capítulo 12

Árvores *N*-árias

> *Ciência é conhecimento organizado. Sabedoria é vida organizada.*
>
> Immanuel Kant

A sabedoria em manter uma boa organização dos dados em uma estrutura de dados árvore, muitas vezes, leva à necessidade de novas escolhas. Ampliar o número de subárvores para um nó da árvore pode significar maior organização dos dados ali armazenados.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- entender o motivo de ampliar o número de subárvores (nós filhos) de uma estrutura de dados árvore;
- criar e manipular dados na forma de árvores *N*-árias.

Para começar

Imagine que você tenha duas jabuticabeiras. Numa delas, cada galho, a partir da raiz, se divide em no máximo dois novos galhos; na outra, cada galho pode se dividir em até cinco novos galhos.

Considere ainda que os frutos nascem sempre ou na ponta dos galhos ou no ponto em que ele se divide em novos galhos. Na primeira jabuticabeira temos apenas um fruto na ponta de cada galho e um fruto nesse ponto de divisão dos novos galhos. Já na segunda temos quatro frutos, tanto na ponta de cada galho, como no ponto de divisão dos novos galhos.

A partir dessas considerações, qual seria o tamanho mínimo das duas jabuticabeiras para que cada uma delas comportasse 200 frutos?

No caso de você decidir colher os frutos, em qual delas haveria maior dificuldade?

Você deve ter concluído que, para comportar as 200 jabuticabas, a primeira jabuticabeira teria altura muito maior que a segunda, levando um tempo maior para que todos os frutos fossem alcançados e colhidos.

Considerando as restrições impostas, quanto maior o volume de frutos, maior deverá ser a altura das jabuticabeiras e maior a dificuldade na colheita, principalmente na primeira árvore. Na estrutura de dados árvore ocorre o mesmo.

Atenção

Conforme aumentamos o volume de dados a serem armazenados numa estrutura de árvore, maior é a dificuldade de manter a eficiência na recuperação, principalmente tendo um número pequeno de subárvores a partir de determinado nó da árvore.

Para compreender melhor esse novo conceito, procure repetir a tarefa proposta no início do Capítulo 11, em que lhe foi solicitado criar uma estruturação hierárquica na forma de árvore de diretórios (pastas) para suas fotografias digitais, de forma que fosse fácil localizar determinada fotografia. Porém, dessa vez, crie tal representação considerando a limitação de que, em cada pasta, somente é possível ter duas novas pastas dentro (como uma árvore binária).

Conseguiu? Procure comparar a estruturação que você construiu no Capítulo 11 (que provavelmente continha várias subpastas dentro de uma pasta) com a nova, relacionando possíveis vantagens e desvantagens de cada forma de estruturação.

Dica

Para estabelecer a comparação, considere a facilidade para manipular suas pastas e o tempo gasto para encontrar uma fotografia desejada em ambas as estruturações.

Depois disso, esperamos que você tenha entendido o motivo da existência das árvores em que não ficamos limitados a dois novos nós filhos (subárvores) a partir de cada nó, como ocorre com as árvores binárias. Essas novas formas de árvores recebem o nome de *árvores N-árias*.

Neste capítulo, num primeiro momento, conheceremos a conceituação de árvores *N*-árias. Depois, estudaremos suas operações básicas e, por fim, procuraremos caracterizar brevemente suas possíveis vantagens, apresentando também um exemplo de aplicação prática. Vamos lá!

Conhecendo a teoria para programar

As definições apresentadas no início do Capítulo 11 para a estrutura de dados árvore a caracterizam de forma geral, portanto, englobam as árvores *N*-árias. Assim, tanto a definição quanto os conceitos apresentados (Nó, nó

raiz, nó filho, nível de um nó ou árvore, grau de um nó ou árvore e altura de um nó ou árvore) são válidos para as árvores *N*-árias.

Entretanto, aqui nosso interesse recai sobre a árvore *N*-ária com organização interna que permita o estabelecimento de determinada ordem entre os valores atribuídos a seus elementos. Esse tipo de árvore é chamada *árvore N-ária de busca.*

Atenção

Assim como no Capítulo11 tratamos as árvores binárias de busca apenas como árvores binárias, neste capítulo, quando fizermos menção a uma árvore *N*-ária, estaremos, na verdade, abordando uma árvore *N*-ária de busca.

Neste contexto, tal como nas árvores binárias (de busca), associa-se a cada elemento da árvore uma *chave*, através da qual se dá a ordenação desses elementos. Neste livro, optamos por utilizar apenas o campo *chave* para caracterizar o elemento que será armazenado na árvore, desconsiderando a presença de dados secundários do elemento. Assim, nos exemplos dados aqui, ao fazer uma referência à *chave*, referenciaremos o próprio valor do elemento. Para adicionar dados secundários ao elemento da árvore, bastaria criar no registro (que será caracterizado posteriormente neste capítulo) outros campos que possam armazená-los, sem qualquer outro impacto nos conceitos e ideias que serão discutidas.

Definição

Uma árvore *N*-ária (de busca) AN pode ser definida como:

- estrutura vazia, AN = {};
- conjunto finito e não vazio de nós, em que existe um nó raiz *R* e nós que fazem parte de subárvores de AN, AN = {R,A1,A2,A3,...,A**n**}, sendo que cada nó contém até n-1 elementos, sendo *n* o número máximo de subárvores a partir de um nó;
- os elementos dentro de um nó estão sempre ordenados por meio de de suas respectivas chaves;
- à esquerda de cada elemento Ei de um nó, no caso de existir uma subárvore, ela conterá elementos com chaves menores que a chave de Ei;
- à direita de cada elemento Ei de um nó, no caso de existir uma subárvore, ela conterá elementos com chaves maiores que a chave de Ei.

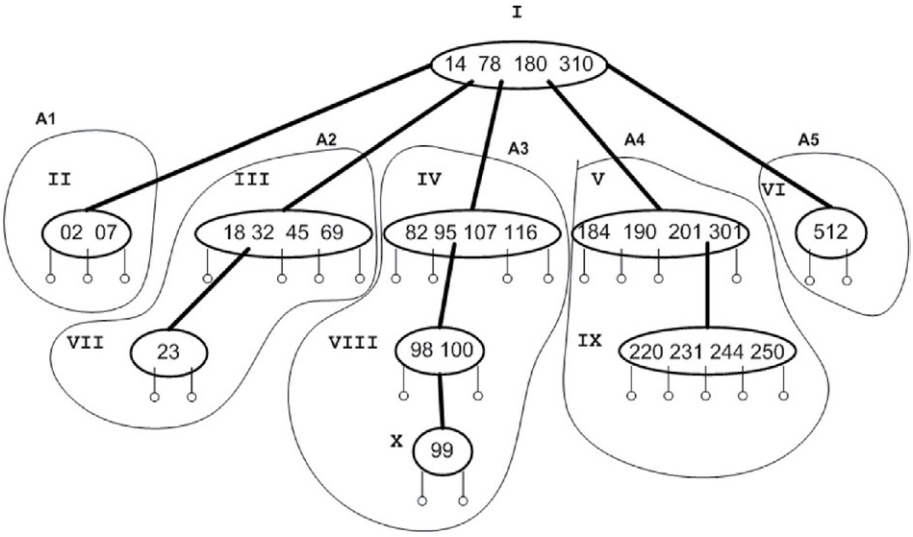


FIGURA 12.1: Árvore *N*-ária AN.

Para ilustrar essa definição, vamos considerar a árvore AN em que temos a raiz I e as subárvores A1, A2 A3, A4 e A5 (observe a Figura 12.1).

Observe que em AN, conforme descrito, cada nó pode conter determinado número de chaves (elementos), diferentemente de uma árvore binária, em que, em cada nó, existe uma única chave (elemento).

Conceitos

Numa árvore *N*-ária é fácil perceber a relação entre o número de nós filhos que a árvore suporta (subárvores de um nó) e o número de chaves (elementos) dentro do nó. No exemplo da árvore AN (Figura 12.1), temos uma árvore em que cada nó pode ter até cinco subárvores filhas e, consequentemente, até quatro chaves (elementos) por nó.

Outra observação importante refere-se ao fato de termos em AN nós que contêm um número de chaves inferior à capacidade máxima do nó – no caso nós II, VI, VII, VIII e X. Nós como esses são chamados de *nós incompletos*. De outro lado, temos nós cujas capacidades máximas de chaves foram atingidas (nós I, III, IV, V e IX), que são chamados de *nós completos*.

Para armazenar as chaves de um nó, em geral utilizamos um vetor, dada a simplicidade de sua manipulação e economia de espaço possíveis quando se evitam os apontadores dinâmicos para as chaves seguintes contidas no nó.

De outro lado, pelo fato de existirem nós incompletos, o uso do vetor leva ao desperdício de espaço de armazenamento dentro do nó, uma vez que, mesmo não havendo um número de chaves máximo, temos o espaço no vetor alocado.

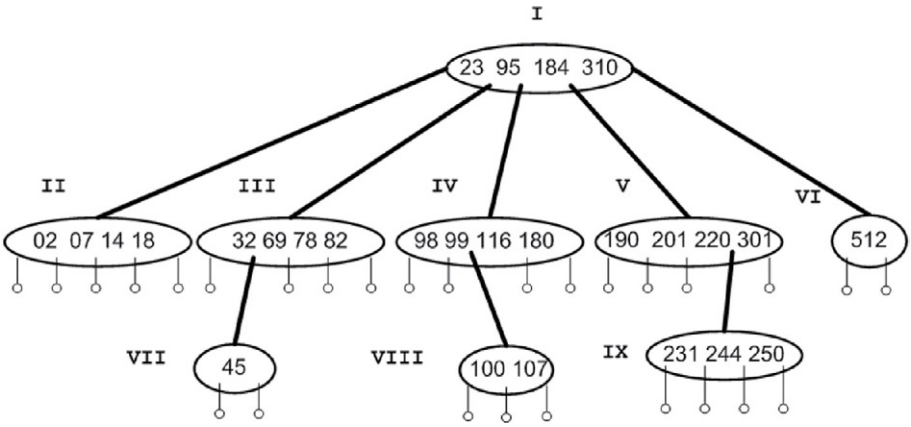


FIGURA 12.2: Árvore *N*-ária *topdown.*

Assim, é interessante evitarmos a existência de nós incompletos, procurando sempre distribuir as chaves de forma que o maior número possível de nós existentes seja completo.

Além disso, a manutenção de nós completos viabiliza a redução do número de nós (com consequente redução na altura da árvore), o que, em caso de busca por determinada chave, leva à redução do número de acessos aos nós da árvore. Dessa forma, podemos considerar que nas árvores *N*-árias é importante tentar manter o maior número de nós completos possível.

Nesse contexto, outro conceito importante refere-se à chamada árvore *N*-ária *topdown* (de cima para baixo). A existência dessa árvore se dá a partir da satisfação da condição de que qualquer nó incompleto deve necessariamente ser um nó folha.

Para ilustrar melhor o conceito de árvore *N*-ária *topdown*, observe a Figura 12.2.

Observe que, na árvore da Figura 12.2, todos os nós que não são folhas (I, III,IV e V) são nós completos.

Ampliando os conceitos, tomemos o exemplo da Figura 12.3. Observe que a árvore nele representada tem tal distribuição de chaves pelos nós que fazem seus nós folhas (VII, VIII, IX, X e XI) estarem no mesmo nível. Essa árvore é considerada *balanceada* (pela altura da árvore) e será tópico de discussão no Capítulo 13. Por enquanto, pense nas possíveis vantagens e desvantagens que ela poderia trazer.

A árvore da Figura 12.3, embora balanceada, não é uma árvore *N*-ária *topdown*, pois apresenta nós não folhas que são incompletos. Para compreender melhor esse conceito, construa uma árvore *N*-ária *topdown* balanceada.

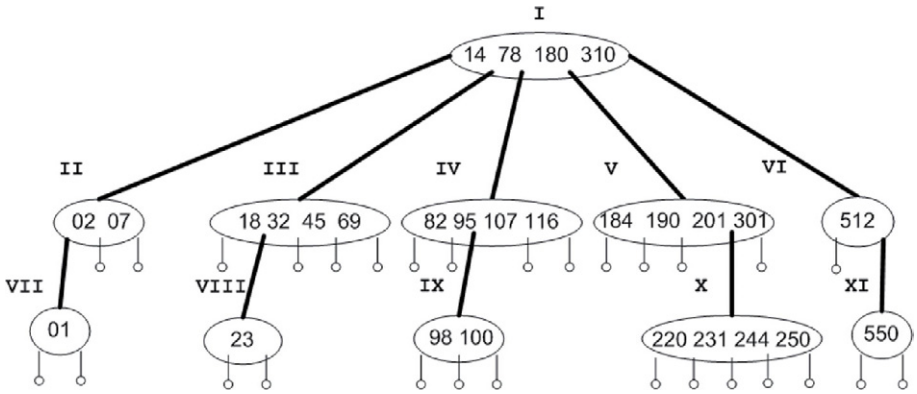


FIGURA 12.3: Árvore *N*-ária balanceada.

Conceito

O número máximo de chaves no nível *N* em uma árvore *N*-ária será o produto de número máximo de chaves do nível N-1 pelo grau da árvore. Dessa forma, uma árvore *N*-ária de grau 5, no nível 0 terá até 4 chaves, no nível 1 terá até 20 chaves, no nível 2 terá até 100 chaves, e assim por diante.

Implementação de uma árvore *N*-ária

Numa árvore *N*-ária, devido à variação que um nó dessa árvore pode apresentar em termos de número de chaves e subárvores, a forma comumente adotada para sua implementação é aquela voltada ao uso de alocação dinâmica de memória.

Embora não exista uma forma única de fazer essa implementação, neste livro optamos por representar um nó da árvore *N*-ária por meio da definição de um registro contendo um valor numérico, em que é armazenado o número de chaves existentes no nó (o máximo será *N*); uma lista de *N* posições que armazenará as chaves (elementos) contidas no nó; e uma lista de N+1 posições que armazenará os apontadores para as subárvores desse nó.

Dica

Para armazenar os dados secundários dos elementos que farão parte do nó da árvore binária, bastaria alterar os elementos para conter não apenas o campo chave, mas também campos adicionais em que os dados secundários ficariam associados aos respectivos elementos. Porém, por motivo de simplificação, como explicamos no início deste capítulo, ficaremos restritos ao armazenamento das chaves como representantes do elemento armazenado.

Na Figura 12.4, temos a representação de um nó dessa árvore:

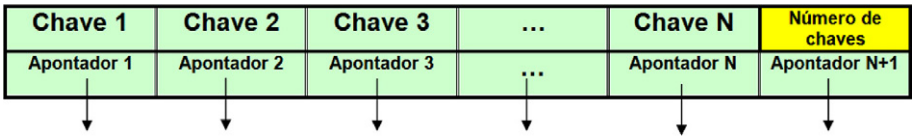


FIGURA 12.4: Representação de um nó da árvore *N*-ária.

Declaração de uma árvore *N*-ária

Consiste na definição do nó representado na Figura 12.4 através de um registro. Para representar os campos descritos na representação desse nó, temos: *nro_chaves* (armazena o número de chaves que estão armazenadas no nó, em determinado momento), *chaves* (vetor com *N* posições que armazena as chaves), e *apontadores* (vetor com N+1 posições que armazena os apontadores para as subárvores).

Código 12.1

```
typedef struct tipo_no no;
struct tipo_no
{
  int nro_chaves;
  tipo_dado chaves[N];
  tipo_no *apontadores[N+1];
};
```

Assim como ocorre nas árvores binárias, nos exemplos que serão apresentados sobre árvores *N*-árias vamos adotar que *tipo_dado* será um valor do tipo *inteiro*. Assim:

```
typedef int tipo_dado;
```

Operações básicas numa árvore *N*-ária

A manipulação de uma árvore *N*-ária pode ocorrer por diferentes operações. Assim como fizemos com as árvores binárias, vamos focar as operações básicas de maior utilidade e difusão: *percurso* (*busca*), *inserção* e *remoção*.

Percurso de árvore* N-*ária

Na definição de árvores *N*-árias (de busca) que estamos abordando neste capítulo, temos que, à esquerda de cada elemento Ei de um nó da árvore, no caso de existir uma subárvore, ela conterá elementos com chaves menores que a chave de Ei; por conseguinte, à direita de cada elemento Ei de um nó, no caso de existir uma subárvore, ela conterá elementos com chaves maiores que a chave de Ei.

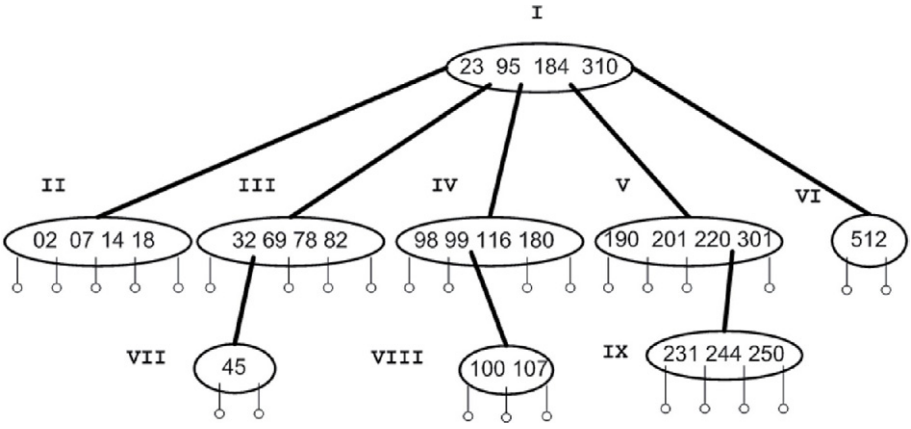
Assim, é desejável a existência de uma forma de percurso que permita a apresentação das chaves de forma ordenada crescente, que aqui denominaremos *percurso ordenado.*

Percurso ordenado (apresentação das chaves em ordem crescente)

Nessa forma de percurso, considerando-se o registro apresentado na Figura 12.4, temos (se raiz for um nó não vazio):

- para o nó raiz**,** obtém-se no campo *nro_chaves* o número de chaves Ni presentes no nó;
- inicia-se um processo repetitivo, em que um índice *j* é inicializado com 0 e vai variar até Ni-1 (de 0 até Ni-1 temos Ni chaves, portanto, em Ni-1 temos a última chave). Para cada *j* desse intervalo, percorre-se em percurso ordenado a *j*-ésima subárvore apontada no nó raiz e, na sequência, apresenta na saída a *j*-ésima chave do nó raiz;
- percorre em percurso ordenado a última subárvore apontada no nó raiz**.**

Reproduzindo novamente a árvore da Figura 12.2, vejamos como ficaria a ordem de apresentação de seus elementos num percurso ordenado, bem como a sequência dos nós visitados.



PERCURSO ORDENADO: 02-07-14-18-23-32-45-69-78-82-95-98-99-100-107-116-180-184-190-201-220-231-244-250-301-310-512

Nós visitados: I-II-I-III-VII-III-I-IV-VIII-IV-I-V-IX-V-I-VI

A seguir, tem-se a implementação da função *ordenado*, que apresenta como saída os elementos da árvore *N*-ária, segundo os critérios já descritos.

Código 12.2

```
void ordenado(no *Raiz)
{
  int j, Ni;

  if (Raiz != NULL)
  {
     Ni = Raiz->nro_chaves;                    // Obtém o número de chaves Ni do nó i
     for (j = 0; j < Ni; j++)                  // Percorre as Ni chaves do nó i
     {
       ordenado(Raiz->apontadores[j]);         // Percorre a j-ésima subárvore do nó i
       printf("%d ", Raiz->chaves[j]);         // Apresenta a j-ésima chave do nó i
     }
     ordenado(Raiz->apontadores[j]);           //Percorre a última subárvore do nó i
   }
}
```

Busca por um elemento (por meio de sua chave)

Outra operação básica importante refere-se à busca de determinada chave associada a um elemento da árvore.

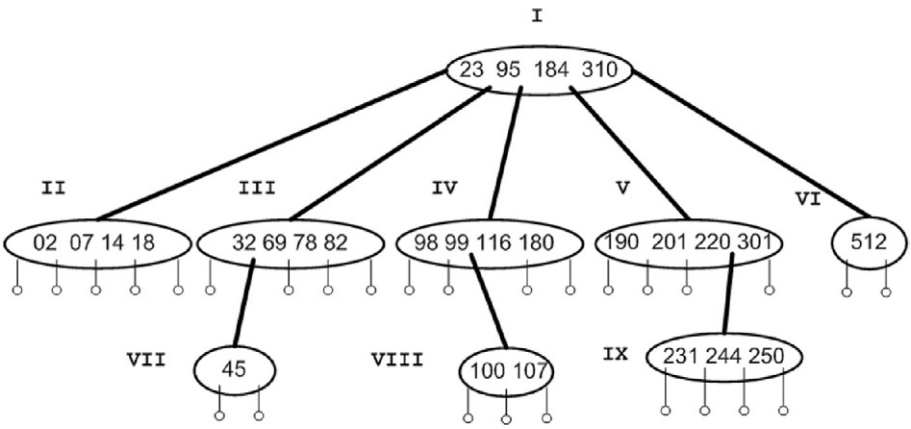
Para realizar essa tarefa, inicialmente vamos considerar que o número máximo de elementos dentro de um nó é reduzido a ponto de ser mais interessante percorrer essa lista interna ao nó de forma sequencial do que usando outra abordagem qualquer de busca em listas.

Dessa forma, considerando-se o registro apresentado na Figura 12.4, temos que a ideia geral dessa BUSCA para determinada chave Ch, usando recursividade, seria:

- verifica se raiz é nó vazio (NULL), então retorna o endereço NULL (como apontador do nó) e a posição -1, sinalizando que a chave não foi encontrada;
- caso contrário, inicializa-se *i* com o endereço do nó raiz. Para o nó *i*, no campo *nro_chaves* obtém-se o número de chaves Ni presentes no nó e coloca-se em prática os passos seguintes;

- inicia-se um processo repetitivo em que um índice *j* é inicializado com 0 e varia até Ni-1 (de 0 até Ni-1 temos Ni chaves, portanto, em Ni-1 temos a última chave);
- dentro do processo repetitivo: verifica se (Ch = *j*-ésima chave), então finaliza o processo de BUSCA (*j* já estará com a posição da chave procurada dentro do nó *i*). Caso contrário, verifica se (Ch < *j*-ésima chave), então interrompe o processo repetitivo, tendo em *j* a posição do vetor apontadores do nó *i*, que aponta a subárvore onde pode estar Ch;
- Se o processo repetitivo chegou até o final é porque ou Ch < *j*-ésima chave ou Ch é maior que a maior chave armazenada no nó *i*. Nas duas situações, temos em *j* a posição do vetor apontadores do nó *i*, que contém o endereço da subárvore do nó *i*, por onde será reiniciado o processo de BUSCA pela posição de inserção de Ch.

Reproduzindo novamente a árvore da Figura 12.2, vejamos como ficaria a ordem das chaves e dos nós visitados na busca pela chave 100, bem como os valores de *i* e *j* retornados.



Ordem de chaves visitadas: 23-95-184-98-99-116-100

Ordem dos nós visitados: I-IV-VIII

Retorno: i com o endereço do nó VIII

j com a posição 0

Agora, vejamos como ficaria na busca pela chave 203 (inexistente).

Ordem de chaves visitadas: 23-95-184-310-190-201-220

Ordem dos nós visitados: I-V

Retorno: i com o endereço NULL

j com a posição -1

Para fixar, mostre como ficaria, nas buscas pelas chaves 250 e 181, as ordens de chaves e dos nós visitados, bem como os valores de *i* e *j* retornados.

Em seguida, tem-se a implementação da função *busca_no*, que buscará pela chave Ch na árvore raiz, retornando em *i* o endereço do nó onde está a chave Ch e em *j* a posição dessa chave dentro desse nó. No caso de inexistência da chave Ch, retornará *i* como NULL e *j* como -1, conforme já descrito.

Código 12.3

```
void busca_no(no *Raiz, tipo_dado Ch, no **i, int *j)
{
  int Ni;
  if (Raiz == NULL)          // Não existe mais subárvores para procurar Ch, ou seja, Ch não existe
  {
      *i = NULL;
      *j = -1;
  }
  else
  {
     *i = Raiz;
     Ni = (*i)->nro_chaves;
     for (*j = 0; *j < Ni; (*j)++)        // Primeira posição do vetor chaves é 0(zero)
     {
       printf("Chave visitada= %d\n", (*i)->chaves[*j]); //Mostra a chave visitada durante a busca
       if ((*i)->chaves[*j] == Ch)              // Encontrou a chave Ch procurada
       {
         return;                                // Finaliza a busca, sendo i o nó e j a posição da chave Ch
       }
       else if (Ch < (*i)->chaves[*j])   // Chave Ch deve estar na na j-ésima subárvore apontada no nó i
              break;                     // Interrompe o laço para buscar a chave Ch na j-ésima subárvore
     }
     busca_no((*i)->apontadores[*j],Ch,&(*i),&(*j));
  }
}
```

Uma questão que pode ser levantada é a situação em que o número de chaves em cada nó se torna tão elevado que a substituição da busca sequencial pelas chaves dentro de um nó por outro mecanismo de busca pode trazer algum benefício. Assim, para ampliar seu conhecimento, tente alterar a função *busca_no* para que a busca pelas chaves dentro de um nó use um dos mecanismos vistos no Capítulo 4 que apresente melhor desempenho no caso de grande volume de dados em uma lista ordenada.

Inserção de um elemento (chave) numa árvore N-*ária*

Quando abordamos a inserção numa árvore *N*-ária (de busca), adotamos uma ideia similar à usada nas árvores binárias, vistas no Capítulo 11, no que se refere à distribuição das chaves de forma ordenada. Essa distribuição foi descrita novamente na definição de árvores *N*-árias dada no início deste tópico. Além disso, numa árvore *N*-ária, para explorarmos o benefício de poder armazenar várias chaves num nó, a inserção sempre priorizará o preenchimento de nós nos níveis mais inferiores possíveis da árvore.

Atenção

Adotaremos que na árvore *N*-ária não será admitida a possibilidade de existência de chaves duplicadas.

A partir dessas considerações, temos que a inserção se dará em duas etapas básicas: *localização* do nó e posição dentro dele para a inserção da chave, e *inserção com possível rearranjo* das chaves e subárvores desse nó para manutenção da ordenação.

Localização do ponto para inserção da chave (elemento)

Na localização do nó e posição para inserção da chave, usaremos uma variação da função *busca_no* vista, a que chamaremos *busca_no_ins*. A

diferença é que *busca_no_ins* não procura pela chave, e sim pela posição na árvore para sua inserção. No caso de existência da chave a ser inserida, devolve um alerta, evitando sua duplicidade. Assim, *busca_no_ins* contém os seguintes passos:

- se raiz é nó vazio (NULL) é porque será necessário criar um novo nó para inserção da chave;
- caso contrário, inicializa-se *i* com o endereço do nó raiz. Para o nó *i*, obtém-se, no campo *nro_chaves*, o número de chaves Ni presentes no nó;
- inicia um processo repetitivo, em que um índice *j* é inicializado com 0 e vai variar até Ni-1 (de 0 até Ni-1 temos Ni chaves, portanto, em Ni-1 temos a última chave);
- dentro do processo repetitivo: verifica se (Ch = *j*-ésima chave), então retorna a posição -1 para *j*, sinalizando que a chave já existe na árvore. Caso contrário, verifica se (Ch < *j*-ésima chave), então se existe espaço para a chave Ch no nó *i* e interrompe o processo de BUSCA, tendo em *j* a posição do vetor chaves do nó *i*, onde deverá ser inserida a chave Ch. Caso não exista espaço, interrompe o processo repetitivo, tendo em *j* a posição do vetor apontadores do nó *i* que aponta a subárvore em que será inserida Ch;
- se o processo repetitivo chegou até o final é porque ou Ch < *j*-ésima chave e não existe espaço para inserção da Ch no nó *i*, ou Ch é maior que a maior chave armazenada no nó *i* e não existe espaço para sua inserção nesse nó. Nas duas situações, temos em *j* a posição do vetor apontadores do nó *i*, que contém o endereço da subárvore do nó *i* por onde será reiniciado o processo de BUSCA pela posição de inserção de Ch.

A seguir, tem-se a implementação da função *busca_no_ins*, que buscará a posição de inserção da chave Ch, conforme já descrito. Note que GRAU será uma constante contendo o grau da árvore *N*-ária.

Código 12.4

```
void busca_no_ins(no *Raiz, tipo_dado Ch, no **i, int *j)
{
  int Ni;

  // Se (Raiz == NULL) é porque não existe mais subárvores para procurar Ch e o nó i está sem espaço.
  // Finaliza a busca, já tendo a posição  j do nó i como ponto onde o novo nó deverá ser criado para inserção de Ch
  if (Raiz != NULL)
  {
     *i = Raiz;
     Ni = (*i)->nro_chaves;
     for (*j = 0; *j < Ni; (*j)++)          // Procura pela 1ª chave do nó i maior que Ch
     {
       if ((*i)->chaves[*j] == Ch)                  // Chave Ch já existe na árvore
       {
         *j = -1;
         return;                                    //  Finaliza a BUSCA
       }
       else if (Ch < (*i)->chaves[*j])       // Encontrou no nó (*i) uma chave maior que  Ch
            {
                if (Ni  <  GRAU-1)    // Existe espaço no nó i para inserção de uma chave Ch menor que uma já existente
                   return;            // Finaliza a busca, sendo i o nó para inserção e j a posição da chave
                else
                   break;             // Interrompe o laço para buscar a chave Ch na j-ésima subárvore
            }
     }
     busca_no_ins((*i)->apontadores[*j],Ch,&(*i),&(*j));
  }
}
```

Inserção da chave (elemento) na árvore N-*ária*

Para a inserção de uma chave, devem ser verificadas as seguintes etapas:

- verificar se Ch será a primeira chave da árvore. Sendo a primeira chave, deve ser criado e inicializado um novo nó, que passará a ser a raiz da árvore;
- caso contrário, inicializa-se *i* com o endereço do nó raiz e *j* como posição 0 dentro do nó;
- chama pela função *busca_no_ins*, que devolverá se Ch já existe ou se os valores de *i* e *j* orientarão a inserção;

- verifica se não existe mais espaço no nó *i* para a inserção de Ch, então cria, inicializa e insere Ch como a primeira chave de um novo nó *E*, associando-o com a *j*-ésima subárvore do nó ***i*;**
- caso contrário, verifica se Ch será a última chave do nó *i*, então cria uma subárvore vazia na (j+1)-ésima posição;
- caso contrário, rearranja o nó *i*, mantendo a ordenação de suas chaves e subárvores. Para isso, desloca todas as chaves e subárvores (da *j*-ésima até a última posição preenchida nesse nó) de uma posição para a direita e cria uma subárvore vazia na *j*-ésima posição;
- por fim, para completar as etapas 5 e 6, insere Ch na *j*-ésima posição e incrementa o número de chaves do nó.

A seguir, tem-se a implementação da função *inserir_n_naria*, que fará a inserção da chave Ch, conforme já descrito. Note que GRAU será uma constante contendo o grau da árvore *N*-ária.

Código 12.5

```
void inserir_n_naria (no **Raiz, tipo_dado Ch)
{
  no *i;                              // Apontador para o nó onde será inserida a chave Ch
  int j;                              // Posição no nó para inserção ordenada da chave Ch
  int Ni, k;

  if (*Raiz == NULL)                          //Primeiro elemento da árvore
  {
    *Raiz = (no *)malloc(sizeof(no));         // Aloca espaço na memória correspondente ao nó Raiz
    for (k = 0; k < GRAU-1; k++)
    {                                         //Inicializa o nó com vazio
       (*Raiz)->chaves[k] = -1;
       (*Raiz)->apontadores[k]= NULL;
    }
    (*Raiz)->apontadores[k] = NULL;           // Inicializa o endereço da última subárvore
    (*Raiz)->chaves[0] = Ch;                  // Insere o conteúdo (chave) como primeira chave do nó E
    (*Raiz)->nro_chaves = 1;                  // Inicializa o número de chaves contidas no nó
  }
  else
  {
     i = *Raiz;                               // Inicializa i e j
     j=0;
     busca_no_ins(*Raiz,Ch,&i,&j);
     if (j == -1)
         printf("Chave ja existente na arvore\n");
     else
     {
       Ni = i->nro_chaves;
       if (Ni == GRAU-1)                      // Não existe espaço no nó para inserção de Ch
       {                                      // Cria novo nó E
          no *E;
          E = (no *)malloc(sizeof(no));       // Aloca espaço na memória correspondente ao nó E
          for (k = 0; k < Ni; k++)
          {                                   // Inicializa o nó com vazio
            E->chaves[k] = -1;
            E->apontadores[k]= NULL;
          }
          E->apontadores[k]= NULL;            // Inicializa o endereço da última subárvore
          E->chaves[0] = Ch;                  // Insere o conteúdo (chave) como primeira chave do nó E
          E->nro_chaves = 1;                  // Inicializa o número de chaves contidas no nó
          i->apontadores[j] = E;              // Insere o nó E como a j-ésima subárvore do nó i
       }
       else
       {
         if (j == Ni)                         // A chave Ch será a última do nó i
            i->apontadores[j+1] = NULL;       // Inicializa o endereço da última (jésima+1) subárvore
         else
         {                                    // Rearranja o nó i, mantendo a ordenação de suas chaves e subárvores
            i->apontadores[Ni+1] = i->apontadores[Ni];     // Desloca última subárvore
            for (k = Ni; k > j; k--)          // Abre espaço para a chave Ch, deslocando as chaves e subárvores
            {
              i->chaves[k] = i->chaves[k-1];
              i->apontadores[k] = i->apontadores[k-1];
            }
            i->apontadores[j] = NULL;         // Inicializa o endereço da j-ésima subárvore
         }
         i->chaves[j] = Ch;                   //Insere a chave Ch na posição j
         i->nro_chaves = i->nro_chaves + 1;   // Incrementa o número de chaves contidas no nó i
       }
     }
  }
}
```

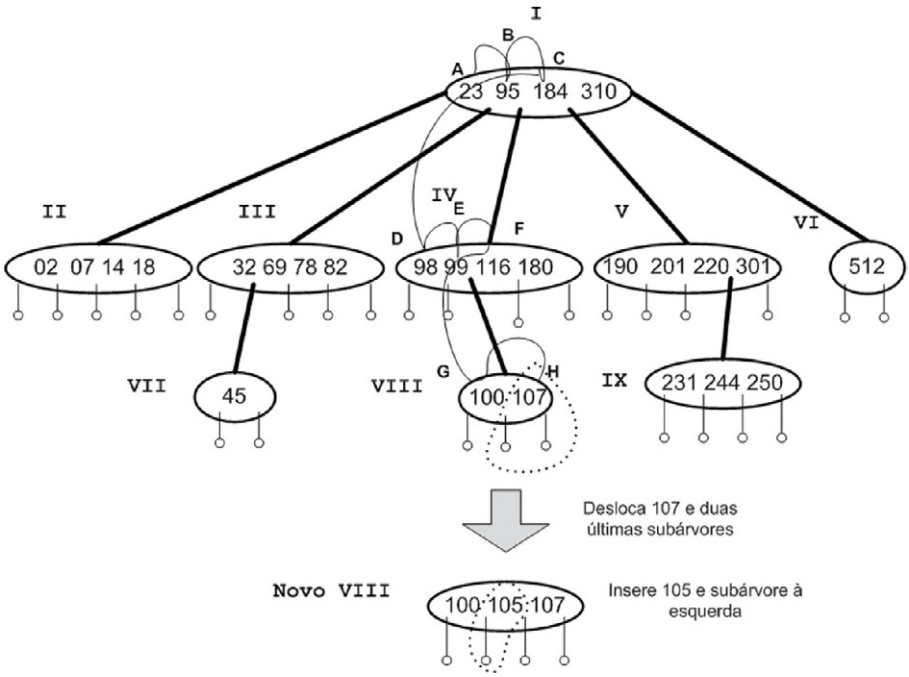


FIGURA 12.5: Inserção da chave 105.

Para ilustrar como será a inserção de chaves numa árvore *N*-ária, vejamos como ficaria a inserção da chave 105 na árvore da Figura 12.5.

Visita: nó I, chaves (A)23-(B)95-(C)184
    não existe espaço no nó I
Visita: nó IV, chaves (D)98-(E)99-(F)116
    não existe espaço no nó IV
Visita: nó VIII, chaves (G)100-(H)107
    existe espaço no nó VIII
    desloca subárvores e chaves para abrir espaço para 105
    insere 105
    cria subárvore vazia à esquerda de 105

Para fixar, mostre como ficaria a inserção das chaves 01 e 513.

É importante notar que a forma de inserção apresentada não considera a possibilidade de existirem subárvores não vazias apontadas por nós que não estejam completos (máximo de chaves). Além disso, não existe uma preocupação em redistribuir as chaves pelos nós, de forma a garantir que um novo nível na árvore somente seja iniciado quando todos os nós dos níveis existentes forem completos (tenham atingido o número máximo de chaves).

Remoção de um elemento (chave) numa árvore N-*ária*

Ao pensarmos em simplicidade na remoção de elementos numa estrutura de dados qualquer, em geral, o que se adota é não remover de fato a chave, mas apenas sinalizar de alguma forma (marca) que naquela posição aquela chave não existe mais. Essa abordagem oferece maior facilidade na operacionalização da remoção, mas provoca desperdício de espaço de armazenamento.

No caso das árvores *N*-árias, além do desperdício de espaço mencionado, temos dificuldade em garantir a manutenção da ordenação com tal ação. Isso tudo leva à ideia de reaproveitar o espaço deixado numa futura reinserção da mesma chave. Para chaves diferentes da removida, não há como garantir a posição em relação às *chaves anterior* e *posterior* na árvore, o que obrigaria sua reorganização. Como *chave anterior* consideramos aquela que possui valor imediatamente menor do que a chave em questão e, de forma análoga, como *chave posterior*, aquela com valor imediatamente maior. Por exemplo, numa lista de chaves com 4-9-12-34-47 para a chave 12, temos 9 como sua chave anterior e 34 como sua chave posterior.

Para melhor compreensão, vejamos como ficaria a remoção da chave 95 presente na árvore da Figura 12.6.

Note que, ao remover 95, deixando em seu lugar uma marca, toda a ordenação e a estruturação da árvore ficarão mantidas, porém com o desperdício daquele espaço de armazenamento.

Se for necessária a reinserção da chave 95, basta armazená-la no seu lugar original (veja a Figura 12.7a). De outro lado, imagine que desejamos inserir a chave 99. Seria possível simplesmente reaproveitar o espaço marcado? A resposta é não, porque perderíamos a ordenação entre as chaves (veja a Figura 12.7b).

Apesar de simples, o reaproveitamento, sendo limitado a chaves iguais (removida e inserida), apenas se mostra adequado quando a situação-problema apresenta com frequência esse tipo de ocorrência, o que não é comum.

Isso nos leva à necessidade de buscar alternativas para a remoção de chaves em árvores *N*-árias. Nesse contexto, nos deparamos com a possibilidade de repetir a ideia de remoção vista para as árvores binárias. Assim, temos as seguintes situações para a chave a ser removida:*

1. **quando não apresenta subárvore à esquerda**: basta rearranjar (compactar) o registro do nó, deslocando todas as chaves e subárvores

*Considerando-se o registro da Figura 12.4, o que aqui chamaremos de *subárvore à esquerda* da *j*-ésima chave de um nó *i*, seria o que lá consideramos a *j*-ésima subárvore apontada por *i*, por conseguinte, a *subárvore à direita* desda chave seria a (j+1)-ésima subárvore apontada por *i*.

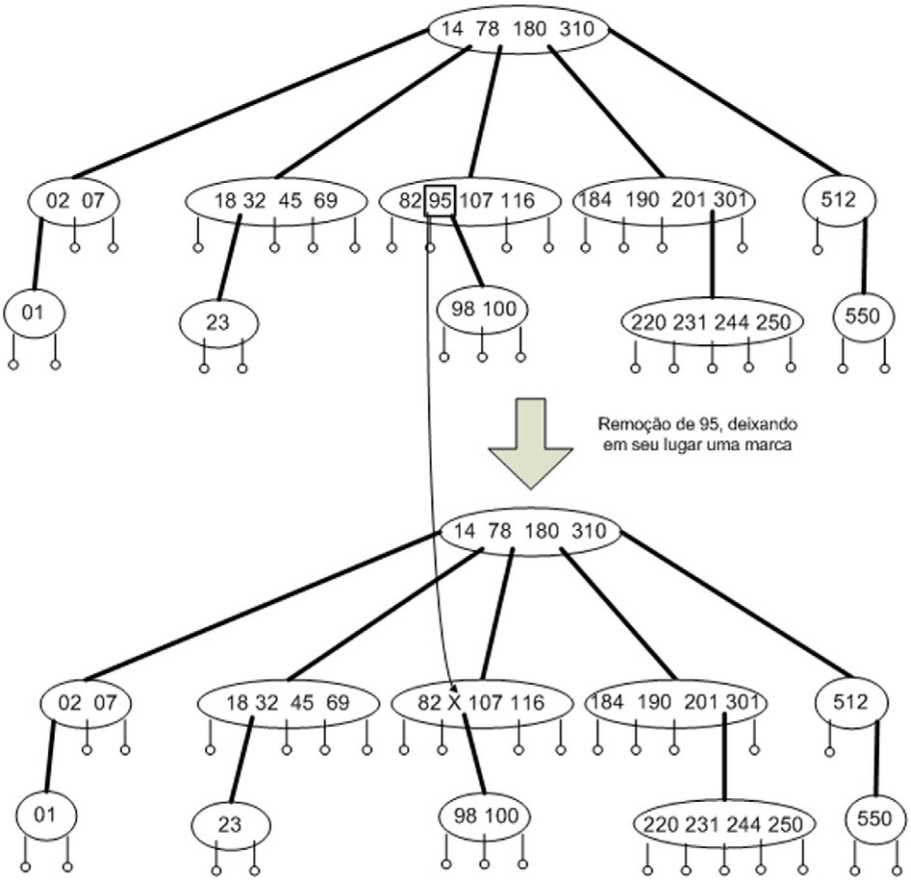


FIGURA 12.6: Remoção colocando uma marca na chave removida.

(da posição posterior no nó até a última existente nesse nó) de uma posição para a esquerda, diminuindo em um o número de chaves no nó. Se a chave for a última do nó, basta deslocar a subárvore à direita de uma posição para a esquerda, diminuindo em um o número de chaves no nó;

2. **quando não apresenta subárvore à direita**: basta rearranjar (compactar) o registro do nó, deslocando todas as chaves (da posição posterior no nó até a última existente nesse nó) de uma posição para a esquerda e deslocando todas as subárvores (da subárvore à direita da chave posterior no nó até a última apontada pelo nó) também de uma posição para a esquerda, diminuindo em um o número de chaves no nó. Se a chave for a última do nó, neste caso, basta diminuir em um o número de chaves do nó;

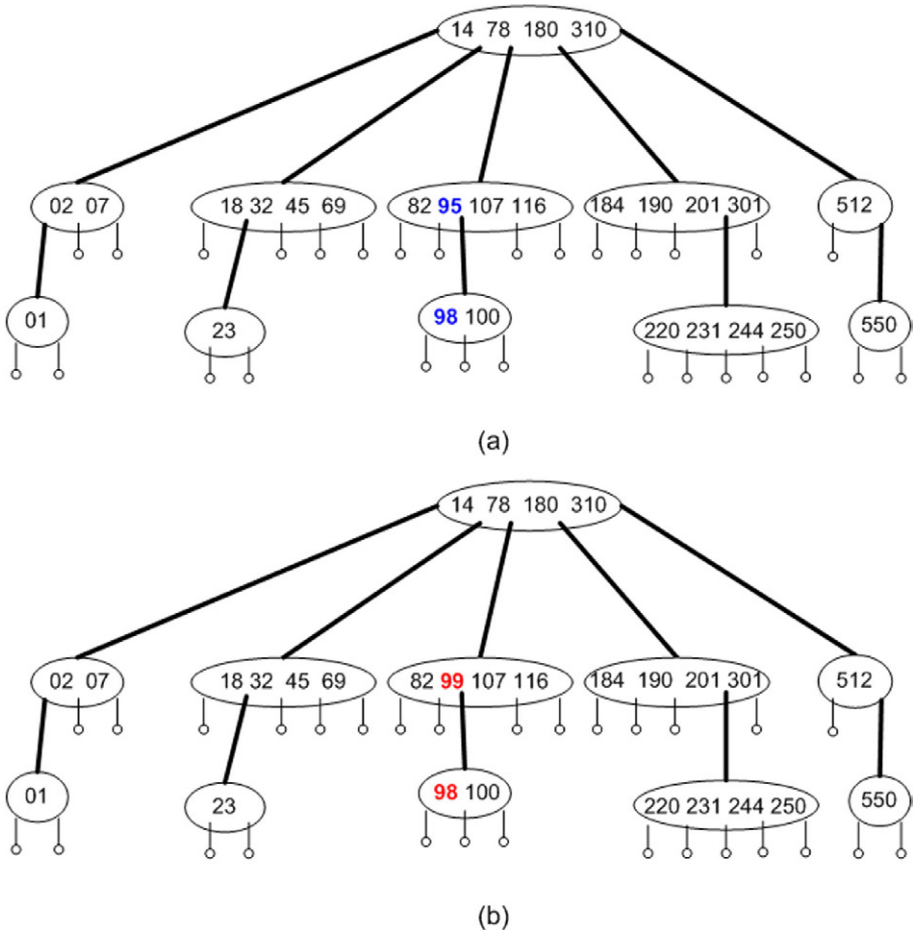


FIGURA 12.7: Reinserção da chave 95 e inserção da chave 99.

3. **quando apresenta subárvores à esquerda e à direita**: deve ser deslocada para a posição em que se encontra a chave removida, sua chave anterior e repetir as etapas 1, 2 e 3, tendo em mente a remoção da chave deslocada do nó em que ela estava originalmente. Esse processo se repete enquanto há necessidade de deslocar chaves anteriores[†]
4. Em todos os casos, quando a chave removida (ou última deslocada) é única no nó, deve-se eliminá-lo da árvore.

[†]Assim como ocorre nas árvores binárias, a escolha da chave a ser deslocada poderia ser pela chave posterior em vez da anterior.

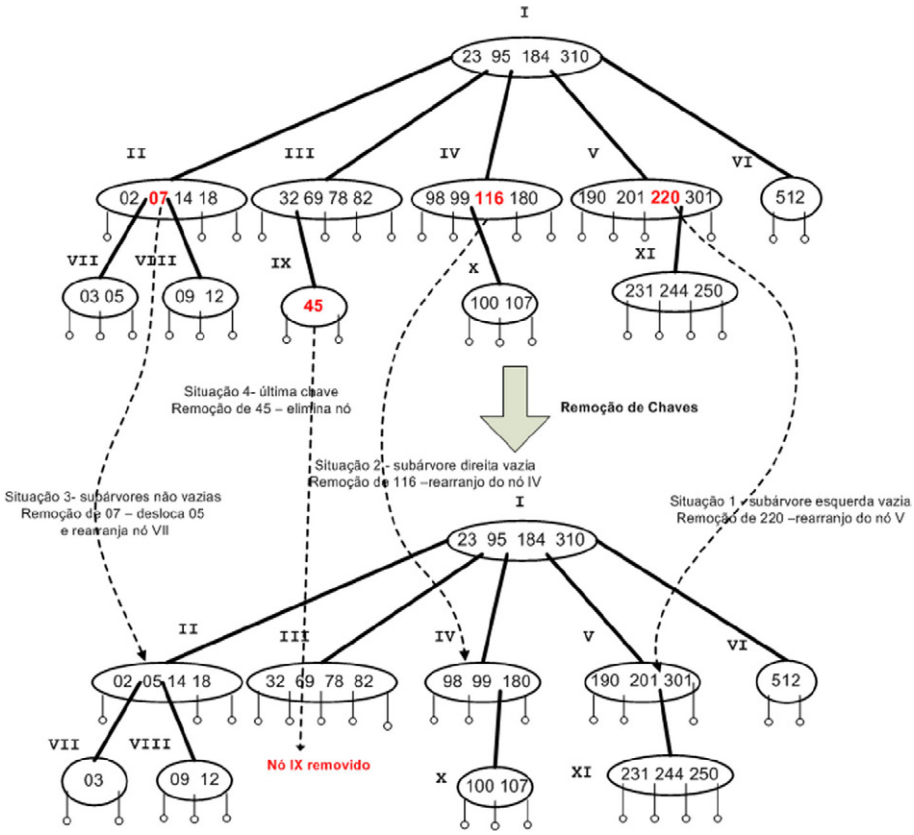


FIGURA 12.8: Remoções de chaves numa árvore *N*-ária.

Para ilustrar, vamos observar um exemplo para cada situação apresentada (veja a Figura 12.8), em que:

- para a situação 1, considere a remoção da chave 220. Essa chave não possui subárvore à esquerda, assim, o nó V é compactado deslocando-se as chaves e subárvores à direita de 220 de uma posição à esquerda, nas listas de chaves e apontadores para subárvore desse nó;
- para a situação 2, considere a remoção da chave 116. Essa chave não possui subárvore à direita, assim, o nó IV é compactado eliminando-se o apontador da subárvore vazia à direita de 116 e deslocando-se as chaves e subárvores restantes à direita de 116 de uma posição à esquerda, nas listas de chaves e apontadores para subárvore desse nó;

- para a situação 3, considere a remoção da chave 07. Essa chave possui subárvores à esquerda e à direita. Nessa situação, a chave 07 é substituída pela maior chave (05) existente na subárvore à esquerda, e o nó V, em que estava a chave deslocada, é compactado;
- para a situação 4, considere a remoção da chave 45. Nessa situação, 45 é a única chave do nó IX, que, portanto, deve ser eliminado.

No caso de necessidade de estudos mais aprofundados, procure elaborar funções que contemplem os algoritmos apresentados para a remoção de chaves numa árvore *N*-ária, tanto por meio da colocação de marcas, como da remoção com rearranjo dos nós. Lembre-se de verificar a necessidade de eventuais ajustes na função de inserção, anteriormente apresentada, para seu correto funcionamento em conjunto com suas implementações.

Vantagens e uma aplicação típica de árvores *N*-árias

De modo geral, foi possível observar que, ao permitir que sejam colocadas várias chaves num único nó, temos a possibilidade de reduzir de forma significativa o número de nós necessários e a altura da árvore. Assim, espera-se menor número de acessos a diferentes nós quando se deseja localizar determinada chave (elemento) dentro da estrutura de árvore *N*-ária, se comparada à árvore binária.

Com isso, podemos considerar que quanto maior o volume de dados a ser armazenado numa estrutura de árvore, maior será a vantagem em utilizar uma árvore *N*-ária. Porém, não podemos perder de vista que sua manipulação apresenta maior necessidade de reorganização da estrutura de árvore existente, o que aumenta a complexidade de implementação e também o tempo despendido com a reorganização da árvore.

Por causa dessas características, esse tipo de árvore se mostra adequada para uso em situações nas quais os dados são armazenados em dispositivos externos (por exemplo, disco). Esses dispositivos são demasiado lentos, se comparados com a memória interna do computador. Isso decorre principalmente do tempo gasto para o correto posicionamento do dispositivo para a leitura dos dados. De outro lado, quando é feito o acesso a esses dispositivos, a leitura sequencial dos dados subsequentes, a partir do posicionamento, não é demorada. Assim, considerando-se que a cada leitura seria possível ler todos os elementos de um nó da árvore, teríamos um ganho significativo de desempenho com o uso desse tipo de estrutura, além da forte redução do espaço de busca que ele oferece. Essa redução decorre do fato de que, ao se definir a subárvore seguinte a ser investigada, todas as restantes (que carregam um número considerável de

elementos) já são eliminadas do espaço de busca restante. Apenas para exemplificar, enquanto numa árvore binária espera-se eliminar em média 50% do espaço de busca a cada nova subárvore escolhida, numa árvore *N*-ária, por exemplo de grau 5, a redução média pode chegar a 80% do espaço de busca.

Eficiência na busca numa árvore *N*-ária

Da mesma forma como visto para as árvores binárias no Capítulo 11, as operações básicas numa árvore *N*-ária tem tempo proporcional à sua altura. A altura da árvore, nesse caso, dependerá da ordem de inserção das chaves, do número *N* total de chaves e da ordem *m* da árvore.

Assim, ao considerarmos a situação ideal de uma árvore *N*-ária, em que todos os nós estão preenchidos com o número máximo de chaves possível e todos os nós folhas estão no mesmo nível, teremos, no pior caso, a execução das operações num tempo $\mathbf{O(\log_m N)}$‡**.**

Para termos uma comparação superficial com as árvores binárias (considerando tanto a árvore *N*-ária quanto a binária com a melhor situação de distribuição de nós), teríamos, de forma aproximada, o seguinte tempo de execução:

- Com N = 1.000 e m = 10

Árvore binária ≈ $O(\log_2 1.000) \approx O(9,97)$

Árvore *N*-ária ≈ $O(\log_{10} 1.000) \approx O(3)$

- Com N = 10.000.000 e m = 10.

Árvore binária ≈ $O(\log_2 10.000.000) \approx O(23,25)$

Árvore *N*-ária ≈ $O(\log_{10} 10.000.000) \approx O(7)$

Esses exemplos ilustram o potencial de maior eficiência das árvores *N*-árias em termos de tempo de execução das operações básicas, principalmente quando se considera um grande volume de dados. Para aprofundamento sobre a eficiência de uma árvore *N*-ária, recomendamos a leitura do material mencionado na seção "Para Saber Mais" deste capítulo.

Vamos programar

Para ampliar a abrangência do conteúdo apresentado, nas próximas seções veremos implementações nas linguagens de programação Java e Phyton.

‡Em termos de eficiência em algoritmos, a notação O (..) representa a ordem associada ao que se deseja mensurar. Assim, O (log) representa uma ordem logarítmica.

Java

Código 12.1

```
public class NoNaria {

      private int nroChaves;
      public int[] chave;
      public NoNaria[] nosFilhos;
       private int grau;


      public NoNaria(int grau) {
            nroChaves = 0;
            nosFilhos = new NoNaria[grau];
            chave = new int[grau-1];
              this.grau = grau;
      }

       public boolean isFull (){
         return nroChaves == this.grau-1;
       }

       public int getNroChaves (){
          return this.nroChaves;
       }

       public void adicionaValor (int valor){
         this.chave[nroChaves] = valor;
         this.nroChaves+=1;
       }
}

public class ArvoreNaria {

  NoNaria raiz;
  private int grau;

  public ArvoreNaria(int grau) {
    this.raiz = null;
    this.grau = grau;
  }
```

Código 12.2

```
  public void ordenado(){
    ordenado(this.raiz);
  }

  private void ordenado(NoNaria no){
    if(no != null){
      int j = 0;
      for (j = 0; j < no.getNroChaves(); j++){
        ordenado(no.nosFilhos[j]);
        System.out.print(no.chave[j] + " ");
      }
      ordenado(no.nosFilhos[j]);
    }
  }
```

Código 12.3 e Código 12.4

```
public NoNaria busca_no(int valor) {
  if (this.raiz == null) {
    return null; // Nao existe o valor
  } else {
    return busca_no(this.raiz, valor);
  }
}

private NoNaria busca_no(NoNaria no, int valor) {
  if (no == null) {
    return null;
  } else {
    for (int i = 0; i < no.getNroChaves(); i++) {
      if (valor == no.chave[i]) {
        return no; // Valor duplicado
      }
      if (valor < no.chave[i]) {
        return busca_no(no.nosFilhos[i], valor);
      }
    }
    if (no.getNroChaves() == grau - 1) {
      if (valor > no.chave[grau - 2]) {
        return busca_no(no.nosFilhos[grau - 1], valor);
      } else {
        return null;
      }
    } else {
      return null;
    }
  }
}
```

Código 12.5

```
public void inserir_n_naria(int valor) {
  if (this.raiz == null) {
    this.raiz = new NoNaria(this.grau);
    this.raiz.adicionaValor(valor);
  } else {
    adicionaReg(this.raiz, valor, raiz, 0);
  }
}

private void adicionaReg(NoNaria no, int valor, NoNaria pai, int indice) {
  if (no == null) {
    no = new NoNaria(this.grau);
    no.adicionaValor(valor);
    pai.nosFilhos[indice] = no;
  } else {
    for (int i = 0; i < no.getNroChaves(); i++) {
      if (valor == no.chave[i]) {
        return; // Valor duplicado
      }
      if (valor < no.chave[i]) {
        adicionaReg(no.nosFilhos[i], valor, no, i);
        return;
      }
    }
    if (no.getNroChaves() == grau - 1) {
      if (valor == no.chave[grau - 2]) {
        return; // Valor duplicado
      }
      if (valor > no.chave[grau - 2]) {
        adicionaReg(no.nosFilhos[grau - 1], valor, no, grau - 1);
        return;
      }
    } else {
      no.adicionaValor(valor);
      return;
    }
  }
}
}
```

Phyton

Código 12.1

```
class NoNaria(object):
            def __init__(self,grau):
                  self.nroChaves = 0
                  self.chave=[None]*(grau)
                  self.nosFilhos = [None]*(grau)
                  self.grau=grau

            def getNroChaves(self):
                  return self.nroChaves


class ArvoreNaria(object):
            def __init__(self,grau):
                  self.raiz = None
                  self.grau=grau
```

Código 12.2

```
            def ordenado1(self):
                  aux=self.raiz
                  self.ordenado(aux)

            def ordenado(self,no):
                  if no <> None:
                        j=0
                        while j < no.nroChaves:
                              self.ordenado(no.nosFilhos[j])
                              print no.chave[j]
                              j+=1
                        self.ordenado(no.nosFilhos[j])
```

Código 12.3 e Código 12.4

```
      def busca_no1(self,valor):
                  if self.raiz == None:
                        print "Nao Existe"
                        return None # Nao existe o valor
                  else:
                        return self.busca_no(self.raiz, valor)

      def busca_no(self,no, valor):
                  if no==None:
                        print "Nao Existe"
                        return None
                  else:
                        i=0
                        while i < no.getNroChaves():
                              if valor == no.chave[i]:
                                    print "Existe"
                                    return no
                              if valor< no.chave[i]:
                                    return self.busca_no(no.nosFilhos[i],valor)
                              i+=1
                        if no.getNroChaves() == self.grau - 1:
                              if valor > no.chave[self.grau - 2]:
                                return self.busca_no(no.nosFilhos[self.grau - 1], valor)
                              else:
                                    print "Nao Existe"
                                    return None
                        else:
                              print "Nao Existe"
                              return None
```

Código 12.5

```
def inserir_n_naria(self,valor):
        if self.raiz == None:
            self.raiz = NoNaria(self.grau)
            self.raiz.chave[self.raiz.nroChaves]=valor
            self.raiz.nroChaves+=1
            i=0
            while i <  self.grau:
                self.raiz.nosFilhos[i]= NoNaria(self.grau)
                i+=1
        else:
            self.adicionaRec(self.raiz, valor, self.raiz, 0)

    def adicionaRec(self,no, valor, pai, indice):
        if no==None:
            no = NoNaria(self.grau)
            no.chave[no.nroChaves]=valor
            no.nroChaves+=1
            i=0
            while i <  self.grau:
                if no.nosFilhos[i] == None:
                    no.nosFilhos[i]= NoNaria(self.grau)
                i+=1
            pai.nosFilhos[indice] = no
        else:
            i=0
            while i < no.getNroChaves():
                if valor == no.chave[i]:
                    return  #Valor duplicado
                if valor < no.chave[i]:
                    self.adicionaRec(no.nosFilhos[i], valor, no, i)
                    return
                i+=1
            if no.getNroChaves() == self.grau - 1:
                if valor == no.chave[self.grau - 2]:
                    return
                if valor > no.chave[self.grau - 2]:
                 self.adicionaRec(no.nosFilhos[self.grau-1],valor,no,self.grau-1)
                    return
            else:
                no.chave[no.nroChaves]=valor
                no.nroChaves+=1
                i=0
                while i <  self.grau:
                    if no.nosFilhos[i] == None:
                        no.nosFilhos[i]= NoNaria(self.grau)
                    i+=1
                return
```

Para fixar!

1. Utilizando o algoritmo para inserção definido neste capítulo, desenhe numa folha de papel duas árvores *N*-árias contendo 100 chaves geradas aleatoriamente. Na primeira delas, faça a inserção na ordem gerada aleatoriamente; na segunda, use os mesmos valores gerados aleatoriamente, mas numa ordem de inserção que distribua de modo homogêneo as chaves pelas subárvores, procurando ter uma árvore com a menor altura possível. Compare as duas árvores e procure tirar suas conclusões sobre a importância de ter as chaves distribuídas homogeneamente pelas subárvores.

2. Utilizando as implementações que você fez para os algoritmos apresentados para a remoção de chaves numa árvore *N*-ária, tanto por meio da colocação de marcas quanto na remoção com rearranjo dos nós da árvore, e usando a função para inserção que estudamos, procure criar a árvore da Figura 12.7 e realizar as remoções ali sugeridas. Teste também as funções de percurso ordenado para verificar a lista de chaves antes e após as remoções.

Para saber mais

Dentro do propósito geral deste livro, que é apresentar de forma clara e numa linguagem acessível uma visão geral básica das principais estruturas de dados existentes, orientamos aqueles que desejam se aprofundar nos estudos sobre árvores *N*-árias que leiam da Parte V, Capítulos 18, do livro *Algoritmos: teoria e prática* (Wirth, 1989), e do Item 7.3 do livro *Estruturas de dados usando C* (Tenenbaum, 2004).

Navegar é preciso

Diferentemente das árvores binárias, quase não existem ferramentas disponíveis na internet para árvores *N*-árias gerais. O que se encontra, geralmente, são ferramentas para tipos específicos de árvores *N*-árias, principalmente para árvores B, que serão vistas no Capítulo 13. Assim, deixaremos para relacionar algumas das ferramentas que constam do "Navegar é preciso" do capítulo seguinte. De outro lado, como em computação tudo é dinâmico e novidades surgem a cada minuto, não deixe de procurar por essas ferramentas – quem sabe você encontra algo interessante?

Exercícios

1. Crie uma função de percurso que devolva a lista das chaves que fazem parte de determinado nível da árvore.

2. Faça as alterações necessárias nas funções *busca_no_ins* e *inserir_n_naria* para permitir a inserção de chaves duplicadas.

3. Para o conjunto de dados da Figura 12.2 (02-07-14-18-23-32-45-69-78-82-95-98-99-100-107-116-180-184-190-201-220-231-244-250-301-310-512), usando as funções apresentadas nos Capítulos 11 e 12, construa uma árvore binária e uma árvore *N*-ária (grau 5), em que em ambas a ordem de inserção das chaves seja gerada de modo aleatório. A partir disso, ajuste a função *busca_no* vista para que conte quantos nós foram visitados na localização de cada chave. Ao final, devolva no programa principal os números médios de nós visitados em ambas as árvores, obtidos na localização de todas as chaves. Tente refazer o exercício, porém, definindo uma ordem de entrada em que cada árvore tenha a menor altura possível. Anote suas conclusões em seu caderno.

4. Crie uma função que devolva a relação de chaves presentes nos nós folhas de uma árvore *N*-ária.

Conseguiu? Parabéns, você já domina os conceitos fundamentais da estrutura de dados árvore *N*-ária!!!

Referências bibliográficas

CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012.

TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 2004.

WIRTH, N. *Algoritmos e estruturas de dados*. Rio de Janeiro: Prentice-Hall, 1989.

O QUE VEM DEPOIS

Ao longo deste capítulo, cujo objetivo era mostrar as principais diferenças que as árvores *N*-árias apresentam em relação ao que foi visto no Capítulo 11, procuramos, de forma sutil, estimular você a perceber que a simples construção de árvores, sem preocupação com a distribuição homogênea das chaves pelas subárvores, pode trazer algum prejuízo em termos de eficiência nas futuras buscas por chaves presentes nessas árvores. Nesse contexto, existe o conceito de *balanceamento* em árvores

de busca, em que, a cada inserção ou remoção de uma chave, procura-se reorganizar a árvore de forma que as chaves continuem distribuídas de forma homogênea pelas subárvores com o intuito de que a árvore, seja binária, seja *N*-ária, tenha a menor altura possível, para reduzir o número de nós visitados numa eventual busca por determinada chave. É isso que veremos no Capítulo 13.