2.1 Introdução

Dentre as estruturas de dados não primitivas, as listas lineares são as de manipulação mais simples. Neste capítulo, são discutidos seus algoritmos e estruturas de armazenamento.

Uma lista linear agrupa informações referentes a um conjunto de elementos que, de alguma forma, se relacionam entre si. Ela pode se constituir, por exemplo, de informações sobre os funcionários de uma empresa, sobre notas de compras, itens de estoque, notas de alunos etc. Na realidade, são inúmeros os tipos de dados que podem ser descritos por listas lineares.

Uma *lista linear*, ou *tabela*, é então um conjunto de $n \geq 0$ nós $L[1], L[2], \ldots, L[n]$ tais que suas propriedades estruturais decorrem, unicamente, da posição relativa dos nós dentro da sequência linear. Tem-se:

– se $n > 0$, $L[1]$ é o primeiro nó,

– para $1 < k \leq n$, o nó $L[k]$ é precedido por $L[k-1]$.

As operações mais frequentes em listas são a *busca*, a *inclusão* e a *remoção* de um determinado elemento, o que, aliás, ocorre na maioria das estruturas de dados. Tais operações podem ser consideradas como básicas e, por essa razão, é necessário que os algoritmos que as implementem sejam eficientes. Outras operações, também importantes, podem ser mencionadas: a alteração de um elemento da lista, a combinação de duas ou mais listas lineares em uma única, a ordenação dos nós segundo um determinado campo, a determinação do primeiro (ou do último) nó da lista, a determinação da cardinalidade da lista e muitas outras, dependendo do problema em estudo.

Casos particulares de listas são de especial interesse. Se as inserções e remoções são permitidas apenas nas extremidades da lista, ela recebe o nome de *deque* (uma abreviatura do inglês *double ended queue*). Se as inserções e as remoções são realizadas somente em um extremo, a lista é chamada *pilha*, sendo denominada *fila* no caso em que inserções são realizadas em um extremo e remoções em outro. Operações referentes a esses casos particulares serão analisadas individualmente.

O tipo de armazenamento de uma lista linear pode ser classificado de acordo com a posição relativa (sempre contígua ou não) na memória de dois nós consecutivos na lista. O primeiro caso corresponde à *alocação sequencial de memória*, enquanto o segundo é conhecido como *alocação encadeada*. A escolha de um ou outro tipo depende essencialmente das operações que serão executadas sobre a lista, do número de listas envolvidas na operação, bem como das características particulares dessas listas. Nas seções que se seguem, tais alocações e suas características serão discutidas.

O capítulo está organizado da seguinte maneira. O estudo é iniciado pela alocação sequencial, apresentada na Seção 2.2. Em seguida, são examinadas as listas lineares estruturadas sob forma sequencial. São descritos os algoritmos básicos de busca e efetuado o cálculo da complexidade média. Nessa mesma seção é também apresentada a busca binária. As pilhas e filas são estudadas na Seção 2.4. A Seção 2.5 apresenta, como aplicação, a notação polonesa para expressões aritméticas. A alocação encadeada é objeto da Seção 2.6. O estudo das listas lineares em alocação encadeada é efetuado na Seção 2.7. São examinadas as listas simplesmente encadeadas: pilhas e filas em alocação encadeada, as listas circulares, bem como as listas duplamente encadeadas. Ainda nessa seção é estudado, como aplicação, o problema da ordenação topológica. A seção se encerra com o exame de listas com nós de tamanho variável.

2.2 Alocação Sequencial

A maneira mais simples de se manter uma lista linear na memória do computador é colocar seus nós em posições contíguas. Nesse caso, o endereço real do $(j + 1)$-ésimo nó da lista se encontra c unidades adiante daquele correspondente ao j-ésimo. A constante c

é o número de palavras de memória que cada nó ocupa. A correspondência entre o índice da tabela e o endereço real é feita automaticamente pela linguagem de programação quando da tradução do programa.

Como a implementação da alocação sequencial em linguagens de alto nível é geralmente realizada com a reserva prévia de memória para cada estrutura utilizada, a inserção e a remoção de nós não ocorrem de fato. Em vez disso utiliza-se algum tipo de simulação para essas operações (por exemplo, variáveis indicando os limites da memória realmente utilizada). Por essa razão, pode-se considerar tal alocação como uma *alocação estática*.

O armazenamento sequencial é particularmente atraente no caso de filas e pilhas porque, nessas estruturas, as operações básicas podem ser implementadas de forma bastante eficiente. Esse tratamento pode, contudo, se tornar oneroso em termos de memória quando se empregam diversas estruturas simultaneamente. Nesse caso, a utilização ou não do armazenamento sequencial dependeria de um estudo cuidadoso das opções existentes.

De início serão apresentadas as operações para listas genéricas.

2.3 Listas Lineares em Alocação Sequencial

Seja uma lista linear. Cada nó é formado por *campos*, que armazenam as características distintas dos elementos da lista. Além disso, cada nó da lista possui, geralmente, um identificador, denominado *chave*. Para evitar ambiguidades, supõe-se que todas as chaves são distintas. A chave, quando presente, se constitui em um dos campos do nó. Os nós podem se encontrar ordenados, ou não, segundo os valores de suas chaves. No primeiro caso a lista é denominada *ordenada*, e *não ordenada* no caso contrário.

Suponha uma lista linear, de nome L, que possui *n* elementos. Um exemplo da constituição dessa tabela é apresentado na Figura 2.1.

O Algoritmo 2.1 apresenta a busca de um nó na lista L, conhecendo-se sua chave. A variável *x* corresponde à chave do nó procurado. A função *busca*1 informa, ao final, o índice do nó que se deseja buscar; se este não for encontrado, o índice é nulo.

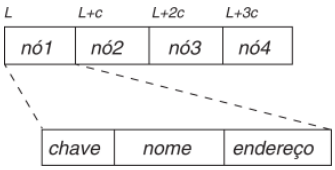

FIGURA 2.1 Exemplo de um nó.

Algoritmo 2.1 Busca de um elemento na lista L

```
função busca1(x)
    i := 1;      busca1 := 0
    enquanto i≤ n faça
        se L[i] . chave = x então
            busca1 := i          % chave encontrada
            i := n + 1
        senão i := i + 1         % pesquisa prossegue
```

Observe que, para cada elemento da tabela referenciado na busca, o algoritmo realiza dois testes: $i \leq n$ e $L[i]$. *chave* = *x*. Muitas vezes um pequeno artifício pode contribuir para a melhoria do processo. Por exemplo, o Algoritmo 2.2 se propõe a efetuar a mesma busca que o Algoritmo 2.1. A diferença entre os dois é a criação de um novo nó, que possui o valor procurado no campo *chave*, na posição *n* + 1. Dessa forma, o algoritmo sempre encontra um nó da tabela com as características desejadas, evitando o teste de fim de tabela.

Algoritmo 2.2 Busca de um elemento na lista L

```
função busca(x)
    L[n + 1] . chave := x;  i := 1
```

```
enquanto L[i] . chave ≠ x faça
    i := i + 1
se i ≠ n + 1 então
        busca := i       % elemento encontrado
senão busca := 0         % elemento não encontrado
```

A complexidade de pior caso dos Algoritmos 2.1 e 2.2 é $O(n)$. Entretanto, o segundo é de execução mais rápida, pois a cada iteração correspondem dois testes no Algoritmo 2.1 e apenas um no 2.2.

As complexidades médias dos Algoritmos 2.1 e 2.2 também são idênticas. Para determiná-las, seja q a probabilidade de sucesso no resultado da busca. Além disso, suponha que sejam idênticas as probabilidades de a chave procurada se encontrar em posições distintas da lista. A observação fundamental para calcular a complexidade média é que, para o algoritmo, entradas distintas que tenham a chave procurada na mesma posição podem ser consideradas como idênticas. Assim, o algoritmo só reconhece $n + 1$ entradas distintas, a saber: entradas em que a chave procurada se encontra na posição 1, posição 2, …, posição n e entradas em que a chave não se encontra na lista.

Pelo Capítulo 1, sabe-se que a complexidade média é dada por $\sum p(E_k)t(E_k)$. No caso, há somente $n + 1$ entradas a considerar. Seja E_i, $1 \leq i \leq n$, uma entrada em que a chave procurada ocupa a i-ésima posição da lista, e E_0 a entrada que corresponde à busca sem sucesso. Logo, as probabilidades das entradas são

$$p(E_k) = q/n,\ 1 \leq k \leq n$$
$$p(E_0) = 1 - q,$$

enquanto o número total de passos efetuados pelo algoritmo é

$$t(E_k) = k,\ 1 \leq k \leq n$$
$$t(E_0) = n.$$

Logo, a expressão da complexidade média é

$$\sum_{0 \leq k \leq n} p(E_k)\, t(E_k) = (1 - q)n + \sum_{1 \leq k \leq n} (q/n)k = (1/2)\,[(2 - q)n + q].$$

Como casos particulares, se $q = 1$, isto é, a chave se encontra sempre na lista, então a complexidade é $\approx n/2$. Se $q = 1/2$, esta cresce para $\approx 3n/4$. Se $q = 0$, isto é, todas as buscas são sem sucesso, a complexidade média atinge o valor n.

Quando a lista está ordenada, pode-se tirar proveito desse fato. Se o número procurado não pertence à lista não há necessidade de percorrê-la até o final. A exemplo do Algoritmo 2.2, a dupla comparação no bloco principal do algoritmo também pode ser evitada por meio da criação de um novo nó. O Algoritmo 2.3 mostra essa busca.

Algoritmo 2.3 Busca de um elemento na lista L, ordenada

```
função busca-ord(x)
    L[n + 1] . chave := x;   i := 1
    enquanto L[i] . chave < x faça
        i := i + 1
    se i = n + 1 ou L[i] . chave ≠ x então
            busca-ord := 0
    senão busca-ord := i
```

A complexidade de pior caso do algoritmo acima é, evidentemente, igual à dos algoritmos anteriores. Contudo, a maior eficiência do algoritmo se traduz na expressão da complexidade média. No seu cálculo, utilizam-se as mesmas premissas. Isto é, q é a probabilidade de sucesso do resultado da busca. Além disso, entradas em que a chave procurada se encontra em posições distintas da lista possuem a mesma probabilidade de ocorrência. Contudo, ao contrário do caso anterior, o algoritmo, agora, é sensível a um total de $2n + 1$ entradas distintas, uma vez que, no Algoritmo 2.3, também o insucesso pode ser reportado em situações distintas. Com isso, o número de passos efetuados t por uma busca sem sucesso torna-se variável. Por hipótese, os diferentes valores de t possuem a mesma probabilidade de ocorrência.

Para resolver o problema do caso médio, é necessário introduzir as definições seguintes. Sejam R_0, …, R_n conjuntos de elementos não pertencentes à lista, representando os "espaços" entre as chaves da lista em que a chave procurada poderia se encontrar. Isto é, R_0 representa todos os valores possíveis menores do que a primeira chave de L, R_n corresponde aos valores maiores do que a última chave de L, enquanto R_k, $1 \leq k < n$, é o conjunto dos valores maiores do que a k-ésima e menores do que a $(k + 1)$-ésima. No caso de uma busca sem sucesso, a chave procurada se encontra em um dos conjuntos R_k.

As $2n + 1$ entradas distintas podem ser descritas como:

E_k = entrada em que a chave procurada é $L[k]$. *chave*, $1 \leq k \leq n$;
E'_k = entrada em que a chave procurada pertence a R_k, $0 \leq k \leq n$.

As probabilidades das entradas são:

$$p(E_k) = q/n,\ 1 \leq k \leq n$$
$$p(E'_k) = (1 - q)/(n + 1),\ 0 \leq k \leq n,$$

enquanto os números de iterações correspondentes são:

$$t(E_k) = k,\ 1 \leq k \leq n$$
$$t(E'_k) = k + 1,\ 0 \leq k \leq n$$

Logo, a expressão da complexidade média é:

$$\sum_{k=1}^{n} p(E_k)\, t(E_k) + \sum_{k=0}^{n} p(E'_k)\, t(E'_k)$$
$$= \sum_{k=1}^{n} (q/n) \cdot k + \sum_{k=0}^{n} [(1 - q)/(n + 1)] \cdot (k + 1)$$
$$= (n - q + 2)/2.$$

Para efeito de comparação com os Algoritmos 2.1 e 2.2, observe que o valor da complexidade média correspondente ao Algoritmo 2.3 é aproximadamente $n/2$, para qualquer probabilidade q.

Ainda no caso de listas ordenadas, um algoritmo diverso e bem mais eficiente pode ser apresentado: a *busca binária*. A ideia básica do algoritmo é percorrer a tabela como se folheia, por exemplo, uma lista telefônica, abandonando-se as partes do catálogo onde o nome procurado, com certeza, não será encontrado. Em tabelas, o primeiro nó pesquisado é o que se encontra no meio; se a comparação não é positiva, metade da tabela pode ser abandonada na busca, uma vez que o valor procurado se encontra ou na metade inferior (se for menor), ou na metade superior (se for maior). Esse procedimento, aplicado recursivamente, esgota a tabela. O Algoritmo 2.4 apresenta a busca.

Algoritmo 2.4 Busca binária

```
função busca-bin(x)
    inf := 1;   sup := n;   busca- bin := 0
    enquanto inf ≤ sup faça
        meio := ⌊ (inf + sup)/2⌋          % índice a ser buscado
        se L[meio] . chave = x então
            busca- bin := meio            % elemento encontrado
            inf := sup + 1
        senão  se L[meio] . chave < x então
                    inf := meio + 1
        senão sup := meio – 1
```

A complexidade do algoritmo pode ser avaliada da seguinte forma. O pior caso ocorre quando o elemento procurado é o último a ser encontrado, ou mesmo não é encontrado, isto é, quando a busca prossegue até a tabela se resumir a um único elemento. Na primeira iteração, a dimensão da tabela é n, e algumas operações são realizadas para situar o valor procurado. Na segunda, a dimensão se reduz a $\lfloor n/2 \rfloor$, e assim sucessivamente. Ora, ao final, a dimensão da tabela é 1 (observe o teste $inf \leq sup$). Então, no pior caso:

1ª iteração: a dimensão da tabela é n,

2ª iteração: a dimensão da tabela é $\lfloor n/2 \rfloor$,
3ª iteração: a dimensão da tabela é $\lfloor (\lfloor n/2 \rfloor)/2 \rfloor$,
...
*m*ª iteração: a dimensão da tabela é 1.

Ou seja, o número de iterações é, no máximo, $1 + \lfloor \log_2 n \rfloor$. O tempo consumido pelas operações em cada iteração é constante. Logo, a complexidade da busca binária é $O(\log n)$.

Ambas as operações de inserção e remoção utilizam o procedimento de busca. No primeiro caso, o objetivo é evitar chaves repetidas e, no segundo, a necessidade de localizar o elemento a ser removido. A construção desses algoritmos implica tarefas complementares, uma vez que a ação de um é o inverso da ação do outro. A implementação dessas operações deve, naturalmente, respeitar tal fato. O Algoritmo 2.5 apresenta a inserção de um nó contido na variável *novo* de chave *x*. O Algoritmo 2.6 efetua a remoção de um nó sendo conhecido um de seus campos, no caso a chave *x*. Ambos os algoritmos consideram tabelas não ordenadas. A memória pressuposta disponível tem *M* posições (na realidade *M* + 1, porque é necessária uma posição extra para o procedimento de busca). Devem-se levar em conta as hipóteses de se tentar fazer inserções numa lista que já ocupa *M* posições (situação conhecida como *overflow*), bem como a tentativa de remoção de um elemento de uma lista vazia (*underflow*). A atitude a ser tomada em cada um desses casos depende do problema tratado. Por essa razão, os procedimentos *overflow* e *underflow* são apenas indicados.

Como pode ser observado, o procedimento de inserção propriamente dito é bem simples, porém depende da busca que tem complexidade de $O(n)$. O algoritmo de remoção, além da busca, em geral efetua movimentação de nós, o que o torna ainda mais lento, se bem que também de complexidade $O(n)$.

Algoritmo 2.5 Inserção de um nó na lista *L*

```
se n < M então
    se busca(x) = 0 então
        L[n + 1] := novo-valor
        n := n + 1
    senão "elemento já existe na tabela"
senão overflow
```

Uma alternativa ao algoritmo de remoção é efetuar o deslocamento do último elemento da lista para a posição vaga. Nesse caso, entretanto, a sequência dos elementos fica alterada.

Algoritmo 2.6 Remoção de um nó da lista *L*

```
se n ≠ 0 então
    indice := busca(x)
    se indice ≠ 0 então
        valor- recuperado := L[indice]
        para i := indice, n – 1 faça
            L[i] := L[i + 1]
        n := n – 1
    senão "elemento não se encontra na tabela"
senão underflow
```

No caso de tabelas ordenadas, o algoritmo de remoção não se modifica. O algoritmo de inserção, entretanto, precisa ser refeito, uma vez que, nesse caso, a posição do nó se torna relevante. Isso implica movimentar parte da tabela, para permitir a inserção na posição correta, de maneira análoga à efetuada na remoção em listas não ordenadas (Algoritmo 2.6). A complexidade de ambos os algoritmos (inserção e remoção) é, então, $O(n)$.

Observe que a utilização da busca binária diminui a complexidade da busca, mas não a da inserção ou da remoção. A complexidade dessas últimas operações é determinada pela movimentação dos nós.

2.4 Pilhas e Filas

Em geral, o armazenamento sequencial de listas é empregado quando as estruturas, ao longo do tempo, sofrem poucas remoções e inserções. Em casos particulares de listas, esse armazenamento é também empregado. Nesse caso, a situação favorável é aquela em que inserções e remoções não acarretam movimentação de nós, o que ocorre se os elementos a serem inseridos e removidos estão em posições especiais, como a primeira ou a última posição. Deques, pilhas e filas satisfazem tais condições.

Na alocação sequencial de listas genéricas, considera-se sempre a primeira posição da lista no endereço 1 da memória disponível. Uma alternativa a essa estratégia consiste na utilização de indicadores especiais, denominados *ponteiros*, para o acesso a posições selecionadas. No caso da pilha, apenas um ponteiro precisa ser considerado, o ponteiro *topo*, pois as inserções e remoções são executadas na mesma extremidade da lista. A Figura 2.2 mostra uma sequência de operações realizadas numa pilha e o resultado de tais operações no ponteiro *topo*. Em seguida, os Algoritmos 2.7 e 2.8 implementam a inserção e a remoção em uma pilha *P*, considerando-se a memória disponível de *M* posições. Os algoritmos levam em consideração as hipóteses de *overflow* e *underflow*. A pilha vazia tem *topo* nulo.

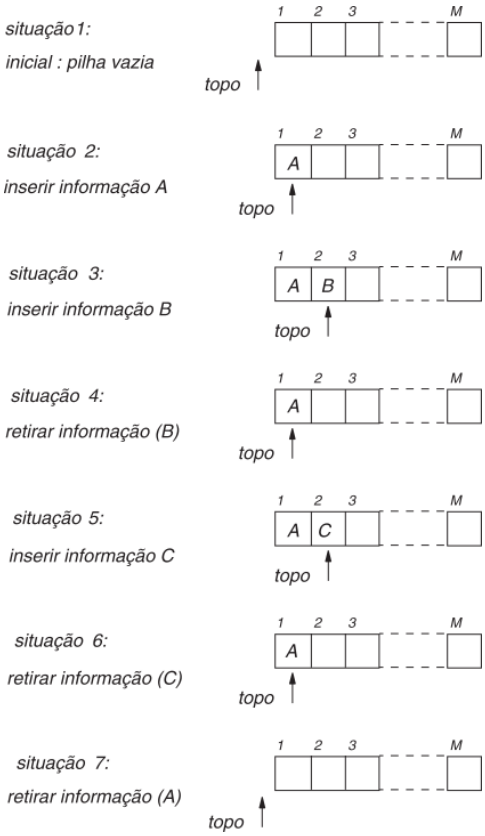

FIGURA 2.2 Operações em uma pilha.

Algoritmo 2.7 Inserção na pilha *P*

se *topo* ≠ *M* **então**
    topo := *topo* + 1
    P[*topo*] := *novo-valor*
senão *overflow*

Algoritmo 2.8 Remoção da pilha *P*

se *topo* ≠ 0 **então**
    valor- recuperado := *P*[*topo*]
    topo := *topo* – 1
senão *underflow*

A complexidade das operações apresentadas é constante, $O(1)$.

As filas exigem uma implementação um pouco mais elaborada. São necessários dois ponteiros: início de fila (f) e retaguarda (r). Para a adição de um elemento, move-se o ponteiro r; para a retirada, move-se o ponteiro f. A situação de fila vazia é representada por $f = r = 0$. Observe, na Figura 2.3, algumas operações realizadas em uma fila F.

Note que, após qualquer operação, deve-se sempre ter o ponteiro f indicando o início da fila, e r, a retaguarda. Isso implica, como já foi dito, movimentar o ponteiro r quando de uma inserção e o ponteiro f quando de uma remoção. Ora, pode-se observar claramente na Figura 2.3 que, à medida que os ponteiros são incrementados na memória disponível, a fila "se move", o que pode dar origem à falsa impressão de memória esgotada. Para eliminar esse problema, consideram-se os M nós alocados como se estivessem em círculo, onde $F[1]$ segue $F[M]$. No algoritmo de inserção, a variável *prov* armazena provisoriamente a posição de memória calculada de forma a respeitar a circularidade, só sendo movimentado o ponteiro r se a operação for possível. A inicialização dos ponteiros f e r é $f = r = 0$.

Algoritmo 2.9 Inserção na fila *F*

```
prov := r mod M + 1
se prov ≠ f então
        r := prov
        F[r] := novo-valor
        se f = 0 então
            f := 1
senão overflow
```

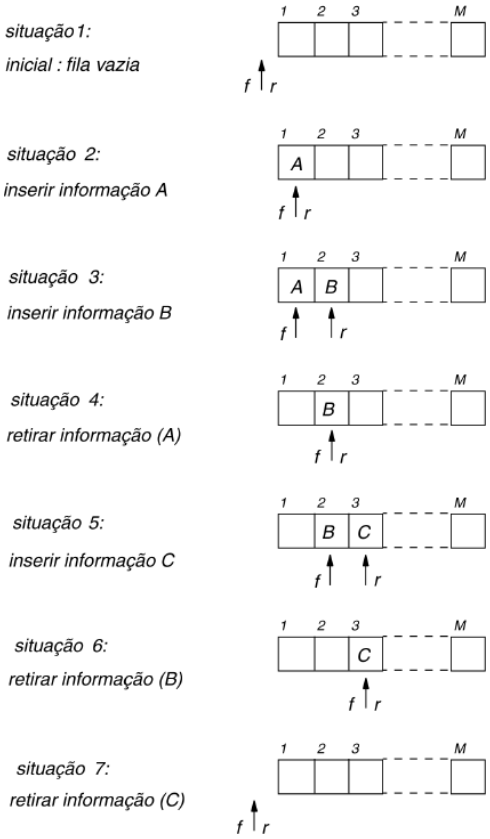

FIGURA 2.3 Operações em uma fila.

Algoritmo 2.10 Remoção da fila *F*

```
se f ≠ 0 então
        valor- recuperado := F[ f]
        se f = r então
```

```
            f := r := 0
        senão f := f mod M + 1
    senão underflow
```

Também nesse caso a complexidade das operações é constante.

2.5 Aplicação: Notação Polonesa

Uma representação para expressões aritméticas, que seja conveniente do ponto de vista computacional, é assunto de interesse, por exemplo, na área de compiladores. A notação tradicional é ambígua e, portanto, obriga ao preestabelecimento de regras de prioridade. Isso, naturalmente, torna a tarefa computacional menos simples. Outras representações são apresentadas a seguir, considerando-se somente operações binárias.

– A notação completamente parentizada: acrescenta-se sempre um par de parênteses a cada par de operandos e a seu operador.

  Exemplo:

  notação tradicional: *A* * *B* –*C*/*D*

  notação parentizada: ((*A* * *B*) – (*C*/*D*))

– A notação polonesa: os operadores aparecem imediatamente antes dos operandos. Esta notação explicita quais operadores, e em que ordem, devem ser calculados. Por esse motivo, dispensa o uso de parênteses, sem ambiguidades.

  Exemplo:

  notação tradicional: *A* * *B* – *C*/*D*

  notação polonesa: – * *AB*/*CD*

– A notação polonesa reversa: a notação polonesa com os operadores aparecendo após os operandos. Esta notação é tradicionalmente utilizada em máquinas de calcular.

  Exemplo:

  notação tradicional: *A* * *B* – *C*/*D*

  notação polonesa reversa: *AB* * *CD*/–

Tanto a expressão original quanto a expressão transformada devem ser armazenadas em listas. Um algoritmo para efetuar a conversão da notação parentizada na notação polonesa reversa é apresentado aqui. Algumas observações auxiliam em tal conversão.

– Os operandos aparecem na mesma ordem na notação tradicional e na notação polonesa reversa.

– Na notação polonesa reversa, os operadores aparecem na ordem em que devem ser calculados (da esquerda para a direita).

– Os operadores aparecem imediatamente depois de seus operandos.

Note que, pela primeira observação, a ordem dos operandos não se modifica, podendo estes ser ignorados. A principal preocupação do algoritmo deve ser a ordem e a posição dos operadores. Na notação parentizada, essa ordem é indicada pelos parênteses; os mais internos significam operações prioritárias.

Algoritmo 2.11 Conversão de notações

```
indexp := 1;   indpol := 0        % índices das expressões
topo := 0
enquanto indexp≤ fim faça
      se exp[indexp] é operando então
         indpol := indpol + 1        % passa para a nova lista
         pol[indpol] := exp[indexp]
      senão  se exp[indexp] é operador então
            topo := topo + 1         % coloca na pilha
```

```
            pilha[topo] := exp[indexp]
        senão se exp[indexp] = ")" então
            se topo ≠ 0 então        % forma a operação
                operador := pilha[topo]
                topo := topo – 1
                indpol := indpol + 1
                pol[indpol] := operador
            senão "expressão errada"
    indexp := indexp + 1
```

Uma análise sintática prévia na expressão é pressuposta. Para efetuar a conversão, devem-se remover os parênteses e estabelecer a ordem conveniente de operadores. Estes então devem ser armazenados até que um ")" seja encontrado, o que indica que a operação mais interna, e por conseguinte a primeira a ser executada, foi detectada. O último operador armazenado corresponde a essa operação. Portanto, a estrutura utilizada no armazenamento dos operadores deve ser uma pilha. No Algoritmo 2.11, a expressão a ser convertida se encontra no vetor *exp*, e o resultado da conversão no vetor *pol*. A variável *fim* indica a dimensão da expressão.

2.6 Alocação Encadeada

Já foi observado que o desempenho dos algoritmos que implementam operações realizadas em listas com alocação sequencial, mesmo sendo estes muito simples, pode ser bastante fraco. E mais, quando está prevista a utilização concomitante de mais de duas listas a gerência de memória se torna mais complexa. Nesses casos se justifica a utilização da alocação encadeada, também conhecida por *alocação dinâmica*, uma vez que posições de memória são alocadas (ou desalocadas) na medida em que são necessárias (ou dispensadas). Os nós de uma lista encontram-se então aleatoriamente dispostos na memória e são interligados por ponteiros, que indicam a posição do próximo elemento da tabela. É necessário o acréscimo de um campo a cada nó, justamente o que indica o endereço do próximo nó da lista. A Figura 2.4 apresenta uma lista linear em suas representações sequencial e encadeada.

Há vantagens e desvantagens associadas a cada tipo de alocação. Estas, entretanto, só podem ser precisamente medidas ao se conhecerem as operações envolvidas na aplicação desejada. De maneira geral pode-se afirmar que a alocação encadeada, a despeito de um gasto de memória maior em virtude da necessidade de um novo campo no nó (o campo do ponteiro), é mais conveniente quando o problema inclui o tratamento de mais de uma lista. Isso se aplica tanto à gerência do armazenamento quanto às operações propriamente ditas envolvidas, como juntar listas, separar listas em sublistas etc. Por outro lado, o acesso ao *k*-ésimo elemento da lista é imediato na alocação sequencial, enquanto na alocação encadeada obriga ao percurso na lista até o elemento desejado.

Como já foi visto, na alocação encadeada os nós de uma lista se encontram em posições não obrigatoriamente contíguas de memória. Se nessa lista são feitas inserções e remoções, há necessidade de encontrar novas posições de memória para armazenamento e liberar outras que possam ser reutilizadas posteriormente. Um algoritmo para gerenciar as posições de memória disponíveis é então imprescindível. Para tal é criada uma lista especial, chamada *Lista de Espaço Disponível* (*LED*), que contém posições de memória ainda não utilizadas ou dispensadas após sua utilização. Note que a organização dessas posições disponíveis independe completamente da estrutura empregada na solução do problema em estudo. Entretanto, deve-se observar que a *LED* e as estruturas estão compartilhando a memória disponível.

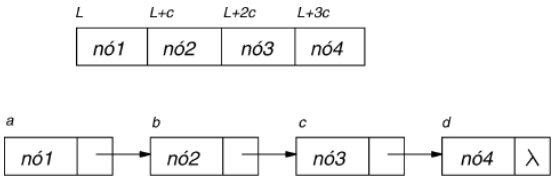

FIGURA 2.4 Alocação sequencial e alocação encadeada.

A implementação da *LED* pode ser executada de duas formas. A primeira, com o dimensionamento de um único vetor de nós M (ou diversos vetores, um para cada campo do nó) simulando a memória total disponível. Nesse caso, o endereço do nó corresponde ao índice de uma posição do vetor. Essa abordagem permite ao usuário o controle total de posições ocupadas e livres, e pode ser implementada na maioria das linguagens de programação existentes. No princípio, como a memória se acha totalmente disponível, todos os seus nós são encadeados na *LED*. A variável ponteiro *vago* se refere ao topo da estrutura. Para se inicializar uma *LED* são então necessários os passos seguintes:

Passo 1: Os campos ponteiros dos nós são encadeados sequencialmente.

Passo 2: O ponteiro *vago* é inicializado com o endereço do primeiro nó da lista que foi encadeada no primeiro passo.

Passo 3: O campo ponteiro do último nó recebe o valor λ, indicando fim de lista.

Para a efetivação das operações de inserção ou remoção de nós em listas encadeadas há necessidade, portanto, de manipular também a *LED*. Quando se precisa de um novo nó para executar o algoritmo de inserção deve-se buscá-lo na *LED*. Analogamente, devolve-se à *LED* um nó dispensado pelo algoritmo de remoção. Procedimentos básicos de busca e devolução de nós à *LED* podem então ser preparados. Esses algoritmos utilizam sempre uma variável *pt* que indica o índice do nó disponível. No procedimento de busca esse nó fica assim reservado para uso futuro, e no procedimento de devolução é o nó a ser reincorporado. Os Algoritmos 2.12 e 2.13 executam essas tarefas. O vetor M, de nós, corresponde à *LED;* o campo que indica o próximo nó disponível tem o nome de *prox*.

- **Algoritmo 2.12 Busca de um elemento na *LED***

```
procedimento ocupar(pt)
    se vago ≠ λ então
        pt := vago
        vago := M[vago] .prox
    senão overflow
```

- **Algoritmo 2.13 Devolução de um nó à *LED***

```
procedimento desocupar(pt)
    M[pt] . prox := vago
    vago := pt
```

É interessante observar que a hipótese de *overflow* é considerada no Algoritmo 2.12. Isso indica que a possibilidade de memória esgotada é estudada de maneira global para todas as estruturas existentes no problema considerado.

A segunda opção, embora mais dependente da linguagem de programação, é a mais utilizada. As linguagens geralmente possuem um módulo de gerência de memória disponível ao usuário, bastando apenas que este se refira às rotinas internas de ocupação e devolução de nós da lista de espaço disponível. Em Pascal, por exemplo, as rotinas *new*(*pt*) e *dispose*(*pt*) executam essa tarefa. Nesse caso, uma notação diferente é utilizada nos algoritmos. Obviamente o nó na memória é o mesmo considerado até agora; a indicação do endereço desse nó, feita por um ponteiro, é que será representada pelo símbolo ↑. Cada ponteiro utilizado é associado a um único tipo de nó. Assim, por exemplo, *pt* ↑. *info* representa o campo *info* do nó apontado por *pt*. Um conjunto de nós encadeados forma uma tabela. A associação do ponteiro *pt* ao tipo de nó é definida previamente na declaração das variáveis.

2.7 Listas Lineares em Alocação Encadeada

2.7.1 Listas Simplesmente Encadeadas

Qualquer estrutura, inclusive listas, que seja armazenada em alocação encadeada requer o uso de um ponteiro que indique o endereço de seu primeiro nó. O percurso de uma lista é feito então a partir desse ponteiro. A ideia consiste em seguir consecutivamente pelos endereços existentes no campo que indica o próximo nó, da mesma forma que na alocação sequencial se

acrescentava uma unidade ao índice do percurso. O Algoritmo 2.14 apresenta o percurso para impressão do campo *info* de uma lista, sendo *ptlista* o ponteiro para o primeiro nó.

Algoritmo 2.14 Impressão da lista apontada por ptlista

```
pont := ptlista
enquanto pont ≠ λ faça
        imprimir(pont ↑.info)
        pont := pont ↑.prox
```

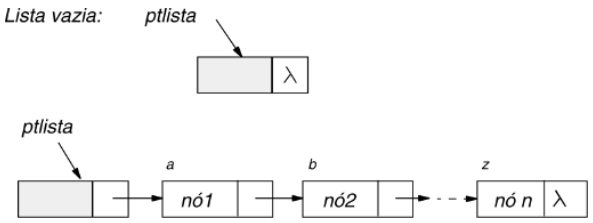

FIGURA 2.5 Lista encadeada com nó-cabeça.

Como já foi mencionado na Seção 2.3, o algoritmo de busca, por ser utilizado em inserções, remoções e outras operações, deve ser muito eficiente. Na alocação encadeada essa necessidade persiste. E mais, surgem novos problemas: por exemplo, a existência de um ponteiro indicando o primeiro nó da lista obriga os algoritmos de inserção e remoção a apresentarem testes especiais para verificar se o nó desejado é o primeiro da lista. Isto pode ser resolvido por uma pequena variação na estrutura de armazenamento: a criação de um nó especial, chamado *nó-cabeça*, nunca removido, que passa a ser o nó indicado pelo ponteiro de início de lista. Esse nó especial não deve conter informações relacionadas à tabela propriamente dita. Algumas vezes, entretanto, pode ser aproveitado para conter dados pertinentes ao algoritmo implementado. A Figura 2.5 mostra a representação gráfica da lista encadeada com nó-cabeça, em sua situação inicial (vazia), e depois de algumas inserções.

O Algoritmo 2.15 implementa a busca em uma tabela ordenada, em alocação encadeada, de maneira simples. Outra abordagem, um pouco mais eficiente, será vista na Seção 2.7.3 (lista circular encadeada). No caso aqui considerado, o nó-cabeça da tabela é apontado por *ptlista*. O parâmetro *x* fornece a chave procurada. O parâmetro *pont* retorna apontando para o elemento procurado, e *ant* para o elemento anterior ao procurado. Caso este não seja encontrado, *pont* aponta para λ e *ant* indica ainda o elemento anterior ao último pesquisado. Deve-se notar que o parâmetro *ant*, apesar de aparentemente inútil, é importante para os algoritmos de inserção e remoção, que serão vistos a seguir. Como o algoritmo estabelece um percurso pela tabela, sua complexidade é $O(n)$, sendo n o número de nós da lista.

Algoritmo 2.15 Busca em uma lista ordenada

```
procedimento busca-enc(x, ant, pont)
        ant := ptlista;  pont := λ
        ptr := ptlista ↑.prox % ptr : ponteiro de percurso
        enquanto ptr ≠ λ faça
                se ptr ↑. chave < x então
                    ant := ptr                  % atualiza ant e ptr
                    ptr := ptr ↑.prox
                senão  se ptr ↑. chave = x então
                            pont := ptr % chave encontrada
                    ptr := λ
```

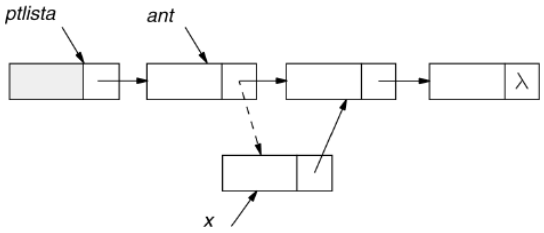

FIGURA 2.6 Inserção de um nó.

Após a realização da busca, as operações de inserção e remoção em uma lista encadeada são triviais. Há três fases a serem cumpridas: a comunicação com a *LED*, o acesso ao campo de informação do nó e o acerto de estrutura. As operações de inserção e remoção realizam essas fases em ordem inversa. A Figura 2.6 e o Algoritmo 2.16 mostram a inserção do nó contido na variável *novo*, após o nó apontado por *ant*. No algoritmo, as três fases se encontram assinaladas pelos comentários.

Algoritmo 2.16 Inserção de um nó após o nó apontado por ant busca-enc(x, ant, pont)

```
se pont = λ então
        ocupar(pt)                  % solicitar nó
        pt ↑. info := novo-valor    % inicializar nó
        pt ↑. chave := x; pt ↑. prox := ant ↑.prox
        ant ↑. prox := pt           % acertar lista
 senão "elemento já está na tabela"
```

A remoção do nó apontado por *pont* é apresentada na Figura 2.7 e no Algoritmo 2.17. Observe as fases mencionadas nos comentários do algoritmo. A complexidade dessas operações é, em virtude da busca, $O(n)$.

Algoritmo 2.17 Remoção do nó apontado por pont na lista

```
busca-enc(x, ant, pont)
 se pont ≠ λ então
        ant ↑. prox := pont ↑.prox          % acertar lista
        valor- recuperado := pont ↑.info    % utilizar nó
        desocupar(pont)                     % devolver nó
 senão "nó não se encontra na tabela"
```

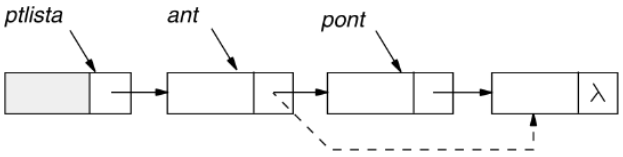

FIGURA 2.7 Remoção de um nó.

Os algoritmos de busca, remoção e inserção em uma lista não ordenada constituem ligeiras variações dos anteriormente apresentados (Exercício 2.11).

2.7.2 Pilhas e Filas

Como casos particulares, algumas modificações são necessárias para implementar operações eficientes em pilhas e filas. No caso de pilhas, as operações são muito simples. Considerando-se listas simplesmente encadeadas (sem nó-cabeça), o topo da pilha é o primeiro nó da lista, apontado por uma variável ponteiro *topo*. Se a pilha estiver vazia então *topo* = λ. Filas exigem duas variáveis do tipo ponteiro: *inicio*, que aponta para o primeiro nó da lista, e *fim*, que aponta para o último. Na fila vazia, ambos apontam para λ. Os algoritmos que se seguem implementam essas operações.

Algoritmo 2.18 Inserção na pilha

```
ocupar(pt)                    % solicitar nó
pt ↑. info := novo-valor      % inicializar nó
pt ↑. prox := topo
topo := pt                    % acertar pilha
```

Algoritmo 2.19 Remoção da pilha

```
se topo ≠ λ então
        pt := topo                          % acertar pilha
        topo := topo ↑.prox
        valor- recuperado := pt ↑.info      % utilizar nó
        desocupar(pt)                       % devolver nó
senão underflow
```

Algoritmo 2.20 Inserção na fila

```
ocupar(pt)                    % solicitar nó
pt ↑. info := novo-valor      % inicializar nó
pt ↑. prox := λ
se fim ≠ λ então              % acertar fila
        fim ↑. prox := pt
senão inicio := pt
fim := pt
```

Algoritmo 2.21 Remoção da fila

```
se inicio ≠ λ então
        pt := inicio
        inicio := inicio ↑. prox            % acertar fila
        se inicio = λ então fim := λ
        valor- recuperado := pt ↑.info      % utilizar nó
        desocupar(pt)                       % devolver nó
senão underflow
```

As complexidades dos algoritmos de manipulação de filas e pilhas são constantes, ou seja, $O(1)$, uma vez que buscas não são empregadas.

Uma aplicação interessante de filas é a *ordenação por distribuição*, descrita a seguir. Seja uma lista L composta de n chaves, cada qual representada por um número inteiro numa base $b > 1$. O problema consiste em ordenar essa lista. O algoritmo utiliza b filas, denotadas por F_i, $0 \leq i \leq b - 1$. Seja d o comprimento máximo da representação das chaves na base b. O algoritmo efetua d iterações, em cada uma das quais a tabela é percorrida. A primeira iteração destaca, em cada nó, o dígito menos significativo da representação b-ária de cada chave. Se este for igual a k, a chave correspondente será inserida na fila F_k. Ao terminar o percurso da tabela, esta se encontra distribuída pelas filas, que devem então ser concatenadas em sequência, isto é, F_0, depois F_1, F_2 etc. Para essa tabela, já disposta numa ordem diferente da original, o processo deve ser repetido levando-se em consideração o segundo dígito da representação, e assim sucessivamente até que tenham sido feitas tantas distribuições quantos são os dígitos na chave de ordenação. Veja um exemplo dessa ordenação na Figura 2.8, onde $b = 10$ e $d = 2$.

O Algoritmo 2.22 descreve o processo. A notação $F_k \Leftarrow L[\, j \,]$ significa a inserção na fila F_k do elemento localizado em $L[\, j \,]$. Analogamente, $L[\, j \,] \Leftarrow F_k$ representa a remoção de um elemento da fila F_k e seu armazenamento em $L[\, j \,]$. A lista L contém os elementos a serem ordenados. Supõe-se que as filas F_i tenham sido inicializadas como vazias.

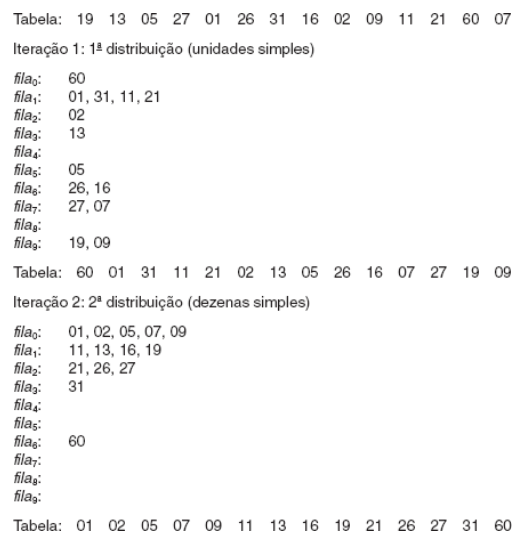

FIGURA 2.8 Ordenação por distribuição.

Algoritmo 2.22 Ordenação por distribuição

para $i = 1, \ldots, d$ **faça**
  para $j = 1, \ldots, n$ **faça**
    k := i-ésimo dígito menos significativo da representação de $L[\,j\,]$. *chave* na base b
    $F_k \Leftarrow L[\,j\,]$
  $j := 1$
  para $k = 0, \ldots, b - 1$ **faça**
    enquanto $F_k \neq \emptyset$ **faça**
      $L[\,j\,] \Leftarrow F_k$
      $j := j + 1$

Ao final do processo, a lista L encontra-se ordenada. Cada operação de inclusão ou remoção de um elemento é realizada em tempo constante. A complexidade do algoritmo é, portanto, igual a $O(n\,d)$. Como a implementação utiliza b filas, é óbvia a necessidade de que estas empreguem alocação encadeada. O algoritmo exige que seja calculada, previamente, a representação de cada chave na base b. Supondo-se que o computador utilize a base 2 para a codificação interna, a definição $b = 2$ dispensaria tal cálculo. Nesse caso, uma forma de acelerar o processo seria dividir a representação binária em grupos de m bits consecutivos, o que corresponderia a utilizar a base 2^m.

Entre os equipamentos de processamento de dados, precursores dos computadores, um dos mais importantes foi a *classificadora de cartões*. Esta efetuava a ordenação física dos cartões segundo chaves representadas por perfurações localizadas em 12 alturas diferentes no cartão, o que permitia ordenação alfabética. O princípio utilizado era o da ordenação por distribuição. Cada fila correspondia a um escaninho existente no equipamento. A distribuição era mecânica, mas a concatenação se realizava manualmente.

2.7.3 Listas Circulares

A busca em uma tabela ordenada, apresentada na Seção 2.7.1, pode ser considerada pouco eficiente quando comparada às outras buscas anteriormente mencionadas. Uma pequena modificação na estrutura física da lista pode ser de grande auxílio: obrigar o último nó da lista a apontar para o nó-cabeça, criando assim uma *lista circular encadeada*, como é visto na Figura 2.9. Dessa forma, o teste de fim de lista nunca é satisfeito. A preocupação passa a ser um critério de parada que possa ser incorporado ao teste da busca propriamente dita. A solução é colocar a chave procurada no nó-cabeça, de forma que uma resposta positiva seja sempre encontrada.

O Algoritmo 2.23 apresenta a nova busca, no caso de listas ordenadas. Ao fim da busca, o ponteiro *pont* aponta para o último nó pesquisado. Naturalmente, modificações correspondentes devem ser introduzidas nos algoritmos de inserção e remoção.

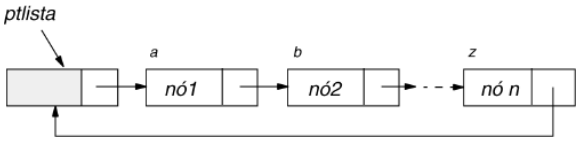

FIGURA 2.9 Lista circular encadeada.

Algoritmo 2.23 Busca numa lista circular encadeada ordenada

```
procedimento busca-cir(x, ant, pont)
    ant := ptlista
    ptlista ↑. chave := x
    pont := ptlista ↑.prox
    enquanto pont ↑. chave < x faça
        ant := pont
        pont := pont ↑.prox
    se pont ≠ ptlista e pont ↑. chave = x então
        "chave localizada"
    senão "chave não localizada"
```

No caso de listas não ordenadas, o mesmo princípio pode ser empregado, com as devidas adaptações.

2.7.4 Listas Duplamente Encadeadas

Nos algoritmos vistos até agora para listas lineares utilizando alocação encadeada, o ponteiro *ant* se mostrou sempre útil. Sua função é "rastrear" o ponteiro que percorre a lista, permitindo sempre o retorno ao nó anterior. Algumas vezes, entretanto, isso não é suficiente, pois pode-se desejar o percurso da lista nos dois sentidos indiferentemente. Nesses casos, o gasto de memória imposto por um novo campo de ponteiro pode ser justificado pela economia em não reprocessar praticamente a lista inteira. A Figura 2.10 apresenta uma *lista circular duplamente encadeada*, com nó-cabeça, que incorpora esse novo campo de ponteiro. Os campos de ponteiros tomam os nomes de *ant* (apontando para o nó anterior) e *post* (apontando para o nó seguinte). Note-se, entretanto, que listas não circulares e listas sem nó-cabeça podem também ser duplamente encadeadas.

Os algoritmos de busca, inserção e remoção em tabelas ordenadas são muito simples, sendo apresentados a seguir. Na busca, a função retorna indicando o nó procurado ou, se este não for encontrado, o nó que seria seu consecutivo. Observe nas Figuras 2.11 e 2.12 as modificações acarretadas por uma inserção e uma remoção.

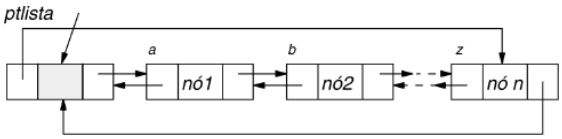

FIGURA 2.10 Lista duplamente encadeada.

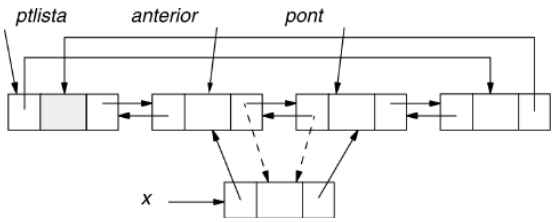

FIGURA 2.11 Inserção em lista duplamente encadeada.

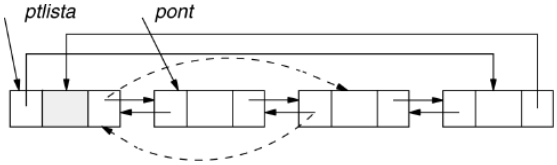

FIGURA 2.12 Remoção em lista duplamente encadeada.

Algoritmo 2.24 Busca em uma lista duplamente encadeada ordenada

```
função busca-dup(x)
    ultimo := ptlista ↑. ant
    se x≤ ultimo ↑. chave então
        pont := ptlista ↑.post
        enquanto pont ↑. chave < x faça
            pont := pont ↑.post
        busca- dup := pont
    senão busca- dup := ptlista
```

Algoritmo 2.25 Inserção de um nó em uma lista duplamente encadeada

```
pont := busca-dup(x)
 se pont = ptlista ou pont ↑. chave ≠ x então
    anterior := pont ↑.ant
    ocupar(pt)                          % solicitar nó
    pt ↑. info := novo-valor            % inicializar nó
    pt ↑. chave := x
    pt ↑. ant := anterior
    pt ↑. post := pont
    anterior ↑. post := pt              % acertar lista
    pont ↑. ant := pt
 senão "elemento já se encontra na lista"
```

Algoritmo 2.26 Remoção de um nó em uma lista duplamente encadeada

```
pont := busca-dup(x)
 se pont ≠ ptlista e pont ↑. chave = x então
    anterior := pont ↑.ant
    posterior := pont ↑.post
    anterior ↑. post := posterior       % acertar lista
    posterior ↑. ant := anterior
    valor- recuperado := pont ↑.info    % utilizar nó
    desocupar(pont)                     % devolver nó
 senão "elemento não se encontra na lista"
```

2.7.5 Aplicação: Ordenação Topológica

Um problema que pode ser caracterizado como uma aplicação de listas lineares é a ordenação topológica. Sua importância se deve ao fato de ter um uso potencial todas as vezes em que o problema abordado envolve uma ordem parcial.

Uma *ordem parcial* de um conjunto S é uma relação entre os objetos de S, representada pelo símbolo "$\preccurlyeq$", satisfazendo as seguintes propriedades para quaisquer objetos x, y e z, não necessariamente distintos em S:

(i) se $x \preccurlyeq y$ e $y \preccurlyeq z$, então $x \preccurlyeq z$ (transitiva);

(ii) se $x \preccurlyeq y$ e $y \preccurlyeq x$, então $x = y$ (antissimétrica);

(iii) $x \preccurlyeq x$ (reflexiva).

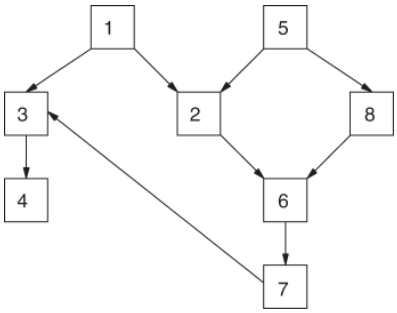

FIGURA 2.13 Remoção em lista duplamente encadeada.

A notação $x \preccurlyeq y$ pode ser lida "x precede ou iguala y". Se $x \preccurlyeq y$ e $x \neq y$, escreve-se $x \prec y$ e diz-se "x precede y". Tem-se então:

(i′) se $x \prec y$ e $y \prec z$, então $x \prec z$ (transitiva);

(ii′) se $x \prec y$, então $y \not\prec x$ (assimétrica);

(iii′) $x \not\prec x$ (irreflexiva).

A notação $y \not\prec x$ significa "y não precede x". Assume-se aqui que S é um conjunto finito, uma vez que se deseja trabalhar no computador.

Muitos exemplos interessantes podem ser mencionados como utilização da ordem parcial. Entre outros, a execução de um conjunto de tarefas necessárias, por exemplo, à montagem de um automóvel. Uma ordem parcial dessas tarefas pode ser representada como na Figura 2.13. Cada caixa na figura representa uma tarefa a ser executada; cada tarefa é numerada arbitrariamente. Se existe a indicação de um caminho da caixa x para a caixa y, isso significa que a tarefa x deve ser executada antes da tarefa y.

A ordenação topológica trata de imergir a ordem parcial em uma ordem linear, isto é, rearrumar os objetos numa sequência a_1, a_2, …, a_n tal que sempre que $a_j \prec a_k$, tem-se $j < k$. A Figura 2.14 mostra a ordem parcial do exemplo da Figura 2.13 após a ordenação topológica.

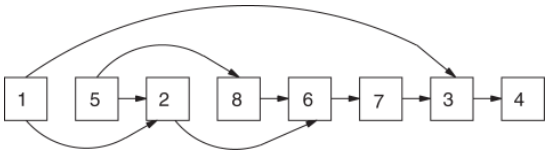

FIGURA 2.14 Ordenação topológica.

Uma forma simples de obter uma ordenação topológica é vista a seguir. Inicialmente, considera-se um objeto que não é precedido por nenhum outro na ordem parcial. Esse objeto é o primeiro na saída, isto é, na ordem final. Agora, remova o objeto do conjunto S. O conjunto resultante obedece novamente a uma ordem parcial. O processo pode então ser repetido até que todo o conjunto esteja ordenado. Esse algoritmo só falharia se, em algum momento, a ordem parcial de um conjunto fosse tal que todos os elementos tivessem um predecessor. Ora, isso é impossível, porque contraria as propriedades (i) e (ii).

O desempenho desse algoritmo depende de sua implementação. Os objetos a serem ordenados são numerados de 1 a n, em qualquer ordem, e alocados sequencialmente na memória. A entrada do algoritmo são os pares (j, k), significando que o objeto j precede o objeto k. O número de objetos n e o número de pares m também são fornecidos. Devem constar da entrada somente os pares necessários à caracterização da ordem parcial; pares supérfluos, entretanto, não constituem erro. Por exemplo, no problema apresentado na Figura 2.13, o par (8, 7) é desnecessário, pois pode ser deduzido dos pares (8, 6) e (6, 7).

O algoritmo lida com n listas encadeadas, uma para cada objeto. Na lista i estão indicados todos os objetos k que aparecem em pares (i, k), isto é, sucessores de i. Para cada lista é criado um nó-cabeça, chamado *CB*. No nó-cabeça da lista i, além do ponteiro para os sucessores de i, está armazenado o número de pares (k, i) existentes, isto é, o número de vezes em que i aparece como sucessor de outro objeto. Essa informação é denominada *contador*. A Figura 2.15 mostra a atuação dessa fase do algoritmo no exemplo da Figura 2.13. Uma vez organizadas as listas, os nós-cabeça são percorridos em busca de um objeto que não seja sucessor de nenhum outro, isto é, cujo campo *contador* seja nulo. Todos aqueles que obedecem a tal condição são encadeados pelo próprio campo *contador*, que passa assim a ter nova função. O resultado dessa fase pode ser visto, em pontilhado, na mesma

figura. Em seguida, o objeto é "retirado" da tabela original, isto é, todos os contadores de seus sucessores são decrementados e imediatamente testados para que, ao chegarem a zero, sejam incluídos na sequência de saída. O algoritmo termina quando todos os objetos são retirados.

A complexidade do algoritmo pode ser deduzida da descrição apresentada a seguir. Existem *n* listas encadeadas, correspondentes a um total de *m* pares de relações "precede". De início, a lista de nós-cabeça é percorrida em busca de objetos sem predecessores, que são encadeados como candidatos à saída. Essa parte do algoritmo tem complexidade $O(n)$. Na saída de cada candidato, os contadores de seus sucessores são decrementados, o que equivale ao percurso da lista encadeada correspondente ao candidato. Ao fim do algoritmo, todas as listas terão sido percorridas, o que leva $O(m)$. A complexidade é então $O(n + m)$.

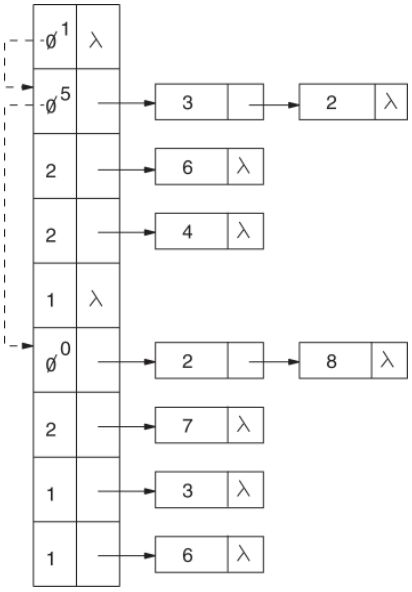

FIGURA 2.15 Armazenamento para a ordenação topológica.

Algoritmo 2.27 Ordenação topológica

```
procedimento inicializar
        para i = 0, …, n faça                          % inicializa nó-cabeça
                CB [i] . contador := 0
                CB [i] . prox := λ
        para i = 1, …, m faça
                ocupar(pt)                             % seja (j, k) o i-ésimo par
                pt ↑. info := k; pt ↑. prox := CB [ j ] .prox
                CB [ j ] . prox := pt
                CB [k] . contador := CB [k] . contador + 1
inicializar
fim := 0;   CB [0] . contador := 0
para i = 1, …, n faça                                 % busca de objetos sem
        se CB [i] . contador = 0 então predecessores
                CB [ fim] . contador := i
                fim := i
objeto := CB [0] .contador
enquanto objeto ≠ 0 faça
        saida(objeto)
        pt := CB [objeto] .prox
        enquanto pt ≠ λ faça
```

```
indice := pt ↑. info
CB [indice] . contador := CB [indice] . contador –1
se CB [indice] . contador = 0 então
        CB [ fim] . contador := indice
        fim := indice
    pt := pt ↑.prox
objeto := CB [objeto] .contador
```

2.8 Alocação de Espaço de Tamanho Variável

Um modelo de memória comumente utilizado corresponde a um vetor M, onde cada elemento é associado a uma unidade de memória denominada *palavra*. Um conjunto de *b* palavras de endereços consecutivos é chamado *bloco de tamanho b*. O *endereço* de um bloco é o índice em M de sua primeira palavra. No modelo, todos os programas e dados competem pelo uso exclusivo de porções de M.

Por exemplo, suponha que um programa deseje criar uma lista linear, contendo *b* nós, cada qual ocupando uma unidade de memória. Nesse caso, deve ser solicitada a reserva de um bloco de tamanho *b*, em algum endereço disponível. Quando da extinção da lista, esse bloco deve ser liberado para possível uso com outra finalidade. Supondo a existência simultânea de diversas estruturas, a memória apresenta, alternadamente, blocos reservados e disponíveis. Essa configuração varia, dinamicamente, com as diferentes solicitações de reserva e liberação de blocos. A variação do tamanho dos blocos pode ser muito grande, pois depende das características individuais das estruturas utilizadas. Esse processo é denominado *sistema de alocação de memória*.

Um caso particular já foi examinado na Seção 2.7.1. Quando todos os blocos possuem tamanho idêntico, a ideia utilizada consistiu em encadear a memória disponível em uma lista denominada *LED*. Os procedimentos *ocupar* e *desocupar*, dos Algoritmos 2.12 e 2.13, respectivamente, efetuam as operações de reserva e liberação de blocos de tamanho fixo. O caso da alocação de memória de tamanho variado é bem mais complexo.

Basicamente, para elaborar os algoritmos de reserva e liberação é necessário, de antemão, decidir as três questões seguintes.

(1) Que estrutura utilizar para representar a memória disponível?

(2) Qual é o critério de seleção de blocos, para atender às solicitações de reserva de memória?

(3) Como definir a operação de liberação de blocos?

Podem-se considerar diversas abordagens para responder a essas questões. A solução descrita a seguir é uma das mais simples.

A memória disponível constituirá uma lista encadeada, onde cada nó está associado a um bloco. Essa lista será mantida na própria memória disponível. Em caso de alocação em memória secundária, contudo, é recomendável o armazenamento da lista na memória principal. As duas primeiras palavras de cada nó serão utilizadas pelo sistema de alocação. Se *pt* é o endereço de um bloco disponível, então $M[pt]$ contém o tamanho do bloco, e $M[pt + 1]$ é o ponteiro para o próximo nó da lista. Isso implica que o tamanho mínimo de qualquer bloco, disponível ou reservado, é igual a 2. Além disso, as duas primeiras palavras do vetor M têm uso reservado. $M[1]$ será inicializada com zero e não será modificada. Enquanto isso, $M[2]$ será o ponteiro para o endereço do primeiro bloco da lista. Ao contrário do par $M[pt]$ e $M[pt + 1]$, as palavras $M[1]$ e $M[2]$ não podem ser utilizadas para atender a qualquer reserva de memória. Assim sendo, supondo que o vetor M possua *m* palavras, a inicialização do processo consiste em efetuar as atribuições $M[1] := 0$, $M[2] := 3$, $M[3] := m - 2$ e $M[4] := \lambda$.

Quanto ao critério de seleção de blocos para as solicitações de reserva, uma das características desejadas é a simplicidade do algoritmo correspondente. O método seguinte se denomina *primeira escolha* e, certamente, satisfaz essa condição. O critério consiste em selecionar o primeiro bloco da lista de memória disponível, cujo tamanho satisfaça as condições da reserva. Seja *b* o tamanho do bloco solicitado. O algoritmo de reserva deve percorrer a lista e selecionar o primeiro bloco de tamanho $b' \geq \max\{2, b\}$. Seja $r = b' - b$. Se $r < 2$, então o bloco inteiro deve ser reservado, o que corresponde a remover o bloco de tamanho b' da lista. Caso contrário, ele é dividido em dois outros, de tamanhos *b* e *r*, respectivamente. O primeiro é o correspondente à reserva, e o segundo é o bloco associado ao nó selecionado, cujo tamanho é reduzido a *r*. Observe que o bloco reservado deve ser o correspondente aos endereços mais altos, para não afetar as duas primeiras palavras, de uso do sistema de alocação.

Uma consequência indesejada desse método é a possível formação de blocos de tamanho muito reduzido, fenômeno conhecido por *fragmentação*. Para procurar evitá-la, uma alternativa é definir um limite inferior $c \geq 2$, tal que o sistema não permita a formação de blocos menores do que c. Suponha que seja solicitada a reserva de um bloco de tamanho b e que o bloco selecionado da lista possua tamanho $b' \geq b$, onde $b' - b < c$. Nesse caso, a ideia é incorporar ao bloco reservado a porção restante $b' - b$.

O procedimento que se segue descreve o processo. A chamada externa é *reserva*(*b*, *pont*), sendo *b* o tamanho do bloco solicitado. O tamanho efetivamente reservado é pelo menos igual a max{2, *b*}, porém menor do que *b* + *c*. Esse valor é atribuído a *b* no final do processo. O parâmetro *pont* indica, nessa ocasião, o endereço do bloco reservado. Se não existir bloco disponível que satisfaça as condições da solicitação, então *pont* = λ.

Algoritmo 2.28 Reserva de um bloco de tamanho *b*

```
procedimento reserva(b, pont)
    pt := M[2];   pont := λ;   ant := 1
    se b < 2 então b := 2
    enquanto pt ≠ λ faça
        se M[pt] ≥ b então
            r := M[pt] – b                % bloco encontrado
            se r < c então
                pont := pt                % fragmento incorporado
                M[ant + 1] := M[pt + 1]
                b := b + r
            senão M[pt] := r
                pont := pt + r
            pt := λ
        senão ant := pt
            pt := M[pt + 1]
```

A operação de liberação de blocos será realizada de acordo com a seguinte estratégia. Sejam *b* e *pont* o tamanho e endereço do bloco liberado, respectivamente. Deve ser verificado se ele é contíguo a algum bloco disponível. Podem existir até dois blocos nessas condições. Em caso negativo, o bloco de endereço *pont* será simplesmente inserido na lista. Caso seja encontrado algum bloco contíguo ao liberado, os dois deverão ser fundidos num único. Aquele de endereço maior será incorporado ao menor. A inserção ou incorporação deve ser realizada de tal modo que a lista seja mantida em ordem crescente de endereços de seus blocos. Observe que a estratégia descrita garante a inexistência de blocos disponíveis e contíguos ao longo de todo o processo. A formulação encontra-se descrita no Algoritmo 2.29. Supõe-se que os blocos tenham sido incluídos na lista através do Algoritmo 2.28, o que implica $b \geq c \geq 2$.

Algoritmo 2.29 Liberação de um bloco

```
procedimento liberar(b, pont)
    pt := M[2];   ant := 1
    enquanto pt ≠ λ e pt < pont faça          % buscar posição do bloco
        ant := pt
        pt := M[pt + 1]
    se pont + b = pt então
        b := b + M[pt]                        % bloco liberado contíguo ao
        M[pont + 1] := M[pt + 1] seguinte
    senão M[pont + 1] := pt
    se ant + M[ant] = pont então
        M[ant] := M[ant] + b                  % bloco liberado contíguo ao
```

M[*ant* + 1] := *M*[*pont* + 1] anterior
senão *M*[*ant* + 1] := *pont*
M[*pont*] := *b*

Seja *n* o número de nós da lista que contém a memória disponível. No pior caso, os algoritmos de reserva e liberação efetuam $O(n)$ passos. Em decorrência, um dos efeitos da fragmentação é o aumento da complexidade dos algoritmos. Utilizando-se uma estrutura mais elaborada para a memória disponível e mantendo-se o critério de reserva e liberação de blocos, é possível formular algoritmos que diminuem a complexidade dessas operações para $O(\log n)$ (Exercício 6.21).

2.9 Exercícios

2.1 Apresentar os algoritmos de inserção e remoção de uma lista ordenada em alocação sequencial.

2.2 Determinar a expressão da complexidade média de uma busca não ordenada, supondo que a probabilidade da busca de qualquer chave, exceto a última, é igual à metade da probabilidade da chave seguinte na lista. Supor também que a probabilidade de a chave procurada se encontrar na lista é igual a *q*.

2.3 Repetir o exercício anterior para o caso da busca ordenada.

2.4 Seja o seguinte algoritmo de busca em uma lista ordenada de tamanho *n* em alocação sequencial:

função *busca-ord*1(*x*)
  se $x \leq L[n]$. *chave* **então**
    i := 1
    enquanto *L*[*i*] . *chave* < *x* **faça**
      i := *i* + 1
    se *L*[*i*] . *chave* ≠ *x* **então**
      *busca-ord*1 := 0 % elemento não encontrado
    senão *busca-ord*1 := *i* % elemento encontrado
  senão *busca-ord*1 := 0 % fora da tabela

Comparar o algoritmo apresentado com o Algoritmo 2.3. Em que situação o desempenho dos dois é equivalente? Qual é a restrição que o algoritmo acima apresenta em relação ao Algoritmo 2.3?

2.5 Determinar a expressão da complexidade média de uma busca não ordenada de *n* chaves, *n* par, em que as probabilidades de busca das chaves de ordem ímpar são iguais entre si, sendo esse valor igual ao dobro da probabilidade de qualquer chave par. Supor, ainda, que a probabilidade de a chave se encontrar na lista é igual a *q*.

2.6 Apresentar algoritmos para um deque em alocação sequencial. São dados fornecidos:
– na inserção: o nó a ser inserido e a extremidade desejada (*E*1 ou *E*2),
– na remoção: a extremidade da remoção.

2.7 Criar algoritmos de inserção e remoção para duas pilhas armazenadas em alocação sequencial que compartilham a memória de dimensão *M*.

2.8 Apresentar os algoritmos de inserção e remoção numa lista circular encadeada.

2.9 Apresentar o algoritmo de alteração do campo *info* de uma lista circular encadeada com nó-cabeça.

2.10 Apresentar o algoritmo de alteração do campo *chave* de uma lista circular encadeada com nó-cabeça.

2.11 Descrever algoritmos de inserção e remoção em uma lista não ordenada, em alocação encadeada.

2.12 Comparar algoritmos de busca, inserção e remoção em uma lista ordenada nas alocações sequencial e encadeada.

2.13 Sejam duas listas, não ordenadas, simplesmente encadeadas com nó-cabeça. Apresentar um algoritmo que, enquanto possível, intercale as duas listas.

2.14 Sejam duas listas, ordenadas, simplesmente encadeadas com nó-cabeça. Apresentar um algoritmo que intercale as duas listas de forma que a lista resultante esteja também ordenada.

2.15 Imprimir, a partir de uma expressão em notação polonesa, a sequência de operações a serem executadas na ordem correta. Introduzir operandos especiais para armazenar resultados parciais.

2.16 Seja *L* uma lista simplesmente encadeada, composta dos números $\ell_1, \ell_2, \ldots, \ell_n$, respectivamente, segundo a ordem de armazenamento. Escrever um algoritmo que, percorrendo *L* uma única vez, constrói uma outra lista *L'*, formada dos elementos seguintes:

(i) $\ell_2, \ell_3, \ldots, \ell_n, \ell_1$;

(ii) $\ell_n, \ell_{n-1}, \ldots, \ell_1$;

(iii) $\ell_1 + \ell_n, \ell_2 + \ell_{n-1}, \ldots, \ell_{n/2} + \ell_{n/2+1}$, onde n é par.

2.17 Uma palavra é um *palíndromo* se a sequência de letras que a forma é a mesma, quer seja lida da esquerda para a direita ou da direita para a esquerda (exemplo: raiar). Escrever um algoritmo eficiente para reconhecer se uma dada palavra é um palíndromo. Escolher a estrutura de dados conveniente para representar a palavra.

2.18 Seja um polinômio da forma $P(x) = a_0 x^n + a_1 x^{n-1} + \ldots + a_n$; Representar $P(x)$ através de uma lista encadeada conveniente e escrever algoritmos eficientes para efetuar as seguintes operações, onde $Q(x)$ é um outro polinômio.

(i) Calcular $P(x_0)$, onde x_0 é um dado valor para x;

(ii) $P(x) + Q(x)$;

(iii) $P(x) \cdot Q(x)$.

•2.19 Generalizar o algoritmo de ordenação topológica, de modo a gerar todas as ordenações topológicas distintas de uma ordem parcial.

2.20 Seja A uma *matriz esparsa* $n \times m$, isto é, boa parte de seus elementos são nulos ou irrelevantes. Descrever uma estrutura de dados que represente A e cujo espaço total seja $O(k)$ em vez de $O(n\,m)$, onde k é o número total de elementos não irrelevantes de A.

2.21 Descrever um algoritmo para localizar um elemento a_{ij} da matriz A, armazenada segundo a estrutura obtida na solução do exercício anterior. Determinar a sua complexidade.

2.22 Descrever um algoritmo para computar a matriz A^2, utilizando a representação dos dois exercícios anteriores.

2.23 Seja 1, 2, …, n uma sequência de elementos que serão inseridos e posteriormente retirados de uma pilha P uma vez cada. A ordem de inclusão dos elementos na pilha é 1, 2, …, n, enquanto a de remoção depende das operações realizadas. Por exemplo, com $n = 3$, a sequência de operações

incluir em P
incluir em P
retirar de P
incluir em P
retirar de P
retirar de P

produzirá a permutação 2, 3, 1 a partir da entrada 1, 2, 3. Representando por I, R, respectivamente, as operações de inserção e remoção da pilha, a permutação 2, 3, 1 pode ser denotada por *IIRIRR*. De um modo geral, uma permutação é chamada *admissível* quando ela puder ser obtida mediante uma sucessão de inclusões e remoções em uma pilha a partir da permutação 1, 2, …, n. Assim, por exemplo, a permutação 2, 3, 1 é admissível. Pede-se:

(i) Determinar a permutação correspondente a *IIIRRIRR*, $n = 4$.

(ii) Dar um exemplo de permutação não admissível.

2.24 Escrever a relação de permutações admissíveis de 1, 2, 3, 4.

°2.25 Provar que uma permutação $p_1, \ldots, p_n$ é admissível se e somente se não existirem índices i, j, k satisfazendo $i < j < k$ e $p_j < p_k < p_i$.

2.26 Repetir os Exercícios 2.23 e 2.24 com uma fila, em vez de uma pilha.

2.27 Determinar as condições necessárias e suficientes para que uma permutação seja admissível, supondo que uma fila seja utilizada no lugar da pilha.

•2.28 Determinar as condições necessárias e suficientes para que uma permutação seja admissível, supondo que um deque seja utilizado no lugar da pilha.

°2.29 No sistema de alocação de memória, o critério de *melhor escolha*, para a reserva de um bloco, seleciona aquele que esteja disponível e cujo tamanho seja o mais próximo possível do solicitado. Provar ou dar contraexemplo da afirmativa a seguir.

O desempenho do método de melhor escolha é sempre não inferior ao da primeira escolha no seguinte sentido. Se for possível selecionar um bloco de tamanho $\geq b$ num sistema que aplica a primeira escolha, o mesmo acontece para o método da melhor escolha.

Notas Bibliográficas

Devido ao seu largo uso, a concepção de listas lineares se deu com o aparecimento do primeiro computador. O uso de pilhas e filas, por exemplo, transcende à computação. De fato, pilhas e filas eram empregadas em diversas áreas, anteriormente aos computadores. Contudo, foi Knuth ([Kn68]) quem, pela primeira vez, descreveu de forma unificada os algoritmos para as estruturas de dados mais básicas. As permutações admissíveis (Exercícios 2.23 a 2.27) podem ser encontradas também em [Kn68]. Além disso, essa referência contém a técnica de implementação utilizada no

algoritmo de ordenações topológicas, da Seção 2.7. Um algoritmo de geração de todas as ordenações topológicas (Exercício 2.19), com complexidade polinomial por ordenação, foi inicialmente apresentado por Knuth e Szwarcfiter [Kn74]. Um algoritmo ótimo para esse problema foi descrito por Pruesse e Ruskey [Pr91]. Os termos *primeira escolha* e *melhor escolha* (Exercício 2.29) são provenientes do inglês *first-fit* e *best-fit*.