
    for (ListaGenNo* p=l->prim; p!=NULL; p=p->prox) {
        cb(p->info,dado);
    }
}
```

Devemos notar que a assinatura da função *callback* foi alterada, pois agora ela recebe dois parâmetros. Podemos usar essa nova versão da função *percorre* para calcular o centro geométrico dos pontos, sem usar variáveis globais. Primeiro, temos que criar um tipo que agrupa os dados que precisamos para calcular o centro geométrico: o número de pontos e as coordenadas acumuladas.

```

typedef struct cg Cg;
struct cg
{
    int n;
    Ponto p;
};

```

Podemos então redefinir a *callback*, que, nesse caso, receberá um ponteiro para um tipo *Cg* que representa a estrutura anterior.

```

static void centro_geom (void* info, void* dado)
{
    Ponto* p = (Ponto*)info;
    Cg* cg = (Cg*)dado;
    cg->p.x += p->x;
    cg->p.y += p->y;
    cg->n++;
}

```

Desta forma, a chamada por parte do cliente pode ser exemplificada pelo trecho de código a seguir, sem uso de variáveis globais:

```

...
Cg cg = {0,{0.0f,0.0f}};
lgen_percorre(l,centro_geom,&cg);
cg.p.x /= cg.n;
cg.p.y /= cg.n;
...

```

17.2.3 Retornando valores de *callbacks*

Vamos agora considerar que queremos verificar a ocorrência de determinado ponto de coordenadas (*x*, *y*) na estrutura da lista. Nesse caso, nossa função *callback* pode receber como dado adicional as coordenadas do ponto que queremos encontrar. No entanto, da forma que a função *percorre* está implementada, todos os elementos da lista serão visitados, mesmo se encontrarmos o ponto de interesse entre os primeiros elementos da lista.

Para evitar esse esforço computacional desnecessário, devemos criar um mecanismo para permitir ao cliente interromper a visitação aos elementos. Esse mecanismo pode ser implementado fazendo com que a função *callback* tenha um valor de retorno. Podemos, por exemplo, adotar a seguinte convenção: se a *callback* tiver zero como valor de retorno, a função deve prosseguir visitando o próximo elemento; se a *callback* tiver um valor diferente de zero como retorno, a função *percorre* deve interromper a visitação aos elementos e ter esse valor retornado pela *callback* como seu valor de retorno. Portanto, a assinatura da função *percorre* também muda, pois passa a ter um valor de retorno: zero, se não houve interrupção, e diferente de zero, se houve interrupção por parte do cliente. Uma possível implementação dessa nova versão da função *percorre* é mostrada a seguir:

```

int lgen_percorre (ListaGen* l, int (*cb)(void*,void*), void* dado)
{
    for (ListaGenNo* p=l->prim; p!=NULL; p=p->prox) {
        int r = cb(p->info,dado);
        if (r != 0)
            return r;
    }
    return 0;
}

```

Se usássemos essa versão de `percorre` para implementar as funções discutidas, teríamos que fazer as *callbacks* retornarem zero. O fato de permitir que a *callback* retorne um valor nos possibilita fazer a busca de um ponto sem precisar continuar percorrendo os elementos após o ponto ser encontrado. Nossa *callback* recebe as coordenadas que buscamos como dado adicional e tem 1 como valor de retorno, caso o ponto sendo visitado tenha as mesmas coordenadas (o teste de igualdade das coordenadas pode ser feito dentro de um intervalo de tolerância; podemos, por exemplo, usar um valor de tolerância igual a 10^{-5}):

```

#define TOL 1e-5

static int igualdade (void* info, void* dado)
{
    Ponto* p = (Ponto*)info;
    Ponto* q = (Ponto*)dado;
    if (fabs(p->x-q->x)<TOL && fabs(p->y-q->y)<TOL)
        return 1;
    else
        return 0;
}

```

Com isso, uma função do cliente para verificar a ocorrência das coordenadas (x,y) na estrutura é exemplificada por:

```

static int pertence (ListaGen* l, float x, float y)
{
    Ponto q = {x,y};
    return lgen_percorre(l,igualdade,&q);
}

```

Podemos também fazer a função de *callback* ter como valor de retorno um ponteiro genérico (`void*`) em vez de um inteiro. Neste caso, a função `percorre` é interrompida quando a *callback* retorna o valor `NULL`. Desta forma, o uso da função `percorre` fica ainda mais flexível: podemos implementar uma função `busca`, que tem como valor de retorno o ponteiro da informação encontrada.

Essas técnicas que utilizam *callbacks* são muito empregadas em programação, pois nos permitem esconder do cliente a forma como os elementos armazenados estão estruturados internamente. O cliente pode visitar e manipular todas as informações armazenadas, independentemente da estrutura de dados utilizada. Para o caso de uma lista simplesmente

encadeada, o leitor pode questionar a real utilidade de implementar estruturas genéricas e funções que utilizam *callbacks*. No entanto, em estruturas mais sofisticadas, essa generalização é muito útil, pois só precisamos implementar a estrutura de dados uma única vez. Como vimos, os algoritmos genéricos de ordenação e busca, implementados pela biblioteca padrão de C, também fazem uso de *callbacks* para poder ser independentes do tipo dos dados manipulados.

17.3 Árvore de busca genérica

Vamos agora considerar a implementação de uma árvore de busca genérica, isto é, uma árvore binária de busca que pode armazenar qualquer informação e que pode usar qualquer critério de ordenação. Análogo ao que fizemos para a lista genérica, a estrutura do nó da árvore deve ter como informação um ponteiro genérico:

```
typedef struct arvgenno ArvGenNo;
struct arvgenno {
    void* info;
    ArvNo* esq;
    ArvNo* dir;
};
```

No caso da árvore de busca, será necessário permitir que o cliente determine o critério de ordenação. O critério de ordenação adotado deve ser único para toda a estrutura da árvore, e será usado, por exemplo, para inserir um novo elemento e para buscar um elemento na estrutura. Podemos então pensar em duas funções de comparação distintas: a função de comparação usada na inserção, que deve comparar duas informações – a informação que está sendo inserida e a informação já existente em um dos nós da estrutura; e a função de comparação usada na busca, que deve comparar uma chave de busca com uma informação existente na estrutura. Já tínhamos visto esta diferença de funções de comparação quando discutimos as funções da biblioteca para ordenação (`naqsort`, a função de comparação recebe dois ponteiros para elementos do vetor) e para busca (`bsearch`, a função de comparação recebe um ponteiro para um elemento e um ponteiro para a chave de busca). Naturalmente, no caso da nossa árvore, as duas funções devem ser coerentes. Para evitar a necessidade de duas funções distintas, optamos por usar uma única função de comparação: chave *versus* informação. Como veremos, isso exigirá passar explicitamente a chave de busca de uma informação sendo inserida na árvore. Isso é redundante, pois a informação tem dentro dela a chave de busca (por exemplo, o nome de um aluno faz parte da estrutura que representa um aluno); no entanto, adotar uma única função de comparação nos parece melhor do que exigir duas funções distintas, mas coerentes.

Para garantir que uma mesma função de comparação será usada em todas as operações da nossa árvore, podemos armazenar o critério de ordenação junto com a estrutura que representa a árvore:

```
typedef struct arvgen ArvGen;
struct arvgen {
    ArvGenNo* raiz;
    int (*cmp) (void* chave, void* info);
};
```

A *callback* responsável pela comparação recebe dois ponteiros genéricos: um ponteiro para a chave de busca e um ponteiro para a informação armazenada em um nó. Por exemplo, a informação armazenada no nó pode ser um ponteiro para um tipo `Aluno` e a chave de busca pode ser um ponteiro para uma cadeia de caracteres (o nome do aluno). A função que cria uma árvore vazia recebe esta *callback* de comparação como parâmetro:

```
ArvGen* agen_cria (int (*cb) (void* chave, void* info))
{
    ArvGen* a = (ArvGen*) malloc(sizeof(ArvGen));
    a->raiz = NULL;
    a->cmp = cb;
    return a;
}
```

A função para inserir um novo nó na árvore deve receber como parâmetros a chave de busca associada e a informação a ser armazenada. A função recursiva recebe da função externa a *callback* de comparação como parâmetro:

```
static ArvGenNo* insere (ArvGenNo* r, void* chave, void* info,
                         int (*cmp) (void* chave, void* info))
{
    if (r==NULL) {
        r = (ArvGenNo*)malloc(sizeof(ArvGenNo));
        r->info = info;
        r->esq = r->dir = NULL;
    }
    else {
        int c = cmp(chave,r->info);
        if (c < 0)
            r->esq = insere(r->esq,chave,info,cmp);
        else
            r->dir = insere(r->dir,chave,info,cmp);
    }
    return r;
}
```

```
void agen_insere (ArvGen* a, void* chave, void* info)
{
    a->raiz = insere(a->raiz,chave,info,a->cmp);
}
```

A função de busca recebe a chave de busca e tem como valor de retorno a informação associada a esta árvore, se existente na árvore (caso contrário, retorna `NULL`). Novamente, a *callback* de comparação é passada internamente como parâmetro para a função recursiva.

```

static void* busca (ArvGenNo* r, void* chave,
                   int (*cmp) (void* chave, void* info))
{
    if (r == NULL) return NULL;
    else {
        int c = cmp(chave,r->info);
        if (c < 0) return busca (r->esq, chave, cmp);
        else if (c > 0) return busca (r->dir, chave, cmp);
        else return r->info;
    }
}

void* agen_busca (ArvGen* a, void* chave)
{
    return busca(a->raiz,chave,a->cmp);
}

```

Com esta implementação, como clientes, podemos fazer uso da nossa árvore de busca genérica. Vamos considerar um exemplo em que armazenamos na estrutura informações de um aluno:

```

typedef struct aluno Aluno;
struct aluno {
    char nome[81];
    float nota;
};

```

Vamos optar por organizar os alunos na estrutura por ordem alfabética do nome. Assim, nossa função de comparação receberá como chave um ponteiro para cadeia de caracteres e como informação um ponteiro para a estrutura `Aluno`:

```

static int cmp_nome_aluno (void* chave, void* info)
{
    char* nome = (char*) chave;
    Aluno* aluno = (Aluno*) info;
    return strcmp(nome,aluno->nome);
}

```

Podemos, ainda como clientes do nosso TAD, implementar uma função auxiliar para inserir os dados de um novo aluno na estrutura:

```

static void insere_aluno (ArvGen* a, char* nome, float nota)
{
    Aluno* aluno = (Aluno*) malloc(sizeof(Aluno));
    strcpy(aluno->nome,nome);
    aluno->nota = nota;
    agen_insere(a,nome,aluno);
}

```

Note a necessidade de passar a chave explicitamente, além da informação, na função de inserção. E, então, podemos escrever uma função *main* para testar o uso do nosso TAD:

```
int main (void)
{
    ArvGen* a = agen_cria(cmp_nome_aluno);
    insere_aluno(a,"Carlos",7.5);
    insere_aluno(a,"Rui",8.2);
    insere_aluno(a,"Marta",7.8);
    insere_aluno(a,"Ana",9.3);
    insere_aluno(a,"Paulo",6.5);

    Aluno* p = (Aluno*) agen_busca(a,"Rui");

    printf("%g\n",p->nota);
    return 0;
}
```

Como vimos na discussão da lista genérica, uma função importante é a que percorre os elementos da estrutura. No caso da árvore, os elementos são visitados de acordo com o critério de ordenação adotado. A *callback* usada na função que percorre os elementos pode seguir os princípios adotados na lista genérica:

```
static int percorre (ArvGenNo* r,
                      int (*cb) (void* info, void* dado), void* dado)
{
    if (r != NULL) {
        int s = percorre(r->esq,cb,dado);
        if (s != 0) return s;
        s = cb(r->info,dado);
        if (s != 0) return s;
        return percorre(r->dir,cb,dado);
    }
    return 0;
}

int agen_percorre (ArvGen* a,
                   int (*cb) (void* info, void* dado), void* dado)
{
    return percorre(a->raiz,cb,dado);
}
```

Com esta função, podemos escrever uma *callback* que imprime nome e nota dos alunos, apresentando-os em ordem alfabética dos nomes:

```
static int cb_imprime (void* info, void* dado)
{
    Aluno* aluno = (Aluno*) info;
    printf("%s: %.1f\n",aluno->nome,aluno->nota);
    return 0;
}
```

E usá-la numa função *main*, como em:

```
...
agen_percorre(a, cb_imprime, NULL);
...
```

Podemos ainda fazer uso do dado extra passado para a *callback*. Vamos supor, por exemplo, que desejamos que seja impresso apenas os primeiros *n* alunos da lista. Nossa dado extra da *callback* pode justamente representar este número, que será decrementado a cada impressão:

```
static int cb_imprime_primeiros (void* info, void* dado)
{
    int* pn = (int*) dado;
    if ((*pn)-- == 0)
        return 1;
    Aluno* aluno = (Aluno*) info;
    printf("%s: %.1f\n", aluno->nome, aluno->nota);
    return 0;
}
```

Com isso, para imprimir apenas os primeiros três alunos da estrutura, podemos escrever o seguinte fragmento de código:

```
...
int n = 3;
agen_percorre(a, cb_imprime_primeiros, &n);
...
```

Por fim, podemos considerar a implementação da função que libera a memória da estrutura. Novamente, cabe ao cliente determinar se e como as informações serão liberadas. Nossa função, então, também recebe uma *callback* do cliente. Esta *callback* pode simplesmente receber a informação associada a cada nó que será liberado:

```
static void libera (ArvGenNo* r, void (*cb) (void*))
{
    if (r != NULL) {
        libera(r->esq, cb);
        libera(r->dir, cb);
        cb(r->info);
        free(r);
    }
}
void agen_libera (ArvGen* a, void (*cb) (void*))
{
    libera(a->raiz, cb);
    free(a);
}
```

Note que a assinatura desta *callback* é a mesma da função `free` da biblioteca padrão. Assim, no cliente, passar `free` como parâmetro é suficiente para liberar a informação. Pode-se fazer:

```
...
agen_libera(a, free);
...
```

Para concluir, vale a pena escrevermos a interface deste TAD, que representa uma árvore de busca genérica, coletando as operações que implementamos no arquivo “`arvgen.h`”:

```
/* Árvore binária de busca genérica*/
typedef struct arvgen ArvGen;
ArvGen* agen_cria (int (*cb) (void* chave, void* info));
void agen_insere (ArvGen* a, void* chave, void* info);
void* agen_busca (ArvGen* a, void* chave);
int agen_percorre (ArvGen* a, int (*cb) (void* info, void* dado), void* dado);
void agen_libera (ArvGen* a, void (*cb) (void*));
```

Note que, neste caso, a estrutura que representa o nó da árvore não precisa ser exportada na interface, pois a função de busca já retorna a informação, e não o ponteiro do nó.

Exercícios

- Crie um TAD para representar vetor dinâmico genérico capaz de armazenar ponteiro para qualquer tipo. A interface deste TAD deve seguir o arquivo “`vetgen.h`”:

```
/* Vetor genérico que armazena void* */
typedef struct vetordin VetorDin;

/* cria vetor vazio */
VetorGen* vgen_cria (void);

/* insere novo elemento no final do vetor */
void vgen_insere (VetorGen* v, void* p);

/* retorna o número de elementos do vetor */
int vgen_tam (VetorGen* v);

/* retorna a informação associada ao elemento de índice i */
void* vgen_acessa (VetorGen* v, int i);

/* percorre os elementos do vetor chamando a callback */
int vgen_percorre (VetorGen* v,
                    int (*cb) (void* info, void* dado),
                    void* dado
                  );

/* libera a memória do vetor, chamando a callback */
void vgen_libera (VetorGen* v, void (*cb) (void*));
```

2. Use o TAD do item anterior para armazenar dados de alunos:

```
typedef struct aluno Aluno;
struct aluno {
    char nome[81];
    float nota;
};
```

Implemente o código necessário para calcular a média das notas dos alunos armazenados no vetor.

3. Altere a função `percorre` e sua respectiva *callback* do tipo lista genérica apresentado: em vez de retornar um valor inteiro, a *callback* deve retornar um tipo `void*` qualquer. Se o valor retornado pela *callback* for diferente de `NULL`, a função `percorre` deve ser interrompida e ter como valor de retorno o valor `void*` retornado pela *callback*. Use essa nova versão da função `percorre` para implementar uma busca na lista, dada uma chave de busca qualquer passada para a função `percorre` via parâmetro que representa o dado extra (`void* dado`). Sua função deve retornar a informação associada à chave de busca passada.
4. Use o TAD que representa uma árvore genérica de busca para armazenar dados de alunos, conforme foi discutido. No entanto, ordene os alunos por valor decrescente de notas. Implemente um código para listar todos os alunos que tiveram nota maior que 8.0.