Análise Para uma pesquisa com sucesso, conforme mostrado na página 6 da Seção 1.3, temos:

$$\begin{array}{ll} \text{melhor caso} & : C(n) = 1 \\ \text{pior caso} & : C(n) = n \\ \text{caso médio} & : C(n) = (n+1)/2 \end{array}$$

Para uma pesquisa sem sucesso temos:

$$C'(n) = n + 1.$$

Observe que o anel interno da função Pesquisa, no Programa 5.2, é extremamente simples: o índice i é decrementado e a chave de pesquisa é comparada com a chave que está no registro. Por essa razão, esta técnica usando **sentinela** é conhecida por **pesquisa sequencial rápida**. Esse algoritmo é a melhor solução para o problema de pesquisa em tabelas com 25 registros ou menos.

5.2 Pesquisa Binária

A pesquisa em uma tabela pode ser muito mais eficiente se os registros forem mantidos em ordem. Para saber se uma chave está presente na tabela, compare a chave com o registro que está na posição do meio da tabela. Se a chave é menor, então o registro procurado está na primeira metade da tabela; se a chave é maior, então o registro procurado está na segunda metade da tabela. Repita o processo até que a chave seja encontrada ou fique apenas um registro cuja chave é diferente da procurada, indicando uma pesquisa sem sucesso. A Figura 5.1 mostra os subconjuntos pesquisados para recuperar o índice da chave G.

	1	2	3	4	5	6	7	8
Chaves iniciais:	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*
	A	*B*	*C*	**D**	*E*	*F*	*G*	*H*
					E	**F**	*G*	*H*
							G	*H*

Figura 5.1 *Exemplo de pesquisa binária para a chave G.*

O Programa 5.3 mostra a implementação do algoritmo para um conjunto de registros implementado como uma tabela.

Análise A cada iteração do algoritmo, o tamanho da tabela é dividido ao meio. Logo, o número de vezes que o tamanho da tabela é dividido ao meio é cerca de

Programa 5.3 *Pesquisa binária*

```
function Binaria (x: TipoChave; var T: Tipotabela): TipoIndice;
var i, Esq, Dir: TipoIndice;
begin
  if T.n = 0
  then Binaria := 0
  else begin
       Esq := 1; Dir := T.n;
       repeat
         i := (Esq + Dir) div 2;
         if x > T.Item[i].Chave
         then Esq := i+1
         else Dir := i-1;
       until (x = T.Item[i].Chave) or (Esq > Dir);
       if x=T.Item[i].Chave
       then Binaria := i
       else Binaria := 0;
       end;
end; { Binaria }
```

$\log n$. Entretanto, o custo para manter a tabela ordenada é alto: cada inserção na posição p da tabela implica o deslocamento dos registros a partir da posição p para as posições seguintes. Consequentemente, a pesquisa binária não deve ser usada em aplicações muito dinâmicas.

5.3 Árvores de Pesquisa

A árvore de pesquisa é uma estrutura de dados muito eficiente para armazenar informação. Ela é particularmente adequada quando existe necessidade de considerar todos ou alguma combinação de requisitos, tais como: (i) acessos direto e sequencial eficientes; (ii) facilidade de inserção e retirada de registros; (iii) boa taxa de utilização de memória; (iv) utilização de memória primária e secundária.

Se alguém considerar separadamente qualquer um dos requisitos no parágrafo anterior, é possível encontrar uma estrutura de dados que seja superior à árvore de pesquisa. Por exemplo, tabelas *hashing* possuem tempos médios de pesquisa melhores e tabelas que usam posições contíguas de memória possuem melhores taxas de utilização de memória. Entretanto, uma tabela que usa *hashing* precisa ser ordenada se existir necessidade de processar os registros sequencialmente em ordem lexicográfica, e a inserção/retirada de registros em tabelas que usam posições contíguas de memória tem custo alto. As árvores de pesquisa representam um compromisso entre esses requisitos conflitantes.