Árvores

Nos capítulos anteriores examinamos as estruturas de dados que podem ser chamadas de unidimensionais ou lineares, como vetores e listas. A importância dessas estruturas é inegável, mas elas não são adequadas para representarmos dados que devem ser dispostos de maneira hierárquica. Por exemplo, os arquivos (documentos) que criamos num computador são armazenados dentro de uma estrutura hierárquica de diretórios (pastas). Existe um diretório base dentro do qual podemos armazenar diversos sub-diretórios e arquivos. Por sua vez, dentro dos sub-diretórios, podemos armazenar outros sub-diretórios e arquivos, e assim por diante, recursivamente. A Figura 1 mostra uma imagem de uma árvore de diretório no Windows 2000.

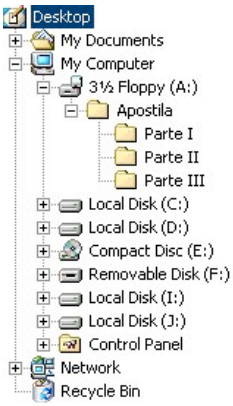

Figura 1: Um exemplo de árvore de diretório.

Neste capítulo, vamos introduzir *árvores*, que são estruturas de dados adequadas para a representação de hierarquias. A forma mais natural para definirmos uma estrutura de árvore é usando recursividade. Uma árvore é composta por um conjunto de nós. Existe um nó `r`, denominado *raiz*, que contém zero ou mais sub-árvores, cujas raízes são ligadas diretamente a `r`. Esses nós raízes das sub-árvores são ditos *filhos* do nó *pai*, `r`. Nós com filhos são comumente chamados de *nós internos* e nós que não têm filhos são chamados de *folhas*, ou nós externos. É tradicional desenhar as árvores com a raiz para cima e folhas para baixo, ao contrário do que seria de se esperar. A Figura 2 exemplifica a estrutura de uma árvore.

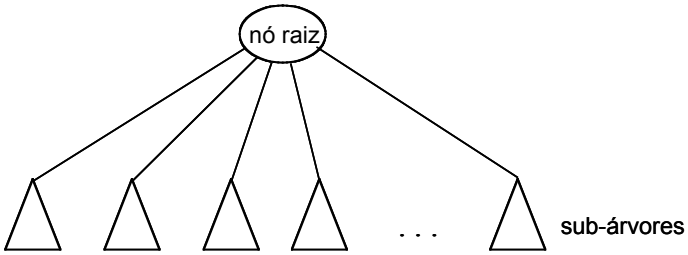

Figura 2: Estrutura de árvore.

Observamos que, por adotarmos essa forma de representação gráfica, não representamos explicitamente a direção dos ponteiros, subentendendo que eles apontam sempre do pai para os filhos.

O número de filhos permitido por nó e as informações armazenadas em cada nó diferenciam os diversos tipos de árvores existentes. Neste capítulo, estudaremos dois tipos de árvores. Primeiro, examinaremos as árvores binárias, onde cada nó tem, no máximo, dois filhos. Depois examinaremos as chamadas árvores genéricas, onde o número de filhos é indefinido. Estruturas recursivas serão usadas como base para o estudo e a implementação das operações com árvores.

Árvores binárias

Um exemplo de utilização de árvores binárias está na avaliação de expressões. Como trabalhamos com operadores que esperam um ou dois operandos, os nós da árvore para representar uma expressão têm no máximo dois filhos. Nessa árvore, os nós folhas representam operandos e os nós internos operadores. Uma árvore que representa, por exemplo a expressão `(3+6)*(4-1)+5` é ilustrada na Figura 3.

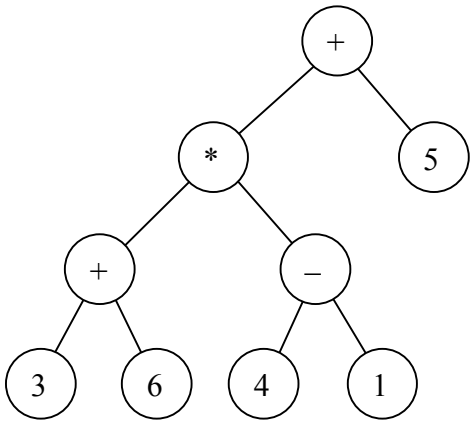

Figura 3: Árvore da expressão: (3+6) * (4-1) + 5.

Numa árvore binária, cada nó tem zero, um ou dois filhos. De maneira recursiva, podemos definir uma árvore binária como sendo:

- uma árvore vazia; ou
- um nó raiz tendo duas sub-árvores, identificadas como a sub-árvore da direita (*sad*) e a sub-árvore da esquerda (*sae*).

A Figura 4 ilustra a definição de árvore binária. Essa definição recursiva será usada na construção de algoritmos, e na verificação (informal) da correção e do desempenho dos mesmos.

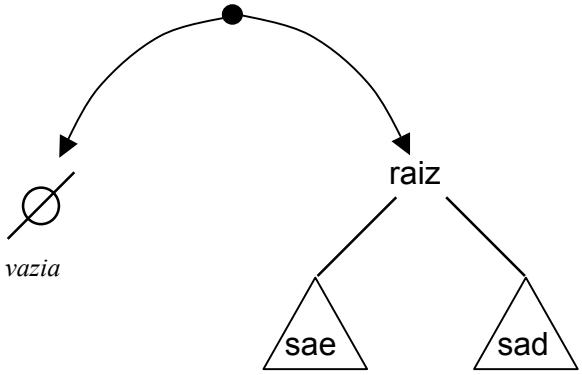

Figura 4: Representação esquemática da definição da estrutura de árvore binária.

A Figura 5 a seguir ilustra uma estrutura de árvore binária. Os nós *a*, *b*, *c*, *d*, *e*, *f* formam uma árvore binária da seguinte maneira: a árvore é composta do nó *a*, da sub-árvore à esquerda formada por *b* e *d*, e da sub-árvore à direita formada por *c*, *e* e *f*. O nó *a* representa a raiz da árvore e os nós *b* e *c* as raízes das sub-árvores. Finalmente, os nós *d*, *e* e *f* são folhas da árvore.

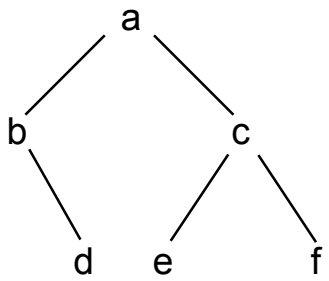

Figura 5: Exemplo de árvore binária

Para descrever árvores binárias, podemos usar a seguinte notação textual: a árvore vazia é representada por `<>`, e árvores não vazias por `<raiz sae sad>`. Com essa notação, a árvore da Figura 5 é representada por: `<a<b<><d<><>>><c<e<><>><f<><>>>>`.

Pela definição, uma sub-árvore de uma árvore binária é sempre especificada como sendo a *sae* ou a *sad* de uma árvore maior, e qualquer das duas sub-árvores pode ser vazia. Assim, as duas árvores da Figura 6 são distintas.

Figura 6: Duas árvores binárias distintas.

Isto também pode ser visto pelas representações textuais das duas árvores, que são, respectivamente: `<a <b<><>> <> >` e `<a <> <b<><>> >`.

Uma propriedade fundamental de todas as árvores é que só existe um caminho da raiz para qualquer nó. Com isto, podemos definir a *altura* de uma árvore como sendo o comprimento do caminho mais longo da raiz até uma das folhas. Por exemplo, a altura da árvore da Figura 5 é 2, e a altura das árvores da Figura 6 é 1. Assim, a altura de uma árvore com um único nó raiz é zero e, por conseguinte, dizemos que a altura de uma árvore vazia é negativa e vale -1.

Representação em C

Análogo ao que fizemos para as demais estruturas de dados, podemos definir um tipo para representar uma árvore binária. Para simplificar a discussão, vamos considerar que a informação que queremos armazenar nos nós da árvore são valores de caracteres simples. Vamos inicialmente discutir como podemos representar uma estrutura de árvore binária em C. Que estrutura podemos usar para representar um nó da árvore? Cada nó deve armazenar três informações: a informação propriamente dita, no caso um caractere, e dois ponteiros para as sub-árvores, à esquerda e à direita. Então a estrutura de C para representar o nó da árvore pode ser dada por:

```
struct arv {
   char info;
   struct arv* esq;
   struct arv* dir;
};
```

Da mesma forma que uma lista encadeada é representada por um ponteiro para o primeiro nó, a estrutura da árvore como um todo é representada por um ponteiro para o nó raiz.

Como acontece com qualquer TAD (tipo abstrato de dados), as operações que fazem sentido para uma árvore binária dependem essencialmente da forma de utilização que se pretende fazer da árvore. Nesta seção, em vez de discutirmos a interface do tipo abstrato para depois mostrarmos sua implementação, vamos optar por discutir algumas operações mostrando simultaneamente suas implementações. Ao final da seção apresentaremos um arquivo que pode representar a interface do tipo. Nas funções que se seguem, consideraremos que existe o tipo `Arv` definido por:

```
typedef struct arv Arv;
```

Como veremos as funções que manipulam árvores são, em geral, implementadas de forma recursiva, usando a definição recursiva da estrutura.

Vamos procurar identificar e descrever apenas operações cuja utilidade seja a mais geral possível. Uma operação que provavelmente deverá ser incluída em todos os casos é a inicialização de uma árvore vazia. Como uma árvore é representada pelo endereço do nó raiz, uma árvore vazia tem que ser representada pelo valor `NULL`. Assim, a função que inicializa uma árvore vazia pode ser simplesmente:

```
Arv* inicializa(void)
{
   return NULL;
}
```

Para criar árvores não vazias, podemos ter uma operação que cria um nó raiz dadas a informação e suas duas sub-árvores, à esquerda e à direita. Essa função tem como valor de retorno o endereço do nó raiz criado e pode ser dada por:

```
Arv* cria(char c, Arv* sae, Arv* sad){
   Arv* p=(Arv*)malloc(sizeof(Arv));
   p->info = c;
   p->esq = sae;
   p->dir = sad;
   return p;
}
```

As duas funções `inicializa` e `cria` representam os dois casos da definição recursiva de árvore binária: uma árvore binária (`Arv* a;`) é vazia (`a = inicializa();`) ou é composta por uma raiz e duas sub-árvores (`a = cria(c,sae,sad);`). Assim, com posse dessas duas funções, podemos criar árvores mais complexas.

Exemplo: Usando as operações `inicializa` e `cria`, crie uma estrutura que represente a árvore da Figura 5.

O exemplo da figura pode ser criada pela seguinte seqüência de atribuições.

```
Arv* a1= cria('d',inicializa(),inicializa());    /* sub-árvore com 'd'
*/
Arv* a2= cria('b',inicializa(),a1);               /* sub-árvore com 'b'
*/
Arv* a3= cria('e',inicializa(),inicializa());    /* sub-árvore com 'e'
*/
Arv* a4= cria('f',inicializa(),inicializa());    /* sub-árvore com 'f'
*/
Arv* a5= cria('c',a3,a4);                         /* sub-árvore com 'c'
*/
Arv* a = cria('a',a2,a5 );                       /* árvore com raiz 'a'
*/
```

Alternativamente, a árvore poderia ser criada com uma única atribuição, seguindo a sua estrutura, “recursivamente”:

```
Arv* a = cria('a',
              cria('b',
                   inicializa(),
                   cria('d', inicializa(), inicializa())
                  ),
              cria('c',
                   cria('e', inicializa(), inicializa()),
                   cria('f', inicializa(), inicializa())
                  )
             );
```

Para tratar a árvore vazia de forma diferente das outras, é importante ter uma operação que diz se uma árvore é ou não vazia. Podemos ter:

```
int vazia(Arv* a)
{
   return a==NULL;
}
```

Uma outra função muito útil consiste em exibir o conteúdo da árvore. Essa função deve percorrer recursivamente a árvore, visitando todos os nós e imprimindo sua informação. A implementação dessa função usa a definição recursiva da árvore. Vimos que uma árvore binária ou é vazia ou é composta pela raiz e por duas sub-árvores. Portanto, para imprimir a informação de todos os nós da árvore, devemos primeiro testar se a árvore é vazia. Se não for, imprimimos a informação associada a raiz e chamamos (recursivamente) a função para imprimir os nós das sub-árvores.

```
void imprime (Arv* a)
{
   if (!vazia(a)){
      printf("%c ", a->info);      /* mostra raiz */
      imprime(a->esq);            /* mostra sae */
      imprime(a->dir);            /* mostra sad */
   }
}
```

Uma outra operação que pode ser acrescentada é a operação para liberar a memória alocada pela estrutura da árvore. Novamente, usaremos uma implementação recursiva. Um cuidado essencial a ser tomado é que as sub-árvores devem ser liberadas antes de se liberar o nó raiz, para que o acesso às sub-árvores não seja perdido antes de sua remoção. Neste caso, vamos optar por fazer com que a função tenha como valor de retorno a árvore atualizada, isto é, uma árvore vazia, representada por `NULL`.

```
Arv* libera (Arv* a){
   if (!vazia(a)){
      libera(a->esq);  /* libera sae */
      libera(a->dir);  /* libera sad */
      free(a);         /* libera raiz */
   }
   return NULL;
}
```

Devemos notar que a definição de árvore, por ser recursiva, não faz distinção entre árvores e sub-árvores. Assim, `cria` pode ser usada para acrescentar (“enxertar”) uma sub-árvore em um ramo de uma árvore, e `libera` pode ser usada para remover (“podar”) uma sub-árvore qualquer de uma árvore dada.

Exemplo: Considerando a criação da árvore feita anteriormente:

```
Arv* a = cria('a',
             cria('b',
                  inicializa(),
                  cria('d', inicializa(), inicializa())
                 ),
             cria('c',
                  cria('e', inicializa(), inicializa()),
                  cria('f', inicializa(), inicializa())
                 )
            );
```

Podemos acrescentar alguns nós, com:

```
a->esq->esq = cria('x',
                  cria('y',inicializa(),inicializa()),
                  cria('z',inicializa(),inicializa())
                 );
```

E podemos liberar alguns outros, com:

```
a->dir->esq = libera(a->dir->esq);
```

Deixamos como exercício a verificação do resultado final dessas operações.

É importante observar que, análogo ao que fizemos para a lista, o código cliente que chama a função libera é responsável por atribuir o valor atualizado retornado pela função, no caso uma árvore vazia. No exemplo acima, se não tivéssemos feito a atribuição, o endereço armazenado em `r->dir->esq` seria o de uma área de memória não mais em uso.

Uma outra função que podemos considerar percorre a árvore buscando a ocorrência de um determinado caractere `c` em um de seus nós. Essa função tem como retorno um valor booleano (um ou zero) indicando a ocorrência ou não do caractere na árvore.

```
int busca (Arv* a, char c){
   if (vazia(a))
      return 0;    /* árvore vazia: não encontrou */
   else
      return a->info==c || busca(a->esq,c) || busca(a->dir,c);
}
```

Note que esta forma de programar `busca`, em C, usando o operador lógico `||` (“ou”) faz com que a busca seja interrompida assim que o elemento é encontrado. Isto acontece porque se `c==a->info` for verdadeiro, as duas outras expressões não chegam a ser avaliadas. Analogamente, se o caractere for encontrado na sub-árvore da esquerda, a busca não prossegue na sub-árvore da direita.

Podemos dizer que a expressão:

```
return c==a->info || busca(a->esq,c) || busca(a->dir,c);
```

é equivalente a:

```
if (c==a->info)
   return 1;
else if (busca(a->esq,c))
   return 1;
else
   return busca(a->dir,c);
```

Finalmente, considerando que as funções discutidas e implementadas acima formam a interface do tipo abstrato para representar uma árvore binária, um arquivo de interface `arvbin.h` pode ser dado por:

```
typedef struct arv Arv;

Arv* inicializa (void);
Arv* cria (char c, Arv* e, Arv* d);
int  vazia (Arv* a);
void imprime (Arv* a);
Arv* libera (Arv* a);
int  busca (Arv* a, char c);
```

Ordens de percurso em árvores binárias

A programação da operação `imprime`, vista anteriormente, seguiu a ordem empregada na definição de árvore binária para decidir a ordem em que as três ações seriam executadas:
Entretanto, dependendo da aplicação em vista, esta ordem poderia não ser a preferível, podendo ser utilizada uma ordem diferente desta, por exemplo:

```
imprime(a->esq);              /* mostra sae */
imprime(a->dir);              /* mostra sad */
printf("%c ", a->info);     /* mostra raiz */
```

Muitas operações em árvores binárias envolvem o percurso de todas as sub-árvores, executando alguma ação de tratamento em cada nó, de forma que é comum percorrer uma árvore em uma das seguintes ordens:

- *pré-ordem*: trata *raiz*, percorre *sae*, percorre *sad*;
- *ordem simétrica*: percorre *sae*, trata *raiz*, percorre *sad*;
- *pós-ordem*: percorre *sae*, percorre *sad*, trata *raiz*.

Para função para liberar a árvore, por exemplo, tivemos que adotar a pós-ordem:

```
libera(a->esq);  /* libera sae */
libera(a->dir);  /* libera sad */
free(a);         /* libera raiz */
```

Na terceira parte do curso, quando tratarmos de árvores binárias de busca, apresentaremos um exemplo de aplicação de árvores binárias em que a ordem de percurso importante é a ordem simétrica. Algumas outras ordens de percurso podem ser definidas, mas a maioria das aplicações envolve uma dessas três ordens, percorrendo a *sae* antes da *sad*.

2. Árvores genéricas

Nesta seção, vamos discutir as estruturas conhecidas como árvores genéricas. Como vimos, numa árvore binária o número de filhos dos nós é limitado em no máximo dois. No caso da árvore genérica, esta restrição não existe. Cada nó pode ter um número arbitrário de filhos. Essa estrutura deve ser usada, por exemplo, para representar uma árvore de diretórios.

Como veremos, as funções para manipularem uma árvore genérica também serão implementadas de forma recursiva, e serão baseadas na seguinte definição: uma árvore genérica é composta por:

- um nó raiz; e
- zero ou mais sub-árvores.

Estritamente, segundo essa definição, uma árvore não pode ser vazia, e a árvore vazia não é sequer mencionada na definição. Assim, uma folha de uma árvore não é um nó com sub-árvores vazias, como no caso da árvore binária, mas é um nó com *zero* sub-árvores. Em qualquer definição recursiva deve haver uma "condição de contorno", que permita a definição de estruturas finitas, e, no nosso caso, a definição de uma árvore se encerra nas *folhas*, que são identificadas como sendo nós com zero sub-árvores.

Como as funções que implementaremos nesta seção serão baseadas nessa definição, não será considerado o caso de árvores vazias. Esta pequena restrição simplifica as implementações recursivas e, em geral, não limita a utilização da estrutura em aplicações reais. Uma árvore de diretório, por exemplo, nunca é vazia, pois sempre existe o diretório base – o diretório raiz.

Como as sub-árvores de um determinado nó formam um conjunto linear e são dispostas numa determinada ordem, faz sentido falarmos em primeira sub-árvore (sa_1), segunda sub-árvore (sa_2), etc. Um exemplo de uma árvore genérica é ilustrado na Figura 7.

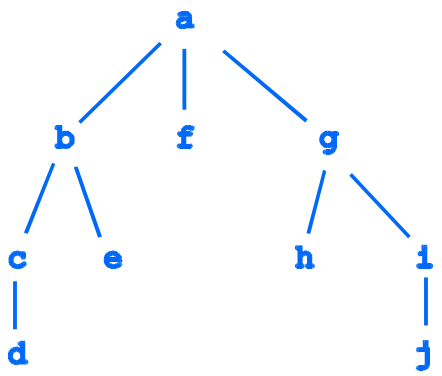

Figura 7: Exemplo de árvore genérica.

Nesse exemplo, podemos notar que o a árvore com raiz no nó `a` tem 3 sub-árvores, ou, equivalentemente, o nó `a` tem 3 filhos. Os nós `b` e `g` tem dois filhos cada um; os nós `c` e `i` tem um filho cada, e os nós `d`, `e`, `h` e `j` são folhas, e tem zero filhos.

De forma semelhante ao que foi feito no caso das árvores binárias, podemos representar essas árvores através de notação textual, seguindo o padrão: <raiz sa_1 sa_2 ... sa_n>. Com esta notação, a árvore da Figura 7 seria representada por:

α = <a <b <c <d>> <e>> <f> <g <h> <i <j>>>>

Podemos verificar que α representa a árvore do exemplo seguindo a seqüência de definição a partir das folhas:

α_1 = <d>
α_2 = <c α_1> = <c <d>>
α_3 = <e>
α_4 = <b α_2 α_3> = <b <c <d>> <e>>
α_5 = <f>
α_6 = <h>
α_7 = <j>
α_8 = <i α_7> = <i <j>>
α_9 = <g α_6 α_8> = <g <h> <i <j>>>
α = <a α_4 α_5 α_9> = <a <b <c <d>> <e>> <f> <g <h> <i <j>>>>

Representação em C

Dependendo da aplicação, podemos usar várias estruturas para representar árvores, levando em consideração o número de filhos que cada nó pode apresentar. Se soubermos, por exemplo, que numa aplicação o número máximo de filhos que um nó pode apresentar é 3, podemos montar uma estrutura com 3 campos para apontadores para os nós filhos, digamos, `f1`, `f2` e `f3`. Os campos não utilizados podem ser preenchidos com o valor nulo `NULL`, sendo sempre utilizados os campos em ordem. Assim, se o nó `n` tem 2 filhos, os campos `f1` e `f2` seriam utilizados, nessa ordem, para apontar para eles, ficando `f3` vazio. Prevendo um número máximo de filhos igual a 3, e considerando a implementação de árvores para armazenar valores de caracteres simples, a declaração do tipo que representa o nó da árvore poderia ser:

```
struct arv3 {
   char val;
   struct no *f1, *f2, *f3;
};
```

A Figura 8 indica a representação da árvore da Figura 7 com esta organização. Como se pode ver no exemplo, em cada um dos nós que tem menos de três filhos, o espaço correspondente aos filhos inexistentes é desperdiçado. Além disso, se não existe um limite superior no número de filhos, esta técnica pode não ser aplicável. O mesmo acontece se existe um limite no número de nós, mas esse limite será raramente alcançado, pois estaríamos tendo um grande desperdício de espaço de memória com os campos não utilizados.

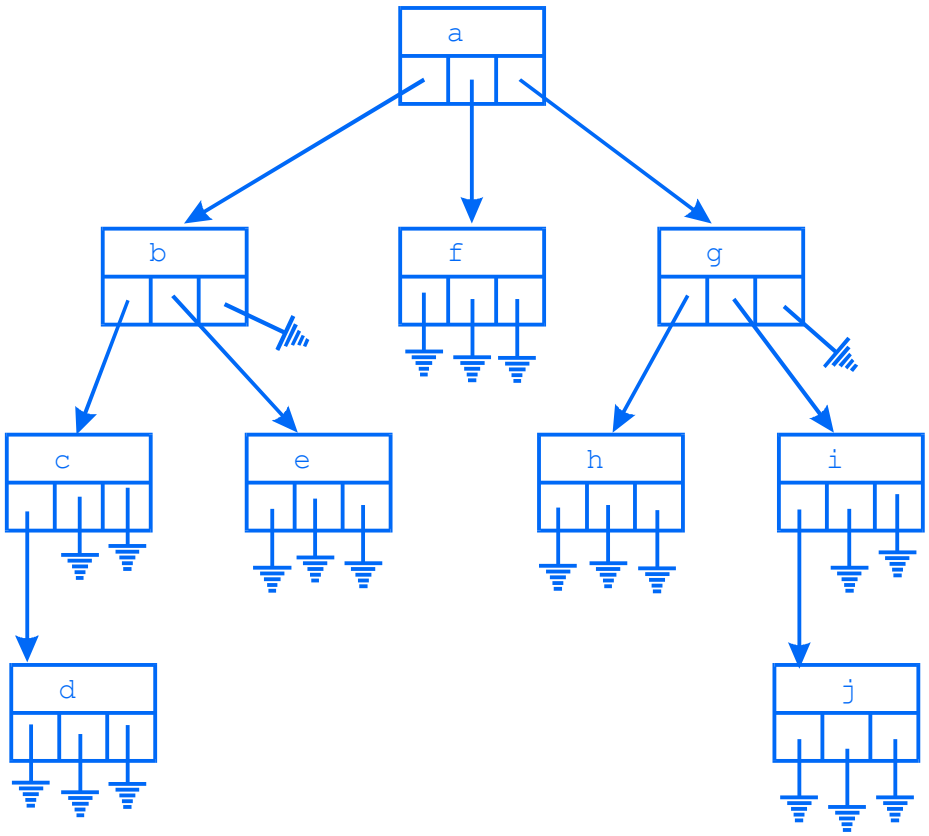

Figura 8: Árvore com no máximo três filhos por nó.

Uma solução que leva a um aproveitamento melhor do espaço utiliza uma "lista de filhos": um nó aponta apenas para seu primeiro (`prim`) filho, e cada um de seus filhos, exceto o último, aponta para o próximo (`prox`) irmão. A declaração de um nó pode ser:

```
struct arvgen {
   char info;
   struct arvgen *prim;
   struct arvgen *prox;
};
```

A Figura 9 mostra o mesmo exemplo representado de acordo com esta estrutura. Uma das vantagens dessa representação é que podemos percorrer os filhos de um nó de forma sistemática, de maneira análoga ao que fizemos para percorrer os nós de uma lista simples.

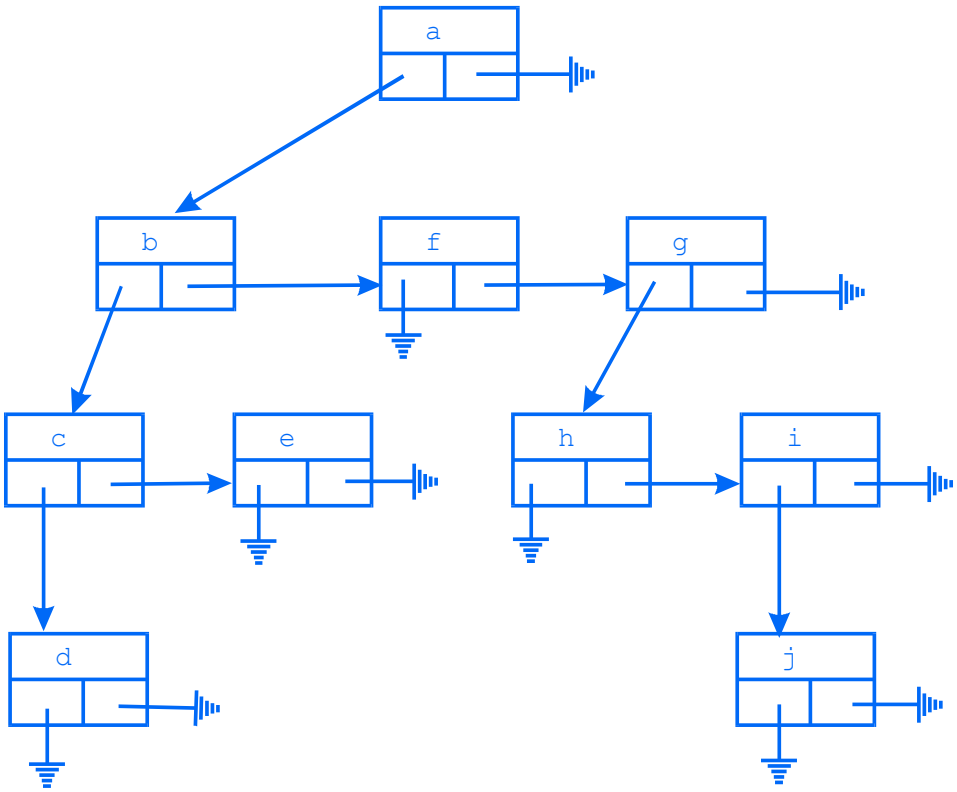

Figura 9: Exemplo usando “lista de filhos”.

Com o uso dessa representação, a generalização da árvore é apenas conceitual, pois, concretamente, a árvore foi transformada em uma árvore binária, com filhos esquerdos apontados por `prim` e direitos apontados por `prox`. Naturalmente, continuaremos a fazer referência aos nós nos termos da definição original. Por exemplo, os nós `b`, `f` e `g` continuarão a ser considerados *filhos* do nó `a`, como indicado na Figura 7, mesmo que a representação usada na Figura 9 os coloque a distâncias variáveis do nó pai.

Tipo abstrato de dado

Para exemplificar a implementação de funções que manipulam uma árvore genérica, vamos considerar a criação de um tipo abstrato de dados para representar árvores onde a informação associada a cada nó é um caractere simples. Nessa implementação, vamos optar por armazenar os filhos de um nó numa lista encadeada. Podemos definir o seguinte conjunto de operações:

- cria: cria um nó folha, dada a informação a ser armazenada;
- insere: insere uma nova sub-árvore como filha de um dado nó;
- imprime: percorre todos os nós e imprime suas informações;
- busca: verifica a ocorrência de um determinado valor num dos nós da árvore;
- libera: libera toda a memória alocada pela árvore.

A interface do tipo pode então ser definida no arquivo `arvgen.h` dado por:

```
typedef struct arvgen ArvGen;

ArvGen* cria (char c);
void    insere (ArvGen* a, ArvGen* sa);
void    imprime (ArvGen* a);
int     busca (ArvGen* a, char c);
void    libera (ArvGen* a);
```

A estrutura `arvgen`, que representa o nó da árvore, é definida conforme mostrado anteriormente. A função para criar uma folha deve alocar o nó e inicializar seus campos, atribuindo `NULL` para os campos `prim` e `prox`, pois trata-se de um nó folha.

```
ArvGen* cria (char c)
{
   ArvGen *a =(ArvGen *) malloc(sizeof(ArvGen));
   a->info = c;
   a->prim = NULL;
   a->prox = NULL;
   return a;
}
```

A função que insere uma nova sub-árvore como filha de um dado nó é muito simples. Como não vamos atribuir nenhum significado especial para a posição de um nó filho, a operação de inserção pode inserir a sub-árvore em qualquer posição. Neste caso, vamos optar por inserir sempre no início da lista que, como já vimos, é a maneira mais simples de inserir um novo elemento numa lista encadeada.

```
void insere (ArvGen* a, ArvGen* sa)
{
   sa->prox = a->prim;
   a->prim = sa;
}
```

Com essas duas funções, podemos construir a árvore do exemplo da Figura 7 com o seguinte fragmento de código:

```
/* cria nós como folhas */
ArvGen* a = cria('a');
ArvGen* b = cria('b');
ArvGen* c = cria('c');
ArvGen* d = cria('d');
ArvGen* e = cria('e');
ArvGen* f = cria('f');
ArvGen* g = cria('g');
ArvGen* h = cria('h');
ArvGen* i = cria('i');
ArvGen* j = cria('j');
/* monta a hierarquia */
insere(c,d);
insere(b,e);
insere(b,c);
insere(i,j);
insere(g,i);
insere(g,h);
insere(a,g);
insere(a,f);
insere(a,b);
```

Para imprimir as informações associadas aos nós da árvore, temos duas opções para percorrer a árvore: pré-ordem, primeiro a raiz e depois as sub-árvores, ou pós-ordem, primeiro as sub-árvores e depois a raiz. Note que neste caso não faz sentido a ordem simétrica, uma vez que o número de sub-árvores é variável. Para essa função, vamos optar por imprimir o conteúdo dos nós em pré-ordem:

```
void imprime (ArvGen* a)
{
   ArvGen* p;
   printf("%c\n",a->info);
   for (p=a->prim; p!=NULL; p=p->prox)
      imprime(p);
}
```

A operação para buscar a ocorrência de uma dada informação na árvore é exemplificada abaixo:

```
int busca (ArvGen* a, char c)
{
   ArvGen* p;
   if (a->info==c)
      return 1;
   else {
      for (p=a->prim; p!=NULL; p=p->prox) {
          if (busca(p,c))
              return 1;
      }
   }
   return 0;
}
```

A última operação apresentada é a que libera a memória alocada pela árvore. O único cuidado que precisamos tomar na programação dessa função é a de liberar as sub-árvores antes de liberar o espaço associado a um nó (isto é, usar pós-ordem).

```
void libera (ArvGen* a)
{
   ArvGen* p = a->prim;
   while (p!=NULL) {
      ArvGen* t = p->prox;
      libera(p);
      p = t;
   }
   free(a);
}
```