ESTRUTURAS DE DADOS

Paulo Eustáquio Duarte Pinto

Instituto de Matemática e Estatística
Departamento de Informática e Ciência da Computação
Universidade Estadual do Rio de Janeiro

Março de 2008

Conteúdo:

1 Introdução a Estruturas de Dados

1.1 Definições Iniciais

Algoritmos são procedimentos escritos de forma que possam ser executados por computadores. Desta forma, soluções para problemas que possam ser resolvidos por computadores são traduzidas numa linguagem algorítmica. Uma pergunta intrigante diz respeito a quais problemas podem ser resolvidos através de computadores. Bem, nesta disciplina estudaremos uma série de problemas que têm solução eficiente em máquinas.

Estruturas de Dados são formas de organizar dados relativos a objetos que tenham importância na solução de problemas computacionais, de maneira que os algoritmos envolvidos possam ser eficientes.

Eficiência de algoritmos diz respeito ao uso parcimonioso de tempo e memória na execução dos mesmos nas máquinas. A eficiência de algoritmos é discutida através da **Análise** de algoritmos, como descrita no ítem 1.3.

Para caracterizar melhor as idéias acima, tomaremos inicialmente o problema de ordenar um conjunto de valores, para o qual discutiremos dois algoritmos sobre uma mesma estrutura de dados.

1.2 Introdução à Ordenação de Vetores

O problema de ordenação de dados pode ser descrito como:

"Dada uma lista contendo n chaves, distintas ou não, encontrar uma permutação dessas chaves tal que elas fiquem em ordem crescente (ou decrescente) na lista".

Há mais de 10 idéias diferentes para ordenação de dados na memória e a maioria delas usa vetores como estrutura de dados. Ao longo do curso várias dessas idéias serão apresentadas. Para cada idéia será mostrado um algoritmo e a análise resumida do mesmo, baseada nos seguintes pontos:

a) Complexidade - Analisa-se a complexidade como descrito no tópico 1.1.3. No caso da ordenação de vetores, o tempo de execução está sempre relacionado às comparações feitas entre as chaves (algumas vezes depende também das trocas de posição), pois todos algoritmos são "loops" de comparação, onde a instrução mais executada é uma comparação de chaves.

b) Estabilidade: é a análise da manutenção ou não da ordem relativa de chaves iguais, durante o processo.

c) Situações especiais: é a discussão de situações especiais dos dados que seja relevante para a complexidade.

d) Memória utilizada: análise da memória adicional utilizada no método.

Os algoritmos de ordenação que serão estudados na presente disciplina são: **Seleção , Inserção , ShellSort, Distribuição, MergeSort, Heapsort**

Esses algoritmos trabalham, em geral, com a seguinte estrutura de dados:

```
Constantes  n = ....; { Número de chaves}
Variáveis   h, i, j, k, m, t, min: inteiro; Vet: vetor[0:n] de inteiro;
```

e com os seguintes dados como exemplo:

1	2	3	4	5	6	7	8	9	10	11	12
E	**X**	**E**	**M**	**P**	**L**	**O**	**F**	**A**	**C**	**I**	**L**

1.2.1 Ordenação de Vetores por Seleção

a) Idéia: Criar, passo a passo, uma lista ordenada, a partir de um conjunto de dados, retirando do conjunto, a cada passo, o menor elemento presente, até esvaziá-lo. Para implementar essa idéia num vetor, ele é dividido em dois subvetores, o primeiro inicialmente nulo, tal que se retire dados do segundo e transfira para o primeiro. A transferência de dados é feita "in place", através de um "loop" de (n -1) passos.

b) Algoritmo Selecao;

```
Inicio:
    Para i de 1 até (n -1):
        min ← i;
        Para j de (i+1) até n:
            Se (Vet[j] < Vet[min]) Então  min ← j  Fs;
        Fp;
        t ← Vet[i]; Vet[i] ← Vet[j]; Vet[j] ← t;

    Fp;
Fim;
```

c) Exemplo , mostrando os elementos envolvidos em comparações e trocas (estas em vermelho):

Passo	1	2	3	4	5	6	7	8	9	10	11	12	
	E	X	E	M	P	L	O	F	A	C	I	L	
1	A	X	E	M	P	L	O	F	E	C	I	L	
2		C	E	M	P	L	O	F	E	X	I	L	
3			E	M	P	L	O	F	E	X	I	L	
4				E	P	L	O	F	M	X	I	L	
5					F	L	O	P	M	X	I	L	
6						I	O	P	M	X	L	L	
7							L	P	M	X	O	L	
8								L	M	X	O	P	
9									M	X	O	P	
10										O	X	P	
11											P	X	
	A	C	E	E	F	I	L	L	M	O	P	X	Situação Final

d) Análise do Algoritmo:

Para sabermos como varia o tempo de execução do algoritmo de acordo com o número e o conteúdo dos dados do vetor, vamos contar o número de instruções executadas e calcular o tempo de execução, considerando que:

t_p = tempo das instruções de controle do "loop" para;

t_a = tempo da instrução de atribuição;

t_s = tempo do teste da instrução se;

De início vemos que a atribuição interna ao comando "Se" pode ser executada ou não. As demais instruções serão sempre executadas. Então temos duas situações, que chamaremos o pior e o melhor caso. O pior caso acontece quando, para cada interação, as instruções internas ao "Se" são executadas e o melhor, quando não são nunca executadas. Desta forma, temos:

Tep(n) = pior tempo possível para ordenar uma entrada com n elementos =

$(n-1)(t_p + t_a + 3t_a) + (\Sigma_{1 \leq j \leq n-1} (t_p + t_s + t_a))$.

Note que na expressão acima separamos as instruções em duas parcelas. Na primeira estão as instruções sempre executadas e, na segunda, as instruções do "loop" mais interno. Calculando, obtemos:

$$Tep(n) = n^2(t_p + t_s + t_a)/2 + n(t_p - t_s + 7t_a)/2 - (t_p + 4t_a)$$

Como, para um dado computador os tempos de execução dos tipos de instrução são fixos, podemos escrever a expressão acima como um polinômio de 2° grau:

$$Tep(n) = a_1n^2 + b_1n + c_1.$$

Se calcularmos, de forma análoga, o tempo de execução no melhor caso, quando a instrução interna ao "Se" nunca é executada, obteremos:

$Tem(n) = n^2(t_p + t_s)/2 + n(t_p - t_s + 8t_a)/2 - (t_p + 4t_a)$, que pode ser escrito como:

$$Tem(n) = a_2n^2 + b_2n + c_2.$$

Vemos, então que, em qualquer caso, o tempo de execução é um polinômio de 2° grau em n. Devemos ressaltar que há uma pequena imprecisão no cálculo acima, pois ele está desconsiderando o último teste de controle de cada "loop". Neste caso, deveríamos acrescentar n.tp a cada um dos tempos calculados. Há , também uma simplificação, supondo-se que as instruções envolvendo vetor e variáveis simples levam o mesmo tempo, o que não é verdade. Mas nada disso

muda o tipo de polinômio obtido. Notar também que, se contássemos apenas o número de comparações de chaves durante o processo, obteríamos também polinômios do 2° grau.

d.2) Estabilidade: Algoritmo não estável

d.3) Situações Especiais: O algoritmo é utilizável para ordenação de um volume razoável de registros (alguns milhares), cujas chaves sejam pequenas, pois o número de movimentações de registros é baixo (= n).

d.4) Memória utilizada: nenhuma memória adicional necessária.

e) Observações:

e.1) Número de comparações feitas: 66
Número de trocas: 11

e.2) Notar que, algumas vezes, o elemento é trocado com ele mesmo. Entretanto, a tentativa de evitar isso pode piorar o algoritmo.

e.3) Este algoritmo foi apresentado mais para efeito didático embora, pela sua simplicidade de compreensão, possa ser utilizado para ordenação de pequenos vetores.

1.2.2 Ordenação de Vetores por Inserção

a) Idéia: Criar, passo a passo, uma lista ordenada, a partir de um conjunto de dados, retirando do conjunto, a cada passo, o primeiro elemento encontrado. Para implementar essa idéia num vetor, ele é dividido em dois subvetores. O primeiro inicialmente contém uma chave, sendo a transferência de dados feita "in place", com um "loop" de (n -1) passos onde, a cada passo, o elemento inserido pode ter que deslocar para a direita elementos maiores.

b) Algoritmo Insercao;

```
Inicio:
    Para i de 2 até n:
        t ← Vet[i] ;  Vet[0] ← t;   j ← i;
        Enquanto (Vet[j -1] > t):
            Vet[j] ← Vet[j -1];   j ← j -1;
        Fe;
        Vet[j] ← t;
    Fp;
Fim;
```

c) Exemplo, mostrando os elementos envolvidos em comparações e trocas (estas em vermelho):

Passo	1	2	3	4	5	6	7	8	9	10	11	12	
	E	**X**	**E**	**M**	**P**	**L**	**O**	**F**	**A**	**C**	**I**	**L**	
1	**E**	***X***											
2	**E**	***E***	***X***										
3		**E**	***M***	***X***									
4			**M**	***P***	***X***								
5		**E**	***L***	***M***	***P***	***X***							
6				**M**	***O***	***P***	***X***						
7		**E**	***F***	***L***	***M***	***O***	***P***	***X***					
8	***A***	***E***	***E***	***F***	***L***	***M***	***O***	***P***	***X***				
9	**A**	***C***	***E***	***E***	***F***	***L***	***M***	***O***	***P***	***X***			
10					**F**	***I***	***L***	***M***	***O***	***P***	***X***		
11							**L**	***L***	***M***	***O***	***P***	***X***	
	A	**C**	**E**	**E**	**F**	**I**	**L**	**L**	**M**	**O**	**P**	**X**	Situação Final

d) Análise do Algoritmo:

d.1) Complexidade:

Vamos fazer uma análise análoga à do algoritmo anterior. Teremos:

Tep(n) = pior tempo possível para ordenar uma entrada com n elementos =

$(n-1)(t_p + 3t_a + t_a) + (\Sigma_{1 \le j \le n-1} (t_{su} + t_s + 2t_{su} + 2t_a))$.

Note que na expressão acima separamos as instruções em duas parcelas. Na primeira estão as instruções sempre executadas e, na segunda, as

instruções do "loop" mais interno. No início deste "loop" há sempre um uma subtração (tsu) e um teste (ts) e, em seu interior, 2 atribuições e duas subtrações. Calculando, obtemos:

$Tep(n) = n^2(t_s + 3t_{su} + 2t_a)/2 + n(2t_p + 6t_a - t_s - 3t_{su})/2 - (t_p + 4t_a)$

Podemos escrever a expressão acima como um polinômio de 2° grau:

$Tep(n) = a_1n^2 + b_1n + c_1.$

Se calcularmos, de forma análoga, o tempo de execução no melhor caso, quando as instruções internas ao segundo "loop" nunca são executadas, obteremos:

$Tem(n) = (n-1)(t_p + 4t_a + t_s + t_{su})$, que pode ser escrito como:

$Tem(n) = b_2n + c_2.$

Vemos, então que os dois casos têm polinômios de grau diferente; no pior caso o tempo de execução é descrito por um polinômio de 2° grau e, no melhor caso, por um polinômio de 1° grau! Portanto, dependendo da entrada, o comportamento do tempo de execução é variável. O melhor caso ocorre quando o vetor já está ordenado e, o pior, quando o vetor está inversamente ordenado. Uma pergunta natural é: e no caso médio? Bem, no caso médio, o "loop" é executado um número de vezes igual à metade da variável "i" e este tempo também é expresso por um polinômio de 2° grau. Notar mais uma vez que se contarmos as comparações no processo de ordenação obteremos polinômios como os mencionados.

d.2) Estabilidade: Algoritmo estável

d.3) Situações Especiais: O algoritmo é adequado para situação de vetor quase ordenado (situação em que se fazem pequenas adições a um vetor já previamente ordenado).

d.4) Memória utilizada: nenhuma memória adicional.

e) Observações:

e.1) Número de comparações: 49 (inclui comparação com sentinela)
Número de trocas: 49

e.2) Notar que o vetor usa o índice 0 para guardar um "**sentinela**", no caso a própria chave que está sendo inserida. Esse procedimento simplifica e torna mais rápido o algoritmo. Essa é uma técnica bastante utilizada na construção de bons algoritmos.

1.3 Introdução à análise de algoritmos: a notação O.

Duas características muito importantes dos algoritmos são o seu tempo de execução e a memória requerida. Quando se faz um algoritmo para resolver determinado problema, não basta que o algoritmo esteja correto. É importante que ele possa ser executado em um tempo razoável e dentro das restrições de memória existentes. Além disso, ele deve permanecer viável, à medida que o tempo passa, quando a quantidade de dados envolvida normalmente cresce. O estudo do comportamento dos algoritmos em termos do tempo de execução e memória, em função do crescimento dos dados envolvidos, denomina-se **Complexidade de Algoritmos**. Os parâmetros estudados normalmente são os seguintes:

a) **Complexidade de pior caso** - caracterização do tempo de execução máximo, para determinado tamanho da entrada, bem como das características da entrada que levam a esse tempo máximo. Este é o principal parâmetro para se avaliar um algoritmo.
b) **Complexidade de caso médio** - caracterização do tempo de execução médio do algoritmo, para determinado tamanho da entrada, considerando a média de todas as possibilidades. Em muitas situações este parâmetro é útil.
c) **Complexidade de melhor caso** - caracterização do tempo de execução mínimo, para determinado tamanho da entrada, bem como das características da entrada que levam a esse tempo mínimo.
d) **Memória** requerida para se executar o algoritmo para determinado tamanho de entrada.

A determinação da complexidade teria que ser feita contando-se todas as instruções executadas e o tempo de execução de cada delas, considerando-se determinada entrada. Normalmente isso não é viável. O que se faz é determinar um limite superior para esse tempo, o mais próximo possível da

realidade. Para tanto, fixa-se o estudo na instrução mais executada do algoritmo e determina-se uma função t(n), que dá a variação do tempo de execução em função de n, o tamanho da entrada.

O limite superior descrito anteriormente é definido pela conceituação O(t(n)), definida da seguinte forma:

Sejam f, h duas funções reais positivas de variável inteira n. Diz-se que f é O(h), escrevendo-se f = O(h), quando existir uma constante c > 0 e um valor inteiro n_0, tal que

$$n > n_0 \Rightarrow f(n) \leq c.h(n_0) .$$

Exemplos:

a) $f = n^3 - 1 \Rightarrow f$ é $O(n^3)$
porque, para $n \geq 1$, temos: $n^3 - 1 \leq 1.n^3$.
Neste caso, $n_0 = 1$ e $c = 1$.

b) $f = 5 + 10 \log n + 3 \log^2 n \Rightarrow f$ é $O(\log^2 n)$
porque, para $n \geq 8$, temos: $5 + 10 \log n + 3 \log^2 n \leq 14.\log^2 n$.
Neste caso, $n_0 = 8$ e $c = 14$.

c) Se f é $O(n^3) \Rightarrow f$ é $O(n^4)$. (Porque?)

1.4 Complementos

Será apresentado o método de ordenação ShellSort, que é uma importante variação do método de Inserção.

1.4.1 Ordenação de Vetores pelo ShellSort

a) Idéia: Modificar o algoritmo de Inserção, fazendo vários passos de inserção, onde cada um deles é parecido com o algoritmo anterior, só que utilizando vetores intercalados. O último passo é exatamente a inserção, mas, nesse caso, o vetor já estará com poucas inversões, o que se constitui numa vantagem significativa.

b) Algoritmo Shellsort.

Inicio:

```
h ← 1; Enquanto (h ≤ n):  h ← 3*h + 1;  Fe;
Enquanto (h > 1):   h ← ⌊ h/3 ⌋;
        Para i de (h+1) até n:  t ← Vet[i] ;   j ← i;
                Enquanto (j > h) e (Vet[j - h] > t):
                        Vet[j] ← Vet[j - h];   j ← j - h;
                Fe;
                Vet[j] ← t;
        Fp;
Fe;
Fim;
```

c) Exemplo, mostrando comparações e trocas.

Passo	1	2	3	4	5	6	7	8	9	10	11	12	
	E	X	E	M	P	L	O	F	A	C	I	L	
1	E				*P*								h = 4
2		*L*				*X*							
3			E				*O*						
4				*F*				*M*					
5	*A*				*E*				*P*				
6		*C*				*L*				*X*			
7			E				*I*				*O*		
8				F				*L*				*M*	
	A	C	E	F	E	L	I	L	P	X	O	M	Sit. Intermediária
9	A	*C*											h = 1
10		C	*E*										
11			E	*F*									
12			E	*E*	*F*								
13					F	*L*							
14					F	*I*	*L*						
15							L	*L*					
16								L	*P*				
17									P	*X*			
18								L	*O*	*P*	*X*		
19								L	*M*	*O*	*P*	*X*	
	A	C	E	E	F	I	L	L	M	O	P	X	Situação Final

d) Análise do Algoritmo:

d.1) Complexidade:

Melhor caso = Vetor Inicialmente Ordenado; NC ≈ n
Pior caso = Desconhecido (não é o vetor inv. ordenado!)
Caso Médio = Desconhecido. Supõe-se que NC = $n^{1.25}$

d.2) Estabilidade: Algoritmo não estável

d.3) Situações Especiais: Algoritmo de uso amplo, especialmente no caso de vetores quase ordenados.

d.4) Memória necessária: nenhuma memória adicional

e) Observações:

e.1) Número de comparações: 30
Número de trocas: 34

e.2) Notar que, neste caso, não há como utilizar sentinelas, já que isso exigiria um número variável de sentinelas à esquerda do vetor.

1.5 Exercícios

1.1) Implementar os algoritmos de Seleção e Inserção.
1.2) Mostrar os passos da ordenação pelos algoritmos de Seleção e Inserção, para o CRSTRING (= string de 12 letras formado pelas 12 letras iniciais do nome). Contar o número de comparações executadas.
1.3) Fazer um estudo do tempo de execução dos algoritmos Inserção e Seleção, para tamanhos e situações variadas da entrada e comparar os resultados teóricos.
1.4) O algoritmo de Seleção faz algumas troca desnecessária na situação em que o menor elemento ainda não ordenado já está na sua posição final. Explicar porque não se deve alterar o algoritmo para evitar essa troca.
1.5) Elaborar uma prova de que o algoritmo de ordenação por Inserção está correto.
1.6) Determinar a complexidade das seguintes funções, em notação O:

a) $f(n) = n.\log n + n^2 + n (\log n)^2$

b) $f(n) = 500. n^2 - n^3 + 200$

1.7) Escrever um algoritmo para cálculo de Fatorial e analisar sua complexidade.

1.8) Implementar o algoritmo ShellSort.

1.9) Mostrar os passos da ordenação pelo algoritmo ShellSort, para o CRSTRING (= string de 12 letras formado pelas 12 letras iniciais do nome). Contar o número de comparações executadas.

1.10) Fazer um estudo do tempo de execução do algoritmoShellSort, para tamanhos e situações variadas da entrada e comparar com os resultados teóricos.

1.11) Escrever um algoritmo que tenha complexidade linear de ordenação, quando a entrada estiver inversamente ordenada.

1.12) Dar exemplos da estabilidade do algoritmo de ordenação por Inserção e da não estabilidade dos algoritmos de Seleção e ShellSort.

1.13) Escrever um algoritmo que faça o número mínimo de comparações para ordenar 5 chaves. Determinar a complexidade do algoritmo.

1.14) Escrever um algoritmo de ordenação por trocas, para um conjunto de elementos onde somente estão presentes 3 valores distintos, 0, 1 e 2, por exemplo. O algoritmo deve fazer o número mínimo de trocas. Determinar a complexidade do algoritmo.

1.15) Usando o algoritmo do Exercício 1.14, mostrar os passos da ordenação do seguinte conjunto de dados:

1 2 1 0 1 2 2 1 0 2 1 0 1 2 0 2 1 1 1 2 2 0 0 2 1 2 0 1

1.16) Demonstrar que o limite inferior para o número de comparações na ordenação de 3 valores é 3 e, para 4, é 5.

1.17) Determinar limites inferiores para o número de comparações na ordenação de 1 a 10 valores.

2 Listas Lineares

Listas Lineares são um conjunto de n nós referentes a n objetos, onde as propriedades estruturais decorrem unicamente das posições relativas dos nós, dentro de uma sequência linear. Essas estruturas são tratadas sob o ponto de vista de como é a alocação de memória para os dados, existindo duas possibilidades: Alocação Sequencial e Alocação Encadeada.

2.1 Alocação Sequencial

A primeira possibilidade é termos os nós, tal que a sequência física de dados seja equivalente à sequência lógica. Vetores são o exemplo mais clássico deste tipo de estrutura. Veremos alguns algoritmos básicos relativos a essas estruturas, ligados à **Busca, Inserção e Remoção** de dados em vetores. Adicionalmente também examinaremos a importante operação de **Merge** de Vetores.

2.1.1) Busca, Inserção, Remoção e Merge em Vetores

a) Considerações iniciais sobre o problema de Busca

O problema fundamental da busca pode ser assim expresso:

"Dada uma lista com n registros, onde temos chaves primárias, isto é, cada nó i é identificado por uma chave única k_i, encontrar o nó de chave k".

Se o interesse é apenas em encontrar a chave, o algoritmo é dito de **busca**; se o interesse é o de inserção no caso da busca falhar, o algoritmo é dito de **busca e inserção**.

Nesta disciplina serão mostradas várias idéias para busca em listas. Para cada, é mostrado o algoritmo e uma análise resumida do mesmo. A análise será feita em 4 etapas:

a.) Complexidade - Analisa-se o tempo de execução do algoritmo, considerando-se o crescimento da tabela e as diversas entradas possíveis. Essa análise é desdobrada em 3 partes:

a.1) **Melhor caso** - caracterização da entrada que leva a um tempo de busca mínimo e também desse tempo.

a.2) **Pior caso** - caracterização análoga para a entrada que leva a um tempo máximo de busca

a.3) **Caso médio** - caracterização do tempo de busca médio em determinado processo de busca, considerando-se as diversas entradas possíveis e as respectivas probabilidades de ocorrência.

Para se determinar o tempo de execução, adota-se uma simplificação que é a de se estudar apenas a instrução do algoritmo mais executada. No caso dos algoritmos de busca essa instrução sempre está relacionada às comparações entre o argumento de busca e as chaves da tabela. O que se faz, portanto, é avaliar quais comparações são feitas no processo de busca e usar esse número como a referência básica.

b) Atualizações - Analisam-se os tempos médios para a inserção e deleção de chaves na tabela.

c) Buscas Especiais - Analisam-se as complexidades de caso médio para alguns tipos de busca típicos:

c.1) **Próxima chave** : busca da chave imediatamente superior a determinada chave encontrada na tabela.

c.2) **Faixa** : busca de chaves compreendidas entre dois limites dados.

c.3) **Prefixo** : busca de chaves que tenham determinado prefixo.

d) Memória: Analisa-se o montante de memória necessária para as estruturas de dados auxiliares ao método de busca.

2.1.1.1) Primeiro algoritmo de Busca sequencial em um vetor

a) Idéia: percorrer sequencialmente o vetor procurando dada chave, até se encontrar a chave ou chegar ao final do vetor.

b) Algoritmo Busca Simples.

```
Busca( k);
Início:
        i ← 1;
        Enquanto ((i ≤ n) e (Vet[i] ≠ k)):
                i ← i + 1;
        Fe;
        Se (i ≤ n) Então   Retornar i
        Senão              Retornar Nulo;
Fim;
```

c) Exemplo: adotaremos o vetor abaixo, com 10 elementos, como exemplo para os casos de busca. Embora o vetor esteja sendo apresentado ordenado, isso nem sempre ocorrerá. Para encontrar o elemento **L** são feitas 6 comparações. Para descobrir que **B** não está na tabela são feitas 10 comparações.

1	2	3	4	5	6	7	8	9	10	11
A	**C**	**E**	**F**	**I**	**L**	**M**	**O**	**P**	**X**	

d) Análise do algoritmo:

d.1) Complexidade:

Melhor caso: NC = 1

Pior caso: NC = n

Caso médio: Buscas bem sucedidas, n entradas distintas, probab. =s

$$NCmed = (1 + 2 + ... + n)/n = n/2$$

Buscas mal sucedidas, 1 entrada

$$NCmed = n/1 = n$$

d.2) Atualizações: será visto nos próximos tópicos

d.3) Buscas Especiais:

Próxima chave: TE = c.n (tem-se que olhar toda a tabela)
Faixa: TE = c.n (tem-se que olhar toda a tabela)
Prefixo: TE = c.n (tem-se que olhar toda a tabela)

d.4) Memória: Nenhuma memória adicional

e) Observação:

e.1) O retorno do algoritmo é a posição da chave no vetor, quando ela é encontrada, ou 0, caso contrário.

2.1.1.2 Busca sequencial em um vetor (otimizada)

a) Idéia: pode-se melhorar levemente o algoritmo anterior através da simplificação das condições do "loop" da busca, tornando o algoritmo otimizado. A idéia é usar-se um sentinela no final do vetor tal que a chave procurada **sempre** seja encontrada.

b) Algoritmo Busca Simples Melhorada.

```
Busca(k);
Início:
        Vet[n+1] ← k;
        i <- 1;
        Enquanto (Vet[i] ≠ k):
                i ← i + 1;
        Fe;
        Se (i ≤ n) Então    Retornar i
        Senão               Retornar Nulo;
Fim;
```

c) Exemplo: busca da chave **L**. A chave é colocada na primeira posição não utilizada do vetor (posição 11).

1	2	3	4	5	6	7	8	9	10	11
A	**C**	**E**	**F**	**I**	**L**	**M**	**O**	**P**	**X**	**L**

d) Análise do Algoritmo: Este algoritmo comporta-se exatamente igual ao anterior. Entretanto seu tempo de execução é melhor pois o loop é mais simples (70% do tempo).

2.1.1.3 Busca sequencial em um vetor ordenado

a) Idéia: outra modificação que pode ser feita na situação inicial é ordenar o vetor. A busca de chaves presentes na tabela não sofre alterações, mas pode-se melhorar a busca de chaves não presentes, tirando partido da ordenação para descobrir essa situação, sem precisar ir até o final do vetor. Para se ter a mesma otimização do caso anterior, coloca-se INFINITO na primeira posição não ocupada do vetor. INFINITO aquí tem o sentido de ser um valor maior que qualquer argumento de busca. Em Pascal, corresponde à constante MAXINT.

b) Algoritmo Busca Simples em Vetor Ordenado.

```
Busca(k);
Início
        Vet[n+1] ← ∞;
        i ← 1;
        Enquanto (Vet[i] < k):
                i ← i + 1;
        Fe;
        Se (Vet[i] = k) Então      Retornar i
        Senão                      Retornar Nulo;
Fim;
```

c) Exemplo: na busca da chave B, após o segundo teste já pode-se determinar que essa chave não está na tabela, utilizando apenas 2 comparações, em detrimento das 10 no caso inicial.

1	2	3	4	5	6	7	8	9	10	11
A	**C**	**E**	**F**	**I**	**L**	**M**	**O**	**P**	**X**	**∞**

d) Análise do Algoritmo:

d.1) Complexidade:

Melhor caso: NC = 1

Pior caso: NC = n

Caso médio:

Buscas bem sucedidas, n entradas distintas, probab. =s

NCmed = (1 + 2 + ... + n)/n = n/2

Buscas mal sucedidas, (n+1) entradas distintas,prob. =s

NCmed = (1 + 2 + ... + n+1)/(n+1) = n/2

d.2) Atualizações: Será visto nos próximos tópicos

d).3 Buscas Especiais:

Próx. chave: TE = cte (próxima posição)

Faixa: TE = c.n (tem-se que olhar metade da tabela em média)

Prefixo: TE = c.n (tem-se que olhar metade da tabela em média)

d.4) Memória: Nenhuma memória adicional

e) Observação: Notar que o sentinela é necessário para a busca de chaves superiores à maior da tabela.

2.1.1.4 Inserção de elementos no vetor

Quando se quer inserir dados no vetor, o algoritmo de inserção depende de se querer que o vetor fique ordenado ou não. Nos algoritmos descritos a seguir, **m** significa o tamanho máximo suportado pelo vetor.

a) Algoritmo Inserção em vetor não ordenado.

```
Insercao (k);
Início:
    Se (n < m) Então
        Vet[n + 1] ← k;
        n ← n + 1;
        Retornar n;
    Senão  Retornar Nulo;
Fim;
```

Como o algoritmo é muito simples, não será feita sua análise. Deve-se observar que **Overflow** significa um procedimento a ser adotado (provavelmente uma mensagem de erro) em caso de o vetor já ter sido totalmente utilizado.

b) Algoritmo Inserção em vetor ordenado.

Este algoritmo também usa uma sentinela na posição 0, tal como no algoritmo de ordenação por Inserção.

```
Insercao (k);
Início:
        Se (n < m) Então
                Vet[0] ← k;    j ← (n + 1);
                Enquanto (Vet[j -1] > k):
                        Vet[j] ← Vet[j -1];   j ← j -1;
                Fe;
                Vet[j] ← k;
                n ← n+1;
                Retornar n;
        Senão  Retornar Nulo;
Fim;
```

O funcionamento do algoritmo é muito semelhante à ordenação por Inserção, na inserção do último elemento.

2.1.1.5 Remoção de elementos no vetor

De forma análoga à inserção temos duas situações:

a) Algoritmo Remoção em vetor não ordenado.

A idéia mais simples é a de substituir o elemento removido pelo último da lista. Suporemos que o procedimento Busca(k) retorne o índice da chave **k** a ser removida do vetor e que retorne 0 quando k não estiver presente.

```
Remoção (k);
Início:
        Se (n ≠ 0) Então
                i ← Busca(k);
                Se (i ≠ 0) Então
                        Vet[i] ← Vet[n];
                        n ← n - 1;
                                Retornar i;
                Senão  Retornar Nulo
        Senão  Retornar Nulo;
```

Como o algoritmo é muito simples, não será feita sua análise. Deve-se observar que **Underflow** significa um procedimento a ser adotado (provavelmente uma mensagem de erro) em caso de o vetor estar vazio.

b) Algoritmo Remoção em vetor ordenado:
Fazendo-se as mesmas considerações que as do caso anterior, teremos:

```
Remocao (k);
Início:
        Se (n ≠ 0) Então
                i ← Busca(k);
                Se (i ≠ 0) Então
                        Para j de i até (n - 1):
                                Vet[j] ← Vet[j + 1];
                        Fp;
                        n ← n - 1;
                        Retornar i;
                Senão Retornar Nulo;
        Senão  Retornar Nulo;
Fim;
```

Este algoritmo também é semelhante à ordenação por Inserção e não será analisado.

2.1.1.6 Merge de Vetores

A operação de Merge é muito importante e útil em várias situações. Consiste da intercalação de duas listas ordenadas. De forma geral, qualquer dessas listas pode ser um vetor, uma lista encadeada, um "*handle*" resultante da pesquisa a um Banco de Dados Relacional ou um arquivo sequencial. Quando a operação é feita entre arquivos sequenciais ela é chamada "Balanced_line", sendo fundamental para a ordenação externa de arquivos.

Quando as listas forem vetores, serão mostrados dois algoritmos básicos, que sempre usam um terceiro vetor para guardar o resultado do Merge.

2.1.1.6.1 Primeiro algoritmo de Merge de Vetores:

a) Idéia: algoritmo apropriado para fazer merge em vetores distintos ou entre dois subvetores de um vetor.

b) Algoritmo Merge de Vetores.

```
Merge1 (V1, e1, d1, V2, e2, d2, V3);
Inicio:
        i ← e1;   j ← e2;    p ← 1;
        Enquanto (i ≤ d1 e j ≤ d2):
                Se (V1[i] ≤ V2 [j]) Então
                        V3[p] ← V1[i];   i ← i + 1;
                Senão
                        V3[p] ← V2[j];   j ← j + 1;
                Fs;
                p ← p + 1;
        Fe;
        Enquanto (i ≤ d1):
                V3 [p] ← V1[i];    i ← i+ 1;    p ← p + 1;
        Fe;
        Enquanto (j ≤ d2):
                V3 [p] ← V2[j];    j ← j + 1;    p ← p + 1;
        Fe;
Fim;
```

c) Exemplo: Tomando-se os seguintes vetores, V_1 e V_2, resultantes, respectivamente das ordenações parciais de EXEMPLO e FACIL,

1	2	3	4	5	6	7
E	E	L	M	O	P	X

1	2	3	4	5
A	C	F	I	L

teremos a seguinte sequência para o Merge, considerando a instrução de atribuição p/ V_3, com o destaque em vermelho para a origem da atribuição a V_3.

p	i	$V_1[i]$	J	$V_2[j]$	$V_3[p]$
1	1	E	1	A	A
2	1	E	2	C	C
3	1	E	3	F	E
4	2	E	3	F	E
5	3	L	3	F	F
6	3	L	4	I	I
7	3	L	5	L	L
8	4	M	5	L	L
9	4	M	6	-	M
10	5	O	6	-	O
11	6	P	6	-	P
12	7	X	6	-	X

d) Análise do Algoritmo:

d.1) Complexidade:

Para se determinar a complexidade, basta contar o total de atribuições a V_3, que é igual à soma dos tamanhos de V_1 e V_2.

d.2) Memória: O algoritmo exige um vetor auxiliar de tamanho igual à soma dos tamanhos dos dois vetores envolvidos.

e) Observações:

e.1) Este algoritmo faz Merge das posições E_1 a D_1 do vetor V_1 com as posições E_2 a D_2 do vetor V_2, colocando o resultado no vetor V_3. A passagem do vetor V_3 como parâmetro deve ser por referência. Eventualmente os vetores V_1 e V_2 podem ser o mesmo.

e.2) Notar que, no caso de se comparar dois valores iguais, o algoritmo faz a atribuição do vetor V_1, devido à forma como está escrito o mesmo.

e.3) Notar, ainda, que apenas um dos "loops" após o "loop" inicial é executado.

2.1.1.6.2 Segundo algoritmo de Merge de Vetores:

a) Idéia: algoritmo apropriado para fazer merge em vetores distintos.

b) Algoritmo Merge de Vetores.

```
Merge2 (V1, V2, n, m, V3);
Inicio:
        V1 [n+1] ← ∞;    V2 [m+1] ← ∞;   i ← 1;   j ← 1;
        Para p de 1 até (n + m):
                Se (V1 [i]  ≤  V2 [j]) Então
                        V3 [p] ← V1 [i];    i ← i + 1;
                Senão
                        V3 [p] ← V2 [j];    j ← j + 1;
        Fp;
Fim;
```

c) Exemplo:

A sequência de atribuições é exatamente à do algoritmo anterior

d) Análise do Algoritmo

O algoritmo tem exatamente a mesma análise do anterior

e) Observação:

Notar a simplicidade deste algoritmo, devido ao uso de sentinelas ao final dos vetores. No caso, a sentinela tem o valor ∞ (INFINITO) que significa uma chave maior que qualquer chave possível dos vetores. Aqui também o resultado fica em V_3.

2.1.2 Pesquisa binária (vetor ordenado)

a) Idéia: Uma importante forma de busca em um vetor ordenado é a Pesquisa Binária, cuja idéia é iniciar a busca no elemento central do vetor e, a cada passo, eliminar metade do conjunto das chaves do vetor, para efeito de comparações.

b) Algoritmo Pesquisa Binária.

```
Pesquisa_Binaria (k);
Inicio:
        c ← 1;    f ← n;
        Enquanto (c ≤ f):
                i ← ⌊(c + f) / 2 ⌋;
                Se (k < Vet[i])  Então              f ← i - 1
                Senão Se (k > Vet[i])  Então        c ← i + 1
                Senão                               c ← f + 1;
        Fe;
        Se (Vet[i] = k)  Então  Retornar i
        Senão                   Retornar Nulo;
Fim;
```

c) Exemplos:

A seguir são mostrados dois exemplos, o primeiro de busca bem sucedida da chave **F** e o segundo de busca mal sucedida da chave **J**, no vetor do algoritmo de Merge. A cada passo é mostrado em vermelho o elemento do vetor que é comparado. Para as variáveis c e f é mostrado o valor do início do loop.

c.1) Busca de F

c	f	i	1	2	3	4	5	6	7	8	9	10	11	12
1	**12**	**6**	**A**	**C**	**D**	**E**	**F**	**I**	**L**	**M**	**O**	**Q**	**S**	**X**
1	**5**	**3**	**A**	**C**	**D**	**E**	**F**							
4	**5**	**4**				**E**	**F**							
5	**5**	**5**					**F**							

c.2) Busca de J

c	f	i	1	2	3	4	5	6	7	8	9	10	11	12
1	12	6	A	C	D	E	F	I	L	M	O	Q	S	X
7	12	9							L	M	O	Q	S	X
7	8	7							L	M				
7	6	6						I						

d) Análise do Algoritmo:

d.1) Complexidade:

Melhor caso: NC = 1

Pior caso: NC = $\log_2(n)$ = altura da árvore de decisão

Caso médio:

d.1.1) Buscas bem sucedidas, n entradas distintas, probab. =s

NCmed = $(1 + 2 + 2 + ... + \log_2 n)/n = \log_2 n$

d.1.2) Buscas mal sucedidas, (n+1) entradas distintas,prob. =s

NCmed = $((\log_2 n - 1) + ... + \log_2 n)/n = \log_2 n$

d.2) Atualizações:

Inserção: TEmed = (1 + 2 + ... + n)/n = n/2 (tabela ordenada)

Deleção: TEmed = (1 + 2 + ... + n)/n = n/2 (tabela ordenada)

d.3) Buscas Especiais:

Próxima chave: TE = cte (próxima posição)

Faixa: TE = $O(\log_2 n)$

Prefixo: TE = $O(\log_2 n)$

d.4) Memória: Nenhuma memória adicional

e) Observações:

e.1) A análise deste algoritmo só ficará melhor compreendida depois do estudo de Árvores.

e.2) A saída do loop é feita de forma artificial em alguns casos, mediante a instrução forçada: c ← f + 1. Isto é feito para tornar o algoritmo mais eficiente.

e.3) Notar que, no exemplo de busca mal sucedida de J, houve uma comparação final com a chave I, fora do loop.

f) Outra versão do algoritmo de Pesquisa Binária

Às vezes pode ser interessante usar uma segunda versão da Pesquisa Binária, mostrada a seguir, onde a mudança é no retorno da função.

```
Pesquisa_Binaria (k);
        c ← 1;    f ← n;
        Enquanto (c ≤ f):
                i ← ⌊(c + f) / 2 ⌋;
                Se (k < Vet[i])  Então          f ← i - 1
                Senão Se (k > Vet[i])  Então    c ← i + 1
                Senão                           c ← f + 1;
        Fe;
        Retornar i;
Fim;
```

Nesta versão, quando se busca uma chave que não está no vetor, obtem-se uma chave muito próxima da chave procurada. Para ser mais preciso, obtem-se sempre a chave imediatamente inferior ou imediatamente superior à chave procurada. Isto é útil, por exemplo, na busca por faixa. No exemplo abaixo, tanto a busca da chave ***P*** quanto a da ***R*** têm ***Q*** como última chave comparada.

c.2) Busca de P

c	f	i	1	2	3	4	5	6	7	8	9	10	11	12
1	**12**	**6**	**A**	**C**	**D**	**E**	**F**	**I**	**L**	**M**	**O**	**Q**	**S**	**X**
7	**12**	**9**							**L**	**M**	**O**	**Q**	**S**	**X**
10	**12**	**11**										**Q**	**S**	**X**
10	**10**	**10**										**Q**		

c.2) Busca de J

c	f	i	1	2	3	4	5	6	7	8	9	10	11	12
1	**12**	**6**	**A**	**C**	**D**	**E**	**F**	**I**	**L**	**M**	**O**	**Q**	**S**	**X**
7	**12**	**9**							**L**	**M**	**O**	**Q**	**S**	**X**
10	**12**	**11**										**Q**	**S**	**X**
10	**10**	**10**										**Q**		

2.1.3 Pilhas e Filas

Listas sequenciais são usadas para uma quantidade pequena de elementos e onde, em geral há poucas atualizações (inserções e remoções). Os algoritmos vistos para vetores em geral estão nesse caso. Em algumas situações, as inserções e remoções são feitas em locais bem determinados das listas. As estruturas são chamadas, então de Pilhas, Filas ou Deques.

Pilhas são listas onde inserções e remoções são feitas no final da lista.

Filas são listas onde inserções são feitas no final e as remoções, no início.

Deques são listas onde inserções e remoções são feitas em qualquer extremidades.

2.1.3.1 Pilhas

Como foi descrito, **Pilhas** são listas sequenciais onde as inserções e remoções são feitas no final das mesmas. Estas estruturas são muitas vezes denominadas **LIFO** (Last In, First Out), pois a remoção é feita no final e então as chaves são retiradas em ordem inversa da Inserção.

Há muitos processos onde esse fato ocorre e nesses casos estas estruturas são as mais adequadas para representar a situação. Pilhas de pratos lavados em restaurantes ilustram bem a idéia.

A seguir veremos os algoritmos para inserção (**PUSH**) em Pilhas e também para remoção (**POP**), onde as pilhas utilizam vetores como subestrutura básica. Denominaremos a pilha como **P**. A cada pilha associa-se uma variável: **topo**, que indica a última posição do vetor utilizada para a pilha (consideraremos que o vetor contém **m** células). Inicialmente temos **topo** = **0**;

a) Algoritmo Inserção na Pilha P.

```
PUSH (k);
Início:
        Se (topo ≠ m) Então
                topo ← topo + 1;   P[topo] ← k;
                Retornar topo;
        Senão
                Retornar Nulo;
Fim;
```

f) Algoritmo Remoção na Pilha P.

```
POP(k);   Obs: k é parâmetro de retorno.
Início:
        Se (topo ≠ 0) Então
                k ← P[topo];
                topo ← topo - 1;
        Senão
                k ← Nulo;
Fim;
```

c) Primeiro exemplo de uso de Pilhas: inversão de sequência.

Um exemplo bastante simples do uso de pilhas seria a inversão de uma sequência lida de um arquivo A. O algoritmo poderia ser o seguinte:

```
Inversão;
Inicio:
        Abrir A;
        Enquanto (A não vazio):
                Ler (A,k);
                PUSH (k);
        Fe;
        Enquanto (topo > 0);
                POP(k);
                Imprimir k;
        Fe;
Fim;
```

d) Segundo exemplo de uso de Pilhas: Cálculo de Expressões em **Notação Polonesa**.

Tomemos uma expressão aritmética escrita na forma convencional:

A + B * (C - D + E / F)

Essa notação não é adequada às máquinas, pelas ambiguidades existentes. Três outras notações são mais convenientes para uso em máquinas. A primeira é uma **notação totalmente parentisada**:

(A + (B * ((C - D) + (E / F))))

onde o número de parêntesis = número de operadores -1

Outras notações mais simples, que não utilizam parêntesis são as **notações polonesa direta e reversa**. Na primeira dessas notações, a notação polonesa direta, os operadores vêm antes dos operandos e, tomando-se a expressão da esquerda para a direita, cada vez que se encontra um par de operandos em sequência eles têm que ser operados segundo a operação mais à esquerda, retirando esses 3 componentes da expressão e substituindo-os pelo operando = resultado. Para o exemplo, teríamos:

+A * B + - C D / E F

Na notação polonesa reversa, a diferença é que os operadores ficam à esquerda dos operandos e cada resultado parcial é calculado quando se encontra um operador. Teríamos, para o exemplo:

A B C D - E F / + * +

A notação polonesa reversa é utilizada em máquinas, especialmente em máquinas de calcular. Esse cálculo pode ser feito através do uso de uma pilha adicional, usando o seguinte algoritmo, onde o vetor Vet, de tamanho n, contém a expressão e P é a pilha auxiliar tal que, ao final do processamento da expressão, o resultado ficará na posição 1 da pilha. No algoritmo a seguir, supõe-se que o vetor Vet contenha uma expressão corretamente escrita na notação polonesa reversa.

```
Calculo;
Início:
        Para i de 1 a n:
                Se (Vet[i] é Operador) Então
                        POP(op2);
                        POP(op1);
                        PUSH ( Resultado(op1, op2, Vet[i]));
                Senão
                        PUSH (Vet[i]);
        Fp;
        Retornar (P[1]);
Fim;
```

e) Exemplo do algoritmo de Cálculo:

Seja a expressão: 1 + 3 * (12 - 2 + 70 / 35), representada em notação polonesa pelo Vetor abaixo:

1	2	3	4	5	6	7	8	9	10	11
1	**3**	**12**	**2**	**-**	**70**	**35**	**/**	**+**	*****	**+**

A situação da pilha P, ao final do loop do algoritmo, é mostrada a seguir:

i	Vet[i]	1	2	3	4	5
1	**1**	**1**				
2	**3**	**1**	**3**			
3	**12**	**1**	**3**	**12**		
4	**2**	**1**	**3**	**12**	**2**	
5	**-**	**1**	**3**	**10**		
6	**70**	**1**	**3**	**10**	**70**	
7	**35**	**1**	**3**	**10**	**70**	**35**
8	**/**	**1**	**3**	**10**	**2**	
9	**+**	**1**	**3**	**12**		
10	*****	**1**	**36**			
11	**+**	**37**				

O resultado do cálculo é, portanto 37.

f) Terceiro exemplo de uso de Pilhas: Conversão de Notação Parentisada para **Notação Polonesa reversa**.

A expressão corretamente parentisada é colocada no vetor Vet, com n posições. a seguir é mostrado um algoritmo para conversão de notações, gerando a notação da expressão aritmética em notação polonesa reversa no vetor PR e utilizando uma pilha P para armazenagem intermediária dos operadores. Na conversão de notações os parêntesis são descartados, os operandos são colocados na mesma ordem que na expressão original e os operadores são empilhados, sendo um operador desempilhado e colocado na expressão convertida toda vez que é encontrado um sinal de fechar parêntesis.

Conversão:

Início:

p ← 0;

Para i de 1 a n:

Se (Vet[i] é operando) Então

p ← p + 1; PR[p] ← Vet[i];

Senão Se (Vet[i] é operador) Então

PUSH (Vet[i]);

Senão Se (Vet[i] = ')') Então

POP(op); p ← p + 1; PR[p] ← op;

Fp;

Fim;

g) Exemplo do algoritmo de Conversão;

Seja a expressão totalmente parentisada: (A + (B * ((C - D) + (E / F)))) representada pelo Vetor abaixo:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
(	A	+	(	B	*	(	(	C	-	D	)	+	(	E	/	F	)	)	)	)

A situação do vetor Pol e da pilha P, ao final do loop do algoritmo, é mostrada a seguir:

		Vetor Pol										
i	Vet[i]	1	2	3	4	5	6	7	8	9	10	11
1	(											
2	A	A										
3	+	A										
4	(	A										
5	B	A	B									
6	*	A	B									
7	(	A	B									
8	(	A	B									
9	C	A	B	C								
10	-	A	B	C								
11	D	A	B	C	D							
12	)	A	B	C	D	-						
13	+	A	B	C	D	-						
14	(	A	B	C	D	-						

Pilha P			
1	2	3	4
+			
+			
+			
+	*		
+	*		
+	*		
+	*		
+	*	-	
+	*	-	
+	*		
+	*	+	
+	*	+	

15	E	A	B	C	D	-	E						+	*	+	
16	/	A	B	C	D	-	E						+	*	+	/
17	F	A	B	C	D	-	E	F					+	*	+	/
18	)	A	B	C	D	-	E	F	/				+	*	+	
19	)	A	B	C	D	-	E	F	/	+			+	*		
20	)	A	B	C	D	-	E	F	/	+	*		+			
21	)	A	B	C	D	-	E	F	/	+	*	+				

2.1.3.2 Filas

Como foi descrito, **Filas** são listas sequenciais onde as inserções são feitas no final e remoções no início das mesmas.

denominadas **FIFO** (First In, First Out), pois a remoção é feita na mesma ordem.

Há muitos processos onde esse fato ocorre e nesses casos estas estruturas são as mais adequadas para representar a situação. Filas em Bancos ilustram bem a idéia.

A seguir veremos os algoritmos para inserção e remoção em Filas, onde se utilizam vetores como subestrutura básica. Denominaremos a pilha como **F**. Para cad Fila é necessário terem-se duas variáveis associadas: **r (retaguarda)**, que indica a última posição do vetor ocupada pela Fila e **f (frente)**, que indica a posição inicial do vetor ocupada pela Fila (consideraremos que o vetor contém **m** células).

Os algoritmos para essas operações são mais elaborados, porque, para se utilizar melhor o vetor, a fila será considerada uma estrutura **circular**, no sentido de que quando a fila atingir a posição **m**, poderá se estender pelo início do vetor, caso haja posições livres no início. As figuras abaixo ilustram a idéia, onde o vetor usado tem m = 7. Inicialmente temos **f = 0, r = 0**;

1	2	3	4	5	6	7
E	X	E	M	P		

a.1) Fila após a inserção dos elementos E, X, E, M, P, L;

f = 1; r = 5;

1	2	3	4	5	6	7
			M	**P**	**L**	***O***

a.2) Fila após a remoção dos elementos E, X, E e inserção de L, O;

f = 4; r = 7;

1	2	3	4	5	6	7
F	***A***		**M**	**P**	**L**	***O***

a.3) Fila após a inserção dos elementos F, A.

f = 4; r = 2;

a) Algoritmo Inserção na Fila Q.

Enfila (k);

```
Início:
        prov ← (r mod m) + 1;
        Se (prov ≠ f) Então
                r ← prov;  Q[r] ← k;
                Se (f = 0) Então
                        f ← 1;
                Retornar r;
        Senão
                Retornar Nulo;
Fim;
```

b) Algoritmo Remoção na Fila Q.

Desenfila(k); Obs: k é parâmetro de retorno.

```
Início:
        Se (f ≠ 0) Então
                k ← Q[f];
                Se (f = r) Então
                        f ← 0;  r ← 0;
                Senão
                        f ← (f mod m) + 1;
        Senão
                k ← Nulo;
Fim;
```

c) Primeiro exemplo do Uso de Filas: Impressão de "runs"

Um exemplo bastante simples do uso de Filas é a impressão de "runs" de um arquivo. Um "run" é uma sequência crescente de chaves. Quer-se, que a impressão de um "run" seja precedida pelo número de elementos do mesmo. O algoritmo poderia ser o seguinte:

```
Impressao_de_Runs;
Inicio:
        Abrir A;
        Enquanto (A não vazio):
                Ler (A, k);
                Se (f ≠ 0) e (k < Q[r]) Então
                        Imprimir (r);
                        Enquanto (f ≠ 0):
                                Desenfila (t);
                                Imprimir (t);
                        Fe;
                Enfila (k);
        Fe;
        Imprimir (r);
        Enquanto (f ≠ 0):
                Desenfila (t);
                Imprimir (t);
        Fe;
Fim;
```

d) Exemplo de Impressão de Runs:

Seja o arquivo A:

1	2	3	4	5	6	7	8	9	10	11
B	**D**	**H**	***C***	***C***	**M**	**Q**	**S**	**T**	**S**	**T**

A situação da Fila seria a seguinte:

1	2	3	4	5	6
B					
B	**D**				
B	**D**	**H**			
C					
C	***C***				

C	C	M			
C	C	M	Q		
C	C	M	Q	S	
C	C	M	A	S	T
S					
S	T				

E seria impresso:

2 B D H
6 C C M Q S T
2 S T

e) Segundo exemplo do Uso de Filas: Coloração de Áreas.

É comum o uso de matrizes para representações de imagens, tal que cada parte da imagem é representada por células contíguas na matriz. O reconhecimento de determinada área (parte) da imagem é dado através da identificação de todas as células contíguas entre sí. Suponhamos que, para uma imagem representada em uma matriz de caracteres, células contíguas sejam apenas aquelas ao norte, oeste, sul e leste de determinada célula. Queremos colorir toda uma área onde dada célula (p, q) esteja inserida e convencionaremos que todos as células dessa área estejam em branco. Desta forma, o contorno da área será ou os limites da matriz ou alguma célula contígua que não esteja em branco (para simplificar, = 'x'). Pode-se usar uma fila para identificar a área, substituindo-se cada célula da área por um caracter que representa uma cor 'c'. O algoritmo a seguir processa uma matriz Mat(n x n), cujos elementos inicialmente têm valor 'x' ou branco e identifica a área à qual pertence a célula (p, q). É usada uma fila F para armazenar células da área. Neste caso cada elemento da fila é um par de números, representando as coordenadas (x, y) da célula na matriz Mat.

Coloração(p, q);
Inicio:
    EsvaziaFila;
    Enfila(p, q);
    Enquanto (f ≠ 0):
        Desenfila(a, b);
        Enfila(a-1, b); Enfila(a, b-1); Enfila(a+1, b); Enfila(a, b+1);
    Fe;
Fim;

No presente caso, foram transferidas para o procedimento Enfila, o testes de limites da matriz e de vizinhança, bem como a coloração de área vizinha. É interessante transferir esses processamentos para o procedimento Enfilar, para não ter que repetí-los 4 vezes no procedimento principal, quando se consideram os 4 tipos de vizinho (norte oeste, sul, leste, nesta ordem). Podemos ter o seguinte procedimento:

```
Enfila (a, b);
Início:
        Se (a ≥ 1) e (a ≤ n) e (b ≥ 1) e (b ≤ n) e (Mat[a,b] = ' ') Então
                prov ← (r mod m) + 1;
                Se (prov ≠ f) Então
                        r ← prov;  Mat[a, b] ← 'c';  Q[r].x ← a;  Q[r].y ← b;
                        Se (f = 0) Então
                                f ← 1;
                        Retornar r;
                Senão
                        Retornar 0;
Fim;
```

f) Exemplo de Coloração de área:

Seja a matriz Mat 5 x 5 abaixo, onde quer-se colorir a área que contém a célula (2, 4):

				x
x		x	.	x
	x			x
		x	x	
x	x			x

A situação da fila F, implementada em um vetor de tamanho 4, ao longo do algoritmo seria a seguinte, considerando-se a observação da fila no início do loop principal:

(2,4			
(1,4)	**(3,4)**		
	(3,4)	**(1,3)**	
		(1,3)	**(3,3)**
(1,2)			**(3,3)**

(1,2)			
(1,1)	(2,2)		
	(2,2)		

Notar que, após a saída do elemento (1,2) a fila esvazia-se. Devido a isso, a próxima entrada na fila foi no endereço 1. Ao final, Mat ficaria assim:

c	c	c	c	x
x	c	x	c	x
	x	c	c	x
		x	x	
x	x			x

Uma questão interessante em relação à fila neste tipo de aplicação é saber-se qual o tamanho mínimo que a fila deve ter, em função da dimensão da matriz n x n. É claro que n^2 é um limite natural para isso (número de células da matriz), mas esse valor está superdimensionado. No pior caso, uma fila de tamanho 2n é suficiente.

Outra característica interessante do processamento através de filas é que normalmente pode-se associar o conceito de distância dos elementos que entram na fila, em relação ao elemento inicial. E, neste caso, o processamento é tal que os elementos entram na fila em ordem crescente da distância ao elemento inicial. Desta forma, esse tipo de algoritmo é usado em muitos problemas de determinação de distâncias mínimas (ver exercício 2.19).

g) Outros exemplos do uso de Filas associados a matrizes:

Inúmeros outros problemas têm estrutura muito parecida com aquela do problema anterior, ou seja, o problema envolve o processamento de uma matriz onde tem-se que identificar e enfilar elementos vizinhos, de forma sucessiva. O uso de uma fila normalmente é adequado a esses problemas. Um mecanismo adicional em muitos algoritmos é a marcação do elemento que já entrou na fila, para evitar múltiplas entradas do mesmo, o que seria um erro. A marcação pode ser feita com o uso de uma matriz auxiliar, inicialmente totalmente desmarcada e que marca a posição de todas as células enfiladas. O enfilamento também tem que testar essa situação. No caso anterior, a atribuição de uma cor à célula funcionou como uma marcação natural.

Dentre os problemas análogos podemos destacar o da identificação de aglomerados numa matriz. Neste caso, a matriz representa aglomerados geográficos tal que cada célula da matriz representa um elemento. Elementos

de um mesmo aglomerado são vizinhos na matriz de forma horizontal, vertical e, eventualmente, diagonal. O algoritmo é análogo ao anterior. No exemplo a seguir os elementos de um aglomerado são identificados horizontal, vertical e diagonalmente, podendo haver vários aglomerados. É feito um loop varrendo-se a matriz e identificando-se o início de um aglomerado. Quando é identificado esse início, interrompe-se a varredura para fazer o processamento daquele particular aglomerado, que será identificado por letras a, b..., sucessivamente.

h) Exemplo do processamento de Aglomerado:

Seja a matriz Mat 5 x 5 abaixo:

	x			
x		x		
	x			x
			x	
x	x			x

A situação da fila F ao longo do algoritmo seria a seguinte:

(1,2)			
(2,1)	(2,3)		
	(2,3)	(3,2)	
		(3,2)	
(3,5)			
(4,4)			
(5,5)			
(5,1)			
(5,2)			

E, ao final, Mat ficaria assim:

	a			
a		a		
	a			b
			b	
c	c			b

2.2 Alocação Encadeada

Como foi visto nos casos de alocação sequencial, em geral as listas são implementadas através de vetores, cujo tamanho máximo é dado desde o início

do algoritmo usado, mesmo que apenas uma pequena porção seja utilizada. Isso pode levar a uma pobre alocação de memória, especialmente quando houver várias listas envolvidas. Uma alternativa é usar-se a **alocação encadeada** ou **alocação dinâmica** de memória.

Na alocação encadeada os nós logicamente vizinhos de uma lista não são mais necessariamente vizinhos em termos de memória. Os nós podem estar aleatoriamente dispostos e a indicação de vizinhança é feita através de um campo de endereço (chamado **ponteiro**). Desta forma, o nó da lista contem pelo menos dois campos: a chave e o ponteiro, como ilustra a figura:

O último nó contém um ponteiro nulo, que pode receber várias denominações: **λ, Null, Nil, Nulo**, dependendo do contexto.

Em Pascal, para se declarar uma lista encadeada usa-se o comando Type da seguinte forma:

```
Type No = Record
            chave: integer;
            prox: ↑ No
          End;
```

O símbolo ↑ indica variável do tipo Endereço. Apontadores para nós da lista são também variáveis tipo Endereço.

Para se efetuar a alocação dinâmica de memória, o sistema operacional mantém uma **lista de espaços disponíveis**, que é uma lista encadeada. A a interação entre um programa e o sistema operacional normalmente é feita através de comandos do tipo:

Alocar(p) ⇒ o sistema operacional retira um nó da lista de espaço disponível e faz a variável p apontar para esse nó.

Desalocar(p) ⇒ o nó apontado por p é retornado para a lista de espaço disponível.

Em um nó pode existir mais de um ponteiro. As listas onde existem apenas um ponteiro são denominadas **Listas Simplesmente Encadeadas**. Qualquer estrutura encadeada necessita de um apontador para o início da lista. Convencionaremos que este apontador tem sempre o nome **cab**. Algumas vezes o nó inicial da lista é apenas um nó com endereço e sem conteúdo. A lista então é dita ter um **nó cabeça**.

2.2.1 Busca, Inserção e Remoção em Listas Simplesmente Encadeadas

Serão mostradas, neste ítem, as versões dos algoritmos básicos de listas lineares para o caso de alocação encadeada. Antes, porém, será visto um algoritmo simples para impressão de uma lista encadeada, para exemplificar esta estrutura de dados. Nos algoritmos a seguir supõe-se que a lista tem o nó cabeça cab.

a) Algoritmo Impressão de uma lista encadeada.

```
Início:
        p ← cab↑.prox;
        Enquanto (p ≠ Nil):
                Imprime (p↑.chave);
                p ← p↑. prox;
        Fe;
Fim;
```

b) Algoritmo Busca em uma lista encadeada.

No algoritmo a seguir pont indicará o nó que contém a chave procurada. Se essa chave não for encontrada, pont recebe Nil.

```
Busca (k, pont);
Início:
        p ← cab↑.prox;   pont ← Nil;
        Enquanto (p ≠ Nil):
                Se (p↑.chave ≠ k) Então
                        p ← p↑.prox;
                Senão
                        pont ← p;
                        p ← Nil;
        Fe;
Fim;
```

Observar o artifício usado para simplificar as condições do loop: a variável p recebe Nil em caso de busca bem sucedida.

c) Algoritmo Busca em uma lista encadeada ordenada.

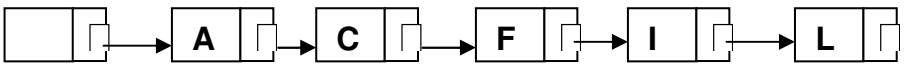
Quando a lista encadeada está ordenada, a busca pode ser levemente modificada, para facilitar futuras inclusões.

No algoritmo a seguir **pont** indicará apontará para o nó que contém a chave procurada. Se essa chave não for encontrada, **pont** recebe Nil. É acrescentado, também um outro parâmetro de retorno, o ponteiro **ant**, que apontará para o elemento imediatamente anterior ao procurado. Essas convenções facilitarão futuras inserções.

```
Busca_ordenada(k, ant, pont);
Início:
        ant ← cab;   p ← cab↑.prox;   pont ← Nil;
        Enquanto (p ≠ Nil):
                Se (p↑.chave < k) Então
                        ant ← p;
                        p ← p↑.prox;
                Senão
                        Se (p↑.chave = k) Então
                                pont ← p;
                        p ← Nil;
        Fe;
Fim;
```

d) Exemplos de busca em lista encadeada ordenada:

A seguir são mostrados dois exemplos, o primeiro de busca bem sucedida da chave **F** e o segundo de busca mal sucedida da chave **J**, na lista encadeada ordenada abaixo. A cada passo é mostrado o elemento do vetor que é comparado, bem como o conteudo dos nós apontados pelas variáveis p, ant, pont, no inicio do loop.



d.1) Busca de F

p	ant	pont
A	**-**	**Nil**
C	**A**	**Nil**
I	**C**	**Nil**
F	**I**	**Nil**
Nil	**I**	**F**

d.2) Busca de J

p	ant	pont
A	**-**	**Nil**
C	**A**	**Nil**
I	**C**	**Nil**
F	**I**	**Nil**
L	**F**	**Nil**
Nil	**F**	**Nil**

e) Algoritmo Inserção em uma lista encadeada.

A inserção de um novo elemento na lista pode ser bem simples, pois pode ser feita no início.

```
Insere(k);
Início:
  Busca (k, p);
  Se (p = Nil) Então
       Alocar(p);  p↑.chave ← k;    p↑.prox ← cab↑.prox;   cab↑.prox ← p;
  Senão
       p ← Nulo;
  Retornar p;
Fim;
```

f) Algoritmo Inserção em uma lista encadeada ordenada.

Quando a lista encadeada está ordenada, é usado o algoritmo de busca adequado, também simples.

```
Insercao_ordenada(k);
Início:
  Busca_ordenada (k, ant, p);
  Se (p = Nil) Então
       Alocar(p);   p↑.chave ← k;  p↑.prox ← ant↑.prox;   ant↑.prox ← p;
  Senão
       p ← Nulo;
  Retornar p;
Fim;
```

g) Exemplos de inserção em lista encadeada ordenada:

A seguir são mostrados dois exemplos sobre a lista do ítem d, o primeiro de inserção bem sucedida da chave **G** e o segundo de inserção mal sucedida da chave **L**. São mostrados os os valores dos ponteiros depois da execução da instrução Busca(k, ant, p) e ao final do algoritmo.

g.1) Inserção de G

cab↑.prox	p	ant↑.prox
A	**Nil**	**L**
A	**G**	**G**

g.2) Inserção de L

cab↑.prox	p	ant↑.prox
A	**L**	**G**
A	**L**	**G**

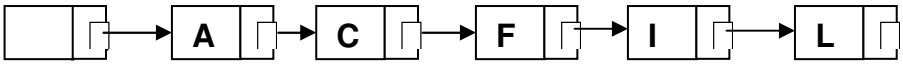
h) Algoritmo Remoção em uma lista encadeada, ordenada ou não.

A remoção em lista não ordenada exige uma rotina não mostrada, que devolve também o elemento anterior na busca, tal como no caso de lista ordenada. Suporemos que exista esse algoritmo para o caso não ordenado. O algoritmo de remoção é o mesmo.

```
Remocao(k);
Início:
        Busca (k, ant, p);
        Se (p ≠ Nulo) Então
                ant↑.prox ← p↑.prox;   Desalocar (p);
        Senão
                ant ← Nulo;
        Retornar ant;
Fim;
```

i) Exemplos de remoção em lista encadeada:

A seguir são mostrados dois exemplos sobre a lista do ítem d, o primeiro de remoção bem sucedida da chave **A** e o segundo de inserção mal sucedida da chave **M**. São mostrados os os valores dos ponteiros depois da execução da instrução Busca(k, ant, p) e ao final do algoritmo.



i.1) Remoção de A

cab↑.prox	p	ant↑.prox
A	***A***	***A***
C	**-**	***C***

i.2) Remoção de M

cab↑.prox	p	ant↑.prox
A	**Nil**	**L**
A	**Nil**	**L**

2.2.2 Pilhas e Filas

A implementação de pilhas com listas encadeadas praticamente não difere daquela da alocação sequencial. Para Filas há uma pequena diferença, já que não há mais necessidade de se considerar a fila como uma lista circular. Os algoritmos são mostrados a seguir, sendo claro que as variáveis **topo, f e r**, passam a ser variáveis de endereço.

a) Algoritmo Inserção em uma pilha.

```
PUSH (k);
Início:
        Alocar (p);  p↑.chave ← k;   p↑.prox ← topo;
        topo ← p;
Fim;
```

b) Algoritmo Remoção em uma pilha.

```
POP(k);  Obs:  k é variável de retorno.
Início:
        Se (topo ≠ Nulo) Então
                p ← topo;   k ← topo↑.chave ;  topo ← topo↑.prox;
                Desalocar (p);
        Senão
                k ← Nulo;
Fim;
```

c) Algoritmo Inserção em uma Fila.

```
Enfila (k);
Início:
        Alocar (p);   p↑.chave ← k;   p↑.prox ← Nulo;
        Se (r ≠ Nulo) Então
                r↑.prox ← p;
        Senão
                f ← p;
        r ← p;
Fim;
```

d) Algoritmo Remoção em uma Fila.

```
Desenfila (k);   Obs: k é variável de retorno.
Início:
        Se (f ≠ Nulo) Então
                p ← f;   k ← f↑.chave;   f ← f↑.prox;
                Se (f = Nulo) Então
                        r ← Nulo;
                Desalocar (p);
        Senão
                k ← Nulo;
Fim;
```

e) Exemplo do Uso de Filas: Ordenação por Distribuição

Este método de ordenação é equivalente ao método mecânico utilizado nas antigas máquinas de ordenação de cartões perfurados.

A chave de ordenação é um número decimal com *nd* dígitos. O método consiste em se fazer *nd* passoss, onde, em cada um deles, usa-se uma das posições da chave como referência para ordenação, tomando-se a sequência de posições da direita para a esquerda. Em cada um desses passos é feita uma ordenação estável, segundo a posição considerada. A ordenação consiste em se distribuir as chaves em 10 listas, uma para cada dígito. Ao final do passo as listas são agregadas, obedecendo-se sua ordem.

Exemplo: Sejam as chaves: 981, 872, 253, 002, 209, 403, 353, 266, 116.

1o passo (dígito das unidades):

1: 981
2: 872 → 002
3: 253 → 403 → 353
6: 266 → 116
9: 209

Agregação final: 981, 872, 002, 253, 403, 353, 266, 116, 209

2o passo (dígito das dezenas):

0: 002 → 403 → 209
1: 116
5: 253 → 353
6: 266
7: 872
8: 981

Agregação final: 002, 403, 209, 116, 253, 353, 266, 872, 981

3o passo (dígito das centenas):

0: 002
1: 116
2: 209 → 253 → 266
3: 353
4: 403
8: 872
9: 981

Agregação Final: 002, 116, 209, 253, 266, 353, 403, 872, 981

Algoritmo Ordenação por Distribuição.

```
Distribuicao;
Início:
        Para d de 1 até nd:
                Para i de 1 a n:
                        Desenfila (10, k);
                        m ← Digito (d, k);
                        Enfila (m, k);
                Fp;
                Para j de 0 até 9:
                        Enquanto (Filas[j].f ≠ Nulo):
                                Desenfila (j, k);
                                Enfila (10, k);
                        Fe;
                Fp;
        Fp;
Fim;
```

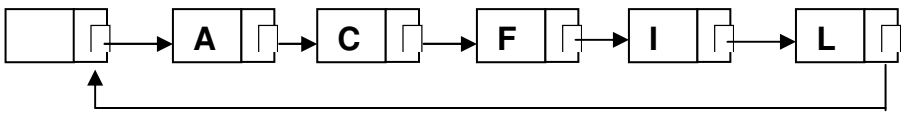
Este algoritmo utiliza 11 filas: as 10 primeiras correspondem aos dígitos 0 a 9 e a fila 10 é usada para agregação, inclusive a inicial.

2.2.3 Listas Circulares e Listas Duplamente Encadeadas

O encadeamento oferece muitas alternativas para criação de estruturas de dados. Duas delas serão mostradas a seguir: Listas Circulares Encadeadas e Listas Duplamente Encadeadas.

a) Listas Circulares Encadeadas

Uma lista circular encadeada é uma lista com cabeça, onde o último elemento, ao invés de apontar para Nil, aponta para o nó cabeça. A figura abaixo ilustra a idéia.



b) Algoritmo Busca em Lista Circular Encadeada.

O algoritmo de busca, neste caso pode usar a mesma idéia mostrada em algoritmos anteriores de se colocar a chave procurada no nó cab, o que garante que sempre se encontrará a chave. O algoritmo é descrito a seguir, usando dois parâmetros pont que indica a posição onde está a chave e ant, que é o ponteiro para o nó anterior.

```
Busca_circular(k, ant, pont);
Início:
        cab↑.chave ← k;  ant ← cab;    pont ← cab↑.prox;
        Enquanto (pont↑.chave ≠ k;):
                ant ← pont;
                pont ← pont↑.prox;
        Fe;
Fim;
```

Evidentemente que o teste para descobrir se a busca foi bem sucedida, deve-se checar se pont = cab;

c) Exemplos de busca na lista circular.

Serão exemplificadas as buscas das chaves **F** (presente na lista) e **M** (ausente da lista).

c.1) Busca de F

ant	pont
F	**A**
A	**C**
C	**F**

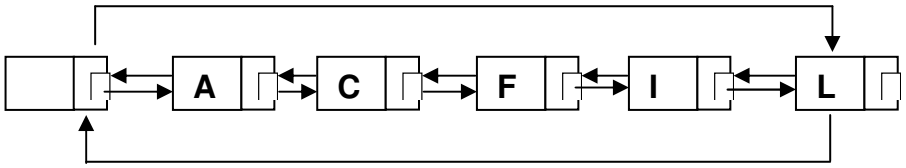
c.2) Busca de M

ant	pont
M	**A**
A	**C**
C	**F**

F	I
I	L
L	M

d) Listas Circulares Duplamente Encadeadas

Uma lista circular duplamente encadeada é uma lista circular encadeada nos dois sentidos. Cada nó, então tem dois links, um para a frente (**prox**) e outro para trás (**ante**). A figura abaixo ilustra a idéia.



e) Algoritmo Busca em Lista Duplamente Encadeada Ordenada.

O algoritmo de busca, neste caso pode usar a mesma idéia mostrada em algoritmos anteriores de se colocar a chave procurada no nó cab, o que garante que sempre se encontrará a chave. O algoritmo é descrito a seguir, e só necessita usar o parâmetro de retorno, pont, que devolve três tipos de retorno: se a chave é encontrada, o endereço do nó, se não, o endereço da primeira chave acima da chave procurada e cab caso essa chave seja maior que todas.

```
Busca_Dupl_Encadeada(k, pont);
Início:
        cab↑.chave ← k;  pont ← cab↑.prox;
        Enquanto (pont↑.chave < k):
                pont ← pont↑.prox;
        Fe;
Fim;
```

f) Algoritmo Inserção em Lista Duplamente Encadeada Ordenada.

```
Insercao_Dupl_Encadeada(k, p);
Início:
        Busca_Dupl_Encadeada(k, pont);
        Se (pont = cab) ou (pont↑.chave ≠ k) Então:
                ant ← pont↑.ante;
                Alocar(p);   p↑.chave ← k;   p↑.prox ← pont; p↑.ante ← ant;
                ant↑.prox ← p;
                pont↑.ante ← p;
        Senão
                p ← Nulo;
Fim;
```

g) Algoritmo Remoção em Lista Duplamente Encadeada Ordenada.

```
Remocao_Dupl_Encadeada(k);
Início:
        Busca_Dupl_Encadeada(k, pont);
        Se (pont ≠ cab) e (pont↑.chave = k) Então:
                ant ← pont↑.ante;     post ← pont↑.prox;
                ant↑.prox ← post;     post↑.ante ← ant;
                Desalocar(pont);
        Senão
                Mensagem de erro;
Fim;
```

2.3 Complementos

2.3.1 Buscas sequenciais com frequências de acesso desiguais

a) Frequências conhecidas:

A melhor situação é quando $f_1 > f_2 > \ldots > f_n$

b) Frequências desconhecidas

Há dois métodos para lidar com essa situação:

b.1) Usar contador de acessos e reorganizar periodicamente a tabela.

b.2) Usar uma tabela autoorganizável, onde cada elemento buscado passa para a posição inicial da tabela.

c) Exemplo de Tabela autoorganizável:

	Situação inicial:	Após a busca de FA:	Após as buscas: FA,FA,SOL,MI,RE,FA,DO,RE
1	DO	FA	RE
2	RE	DO	DO
3	MI	RE	FA
4	FA	MI	MI
5	SOL	SOL	SOL
6	LA	LA	LA
7	SI	SI	SI

Obs: Este método só funciona eficientemente usando-se alocação encadeada

Exercício 3: Implementar um programa para a tabela autoorganizável.

2.3.2 Árvore de Decisão da Pesquisa Binária (ADPB)

É uma árvore que auxilia a compreensão das sequências de comparações nas buscas feitas através da pesquisa binária.

a) Processo de construção da ADPB

Processo descritivo:

A raiz da árvore é o nó de índice (n div 2). A subárvore esquerda é formada pelos índices da faixa 1 - (n-1). A subárvore direita é formada pelos índices da faixa (n+1) - n. Faz-se recursivamente o processo até que a faixa cheque a 1 elemento.

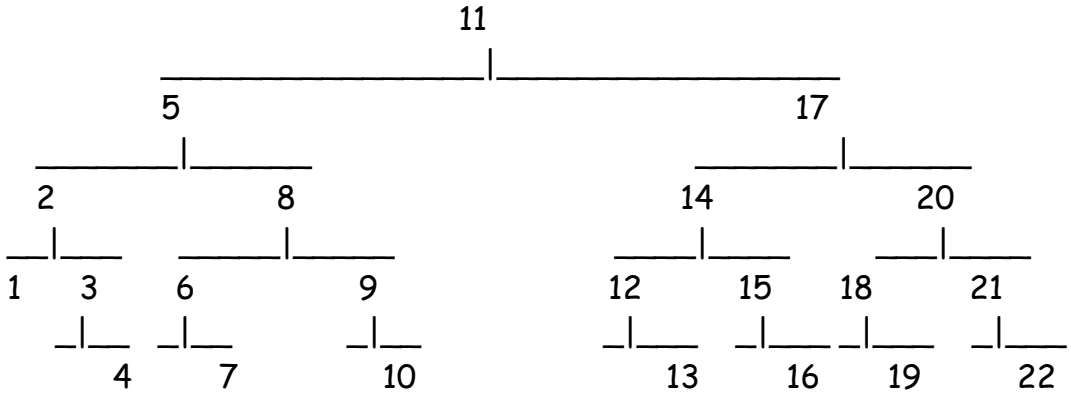
b)Propriedades da ADBP:

b.1) A ADPB é uma árvore cheia.

b.2) A altura da ADPB = int(log2(n))+1 = teto(log2(n+1))

b.3) O percurso, em ordem simétrica da ADPB fornece as chaves em ordem crescente.

c) Exemplo de ADPB p/ n = 22.



2.4 Exercícios

2.1) Escrever um algoritmo para transpor os dados de um vetor ("espelhar" os elementos do vetor)

2.2) Escrever um algoritmo para verificar se um vetor é um palíndromo (palíndromo é um string cuja leitura é idêntica se feita da esquerda para a direita ou vice_versa)

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

2.4) Implementar um dos algoritmos para Merge de Vetores.

2.5) Modificar o segundo algoritmo de Merge para a situação de alocação encadeada.

2.6) Escrever um algoritmo que faz a busca por faixa num vetor ordenado, aproveitando-se da pesquisa binária.

2.7) Implementar o algoritmo de Pesquisa Binária.

2.8) Demonstrar que $\lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil$.

2.9) Implementar o algoritmo de cálculo de uma expressão em Notação polonesa reversa.

2.10) Implementar um algoritmo de conversão de uma expressão aritmética totalmente parentisada para notação polonesa reversa.

2.11) Fazer um algoritmo para verificar se um vetor contém uma expressão aritmética correta totalmente parentisada.

2.12) Fazer um algoritmo para verificar se um vetor contém uma expressão aritmética correta em notação polonesa reversa.

2.13) Converter as seguintes expressões aritméticas em expressões totalmente parentisadas, em notação polonesa e em notação polonesa reversa:

a) (A + B - C) /E * (F / G - H * J)

b) A / B / C + D * E * F - G / H * J

c) A - B * (C - D/ (E + F - G / (H - J)))

2.14) Implementar o algoritmo de Identificação de aglomerados.

2.15) Implementar um algoritmo para o problema de Josephus: (colocam-se n n elementos em círculo, sorteia-se um inteiro p e, sucesivamente, conta-se no sentido horário eliminando-se a cada etapa aquele que recebeu a contagem p. O problema é determinar qual elemento resta após (n-1) rodadas de eliminação). Usar lista circular para a situação.

2.16) Fazer um algoritmo para, dada uma **lista encadeada**, transformá-la em duas outras, através da transferência, de forma alternada, dos nós da primeira para cada uma das duas outras.

2.17) Fazer um algoritmo para simular um conhecido jogo de paciência feito com um baralho: um baralho é embaralhado sem o Rei e a Dama de Copas; em seguida o Rei é colocado no início e a Dama do final. Durante o jogo retira-se do baralho, uma carta de cada vez e coloca-se esta carta numa pilha; se a terceira carta abaixo do topo da pilha contiver o mesmo número ou o mesmo naipe desse topo, as duas cartas do meio são retiradas. Se, ao se colocar a última carta na pilha, sobrarem apenas o Rei e a Dama de Copas, a paciência deu certo. Caso contrário, não.

2.18) Um **labirinto** é expresso numa matriz n x m, onde sequências de 0 indicam caminhos e X, paredes. A movimentação só pode ser feita horizontal e verticalmente. Na matriz, uma letra E indica a entrada do labirinto e S, a saída. Fazer um algoritmo, usando uma fila, para verificar se **existe um caminho** que leve de E a S.

Exemplo, onde há solução:

0	E	0	0	X	0
0	**0**	X	X	0	X
X	**0**	X	**0**	**0**	S
0	**0**	**0**	**0**	X	0

2.19) Fazer um algoritmo para, dadas as posições inicial (a, b) e final (c, d) de um cavalo, em um tabuleiro de xadrez n x n, n > 3, determinar **o número mínimo de movimentos** para o cavalo partir da posição inicial e chegar à posição final.

2.20) Fazer um algoritmo, que, sem uso de vetores, **inverte a orientação** de uma lista encadeada simples, com um nó cabeça. Por exemplo: a lista cab $\rightarrow$ A $\rightarrow$ B $\rightarrow$ C deve ser transformada em cab $\rightarrow$ C $\rightarrow$ B $\rightarrow$ A.

2.21)Fazer um algoritmo que usa uma fila para gerar os primeiros 1000 elementos da sequência denominada: "**Sequência de números feios**", que é uma sequência ordenada de números cujos fatores são apenas os números 2, 3 ou 5. O início dessa sequência é: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12...

Dicas: Colocar os elementos na fila de forma ordenada, gerando cada número a partir de números anteriores da fila. Evitar gerar números repetidos.

2.22) A **Pesquisa por Interpolação** é uma ampliação da Pesquisa Binária, tal que a posição onde se vai procurar uma chave não é no meio do intervalo de interesse, mas em um **ponto proporcional do intervalo** que leve em

conta o valor da chave procurada e os limites do intervalo. Considere a busca inicial da chave 20 no vetor abaixo.

1	2	3	4	5	6	7	8	9
10	**15**	**22**	**45**	**46**	**57**	**62**	**75**	**90**

A comparação inicial seria na posição 3 = (1 + 8 * (20 / 80)), ao invés da posição 5 que seria a escolhida para a Pesquisa Binária. Modificar o algoritmo de Pesquisa Binária para esta situação.

2.23) Um **"grid encadeado"** é uma lista onde cada nó tem 2 ponteiros (direita, baixo), sendo uma alternativa para representação de matrizes (ver exemplo de um grid 3 x 4); Os nós da última coluna têm ponteiro direito nulo e os da última linha, ponteiro baixo nulo. O início do grid é indicado por um ponteiro CAB. Escrever um algoritmo que transforma uma matriz (n x m) em um grid encadeado.

Ex: de um grid encadeado 3 x 4:

$$\begin{array}{ccccccc} CAB & \rightarrow & 4 & \rightarrow & 1 & \rightarrow & 3 & \rightarrow & 2 \\ \downarrow & & \downarrow & & \downarrow & & \downarrow \\ 7 & \rightarrow & 0 & \rightarrow & 5 & \rightarrow & 1 \\ \downarrow & & \downarrow & & \downarrow & & \downarrow \\ 2 & \rightarrow & 2 & \rightarrow & 8 & \rightarrow & 7 \end{array}$$

2.23) Escrever um algoritmo que soma os elementos do grid definido no exercício anterior.

3 Árvores

3.1 Motivação

Muitas vezes precisa-se de estruturas mais complexas do que as sequenciais, especialmente quando se trabalha com grandes volumes de dados. Uma classe de estruturas desse tipo são as árvores, que servem para representar estruturas hierarquizadas, onde um elemento pode apontar para vários outros. Uma das importantes aplicações desta estrutura será vista no próximo capítulo, as *árvores de busca*, adequadas para o problema de busca onde há grande volume de dados.

a) Definição formal de árvore

Uma **árvore enraizada** T, ou simplesmente **árvore**, é um conjunto finito de elementos denominados de **nós** ou **vértices**, tais que, ou:

-T é um conjunto vazio e então a árvore é dita vazia.

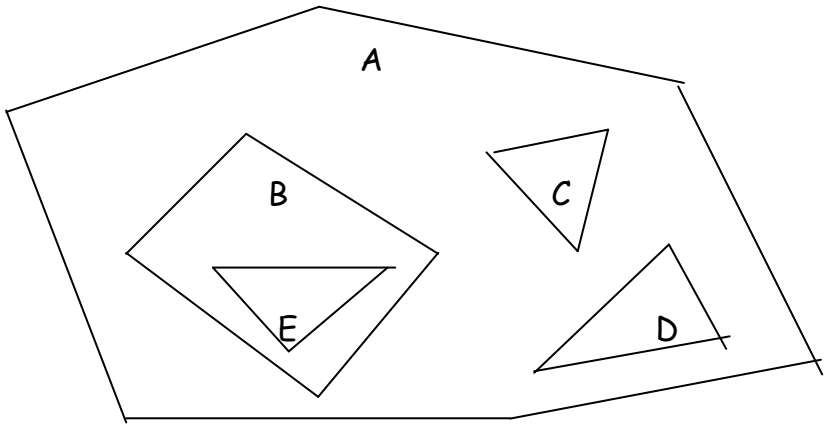
-existe um nó especial r, chamado **raiz** de T; os restantes constituem um único conjunto vazio ou são particionados em **m** $\geq$ **1** conjuntos disjuntos não vazios, as **subárvores** de r, ou simplesmente subárvores, cada qual por sua vez uma árvore.

Uma **floresta** é um conjunto de árvores.

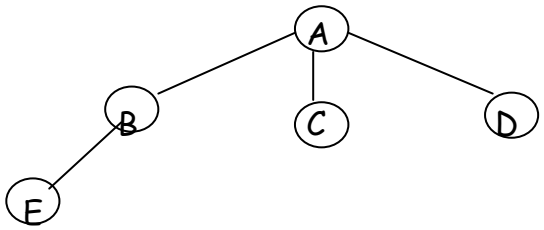
b) Representações de árvores

Árvores podem ser graficamente representadas de várias formas: **diagramas de inclusão**, **diagramas hierárquicos**, **diagramas de barra**, **parêntesis aninhados**, dentre outras.

Um exemplo de um **diagrama de inclusão**, que é usado também para representar conjuntos, seria:



A representação da mesma árvore por **diagrama hierárquico** seria:



Por **diagrama de barras**, teríamos:

A ----------

B ----------

E ----------

C ----------

D ----------

Finalmente, a representação por **parênteses aninhados**, seria:

(A (B (E)) (C) (D))

c) Definições básicas numa árvore:

Se v é o nó raiz de T, então as raízes das subárvores de T são denominadas ***filhos*** de v e v é dito ***pai*** dessas raízes. Nós com mesmo pai são denominados ***irmãos***. Nós que não tenham filhos são denominados ***folhas***. uma sequência de nós distintos v_1, v_2, ... v_k, com k nós, onde cada um é filho ou pai do anterior é dito um ***caminho*** na árvore. O ***comprimento*** do caminho é (k - 1). ***Nível*** de um nó w é o número de nós do caminho da raiz até o nó w. A ***altura*** de um nó w é o número de nós do maior caminho de w até um de seus descendentes. A ***altura*** da árvore é o maior dos níveis de seus nós.

Como exemplo, na árvore mostrada, temos:

- o nível do nó B é 2.
- o caminho de E a C tem comprimento 3.
- a altura da árvore é 3.

3.2 Árvores Binárias

a) Definições

O tipo de árvore mais importante utilizado é a árvore binária, cuja definição tem uma sutil diferença em relação à definição mais geral.

Uma ***árvore binária*** T é um conjunto finito de elementos denominados **nós ou vértices**, tal que ou:

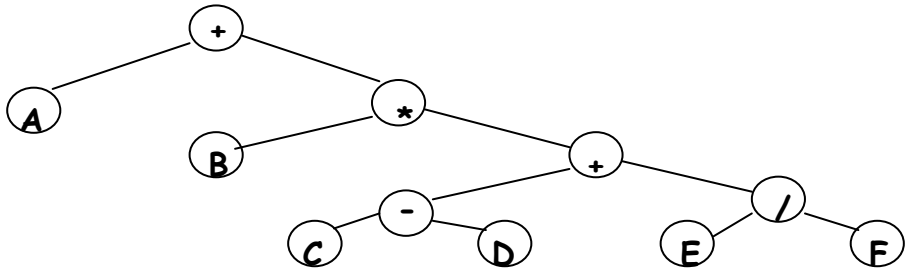
-T é um conjunto vazio e então **T é dita *vazia***

-existe um nó especial $\mathbf{r_T}$, chamado ***raiz*** de T, tal que os nós restantes podem ser divididos em dois subconjuntos disjuntos T_e e T_d, a ***subárvore esquerda*** e a ***subárvore direita*** de r_T, respectivamente, as quais, por sua vez, também são árvores binárias.

A sutil diferença das definições é que as subárvores podem ser vazias. Isso ajuda em alguns formalismos.

No exemplo a seguir, uma árvore binária é utilizada para representar a expressão aritmética:

A + B * (C - D + E / F)



Nessa representação, cada raiz contem um operador que opera sobre o resultado de cada subárvore. Os operandos estão todos nas folhas.

b) Conceitos complementares.

Os seguintes tipos de árvore têm papel importante em muitos algoritmos:

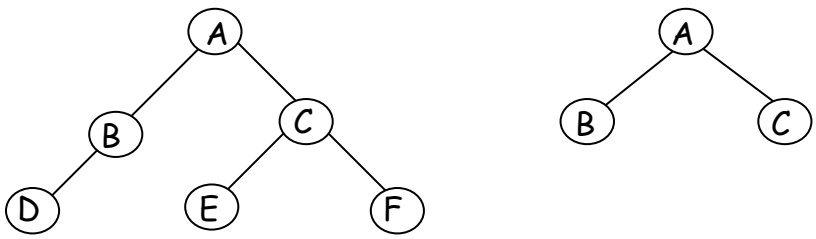
Árvore ***estritamente binária*** é uma árvore binária onde todo nó contem 0 ou 2 filhos. A árvore anterior enquadra-se neste caso.

Árvore binária completa é aquela que tem a seguinte propriedade: se v é um nó que tem alguma subárvore vazia, então v localiza-se no penúltimo ou no último nível da árvore.

Árvore binária cheia é aquela em que se v é um nó que tem alguma subárvore vazia então v localiza-se no último nível da árvore.

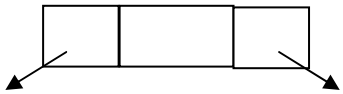
Nos exemplos a seguir, a árvore da esquerda é uma árvore completa, enquanto aquela da direita é uma árvore cheia:

Exemplos:



c) Representação computacional de árvores binárias.

O nó de uma árvore binária tem que conter 3 campos pelo menos, dois dos quais são apontadores para as subárvores esquerda e direita. A representação mais comum usa alocação encadeada onde um nó tem dois ponteiros, conforme a figura abaixo:



Em Pascal, a definição de uma árvore é feita assim:

```
Type  ponteiro = ↑ Arv;
      Arv = Record
              chave: string;
              le, ld: ponteiro;
            End;
```

d) Propriedadas importantes de uma Árvore Binária

d.1) O número de subárvores vazias em uma árvore binária com *n* nós é (*n + 1*).

d.2) A altura de uma árvore binária completa é
$h(T) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil$.

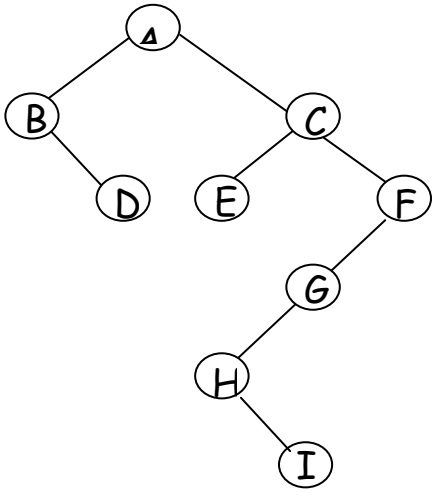
Definição: CI = ***Caminho interno*** de uma árvore binária = soma das distâncias da raiz até cada nó interno.

Definição: CE = ***Caminho externo*** de uma árvore binária = soma das distâncias da raiz até cada nó externo.

d.3) CE = CI + 2n + 1

e) Percursos em Árvores Binárias

Dentre as operações que podem ser executadas em árvores binárias, destacam-se aquelas que fornecem maneiras sistemáticas de visitar todos os nós da mesma. Serão apresentados os percursos em nível, pré-ordem, ordem simétrica, pós-ordem. Para exemplificar esses percursos, será utilizada a seguinte árvore binária:



e.1) Percurso em níveis

Neste tipo de percurso os nós são visitados de cima para baixo e da esquerda para a direita. Em relação ao exemplo, a ordem de visita dos nós seria:

A B C D E F G H I.

O algoritmo que será apresentado utiliza uma fila. A idéia é colocar a raiz da árvore binária na fila e, a partir daí, visitar o nó do início da fila, ao mesmo tempo em que se colocam seus filhos no final da mesma. O algoritmo é mostrado a seguir, onde a raiz da árvore é **T** e a fila usada é **Fil**.

```
Percurso_Nivel;
Início:
        Esvazia Fil;    Enfila (T);
        Enquanto (f <> 0):
                Desenfila (p);
                Visita (p);
                Enfila (p↑.le );   Enfila (p↑.ld );
        Fe;
Fim;
```

Esse algoritmo tem, evidentemente, complexidade O(n). Neste e nos próximos algoritmos, fica subentido que o enfilamento ou empilhamento de um nó só é feito quando o nó não é nulo.

Vejamos, para o exemplo anterior, a situação da fila **Fil** ao início do loop:

A				
B	**C**			
	C	**D**		
		D	**E**	
			E	**F**
				F
G				
H				
I				

A visita em níveis fornece uma maneira fácil de se determinar distâncias de caminhos na árvore e por isso tem importância específica em muitos problemas. Para a determinação de distâncias, basta se colocar, também na fila, as distâncias, que são calculadas de maneira bem simples.

As próximas formas de percurso da árvore seguem o procedimento sistemático de visitar a raiz, a subárvore esquerda e a subárvore direita. A diferenciação estará na prioridade dada à visita da raiz em relação às subárvores. A subárvore esquerda é visitada sempre antes da direita.

e.2) Percurso em pré-ordem.

Neste tipo de percurso visita-se prioritariamente a raiz, em seguida a subárvore da esquerda e, depois, a subárvore da direita. Para o exemplo adotado, a ordem de visita seria:

A B D C E F G H I.

O algoritmo que será apresentado utiliza uma pilha. A idéia é colocar a raiz na pilha e, a partir daí, visitar o nó do topo da pilha, ao mesmo tempo em que se empilham a raiz da subárvore direita e da subárvore esquerda. O algoritmo é mostrado a seguir, onde a raiz da árvore é **T** e a pilha usada é **Pil**.

Percurso_Pre_Ordem;

Início:

Esvazia Pil;

PUSH (T);

Enquanto (topo ≠ 0):

POP (p);

Visita (p);

PUSH (p↑.ld); PUSH (p↑.le);

Fe;

Fim;

Esse algoritmo também tem complexidade O(n).

Vejamos, para o exemplo anterior, a situação da pilha Pil ao início do loop:

A					
C	**B**				
C	**D**				
C					
F	**E**				
F					
G					
H					
I					

Esta forma de visita tem muitas aplicações. Uma delas é que, se a árvore representa uma expressão aritmética, a visita em pré-ordem fornece a notação polonesa para a expressão. Para o exemplo de árvore binária do ítem a, teríamos:

+ A * B + - C D / E F

e.3) Percurso em ordem simétrica.

Neste tipo de percurso visita-se prioritariamente a subárvore da esquerda, em seguida a raiz e, finalmente, a subárvore da direita. No exemplo, teríamos:

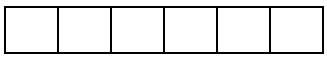
B D A E C H I G F.

O algoritmo que será apresentado também utiliza uma pilha. A idéia é colocar, sucessivamente, a raiz da subárvore esquerda na pilha até se chegar a um link nulo, quando então se visita o topo da pilha, repetindo-se o processo para a subárvore da direita. O algoritmo é mostrado a seguir, onde a raiz da árvore é **T** e a pilha usada é **Pil**.

```
Percurso_Ordem_Simétrica;
Início:
        Esvazia Pil;   p ← T;
        Enquanto ( p ≠ Nulo ) Ou ( topo  ≠ 0 ):
                Enquanto ( p ≠ Nulo ):
                        PUSH (p);        p ← p↑.le;
                Fe;
                POP (p);   Visita (p);   p ← p↑.ld;
        Fe;
Fim;
```

Esse algoritmo também tem complexidade O(n).
Vejamos, para o exemplo, a situação de Pil no início do segundo loop:

A					
A	**B**				
A					
A	**D**				
A					
C					
C	**E**				
C					
F					
F	**G**				
F	**G**	**H**			
F	**G**				
F	**G**	**I**			
F	**G**				
F					



Esta forma de visita tem muitas aplicações. Uma delas será vista no próximo capítulo que trata das árvores de busca.

e.4) Percurso em pós-ordem.

Neste tipo de percurso visita-se prioritariamente a subárvore da esquerda, em seguida a subárvore da direita e, por fim, a raiz.

Para o exemplo, a ordem de visita seria: **D B E I H G F C A.**

O algoritmo que será apresentado utiliza uma pilha e é um algoritmo de difícil entendimento, devendo ser verificado cuidadosamente. Cada nó é colocado na pilha duas vezes. A primeira quando se está empilhando ramos à esquerda da árvore; a segunda quando o empilhamento é pela direita. Um nó só é visitado quando ele é o topo da pilha e foi a segunda vez que entrou na mesma. O algoritmo é mostrado a seguir, onde a raiz da árvore é **T** e a pilha usada é **Pil**.

```
Percurso_Pos_Ordem;
Início:
        Esvazia Pil;   p ← T;
        Enquanto ( p ≠ Nulo ) Ou ( topo  ≠ 0 ):
                Enquanto ( p ≠ Nulo ):
                        PUSH (p, 1);
                        p ← p↑.le;
                Fe;
                Enquanto ( p = Nulo )  e  ( topo  ≠ 0 ):
                        POP (p, vez);
                        Se (vez = 1) e (p↑.ld ≠ Nulo) Então
                                PUSH ( p, 2 );
                                        p ← p↑.ld;
                        Senão
                                Visita ( p );
                                p ← Nulo;
                Fe;
        Fe;
Fim;
```

Esse algoritmo também tem complexidade O(n).

Vejamos, para o exemplo anterior, a situação da pilha Pil a cada mudança, indicando, também, o valor da vez:

A/1					
A/1	**B/1**				
A/1					
A/1	**B/2**				
A/1	**B/2**	**D/1**			
A/1	**B/2**				
A/1					
A/2					
A/2	**C/1**				
A/2	**C/1**	**E/1**			
A/2	**C/1**				
A/2					
A/2	**C/2**				
A/2	**C/2**	**F/1**			
A/2	**C/2**	**F/1**	**G/1**		
A/2	**C/2**	**F/1**	**G/1**	**H/1**	
A/2	**C/2**	**F/1**	**G/1**		
A/2	**C/2**	**F/1**	**G/1**	**H/2**	
A/2	**C/2**	**F/1**	**G/1**	**H/2**	**I/1**
A/2	**C/2**	**F/1**	**G/1**	**H/2**	
A/2	**C/2**	**F/1**	**G/1**		
A/2	**C/2**	**F/1**			
A/2	**C/2**				
A/2					

Esta forma de visita também tem aplicações. Uma delas é que, se a árvore representa uma expressão aritmética, a visita em pós-ordem fornece a notação polonesa reversa para a expressão. Para o exemplo de árvore binária do ítem a, teríamos:

A B C D - E F / + * +

3.3 Recursão e MergeSort

1.1 Conceitos básicos

Recursão é uma técnica de construção de algoritmos que consiste, basicamente, em se subdividir um problema em problemas menores de mesma natureza que o problema original e obter a solução como uma composição das soluções dos problemas menores. Para a solução dos problemas menores adota-se a mesma estratégia de subdivisão, até o nível em que o subproblema seja muito simples, quando então sua solução é exibida, geralmente com poucos passos.

A maneira de subdividir e de compor a solução pode ser diferente para cada problema. A seguir são mostrados dois exemplos clássicos de algoritmos recursivos: Fatorial e Fibonacci.

O cálculo de fatorial tem a seguinte recursão:

```
Fatorial (p):
        Se p = 0 Então
                Retornar 1
        Senão
                Retornar  p.Fatorial (p -1);
Fim;
```

Nesta recursão o único problema resolvido diretamente é o de 0! . Os demais são resolvidos a partir da solução de cada problema imediatamente menor.

Para a recursão da série de Fibonacci, temos:

```
Fibonacci(p):
        Se p ≤ 1 Então
                Retornar p
        Senão
                Retornar Fibonacci (p - 1) + Fibonacci (p - 2);
Fim;
```

Há quatro outras visões sobre a técnica de recursão, que certamente complementam essa visão inicial:

a) A técnica pode ser vista como uma maneira de se resolver problemas de "*trás para frente*", isto é: a solução enfatiza os passos finais para a solução do problema, supondo problemas menores resolvidos. No caso Fatorial, para se obter Fatorial(n), a idéia é multiplicar Fatorial (n -1) por n .

b) A técnica pode ser ainda imaginada como o *equivalente* algorítmico da *indução finita*, técnica de demonstração da matemática. Toda recursão consiste de duas etapas bem explícitas:

b.1) Quando o problema é "grande", ele é dividido em problema(s) menor(es), cuja(s) solução(ções) supõe-se factíveis. Tem-se então que explicitar como a(s) solução(ções) menor(es) tem que ser composta(s) para se chegar à solução final. Ou seja, a preocupação é com a composição de soluções.

b.2) Quando o problema é "pequeno *(infantil)* ", tem-se que mostrar sua solução.

c) A técnica é o *equivalente* procedural da formulação de *recorrências matemáticas* (funções recursivas), onde, de forma análoga ao ítem b), uma função é definida em duas partes. Na primeira, é dada uma fórmula fechada para um ou mais valores de n. Na segunda, a definição da função para n é feita a partir da mesma função aplicada a valores menores que n, para valores superiores aos da primeira parte.

d) Os procedimentos recursivos são aqueles que "chamam a sí mesmo". É claro, então, que a chamada a sí mesmo sempre se dá no contexto de buscar a solução de problemas menores, para compor a solução do maior. Além disso todo procedimento recursivo tem que ter também uma chamada externa.

Note que todo algoritmo recursivo tem uma solução não recursiva equivalente. Muitas vezes, entretanto, a expressão recursiva é mais natural e mais clara, como para o exemplo mostrado a seguir, que é a ordenação de um vetor através de Merges.

3.3.2 MergeSort

a) Idéia: sucessivas divisões do vetor em dois subvetores, ordenação dessas divisões e merge dos subvetores gerados. Notar que o procedimento Merge é uma pequena modificação de Merge1, mostrado no Capítulo 2, já que se faz merge de subvetores contíguos. O vetor tem tamanho *n*.

b) Algoritmo:

```
Inicio
        Sort(1, n);
Fim;

Procedimento Sort (E, D);
Inicio:
        Se (D > E)  Então
                i ← ⌊ (E + D)/2) ⌋;
                Sort (E, i);
                Sort (i+1, D)
                Merge (E, i, D)
        Fs;
Fim;

Procedimento Merge (E, i, D);
Inicio
        Merge1 ( V1, E, i, V1, i+1, D, V3);
        j ← E
        Para p de 1 até ( D - E + 1):
                V1 [ j] ← V3 [p];    j ← j + 1;
        Fp;
Fim;
```

c) Exemplo, mostrando os elementos envolvidos em comparações e trocas(estas indicadas em vermelho) e evidenciando apenas as operações de Merge.

Passo	1	2	3	4	5	6	7	8	9	10	11	12	
	E	**X**	**E**	**M**	**P**	**L**	**O**	**F**	**A**	**C**	**I**	**L**	
1	*E*	*X*											Merge (1 ,1 ,2)
2	*E*	*E*	*X*										Merge (1, 2, 2)
3				*M*	*P*								Merge (4, 4, 5)
4				*L*	*M*	*P*							Merge (4, 5, 6)
5	*E*	*E*	*L*	*M*	*P*	*X*							Merge (1, 3, 6)
6							*F*	*O*					Merge (7, 7, 8)
7							*A*	*F*	*O*				Merge (7, 8, 9)
8										*C*	*I*		Merge (10, 10, 11)

9										*C*	*I*	*L*	Merge (10, 11, 12)
10							*A*	*C*	*F*	*I*	*L*	*O*	Merge (7, 9, 12)
11	*A*	*C*	*E*	*E*	*F*	*I*	*L*	*L*	*M*	*O*	*P*	*X*	Merge (1. 6. 12)
	A	**C**	**E**	**E**	**F**	**I**	**L**	**L**	**M**	**O**	**P**	**X**	Situação Final

d) Análise do Algoritmo

d.1) Complexidade: Melhor caso = Pior Caso = Caso Médio: NC =~N* log(N) = O(N log N)

d.2) Estabilidade: Algoritmo estável

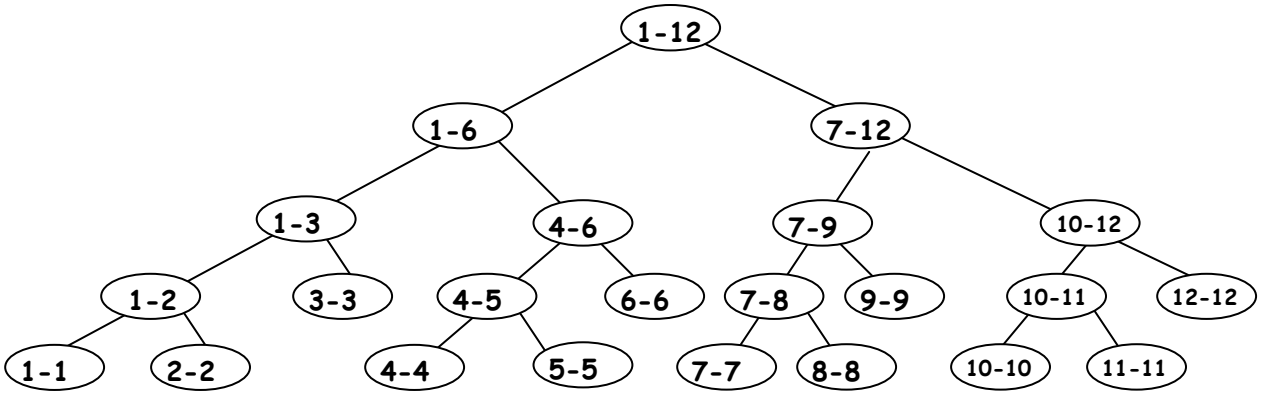
d.3) Situações Especiais: Algoritmo de uso amplo, não tão rápido quando o QuickSort.

d.4) Memória necessária: vetor intermediário de tamanho N e pilha para recursão.

e) Observações:

e.1) Número de comparações: 44 ; Número de trocas: 44

e.2) No algoritmo foram mostradas apenas as etapas de Merge. Entretanto, para uma compreensão maior é necessário se verificar a árvore de recursão do algoritmo. Essa árvore é uma árvore completa, com 2*n-1 elementos e altura $\log_2 n$. No caso particular a árvore é a seguinte:



3.4 Percursos Recursivos em Árvores Binárias

Os três últimos algoritmos de percurso em árvore binária mostrados no ítem 3.2 podem ser reformulados em termos recursivos, de forma surpreendentemente simples. Pode-se verificar também que a única diferença entre os algoritmos é a colocação da instrução **Visitar (t)**. Numa comparação com a versão não recursiva, estes algoritmos são bastante mais claros, especialmente na visita em pós-ordem.

a) Percurso em pré-ordem.

A formulação recursiva é a seguinte:

"Se a árvore é nula, nada é feito; senão, visitar a raiz da árvore, em seguida visitar recursivamente a subárvore esquerda e depois a subárvore direita".

Essa formulação leva ao seguinte algoritmo:

```
Visita_pre_ordem (t);
Início:
        Se ( t ≠ Nil ) Então
                Visitar (t);
                Visita_pre_ordem (t↑.le);
                Visita_pre_ordem (t↑.ld);
Fim;
```

b) Percurso em ordem simétrica.

A formulação recursiva é a seguinte:

"Se a árvore é nula, nada é feito; senão, visitar recursivamente a subárvore da esquerda, em seguida visitar a raiz e, depois, visitar recursivamente a subárvore direita".

Essa formulação leva ao seguinte algoritmo:

```
Visita_ordem_simetrica (t);
Início:
        Se ( t ≠ Nil ) Então
                Visita_ordem_simetrica (t↑.le);
                Visitar (t);
                Visita_ordem_simetrica (t↑.ld);
Fim;
```

c) Percurso em pós ordem.

Finalmente, para este caso, a formulação recursiva é a seguinte:

"Se a árvore é nula, nada é feito; senão, visitar recursivamente a subárvore da esquerda; depois, visitar recursivamente a subárvore direita e, por fim, visitar a raiz".

Essa formulação leva ao seguinte algoritmo:

```
Visita_pos_ordem (t);
Início:
        Se ( t ≠ Nil ) Então
                Visita_pos_ordem (t↑.le);
                Visita_pos_ordem (t↑.ld);
                Visitar (t);
Fim;
```

3.5 Complementos

3.6 Exercícios

3.1) Demonstrar as propriedades 1 2 e 3, definidas no ítem 3.2.

3.2) Escrever um algoritmo para listar uma árvore, por níveis.

3.3) Escrever um algoritmo para a criação de uma árvore binária.

3.4) Demonstrar que a ADPB é uma árvore completa.

3.5) Desenhar a ADPB para n = 30 + dv da matricula.

3.6) Calcular o número de chaves no último nível da ADPB.

3.7) Calcular o número de árvores binárias completas distintas p/ n nós.

3.8) Uma **expressão aritmética** pode ser armazenada em uma **árvore estritamente binária**, onde as folhas contêm operandos e os nós intermediários, operadores. Cada operador opera sobre as duas subárvores. Modificar o algoritmo não recursivo de percurso em pós-ordem numa Árvore Binária de Busca, usando pilhas, para que a expressão seja calculada.

3.9) Fazer um algoritmo, **usando uma fila**, para listar todos os nós de uma árvore binária que não são folhas.

3.10) Fazer um algoritmo recursivo para listar, de forma inversa, uma lista simplesmente encadeada, com um nó cabeça.

3.11) Implementar um algoritmo recursivo para criar a ADPB.

3.12) Escrever um algoritmo recursivo para construir uma árvore binária completa para n chaves.

3.13) Escrever um algoritmo recursivo para construir uma árvore binária cheia de altura h.

3.14) Escrever um algoritmo para desenhar uma árvore binária usando o percurso em ordem anti-simétrica (subárvore esquerda, raiz, subárvore direita).

3.15) A altura de um nó em uma árvore binária é a altura da subárvore com raiz nesse nó. O ***percurso em altura*** numa árvore binária é aquele em que os nós são dispostos em ordem não decrescentes de suas alturas. Descrever um algoritmo para efetuar um percurso em altura em uma árvore binária.

3.16) Fazer um algoritmo **recursivo** para determinar o número de descendentes de cada nó, em uma árvore binária.

3.17) Fazer um algoritmo **recursivo** para determinar a altura de uma árvore binária.

3.18) Fazer um algoritmo para determinar os dois nós mais distantes em uma árvore binária. Se houver mais de um par qualquer um pode ser escolhido.

3.19) Implementar os algoritmos de percurso numa árvore binária.

3.20) Implementar os algoritmos de MergeSort.

3.21) Mostrar os passos da ordenação pelos algoritmos de MergeSort, para o CRSTRING (= string de 12 letras formado pelas 12 letras iniciais do nome). Contar o número de comparações executadas.

3.22) Argumentar que o algoritmo MergeSort é estável.

3.23) Desenhar as árvores de recursão para os percursos recursivos na árvore binária usada nos exemplos.

3.24) Definir a recursão e escrever algoritmos recursivos para resolver os seguintes problemas:

a) soma dos elementos de um conjunto
b) média dos elementos de um conjunto
c) determinação do menor elemento de um conjunto.

3.25) Definir a recursão e escrever algoritmos recursivos para resolver os seguintes problemas:

a) verificar se dada árvore binária é uma árvore estritamente binária.
b) verificar se dada árvore binária está balanceada, segundo a definição de uma árvore AVL.
c) Verificar se dada árvore binária está equilibrada. Uma árvore está equilibrada se a soma dos pesos das folhas da subárvore direita é igual à soma dos pesos das folhas da subárvore direita, para cada nó da árvore.

4 Árvores de Busca

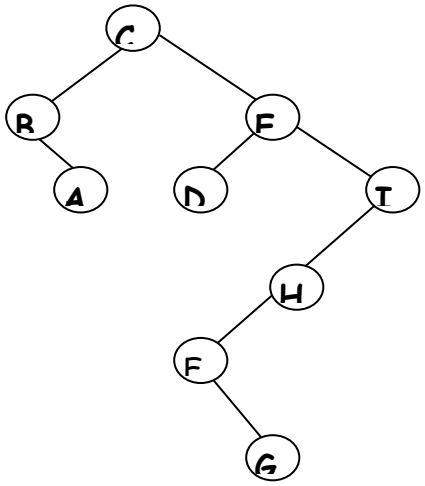
Árvores de busca são famílias de árvores utilizadas para armazenamento e busca de dados. O objetivo dessas estruturas é obter uma performance média bem melhor que aquela obtida com listas sequenciais. O mecanismo básico utilizado nas árvores de busca é o de, numa procura, comparar o argumento de busca com determinada chave de determinado nó da árvore. Se a chave for maior, a busca prossegue pela subárvore da direita; se for menor, pela subárvore da esquerda. Serão estudadas as árvores binárias de busca e um caso particular das mesmas, as árvores AVL. Serão também estudadas as árvores B, principal estrutura presente nos Bancos de Dados.

4.1 Árvores Binárias de Busca

4.1.1 Definições

Definição: **Árvore Binária de Busca** (**ABB**) é uma árvore binária onde as chaves dos nós da subárvore direita de cada nó são maiores que as chaves desse nó e as chaves da subárvore esquerda são menores que a mesma.

Uma árvore binária que não é ABB é aquela usada nos exemplos do ítem 3.2, pois a chave A está na raiz e existem chaves maiores que A na subárvore esquerda. Um exemplo de ABB é mostrado a seguir:



Observação: A definição de ABB pode ser estendida adotando-se alternativamente um dos critérios maior ou igual/menor ou igual para as chaves da subárvore direita/esquerda, respectivamente.

4.1.2 Busca, Inserção e Deleção em Árvores Binárias de Busca

Serão apresentados, a seguir, algoritmos para busca e busca e inserção (em caso de insucesso na busca) e deleção numa ABB. Esses algoritmos utilizam uma convenção especial: é criado um nó especial (z), tal que todo link nulo na árvore é transformado num apontador para esse nó. Essa convenção tem várias utilidades. Uma é a de simplificar os algoritmos.

a) Algoritmo básico de busca

```
Busca( k, p);
Inicio:
        p ← T;   z↑.chave ← k;
        Enquanto ( p↑.chave ≠ k ):
                Se  ( k < p↑.chave ) Então
                        p ← p↑.le
                Senão
                        p ← p↑.ld;
        Fe;
        Se (p = z) Então
                p = Nulo;
Fim;
```

b) Observações

b.1) Quando este algoritmo for usado, tem que ser feita a seguinte inicialização:

```
Inicializacao;
Inicio:
        Alocar(z);   z↑.le ← z;    z↑.ld ← z;    T ← z;
Fim;
```

b.2) Se for ser feita busca e inserção, muda-se ligeiramente o algoritmo. Basicamente tem-se que guardar, durante a busca, o endereço do último nó visitado:

```
Busca_Insercao( k, p);
    Inicio:
        f ← T;   p ← T;  z↑.chave ← k;
        Enquanto ( p↑.chave ≠ k ):
            f ← p;
            Se  ( k < p↑.chave ) Então
                    p ← p↑.le
            Senão
                    p ← p↑.ld;
        Fe;
        Se (p = z) Então
            Alocar(p);   p↑.le ← z;    p↑.ld ← z;    p↑.chave ← k;
            Se ( p = z ) Então                T ← p
            Senão  Se ( k < p↑.le) Então      f↑.le  ← p;
            Senão                             f↑.ld  ← p;
    Fim;
```

c) Análise da Busca e Inserção

c.1) Complexidade da Busca:

c.1) Melhor caso médio: NC = $\log_2 n$ (árvore completa) = $O(\log_2 n)$

c.2) Pior caso médio: NC = n/2 (árvore degenerada)

c.3) Caso médio de buscas bem sucedidas, n entradas distintas, probab. =s

NCmed = 1.4 $\log_2 n$ = $O(\log_2 n)$

c.2) Complexidade da Inserção: TEmed = $O(\log_2 n)$

c.3) Buscas Especiais:

Próxima chave: TE = cte = $O(1)$ (se a árvore for costurada)

= $O(\log_2 n)$ (se a árvore não for costurada)

Faixa: TE = $O(\log_2 n)$

Prefixo: TE = $O(\log_2 n)$

c.4) Memória adicional: Memória para os links da árvore.

d) Deleção em ABB

A deleção em ABBs é acentuadamente mais complicada que a inserção, para que se mantenha as propriedades estruturais da árvore. A seguir serão definidos alguns conceitos que leva a um algoritmo recursivo,

Def: Dado um nó v não folha de uma ABB, diz-se que w é um ***nó colado*** de v se w é *descendente* de v e w é *antecessor* ou *sucessor* de v.

Se v tem subárvore esquerda não nula, então o antecessor de v pertence à essa subárvore. Similarmente, se v possui subárvore direita não nula, então o sucessor de v pertence a essa subárvore.

Como exemplos, na árvore adotada, C tem ddois nós colados; o antecessor de **C** (**B**) está na subárvore esquerda e o sucessor (**D**), na direita. O nó **F** só tem um nó colado à direita (**G**), seu sucessor. O nó **D** não tem nós colados.

Algoritmo para se determinar nós colados:

```
No_colado(p):
Início:
        Se (p↑.le  ≠ Nil)  Então
                q ← p↑.le;
                Enquanto ( q↑.ld ≠ Nil):
                        q ← q↑.ld;
                Fe;
                Imprimir ('Nó colado antecessor de p é q');
        Se (p↑.ld  ≠ Nil)  Então
                q ← p↑.ld;
                Enquanto ( q↑.le ≠ Nil):
                        q ← q↑.le;
                Fe;
                Imprimir ('Nó colado sucessor de p  é q');
Fim;
```

A definição anterior permite formular o seguinte algoritmo recursivo para a Deleção em uma ABB:

Deleção (p);

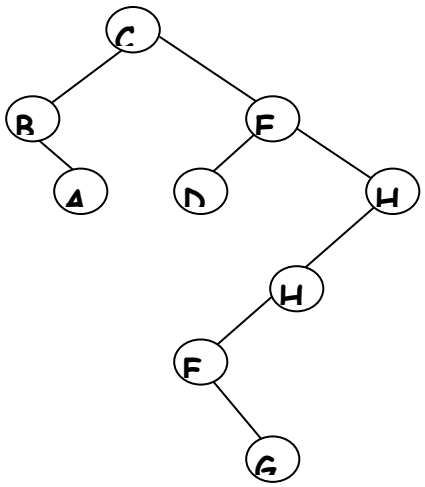
```
Início:
        Se p é folha Então
                Seja q o pai de p;
                Se (q↑.le = p)  Então  q↑.le ← Nil
                Senão                  q↑.ld ← Nil
                Desalocar(p);
        Senão
                Seja q um nó colado de p;
                p↑.chave ← q↑.chave;
                Deleção (q);
Fim;
```

e) Análise da Deleção

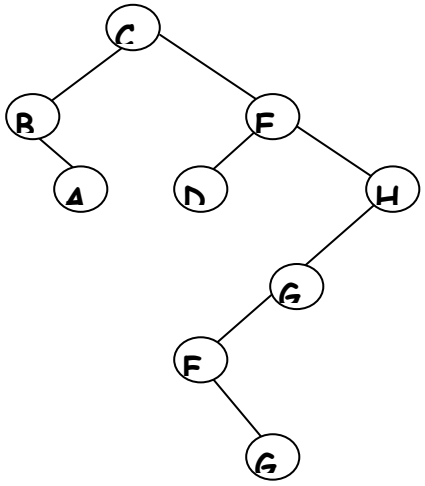
e.1) Complexidade da Deleção:

- e.1) Melhor caso: Tdel = Deleção de folha = $O(1)$
- e.2) Pior caso: Tdel = Deleção de folha em árvore degenerada = $O(n)$
- e.3) Caso médio: Tdel = Deleção de folha em árvore média = $O(\log_2 n)$

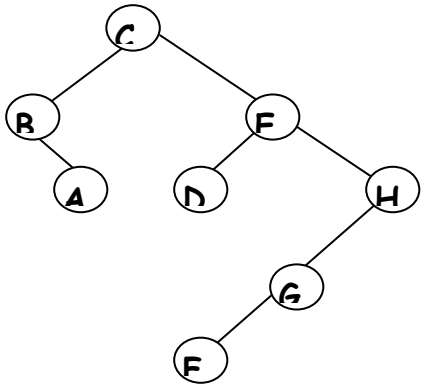
f) Exemplo: Se, na árvore exemplo, quisermos eliminar a chave **I**, teremos, sucessivamente:



A chave **I** foi substituída pela chave antecessora colada **H** e, agora vai-se eliminar o nó original que continha **H**.



A chave **G** foi substituída pela chave antecessora colada **G** e, agora vai-se eliminar o nó original que continha **G**. Como esse nó é folha, ele é eliminado.



4.1.3 Propriedades de uma ABB

Existem propriedades importantes na ABB, sendo a principal delas uma propriedade que relaciona o percurso em ordem simétrica com esse tipo de árvore.

Propriedade 1: O percurso em ordem simétrica de uma ABB fornece as chaves ordenadas.

Essa propriedade decorre diretamente da definição da árvore e é base para um método de Ordenação (TreeSort) que consiste em criar uma ABB com as chaves e depois percorrer a ABB em ordem simétrica. A análise deste método fica como exercício.

Na árvore usada como exemplo, esse percurso fornece:

A B C D E F G H I

Outra propriedade importante (na verdade, mais um resumo da análise do algoritmo feita) diz respeito à altura da árvore:

Propriedade 2: A altura de uma ABB com n chaves varia entre $\lfloor \log_2 n \rfloor + 1$ e n, conforme já comentado.

Esse foi um resultado mostrado na análise do algoritmo de busca. A melhor árvore que pode ser obtida é uma árvore com o mesmo parâmetro de uma árvore completa. Essa árvore corresponde à Pesquisa Binária em um vetor. Já a pior árvore corresponde a uma árvore de altura n, correspondendo à Busca sequencial. Entretanto, o caso médio deste tipo de estrutura é razoável (árvore com altura = $1.4 \log_2 n$).

A possibilidade de termos árvores degeneradas é que levará à introdução de árvores balanceadas (árvores AVL e árvores B), nos próximos ítens.

Uma terceira propriedade diz respeito à ordem de entrada das chaves.

Propriedade 3: A estrutura de uma ABB depende da ordem de entrada das chaves.

A nomeação desta propriedade serve para chamar a atenção para o fato explicitado. Se, por exemplo, as chaves forem inseridas em ordem ascendente ou descendente, a árvore resultante é degenerada.

4.1.4 Enumeração de Árvores Binárias de Busca

Um importante estudo para aumentar a compreensão das ABBs é o da enumeração de árvores de Busca, que é aquele de determinar quantas árvores ABB distintas existem para n chaves dadas. Este problema é resolvido através de uma recorrência clássica.

Seja **T(n)** o número de ABBs distintas para n chaves

Esse número é calculado por recorrência, usando-se a idéia de que o número de árvores distintas é obtido pelo somatório de n situações distintas, onde em cada uma delas uma das n chaves está na raiz.

$T(0) = 1;\ T(1) = 1;$

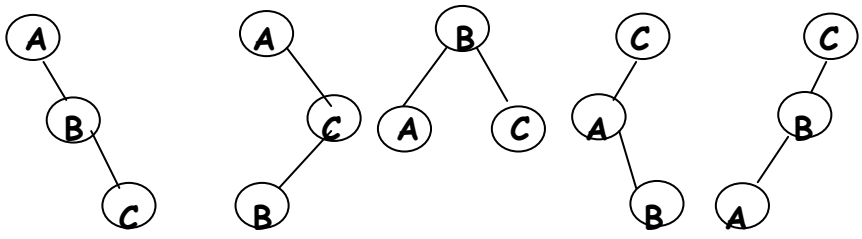
$T(n) = T(0).T(n-1) + T(1).T(n-2) + \ldots + T(n-1).T(0)$

$= \Sigma\ T_{i-1} \,.\, T_{n-i}, \quad 1 \leq i \leq n$

A solução de tal recorrência (resolvida por equações diferenciais), leva a uma expressão famosa denominada **número de Catalão**:

T(n) = Comb(2n,n)/(n+1)

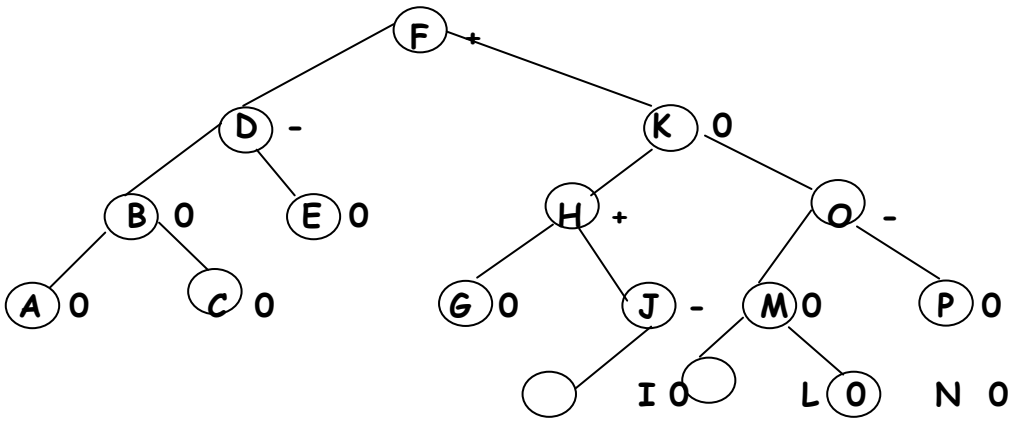
Exemplo: Para n = 3, temos T(3) = Comb(6,3)/4 = ((6.5.4)/(1.2.3))/4 = 5
Se supusermos as chaves A, B e C, teremos as seguintes ABBs:



4.2 Árvores AVL

4.2.1 Definições

As Árvores AVL foram inventadas pelos russos **A**del'son-**V**els'ii e E.M. **L**andis e são árvores binárias de busca nas quais a altura da subárvore esquerda de cada nó nunca difere de $\pm$ 1 da altura da correspondente subárvore direita. A idéia é evitar árvores binárias de busca degeneradas e, assim, melhorar muito o pior caso de busca nessas árvores. A árvore a seguir é um exemplo de árvore AVL:



Convenção usada:

(-) a altura da subárvore esquerda > da direita
(0) a altura da subárvore esquerda = da direita
(+) a altura da subárvore esquerda < da direita

4.2.2 Processo de Rebalanceamento da Árvore AVL, na inserção

Para que a árvore esteja sempre balanceada, a idéia é consertar (rebalancear) a árvore a cada inserção de chave que a desbalanceie. Os inventores da árvore AVL descobriram que há apenas dois casos de desbalanceamento. Nos dois casos, o processo de rebalanceamento ocorre da seguinte forma:

a) Insere-se uma chave nova.

b) Percorre-se a árvore a partir do ponto de inserção, em direção à raiz, atualizando os fatores de balanceamento (Atenção: apenas nós ascendentes do nó inserido têm seu fator de balanceamento alterado) até o ponto necessário que pode ser: ou a raiz, ou um ponto de desbalanceamento, ou um ponto cujo fator de balanceamento passe para zero).

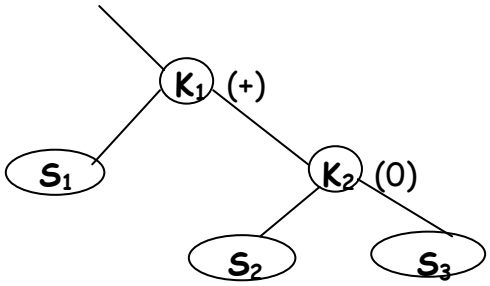
c) Se algum ponto desse caminho se desbalancear, então deve ser feito o rebalanceamento.

4.2.2.1 Rebalanceamento - Caso 1: Rotação Simples

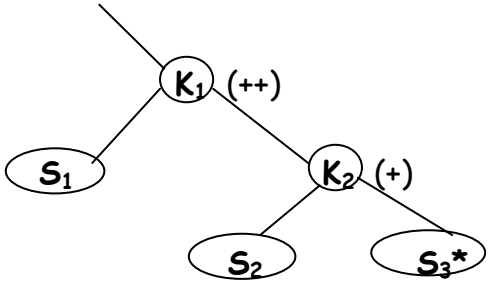
Esse caso ocorre segundo o esquema mostrado a seguir (há também um caso simétrico de desbalanceamento na esquerda).

a)Situação inicial:

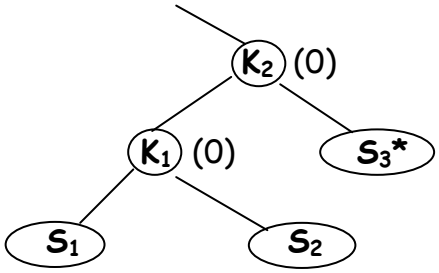
Existe um nó (K_1), cujo fator de balanceamento é (+). Este nó tem um filho à direita (K_2), neutro. As subárvores S_1, S_2 e S_3 são todas subárvores de altura h. A subárvore total tem altura (h + 2).



b) Situação de desbalanceamento: Inseriu-se uma chave na subárvore S_3, que passa a ser S_3*, de tal forma que sua altura passou para (h + 1). Portanto o fator de balanceamento do nó K_2 passa para + e o nó K_1 torna-se desbalanceado (fator de balanceamento ++).



c) Solução: A solução é fazer uma rotação simples em torno de K_2, que passa a ser a nova raiz da subárvore.



d) |Observações

Observar que esta operação deve ser feita no nível mais baixo possível da árvore. Além disso, ela só é possível porque preserva os seguintes aspectos:

1) Mantém a mesma altura de antes da inserção (h + 2)

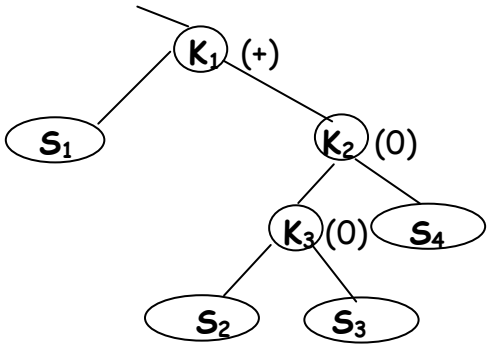
2) Mantém a ordem de busca das chaves.

4.2.2.2 Desbalanceamento - Caso 2: Rotação Dupla

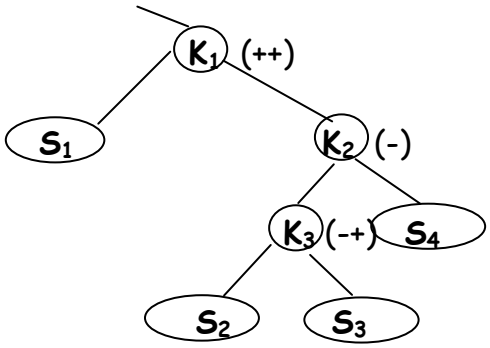
Esse caso ocorre segundo o esquema mostrado a seguir (há também um caso simétrico de desbalanceamento na esquerda).

a) Situação inicial:

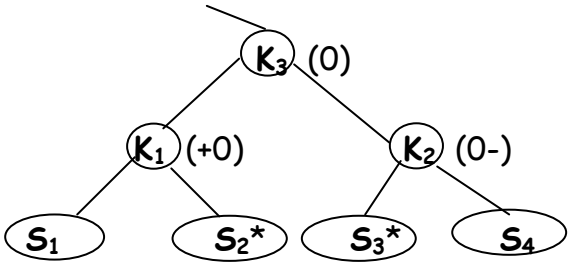
Existe um nó (K_1), cujo fator de balanceamento é (+). Este nó tem um filho à direita (K_2), neutro. Este, por sua vez tem um filho à esquerda (K_3), neutro. As subárvores S_1, S_4 têm altura h, enquanto que as subárvores S_2 e S_3, altura (h-1). A subárvore total tem altura (h + 2).



g) **Situação de desbalanceamento**: Inseriu-se uma chave em uma das subárvores S_2 ou S_3, que passam a ser S_2* e S_3*. mudando o fator de balanceamento de K_3 para - ou +, respectivamente. Além disso, a altura da subárvore esquerda de K_2 passou para (h + 1). Portanto o fator de balanceamento do nó K_2 passa para + e o nó K_1 se desbalanceia (fator de balanceamento ++).



c) Solução: A solução é fazer uma rotação dupla em torno de K_3, inicialmente para a direita e depois para a esquerda. K_3 passa a ser a nova raiz da subárvore.



d) Observações

Observar que esta operação, tal como no caso anterior, deve ser feita no ponto mais baixo possível da árvore. Além disso, ela só é possível porque preserva os seguintes aspectos:

1) Mantém a mesma altura de antes da inserção (h + 2)

2) Mantém a ordem de busca das chaves.

3) Há um caso especial, que se balanceia como o caso 2 que é a situação em que S1, S2, S3 e S4 são nulas e o nó inserido é o próprio nó K_3.

4) Há dois casos simétricos aos casos 1 e 2, do lado esquerdo da árvore, cujo rebalanceamento é feito de forma análoga aos mesmos.

4.2.4 Algoritmo de Busca e Inserção em Árvore AVL

O algoritmo apresentado a seguir é um algoritmo recursivo para Busca e Inserção. Ele utiliza as declarações a seguir:

```
Type  ref = ^no;
      no = record
             chave,cont: integer;
             le,ld:ref;
             bal: -1..+1
      end;

Var   raiz: ref; k: integer;
      h:boolean;
```

No algoritmo apresentado, a variável global **h** tem o importante papel de indicar que a subárvore examinada já está balanceada. Sempre que, na volta da recursão esta variável estiver "Verdadeira", está indicando que a subárvore respectiva aumentou de altura. Esse aumento pode indicar ou não necessidade de rebalanceamento. Por exemplo, se o nó tivesse fator de balanceamento (-1) e houvesse uma indicação de que o ramo direito aumentou de altura, o nó passaria a estar mais balanceado; não há rebalanceamento, somente ajustes de fatores.

O algoritmo está construído com códigos distintos para rebalanceamento à direita e à esquerda, usando os procedimentos RotSimpEsq, RotDupEsq, RotSimDir, RotDupDir, indicando rotações sinmples e dupla à esquerda e à direita, respectivamente.

```
Procedure Busca_insercao (x:integer; Var p:ref; Var h: boolean);
Var p1, p2: ref;
Begin
   If (p = Nil) Then Begin {insercao}
        h := true;     New(p);
        With p^ do Begin  chave:= x; cont:=1; le:=Nil; ld:= Nil; bal:= 0;  End
   End
   Else If (x < p^.chave) Then Begin
        Busca_Insercao (x,  p^.le, h);
        If h Then      {o ramo esquerdo aumentou de altura}
           Case p^.bal of
                1: Begin p^.bal := 0; h:= false End;
                0: p^.bal := -1;
                -1: Begin {rebalanceamento }
                        p1 := p^.le;
                        If  (p1^.bal = -1) Then
                           RotSimpEsq (p, p1)      { rot. simples à esquerda  }
                        Else
                           RotDupEsq (p, p1);      { rot. dupla à esquerda }
                        p^.bal := 0; h := false;
                    End
            End
   End
```

```
    Else If (x > p^.chave) Then Begin
       Busca_Insercao (x,   p^.ld,  h);
       If h Then      {o ramo direito aumentou de altura}
         Case p^.bal of
             -1: Begin p^.bal := 0; h := false End;
             0: p^.bal := +1;
             1: Begin    {Rebalanceamento}
                    p1 := p^.ld;
                    If p1^.bal = +1 Then
                           RotSimpDir(p, p1); { rot. simples à direita  }
                    Else
                           RotDupDir(p, p1);   { rot. dupla à direita  }
                    p^.bal := 0; h := false
                End
          End
      End
      Else  Begin
             p^.cont := p^.cont + 1; h := false
      End
End;    {Busca_Insercao}
```

Os diversos procedimentos estão descritos a seguir:

```
Procedure RotSimpEsq (Var p, p1:ref);
Begin
      p^.le := p1^.ld;
      p1^.ld := p;
      p^.bal := 0;
      p := p1;
End
Procedure RotDupEsq (Var  p, p1: ref);
Begin
      p2:= p1^.ld; p1^.ld := p2^.le; p2^.le := p1; p^.le := p2^.ld;p2^.ld := p;
      If p2^.bal = -1 Then p^.bal := +1
      Else p^.bal := 0;
      If p2^.bal = +1 Then p1^.bal := -1
      Else p1^.bal := 0;
      p := p2;
End;
```

```
Procedure RotSimpDir (Var p, p1: ref);
Begin
        p^.ld:= p1^.le;
        p1^.le:= p;
        p^.bal:= 0;
        p:=p1;
End;

Procedure RotDupDir (Var p, p1: ref);
Begin
        p2:= p1^.le; p1^.le := p2^.ld;p2^.ld := p1; p^.ld := p2^.le; p2^.le := p;
        If p2^.bal= +1 Then p^.bal:= -1
        Else p^.bal :=0;
        If p2^.bal= -1 Then p1^.bal:=+1
        Else p1^.bal:=0;
        p := p2;
End;
```

a) Análise do algoritmo

a) Complexidade da busca:

a.1) Melhor caso médio: NC = $O(\log_2 n)$ (árvore cheia)

a.2) Pior caso médio: NC = 1,44 log2(n) (árvore mínima)

a.3) Caso médio:

a.3.1) Buscas bem sucedidas, n entradas distintas, probab. =s

NCmed = $O(\log_2 n)$

b) Complexidade de atualizações:

Inserção: TEmed = $O(\log_2 n)$

Deleção: TEmed = $O(\log_2 n)$

c) Buscas Especiais:

c.1)Próxima chave: TE = $O(1)$ (p/ árvore costurada)
= $O(\log_2 n)$ (p/ árvore não costurada)

c.2) Faixa: TE = $O(\log_2 n).m$ (m = tamanho da faixa)

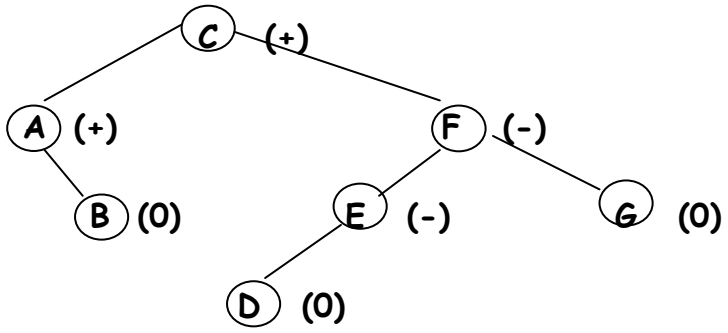
c.3) Prefixo: TE = $O(\log_2 n)$

d) Memória: Memória para os links da árvore

4.2.5 Árvores AVL Mínimas

Def: Uma **árvore AVL mínima** é uma árvore AVL de altura máxima, para dado número de chaves.

Exemplo: Para n = 7, uma possível árvore AVL mínima é



Queremos avaliar a altura de uma AVL mínima com n chaves. Entretanto é mais fácil iniciar verificando um problema dual: qual o número mínimo de chaves em uma AVL com dada altura h.

Seja T(h) o número mínimo de nós da árvore AVL de altura h.

Pode-se estabelecer a recorrência descrita a seguir, baseando-se na constatação de que uma árvore desse tipo tem uma raiz e, como subárvores, árvores de mesma natureza, mas uma delas diferindo da outra na altura em 1. Ou seja: uma das subárvores tem altura (h - 1) e a outra (h - 2). Além disso, as árvores elementares com altura 0 e 1 nós têm T(0) = 0 e T(1) = 1, respectivamente. Portanto:

$T(h) = 1 + T(h - 2) + T(h - 1),\ h > 1$
$T(h) = h,\ h \leq 1$

Para exemplificar a recorrência, vejamos T(0)...T(4)

$T(0) = 0$
$T(1) = 1$
$T(2) = 1 + T(0) + T(1) \quad = 1 + 0 + 1 \quad = 2$
$T(3) = 1 + T(1) + T(2) \quad = 1 + 1 + 2 \quad = 4$
$T(4) = 1 + T(2) + T(3) \quad = 1 + 2 + 4 \quad = 7$

Analogamente, calculamos:
T(5) = 12
T(6) = 20
T(7) = 33

A solução desta recorrência pode ser feita de duas formas:

1) Através de equações diferenciais (diferenças finitas)

2) Por indução finita, baseando-se na observação de que essa recorrência tem a aparência da recorrência de Fibonacci.

Temos a comparação:

h	0	1	2	3	4	5	6	7	8	9	10
Fib(h)	0	1	1	2	3	5	8	13	21	34	55
T(h)	0	1	2	4	7	12	20	33	54	88	133

Parece haver a seguinte relação: **T(h) = Fib(h + 2) - 1**;

Lema: T(h) = Fib(h+2)-1;

Prova (por indução em h):

a) Vemos que para casos particulares a relação é válida pois

T(0) = 0 e Fib(0 + 2) - 1 = Fib(2) - 1 = 1 - 1 = 0 = T(0)

T(1) = 1 e Fib(1 + 2) - 1 = Fib(3) - 1 = 2 - 1 = 1 = T(1)

b) Suponhamos, agora que a relação seja válida para todos os valores < h

Então:

T(h) = 1 + T(h - 2) + T(h - 1) (definição)

= 1 + (Fib (h - 2 + 2) -1) + (Fib(h - 1 + 2) - 1) (hipótese)

= (Fib(h) + Fib(h + 1)) - 1 (agrupamento)

= (Fib(h + 2)) - 1 = (definição)

= Fib(h + 2) - 1 ,

permanecendo válida a relação p/ h. Portanto a fórmula é válida.

A relação acima pode ser melhor explicitada, como mostrado a seguir.

Sabemos (De Moivre) que

Fib(h) = $(\Phi^h - \Theta^h)/(Sqrt(5) * 2^h)$, onde

Φ = (1+Sqrt(5)) , Θ = (1-Sqrt(5)) (Sqrt(n) = raiz quadrada de n)

Substituindo:

T(h) = $(\Phi^{h+2} - \Theta^{h+2})/(Sqrt(5) * 2^{h+2}) - 1$

$\cong \lfloor \Phi^{h+2}/(Sqrt(5) * 2^{h+2}) - 1 \rfloor$, já que a parcela
$\Theta^{h+2}/(Sqrt(5) * 2^{h+2})$ tende a zero quando h cresce

Ou

$n \cong \lfloor 3{,}236^{h+2}/(2{,}236 * 2^{h+2}) - 1 \rfloor$
$\cong \lfloor 1{,}618^{h+2}/2{,}236 - 1 \rfloor$

Ou

$h+2 \cong \log_{1{,}618}(2{,}236.(n+1)) = \log_{1{,}618} 2{,}236 + \log_{1{,}618} n$
$\cong 1{,}67 + 1{,}44.\log_2 n$

$h \cong 1{,}44 \log_2 n$

Resolvido o problema dual pode-se voltar ao problema original, tabelando-se os diversos pares h/n correspondentes às árvores do tipo mostrado e verificando em que faixa enquadra-se a altura para um valor qualquer de n. A fórmula acima é uma aproximação muito razoável.

Retomando a tabela gerada anteriormente, podemos ver, por exemplo que para n = 100, a altura máxima da AVL é 9, pois o número mínimo de chaves de uma AVL de altura 10 é 133.

h	0	1	2	3	4	5	6	7	8	9	10
Fib(h)	0	1	1	2	3	5	8	13	21	34	55
T(h)	0	1	2	4	7	12	20	33	54	88	133

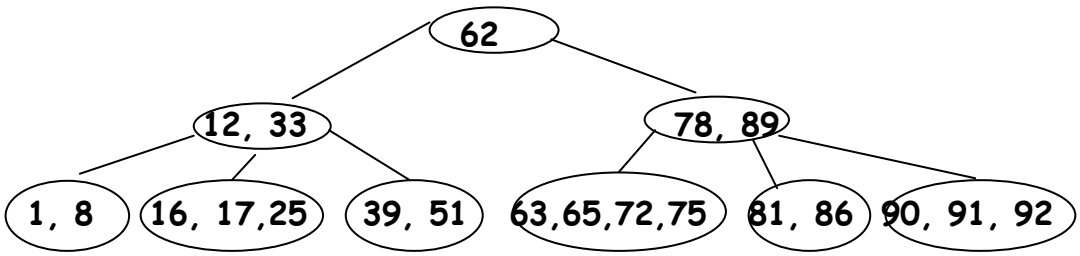
4.3 ÁRVORES B

4.3.1 Introdução

As árvores B foram inventadas por R.**B**ayer e McCreight. Uma árvore B é uma árvore m-ária de busca, que pode ser compreendida como uma extensão da árvore binária de busca e é balanceada, como se verá. Foi criada para se ter árvores balanceadas em disco, sendo base para os diversos gerenciadores de Bancos de Dados. Para otimizar acessos, a idéia é que cada nó da árvore constitui-se uma unidade de I/O. Todo o nó é lido ou gravado de uma só vez. Existe uma definição formal para as árvores B. Elas devem obedecer a uma série de regras, quanto ao número de chaves e de filhos de cada nó. Cada árvore B está associada a um parâmetro, a **ordem da árvore** (denominaremos d essa ordem), que se relaciona ao número máximo de chaves em cada nó. As regras são as seguintes:

1) **Todo nó, exceto a raiz, tem entre d e 2d chaves, inclusive.**
2) **A raiz tem entre 1 e 2d chaves.**
3) **Todo nó, exceto as folhas, tem (k+1) filhos, se k é o número de chaves do nó.**
4) **Todas as folhas estão no mesmo nível.**

Exemplo de árvore B de ordem 2:



Observações:

1) Os nós intermediários têm entre 2 e 4 chaves, 3 e 5 filhos

2) A raiz tem 1 a 4 chaves (1).

3) As folhas estão todas no mesmo nível (3).

4) Pode-se verificar a extensão do conceito de visita em ordem simétrica a essa árvore e perceber que as chaves são obtidas, então, em ordem ascendente.

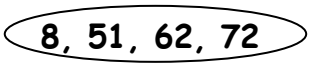
4.3.2 O processo de crescimento da Árvore B: Split e Promoção

O processo descrito a seguir sempre garante o cumprimento das regras anteriores, razão pela qual pode-se concentrar apenas neste processo de atualização, para se entender esta estrutura de dados. As inserções na árvore B são feitas sempre nas folhas. O processo de crescimento da árvore é baseado na operação de **split e promoção**. Essa operação pode ser imaginada da seguinte forma: quando se tenta colocar em um nó completo, com 2d chaves, a (2d+1)-ésima chave, esse nó se divide em dois (**split**). Cada uma das metades fica com **d** chaves e a chave do meio sobe (**promoção**) para o nó pai. Esse processo pode se propagar árvore acima até, eventualmente, a criação de uma nova raiz.

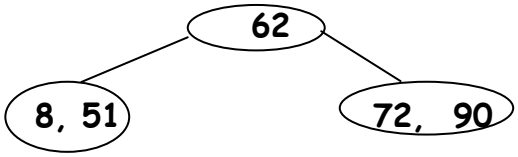
Exemplo: Vejamos a criação da árvore do exemplo anterior, uma árvore B de ordem 2, a partir da inserção da seguinte sequência de chaves:

51, 62, 8, 72, 90, 1, 12, 16, 75, 78, 91, 81, 86, 89, 33, 39, 17, 92, 63, 65 e 25.

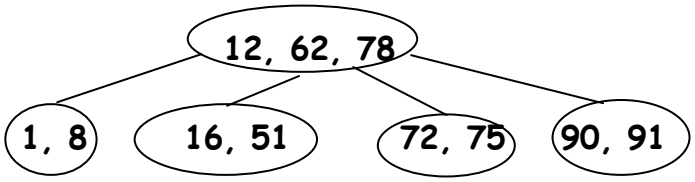
Após a inserção de 51, 62, 8, 72, a árvore é constituída apenas pela raiz:



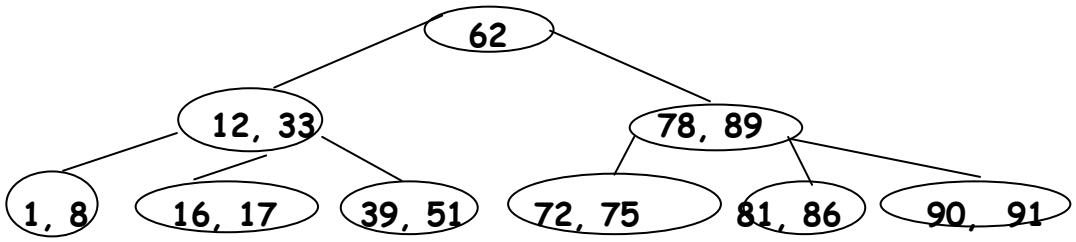
A inserção de 90 provoca o split dessa raiz, obtendo:



A inserção de 1, 12, 16, provoca o split da folha da esquerda, promovendo o 12. A inserção de 75, 78, 91, força o split da folha da direita, com as promoção da chave 78:



A seguir, com a inserção das chaves 81, 86, 89, a folha da direita sofre **split**. E, com a inserção das chaves 33, 39, 17, a segunda folha sofre **split**. Com a promoção das chaves 33 e 89, a raiz também sofre **split**, obtendo-se:



Finalmente, a inserção das chaves 92, 63, 65 e 25 completa a árvore, sem mais nenhum split.

4.3.3 Algoritmo de Busca e Inserção em Árvore B

O algoritmo apresentado a seguir é um algoritmo recursivo para Busca e Inserção. Ele utiliza as declarações a seguir, onde um nó contém (dd+1) chaves:

```
Const d = 2; dd = 4;

Type  ref   = ^pag;
      item  = Record
                  chave: integer;
                  p: ref;
                  cont: integer
              End;
      pag   = Record
                  m: 0..dd {num. de itens}
                  p0: ref;
                  e: array[1..dd] of item;
              End;

Var   raiz,q: ref;  x:integer;   h:boolean;   u:item;
```

No algoritmo apresentado, a variável global **h** tem o importante papel de indicar que a chave ainda não foi encontrada.

```
Procedure Busca_Insercao (x:integer; a:ref; Var h:boolean; Var v:item);
Var k, l, r: integer;   q:ref;    u:item;
Procedure Insercao   {insere u à direita de a^.e[r]}
Var i:integer;   b:ref;
Begin
   With a^ do Begin
      If m < dd Then Begin m := m+1; h := false;
         For i := m downto r+2 do e[i]:= e[i-1]; e[r+1] := u
      End
      Else  Begin   {pag. a^ está cheia; split e promoção}
            New(b);
            If r <= d Then Begin
                  If r = d Then v := u
                  Else Begin
                        v:= e[d]; For i := d downto r+2 do e[i] := e[i-1];
                        e[r+1] := u
                  End;
                  For i := 1 to d do b^.e[i] := a^.e[i+d]
             End
             Else Begin
                  r := r- d; v := e[d+1];
                  For i := 1 to r-1 do b^.e[i] := a^.e[i+d+1];
                  b^.e[r] := u;   For i := r+1 to d do b^.e[i] := a^.e[i+d];
             End;
             m := d; b^.m := d; b^.p0 := v.p; v.p := b
      End
   End   {With}
End;  {Insercao}
```

```
Begin  {busca chave x na pag a^. h = false}
       If a = Nil Then Begin  {item com chave x não está na árvore}
              h := true;  With v do Begin chave := x; cont := 1; p := Nil  End
       End
       Else With a^ do Begin
              l := 1; r := m  {Pesquisa binária}
              repeat k := (l+r) div 2;
                 If x <= e[k].chave Then r := k-1;
                 If x => e[k].chave Then l := k+1;
              until r < l;
              If (l - r) > 1 Then Begin   {Encontrou}
                  e[k].cont := e[k].cont+1;    h := false
              End
              Else  Begin  {item não está nesta página}
                     If r = 0 Then q := p0
                     Else q := e[r].p;
                     Busca_insercao (x,q,h,u); If h Then Insercao
              End
       End {With}
End   {Busca_Insercao}
Procedure Imprimearv (p:ref; l:integer);   Var i : integer
Begin
   If p <> Nil Then With p^ do Begin
       For i := 1 to l do Write ("    ");
       For i := 1 to m do Write (e[i].chave: 4); Writeln;
       Imprimearv (p0,l+1);    For i := 1 to m do Imprimearv (e[i].p,l+1);
   End
End;
Begin
       raiz := Nil; Read(x);
       While x <> 0 do Begin
          Writeln ("Busca da chave: ",x);    Busca_Insercao(x,raiz,h,u);
          If h Then Begin     q := raiz; New(raiz);
              With raiz^ do Begin    m := 1; p0 := q; e[1] := u
              End
          End;
          Imprimearv(raiz,1); Read(x);
       End;
End.
```

Análise do algoritmo:

a) Complexidade:
a.1) Melhor caso médio: NCmed = $\log_2 n$
NANmed = cte (número de acessos a nós)
a.2) Pior caso médio: NCmed = $\log_2 n$
NANmed = cte
a.3) Caso médio:
a.3.1) Buscas bem sucedidas,n entradas distintas,prob.acesso =s
NCmed = $\log_2 n$
a.3.2) Buscas bem sucedidas, acessos a nós
NANmed = $\log_{m+1}(n+1)/2$

b) Atualizações:
Inserção: TE = $O(\log_2 n)$
Deleção: TE = $O(\log_2 n)$

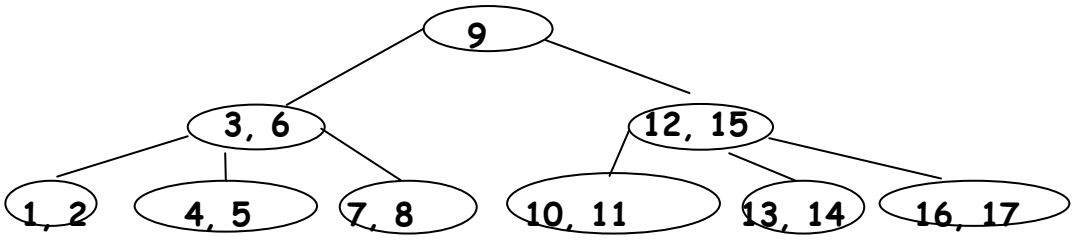
c) Buscas Especiais:
Próxima chave: TE = cte (se a árvore for costurada)
= $O(\log_2 n)$ (se a árvore não for costurada)
Faixa: TE = $O(\log_2 n)$
Prefixo: TE = $O(\log_2 n)$
d) Espaço:
Se usado o processo normal de inserção, o espaço médio ocupado na árvore é de 75%. Gasta-se, portanto, espaço para os links, além de se desperdiçar 25% de todo o espaço, devido ao próprio método de crescimento da árvore.

4.3.4 Árvores B Mínimas e Máximas

Para se avaliar a altura máxima que uma árvore B com n chaves pode atingir, utiliza-se os dados das árvores B mínimas.

Def: Uma **árvore B mínima** é uma árvore B de altura máxima, para dado número de chaves n e ordem d.

Exemplo: Para n = 17, d = 2, uma possível árvore B mínima é



Queremos avaliar a altura de uma árvore B mínima com n chaves. Entretanto é mais fácil iniciar verificando um problema dual: qual o número mínimo de chaves em uma árvore B com dada altura h.

Seja T(h) o número mínimo de chaves da árvore B de altura h, ordem d.

Pode-se calcular esse valor, examinando esse tipo de árvore. Ela tem uma chave na raiz e duas subárvores idênticas, todas com d chaves em cada nó, (d+1) filhos e altura (h - 1). Portanto

T(h) = número mínimo de chaves da árvore B de ordem d e altura h

$= 1 + 2(d + d(d+1) +...+ d(d+1)^{(h-2)})$

$= 1 + 2d(1 + (d+1) +...+ (d+1)^{(h-2)})$

$= 1 + 2d((d+1)^{(h-1)} -1) - 1)/(d+1-1) = 1 + 2((d+1)^{(h-1)} -1) = 2(d+1)^{(h-1)} - 1$

Obtemos, então a seguinte relação entre h e n para a árvore B mínima:

$n = 2(d+1)^{(h-1)} - 1 \Rightarrow (n+1)/2 = (d+1)^{(h-1)} \Rightarrow h - 1 = \log_{d+1} ((n+1)/2)$

$h = \log_{d+1} ((n+1)/2) + 1 \approx \log_{d+1} ((n+1)/2)$

Def: Uma **árvore B máxima** é uma árvore B de altura mínima, para dado número de chaves n e ordem d.

De forma análoga à anterior, para avaliarmos a altura de uma árvore B máxima com n chaves, ordem d, é mais fácil verificar um problema dual: qual o número máximo de chaves em uma árvore B, ordem d com dada altura h. Esse valor é fácil de ser encontrado, pois essa árvore tem, em todos os nós, 2d chaves e, portanto (2d + 1) filhos.

T(*h*) = número máximo de chaves da árvore B de ordem d e altura *h*
= 2d+ 2d(d+1)+...+2d(2d+1)$^{(h-1)}$ = 2d(1 +(2d+1)+...+(2d+1)$^{(h-1)}$)
= 2d((2d+1)^h - 1)/(2d+1-1) = (2*d*+1)^h - 1

Obtemos, então a seguinte relação entre h e n para a árvore B máxima:

n = (2d+1)^h - 1 $\Rightarrow$ n+1 = (2d+1)^h => h = $\log_{2d+1}(n+1)$

Para melhor compreendermos os resultados anteriores, podemos tabelar as alturas mínimas e máximas para uma árvore B com n chaves e ordem d:

Árvore mínima			
n\d	10	50	100
100	**2**	**2**	**1**
10.000	**4**	**3**	**2**
1.000.000	**6**	**4**	**3**

Árvore máxima		
10	50	100
2	**1**	**1**
4	**2**	**2**
5	**3**	**3**

A tabela acima evidencia que, à medida que o número de chaves e da ordem da árvore cresce, praticamente não há diferença entre as alturas das árvores mínimas ou máximas, o que quer dizer que qualquer árvore B com esse número de chaves tem aproximadamente a mesma altura.

4.3.5 Deleção em Árvores B

A deleção em árvores B também inicia-se nas folhas. Esta operação não possui a simetria em relação à inclusão. Apenas o mecanismo oposto ao do split e promoção não é suficiente. Esse mecanismo, denominado **fusão**, ocorre quando um nó fica com menos que d chaves e quando há um nó irmão com exatamente d chaves. Os dois nós se fundem, juntamente com uma chave do nó pai, que desce. Como nem sempre o nó irmão tem exatamente d chaves, é necessário o mecanismo de **recombinação**: algumas chaves passam do nó irmão para o nó que sofreu "**underflow**" , com uma passagem intermediária no nó pai.

Então as regras para deleção são as seguintes:

a) Se a chave a ser deletada está na **folha** ela é apagada. Caso ocorra "**underflow**", tenta-se, inicialmente, fazer uma **recombinação** com um nó irmão, ou seja, a redistribuição das chaves entre esses dois nós.

Se não for possível fazer recombinação (isso ocorre no caso em que o número de chaves do nó irmão é mínimo) faz-se **fusão**: juntam-se as chaves dos nós irmãos com a correspondente do nó pai. Essa operação é a inversa do "split e promoção".

Como há uma descida de chave do nó pai, pode ocorrer "**underflow**" nesse nó. Para a correção do problema, faz-se um processo de recombinação e fusão, só que agora no nível do nó pai, o que pode, sucessivamente, propagar o processo de fusão até a raiz, levando, eventualmente, a uma diminuição da altura da árvore.

b) Se a chave a ser deletada está em algum nó **intermediário**, então ela é apagada e substituida pela chave sucessora, que sempre está em uma folha e recai-se, então, no caso anterior.

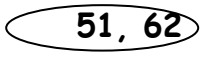
4.3.6 Índices e Árvores B+

As tabelas para árvores B mínima e máxima mostraram que, quanto maior a ordem, menor a altura da árvore. Por outro lado, quanto menor essa altura, menor número de acessos a disco, pois o acesso a cada nó implica um acesso a disco. Isso leva à idéia de separar dados de chaves e constuir-se uma estrutura de acesso aos dados, considerando-se apenas as chaves. Esta estrutura é normalmente chamade de **Índice**. No Índice temos, então chaves e ponteiros para os dados relativos às chaves. Normalmente podem ser criados vários Índices para uma mesma tabela de Banco de Dados.

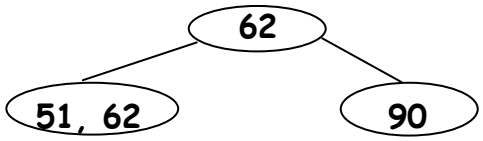
Outra possibilidade é ter-se uma integração Índice + Dados. Neste caso, temos uma estrutura denominada **Árvore B+.** Numa Árvore B+, os dados ficam guardados nas folhas e os demais nós contêm apenas chaves. Normalmente, a ordem das folhas é diferente da ordem dos demais nós, pois estes contêm apenas as chaves, cabendo então mais chaves nos mesmos. O que diferencia uma árvore B+ de uma árvore B é o fato de que o split de folha tem que ser diferente do split de um nó intermediário, pois, embora no split de folha a chave seja promovida, ela também fica na folha, juntamente com os dados respectivos.

O exemplo a seguir ilustra esse princípio. Vejamos a criação de uma árvore B+ de ordem 1, a partir da inserção da seguinte sequência de chaves:
51, 62, 90, 1, 81, 86.

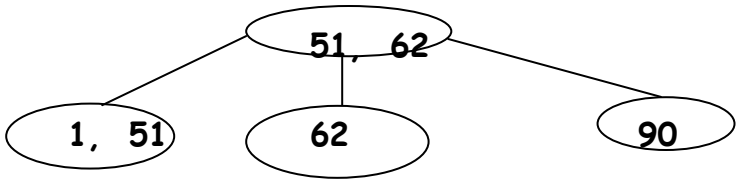
Após a inserção de 51, 62, temos a árvore:



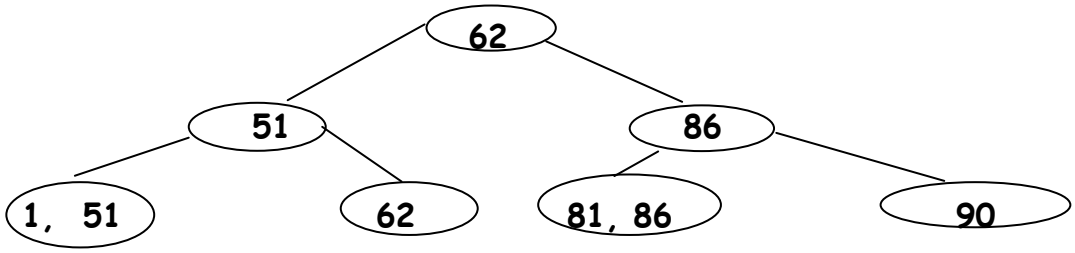
A inserção de 90 provoca o split, promovendo a chave 62, mas esta chave também fica na folha:



A inserção de 1, provoca novo split da folha da esquerda, com a promoçãoções de 51:



A seguir, com a inserção das chaves 81 e 86, a folha da direita sofre **split**. Só que a promoção de 86 levará a um split na raiz. Entretanto, neste novo split a chave promovida não permanece no nó original! Obtemos:



Excetuando-se o split de folha, o comportamento da Árvore B+ é idêntico ao da árvore B.

A organização de um arquivo (tabela) com a estrutura de Árvore B+ permite ainda a possibilidade do tratamento sequencial por chave (de forma ordenada), desde que se coloque um link horizontal nas folhas, apontando para a próxima folha onde estão as próximas chaves. Note-se que nas folhas estão presentes todos os dados do arquivo.

4.4 Complementos

4.4. Árvores de Busca com frequências de acesso distintas.

a) Frequências conhecidas

Neste caso usa-se um algoritmo, baseado em programação dinâmica, para construir a árvore de busca ótima para as chaves e frequências dadas. A árvore ótima é a árvore que têm a complexidade de caso médio mínima, ou seja, onde o número médio de comparações é mínimo.

4.5 Exercícios

4.1 Mostrar todas as ABBs distintas p/ as chaves A,B,C

4.2 Criar ABBs para as seguintes sequências de chaves:

a) CAP,AQU,PEI,ARI,TOU,GEM,CAN,LEA,VIR,LIB,ESC,SAG

b) AQU,ARI,CAN,CAP,ESC,VIR,TOU,SAG,PEI,LIB,LEA,GEM

c) AQU,VIR,ARI,TOU,CAN,SAG,CAP,PEI,ESC,LIB,GEM,LEA

4.3 Demonstrar que o número de árvores degeneradas distintas para n chaves = T(n) = 2^{n-1}.

4.4 Implementar o algoritmo de Busca e Inserção em ABB.

4.5 Implementar um algoritmo para impressão de uma ABB de forma a se visualizar rapidamente a árvore. Sugestão: imprimir um nó por linha, deslocando a impressão de acordo com o nível do nó e percorrendo a árvore em ordem antissimétrica.

4.6 Desenhar todas as ABBs distintas p/ as chaves A,B,C.

4.7 Calcular o número de árvores binárias de busca distintas,T(7) através de dois métodos diferentes: usando a recorrência e usando a fórmula fórmula.

4.8 Implementar um algoritmo para calcular o número de árvores binárias de busca distintas, T(n) para o maior n possível.

4.9 Usar a aproximação de Stirling para n! para comparar o número de ABBs distintas para n chaves com o número de ABBs degeneradas p/ as mesmas n chaves.

4.10 Implementar um algoritmo para buscas por faixas em ABBs.

4.11 Desenvolver e implementar um algoritmo para gerar uma boa ABB para um grupo conhecido de chaves.

4.12 Analisar o algoritmo do exercício 4.11.

4.13 Desenhar o resultado do exercício 4.11, para grupos notáveis de chaves (26 letras do alfabeto, 12 notas musicais, astros do sistema solar etc).

4.14 Desenvolver e implementar um algoritmo para a construção de uma ABB ótima.

4.15 Verificar se a estratégia de construir uma ABB fixa, colocando as chaves na árvore por ordem decrescente de frequência é boa.

4.16 Mostrar, com um exemplo simples, que a chave de maior frequência não está necessariamente na raiz da ABB ótima.

4.17 Criar e balancear, passo a passo, as árvores AVL usando as entradas do exercício 4.2.

4.18 Implementar o algoritmo de Busca e Inserção numa AVL.

4.19 Pesquisar um algoritmo para deleção em árvore AVL.

4.20 Criar e balancear, passo a passo, a árvore AVL usando como entrada as letras do SRTRING (string formado pelas 12 primeiras letras não repetidas de seu nome, completando, se necessário, com letras iniciais do alfabeto)

4.21 Imaginar um algoritmo para balancear uma árvore binária, ao final do processo de inserção de uma série de chaves, sem balanceamento intermediário.

4.22 Calcular o número mínimo de chaves numa AVL mínima de altura 8, T(8), de 2 maneiras: pela recorrência e pela fórmula.

4.23 Desenvolver algebricamente a fórmula de De Moivre p/ n=7

4.24 Desenvolver e resolver a recorrência que calcula o número de árvores AVL mínimas distintas, p/ determinado h.

4.25 Desenvolver uma fórmula para indicar a altura máxima de uma AVL para n qualquer.

4.26 Pesquisar a recorrência que fornece o número de AVLs distintas para dado n.

4.27 Mostrar com um exemplo, que nem sempre funciona a estratégia de somente balancear uma árvore AVL após uma sequência de inclusões.

4.28 Criar, passo a passo, uma árvore B de ordem 1, usando-se o SRSTRING (ver exercício 4.20).

4.29 Mostrar falsas árvores B, cada uma delas quebrando apenas uma das regras de definição.

4.30 Criar a árvore B de ordem 2 para a seguinte sequência de chaves:

A E I O U D W Q F R J P K T C B G H L M S V Z N Y X

4.31 Implementar o algoritmo de Busca e Inserção em árvores B.

4.32 Desenvolver um algoritmo que modifica a inserção na árvore B, fazendo redistribuição de chaves na inserção.

4.33 Desenvolver um algoritmo para inserção e busca na árvore B*, que é uma variante da árvore B, onde o "split" é feito de 2 para 3, ou seja: aumenta-se a ocupação de cada nó para, no mínimo 2/3. Quando há overflow faz redistribuição com um nó irmão. O "split" só é feito quando dois nós estão cheios e, nesse caso, o split é feito jogando-se as chaves de dois nós para 3.

4.34 Pesquisar algoritmos para compactar árvores B.

4.35 Demonstrar que, em uma árvore B, a chave sucessora de qualquerchave em nó intermediário está sempre na folha.

4.36 Deletar as seguintes chaves na árvore B do exercício 4.29: H D E R X F

4.37 Encontrar a expressão para o número mínimo de chaves distintas em uma árvore B+ de ordem d e altura h.

5 Listas de Prioridade

5.1 Definições

Listas de prioridade são estruturas de dados para diversas situações práticas onde trabalha-se com um conjunto de dados dinâmico no qual a principal operação é a seleção, em determinados momentos, do elemento de maior prioridade. Normalmente não é eficiente manter-se o conjunto ordenado, devido à sua natureza dinâmica. A lista de prioridade tem um caráter de ordenação parcial e resolve mais eficientemente os problemas desta estrutura.

Exemplos dessa situação são o controle de execução de jobs pelo Sistema Operacional, controle de listas de candidatos a promoção numa empresa, listas de candidatos a transplante, dentre outras.

As principais operações que tornam dinâmica a lista de prioridade, além da seleção do elemento de maior prioridade, são:

a) inserção de um novo elemento
b) deleção de um elemento
c) mudança de prioridade de algum elemento da lista.

5.2 Heaps

Uma das estruturas de dados adequadas à implementação de listas de prioridade é o Heap. Um Heap é um vetor de n elementos, que pode ser imaginado como uma árvore binária cheia, com as seguintes características:

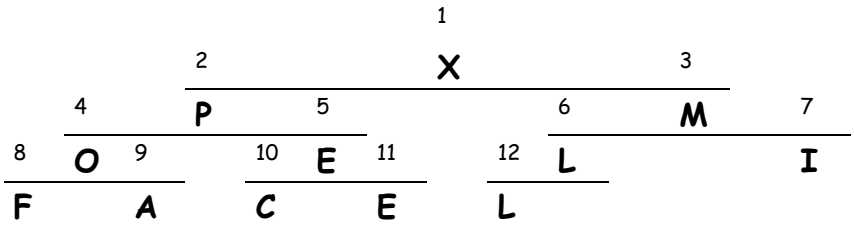
a) a raiz é o elemento de índice 1
b) os nós intermediários são aqueles de índice 1 a $\lfloor n/2 \rfloor$
c) os filhos de um nó intermediário i estão nos índices 2i e 2i+1, não existindo esse último filho para o último nó intermediário, se n for par.
d) o critério de organização do Heap é o de que a chave de cada nó intermediário é maior ou igual às dos filhos. (Alternativamente, o critério de organização pode se inverter, mantendo-se os elementos de menor prioridade no alto do Heap).

Exemplo:

Vetor

1	2	3	4	5	6	7	8	9	10	11	12
X	**P**	**M**	**O**	**E**	**L**	**I**	**F**	**A**	**C**	**E**	**L**

Heap



5.2.1 Operações básicas em Heap

Há duas operações básicas num Heap das quais derivam as demais: ***SobeHeap*** e ***DesceHeap***.

SobeHeap é a operação de inserção de uma chave no final de um Heap, com o consequente acerto para manutenção do critério de organização do mesmo.

DesceHeap é a operação de substituição de uma chave em um ponto qualquer do Heap, sabendo-se que a prioridade desta chave é menor ou igual à da chave pai. Aqui também tem que ser feito o acerto do Heap. Tipicamente esta operação é feita quando se retira o elemento de maior prioridade do Heap, colocando em seu lugar o último elemento do Heap.

```
Procedimento SobeHeap (k);
Inicio
        A[0] ← A[k];
        Enquanto (A[⌊k/2⌋] < A[0]):
           A[k]  ← A[⌊k/2⌋];
           k     ← ⌊k/2⌋ ;
        Fe;
        A[k] ← A[0];
Fim;
```

```
Procedimento DesceHeap (k, m);
Inicio
        t ← A[k];
        Enquanto (k ≤ ⌊m/2⌋)):
                j ← 2 * k;
                Se (j < m) e (A[j] < A[j+1])  Então  j ← j+1;
                Se (t ≥ A[j])  Então
                        Parar loop;
                Senão
                        A[k] ← A[j];   k ← j;
                Fs;
         Fe;
        A[k] ←  t;
Fim;
```

5.2.2 Transformação de um vetor em um Heap

Muitas vezes um Heap deve ser criado a partir de chaves já existentes em um vetor. A criação do Heap, neste caso, pode ser feita a partir de qualquer uma das duas operações básicas.

a) com SobeHeap, inserindo cada uma das chaves a partir do segundo elemento.

```
Procedimento CriaHeap;
Inicio;
    Para i de 2 até n:
            SobeHeap(i);
    Fp;
Fim;
```

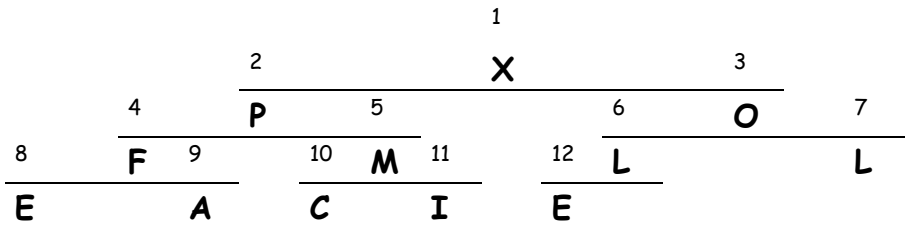
Exemplo:

Passo	1	2	3	4	5	6	7	8	9	10	11	12
	E	X	E	M	P	L	O	F	A	C	I	L
1	X	E										
2	X		E									
3	X	M		E								
4	X	P			M							
5	X		L			E						
6	X		O				L					

7		P		F				E				
8				F					A			
9					M					C		
10					M						I	
11			O			L						E
	X	P	O	F	M	L	L	E	A	C	I	E

Situação Final

Heap



b) com DesceHeap, acertando sistematicamente subHeaps menores, através de um loop descendente do último nó intermediário até a raiz:

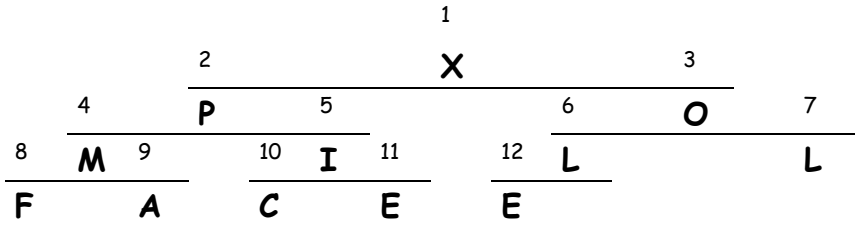
Procedimento **CriaHeap**;
Inicio
  Para i descendo de ⌊n/2⌋ até 1:
    DesceHeap (i, n);
  Fp;

Exemplo:

Passo	1	2	3	4	5	6	7	8	9	10	11	12
	E	X	E	M	P	L	O	F	A	C	I	L
1						L						L
2					P					C	I	
3				M				F	A			
4			O			L	E					
5		X		M	P							
6	X	P	O	M	I					C	E	
	X	P	O	M	I	L	E	F	A	C	E	L

Situação Final

Heap



5.3 HeapSort

Este é um dos mais importantes métodos de ordenação em vetores.

a) **Idéia**: Transformar o vetor em um "Heap" e, sucessivamente, trocar o primeiro elemento do "Heap" com o último do Heap, diminuindo o tamanho do Heap e acertando-o.

b) **Algoritmo**:

```
Procedimento Heapsort;
Inicio:
        CriaHeap;
        Para i decrescendo de n a 2:
                Troca_Elem (1, i);
                DesceHeap(1, i-1);
        Fe;
Fim;
```

c) **Exemplo**, mostrando os elementos envolvidos em comparações e trocas (estas indicadas em vermelho).

Passo	1	2	3	4	5	6	7	8	9	10	11	12	
	E	X	E	M	P	L	O	F	A	C	I	L	
1						L						L	Criação do Heap
2					P					C	I		
3				M				F	A				
4			O			L	E						
5		X			P	L							
6	X	P	O	M	I					C	E		
	X	P	O	M	I	L	E	F	A	C	E	L	Heap criado
7	P	M	O	L	I			F	A			X	X <-> L
8	O	M	L			E	E				P		P<-> E
9	M	L	L	F	I			C	A	O			O <-> C
10	L	I	L	F	A				M				M <-> A
11	L	I	E			C		L					L <-> C
12	I	F	E	E	A		L						L <-> E
13	F	E	E	C	A	I							I <-> C
14	E	C	E	A	F								F <-> A
15	E	C	A	E									E <-> A
16	C	A	E										E <-> A
17	A	C											C <-> A
	A	C	E	E	F	I	L	L	M	O	P	X	Situação Final

d) **Análise do Algoritmo:**

d.1) Complexidade:

Melhor caso = Vetor Inv. Ordenado; NC =~N log(N)

= O(N log N)

Pior caso = Caso Médio = N * log(N) = O(N log N)

d.2) Estabilidade: Algoritmo não estável

d.3) Situações Especiais: Algoritmo de uso amplo, não tendo pior caso.

d.4) Memória necessária: nenhuma memória adicional.

e) Observações:

e.1) Número de comparações: 67
Número de trocas : 50

e.2) Notar que o algoritmo foi implementado apenas com a operação DesceHeap. Pode-se verificar que a complexidade de criação de um Heap usando o DesceHeap é linear enquanto que usando o SobeHeap, não o é (!).

5.4 Complementos

5.4.1 Complexidade da Transformação de Vetor em Heap usando o SobeHeap

O número de comparações na transformação, NC, é dado por:

$NC = \Sigma_{1 \le i \le (h-1)} i.2^i$

$= \Sigma_{1 \le i \le (h-1)} \Sigma_{i \le j \le (h-1)} 2^j$

$= \Sigma_{1 \le i \le (h-1)} (\Sigma_{0 \le j \le (h-1)} 2^j - \Sigma_{0 \le j \le (i-1)} 2^j)$

$= \Sigma_{1 \le i \le (h-1)} (2^h - 1 - (2^i - 1))$

$= 2^h.\Sigma_{1 \le i \le (h-1)} 1 - \Sigma_{1 \le i \le (h-1)} 2^i$

$= 2^h(h-1) - (2^h - 2) = 2^h(h-2) + 2$

Quando (n + 1) é potência de 2, temos $2^h = (n + 1)$

$NC = (n+1).(\log_2 (n+1) - 2) + 2 =$ **$(n+1).\log_2 (n+1) - 2n$**

$= O(n \log n)$

5.4.2 Complexidade da Transformação de Vetor em Heap usando o DesceHeap

O número de comparações na transformação, NC, é dado por:

$NC = 2.\Sigma\ 2^{i-1}.(h - i),\ 1 \le i \le (h-1) =$

$= 2.(\Sigma_i\ h.2^{i-1} - \Sigma_i\ i.2^{i-1})\ 1 \le i \le (h-1)$

$= 2.(h.\Sigma_i\ 2^{i-1} - 1/2.\ \Sigma_i\ i.2^i)\ 1 \leq i \leq (h-1)$
$= 2.(h.(2^{h-1} - 1) - (2^{h-1} - (h - 2) + 1)$
$= 2.(h.\ 2^{h-1} - h - h.\ 2^{h-1} - h + 2^h - 1)$
$= 2.(2^h - h - 1) \quad = 2^{h+1} - 2.h - 2$

Quando (n + 1) é potência de 2, temos $2^h = (n + 1)$

NC $= 2.((n+1) - \log_2 (n + 1) - 1) =$
$= 2.(n+1 - \log_2 (n + 1) - 1) =$
$= 2.(n - \log_2 (n + 1)) =$
$= O(n)$

Este exemplo ilustra bem como algoritmos funcionalmente equivalentes podem ter complexidades variadas. O algoritmo que usa DesceHeap é mais eficiente que o que utiliza SobeHeap. Uma explicação simplificada para isso é que, no processo que usa SobeHeap, muitas chaves exigem comparações da ordem da altura do Heap, já que as comparações são feitas com chaves de nível superior no Heap. Já no DesceHeap, poucas chaves caem nesse caso. Para a maioria das chaves, as comparações são poucas, já que elas são feitas com chaves de nível inferior, daí o resultado linear.

5.5 Exercícios

5.1) Mostrar os passos da criação de um Heap, usando SobeHeap e DesceHeap, para o CRSTRING.

5.2) Mostrar que a complexidade da criação de um Heap a partir de um vetor é $O(n.\log_2 n)$ usando o SobeHeap e $O(n)$ usando o DesceHeap.

5.3) Escrever um algoritmo para ajuste de um Heap, quando há mudança de prioridade de uma chave.

5.4) Mostrar os passos da ordenação do CRSTRING, usando o Heapsort.

5.5) Fazer um algoritmo de ordenação cuja complexidade de **melhor** caso **seja O(n)** e cuja complexidade de **pior caso seja O(n log n). Dica**: combinar idéias de algoritmos estudados.

5.6) Escrever um algoritmo para seleção do k-ésimo menor elemento de um vetor. Descrever a complexidade do algoritmo.

6 Hashing

6.1 Conceitos

Hashing, ou mais apropriadamente ***Transformação Chave-Endereço***, é um conjunto de métodos que busca diminuir o tempo médio de busca em tabelas. Está baseado na idéia de se trabalhar com um vetor e calcular o índice de busca através de uma função sobre a chave (função hash), que, idealmente, permitiria encontrar a chave em uma tentativa.

A situação ideal corresponderia a se ter funções hash injetoras. Entretanto, funções injetoras são relativamente raras. Além disso, o universo de chaves normalmente é extremamente maior que o número de endereços. Isso leva a que se trabalhe com a possibilidade de coincidência de endereços. Nesse caso, as chaves são chamadas de ***sinônimos***. O uso de hashing, consequentemente, exige que se tenha ***métodos de tratamento de sinônimos***, o que dá origem a várias alternativas, das quais 4 serão examinadas.

Elementos envolvidos em hashing.

Trabalha-se com um vetor, contendo M endereços, dos quais N estarão preenchidos. Usa-se uma função hash (h(k)) para a tentativa inicial de encontrar a chave procurada e um método de tratamento de sinônimos para a continuação da busca, caso não se ache a chave na primeira tentativa.

Como foi dito, normalmente o universo de chaves de interesse contém um número de chaves muito maior que o número de endereços. Deve-se notar, no entanto, que, na construção da tabela, só algumas das chaves serão escolhidas, normalmente de forma aleatória. Isso quer dizer que, em geral, $N < M$. Note-se, também, que M é prefixado, pois é, obrigatoriamente, um parâmetro da função hash. Isso dificulta ampliações da tabela.

Outro ponto a ser notado é que, a aplicação da função hash sobre uma chave pode ser vista como um processo de redução dessa chave.

6.2 Funções hash.

Devem ser usados os seguintes critérios na escolha da função hash:

a) A imagem da função deve ser o **intervalo 1- M (ou 0 - (M-1))**.

b) Ela deve ser de **cálculo rápido**.

c) A **distribuição** de sinônimos deve ser **uniforme**.

d) Deve-se ter cuidado para a função hash **não preservar propriedades** específicas das chaves.

O primeiro critério é óbvio. O segundo também, pois se o cálculo for muito demorado anularia as vantagens do método. O terceiro critério é uma forma de minimizar a situação inevitável de sinônimos. Quer-se que os endereços tenham probabilidades iguais (ou muito próximas) de ocupação, para que se evite aglomerações em certas áreas da tabela. O quarto e último critério procura levar em conta situações específicas.

Por exemplo, se houver a tendência de muitas chaves sequenciais serem inseridas na tabela, deve-se evitar a escolha de funções que levem a imagens sequenciais (que é o caso da principal função usada, resto da divisão), para evitar aglomerações na tabela.

Principais funções hash usadas:

Há uma grande variedade de funções usadas. Algumas até constituem segredo industrial de fabricantes de software. As três primeiras relacionadas a seguir são mencionadas na maioria dos livros sobre o assunto, mas não têm grande importância. As funções realmente importantes são as três últimas descritas.

a) Escolha de partes da chave

Podem-se concatenar grupos de caracteres extraídos de pedaços da chave. Um exemplo comum é o da escolha de dígitos finais de chaves numéricas. Neste caso M deve ser uma potência de 10. Esta função obedece os três primeiros critérios, mas geralmente falha no último. Além disso obriga a que se trabalhe com M potência de 10, o que limita o seu uso.

b) Dobradura

A idéia é se reduzir o tamanho da chave, através de sucessivas operações de "quebra" da chave em duas partes e "soma" dessas partes. Essa função tem as mesmas limitações da anterior, além de não gerar distribuição uniforme.

Ex: k = 344865 $\Rightarrow$ 344 + 865 = 1209 $\Rightarrow$ 12 + 09 = 21 $\Rightarrow$ h(k) = 21

c) Metade do quadrado

Esta foi uma das primeiras funções de geração de números pseudorandômicos em computadores. A idéia é elevar ao quadrado uma chave numérica e extrair os dígitos centrais do resultado. Essa função tem as mesmas desvantagens da anterior.

Ex: k = 1226, $\Rightarrow$ k^2 = 1503076

h(k) = 030

d) Resto da divisão

Das funções importantes esta é a mais simples. Ela só falha no último critério, quando a situação de uso é tal que haja muitas chaves em sequência, o que costuma ocorrer com frequência. Normalmente usa-se M primo.

h(k) = k mod M + 1

Ex: k = 205; M = 29; h(k) = 205 mod 29 + 1 = 3

d) Ou exclusivo

A idéia é parecida com a de dobradura. Divide-se sucessivamente a chave em duas partes e aplica-se a operação ou exclusivo sobre os bits das duas partes. No final interpreta-se o resultado como um número binário. A única limitação deste método é a de que M deve ser potência de 2.

Ex: k = 10100101001101011101011101010100 (expresso em 32 bits)

Temos: xor(1010010100110101, 1101011101010100)= 0111001001100001

xor(01110010, 01100001) = 00010011 $\Rightarrow$ h(k) = 35

e) Método da multiplicação

Esta função é um aperfeiçoamento da função resto da divisão, visando evitar a manutenção da propriedade de sequência. A idéia é aplicar resto da divisão sobre o produto da chave por um número especial. A teoria sobre este método ainda não está completa. Verificou-se que um número da forma xxxp21 é um bom multiplicador.

h(k) = (kb mod R) mod M + 1

R é uma potência de 10 ou de 2, dependente da máquina, de forma a simplificar a operação kb mod R. Essa operação pode ser implementada como uma multiplicação seguida do abandono de um dos registradores da operação.

Ex : $R = 2^{16}$, M = 613, b = 31415821

h(1) = (1.31415821 mod 65536) mod 613 + 1 = 24077 mod 613 + 1
= 170 + 1 = 171

h(2) = 341, h(3) = 566, h(4) = 123, h(5) = 293,
h(6) = 518, h(7) = 75, h(8) = 245, h(9) = 440,
h(10) = 27

Obs: Notar o espalhamento da função para chaves sequenciais.

6.3 Métodos de Tratamento de Sinônimos

Serão descritos 4 dos principais métodos de tratamento de sinônimos:

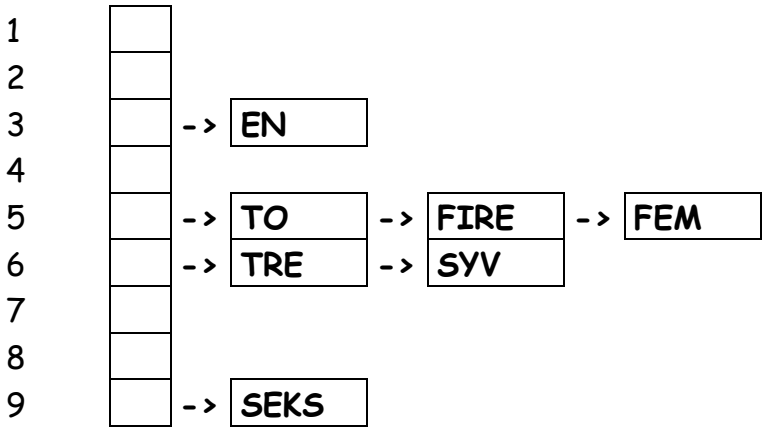
6.3.1 Encadeamento exterior

A idéia é construir uma lista encadeada de sinônimos para cada endereço da tabela. Para a realização deste método, utilizam-se duas estruturas de dados: um vetor de M posições e uma lista encadeada para cada índice do vetor. Há duas variantes deste método, decorrentes da escolha de se colocar ou não uma chave no vetor.

Ex: Suponhamos M = 9 e os seguintes dados:

k	**EN**	**TO**	**TRE**	**FIRE**	**FEM**	**SEKS**	**SYV**
h(k)	**3**	**5**	**6**	**5**	**5**	**9**	**6**

A estrutura resultante é a seguinte:



Algoritmo Busca e inserção em tabela hashing com encadeamento. Ver Exercício 56.

Análise do algoritmo:

a) Complexidade:
- a.1) Melhor caso médio: NCmed = 1 (sem sinônimos)
- a.2) Pior caso médio: NCmed = n/2 (equivalente a sequencial)
- a.3) Caso médio: NCmed = 1 + $\alpha/2$, onde
  α = fator de carga da tabela = N/M

b) Atualizações:
- Inserção/Deleção: TEmed = cte

c) Buscas Especiais:
- c.1) Próxima chave: TEmed = n/2
- c.2) Faixa: TEmed = n/2
- c.3) Prefixo: TEmed = n/2

d) Memória: Memória para os links das listas encadeadas.

6.3.2 Encadeamento Interior (Listas coligadas)

Este método é derivado do anterior. A idéia é fazer o encadeamento na própria tabela. Com isso pode-se ter links "pequenos", pois só precisam endereçar uma área restrita da memória. Como o encadeamento é feito na própria tabela, os sinônimos ocupam endereços de outras chaves, que podem ser obrigadas a entrar nessa lista de sinônimos, resultando em uma coligação de listas de sinônimos.

Ex: Suponhamos M = 9 e os seguintes dados:

k	**EN**	**TO**	**TRE**	**FIRE**	**FEM**	**SEKS**	**SYV**
h(k)	**3**	**5**	**6**	**5**	**5**	**9**	**6**

A estrutura resultante é a seguinte:

End.	X	Chave	Pont
1	F		
2	F		
3	T	EN	0
4	T	SYV	0
5	T	TO	9
6	T	TRE	4
7	T	SEKS	0
8	T	FEM	7
9	T	FIRE	8

Observações:

1) A estrutura é um vetor de 3 campos. O primeiro (X) é um indicador de a posição estar ocupada ou não, o segundo (Chave) é a chave e o terceiro (Pont) é o ponteiro da lista coligada de sinônimos.

2) Há, nessa tabela, duas listas coligadas formadas pelas sequências de endereços 5-9-8-7 e 6-4.

3) A procura de espaço para colocação dos sinônimos é feita partindo-se do fim da tabela para o início. Há outras alternativas, porém essa é a mais eficiente.

Algoritmo Busca e inserção em tabela hashing com Encadeamento Interior (listas coligadas)

```
Type vetor = array[1..M] of record
        x: boolean; chave, pont: integer;
end;

Inicializacao (vet: vetor; pos);
Início
   pos ← M;
   Para i de 1 a M:   vet[i].x ← false;   vet[i].pont ← Nulo;   Fp;
Fim;

Busca_Insercao (Var vet: vetor; Var k: integer);
Início
   i ← hash(k);
   Se  (vet[i].x = false) Então
        vet[i].x  ← true;   vet[i].chave ← k;
   Senão
        Enquanto (vet[i].chave ≠ k)  e  (vet[i].pont ≠ Nulo):
             i  ← vet[i].pont;
        Fe;
        Se (vet[i].chave ≠ k) Então
             Enquanto (vet[pos].x = true) e (pos > 0):
                  pos := pos - 1;
             Fe;
             Se (pos = 0) Então Overflow
             Senão
                  vet[pos].x ← true; vet[pos].chave ← k;
                  vet[i].pont ← pos;
Fim;
```

Análise do algoritmo:

a) Complexidade:

a.1) Melhor caso médio: NCmed = 1 (sem sinônimos)

a.2) Pior caso médio: NCmed = n/2 (equivalente a sequencial)

a.3) Caso médio: $NCmed = 1 + \alpha/4 + (e^2\alpha - 1 - 2\alpha)/8$, onde

α = fator de carga da tabela = N/M

Obs: $\alpha = 1 \Rightarrow$ NCmed = 1.8

b) Atualizações:
Inserção/Deleção: TEmed = cte

c) Buscas Especiais:
c.1) Próxima chave: TEmed = n/2
c.2) Faixa: TEmed = n/2
c.3) Prefixo: TEmed = n/2

d) Memória: Memória para os links das listas coligadas e para os indicadores de uso do endereço.

6.3.4 Endereçamento Aberto

Este é o método mais utilizado para tratamento de sinônimos, devido à simplicidade. A idéia é não utilizar links. Desta forma, os sinônimos são colocados na primeira posição disponível, encontrada percorrendo-se a tabela sequencialmente a partir do endereço calculado. De certa forma, um conjunto de endereços contíguos ocupados constitui uma lista coligada de sinônimos. A tabela é percorrida de forma circular, isto é, após o endereço M pula-se para o endereço 1.

Ex: Suponhamos M = 9 e os seguintes dados:

k	**EN**	**TO**	**TRE**	**FIRE**	**FEM**	**SEKS**	**SYV**
h(k)	**3**	**5**	**6**	**5**	**5**	**9**	**6**

A estrutura resultante é a seguinte:

End.	
1	**SYV**
2	
3	**EN**
4	
5	**TO**
6	**TRE**
7	**FIRE**
8	**FEM**
9	**SEKS**

Observações:

1) O sentido escolhido da busca foi o de cima para baixo. Poderia, também ser o oposto.

2) No algoritmo que será apresentado, supõe-se que haja pelo menos uma posição nula na tabela, o que limita o número de chaves ao máximo de M - 1.

Algoritmo Busca e inserção em tabela hashing com endereçamento aberto

Inicialização;
Início
  n ← 0;
  Para i de 1 a M: vet[i] ← Nulo; Fp;
Fim;

Busca_inserção (k);
Início:
  i ← hash(k);
  Enquanto (vet[i] ≠ Nulo) e (vet[i] ≠ k):
      i ← i mod M + 1;
  Fe;
  Se (vet[i] = k) Então
      BBS
  Senão
      n ← n + 1;
      Se (n = M) Então Overflow
      Senão vet[i] ← k;
Fim;

Análise do algoritmo:

a) Complexidade:
a.1) Melhor caso médio: NCmed = 1 (sem sinônimos)
a.2) Pior caso médio: NCmed = n/2 (equivalente a sequencial)
a.3) Caso médio: NCmed = 1/2 + (1 - α)/α, onde
α = fator de carga da tabela = N/M

b) Atualizações: Inserção/Deleção: TEmed = cte

c) Buscas Especiais:
c.1) Próxima chave: TEmed = n/2 c.2) Faixa: TEmed = n/2
c.3) Prefixo: TEmed = n/2

d) Memória: Como a busca começa a se degradar a partir de fc = .75, deixa-se de reserva pelo menos 25% de todo o espaço, ou seja, superestima-se o tamanho da tabela.

6.3.5 Endereçamento Aberto com Duplo Hashing

Este método é baseado no anterior, aperfeiçoando-o para uma situação comum, quando se usa como função hash o resto da divisão e as chaves possuem a tendência de sequência. O problema dessa situação é que se formam aglomerados na tabela, prejudicando a performance da busca. A idéia é que a busca de um sinônimo não seja feita sequencialmente como no método anterior, mas aos pulos, onde o pulo é calculado por uma segunda função (h_2), calculada para cada chave. O método anterior seria um caso particular do atual para a situação em que h_2 (k) = cte = 1.

Ex: Suponhamos M = 9 e os seguintes dados:

k	**EN**	**TO**	**TRE**	**FIRE**	**FEM**	**SEKS**	**SYV**
h(k)	**3**	**5**	**6**	**5**	**5**	**9**	**6**
h_2 (k)	**4**	**2**	**2**	**4**	**5**	**2**	**5**

A estrutura resultante é a seguinte:

End.	
1	**FEM**
2	**SEKS**
3	**EN**
4	
5	**TO**
6	**TRE**
7	**SYV**
8	**FEM**
9	**FIRE**

Observações:

1) O sentido escolhido da busca foi de cima para baixo. Poderia, também ser o oposto.

2) No algoritmo que será apresentado, supõe-se que haja pelo menos 1 posição nula na tabela, o que limita o número de chaves ao máximo de M - 1.

3) A função $h_2(k)$ tem que ter a propriedade de só gerar números que sejam relativamente primos a M. Caso contrário pode haver problemas ilustrados no exercício 73.

Algoritmo Busca e inserção em tabela hashing com endereçamento aberto e duplo hashing

```
Inicialização;
Início
   n ← 0;
   Para i de 1 a M:  vet[i] ← Nulo;  Fp;
Fim;

Busca_inserção (k);
Início:
   i ← hash(k);   u ←  hash2(k);
   Enquanto (vet[i] ≠ Nulo)  e (vet[i] ≠ k):
         i ← (i + u – 1) mod M + 1;
   Fe;
   Se (vet[i] = k) Então
         BBS
   Senão
         n ← n + 1;
         Se  (n = M)  Então Overflow
         Senão vet[i] ← k;
Fim;
```

Análise do algoritmo:

a) Complexidade:

a.1) Melhor caso médio: NCmed = 1 (sem sinônimos)

a.2) Pior caso médio: NCmed = n/2 (equivalente a sequencial)

a.3) Caso médio: NCmed = -log (1 - α)/α, onde

α = fator de carga da tabela = N/M

b) Atualizações:
Inserção/Deleção: TEmed = cte

c) Buscas Especiais:
c.1) Próxima chave: TEmed = n/2
c.2) Faixa: TEmed = n/2
c.3) Prefixo: TEmed = n/2

d) Memória: Nenhuma memória adicional.

6.4 Complementos

6.5 Exercícios

6.1 Implementar a função de multiplicação e descobrir pares de valores de b e M para os quais se tem distribuição uniforme.

6.2 Implementar a função do meio do quadrado e verificar se ela gera distribuição uniforme.

6.3 Implementar a função ou exclusivo.

6.4 Analisar se as seguintes funções são boas funções hash:

a) $h(k) = k^2 \bmod M + 1$

b) $h(k) = (k \bmod P) \bmod M + 1$

6.5 Mostrar que a função dobradura não gera distribuição uniforme.

6.6 Verificar que as funções AND e NAND, aplicadas de forma análoga ao XOR, não geram distribuição uniforme.

6.7 Fazer um algoritmo para transformar chaves não numéricas em chaves numéricas.

6.8 Implementar o algoritmo para tabela hashing com encadeamento.

6.9 Verificar que a tabela com encadeamento tem NC = N/(2M), quando N >> M.

6.10 Criar a tabela com encadeamento a partir do Selfstring (Ver Exercício 31), usando-se M = 13 e h(k) = (posição da letra k no alfabeto) mod M + 1. Calcular NC para a tabela gerada.

6.11 Modificar o algoritmo de tabela hashing com encadeamento, para permitir deleções na tabela.

6.12: Implementar o algoritmo de tabela hashing com lista coligada.

6.13 Construir um exemplo para mostrar que uma tabela com lista coligadas pode ter o NC minimizado se a tabela for recriada em dois passos: no primeiro recolocam-se na tabela apenas chaves não sinônimas e no segundo as sinônimas.

6.14 Explique, baseado no algoritmo de tabela hashing com lista coligada, porque a estratégia de procurar espaços para sinônimos, partindo-se de baixo para cima é eficiente.

6.15 Criar uma tabela com listas coligadas de forma análoga ao Exercício 6.10.

6.16 Modificar o algoritmo de tabela com lista coligada para permitir deleções na tabela. Notar que a deleção não é apenas recuar a lista coligada de sinônimos.

6.17 Implementar o algoritmo de tabela com endereçamento aberto.

6.18 Criar a tabela com Endereçamento aberto de forma análoga à do Exercício 6.10.

6.19 Criar um exemplo para mostrar que a deleção de chaves na tabela com Endereçamento aberto não pode ser feita apenas apagando-se a chave.

6.20 Modificar o algoritmo de tabela hashing com endereçamento aberto, para permitir deleções de chaves.

6.21 Modificar o algoritmo de tabela hashing com endereçamento aberto, de forma que se possa colocar M chaves na tabela.

6.22 Implementar o algoritmo de tabela com duplo hashing.

6.23 Criar a tabela com Endereçamento aberto e duplo hashing, de forma análoga à do Exercício 58 e usando-se $h_2(k) = 2^p$, p = (posição de k no alfabeto) mod 4.

6.24 Criar a tabela hashing com eadh, para M = 9 e

k	A	B	C	D	E	F	G
h(k)	5	9	5	4	1	8	6
h_2 (k)	2	5	4	5	4	5	4

6.25 Inventar um exemplo para mostrar que o algoritmo de tabela com duplo hashing pode entrar em loop se h2(k) gerar um endereço que seja submúltiplo ou que tenha um divisor comum com M.

6.26 Criar os 4 tipos de tabela para M = 11 e

k	A	B	C	D	E	F	G	H	I	J
h(k)	3	4	5	3	5	4	6	1	9	11
h_2 (k)	2	4	3	5	7	4	8	3	3	2

Comparar as tabelas segundo o NC.

7 Pesquisa Digital

7.1 Conceitos

É um conjunto de métodos de busca que baseia-se na **estrutura interna** das chaves para o processo de busca. A Pesquisa Digital tem tido aplicações crescentes, combinada a outros métodos, notadamente Hashing. Aplica-se especialmente para o **processamento de texto** e de chaves de tamanho variável. Só não tem sido mais universalmente usada porque, como leva em conta a estrutura interna da chave, as estruturas resultantes dependem da codificação utilizada. Como ainda não há um padrão para codificação de dados, tem-se uma certa restrição na portabilidade das tabelas e arquivos gerados.

Para os exemplos apresentados, serão usadas como chaves as 26 letras do alfabeto, codificadas em 5 bits, representando a codificação binária da posição da letra no alfabeto. Temos a seguinte tabela:

A - 00001	F - 00110	K - 01011	P - 10000	U - 10101	Z - 11010
B - 00010	G - 00111	L - 01100	Q - 10001	V - 10110	
C - 00011	H - 01000	M - 01101	R - 10010	W - 10111	
D - 00100	I - 01001	N - 01110	S - 10011	X - 11000	
E - 00101	J - 01010	0 - 01111	T - 10100	Y - 11001	

A numeração dos bits é uma comumente utilizada, que numera os bits da direita para a esquerda, começando por 0 e terminando por MAXBIT. Desta forma, o número de bits da chave é designado por MAXBIT + 1.

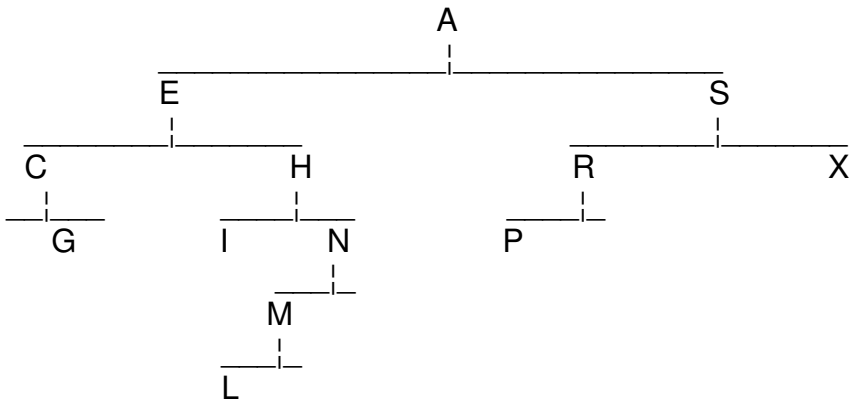
7.2 Árvores de Busca Digitais

7.2.1 Definições

Árvore digital de busca é uma árvore muito parecida com a ABB, só que o processo de percurso na árvore é baseado na comparação de determinado bit da chave, desviando-se para a subárvore da direita quando esse bit for 1 e para a da esquerda quando for 0. A cada nível examina-se determinado bit da chave, começando-se pelo bit mais à esquerda e decrescendo para a direita. Desta forma, a altura máxima da árvore é MAXBIT + 2.

Ex: A ADB criada a partir da sequência de chaves:

A S E R C H I N G X M P L é a seguinte:



7.2.2 Algoritmo para Busca e inserção em Árvore Digital de Busca

```
Const Maxbit = 5;
Type  link   = ^arvd
      arvd   = record chave: char; le,ld: link  end;
Var   z,raiz: link;
      k: char;
Procedure Inicializacao (Var raiz,z:link);
Begin
  New(z); z^.le := z; z^.ld := z; raiz := z;
End;
Procedure Busca (Var raiz,z:link; Var k:char);
Var x: link;
    b: byte;
Begin
  x := raiz; z^.chave := k; b := Maxbit;
  While (x^.chave <> k ) do
    Begin
      If Bit(k,b) = 0 Then x := x^.le Else x := x^.ld;
      b := b -1;
    End;
  If (x <> z) Then BBS
  Else BMS;
End;


Procedure Insercao (Var raiz,z:link; Var k:char);
Var f,x:link;
    b: byte;
Begin
  x := raiz; b := Maxbit;
  Repita
    f := x;
    If Bit(k,b) = 0 Then  x := x^.le Else x := x^.ld;
    b := b -1;
```

```
  Until (x = z);
  New (x); x^.chave := k; x^.le := z; x^.ld := z;
  If raiz = z Then raiz := x
  Else If Bit(k,b+1) = 0 Then f^.le := x Else f^.ld := x;
End;
```

Análise do algoritmo:

a) Complexidade:
a.1) Melhor caso: = caso médio
a.2) Pior caso: = caso médio
a.3) Caso médio: NCmed = log2(n);

b) Atualizações:
Inserção: TEmed = c13.log2(n)
Deleção: TEmed = c13.log2(n)

c) Buscas Especiais:
c.1) Próxima chave: TEmed = c13.n
c.2) Faixa: TEmed = c13.n
c.3) Prefixo: TEmed = c13.log2(n)

d) Memória: Memória para os links da árvore.

Observações:

1) A ADB é uma alternativa natural para a ABBB, pois o balanceamento decorre naturalmente do processo de inserção.

2) A busca por prefixo nesta árvore funciona muito bem. Para se encontrar as chaves com determinado prefixo, busca-se o prefixo na árvore. Todas as chaves da subárvore do nó onde se chegou têm o prefixo dado. Além disso, algumas chaves do caminho de busca também podem ter o prefixo. Isso tem que ser testado.

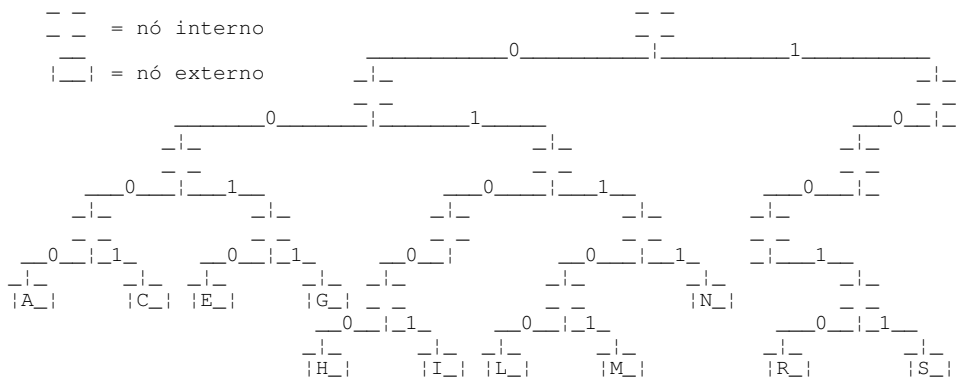
3) A estrutura da ADB não depende muito da ordem de entrada das chaves mas, fundamentalmente, da estrutura dessas chaves e da relação entre elas. Por exemplo, se houver muitos zeros nas chaves, a árvore tenderá a se alongar para a esquerda.

7.3 TRIES

7.3.1 Conceitos

Tries são estruturas que têm semelhança com as ADBs e que foram originalmente criadas para se evitar o teste completo da chave em cada nó visitado numa ADB. O critério de desvio no percurso da árvore é o mesmo daquelas. Entretanto, uma Trie tem dois tipos de nó: nó interno e nó externo. Os nós internos constituem apenas uma estrutura de desvio. As chaves estão nos nós externos, de tal forma que só se faz uma comparação completa, no processo de busca. Essa comparação ocorre quando se chega a um nó externo. O nome Trie deriva da palavra reTRIEval (recuperação), porque essa estrutura foi imagina para recuperação rápida de textos. De forma semelhante às ADBs as Tries possuem altura máxima = MAXBIT + 2.

Ex: A Trie criada a partir da sequência de chaves
A S E R C H I N G M P L é a seguinte:



Observações:

1. A busca de determinada chave leva à chave com estrutura mais parecida à mesma, considerando-se a comparação da esquerda para a direita, ou a um link nulo.

2. A inserção de nós pode dar-se de duas formas. Se, na busca, chegar-se a um link nulo, basta inserir a chave nesse link. Se chegar-se a um nó externo, tem que ser criado um caminho para diferenciar a chave a inserir da chave encontrada, que é a mais parecida com ela. No início da criação da tabela isso leva a algumas não linearidades.

3. A Trie normalmente tem mais nós internos que externos.

7.3.2 Algoritmo A14: Busca e inserção em Tries

Ver Exercício 82.

Análise de A14

a) Complexidade:
a.1) Melhor caso: = caso médio
a.2) Pior caso: = caso médio
a.3) Caso médio: NCmed = log2(n);

b) Atualizações:
Inserção: TEmed = c13.log2(n)
Deleção: TEmed = c13.log2(n)

c) Buscas Especiais:
c.1) Próxima chave: TEmed = c13.n
c.2) Faixa: TEmed = c13.n
c.3) Prefixo: TEmed = c13.log2(n)

d) Memória: Memória para os links da árvore e para os nós de desvio.

Observações:

1) A Trie é uma alternativa natural para a ABBB, pois o balanceamento decorre naturalmente do processo de inserção.

2) A busca por prefixo nesta árvore funciona muito bem. Para se encontrar as chaves com determinado prefixo, busca-se o prefixo na árvore. Todas as chaves da subárvore do nó onde se chegou têm o prefixo dado.

3) A estrutura da Trie não depende da ordem de entrada das chaves mas, fundamentalmente, da estrutura dessas chaves e da relação entre elas. Por exemplo, se houver muitos zeros nas chaves, a Trie tenderá a se alongar para a esquerda.

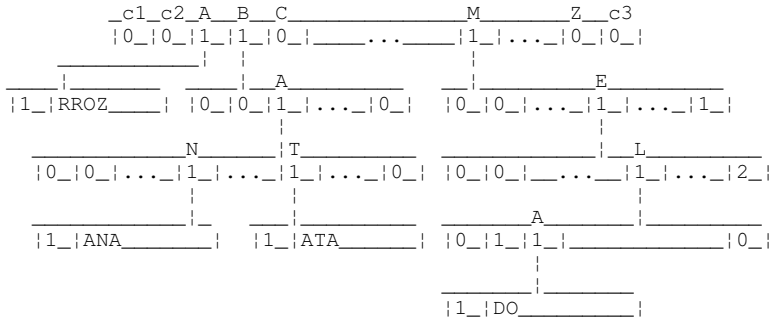
4) Pode-se melhorar a utilização de espaço, pois o caminho de desvio já constitui o prefixo da chave. Entretanto isso só vale a pena se houver muitos elementos na Trie.

7.4 TRIE C

7.4.1 Conceitos

Tries m-árias são extensões da Trie binária, onde em cada nó pode haver várias possibilidades de desvio, considerando-se a análise de um grupo de bits e não de apenas um deles. Tries m-árias têm se mostrado estruturas muito interessantes para armazenamento de vocabulários, devido à possibilidade de rápido acesso às chaves. Uma das implementações de Tries m-árias é a Trie-C (Comprimida), que usa para desvio um caracter a cada nível. Como essa estrutura é só para guardar palavras, há 27 possibilidades de desvio em cada nó (26 letras + branco). Além disso criou-se um esquema que evita representar diretamente os links, pois eles consomem muita memória.

Ex: Trie-C para guardar as palavras:
ARROZ, BANANA, BATATA, MEL, MELADO



Observações:

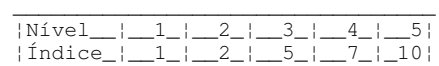
1. Essa estrutura só é boa para tabelas fixas.

2. A idéia é que ela seja representada em um vetor de campos de tamanho fixos, embora havendo dois tipos de nó. É necessária uma tabela (TAB1) que guarda o índice do nó mais à esquerda de cada nível.

3. c1 indica o tipo de nó (0 = interno, 1 = externo)

4. Os links são virtuais, representados por apenas 1 bit. Para se acessar o nó filho, procura-se em TAB1 o índice do primeiro nó do nível seguinte e desloca-se o valor apresentado em c3 (este número é o de sobrinhos à esquerda do nó).

TAB1 para o exemplo seria:



5. Só se representam nos nós externos as partes finais das palavras, pois o início é o prefixo de percurso. Como os nós são de tamanho fixo, a palavra pode não caber e deve-se usar um link para a continuação da palavra. Esse link não foi mostrado.

6. Há ainda o problema de palavras "incluídas", tais como MEL e MELADO. A solução para essa situação é imaginar que a palavra incluída

em outra tem uma terminação com o caracter "branco" e usa-se o campo c2 para expressar isso. c2 = 1 indica que no nível anterior terminou uma palavra incluída.

7.4 Complementos
7.5 Exercícios

Ex. 7.1: Implementar o algoritmo A13.

Ex. 7.2: Demonstre que sempre se poderá incluir uma chave não existente numa ADB (ou seja, nunca é possível esgotar-se todos os bits e chegar-se a um nó ocupado, para uma chave não existente na ADB).

Ex. 7.3: Criar a ADB a partir do Selfstring (Ver Exercício 31) e a codificação usada nesta apostila.

Ex. 7.4: Refaça o Exercício 77 usando a codificação EBCDIC.

Ex. 7.5: Escrever e implementar um algoritmo para busca por prefixo numa ADB.

Ex. 7.6: Criar dois exemplos para mostrar que a estrutura de uma ADB não depende muito da ordem de entrada das chaves.

Ex. 7.7: Pesquisar a recorrência que fornece o número de ADBs distintas para n chaves.

Ex. 7.8: Implementar o algoritmo A14.

Ex. 7.9: Criar a Trie de forma análoga à do Exercício 77.

Ex. 7.10: Criar a Trie de forma análoga ao Exercício 78.

Ex. 7.11: Escrever e implementar um algoritmo para busca por prefixo numa Trie.

Ex. 7.12: Criar um exemplo para mostrar que a estrutura de uma Trie não depende da ordem de entrada das chaves.

Ex. 7.13: Implementar o algoritmo do Alexandre Mele para a criação de uma Trie fixa. Nesse algoritmo ordenam-se as chaves e passa-se a analisar sucessivamente, da esquerda para a direita da ordenação, grupos de 3 chaves seguidas. As mais parecidas são incluídas imediatamente e a terceira, somente se estiver à esquerda.

8 Bibliografia

Cormen, T.H; Leiserson, C.E; Rivest, R.L.; Stein,C; **Algoritmos** , Ed. Campus, 2002

Knuth, D. **The art of computer programming,** Vol 1, Second Edition: Fundamental Algorithms, Addison-Wesley, 1973

Sedgewick, R. **Algorithms in C**, 3rd edition, Parts 1-4 , Addison-Wesley, 1998

Szwarcfiter, J; Markenzon,L; **Estruturas de Dados e Seus Algoritmos**, LTC, 1994