Computação

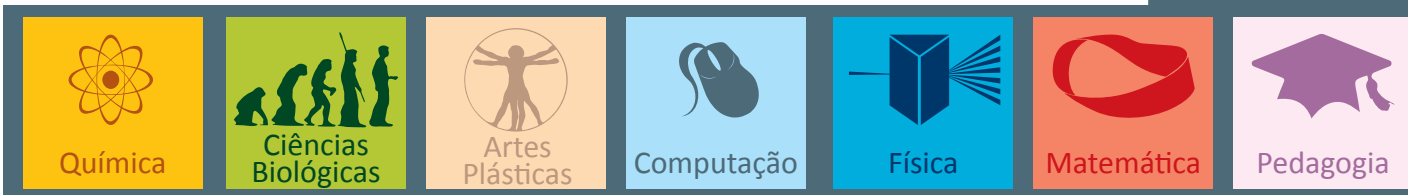
Estrutura de Dados

Mariela Inés Cortés

3ª edição
Fortaleza - Ceará

Ed UECE

2015



Editora Filiada à

ASSOCIAÇÃO BRASILEIRA DAS EDITORAS UNIVERSITÁRIAS

Editora da Universidade Estadual do Ceará – EdUECE
Av. Dr. Silas Munguba, 1700 – Campus do Itaperi – Reitoria – Fortaleza – Ceará
CEP: 60714-903 – Fone: (85) 3101-9893
Internet: www.uece.br – E-mail: eduece@uece.br

Secretaria de Apoio às Tecnologias Educacionais
Fone: (85) 3101-9962

Sumário

Apresentação

O constante aumento da complexidade dos sistemas e suas demandas computacionais relacionadas a tempo e espaço, impõem o desafío de projetar soluções algorítmicas cada vez mais eficientes. Neste contexto, as estruturas de dados e seus algoritmos têm um papel fundamental uma vez que constituem os blocos construtores utilizados na resolução dos mais diversos problemas em todas as areas da computação.

O objetivo do presente livro é apresentar de forma clara e amigável diferentes estruturas de dados e seus algoritmos de manipulação, analisando a estratégia mais eficiente para a sua implementação, em termos de complexidade. Esta análise envolve a utilização de técnicas de projeto associadas a técnicas de programação, as quais são adequadas às caracteristicas da aplicação específica.

O livro está organizado em cinco partes. As primeiras fornecem as bases necessárias para a análise e o projeto de uma boa solução algorítmica incluindo conceitos básicos sobre complexidade, tipos e estruturas de dados. A Parte 3 apresenta o conceito central de Listas, a partir do qual duas importantes e amplamente utilizadas estruturas são derivadas: Pilhas e Filas.

A Parte 4 apresenta a estrutura de Árvore, sua conceituação, implementação e algoritmos de manipulação básicos. Mais especificamente, nesta parte são exploradas as Árvores Binárias de Busca analizando suas características e as implicações em relação à eficiência. Neste contexto, a fundamentação e funcionamento das árvores balanceadas (AVL) são apresentados. Finalmente, na Parte 5 são descritas as técnicas avançadas de pesquisa aplicadas sobre estruturas de dados específicas, tais como tabelas de dispersão, busca digital e estruturas autoajustáveis.

O conteúdo apresentado destina-se principalmente para professores e alunos de graduação em Ciências da Computação ou áreas afins, fornecendo um embassamento teórico e uma clara noção das estratégias a serem seguidas na implementação dos algoritmos para a manipulação eficiente das estruturas de dados mais amplamente utilizadas.

A autora

Capítulo 1

Introdução à Complexidade de Algoritmos

Objetivos

- Neste capítulo é introduzido o conceito de complexidade de algoritmos. Este conceito é central na Ciência da Computação, uma vez que possibilita avaliar e comparar soluções algorítmicas, fornecendo os insumos necessários para determinar qual é o algoritmo mais adequado para resolver determinada classe de problemas.

Introdução

Um algoritmo determina um conjunto de regras não ambíguas as quais especificam, para cada entrada, uma seqüência finita de operações, gerando como resultado uma saída. O algoritmo representa uma solução para um problema se, para cada entrada, gera uma resposta correta, sempre que dispor de tempo e memória suficientes.

Um algoritmo pode ser implementado através de diferentes programas[1]. Ou seja, diferentes implementações podem ser propostas a partir de um único algoritmo. Esta situação nos coloca na dificuldade de escolher qual é a melhor solução para o problema específico. Assim sendo, apenas resolver o problema parece não ser suficiente uma vez que diferentes soluções podem ser idealizadas a partir de algoritmos, para a resolução de um único problema. Neste contexto, se torna necessário um mecanismo que permita determinar se uma solução é melhor do que uma outra, de forma a fornecer uma ferramenta de apoio à decição em relação à qualidade das soluções propostas.

[1] Programa codifica um algoritmo para ser executado no computador resolvendo um problema. É fundamental que o programa produza a solução com dispêndio de tempo e memória: Importância de Projeto e Analise de Algoritmos.

De forma geral, a qualidade de um programa depende do ponto de vista. Por um lado, ***o usuário*** determina a qualidade de um programa através de critérios, tais como:

- Facilidade de uso e entendibilidade da *interface* do programa levando em conta diferentes níveis de experiência.
- Compatibilidade do programa com outros programas ou versões de programas, de forma a facilitar a troca de informação com outros sistemas existentes.

- Desempenho, em relação à velocidade de execução e tempo de resposta.
- Quantidade de memória utilizada pelo programa durante a sua execução, aspecto que pode se tornar um fator limitante.

Os últimos dois itens estão diretamente ligados à quantidade de dados a serem processados, ou seja, ao tamanho da entrada.

Por outro lado, critérios que podem ser determinantes desde o ponto de vista da ***organização*** desenvolvedora incluem:

- Portabilidade do código entre diferentes plataformas.
- Documentação e padronização do código.
- Facilidade de evolução e manutenção.
- Reusabilidade do código, permitindo que porções de um programa sejam reaproveitadas para desenvolver outros produtos, aumentando a produtividade.

A correta avaliação destes critérios é influenciada por diversos fatores, tais como: características do *hardware*, sistema operacional, linguagens e compiladores, plataforma, etc. Estes fatores são considerados acidentais uma vez que não estão diretamente relacionados à qualidade da solução. Em contrapartida, os fatores essenciais são inerentes à solução e determinantes para a sua qualidade. O tempo gasto e o espaço físico na memória são considerados fatores essenciais. Consequentemente é preciso de um método formal para medir o tempo e a memória requeridos pelo algoritmo, independentemente das características de plataforma de *hardware* e software sendo utilizadas.

1. O que é análise de algoritmos?

A análise de algoritmos é o coração da Ciência da Computação e tem por objetivo estabelecer medidas de desempenho dos algoritmos, com vistas à geração de algoritmos cada vez mais eficientes. Adicionalmente fornece fundamentos para a escolha do melhor algoritmo para a resolução de um problema específico, com base na sua ***complexidade computacional***.

O tempo de execução e o espaço de memória alocado são os dois fatores principais que determinam a ***complexidade computacional*** de uma algoritmo.

- Complexidade temporal consiste no número (aproximado) de instruções executadas.
- Complexidade espacial consiste na quantidade de memória utilizada.

De forma geral, tanto a complexidade temporal quanto a espacial podem ser descritas por funções que têm como parâmetro principal o tamanho da entrada sendo processada. Como exemplos temos que:

- Ordenar 100.000 elementos leva mais tempo que ordenar 10 elementos.

- A abertura de um editor de textos leva mais tempo e consume mais memória quando é aberto com um arquivo grande do que com um arquivo pequeno.

Além do tamanho da entrada, as características apresentadas na organização dos dados, também podem influenciar na eficiência de um algoritmo em relação a uma outra solução. Por exemplo, o desempenho de um algoritmo específico para ordenar um conjunto onde os dados se encontram parcialmente ordenados pode ser muito diferente se utilizado para ordenar um conjunto de dados totalmente desordenados, considerando conjuntos de igual tamanho. Logo é importante estabelecer de que forma o tamanho e características da entrada podem influenciar no comportamento do algoritmo.

Em alguns casos, se o algoritmo não é **recursivo**, não contem iterações e não usa algoritmos com essas características, o número de passos necessários pode ser independente do tamanho da entrada, e consequentemente a complexidade temporal é constante.
Um exemplo de algoritmo com estas características[2] é o algoritmo que imprime "HELLO WORD", que possui complexidade constante.

```
#include <stdio.h>
main() {
    Printf (Hello Word!\n");
}
```

Considerando que é impossível fazer uma predição exata do tempo e memória a serem utilizados a estratégia consiste em estabelecer uma *aproximação*, com base em *conceitos e modelos matemáticos*.

[2] A principal característica de um algoritmo recursivo é a ocorrência de chamadas a ele próprio. Para avaliar a complexidade de um algoritmo recursivo é preciso analisar quantas vezes a função vai ser chamada, e quantas operações acarretam cada chamada.

Atividades de avaliação

1. Descreva com suas palavras a relação entre os conceitos de *algoritmo* e *programa*.
2. Determine em quais casos e de que forma as especificações a seguir podem depender do tamanho ou organização dos dados da entrada. Justifique a sua resposta.

a. Procurar o maior elemento em uma sequência.

b. Modificar o conteúdo de uma variável.

c. Imprimir o conteúdo de uma sequência de elementos.

d. A partir dos dados de uma pessoa: nome, data de nascimento, sexo, determinar se a pessoa é maior de 18 anos.

2. Medidas de Complexidade

De forma geral, a complexidade de um algoritmo é determinada pela quantidade de trabalho requerido sobre um determinado tamanho da entrada. O trabalho depende diretamente do número de *operações básicas efetuadas*.

Considere o problema de estabelecer se um determinado elemento pertence a um conjunto de 100 elementos. A solução mais simples para este problema envolve uma pesquisa sequencial onde o elemento procurado é comparado a cada um dos elementos pertencentes ao conjunto. O número de comparações realizadas irá depender dos diversos cenários possíveis. A principio podemos analizar duas situações: o elemento é encontrado na primeira comparação realizada ou, o elemento não existe no conjunto. No primeiro caso seria preciso somente uma comparação, enquanto que no último caso todo o conjunto precisaria ser checado, envolvendo, portanto 100 comparações.

Situação 1:

Elemento procurado: 10

Conjunto de elementos:

10	7	50	13	99	5	22	...	2	66	29	15	88	77	0
1	2	3	4	5	6	7		94	95	96	97	98	99	100

Neste caso, na primeira comparação o logaritmo encontra o elemento procurado. A comparação é a seguinte: Elemento procurado é igual ao primeiro elemento do vetor? R-Sim. 10=10.

Situação 2:

Elemento procurado: 10

Conjunto de elementos:

10	7	50	13	99	5	22	...	2	66	29	15	88	77	0
1	2	3	4	5	6	7		94	95	96	97	98	99	100

Neste caso, o elemento é comparado com todos os elementos do vetor e não é encontrado. As comparações são as seguintes:

1) Elemento procurado é igual ao primeiro elemento do vetor? R-não. 10 5.

2) Elemento procurado é igual ao segundo elemento do vetor? R-não. 10 7.

...

100) Elemento procurado é igual ao centésimo elemento do vetor? R-não. 10 0.

A partir das situações apresentadas podemos concluir que a complexidade de um algoritmo pode ser estabelecida utilizando uma estratégia *pessimista* ou máxima, ou *otimista* ou mínima. Na estratégia pessimista, o algoritmo é analisado levando em conta o cenário mais adverso, o que normalmente irá resultar no maior tempo de execução. Este critério normalmente é o mais utilizado uma vez que estabelece uma *cota ou limite superior* no tempo requerido.

Em oposição a esta abordagem, a complexidade *otimista* ou mínima é obtida quando o problema é analisado levando em conta um cenário ideal demandando, portanto, menos tempo de execução. Pode ainda ser considerado um caso *médio* ou esperado, correspondente à média dos tempos de execução de todas as entradas de tamanho *n*. Estas estratégias podem ser utilizadas tanto para estabelecer a complexidade espacial quanto temporal dos algoritmos.

A análise para o cálculo destas medidas pode ser realizada empiricamente, isto é, com base na experiência e na observação prática, exclusivamente, sem se basear em nenhuma teoria. No entanto, a medição obtida pode ser influenciada por fatores acidentais referentes à plataforma específica, como por exemplo, a capacidade de processamento do computador sendo utilizado.

Consequentemente, a execução de um algoritmo em um determinado computador pode ter um desempenho diferente à medição resultante a partir da execução do mesmo algoritmo em um outro computador com características diferentes. Estas possíveis divergências tornam a abordagem baseada na medição empírica pouco confiável.

Atividades de avaliação

1. Determine o caso otimista e o caso pessimista a partir das seguintes especificações:

a. Encontrar o maior elemento de um vetor (desordenado) de inteiros.

b. Encontrar o maior elemento de um vetor ordenado de forma decrescente de inteiros.

c. Remover um dado elemento de um vetor de inteiros.

d. Idem anterior considerando um vetor ordenado em forma crescente.

2. Considere o problema de encontrar o maior elemento de um vetor de inteiros. O algoritmo a seguir apresenta uma solução simples para o problema.

```
int  Max (A Vetor){
  int i, Temp;

  Temp = A[0];
  for (i := 1; i < n; i++)
        if (Temp < A[i]) Temp := A[i];
  return (Temp);
}
```

a. Determine o número de comparações realizadas entre os elementos de A, considerando que A contem *n* elementos.

b. Descreva quais são as situações que representam o melhor caso, o pior caso e o caso médio para o algoritmo.

3. Comparação entre algoritmos

Como dito anteriormente, para um dado problema podem existir diversos algoritmos possíveis. Cabe ao programador analisar as possibilidades e escolher o algoritmo mais adequado utilizando como critério básico a sua complexidade.

Uma abordagem amplamente adotada para analizar a complexidade de algoritmos é baseada na análise sobre as *operações fundamentais* que compõem o algoritmo, a partir das quais é derivada uma *função custo* modelando o comportamento que o algoritmo irá a adotar de acordo com os dados de entrada fornecidos. Em particular, quando consideradas entradas suficientemente pequenas, o custo é reduzido, mesmo no caso de algoritmos ineficientes. Já para tamanhos de entrada suficientemente grandes, a escolha por um determinado algoritmo pode ser um problema crítico. Logo, a análise de algoritmos é interessante para valores grandes da entrada, ou seja, no caso de algoritmos que manipulam grandes quantidades de dados.

O estudo da complexidade consiste em determinar a *ordem de magnitude* do número de operações fundamentais realizadas pelo algoritmo, descritas a partir da definição da *função custo*. A partir desse estudo é possível realizar a *comparação* do **comportamento assintótico** através da análise dos *gráficos*[3] correspondentes.

[3] Comportamento observado de f*(n)* quando *n* tende a infinito.

A comparação entre funções com base no critério de *comportamento assintótico* consiste no estudo do crescimento de funções para valores grandes de *n*, no nosso caso, referente ao tamanho da entrada. A partir dessa análise, as funções são classificadas em ordens, onde cada ordem agrupa funções de crescimento semelhante. O algoritmo *assintoticamente mais eficiente*

é o melhor para todas as entradas. Neste contexto, uma função é considerada uma *cota assintótica superior* ou *domina assintoticamente* outra, quando cresce mais rapidamente do que outra: graficamente, o gráfico da primeira função fica por cima da segunda a partir de certo ponto *m*. No caso geral, nem sempre é possível determinar se *f* (*n*) < g (*n*).

Sejam *f* e *g* as duas funções de custo que queremos comparar:

1. Se *f* é sempre inferior a *g*, ou seja, o gráfico de *f* fica sempre por baixo do gráfico de *g*, então a escolha para o algoritmo correspondente a *f* é óbvia.

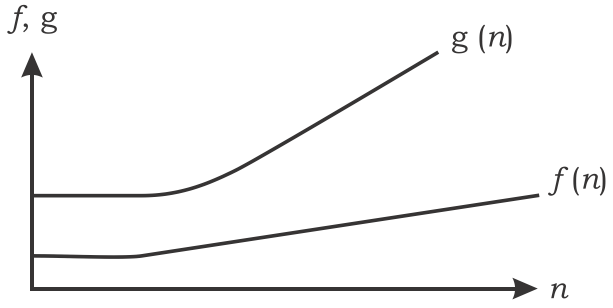


Gráfico 1 - ***f*** sempre é inferior a g. Neste caso, o algoritimo ***f*** é melhor.

2. Se *f* às vezes é inferior a *g*, e vice-versa, e os gráficos de *f* e *g* se interceptam em um número infinito de pontos. Neste caso, consideramos que há empate, e a função custo não ajuda a escolher um algoritmo.

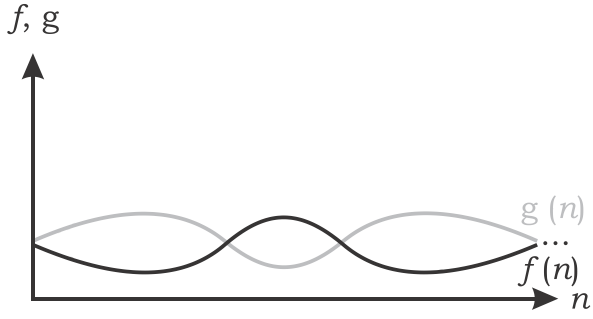


Gráfico 2 - Às vezes ***f*** é superior e às vezes g é superior e os gráficos se interceptam em um número infinito de pontos. Empate.

3. Se *f* às vezes é *g* inferior a *g*, e vice-versa, e os gráficos de *f* e *g* se cortam em um número finito de pontos. Portanto, a partir de certo valor de *n*, *f* é sempre superior a *g*, ou é sempre inferior. Neste caso, consideramos melhor aquele algoritmo que é inferior ao outro para grandes valores de *n*.

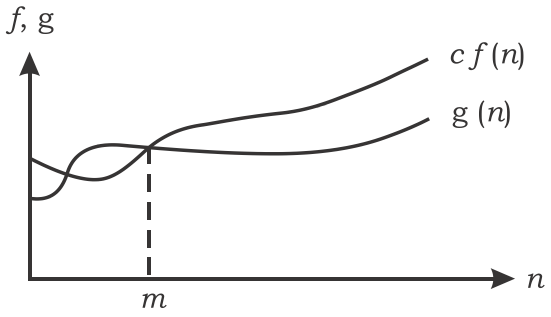
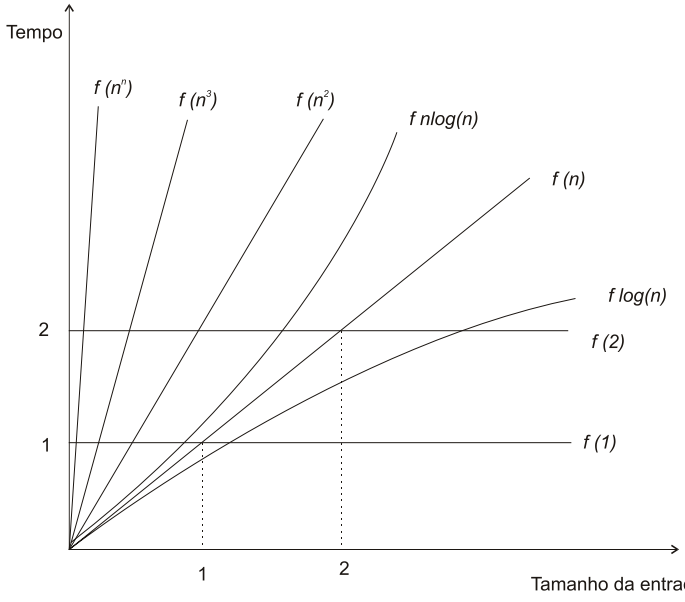


Gráfico 3 - Se os gráficos se interceptam uma quantidade finita de vezes, o melhor é aquele que menor que o outro para valores grandes de ***n.***

As funções mais comumente encontradas em análise de programas, considerando *n* como tamanho da entrada, são:

f(n) = k	*Constante*
f(n) = k . n	*Linear*
f(n) = n . log n	*Logarítmica*
f(n) = k . n2	*Quadrática*
f(n) = k . n3	*Cúbica*
f(n) = nk	*Polinomial*
f(n) = k . en	*Exponencial*

A figura a seguir representa a ordem de crescimento das funções mais comumente utilizadas na representação da complexidade dos algoritmos. Comparativamente podemos concluir que, na medida em que o tamanho da entrada *n* aumenta, a função linear *n* cresce mais rapidamente do que a função *log(n)*, que por sua vez cresce mais lentamente do que a função quadrática n^2, e assim por diante.



Sempre é possível determinar a taxa de crescimento relativo de duas funções *f(n)* e g*(n)* calculando Lim *f(n)* / g*(n)*

$x \rightarrow$

A partir deste cálculo existem os seguintes valores possíveis:

- 0, então *g(n)* é limite superior para = *f(n)*
- c, então *f(n)* e g*(n)* tem complexidades equivalentes
- infinito, então *f(n)* é limite superior para *g(n)*

No caso de funções oscilantes, como o caso de *sen(n)* ou *cos(n)*, nenhuma afirmação pode ser feita.

Atividades de avaliação

1. Considere os algoritmos A e B com complexidades:

$C_A(n) = 1000 \times n^2$

$C_B(n) = 0,1 \times n^3$

Determine a partir de qual valor de n, C_B domina assintoticamente C_A?

2. Para um determinado problema P, temos algoritmos A, B, C, e D com as seguintes complexidades.

$C_A(n) = 100 \times n \times \log n$

$C_B(n) = 1000 \times n$

$C_C(n) = 4 \times n^2$

$C_D(n) = 10 - 5 \times e^n$

Classificar os algoritmos do melhor até o pior, em termos de complexidade, sempre considerando valores grandes da variável *n* (tamanho da entrada). Justifique.

A notação utilizada para denotar a dominação assintótica foi introduzida por **Knuth** em 1968. De acordo com esta notação, a expressão *g(n) = O(f(n))* expressa que *f(n)* domina assintóticamente *g(n)*. A expressão pode ser lida da seguinte forma: *g(n)* é da ordem no máximo de f*(n)*. As definições a seguir formalizam a notação de Knuth[4].

A notação mais comumente usada para medir algoritmos é O, uma vez que determina um limite superior da complexidade:

Definição. A função custo *C(n)* é *O(F(n))* se existem constantes positivas *c* e *m* tais que:

$$C(n) \leq c \cdot F(n), \text{ quando } n \geq m$$

A definição acima afirma que existe algum ponto *m* a partir do qual *c* . *F(n)* é sempre pelo menos tão grande quanto *C(n)*. Assim, se os fatores constantes são ignorados, *F(n)* é pelo menos tão grande quanto *C(n)*.

[4] Donald Ervin Knuth, nascido em Milwaukee, em 10 de Janeiro de 1938, é um cientista computacional de renome e professor emérito da Universidade de Stanford. É o autor do livro The Art of Computer Programming, uma das principais referências da Ciência da Computação. É considerado o criador do campo de Análise de Algoritmos e fez diversas e importantes contribuições a vários ramos da teoria da computação

Como exemplo, seja $g(n) = (n + 1)^2$. Logo, $g(n)$ é $O(n^2)$, quando $m = 1$ e $c = 4$, uma vez que $(n + 1)^2 \quad 4n^2$ para n 1.

Em outras palavras, quanto dizemos que $C(n) = O(F(n))$ estamos garantindo que a função $C(n)$ não cresce mais rápido do que $F(n)$. Assim, $F(n)$ é um limite superior para $C(n)$.

De forma geral, os termos de menor peso e os fatores podem sempre ser eliminados uma vez que para valores grandes da variaveis, estes componentes se tornam despresiveis. Assim $O(4n^3 + 10n^2)$ e $O(n^3)$ são sinônimos, mas o segundo termo é mais preciso.

Se um algoritmo é $O(1)$ significa que o número de operações fundamentais executadas é limitado por uma constante.

Intuitivamente, se $g(n) = 2n^2$, então $g(n) = O(n^4)$, $g(n) = O(n^3)$ e $g(n) = O(n^2)$. Todas as respostas são tecnicamente corretas, mas a menor é a melhor resposta.

Outras definições formais

- **Limite inferior: Ω**

  A notação **Ω** é usada para especificar o limite inferior da complexidade de um algoritmo.

- **Complexidade exata: Θ**

  A notação **Θ** é usada para especificar exatamente a complexidade de um algoritmo.

- Limite superior estrito: o

  Enfim a notação ***o*** é usada para especificar que a complexidade de um algoritmo e inferior estritamente a certa função.

Atividades de avaliação

1. Suponha $g(n) = n$ e $f(n) = n^2$, demostre qual das seguintes afirmações é verdadeira:

a. $f(n) = O(g(n))$

b. $g(n) = O(f(n))$

2. Determine a ordem máxima para as seguintes funções e justifique a sua resposta detalhando os valores de c e n adequados.

a. $g(n) = 3n^3 + 2n^2 + n$

b. $h(n) = n^2 + n + 1$

4. Operações com a notação *O*

Algumas operações que podem ser realizadas com a notação O são apresentadas a seguir:

> *f(n) = O(f(n))*
> *c . O(f(n)) = O(f(n)), onde c é uma constante*
> *O(f(n)) + O(f(n)) = O(f(n))*
> *O(O(f(n))) = O(f(n))*
> *O(f(n)) + O(g(n)) = O(max (f(n), g(n)))*
> *O(f(n)) . O(g(n)) =O(f(n) . g(n))*
> *f(n). O(g(n)) = O(f(n) . g(n))*

Estas operações foram demostradas matematicamente na literatura e tem aplicação direta para o cálculo do tempo de execução de (trechos de) programas. Por exemplo, a regra da soma, *O(f(n))* + *O(g(n))* pode ser utilizada para calcular o tempo de execução de uma sequência de trechos ou sentenças de programas. Aplicando essa regra, somente será considerado o trecho que tiver atribuído o custo máximo dentre os trechos ou sentenças considerados no sumatório.

Aplicando a abordagem de *dividir para conquistar,* a complexidade de um algoritmo pode ser determinada a partir da complexidade das suas partes. De forma geral, a análise da complexidade é feita em forma particular para cada algoritmo, mas existem conceitos gerais que só dependem das estruturas algorítmicas ou comandos de controle utilizados no algoritmo:

1) Sequência ou conjunção é um tipo de comando que, no fluxo lógico do programa, é executado e o controle passa para o próximo comando na sequência.

- A análise da complexidade neste caso envolve a aplicação da regra da soma para a notação *O* com base na complexidade dos comandos.

2) Seleção ou disjunção é um tipo de comando que, no fluxo de execução permite que desvios possam ser feitos a partir do resultado da avaliação de uma condição, executando um bloco de comandos e outros não.

- A análise sempre é feita considerando o pior caso. Consequentemente, na avaliação da complexidade é considerado o desvio que envolve a execução do bloco de comandos mais custoso, a partir da avaliação da condição. Adicionalmente, a avaliação da condição consome tempo constante

3) Repetição é um tipo de comando que, no fluxo de execução do programa, permite que um bloco de comandos seja repetido enquanto uma condição é satisfeita.

- Além da checagem da condição *(O(1))*, o custo de um comando de repetição envolve o custo do bloco envolvido, multiplicado pelo número de vezes que será executado, no pior caso. Vale ressaltar que pode existir aninhamento de comandos de repetição. Neste caso a análise é feita de dentro para fora. O laço pode ser definido *(for)* ou indefinido *(while)*.

4) Um comando de atribuição possibilita que um valor possa ser atribuído a uma variável.

- Um comando de atribuição consome tempo constante.

Atividades de avaliação

Por exemplo, dado um vetor *v* de números inteiros de tamanho *n*, retornar o maior elemento deste vetor.

```
1: int Maximo (int v[], int n){
2:          int i: n;
3:          int max;
4:
5:          if (n = 0) printf (" Vetor vazio ");
6:          else {
7:                    max = v[0];
8:                    for (i = 1; i < n; i++)
9:                              if (v[i] > max)max = v[i];
10:         }
11:         return max;
12: }
```

O número *n* de elementos do vetor representa o tamanho da entrada. O programa contém um comando de iteração (8) que contém um comando de desição que, por sua vez, contém apenas um comando de atribuição (9). O comando de atribuição requer tempo constante *O(1)* para ser executado, assim como a avaliação do comando de desição. Considerando o pior caso, o comando de atribuição sempre será executado.

O tempo para incrementar o índice do laço e avaliar sua condição de terminação também é *O(1)*, e o tempo combinado para executar uma vez o

laço composto pelas linhas (8) e (9) é *O(max(1, 1))* = *O(1)*, conforme a regra de soma para a notação O, e considerando que o número de iterações do laço é *n - 1*, então o tempo gasto no laço (8) é *O((n - 1) x 1)* = *O(n - 1)*, conforme a regra do produto para a notação *O*.

O algoritmo é estruturado com base em um comando de desição, cuja condição é checada em tempo constante, da mesma forma que a edição da mensagem de erro *(O(1))*. Por outro lado, utilizando a regra da soma novamente, temos que a execução das linhas (7), (8) e (9) consome *O(max(1, n - 1))* = *O(n - 1)*. No pior caso temos que a execução do bloco (5) consome *O(n - 1)*. Finalmente, o comando (11) é executado em tempo *O(1)*. Aplicando novamente a regra da soma entre (5) e (11) temos que *O(max(n – 1, 1))* = *O(n - 1)*. Portanto a complexidade temporal do algoritmo *Maximo* é linear.

Simplificando a análise, a quantidade de trabalho do algoritmo pode ser determinada pela quantidade de execuções da operação fundamental. No entanto, pode existir mais de uma operação fundamental com pesos diferentes. Neste caso, a regra de soma para a notação *O* pode ser aplicada.

Atividades de avaliação

Considere o algoritmo buscaBinaria descrito a seguir:

```
int buscaBinaria (int array[], int chave, int n){
    int inf = 0;          //Limite inferior
    int sup = n - 1;      //Limite superior
    int meio;
    while (inf <= sup) {
        meio = (inf + sup) / 2;
        if (chave == array[meio])
            return meio;
        else if (chave < array[meio])
                sup = meio - 1;
            else inf = meio + 1;
    }
    return -1;            // não encontrado
}
```

1. Descreva o funcionamento do algoritmo *buscaBinaria*. Qual situação representa o pior caso? e o melhor?

2. Considerando a execução do algoritmo *buscaBinaria* em um vetor com 15 elementos, quantas repetições (número de passos) são necessários para o algoritmo detectar que uma determinada chave de busca não se encontra no vetor.
3. Determine a complexidade temporal do algoritmo *buscaBinaria* analisando passo-a-passo a complexidade de cada comando.

Síntese do capítulo

Neste capítulo foi apresentado um estudo introdutório sobre os fundamentos da teoria sobre complexidade de algoritmos. Esta teoria é de fundamental importância uma vez que possibilita determinar a melhor solução para um determinado tipo de problema, assim como também elaborar projetos de algoritmos cada vez mais eficientes.

Referências

CORMEN T. H., LEISERSON C. E., RIVEST R. L., STEIN C. (2001). **Introduction to Algorithms**. McGraw-Hill e The Mit Press.

KNUTH D. E. (1968). **The Art of Computer Programming**, Vol. 1: Fundamental Algorithms. Addison-Wesley.

KNUTH D. E. (1971). **Mathematical Analysis of Algorithms**. Procedings IFIP Congress 71, vol. 1, North Holland, 135-143.

WIRTH N. (1986). **Algorithms and Data Strutures**. Prentice-Hall.

ZIVIANI N. (2005). **Projeto de Algoritmos com implementações em Pascal e C**, 2da. Edição. Thomson.

Capítulo 2

Representações de Dados

Objetivos

- Neste capítulo é descrito o processo de abstração seguindo para a modelagem e desenvolvimento de uma solução computacional a partir de um problema do mundo real. Neste contexto, os conceitos de tipos de dados, estruturas e tipos abstratos são introduzidos, ressaltando sua importância para a adequada modelagem, manipulação e representação na memória do computador.
-

1.Modelagem Conceitual

Para que um problema do mundo real possa ser resolvido computacionalmente é necessário utilizar métodos e técnicas que possibilitem a modelagem adequada do problema, de forma que possa ser interpretado e processado pelo computador para gerar uma solução.

Os modelos são utilizados para representar o mundo real de forma simplificada, com o objetivo de facilitar o gerenciamento da complexidade. Um modelo reflete os aspectos considerados importantes para o desenvolvimento da aplicação, deixando em um segundo plano, os aspectos que não são relevantes.

A modelagem de situações do mundo real é alcançada através de um processo de abstração, a partir do qual, somente as propriedades relevantes para a aplicação são consideradas no modelo.

Entidade observada *Entidade representada*

O *processo de abstração* envolve a observação das entidades presentes no domínio do problema para a correspondente representação no domínio computacional.

No nível mais baixo de abstração, a representação dos dados a nível de máquina acontece de acordo a padrões de *bits*[5] e ***bytes*** de memória. No entanto, as linguagens de programação modernas possibilitam que o programador possa trabalhar com objetos relativamente complexos em um nível mais alto de abstração, tornando transparente ao desenvolvedor a manipulação dos dados ao nível da sua representação interna no computador. Esta facilidade é alcançada através da utilização de uma variedade de *tipos de dados*.

[5] Bit significa dígito binário, do ingles *BInary digiT*. Um bit é a menor unidade de informação que pode ser armazenada ou transmitida. Um bit pode assumir somente 2 valores: 0 ou 1, verdadeiro ou falso. O conjunto de 8 bits é denominado de Byte.

2. Tipo de dados

O tipo de dados associado a uma variável, numa linguagem de programação, define o conjunto de valores que a variável pode assumir. Isto é, a declaração da variável como sendo de um tipo específico determina:

1. A quantidade de *bits* que devem ser reservados na memória.
2. Como o dado representado por esse padrão de *bits* deve ser interpretado (p.e., uma cadeia de *bits* pode ser interpretada como sendo um inteiro ou real).

Os tipos têm geralmente associações com valores na memória ou variáveis. Consequentemente, tipos de dados podem ser vistos como métodos para interpretar o conteúdo da memória do computador, o que pode variar conforme o sistema operacional e a linguagem de implementação.

O sistema operacional (SO) é um programa ou conjunto de programas responsável por gerenciar todos os recursos do sistema, tais como: comandos do usuário, arquivos, memória, etc.

> Na sua maioria, as linguagens de programação exigem que um comando declarativo que define uma variável especifique também o tipo de dados associado à variável. Estas linguagens são chamadas de *fortemente tipadas*. Em contrapartida, as linguagens *fracamente tipadas* permitem que a definição do tipo correspondente a uma variável possa ser determinada dinamicamente, em tempo de execução.

Os tipos de dados podem ser classificados em dois grupos: *primitivos* ou básicos e *compostos* ou estruturados.

Os tipos de dados primitivos são fornecidos pela linguagem de programação como um bloco de construção básico. Dependendo da implementação da linguagem, os tipos primitivos podem ou não possuir correpondência direta com objetos na memória[6]. Exemplos de tipos primitivos comuns são: inteiro, real, caractere, *booleano*, dentre outros.

[6] A memória é o dispositivo que permite a um computador armazenar dados de forma temporária ou permanente.

Além de estabelecer um esquema predeterminado de armazenamento, a definição de um tipo de dados primitivo estabelece também um conjunto de operações predefinidas sobre aquele tipo. Consequentemente, a definição

de uma variável como instância de um desses tipos determina o conjunto de operações que podem ser realizadas utilizando essas variáveis. Por exemplo, ao considerar variáveis do tipo primitivo inteiro (int), as operações permitidas se traduzem nos operadores aritméticos válidos para os valores desse tipo, no caso: soma (+), subtração (-), multiplicação (*), divisão inteira (DIV) e resto da divisão inteira (MOD). Estas operações são implementadas de forma nativa por qualquer linguagem de programação, e seu desenvolvimento fica transparente ao usuário.

Dada a complexidade das entidades do domínio do problema a serem modeladas e representadas no universo computacional, o fosso semântico[7] (*gap* semântico) envolvendo a descrição de alto nível de uma entidade (domínio do problema), e a descrição de baixo nível (domínio da solução) pode ser inconciliável. Neste contexto, a possibilidade de definir tipos de dados que possibilitem a representação destas entidades de forma mais próxima da realidade facilita o processo de abstração, assim como contribui para a redução do fosso semântico entre ambos os domínios.

[7] O fosso semântico representa a diferença entre uma descrição de alto nível e outra de baixo nível relativa a uma mesma entidade.

O programador pode definir *tipos de dados próprios* que mais correspondam às necessidades de suas *aplicações*. Linguagens de programação atuais permitem aos programadores definir tipos de dados *adicionais*, utilizando os tipos primitivos e as estruturas como blocos construtivos. Por exemplo, para definir a variável *Empregado* com esta estrutura heterogênea em C seria:

```
struct Empregado{
        char Nome[9];
        int Idade;
        float Qualificação;
};
```

No entanto, se a estrutura for muito frequente, o programa pode ficar volumoso e difícil de ser lido. O método mais adequado consiste em descrever a estrutura de dados correspondente uma única vez, associando-a a um nome descritivo, e utilizar esse nome todas as vezes que for necessário.

```
typedef struct {
        char Nome[9];
        int Idade;
        float Qualificação;
} TipoEmpregado;
```

Esta declaração define um novo tipo denominado *TipoEmpregado* que pode ser usado para declarar variáveis da mesma forma como um tipo primitivo.

TipoEmpregado Empregado;

A partir desta definição é possível gerar as correspondentes variáveis instância, as quais irão assumir valores nos atributos, dependendo do tipo. Por exemplo, valores válidos para uma variável do tipo *Empregado* podem ser:

Nome: João Soares
Idade: 25 anos
Qualificação: 8.58

Um tipo definido pelo usuário é essencialmente um *modelo* usado para construir *instâncias* de um dado tipo, que descreve todas as características que todas as instâncias deste tipo devem assumir mas não constitui, ele próprio, uma ocorrência real deste tipo.

A *forma conceitual dos dados* é materializada pela *estrutura de dados* utilizada na implementação do tipo. Uma estrutura de dados é uma forma particular de se implementar um tipo, e é construída dos tipos primitivos e/ou compostos de uma linguagem de programação, e por este motivo são chamados de tipos *compostos* ou estruturados.

Um tipo composto envolve um conjunto de campos e membros organizados de forma coerente, onde o tamanho total da estrutura corresponde à soma dos tamanhos dos campos constituintes. Exemplos de tipos estruturados são: registros, vetores, matrizes, arquivos, árvores, dentre outros.

A escolha por uma estrutura de dados é determinante na qualidade e esforço requerido para o desenvolvimento da solução. As estruturas de dados e algoritmos são escolhidos com base em critérios diversos, tais como desempenho, restrições de plataforma de *hardware e software*, capacidade do equipamento, volume de dados, etc. As estruturas de dados dividem-se em homogêneas e heterogêneos. As estruturas homogêneas são conjuntos de dados formados por componentes do mesmo tipo de dado (p.e., vetores, matrizes, pilhas, listas, etc.). Em contrapartida, as estruturas heterogêneas são conjuntos de dados formados por componentes pertencentes a tipos de dados diferentes (p.e., registros). A escolha de uma estrutura de dados apropriada pode tornar um problema complicado em uma solução bastante trivial.

Diferentemente do que acontece na definição de tipos de dados básicos, onde um conjunto de operações de manipulação é fornecido pela linguagem de progamação (soma, subtração,...), no caso dos tipos estruturados definidos pelo usuário apenas novos esquemas de armazenamento são definidos. Isto significa que, a principio, não são fornecidos meios para definir as operações a serem executadas sobre instâncias de tais estruturas. Consequentemente, algoritmos de manipulação precisam ser desenvolvidos de forma a possibilitar a correta utilização e acesso às novas estruturas definidas.

1. Defina os conceitos de tipo de dado básico e tipo de dado definido pelo usuário.
2. Cite como exemplos tipos de dados básicos que você conhece e detalhe suas características e as operações permitidas sobre esses tipos.
3. Defina um tipo de dado estruturado que descreva de forma adequada e completa as seguintes informações:
   a. Livro
   b. Círculo
   c. Filme
   d. Pessoa
   e. Aluno
   f. Item de estoque
   g. Conta bancaria

3. Tipo abstratos de dados

Um tipo de dado definido pelo usuário, incrementado com a definição e implementação das operações necessárias para a sua manipulação, constitui um Tipo Abstrato de Dados (TAD), conceito central no contexto do Paradigma Orientado a Objetos[8]. A utilização de TADs permite que linguagens de programação de propósito geral sejam personalizadas para um domínio de aplicação mais específico. Uma vez definidos, podem ser empregados como primitivas da linguagem, e possibilitam o desenvolvimento de componentes de software reusáveis e extensíveis.

[8] O paradigma orientado a objetos envolve um conjunto de técnicas, métodos e ferramentas para análise, projeto e implementação de sistemas de software baseado na composição e interação de componentes de software denominados de objetos.

TADs estendem a noção de tipo de dado (estrutura de dados + operações) com base na utilização de técnicas de ocultamento da informação referente à estrutura de dados utilizada e à implementação das operações definidas. Este objetivo é alcançado através de uma clara separação entre **interface** e implementação.

A seguir é apresentado como exemplo a definição do tipo abstrato Retângulo. Um retângulo pode ser definido pela sua largura e altura. A especificação do tipo retângulo pode ser definida como:

```
typedef struct {
        float altura, largura;
} TipoRetangulo;
```

A partir destas informações é possível calcular sua área e perímetro. Portanto, além das operações de consulta sobre as informações do retângulo, as operações de cálculo de área e perímetro são requeridas.

O construtor é um método distinguido que tem como função principal a de instanciar o objeto corretamente, para seu uso posterior.

Desta forma, a interface do tipo abstrato retângulo inclui as seguintes operações de manipulação.

```
//inicia valores do retangulo
void inicia_retangulo (float a, float l);

// atribui um valor à altura
void InitAltura (float a);

// retorna o valor da altura
float Altura (void);

// atribui um valor à largura
void InitLargura (float l);
// retorna o valor da largura
float Largura;

// calcula o valor do perímetro
float Perimetro;
```

Protótipos na linguagem C define o cabeçalho das funções.

```
// calcula o valor da área
float Area (protótipos das operações);
```

Finalmente, todos os métodos implementados na linguagem de programação. Exemplos da implementação de alguns dos métodos são apresentados a seguir.

```
void inicia_retangulo (float a, float l)
altura = a;
  largura = l;
}

float Altura () {
  return altura;
}

float Perimetro () {
  return  2 * (altura + largura);
}
```

O projeto[9] de um tipo abstrato é uma tarefa difícil, pois este deve ser idealizado de forma a possibilitar a sua utilização por parte de terceiros, favorecendo o reuso e agilizando o processo de desenvolvimento de *software*. A atividade de projeto envolve a escolha de operações adequadas, delineando seu comportamento consistente, de forma que estas possam ser combinadas para realizar funções mais complexas, a partir de operações simples.

[9] A fase de projeto produz uma descrição computacional do que o software deve fazer, e deve ser coerente com as especificações geradas na análise, de acordo com os recursos tecnológicos existentes.

No intuito de aumentar o reuso das operações, estas devem ser definidas de forma **coesa**, e ter um comportamento coerente, com um propósito específico e evitando considerar diversos casos especiais em um mesmo código.

O conjunto de operações que integram a interface do TAD deve oferecer todas as operações necessárias para que os usuários possam manipular a estrutura adequadamente. Um bom teste consiste em checar se todas as propriedades do objeto de um determinado tipo podem ser acessadas.

A escolha pela representação mais adequada ao problema envolve uma análise aprofundada, uma vez que cada representação possível possui diferentes vantagens e desvantagens.

Atividades de avaliação

1. Defina os conceitos de Tipo Abstrato de Dados. Estabeleça o relacionamento que vincula ambos conceitos.
2. Defina TADs para os tipos de dados estruturados para as entidades listadas a seguir especificando detalhadamente a interface completa. A interface deve incluir todas as operações necessárias para a correta manipulação do tipo, dentre elas os métodos de inicialização, modificação, e consulta.
   a. Livro
   b. Círculo
   c. Filme
   d. Pessoa
   e. Aluno
   f. Item de estoque
   g. Conta bancária
3. Implemente dois TADs do exercício anterior utilizando a linguagem de programação da sua escolha ou pseudo-código próximo da linguagem.

4. Critérios para a escolha da estrutura de dados adequada

Como dito anteriormente, a interface do TAD independe da implementação e, portanto, diferentes estruturas de dados podem ser utilizadas como esquema de armazenamento na memória[10] para seus atributos. A partir do esquema utilizado, os dados podem ser armazenados e recuperados.

A forma em que a alocação de memória acontece, pode ser um fator determinante para a escolha de uma determinada estrutura de dados.

[10] A memória é o dispositivo que permite a um computador guardar dados de forma temporária ou permanente.

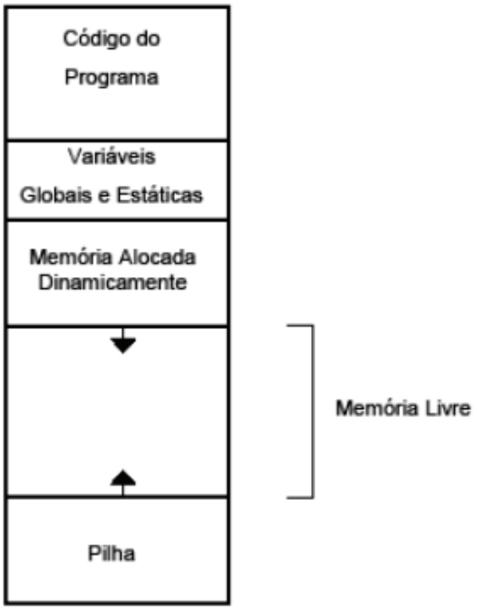
4.1. Uso da memória

Informalmente, podemos dizer que existem três maneiras de reservarmos espaço de memória para o armazenamento de informações:

- Uso de variáveis globais (e estáticas). O espaço reservado para uma variável global existe enquanto o programa estiver sendo executado.
- Uso de variáveis locais. Neste caso o espaço fica reservado apenas enquanto a função que declarou a variável está sendo executada, sendo liberado para outros usos quando a execução da função termina. Por este

motivo, a função que chama não pode fazer referência ao espaço local da função chamada. As variáveis globais ou locais podem ser simples ou vetores, sendo que no caso dos vetores, é preciso informar o número máximo de elementos ou tamanho do vetor. A partir do tamanho informado, o compilador reserva o espaço correspondente.

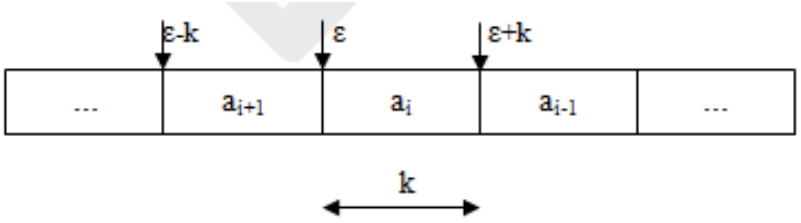
- Requisições ao sistema em tempo de execução. Este espaço alocado dinamicamente permanece reservado até que explicitamente seja liberado pelo programa. Uma vez que o espaço é liberado fica disponível para outros usos. Se o espaço alocado não for liberado explicitamente pelo programa, este será automaticamente liberado quando a sua execução terminar. A seguir é ilustrada esquematicamente a alocação da memória pelo sistema operacional.



Quando requisitamos ao sistema operacional para executar um determinado programa, o código em linguagem de máquina do programa deve ser carregado na memória. O sistema operacional reserva também o espaço necessário para armazenarmos as variáveis globais (e estáticas) utilizadas ao longo do programa. O restante da memória é utilizado pelas variáveis locais e pelas variáveis alocadas dinamicamente enquanto o programa está executando.

Cada vez que uma determinada função é chamada, o sistema reserva o espaço necessário para as variáveis locais da função. Este espaço pertence à pilha de execução e, quando a função termina, é desempilhado e liberado. A parte da memória não ocupada pela pilha de execução pode ser requisitada dinamicamente. Se ao longo de diversas chamadas a função, a pilha cresce atingindo o espaço disponível existente, dizemos que ela "estourou" e o programa é abortado com erro. Similarmente, se o espaço de memória livre for menor que o espaço requisitado dinamicamente, a alocação não é feita e o programa pode prever um tratamento de erro adequado (por exemplo, podemos imprimir a mensagem "Memória insuficiente" e interromper a execução do programa).

Como mencionado anteriormente, a alocação de memória pode acontecer em tempo de compilação (estática) ou em tempo de execução (dinâmica). No caso da alocação de memória em forma estática, o espaço destinado para o armazenamento dos dados possui um tamanho fixo, que não pode ser modificado ao longo da execução do programa. Adicionalmente, a alocação de memória acontece em espaços contíguos, isto é, um do lado do outro. Exemplos de alocação estática de memória são variáveis globais e vetores. No caso do vetor, a partir de certo endereço ε que armazena o primeiro elemento a_1 do vetor, os elementos subsequentes podem ser acessados diretamente incrementando em *k* o endereço inicial do vetor, onde *k* é o tamanho de memória ocupado por cada elemento do vetor.



Em contrapartida, no caso de alocação dinâmica, a memória é gerenciada sob demanda, ou seja, os espaços de memória são alocados e desalocados dependendo da necessidade, durante a execução do programa. Consequentemente, a alocação não acontece necessariamente de forma contígua (ou sequencial), podendo, os dados, ficarem esparsos na memória do computador. A partir desta configuração onde os dados são armazenados de forma não sequencial, o acesso é alcançado através de variáveis do tipo ponteiro[11], indicando o endereço de memória correspondente. A seguir é ilustrada esquematicamente a distribuição dos elementos integrantes de uma cadeia ao longo da memória. Na primeira coluna é indicado o endereço correspondente ao conteúdo. A coluna *ponteiro* indica o endereço do próximo elemento na cadeia. Note que o segundo elemento não ocupa o endereço consecutivo ao a_1.

[11] O ponteiro ou apontador é um tipo de dado que armazena o endereço de um outro dado. A partir do ponteiro, o dado que se encontra no respectivo endereço pode ser acessado e manipulado.

Endereço	Conteúdo	Ponteiro	Observação
L=3FFB	a_1	1D34	Primeiro elemento acessado a partir de L
1D34	a_2	BD2F	
BD2F	a_3	AC13	
1500	a_n-2	16F7	
16F7	a_n-1	5D4A	
5D4A	a_n	null	Último elemento da cadeia. O endereço null indica que o elemento não tem sucessor.

A partir das características de cada uma das abordagens para a alocação dos dados, vantagens e desvantagens podem ser estabelecidas.

Alocação estática:

- Vantagem: Possibilita acesso direto ao local da memória, uma vez que os dados se encontram armazenados de forma contígua ou sequencial. Consequentemente, em alguns casos, as operações de busca podem ter custo constante *O(1)*.
- Desvantagem: É preciso determinar em tempo de codificação, quanto espaço é necessário. Esta estimativa pode ser difícil de ser estabelecida, e está sujeita a flutuações ao longo da execução. Consequentemente o espaço pode resultar insuficiente ou pode ter sido sobreestimado. Por outro lado, a alocação sequencial na memória prejudica as operações de inserção e remoção, uma vez que a sequencialidade dos dados precisa ser preservada. Com isso, no pior caso, o custo destas operações pode ser de *O(n)* por conta dos deslocamentos necessários para abrir ou fechar espaços para inserção e remoção, respectivamente.

Alocação dinâmica:

- Vantagem: não é necessário fixar o tamanho da estrutura a *priori*, uma vez que a alocação de memória é feita sob demanda em tempo de execução. Com isso é evitado o desperdício de espaço, e o risco de ficar sem espaço é reduzido. Consequentemente, não existe restrição encima do número de inserções e remoções. Adicionalmente, estas operações não requerem de nenhum esforço adicional uma vez que envolvem somente o ajuste dos ponteiros já que os elementos se encontram esparsos na memória.
- Desvantagem: o gerenciamento dos dados diretamente da memória pode ser trabalhoso e propenso a erros. Como conseqüência da não linearidade na alocação dos dados, o acesso a um determinado elemento *i* torna necessário o percurso pelos *i – 1* elementos anteriores na sequência. Esta propriedade, que caracteriza o acesso sequencial aos dados, torna a operação de busca por um elemento de custo *O(n)*.

Síntese do capítulo

Neste capítulo foi apresentado o conceito de abstração e seus níveis, e a sua importância no processo de modelagem. Tipos de dados básicos e estruturados foram definidos neste contexto, como meios de representar e interpretar as informações manipuladas pela aplicação. Adicionalmente, o conceito de

tipo abstrato é estabelecido como um mecanismo de extensão e customização de linguagens de programação, no intuito de facilitar o desenvolvimento de sistemas. TADs são caracterizados por suas operações, as quais são encapsuladas junto à sua estrutura, sendo acessíveis exclusivamente através da interface especificada, garantindo independência da implementação utilizada. A independência de representação torna possível alterar a representação de um tipo sem que seus clientes sejam afetados. Finalmente, noções de gerência de memória foram introduzidas, focando nas principais características que influenciam na escolha pela alocação estática ou dinâmica da memória. Uma análise dos fatores que contribuem na tomada de decisão foi apresentada.

Referências

CORMEN T. H., LEISERSON C. E., RIVEST R. L., STEIN C. (2001). **Introduction to Algorithms**. McGraw-Hill e The Mit Press.

TENENBAUM A. M., LANGSAM Y., AUGENSTEIN M. J. (1995). **Estruturas de Dados Usando C**, Makron Books/Pearson Education.

WIRTH N. (1986). **Algorithms and Data Structures**. Prentice-Hall.

ZIVIANI N. (2005). **Projeto de Algoritmos com implementações em Pascal e C**, 2da. Edição. Thomson.

Capítulo 3

Listas

Objetivos

- Existem certas estruturas clássicas que se comportam como padrões uma vez que são utilizadas na prática em diversos domínios de aplicação. Neste capítulo é apresentada a estrutura de dados Lista e o correspondente Tipo Abstrato, detalhando a sua interface e apresentando duas implementações: vetores e ponteiros. Variantes do tipo abstrato listas, na forma de Pilhas e Filas, de ampla utilização prática, são descritas.

Introdução

Um conjunto de elementos pode ser intuitivamente representado através de uma lista linear. Listas são estruturas extremamente flexíveis que possibilitam uma ampla manipulação das informações uma vez que inserções e remoções podem acontecer em qualquer posição.

Uma lista pode ser definida como uma estrutura linear[12], finita cuja ordem é dada a partir da inserção dos seus elementos componentes. As listas são estruturas compostas, constituídas por dados de forma a preservar a relação de ordem linear entre eles. Cada elemento na lista pode ser um dado primitivo ou arbitrariamente complexo.

Em geral, uma lista segue a forma a_1, a_2, a_3, ..., a_n, onde n determina o tamanho da lista. Quando $n = 0$ a lista é chamada *nula* ou *vazia*. Para toda lista, exceto a nula, a_{i+1} segue (ou sucede) a_i ($i < n$), e a_{i-1} precede a_i ($i > 1$). O primeiro elemento da lista é a_1, e o último a_n. A posição correspondente ao elemento a_i na lista é i. A lista pode ser representada visualizando-se um vetor, por exemplo.

As características básicas da estrutura de dados lista são as seguintes:

- Homogênea. Todos os elementos da lista são do mesmo tipo.
- A ordem nos elementos é decorrente da sua estrutura linear, no entanto os elementos não estão ordenados pelo seu conteúdo, mas pela posição ocupada a partir da sua inserção.

[12] Uma estrutura é dita de linear uma vez que seus elementos componentes se encontram organizados de forma que todos, a exceção do primeiro e último, possuem um elemento anterior e um posterior, somente.

- Para cada elemento existe anterior e seguinte, exceto o *primeiro*, que não possui anterior, e o *último*, que não possui seguinte.
- É possível acessar e consultar qualquer elemento na lista.
- É possível inserir e remover elementos em qualquer posição.

1. Definição do TAD Lista

Como apresentado na parte anterior, a definição de um Tipo Abstrato de Dados (TAD) envolve a especificação da interface de acesso para a manipulação adequada da estrutura, a partir da qual são definidas em detalhe, as operações permitidas e os parâmetros requeridos. O conjunto de operações depende fortemente das características de cada aplicação, no entanto, é possível definir um conjunto de operações mínimo, necessário e comum a todas as aplicações.

Interface do TAD Lista

```
// Cria uma lista vazia
Lista Criar ()

//insere numa dada posição na lista.
int Inserir (Lista l, tipo_base  dado, corrente pos)

// Retorna o elemento de uma dada posição
tipo_base  consultaElemento (Lista l, corrente pos);

// Remove o elemento de uma determinada posição
int Remover (Lista l, corrente pos);

// Retorna 1 a lista está vazia, ou 0 em caso contrario.
int Vazia (Lista l);

// Retorna 1 se a lista está cheia, ou 0 em caso contrario.
int Cheia (Lista l);

// Retorna a quantidade de elementos na lista.
int Tamanho (Lista l);

// Retorna o próximo elemento na lista a partir da posição corrente.
corrente proximoElemento (Lista l, corrente pos);
```

```
/*Busca por um determinado elemento e retorna sua posição corrente, ou -1 caso não seja encontrado.*/
corrente Busca (Lista l, tipo_base  dado);
```

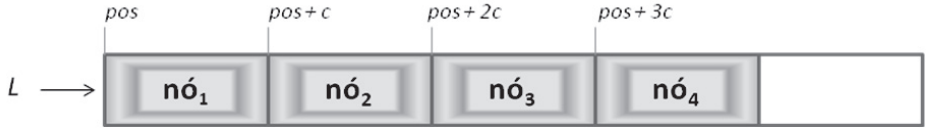
Uma vez definida a interface, esta pode ser implementada utilizando uma representação dos dados adequada. Existindo mais de uma estrutura adequada, a escolha depende principalmente das necessidades e características dos dados a serem manipulados pela aplicação.

A seguir, o Tipo Abstrato Lista é implementado utilizando duas estruturas de dados comumente utilizadas e adequadas às necessidades, cada uma com vantagens e desvantagens particulares.

1.1. Implementação do TAD Lista usando alocação estática

Na implementação de lista adotando alocação de memória estática os elementos componentes são organizados em posições contíguas de memória utilizando *arranjos* ou vetores.

Vantagens e desvantagens desta estrutura foram discutidas na parte 3. Em particular, a utilização de vetores se torna adequada no caso em que existe uma clara noção do tamanho da entrada a ser processada, e uma perspectiva que indica que as aplicações que irão utilizar o TAD não estarão executando muitas operações de inserção e remoção que possam vir a alterar significativamente o tamanho preestabelecido. A estruturação da lista utilizando alocação estática é apresentada graficamente a seguir. Note que, a partir do endereço correspondente ao primeiro elemento no vetor (*pos*), e conhecendo o tamanho (*c*) de cada componente na lista, é possível calcular o endereço na memória de qualquer elemento armazenado no vetor. Isso garante o acesso direto aos elemento em *O(1)*.



A seguir é apresentada a definição da estrutura de dados e implementação[13] das operações definidas na interface utilizando alocação estática de memória através da definição de vetores. Considerando a utilização de alocação estática de memória, o tamanho da estrutura de dados precisa ser determinado em tempo de compilação.

[13] A notação utilizada na implementação é próxima à linguagem C.

```
#define tamanho
```

Como o protótipo da lista é definido de modo a ser implementado usando vetor ou outro recurso, torna-se necessário definir um tipo que possa ser utilizado como elemento que é acessado nas operações (ou corrente). No caso desta implementação inicial, utilizando vetor, os elementos são acessados através de suas posições no vetor, sendo que estas posições são representadas como interiores que iniciam em 0 (zero) na primeira posição e seguem até n-1 (tamanho do vetor - 1). Portanto, torna-se necessário definir o tipo corrente como inteiro.

```
typedef int corrente;
```

Uma lista é um tipo de dado que estrutura elementos cujo tipo pode ser arbitrariamente complexo, envolvendo inclusive, a utilização de outros TADs. A definição a seguir especifica o tipo base dos elementos da lista como inteiros.

```
typedef int tipo_base;
```

Os elementos da lista são organizados em um vetor, de tamanho predefinido. Adicionalmente, um atributo contendo a quantidade de elementos da lista (*quant_Elem*) é incluído na estrutura no intuito de tornar mais fácil e ágil o acesso aos elementos e possibilitar o controle do crescimento da estrutura.

```
typedef struct {
        tipo_base v[tamanho];
        int quant_Elem;
} no_Lista;
```

A informação relativa à quantidade de elementos existente na lista pode ser útil em diversas situações uma vez que, conhecendo a posição na memória (endereço) do primeiro elemento da lista, é possível calcular o endereço do último elemento e, consequentemente, da primeira posição disponível. Isto é possível por conta da propriedade de armazenamento contíguo propiciada pela estrutura de dados.

Na linguagem C, a posição que corresponde ao primeiro elemento do vetor corresponde ao índice i = 0. Em outras linguagens, como por exemplo Pascal, o primeiro elemento no vetor se encontra na posição i=1.

Finalmente a lista é definida como um ponteiro à estrutura de dados onde os elementos são agregados.

```
typedef no_Lista *Lista;
```

A criação de uma lista inicialmente vazia envolve a definição do ponteiro correspondente, apontando a um endereço de memória reservado com tamanho adequado para o armazenamento da estrutura, em particular, o primeiro nó representando a cabeça da lista. O tamanho da lista é inicializado em zero uma vez que inicialmente não contém elementos.

```
Lista Criar () {
        Lista l = (Lista) malloc (sizeof (no_Lista));
        if (l != NULL){
```

```
                l -> quant_Elem = 0;
        return (l);
        }
        else printf ("Não existe memória suficiente");
        return;
}
```

Normalmente, a lista precisa ser percorrida de forma a realizar algum tipo de processamento sobre os dados que a compõem. Consequentemente se torna necessário um mecanismo que possibilite checar no momento em que não seja possível processar mais nenhum elemento. O método *ultimoElemento* é responsável por fazer esta checagem.

```
int ultimoElemento (Lista l, corrente pos) {
        if (pos + 1 == l -> quant_elem) return (1)
        else return (0);
}
```

A execução de uma operação de remoção requer a existencia de no mínimo um elemento na lista. A função *Vazia* é utilizada para informar se há elementos no vetor, retorna o valor 1 no caso da lista se encontrar vazia, e 0 em caso contrário.

```
int Vazia (Lista l) {
        if (l -> quant_Elem == 0) return (1)
        else return (0);
}
```

Outra operação que pode ser de utilidade é a checagem pelo caso em que a estrutura que armazena a lista possa estar cheia, uma vez que esta situação pode inviabilizar a inserção de um novo elemento. Este método é de fundamental importância, principalmente no caso de utilização de alocação de memória estática onde a tentativa de inserção de um novo elemento pode acarretar o estouro da memória, fazendo com que o programa termine com erro.

```
int Cheia (Lista l) {
        if (l -> quant_Elem == tamanho) return (1)
        else return (0);
}
```

Uma função que pode ser definida para auxiliar a verificação de próximo elemento e a inserção é uma função vailadPos. Esta recebe a lista e a posição atual e verifica se a posição é maior ou igual a zero e se a posição é maior que a quantidade de elementos -1.

```
int validaPos(Lista l, corrente pos){
        if(pos >=0&&pos< ((l-> quant_Elem) -1)){
                return (1);
        }else{
                return (0);
        }
}
```

Adicionalmente, o percurso ao longo dos elementos de uma lista requer de uma operação que possibilite a movimentação ao longo da estrutura, elemento a elemento. A função *proximoElemento* retorna o índice no vetor correspondente à posição do próximo elemento, se for o último e não tiver próximo, retorna -1.

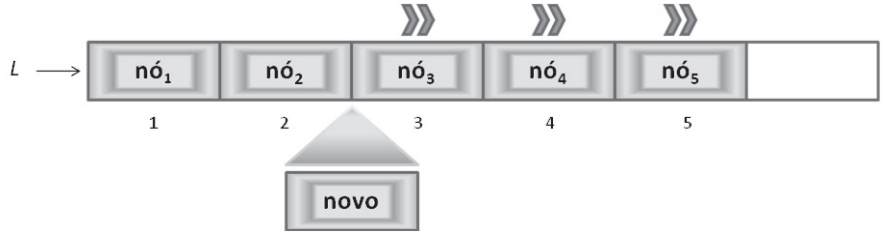
```
corrente proximoElemento (Lista l, corrente pos) {
        pos = pos++;
                if (validaPos(l, pos)
                        return (pos);
                return (-1);
}
```

A operação de inserção é possivelmente uma das mais importantes, uma vez que é através dela que a lista será construída. Em particular, o tipo lista não possui nenhuma restrição em relação à inserção, podendo acontecer em qualquer posição. Desta forma, na hora de inserir um elemento na lista, é necessário informar qual a posição correspondente ao novo elemento, seja no inicio, meio ou fim da lista. Considerando que a inserção nem sempre é possível por conta da limitação de espaço da estrutura de dados utilizada, a função retorna 1 se a inserção foi realizada com sucesso e 0 em caso contrário

Algoritmo 0

```
int Inserir (Lista l, tipo_base  dado, corrente pos) {
        int i;
        if (!validaPos(l pos) || (cheia (l))) return (0);

        for (i = l->) quant_elem ; i > = pos; i--){
        l -> v[i] = l -> v[i-1];
        }
        l -> v[pos] = dado;
        l -> quant_elem = (l -> quant_elem)+1;

        return (1);
}
```

Dependendo da posição onde o elemento será inserido, o trabalho requerido pode ser estimado de forma a estabelecer o custo da função. Pelo fato de se utilizar alocação estática e contígua de memória para o armazenamento dos elementos da lista, a inserção de um elemento em uma execução requer que o espaço físico na memória seja providenciado em tempo de compilação. Para isso é necessário deslocar (*shift*) todos os elementos necessários, desde a posição requerida até o final da lista. Por exemplo, se a lista possui 5 elementos e o novo elemento precisa ser inserido na posição 3, os últimos 3 elementos precisarão ser deslocados à direita, para que a posição de inserção requerida fique disponível para o novo elemento a ser inserido. A situação é ilustrada na figura a seguir.

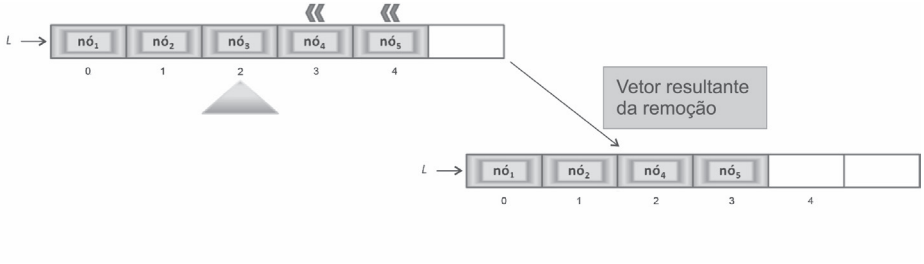


O pior cenário neste caso acontece quando é requerida a inserção na primeira posição da lista, obrigando o deslocamento de todos os elementos da lista em uma posição à direita. Nete caso, o custo requerido é *O(n)*.

Analogamente à operação de inserção, a operação de remoção exclui um elemento em qualquer posição, portanto esta posição precisa ser informada explicitamente. O algoritmo é responsável por checar se a posição informada representa uma posição válida dentro da estrutura.

Algoritmo 0.1

```
int Remover (Lista l, corrente pos) {
        int i;
        if (vazia(l) || (!validaPos(l)) return (0);

        int dado = l -> v[pos];
        for (i = pos + 1 ; i < l -> quant_elem; i++)
                l -> v[i-1] = l -> v[i];
          l -> quant_elem = (l -> quant_elem)-1;
        return (1);
}
```



Realizando uma análise análoga ao caso da inserção, o custo para a remoção é *O(n)*.

No caso em que seja necessário remover um elemento de acordo com algum conteúdo específico, o elemento em questão precisa ser previamente localizado através da operação de *Busca*. A partir da posição retornada pela busca, caso o elemento seja efetivamente encontrado na estrutura, a remoção poderá ser efetivamente realizada.

A busca por um determinado elemento pode ser originada de duas formas. Na primeira variante, a busca pode ser orientada por um determinado conteúdo, retornando como resultado a sua posição na lista, no caso em que o elemento for efetivamente encontrado, caso contrário retorna -1.

```
corrente Busca (Lista l, tipo_base  dado) {
        int i;
        for (i = 0; i <= tamanho(l); i++)
        if (l -> v[i] == dado) return (i);
        return (-1);
}
```

O pior caso possível para esta busca, consiste na situação onde o elemento procurado não se encontra na lista. Nesta situação a lista será percorrida na totalidade, sem sucesso, passando pelos *n* elementos que determinam seu tamanho. Consequentemente, o custo desta operação no seu pior caso é *O(n),* ou seja linear.

Na segunda variante, a busca pode acontecer a partir de uma determinada posição na lista, retornando o elemento contido nessa posição.

```
tipo_base  Consulta (Lista l, corrente pos) {
return (l -> v[pos-1]);
}
```

A complexidade da busca neste caso é *O(1)* uma vez que consiste no acesso direto à posição correspondente, demandando para isso tempo constante.

1. Considerando a implementação do TAD utilizando alocação estática de memória resolva as questões a seguir:

a) Explique por que o custo da remoção em uma lista implementada utilizando alocação estática e contígua de memória é *O(n)*.

b) Implemente a operação que retorna a quantidade de elementos na lista, cujo cabeçalho é: int tamanho (lista l). Determine a complexidade da operação implementada.

c) Implemente o método auxiliar que verifique se uma determinada posição é válida, isto é, se encontra dentro dos limites do vetor. O cabeçalho da operação é int validaPos (corrente pos).

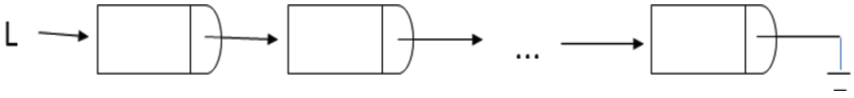
1.2. Implementação do TAD Lista usando alocação dinâmica

Na implementação do Tipo Abstrato lista adotando alocação de memória dinâmica, a alocação de memória é gerenciada sob demanda em tempo de execução. Esta característica determina que os elementos componentes são organizados em posições não-contíguas, ficando espalhados ao longo da memória. Consequentemente, não é preciso estimar *a priori* o tamanho da estrutura uma vez que o espaço é alocado na medida da necessidade, dependendo das operações de inserção e remoção realizadas.

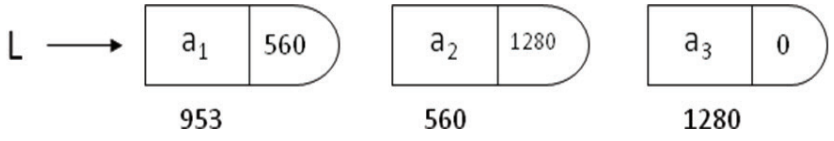
Vantagens e desvantagens desta estrutura foram discutidas na Parte

3. Em particular, esta estrutura se torna adequada quando o tamanho da estrutura é desconhecido e pode variar de forma imprevisível. No entanto, a gerência da memória torna a implementação mais trabalhosa e propensa a erros, podendo acarretar em perda de informação. Adicionalmente, o acesso aos dados é seqüencial, no sentido que para acessar o elemento na posição *m*, se torna necessário percorrer os *m - 1* elementos anteriores. Com isso, no pior caso, a busca por um elemento na lista demanda custo *O(n).*

A seguir é apresentada a definição da estrutura de dados e implementação das operações definidas na interface para o TAD Lista utilizando alocação dinâmica de memória através de ponteiros. A notação utilizada na implementação é próxima à linguagem C. Esta estrutura representa uma seqüência de elementos encadeados por ponteiros, ou seja, cada elemento deve conter, além do dado propriamente dito, uma referência para o próximo elemento da lista. Graficamente, a estrutura de dados para a implementação de listas utilizando ponteiros é a seguinte:



Por exemplo, uma lista com valores de ponteiros a endereços reais tem a seguinte forma:



O espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenados: para cada novo elemento inserido na estrutura é alocado um espaço de memória para armazená-lo. Consequentemente, não é possível garantir que os elementos armazenados na lista ocuparão um espaço de memória contíguo, e, portanto não é possível ter acesso direto aos elementos da lista. Isto implica que, para percorrer todos os elementos da lista, devemos explicitamente seguir o encadeamento dos elementos. Para isto, é preciso definir a estrutura do nó, como uma estrutura auto-referenciada contendo, além do conteúdo de informação, um ponteiro ao próximo elemento na sequência. As definições a seguir implementam a estrutura correspondente.

```
typedef struct node *no_ptr;
```

```
struct node {
        tipo_base  elemento;
        no_ptr prox;
};
```

Finalmente a lista é definida como um ponteiro ao primeiro nó da lista, a partir do qual a sequência de nós é encadeada.

```
typedef no_PTR Lista;
```

Uma lista é um tipo de dado que estrutura elementos cujo tipo pode ser arbitrariamente complexo, envolvendo inclusive, a utilização de outros TADs. A definição a seguir especifica o tipo base dos elementos da lista como inteiros.

```
typedef int tipo_base;
```

A implementação de listas com ponteiros requer um cuidado especial uma vez que qualquer erro na manipulação dos ponteiros pode acarretar em perda parcial ou total da lista de elementos. Assim sendo, a utilização de um ponteiro auxiliar para o percurso ao longo da lista pode ser de grande utilidade. Com esse objetivo definimos um tipo *Corrente*, a ser utilizado como cópia da lista de forma a possibilitar a sua manipulação com segurança.

```
typedef no_ptr Corrente;
```

A função que *cria* uma lista vazia utilizando alocação dinâmica de memória é apresentada a seguir. A função tem como valor de retorno a lista vazia inicializada, isto é, o valor de retorno é *NULL,* pois não existem elementos na lista.

```
Lista Criar (){
  return (NULL);
}
```

O método Inicializar posiciona o índice da posição corrente no inicio da lista. Desta forma o ponteiro *pos* aponta para o mesmo local onde se encontra o primeiro elemento da lista. Este método é útil quando a lista precisa ser percorrida desde o inicio.

```
corrente Inicializar (Lista L){
        corrente pos = L;
        return (pos);
}
```

O deslocamento de um elemento para o seguinte na lista é dado pelo percurso ao longo dos ponteiros, onde, a partir do nó atual, a função a seguir retorna o ponteiro onde se localiza o próximo elemento na lista.

```
corrente proximoElemento (corrente p) {
        return (p -> prox);
}
```

A procura por um conteúdo na lista é realizada através da função *Busca*. Esta função percorre a lista desde o inicio, enquanto o elemento não for encontrado. A função retorna a posição do elemento na lista em caso de sucesso na procura, ou *NULL* se o elemento não for encontrado.

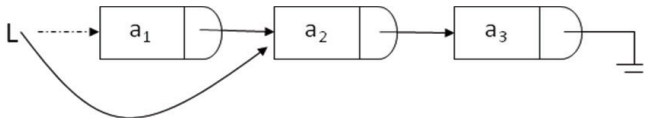
```
corrente Busca (Lista L, tipo_base  x) {
        corrente p = L;
        while ((p != NULL) && (p -> elemento != x))
        p = p -> prox;
  return p;
}
```

O acesso às informações contidas nos nós da lista é realizado através da função *Consulta*, que retorna o conteúdo de informação armazenado no nó referenciado pelo ponteiro corrente.

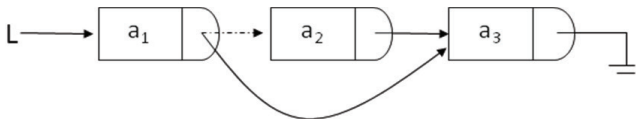
```
tipo_base Consulta (Lista L, corrente p){
        if (p != NULL) return (p -> elemento);
}
```

A remoção de um elemento da lista envolve a análise de duas situações: a remoção do primeiro nó da lista ou de um nó em outra posição qualquer, sem ser no inicio da lista. A seguir, o processo de remoção em cada caso é ilustrado.

No caso da remoção do primeiro elemento da lista é necessário que o ponteiro à lista seja atualizado, indicando o novo primeiro elemento.



No caso de remoção de um elemento, sem ser o primeiro, o processo consiste em fazer um *by pass* sobre esse elemento através do ajuste correto dos ponteiros, para posteriormente liberar a memória correspondente. Para efetivar a remoção é preciso o nó anterior ao nó a ser removido, que pode ser obtido a partir da função auxiliar *Anterior*.



A função *Remover* apresentada a seguir, remove o elemento correspondente a uma determinada posição *pos*, passada por parâmetro. Esta posição pode ser resultado de um processo de busca, a partir de um determinado conteúdo. Em ambos os casos é preciso liberar a memória correspondente ao nó removido.

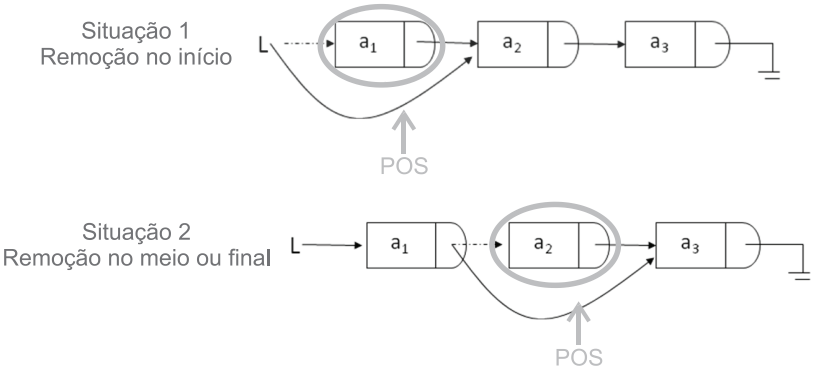
A seguir é apresentado o algoritmo que implementa o processo de remoção.

Algoritmo 0.2

A função inerior retorna o nó anterior ao nó passado (pos). Ela percorre a toda a lista verificando se o próximo elemento é o elemento pos.

```
corrente Anterior(Lista l, corrente pos){
        corrente ant = null;

  if(pos!= l){
        corrente atual = l;
        while(atual != null & atual -> prox != pos){
              atual = atual -> prox;
        }
        ant = atual
}

return (1);
}
```

Na remoção encontramos duas situações: 1) quando o elemento a ser removido é a primeira posição (Nó anterior é null); e 2) quando o elemento a ser removido não é o elemento da primeira posição (Nó anterior não é null).



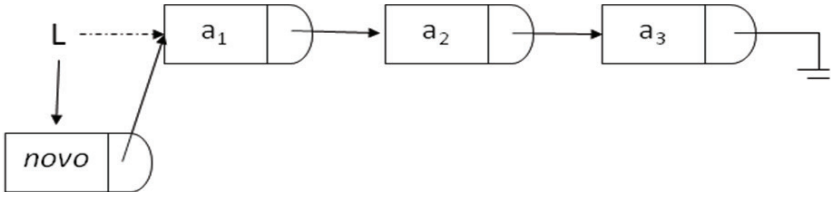

```
int Remover(Lista l, corrente pos){
        corrente noAnterior = Anterior (L,pos);

        corrente tmp_cell = pos;

        if(noAnterior == NULL){
                l = l -> prox;
        }else{
                noAnterior -> prox = pos -> prox;
        }

        free (tmp_cell);

        return (1);
}
```

O algoritmo apresentado para a função remover utiliza uma função anterior. Esta função anterior tem o papel de retornar o elemento que antecede a posição atual. Tendo em vista que para o elemento atual ser removido, basta ligar o elemento anterior ao próximo. A seguir a função anterior é apresentada:

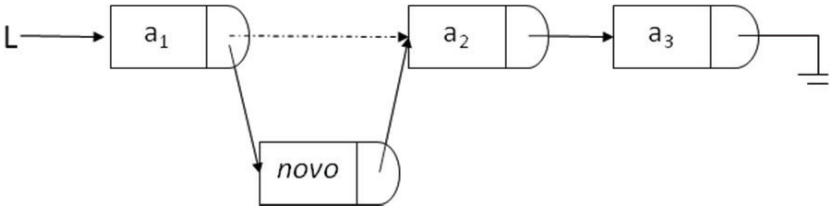
```
corrente Anterior(Lista l, corrente pos){;
        corrente ant = null;
```

```
        if(pos != NULL){
         corrente atual = l;
        while(atual != null & atual ->prox !=pos){
                atual = atual -> prox;
        }
                ant = atual;

        return (ant);
}
```

A inserção em uma lista pode acontecer em qualquer posição, que pode ser no inicio, no final ou qualquer outra posição no meio da lista. O conteúdo do parâmetro *pos* representa a posição de inserção requerida para o elemento *novo* a ser inserido, no caso de inserção na cabeça da lista *pos é null*. Neste caso, o ponteiro L que apontava ao primeiro elemento da lista aponta agora para o novo elemento inserido.



Para qualquer outro valor de *pos*, o processo de inserção acontece como ilustrado a seguir. A lista precisa ser percorrida até a posição de inserção requerida. Nesse ponto, o novo elemento será inserido atualizando os ponteiros correspondentes.




```
int Inserir (Lista l, tipo_base  dado, corrente pos){
  int i;
  Corrente atual = Inicializar (l);
```

```
    Lista novo = (Lista) malloc(sizeof(struct node));
    novo -> elemento = dado;
    if (pos == NULL){
            novo -> prox = l;
            l = novo;
    }

    else {
      while (atual -> next != NULL) and (atual ->
                    next != pos)
                    atual = atual -> prox;
                    novo -> prox = atual -> prox;
                    atual -> prox = novo;
    }
    else return (1);
}
```

A função *Vazia* é utilizada para verificar se a lista possui ou não elementos armazenados. A verificação consiste em checar se o ponteiro ao primeiro elemento é *null*.

```
int Vazia (Lista L) {
        if (L == NULL) return (1)
        else return (0);
}
```

A função *ultimoElemento* é utilizada para verificar o final da lista. Esta checagem consiste em determinar se o próximo elemento que segue ao atual é *NULL*.

```
int ultimoElemento (corrente p) {
        if (p -> prox == NULL) return (1)
        return (0);
}
```

Com exceção de *Busca* e *Anterior* todas as operações consomem tempo *O(1)*. Isto por que somente um número fixo de instruções é executado sem

levar em conta o tamanho da lista. Para *Busca* e *Anterior* o custo é *O*(*n*), pois a lista inteira pode precisar ser percorrida se o elemento não se encontra ou for o último da lista.

Atividades de avaliação

Utilizando o TAD Lista definido nesta parte, implemente como aplicação um programa que, dadas duas listas A e B, crie uma terceira lista L intercalando os elementos das duas listas A e B.

```
Lista Intercala (Lista A , Lista B) {
  Lista L = cria ();
  corrente pos_L = Inicializar (L);
  corrente pos_A = Inicializar (A);
  corrente pos_B = Inicializar (B);

  /*Assumimos que A e A tem o mesmo tamanho
  while (not ultimoElemento(pos_A)) &&
      (not ultimoElemento (pos_B)){
    Inserir (L, Consulta (pos_A), null);
    pos_A = proximoElemento (pos_A);
    Inserir (L, Consulta (pos_B), null);
    pos_B = proximoElemento (pos_B);
  }
  return(L);
}
```

Atividades de avaliação

Considerando a implementação do TAD utilizando alocação dinâmica de memória resolva as questões a seguir:

1. Implemente a operação que retorna a quantidade de elementos na lista, cujo cabeçalho é: int Tamanho (Lista l). Determine a complexidade da operação implementada.

2. Implemente uma rotina para a remoção de uma lista desalocando a memória utilizada. O cabeçalho da rotina é void Remove_list (Lista L).
3. Implemente a rotina auxiliar chamada *anterior* usada na remoção, de acordo com o seguinte cabeçalho corrente anterior (Lista L, corrente pos). Esta rotina retorna a posição do elemento anterior a uma outra posição pos. Se o elemento não for encontrado retorna NULL.
4. Utilizando as operações definidas na interface do TAD Lista implemente um método que dadas duas listas L1 e L2, calcule L1 ∪ L2 (união) e L1 ∩ L2 (interseção). O resultado das operações deve ser retornado em uma terceira lista L3.
5. Utilizando as operações definidas na interface do TAD Lista implemente um método que dada uma lista retorne uma segunda lista onde os elementos pertencentes à primeira estejam ordenados em forma crescente. Determine a complexidade do seu algoritmo.
6. Escreva um programa que, utilizando o TAD Lista, faça o seguinte:

a) Crie quatro listas (L1, L2, L3 e L4);

b) Insira sequencialmente, na lista L1, 10 números inteiros obtidos de forma randômica (entre 0 e 99);

c) Idem para a lista L2;

d) Concatene as listas L1 e L2, armazenando o resultado na lista L3;

e) Armazene na lista L4 os elementos da lista L3 (na ordem inversa);

f) Exiba o conteúdo das listas L1, L2, L3 e L4.

Texto Complementar

Variações sobre o TAD Lista

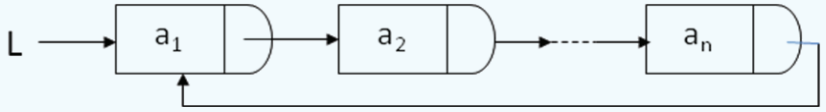
Lista ordenada

Uma lista ordenada é uma lista onde seus elementos componentes são organizados de acordo a um critério de ordenação com base em um campo chave. A ordem estabelecida determina que a inserção de um determinado elemento na lista irá a acontecer no lugar correto. A lista pode ser ordenada de forma crescente ou decrescente.
A partir da existência de um critério de ordenação na lista, a função responsável pela busca por um determinado conteúdo na lista pode ser adaptado de forma a tornar a busca mais eficiente.

Lista circular

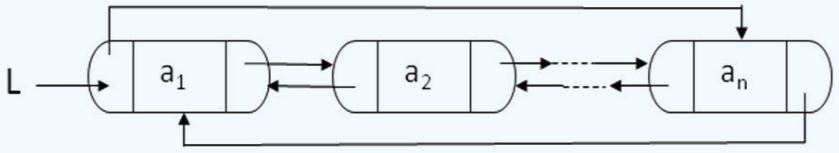
A convenção consiste em manter a última célula apontando para a primeira. Desta forma, o teste por fim de lista nunca é satisfeito. Com isso, precisa ser estabelecido

um critério de parada de forma a evitar que o percurso na lista não encontre nunca o fim. Uma forma padrão é estabelecido com base no número de elementos existentes na lista.



Lista duplamente encadeada

Em alguns casos pode ser conveniente o percurso da lista de trás para frente através da adição de um atributo extra na estrutura de dados, contendo um ponteiro para a célula anterior. Esta mudança na estrutura física acarreta um custo extra no espaço requerido e também aumenta o trabalho requerido nas inserções e remoções, uma vez que existem mais ponteiros a serem ajustados. Por outro lado simplifica a remoção, pois não mais precisamos procurar a célula anterior *(O(n))*, uma vez que esta pode ser acessada diretamente através do ponteiro correspondente.



Capítulo 4

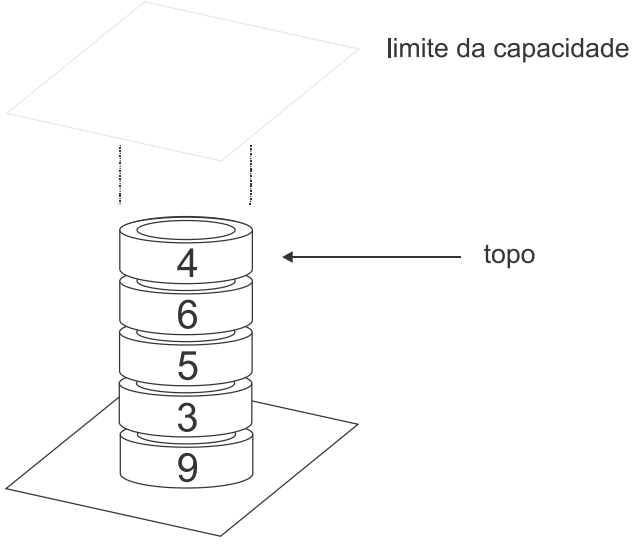
Pilhas

Introdução

Em geral, as operações de inserção e remoção realizadas sobre listas são custosas. No caso da implementação utilizando alocação de memória estática, estas operações acarretam a movimentação dos elementos. No caso da alocação dinâmica o deslocamento até a posição correta de inserção ou remoção envolve o percurso ao longo do encadeamento pelos elementos. Em ambos os casos, o custo destas operações é *O(n)*. Estas situações desfavoráveis podem ser contornadas se os elementos a serem inseridos e removidos se encontram em posições determinadas especialmente, como a primeira ou última posição.

Uma *Pilha* é uma lista com a restrição de que inserções e remoções são executadas exclusivamente em uma posição, referenciada como fim ou *topo*.

Pilhas são conhecidas como estruturas LIFO do inglês *Last In First Out* ou último que entra primeiro que sai. Em uma Pilha o único elemento acessível é o elemento que se encontra no topo. Consequentemente, a operação de busca ao longo da estrutura, por exemplo, não é uma operação aplicável para esta estrutura de dados. Graficamente, uma pilha pode ser representada da seguinte forma:



O funcionamento de uma pilha pode ser facilmente interpretado a partir de uma analogia simples com uma pilha de livros pesados. Livros são empilhados, um encima de outro, sendo que o último livro empilhado é o que fica no topo da pilha, e, portanto é o único visível e que pode ser consultado sem precisar movimentar outros exemplares. Por se tratar de livros pesados, o acesso a um livro determinado na pilha, requer que os livros encima deste sejam retirados, um a um, a partir do topo. Desta forma, o último livro empilhado será o primeiro a ser retirado da pilha. A partir desta descrição, o funcionamento da pilha pode ser modelado de acordo com a seguinte interface.

```
Interface do TAD Pilha

// Cria uma pilha vazia
Pilha Criar ();

//insere um novo elemento no topo da pilha.
int Push (Pilha p, tipo_base  dado);

// consulta pelo elemento que se encontra no topo
tipo_base  Top (Pilha p);

//Remove e retorna o elemento do topo
tipo_base Pop (Pilha p);

/* Retorna 1 se não tem mais elementos na pilha, ou 0 em
   caso contrario.*/
int Vazia (Pilha p)

// Retorna 1 se a pilha estiver cheia, e 0 em caso contrario.
int Cheia (Pilha p);

// Retorna a quantidade de elementos na pilha.
int Tamanho (Pilha p);
```

O fato de máquinas modernas possuírem operações sobre pilhas como parte do conjunto de instruções, faz desta estrutura uma abstração fundamental na Ciência da Computação, depois do vetor.

Pilhas são *listas*, portanto as abordagens de implementação utilizadas para listas são válidas para o caso da implementação de pilhas. Considerando que a implementação de pilhas representa uma variação de listas, boa parte da especificação e implementação de listas pode ser aproveitada.

1. Implementações do TAD Pilha usando vetores

Levando em conta as restrições inerentes à própria estrutura de dados e as restrições na manipulação da pilha, a estrutura projetada para a implementação de listas é modificada, adicionando mais um campo de informação referente à localização do elemento que se encontra no topo da pilha. Esta informação é indispensável na implementação das operações de inserção e remoção, de forma a possibilitar o acesso direto ao local. Com essa pequena alteração na estrutura de dados as operações passam a demandar tempo constante.

```
typedef int tipo_base
#define tamanho;

struct estrutura_Pilha {
        int topo;
        tipo_base elementos [tamanho];
};

typedef struct estrutura_Pilha *Pilha;
```

Em C uma pilha é definida como um ponteiro à estrutura, que é passado por valor para as funções que irão modificar o conteúdo da pilha.

Dada a semelhança com a estrutura de dados utilizada para o caso de listas, o método *Criar* se mantem essencialmente o mesmo, adicionando somente a sentença de inicialização do campo *topo* para a primeira posição.

Algoritmo 1

```
Pilha Criar(){
 Pilha p = (Pilha) Malloc (sizeof(estrutura_Pilha));
 if(p != NULL){
    p -> topo = -1;
    return (p);
  }
  else printf ("Não existe memória suficiente");
  return;
 }
```

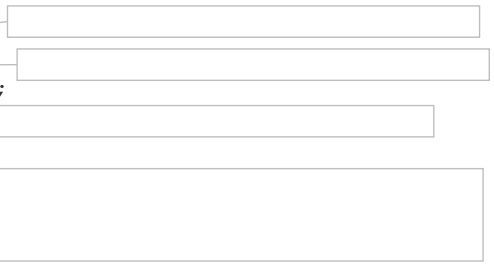
Na escolha pela extremidade do vetor a ser definida como o *topo* onde a inserção e remoção de elementos estará acontecendo, é importante analisar os custos decorrentes desta escolha. Como discutido anteriormente, a propriedade de alocação contígua de memória traz como desvantagem a necessidade de movimentações dos elementos ao longo da estrutura de dados. Nesse caso, se o topo for escolhido como a primeira posição do vetor, o custo da inserção e remoção seria de *O(n)*. Em contrapartida, a definição de topo como sendo o último elemento inserido é mais conveniente uma vez que este pode ser acessado em tempo constante a partir do tamanho da estrutura.

Algoritmo 2

A operação de inserção de um elemento é feita no tipo da pilha caso a pilha não esteja cheia. A seguir seguem os algoritmos de verificação de pilha cheia e de inserção.

```
int Cheia(Pilha p){
        if(p -> topo == tamanho -1 return (1);
        else return (0);
        }
```

```
int Cheia(Pilha p) {
        if(Cheia(p) ! =1){;
        p-> topo++;
        p-> elementos [p-> topo] = dado;
        return (1);
    }else{
        return (0);
    }
}
```



A operação de remoção do elemento que se encontra no topo da pilha é descrita no algoritmo a seguir.

```
tipo_base Pop (Pilha p) {
        tipo_base v;
        v = p -> elementos [p -> topo];
        (p -> topo)--;
        return v;
}
```

A remoção de um elemento somente é possível sempre que a pilha contiver pelo menos um elemento. Nesse caso o elemento é copiado em uma variável auxiliar para seu retorno posterior, decrementando em 1, consequentemente, a posição do último elemento. A lógica seguida para a implementação do algoritmo de consulta do topo, *Top*, é a mesma, com a diferença de que não é preciso alterar a posição do *topo* uma vez que nenhum elemento é removido da pilha.

O algoritmo que verifica se a pilha se encontra vazia ou não é baseado na informação contida no campo *topo*. Considerando que o *topo* indica indiretamente a quantidade de elementos efetivamente contidos na pilha, a checagem pela posição onde este elemento se encontra é utilizado para estabelecer se a pilha está vazia. Nesse caso a função retorna 1.

```
int Vazia (Pilha p) {
        if (p -> topo == 0) return (1)
        else return (0);
}
```

Uma estratégia similar pode ser utilizada para estabelecer se a pilha se encontra cheia, só que neste caso a posição do elemento no topo precisa ser confrontada contra o tamanho total reservado para a estrutura de dados.

Atividades de avaliação

Considerando a implementação do TAD utilizando alocação estática de memoria resolva as questões a seguir:

1. Implemente o algoritmo que consulta e retorna o conteúdo correspondente ao elemento que se encontra no topo, atendendo ao seguinte cabeçalho: tipo_base Top (Pilha p).
2. Explique por que é mais conveniente a definição do topo da pilha no final e não no inicio da estrutura.

2. Implementação do TAD Pilha usando ponteiros

No caso da implementação de pilhas utilizando alocação dinâmica de memória, a desvantagem do acesso sequencial necessário para alcançar qualquer elemento da pilha pode ser contornado eficientemente, uma vez que no caso da pilha, as operações acontecem necessariamente a partir de um extremo. A escolha pelo extremo da estrutura a ser considerado como *topo* irá a determinar o custo envolvido na execução das operações: enquanto que o acesso ao último elemento da pilha envolve custo *O(n),* o acesso ao primeiro elemento é constante. Consequentemente, no caso da implementação da pilha utilizando ponteiros, a definição do *topo* como sendo o inicio (cabeça da lista) da pilha é mais vantajoso em termos de desempenho e complexidade.

A estrutura de dados utilizada na implementação da pilha é idêntica aquela definida no caso de listas, consequentemente, a maior parte das operações são coincidentes, tais como *Criar* e *Vazia*.

O nó pode ser visualizado de acordo com a ilustração a seguir:


```
typedef struct node *no_ptr;

struct no dl{
        tipo_base elemento;
        no_ptr prox;
};

typedef no_ptr Pilha;
```

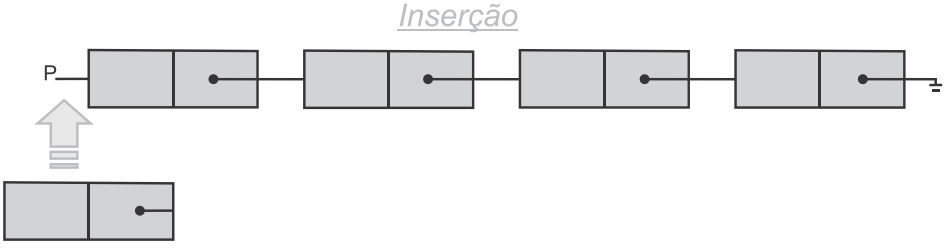
A partir da decisão de projeto para o TAD Pilha de considerar o topo para as operações de inserção (*push*) e remoção (*pop*) na cabeça da lista de elementos, cuidados especiais na implementação de tais operações são requeridos de forma a evitar perda de informação.

A seguir, o algoritmo de criação da pilha. Este é bom simples, apenas devolve nulo como valor inicial da pilha.

```
Pilha Criar(){
    return (NULL);
}
```

A operação *Push*, responsável pela inserção de um novo elemento no topo da pilha, consiste na alocação de memória para o novo elemento. Se a alocação de memória é realizada com sucesso, o novo componente é instanciado com a informação correspondente e inserido como primeiro elemento na lista, atualizando consequentemente a cabeça da lista com o endereço do novo elemento. No caso em que a inserção aconteça com sucesso o algoritmo retorna 1, e 0 em caso contrario.

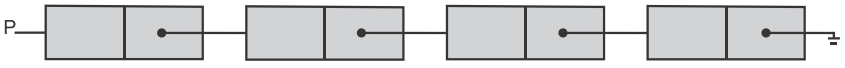
```
int Push (Pilha p, tipo_base x) {
        no_ptr novo = (no_ptr) malloc (sizeof (struct no));
        if (no_tmp == NULL){
          printf ("Memoria insuficiente!!");
         return (0);
        }
        else {
         novo -> elemento = x;
         novo -> prox = p;
         p = novo;
         return (1);
          }
}
```



A operação de desempilhar, pop, realiza a remoção de m elemento no inicio da pilha.

```
tipo_base Pop (Pilha p) {
    no_ptr temp;
    if (p == NULL){
     printf ("Pilha vazia.");
            return (0);
     }else{
            temp = p;
            int valor = temp -> elemento;
            p = p -> prox;
            free(temp);
      return (valor);
   }
}
```

Remoção



Antes da remoção

Após a remoção

Atividades de avaliação

Considerando a implementação do TAD utilizando alocação dinâmica de memoria resolva as questões a seguir:

1. Implemente o algoritmo que consulta e retorna o conteúdo correspondente ao elemento que se encontra no topo, atendendo ao seguinte cabeçalho: tipo_base Top (Pilha p).
2. Implemente uma operação que libere a memória alocada pela pilha. Determine a complexidade do seu algoritmo.
3. Explique por que é mais conveniente a definição do topo da pilha no inicio e não no fim da estrutura, no caso de utilização de ponteiros.

4. Utilizando as operações definidas na interface do TAD Lista e do TAD Pilha, elabore um algoritmo que dada uma lista ordenada, inverta a ordem dos elementos na lista, utilizando para isso uma pilha. Determine a complexidade do seu algoritmo.
5. Utilizando as operações definidas na interface do TAD Pilha escreva um algoritmo para ordenar pilhas, sendo que no final do processamento os elementos da pilha devem estar dispostos em ordem crescente de seus valores. Determine qual a estrutura auxiliar mais adequada para suportar o processo. Determine a complexidade do seu algoritmo.
6. Utilizando as operações definidas na interface do TAD Pilha escreva um algoritmo que forneça o maior, o menor e a média aritmética dos elementos de uma pilha dada como entrada.

Capítulo 5

Filas

Introdução

Assim como pilhas, filas são listas que possuem algumas restrições específicas para a execução das operações de inserção e remoção. Na fila, as inserções são realizadas em um extremo, enquanto a remoção ocorre no outro. A fila segue o modelo FIFO, do inglês *First In First Out*, ou seja, que o primeiro que entra na fila é o primeiro em sair.

A estrutura de dados fila, como seu próprio nome já sugere, é semelhante ao funcionamento de uma fila de banco. Onde: 1) Se não há ninguém na fila e chega uma pessoa, está será o inicio e o fim da fila; 2) A partir de então, quaçquer pessoa que chegar irá para o final da fila (após a pessoa que está no fim); 3) Cada elemento a ser removido será do inicio da fila (O primeiro que chega na fila é o primeiro que sai).

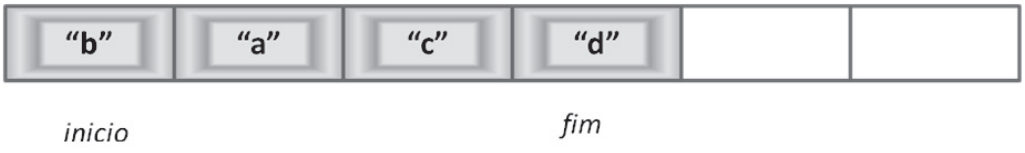
Interface do TAD Fila

```
// Cria uma fila vazia
Fila Criar ();

//insere um novo elemento no topo da fila.
void Inserir (Fila p, tipo_base  dado);

// Consulta pelo elemento que se encontra no topo
tipo_base  Top (Fila f);
```

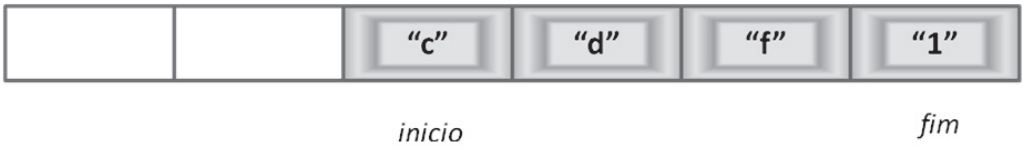
```
//Remove e retorna o elemento do topo
int Remover (Fila f);

// Retorna 1 se não tem mais elementos na fila, ou 0 em caso contrario.*/
int Vazia (Fila f);

// Retorna 1 se a fila estiver cheia, e 0 em caso contrario.
int Cheia (Fila f);

// Retorna a quantidade de elementos na fila.
int Tamanho (Fila f);
```

1. Implementação do TAD Fila usando vetores

Considerando que a operação de remoção acontece em uma extremidade e a inserção na outra, é interessante manter apontadores indicando ambos extremos da estrutura, de forma que o acesso seja direto (*O(n)*). Consequentemente, a estrutura de dados fila inclui, além do vetor que irá a comportar, as informações correspondentes, os índices correspondentes ao inicio e fim da fila, e o tamanho da fila relativo ao número de elementos contidos. A seguinte figura mostra uma fila em algum estado intermediário.



De acordo com esta estratégia e depois da remoção de *"b"* e *"a"*, e da inserção de *"f"* e *"1"*, a situação da fila seria a seguinte:



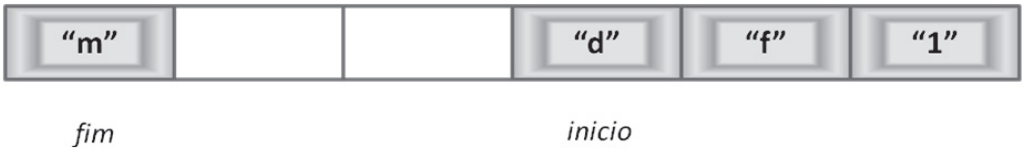
A inserção de um elemento na fila incrementa o tamanho e o índice que indica o fim da fila, enquanto que na remoção de um elemento o tamanho da fila é decrementado e o índice que indica o inicio é incrementado. Esta estratégia evita a movimentação de elementos sempre que uma inserção ou remoção for realizada. Com isso o custo de ambas operações fica constante.

Levando em conta as restrições inerentes à manipulação da fila, a estrutura projetada para a sua implementação utilizando alocação de memória estática ou vetor precisa ser modificada em relação à pilha. Neste caso é preciso indicar mais um campo de informação referente à localização do elemento que se encontra no inicio da fila. Esta informação é indispensável na implementação das operações de inserção e remoção, de forma a possibilitar o acesso direto ao local. Com essa pequena alteração na estrutura de dados as operações passam a ser executadas em tempo constante.

```
typedef int tipo_base; //definindo o tipo base da fiça
#define tamanho;

struct estrutura_Fila {
        int ini;
        int fim;
        int quant_elementos;
        tipo_base elementos [tamanho];
};

typedef struct estrutura_Fila *Fila;
```

No entanto existe um problema potencial uma vez que a fila pode ficar aparentemente cheia, no entanto vários elementos podem ter sido removidos, podendo existir na verdade poucos elementos na fila (observe a figura anterior na qual o fim está na ultima posição do vetor, no entando há duas posições vazias no inicio do vetor).

A solução para contornar este problema é implementar o chamado *incremento circular* no vetor, onde **sempre que os índices de inicio ou de fim chegam no final do vetor, estes são redefinidos na primeira posição do vetor**. Esta estratégia requer um cuidado especial na hora do percurso sobre a estrutura, uma vez que o fim do vetor precisa ser determinado logicamente (p.e., pela **quantidade de elementos** na fila), e não mais fisicamente pelo fim da estrutura. A figura a seguir exemplifica o funcionamento da fila utilização o vetor circular, onde o conteúdo *"c"* foi removido e adicionado o conteúdo *"m"*.



A criação de uma fila vazia, similarmente à criação de uma lista, envolve a alocação de memória para a estrutura de dados respectiva, e a posterior inicialização dos campos envolvidos, no caso os índices de inicio e fim, e a quantidade de elementos[14] que precisam ser inicializados em 0.

[14] Quando a fila está cheia, sua quantidade de elementos é igual ao tamanho do vetor.

Algoritmo 3

```
Fila Criar(){
        Fila
        if (f = (Fila) malloc (sizeof (estrutura_Fila));
                f -> ini = -1;
                f -> fim = -1;
                f -> quant_elementos = 0;
                return(f);
        }else{
                printf("Não existe memória suficiente.");
                return;
        }
}
```

Considerando a inserção de elementos acontecendo no fim e a remoção no inicio, o código das respectivas operações é apresentado a seguir. No caso da inserção, e levando em conta que o tamanho da estrutura de dados estática foi estabelecida em tempo de compilação, é importante verificar que exista espaço disponível para armazenar um novo elemento. Após esta verificação, o novo elemento é inserido na primeira posição livre indicada por *fim*. Após isso, tanto *fim* quando a quantidade de elementos precisam ser atualizados.

```
void inserir (Fila f, tipo_base v) {
        if (f -> quant_elementos == tamanho) {
                printf ("Fila cheia!");
                return;
        }
        f -> fim = incrementar (f -> fim);
                if(f-> quant_elementos == 0){
        }

        f -> elementos [f-> fim] = v;
        f -> quant_elementos++;
}
```

A implementação do vetor circular requer que seja realizado um incremento especial onde, a partir de uma posição corrente, o método retorna a posição seguinte ou, no caso de atingir o final do vetor, a posição corrente é especificada como sendo no inicio da estrutura, no caso 0.

```
int incrementar (int pos){
        if (pos == tamanho - 1) return (0);
        else return (pos++);
}
```

Os elementos da fila são removidos do seu inicio. Deste modo, as operações de remoção movimentam o elemento ini, onde o inicio passa uma posição para frente no vetor e a quantidade de elementos é reduzida em 1. Duas situações merecem o cuidado especial, quando a fila está vazia (quantidade de elementos igual é zero) e quando a fila tem somente o elemento que será removido (neste caso quando a quantidade de elementos para a ser zero inicio e fim devem receber o valor -1 para identificar que a fila está vazia).

```
int remover(Fila f){
        if (f -> quant_elementos == 0);
                printf("Fila vazia.");
                return;
        }
```

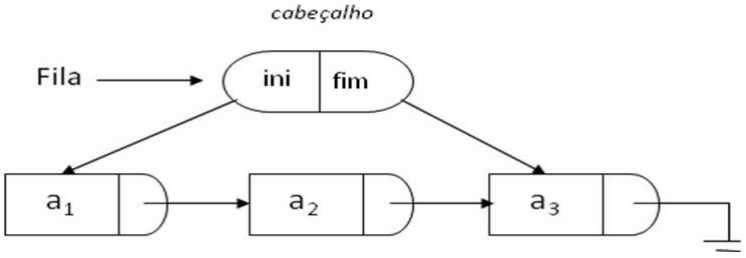
```
        int temp = f -> elementos[f -> ini];

        f -> ini = incrementar(f -> ini);

        (f -> quant_elementos)--;

        if(f -> quant_elementos == 0){;
                f -> ini = -1;
                f -> fim = -1;
        }

        return(temp);
}
```

Considerando a implementação do TAD utilizando alocação estática de memória resolva as questões a seguir:

1. Implemente os algoritmos Tamanho, Cheia, Topo e Vazia de acordo com os respectivos cabeçalhos definidos na interface do TAD.
2. Explique por que é mais conveniente a realização da operação de inserção no fim e de remoção no inicio. Como seria se fosse ao contrario?
3. Escreva um algoritmo para ordenar filas, sendo que no final do processamento os elementos da fila devem estar dispostos em ordem crescente de seus valores. Determine qual a estrutura auxiliar mais adequada para suportar o processo. Determine a complexidade do seu algoritmo.

2. Implementação do TAD Fila usando ponteiros

A implementação de filas utilizando ponteiros segue a mesma estratégia analisada na seção anterior. Como teremos que inserir e retirar elementos das extremidades opostas da lista, representando o início e o fim da fila, é preciso utilizar dois ponteiros, *ini* e *fim*, que apontam respectivamente para o primeiro e para o último nó da fila. A partir desta necessidade se faz indispensável a implementação da fila utilizando um nó diferenciado, chamado de *header* ou cabeçalho, para conter estes ponteiros. Essa situação é ilustrada na figura abaixo:



A definição da estrutura de dados que implementa a abordagem descrita é a seguinte:

```
typedef struct no *no_ptr;

typedef struct no {
        tipo_base elemento;
        no_ptr prox;
};
```

A estrutura de dados envolve a definição de uma lista de elementos, no mesmo formato em que foi definida para o caso de listas e pilhas. Adicionalmente, a estrutura correspondente ao nó cabeçalho precisa ser especificada uma vez que inclui campos diferenciados.

```
struct cabeçalho {
        no_ptr ini;
        no_ptr fim;
};
```

Finalmente, a fila é definida como sendo um ponteiro ao nó cabeçalho.

```
typedef struct cabeçalho *Fila;
```

A partir da interface especificada para o TAD fila, as operações de criação e verificação pela fila vazia seguem as mesmas estratégias anteriormente descritas.

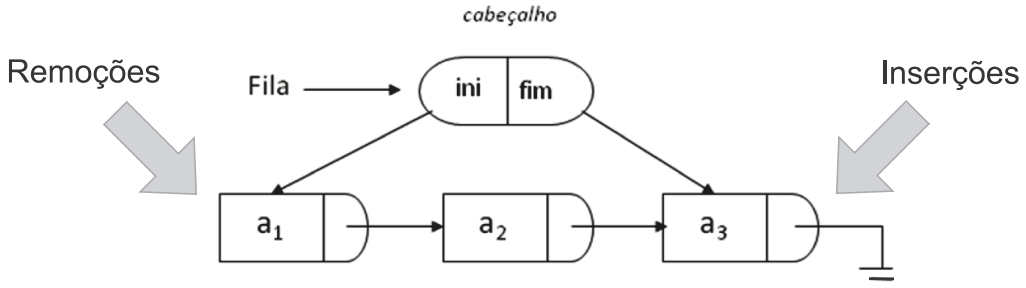
Algoritmo 5

```
Fila Criar(){
        Fila f = (Fila) malloc(siziof(cabeçálho));
        if(f != NULL){;
                f -> ini = NULL;
                f -> fim = NULL;
                return;
        }else{
        printf("Não existe memória suficiente.");
                return;
        }
}
```

Algoritmo 6

```
int Vazia(Fila f){
        if(f -> ini == NULL){
                return (1);
        }else{
                return (0);
        }
}
```

A decisão de projeto em relação às operações de inserção e remoção envolve estabelecer em qual extremidade da fila cada uma será executada. Em particular, a lógica na implementação da operação de remoção de um elemento em uma lista implementada com ponteiros, requer a procura pelo elemento anterior na lista, de forma a atualizar o ponteiro para o próximo elemento (Ver remoção em listas). O custo desta procura pelo anterior é *O(n)*. Esta situação pode ser evitada se a remoção acontece sempre no inicio da lista, uma vez que não existe anterior que precise ser atualizado. Esta constatação determina que a escolha mais conveniente em termos de complexidade é que cada novo elemento seja inserido no fim da lista enquanto que a remoção de um elemento seja realizada no início.



No caso da operação de inserção, onde o elemento é inserido no fim da lista, na situação específica de inserção do primeiro e único elemento, ambos os ponteiros *ini* e *fim* precisam ser atualizados apontando para o único nó inserido na fila. Em qualquer outro caso, somente o ponteiro que indica o final da fila será atualizado.

Algoritmo 7

```
int inserir (Fila f, tipo_base dado){
        no_ptr novo = (no_ptr) malloc (sizeof (no));
        novo -> elemento = dado;
        novo -> prox = NULL;
        if (f -> fim != NULL){
                f -> fim -> prox = novo;
}else{
        f -> ini = novo;
}
  f -> fim = novo;
return(1);
}
```

Analogamente, a função para remover um elemento da fila deve atualizar ambos os ponteiros no caso em que for removido o último e único elemento existente, tornando a fila vazia (*ini* e *fim* nulos).

Algoritmo 8

```
tipo_base remover (Fila f){
        no_ptr temp;
        if (f == NULL){
                printf("Fila Vazia.");
                return(0);
        }else{
                if(f ->ini == NULL){
                        printf("Fila Vazia.");
                }else{
                        temp = f ->ini;
                        int valor = temp -> elemento;
                        f ->ini = f ->ini ->prox;

                        if(f ->ini == NULL){
                                f ->ini == NULL;
                        }

                        free(temp);
                        return(valor);
                }
        }
}
```

Atividades de avaliação

1. Implemente os algoritmos Top e Tamanho de acordo com os respectivos cabeçalhos definidos na interface do TAD Fila utilizando ponteiros.
2. Explique por que é mais conveniente a realização da operação de inserção no fim e de remoção no inicio. Como seria se fosse ao contrário?
3. Utilizando as operações definidas na interface do TAD implemente como aplicação uma operação que libere a memória alocada pela fila. Determine a complexidade do seu algoritmo.

4. Utilizando as operações definidas na interface do TAD Fila, escreva um algoritmo que forneça o maior, o menor e a média aritmética dos elementos de uma Fila.
5. Utilizando as operações definidas na interface dos TAD Fila e Pilha escrever um algoritmo que leia um número indeterminado de valores inteiros. Considere que o valor 0 (zero) finaliza a entrada de dados. Para cada valor lido, determinar se ele é um número par ou ímpar. Se o número for par, então incluí-lo na FILA PAR; caso contrário incluí-lo na FILA ÍMPAR. Após o término da entrada de dados, retirar um elemento de cada fila alternadamente (iniciando-se pela FILA ÍMPAR) até que ambas as filas estejam vazias. Se o elemento retirado de uma das filas for um valor positivo, então incluí-lo em uma PILHA; caso contrário, remover um elemento da PILHA. Finalmente, escrever o conteúdo da pilha.

Síntese do capítulo

Nesta parte foi apresentado o tipo abstrato de dados Lista, a partir da definição da sua interface. O tipo abstrato foi implementado de acordo com duas abordagens tradicionais: vetores e ponteiros, analisando as vantagens e desvantagens de cada uma das abordagens de acordo com as características das aplicações e operações mais frequentes.

Os TADs Pilha e Fila foram descritos e implementados como variantes do TAD Lista. Estes TADs são amplamente difundidos e de comprovada utilidade na resolução e modelagem de problemas do mundo real.

Referências

CORMEN T. H., LEISERSON C. E., RIVEST R. L., STEIN C. (2001). **Introduction to Algorithms**. McGraw-Hill e The Mit Press.

KNUTH D. E. (1968). **The Art of Computer Programming**, Vol. 1: Fundamental Algorithms. Addison-Wesley.

KNUTH D. E. (1971). **Mathematical Analysis of Algorithms**. Prociedings IFIP Congress 71, vol. 1, North Holland, 135-143.

SZWARCFITER J. I., MARKENZON L. (2010). **Estruturas de Dados e Seus Algoritmos**. 3ª. Edição. LTC.

WIRTH N. (1986). **Algorithms and Data Strutures**. Prentice-Hall.

ZIVIANI N. (2005). **Projeto de Algoritmos com implementações em Pascal e C**, 2da. Edição. Thomson.

Capítulo 6

Árvores

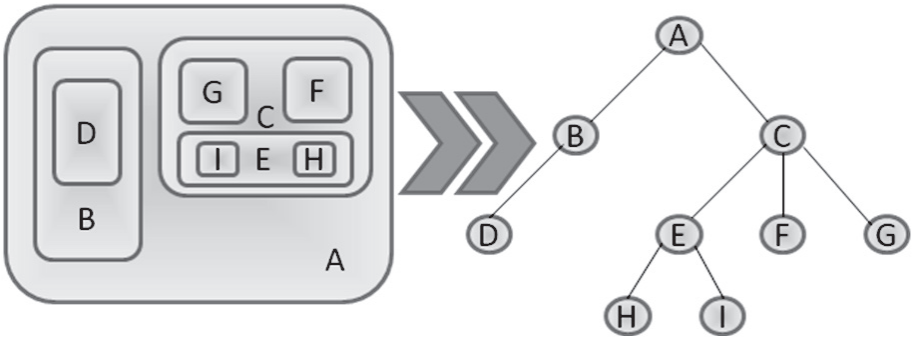
Objetivos

- Diversas aplicações requerem de uma organização e manipulação de dados mais complexa daquela propiciada através de estruturas lineares, analisadas na parte anterior. Uma árvore é uma estrutura de dados não linear muito eficiente para armazenar informação de forma a representar relacionamentos de aninhamento ou hierarquia entre os elementos envolvidos. Nesta parte são apresentados os conceitos iniciais relativos a árvores e a sua representação clássica e implementação através da definição do tipo abstrato correspondente. Árvores binárias de busca e balanceadas (AVL) são introduzidas.

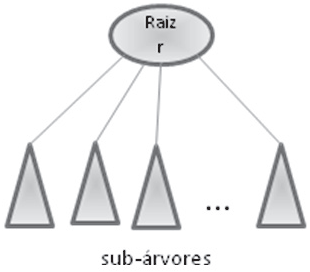
Introdução

Vetores e listas são estruturas de dados chamadas de unidimensionais ou lineares, e portanto, não são adequadas para representarmos dados que devem ser dispostos de maneira hierárquica. Por exemplo, a estrutura hierárquica de diretórios (pastas), aninhamento, etc. Estruturas de dados não lineares, como árvores, são ideais para representar este tipo de relacionamento.

A ilustração a seguir representa à esquerda o aninhamento de conjuntos e à direita a representação hierárquica deste aninhamento utilizando uma árvore.



Uma árvore é composta por um conjunto de nós. Existe um nó *r*, denominado raiz, que contém zero ou mais sub-árvores, cujas raízes são ligadas diretamente a *r*. Esses nós raízes das sub-árvores são ditos filhos do nó pai, *r*.



O número de filhos permitido por nó e as informações armazenadas em cada nó diferenciam os diversos tipos de árvores existentes:

- Árvores binárias, onde cada nó tem, no máximo, dois filhos.
- Árvores genéricas, onde o número de filhos é indefinido.

A forma mais natural para definirmos uma estrutura de árvore é usando recursividade. Uma árvore *T* é um conjunto finito de *n* nós ou vértices, tais que:

- Se *T* é um conjunto vazio, ou seja *n = 0*, a árvore é nula, ou vazia, ou
- Existe um nó especial, *r*, chamado raiz da árvore;
- Os demais nós constituem um conjunto vazio ou são particionados em conjuntos disjuntos não vazios, as sub-árvores de *r*;
- Cada sub-árvore, por sua vez, também é uma árvore.

Sejam *T* uma árvore e *v* um nó, tal que $v \varepsilon T$, T_v é a sub-árvore de *T* que tem *v* como raiz. Definem-se as seguintes propriedades:

- O grau de saída do no v é o número de sub-árvores que ele possui.
- O grau da árvore *T* é o maior grau dentre os de todos os seus nós.
- Os filhos de *v* são as raízes das sub-árvores de *v*.
- Um nó que não tem descendentes é chamado de folha ou terminal.

Uma propriedade fundamental de todas as árvores é que só existe um caminho da raiz para qualquer nó. Com isto, podemos definir a *altura* de uma árvore como sendo o comprimento do caminho mais longo da raiz até uma das folhas. Assim, a altura de uma árvore com um único nó raiz é zero.

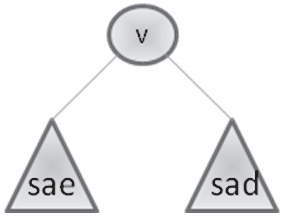
- Um *caminho* em *T* é uma sequência de nós *v1, v2*, ..., v_m, tal que para cada par (v_i, v_{i+1}), v_i é pai de v_{i+1}. O comprimento do caminho é *m-1*.
- O nível do nó *v* é o número de nós no caminho da raiz até *v*. O nível do nó raiz é 0, por definição.
- A altura de um nó *v* é o número de nós no maior caminho de v até um dos seus descendentes. Folhas têm altura 1. A altura da raiz determina a altura da árvore.

Pesquise sobre aplicações da estrutura de dados árvore.

1. Árvore binária

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

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

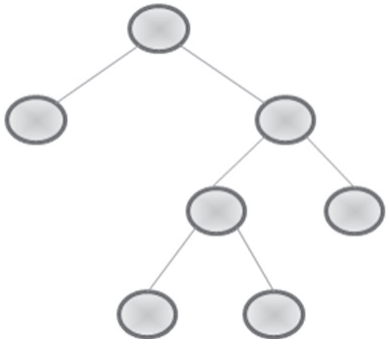


Pela definição, uma sub-árvore de uma árvore binária é sempre especificada como sendo a *sae* ou a *sad* de uma árvore maior, e qualquer das duas sub-árvores pode ser vazia.

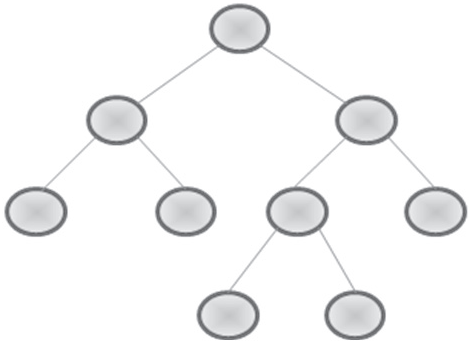
Uma árvore binária *T* é um conjunto finito de *n* nós ou vértices, tais que:

- Se *T* é um conjunto vazio, ou seja, *n=0*, a árvore é nula ou vazia, ou
- Existe um nó especial, *r*, chamado raiz da árvore;
- Os demais nós constituem um conjunto vazio ou são particionados em dois conjuntos disjuntos: sub-árvore esquerda e direita de *r,* cujas raízes são chamadas de filho esquerdo e direito de *r*;
- Cada sub-árvore, por sua vez, também é uma árvore binária;

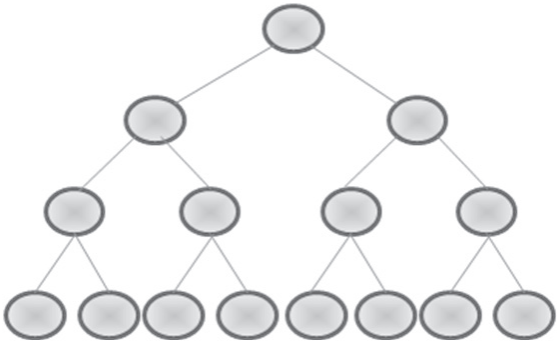
Uma *árvore estritamente binária* é aquela em que cada nó possui zero ou dois filhos, como representado na figura a seguir.



Uma *árvore binária completa* é aquela cujos nós com sub-árvore vazias localizam-se no último ou no penúltimo nível. Um exemplo de árvore binária completa é ilustrado na figura abaixo.

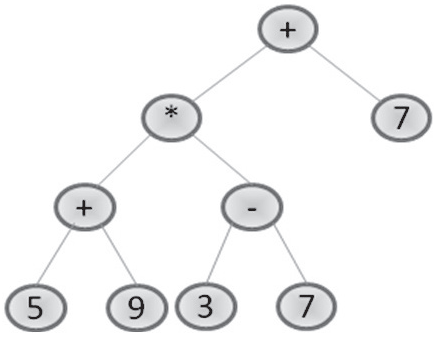


Uma *árvore binária cheia* é aquela cujos nós com sub-árvores vazias localizam-se todos no último nível. Logo, ela também é estritamente binária.



Um exemplo de utilização de árvores binárias está na avaliação de expressões. Nessa árvore, os nós folhas representam operandos e os nós inter-

nos operadores binários. Uma árvore que representa, por exemplo, a expressão (5 + 9) * (3 - 7) + 7 é ilustrada na seguinte figura:



Dependendo do percurso da árvore a mesma expressão pode ser representada em forma expressões infixas, prefixas e pósfixas.

No caso de percorrer uma árvore de forma infixa, inicialmente a sub-árvore esquerda é percorrida, em seguida o nó raiz é percorrido, por ultimo a sub-árvore direita é percorrida. No caso da árvore acima ser percorrida de forma infixa, o resultado seria: 5 + 9 * 3 - 7 + 7. Neste caso, o percurso resulta em uma expressão matemática válida.

No caso de percorrer uma árvore de forma prefixa, inicialmente o nó raiz é percorrido, em seguida a sub-árvore esquerda é percorrida, por ultimo a sub-árvore direita é percorrida. No caso da árvore acima ser percorrida de forma prefixa, o resultado seria: + * + 5 9 - 3 7 7. Observe que neste caso, o percurso não resulta em uma expressão matemática válida.

No caso de percorrer uma árvore de forma pósfixa, inicialmente a sub-árvore esquerda é percorrida, em seguida a sub-árvore direita é percorrida, por ultimo o nó raiz é percorrido. No caso da árvore acima ser percorrida de forma pósfixa, o resultado seria: 59 + 37 -* 7+. Observe que neste caso, o percurso também não resulta em uma expressão matemática válida.

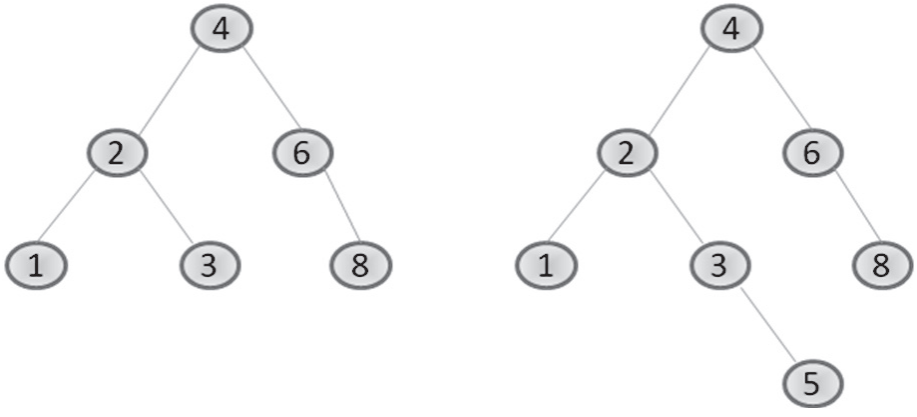
No próximo capítulo serão abordados estas formas de percursos em árvores.

Atividades de avaliação

1. Pesquise sobre aplicações da estrutura de dados árvore binária.

2. Árvore binária de busca

Uma árvore de busca é uma árvore binária com a propriedade de que para todo no *x* na árvore, os valores de todas as chaves na sub-árvore esquerda são menores ou iguais do que o valor chave em *x*, e que os valores de todas as chaves na sub-árvore direita são maiores ou iguais do que o valor chave em *x*. Esta definição se aplica recursivamente a cada nó da árvore. Assumimos que cada nó na árvore possui um valor chave, e em principio não é considerada a ocorrência de chaves repetidas. No caso das árvores representadas a seguir, a propriedade de ordenação pode ser verificada em todos os nós na árvore esquerda. Na árvore direita, a propriedade não se verifica uma vez que na sub-árvore esquerda do nó raiz aparece um nó com chave maior (5) àquela encontrada na raiz (4).



Como a profundidade média das árvores binárias de busca é *O(log n)*, as operações sobre elas executadas também.

Desde que todos os elementos na árvore seguem um critério de ordem os operadores <, > e = podem ser aplicados de forma a estabelecer comparações entre eles.

2.1. Definição do TAD Árvore Binária de Busca

O conjunto de operações necessário para a manipulação correta e satisfatória de uma árvore binária de busca é definido a seguir.

```
Interface do TAD ABB

// Retorna uma árvore vazia.
ABB Inicializar (void);
```

```
// Cria e retorna uma árvore inicializada.
ABB Rriar (element_type c, no_ptr e, no_ptr d);

//Insere um dado na árvore.
no_ptr Inserir (no_ptr pai, no_ptr filhoEsq);

/*Busca por um determinado elemento e retorna o nó correspondente, ou null caso não seja encontrado.
no_ptr Buscar (element_type x, ABB t);

// Retorna o conteúdo de um nó.
element_type Conteudo (no_ptr a);

// Retorna o filho esquerdo de um nó.
ABB retornaSAE (no_ptr a);

// Retorna o filho direito de um nó.
ABB retornaSAD (no_ptr a);

//Remove um elemento.
void Remove (element_type x, pai, ABB pai, ABB nó);

// Retorna 1 se o nó for nulo, ou 0 em caso contrário.
int Vazia (no_ptr a);
```

2.2. Implementação do TAD Árvore Binária de Busca

O armazenamento de árvores pode utilizar alocação dinámica ou estática, cujas vantagens e desvantagens foram analizadas em partes anteriores. No entanto, por se tratar de uma estrutura mais complexa o potencial desperdício de espaço pode ser reduzido pela utilização de ponteiros. A definição da estrutura de dados surge naturalmente a partir da definição recursiva da árvore. Cada nó na árvore possui, além do campo de informação, dois ponteiros, que apontam para cada uma das suas sub-árvores.

```
typedef struct no_Árvore *no_ptr;

typedef int element_type;
struct no_Árvore {
        element_type info;
```

```
        no_ptr esq;
        no_ptr dir;
};
```

Da mesma forma que uma lista encadeada é representada por um ponteiro para o primeiro nó, a estrutura da árvore como um todo é representada por um ponteiro para o nó raiz, a partir do qual todos os nós da árvore podem ser alcançados.

```
typedef no_ptr ABB;
```

Como uma árvore é representada pelo endereço do nó raiz, uma árvore vazia tem que ser representada pelo valor *NULL*.

```
ABB Inicializar (void){
        return NULL;
}
```

Para criar árvores não vazias, podemos ter uma operação que cria um nó raiz contendo a informação e os ponteiros às suas duas sub-árvores, à esquerda e à direita. Essa função tem como valor de retorno o endereço do nó raiz criado e inicializado, como segue:

```
ABB Criar (element_type c, ABB sae, ABB sad){
        p = (ABB) malloc (sizeof (no_Árvore));
        p -> info = c;
        p -> esq = sae;
        p -> dir = sad;
        return p;
}
```

As duas funções *Inicializar* e *Criar* representam os dois casos da definição recursiva de árvore binária: uma árvore binária é vazia (a = Inicializar ();) ou é composta por uma raiz e duas sub-árvores (a = Criar (c, sae, sad);).

As operações que possibilitam a consulta dos conteúdos dos campos que compõem o nó são apresentadas a seguir. O método *Conteudo* retorna

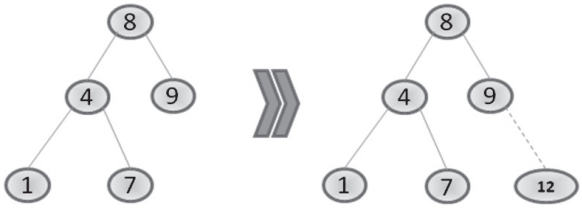
a informação contida no nó, enquanto que *retornaSAE* retorna a sub-árvore esquerda do nó. Analogamente o método *retornaSAD* retorna a sub-árvore direita.

```
element_type Conteudo (ABB a){
        return (a -> info);
}

        ABB retornaSAE (ABB a){
        return (a -> esq);
}
```

A busca por um conteúdo em uma árvore binária de busca leva em conta o critério de ordenação estabelecido entre os nós que a compõem. Desta forma, enquanto a chave procurada não for encontrada, se for menor do que a chave que se encontra no nó da árvore, a procura continua na sub-árvore da esquerda, e no caso contrário na sub-árvore direita, recursivamente.

```
no_ptr Buscar (element_type x, ABB T) {
  if (T == NULL) return NULL;
  if (x < T -> info)
        return (Buscar (x, T -> esq));
  else
        if (x > T -> info)
           return (Buscar (x, T -> dir));
        else return T;
}
```

O processo de inserção na árvore segue a mesma lógica do algoritmo *Buscar*. Se o elemento for encontrado nada é feito, uma vez que não estamos considerando a ocorrência de chaves repetidas. Em outro caso, o elemento é inserido. Note que seja qual for a chave a ser inserida, a inserção sempre acontece em um nó folha. O exemplo a seguir ilustra a inserção da chave com valor (12).

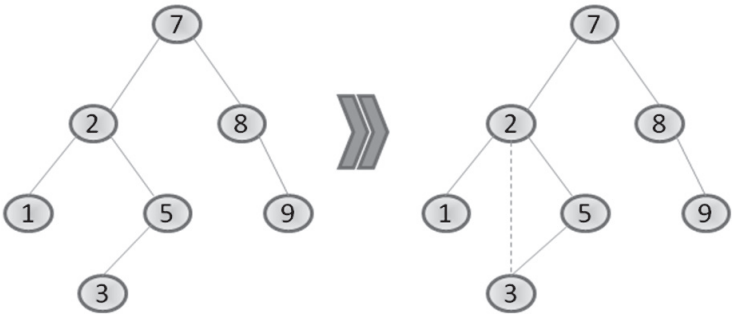


Duplicações podem ser manipuladas mantendo um campo extra no registro do nó indicando a freqüência da ocorrência. Esta solução é mais eficiente uma vez que replicar nós na árvore tornaria a árvore mais profunda aumentando o custo médio requerido nas operações sobre as árvores, além de alocar mais espaço de memória.

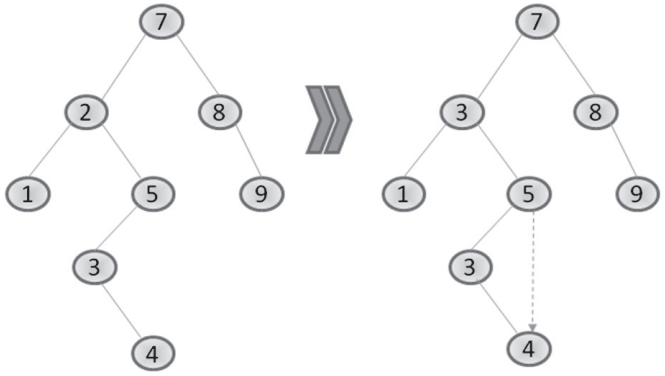
```
no_ptr Inserir (element_type x, ABB T) {
  if (T == NULL) {
    T = (ABB) malloc (sizeof (no_Árvore));
        T -> element = x;
        T -> esq = NULL;
        T -> dir = NULL;
  }
  else
     if (x < T -> info)
           T -> esq = Inserir (x, T -> esq);

     else
        if (x > T -> info)
              T -> dir = Inserir (x, T -> dir);
  return T;
}
```

Na remoção várias possibilidades devem ser consideradas. Se o nó que vai ser removido é folha a remoção é imediata.

Se o nó tem somente um filho ele pode ser removido ajustando os ponteiros entre seu pai e seu filho de modo de realizar um *by pass* encima do nó. Por exemplo, considere a árvore a seguir, onde o nó correspondente ao dado (5) precisa ser removido. O processo a ser seguido é descrito na árvore que aparece à direita.



Se o nó a ser removido tem dois filhos a estratégia geral consiste em reemplazar a chave do nó com a menor chave da sub-árvore direita (ou maior da sub-árvore esquerda), e recursivamente remover esse nó. Como o filho mais esquerdo da sub-árvore direita não pode ter filho esquerdo, a segunda remoção é mais fácil. O exemplo a seguir ilustra a remoção do nó cujo conteúdo é (2), que possui dois filhos. Nesse caso o nó é substituído pelo filho mais a esquerda da sub-árvore direita (ou o menor dos maiores). Posteriormente, o nó utilizado na substituição precisa ser removido (3) da sub-árvore correspondente.



O método *Remover* recebe por parâmetro o ponteiro ao nó que será removido. Este ponteiro pode ter sido obtido a partir da execução da operação de busca por um conteúdo específico. O algoritmo também precisa do ponteiro correspondente ao nó pai do nó que será removido, já que o respectivo ponteiro precisa ser ajustado.

O código realiza dois passes na árvore para encontrar e remover o menor nó da sub-árvore direita. Esta ineficiência pode ser removida escrevendo uma função *delete_min.*

void Remover (element_type x, ABB pai, ABB no){

```
        no_ptr tmp_no, filho;

        if (no -> esq != null && v -> dir != null) {
                tmp_cell = buscaMin (no -> dir);
                no -> info = tmp_cell -> info;
                no -> dir = Remover (no -> info, no -> dir);
        }

        else {
                tmp_no = no;
                if(no -> esq == NULL
                        filho = no -> dir;
                if(no -> dir == NULL )
                        filho = no -> esq;
                if (pai -> dir ==T)
                        pai -> dir = filho;
        else
        pai -> esq = filho;

        free (tmp_no);
    }
}
```

O método auxiliar que procura o nó que contem o conteúdo mínimo possui duas versões, uma recursiva e outra iterativa.

```
    no_ptr buscaMin (ABB T){
            if (T == NULL ) return NULL;

            else
              if (T -> esq == NULL )
                    return (T);
             else return (buscaMin (T -> esq));
    }
```

```
no_ptr buscaMin (ABB T){
if (T != NULL)
        while (T -> esq != NULL)
        T = T -> esq;
        return T;
}
```

Os dois algoritmos possuem uma lógica simples, no entanto a versão não recursiva pode ser mais eficiente em termo de custo espacial e temporal do que a versão recursiva.

Escreva uma função que verifique se uma árvore é cheia. Uma árvore é dita cheia se todos os nós que não são folhas têm os dois filhos, isto é, não pode existir nó com apenas um filho. A função deve retornar 1 no caso da árvore ser cheia ou 0 no caso de não ser. No caso da árvore ser vazia, a função deve retornar 1.

```
int cheia (tree_ptr a) {
  if (vazia (a))
        return (1);
  else
        if ((vazia (retornaSAE (a)) && !vazia
                (retornaSAD (a)) ||((!vazia (retornaSAE (a))
                                        && vazia (retornaSAD(a))
                return (0);
        else
        return cheia (retornaSAE (a)) && cheia
                                        (retornaSAD(a));
}
```

2.3. Ordens de percurso em árvores binárias

O percurso de todas as sub-árvores executando alguma ação de tratamento em cada nó, pode ser feito seguindo uma das seguintes ordens:

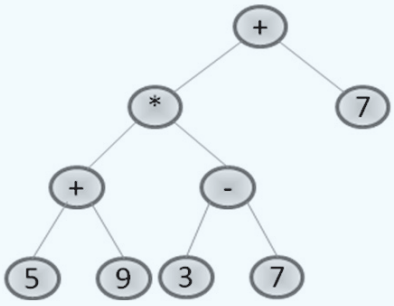
- *pré-ordem:* trata *raiz*, percorre *sae*, percorre *sad*;

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

Por exemplo, no caso de imprimir o conteúdo dos nós de uma árvore binária, os algoritmos que implementam esta ação de acordo com cada percurso são apresentados a continuação.

```
void ImprimirPosordem (no_ptr a){
        if ( !Vazia(a) ){
                ImprimirPosordem (retornaSAE(a));
                ImprimirPosordem (retornaSAD(a));
                printf (Conteudo(a));
        }
        return;
}

void ImprimirSimetrica (no_ptr a){
        if ( !Vazia(a) ){
                ImprimirSimetrica (retornaSAE(a));
                printf (Conteudo(a));
                ImprimirSimetrica (retornaSAD(a));
        }
        return;
}

void ImprimirPreordem (no_ptr a){
        if ( !Vazia(a) ){
                printf (Conteudo(a));
                ImprimirPreordem (retornaSAE(a));
                ImprimirPreordem (retornaSAD(a));
        }
        return;
}
```

Por exemplo, dada a árvore de expressão a seguir, como resultado dos percursos apresentados, as expressões resultantes são as seguintes:



Inordem: 5 + 9 * 3 - 7 + 7

Preordem: + * +5 9 – 3 7 7

Posordem: 5 9 + 3 7 - * 7 +

Atividades de avaliação

1. Crie a árvore binária de acordo com a seguinte sequência de números, na ordem:

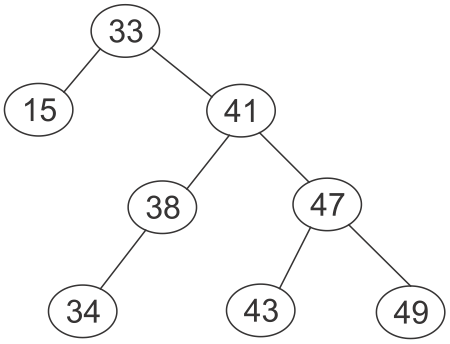
a) 1, 2, 3, 4, 5, 6, 7. Analise a árvore binária obtida em termo de desempenho na execução das operações sobre ela.

b) 20, 5, 12, 36, 27, 45, 9, 2, 6, 17, 40.

c) A partir da árvore obtida no inciso anterior, remover os nós: 9, a seguir o 5, e finalmente o 20.

2. Implemente um algoritmo que determine se uma árvore binária de busca é efetivamente uma árvore binária de busca.
3. Implemente o método que retorna o maior elemento a partir de um determinado nó cujo cabeçalho é: `no_ptr buscaMAX (ABB T)`. Implemente o método na sua versão recursiva e não recursiva.
4. Dada uma árvore binária de busca, onde cada nó é constituído pelas seguintes informações: NOME, SEXO ('M' ou 'F'), IDADE e PESO. Sabendo que a árvore foi construída com a chave NOME e que já existe um ponteiro chamado RAIZ que aponta para o nó raiz da árvore, construir um algoritmo que, a partir desta árvore, gere duas listas ordenadas por NOME, uma para homens e outra para mulheres.
5. Escreva um algoritmo recursivo que encontre o maior valor armazenado em uma árvore binária de busca já construída.
6. Adapte os algoritmos de inserção e remoção em árvores binárias de busca de forma a tratar a ocorrência de conteúdos-chave repetidos, mantendo um contador de ocorrências em cada nó.

7. Para a árvore binária a seguir, escreva as sequências de nós visitados após a execução dos percursos pré-ordem, inordem e pós-ordem. Codifique os algoritmos correspondentes a cada um dos percursos.



8. Utilizando as operações definidas na interface do TAD ABB de números, escreva um método que retorne quantos nós de uma ABB armazenam valores contidos em um intervalo $[x_1, x_2]$.

Texto Complementar

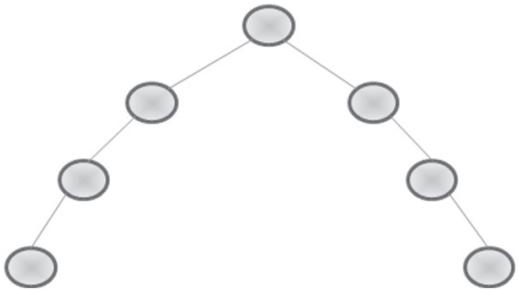
Relação entre o número de nós de uma árvore binária e sua altura

A cada nível o número potencial de nós numa árvore binária vai dobrando, de forma que para uma altura *h* da árvore existe um número máximo de nós, dado por:
$2^0 + 2^1 + 2^2 + ... + 2^{h-1} + 2^h = 2^{h+1} - 1$ nós
Portanto, uma AB de altura *h* pode ter no máximo $O(2^h)$ nós. A partir desta definição temos que o número de nós em uma AB é dada por $n = 2^h$ nós. Para despejar *h* temos que aplicar a definição de logaritmo: $\log n = h . \log 2$. Portanto uma árvore binária com *n* nós pode ter uma altura mínima de $O(\log n)$.
Por outro lado, se a árvore tem altura *h*, deve existir um caminho de comprimento *h* da raiz até um dos nós, digamos $n^0, n^1, ... n^h$, e todos os *h+1* nós deste caminho devem ficar em níveis diferentes. Assim, a árvore deverá ter pelo menos *h+1* nós.
Esta relação entre o número de nós e a sua altura é importante, pois significa que a partir da raiz, qualquer nó pode ser alcançado em no máximo $O(\log n)$ passos. Note que se tivéssemos *n* nós numa lista linear o número máximo de passos seria $O(n)$.
A altura da árvore é uma medida do tempo necessário para encontrar um nó. Esta propriedade atinge a eficiência máxima quando a árvore binária é balanceada, ou seja, todos os nós internos, ou quase todos possuem dois filhos.
É fácil prever que após várias operações de inserção e remoção, a árvore tende a ficar desbalanceada. Em especial, a operação de remoção numa ABB dependendo da estratégia utilizada (substituição do nó a ser removido pelo maior da sub-árvore esquerda ou o menor da sub-árvore direita), favorece sistematicamente uma das sub-árvores.
A solução a este problema consiste em sempre manter a altura das sub-árvores no mínimo, ou próximo do mínimo. Para isso é necessário de processos de inserção e remoção mais complexos que mantenham as sub-árvores balanceadas.

3. Árvores AVL

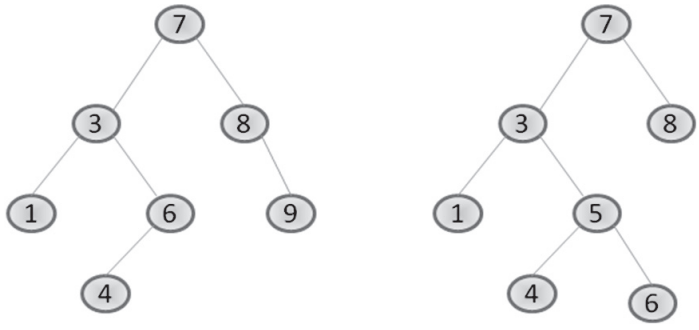
Uma AVL é uma árvore binária de busca dinamicamente *equilibrada* ou balanceada, na qual se busca manter, a um custo razoável, um tempo de busca próximo àquele que se conseguiria se a árvore fosse completa, o que garante que a altura da árvore é *O(log n).* O nome AVL deve-se aos seus criadores, os matemáticos ADEL'SON-VEL'SKII e LANDIS.

A propriedade de balanceamento consiste em manter as sub-árvores esquerda e direita de cada nó, na mesma altura, podendo diferir no máximo em 1 nível. A figura a seguir ilustra uma árvore, onde a raiz se encontra balanceada uma vez que a suas sub-árvores esquerda e direita possuem a mesma altura, no entanto os nós internos não satisfazem essa propriedade.



Com esta restrição, todas as operações sobre árvores podem ser executadas em tempo *O(log n),* exceto possivelmente a inserção. No entanto, para manter a propriedade de balanceamento, além dos algoritmos de percurso, inclusão e exclusão já discutidos, são necessários algoritmos que restabeleçam o equilíbrio após inclusões e exclusões, caso algum nó fique desregulado. Por exemplo, na inserção, pode ser preciso atualizar a informação de balanceamento para os nós no caminho de volta para a raiz, pois somente aqueles nós tiveram as suas sub-árvores alteradas.

No caso das árvores binárias de busca representadas a seguir, a árvore da esquerda se encontra balanceada uma vez que todos os nós mantem suas sub-árvores com uma diferença de até um nível. Já a árvore da direita apresenta um desbalanceamento na raiz.



A inserção de 6.5 na primeira árvore provocará o desbalanceamento do nó 8. A propriedade de balanceamento é restaurada através de operações de ***rotação***.

Seja α o no desbalanceado. Desde que todo nó tem no máximo dois filhos e o desbalanceamento da altura requer que a altura das duas sub-árvores deferem em 2, a violação da propriedade pode acontecer como consequência de operações de inserção ou remoção.

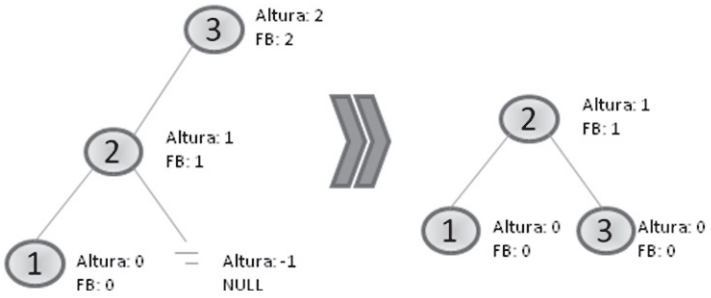
O *fator de balanceamento* (FB) de um determinado nó *r* é calculado como a diferencia entre a altura da sub-árvore esquerda de *r* e da sub-árvore direita de *r*. Se o valor obtido for menor ou igual a 1 o nó está balanceado. Caso contrário o nó se encontra desbalanceado. O FB de um nó folha é 0.

Algoritmo que calcula a altura de um nó

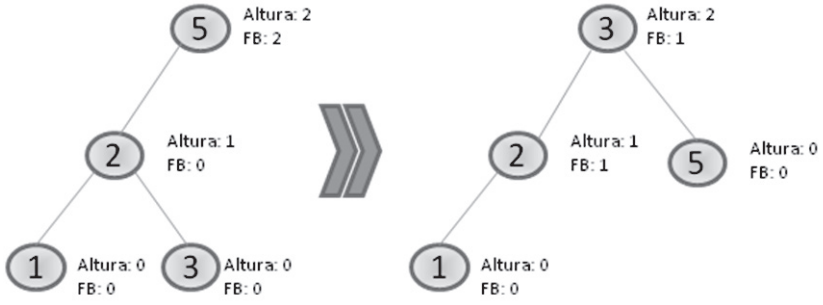
```
int Altura (ABB no){
int Alt_Esq, Alt_Dir;

if (no = NULL)
        Altura := -1
else {
   Alt_Esq := Altura (retornaSAD (no));
   Alt_Dir := Altura (retornaSAE (no));

   if (Alt_Esq > Alt_Dir)
      Altura := 1 + Alt_Esq
   else Altura := 1 + Alt_Dir;
 }
}
```

No caso em que for constatado o desbalanceamento em um determinado nó, quatro situações possíveis podem ser constatadas.

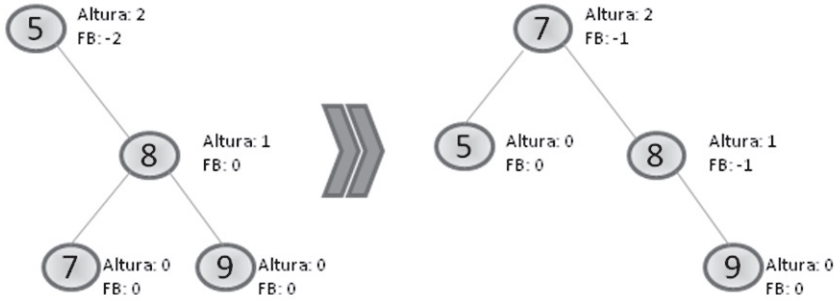
- **Tipo I** : Se a sub-árvore esquerda é maior que a sub-árvore direita (FB > 1), e a sub-árvore esquerda desta sub-árvore esquerda é maior que a sub-árvore direita dela, então realizar uma **rotação simples para a direita**;



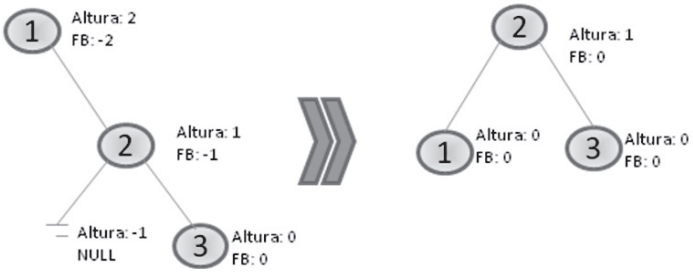
- **Tipo II** : Se a sub-árvore esquerda é maior que a sub-árvore direita (FB > 1), e a sub-árvore esquerda desta sub-árvore esquerda é *menor ou igual* que a sub-árvore direita, então realizar uma **rotação dupla para a direita**;



- **Tipo III**: Se a sub-árvore esquerda é menor que a sub-árvore direita (FB < -1), e a sub-árvore direita desta sub-árvore direita é *menor ou igual* que a sub-árvore esquerda dela, então realizar uma **rotação dupla para a esquerda**;



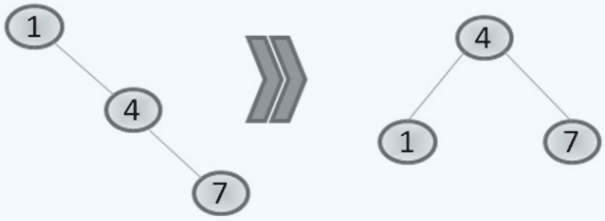
- **Tipo IV**: Se a sub-árvore esquerda é menor que a sub-árvore direita (FB < -1), e a sub-árvore direita desta sub-árvore direita é *maior* que a sub-árvore esquerda dela, então realizar uma **rotação simples para a esquerda**.



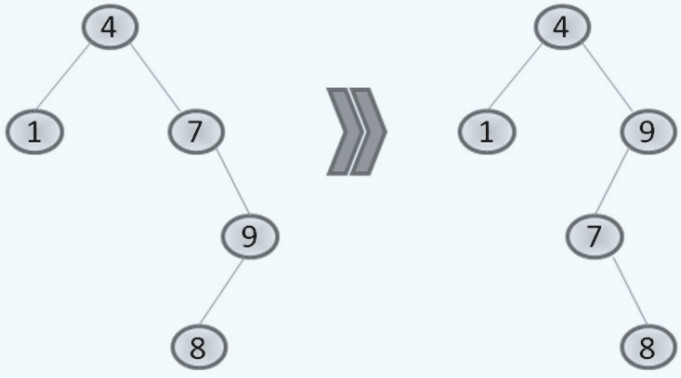
A partir das situações apresentadas, qualquer tipo de desbalanceamento pode ser corrigido aplicando uma das 4 rotações descritas.

Exemplo

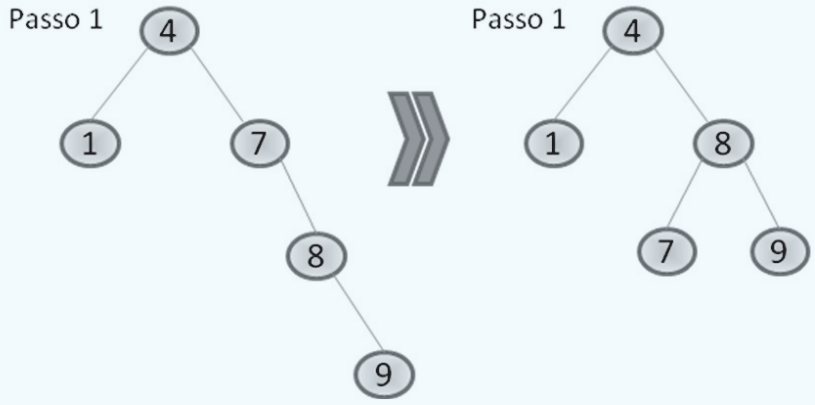
Começando a partir de uma árvore vazia, são inseridos os números de (1), (4) e (7). O primeiro problema acontece na inserção do (7): a propriedade de balanceamento é violada na raiz. Para resolver é executada uma rotação simples a esquerda.



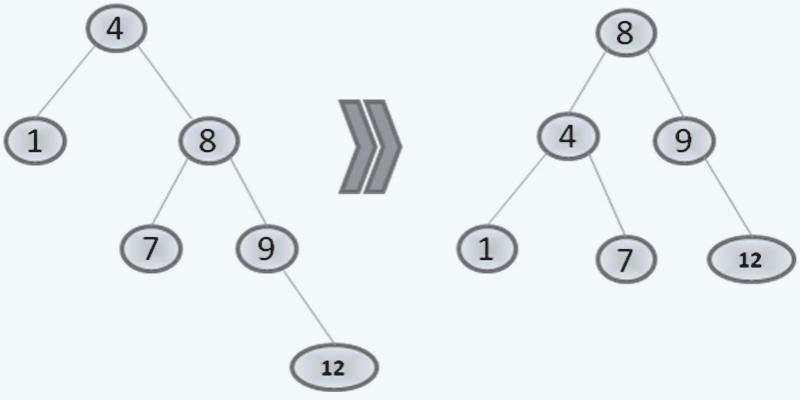
A seguir é inserida a chave (9), sem prejuízo do balanceamento da árvore. A inserção do (8) provoca uma nova violação no nó (7) (e também na raiz da árvore). Repare que neste caso uma rotação simples a esquerda não resolve o problema.



O desbalanceamento no nó (7) é resolvido em dois passos, através de uma rotação dupla a esquerda. No primeiro passo é realizada uma rotação simples a direita envolvendo a sub-árvore do nó desbalanceado. Já no segundo passo, uma rotação a esquerda envolvendo o nó desbalanceado devolve o balanceamento à árvore.

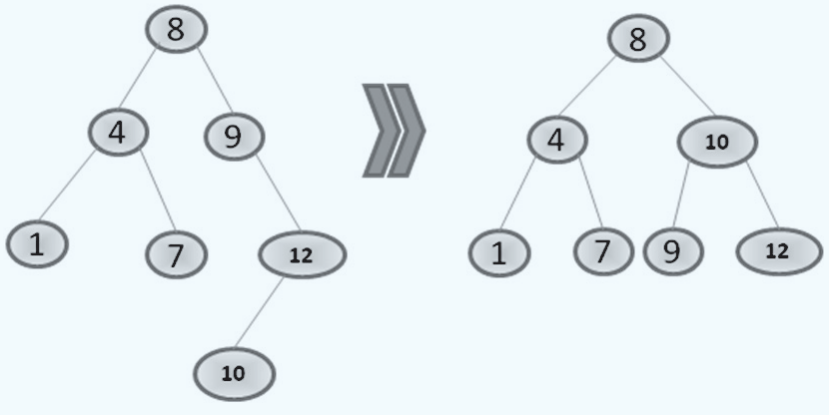


A inserção do (12) provoca o desbalanceamento da raiz desde que a sub-árvore esquerda de altura 0, enquanto que a direita tem altura 2. para resolver é executada uma rotação simples a esquerda.



Como resultado da rotação, o nó com a chave (8) se torna a nova raiz da árvore, e com isso, a sua sub-árvore esquerda (7) se transforma na nova sub-árvore direita do (4).

Continuando o exemplo, a inserção da chave (10) desbalanceia o nó (9), desencadeando uma nova rotação dupla a esquerda.



É importante salientar que se a árvore não fosse AVL, o resultado das inserções teria gerado uma árvore de altura 5, enquanto que a AVL obtida a partir da mesma sequencia de inserção possui altura 2.

Atividades de avaliação

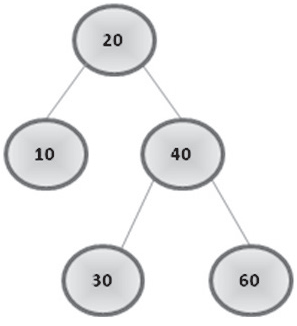
1. Considerando que a altura (h) de uma árvore é dada pelo caminho mais longo desde a raiz até uma folha, explique com as suas palavras a relação existente entre altura (h) de uma árvore binária e a quantidade (mínima e máxima) de nós.

a) Qual a relação entre a altura (h) de uma árvore e o tempo requerido (custo) para encontrar um nó?

b) Em qual situação a busca em uma árvore binária pode atingir eficiência máxima?

2. Defina uma árvore AVL estabelecendo as suas propriedades, vantagens e desvantagens da sua utilização.

3. Dada a seguinte árvore (binária de busca) AVL, simule o processo de inserção e remoção das seguintes chaves na árvore na ordem dada: inserção 25 - inserção 70 - inserção 15 - inserção 12 - inserção 18 - remoção 15. Verifique a cada passo se a propriedade de balanceamento continua sendo mantida. Em caso de desbalanceamento, indicar o nó desbalanceado e as operações de rotação indicadas para resolver o problema.



4. Considerando a seguinte sequência de caracteres: A, Z, B, Y, C, X, D, P, E, V, F. Simule a construção passo a passo de uma árvore AVL de caracteres, e indique em que situações ocorrem rotações simples ou duplas, à direita ou à esquerda, dos vários elementos da árvore.

5. Considerando a seguinte sequência de números: inserção 50, inserção 20, inserção 30, inserção 25, inserção 10, inserção 15, remoção 20, inserção 20, inserção 40. Simule a construção passo a passo de uma árvore AVL de números com a sequência acima, e indique em que situações ocorrem rotações simples ou duplas, à direita ou à esquerda, dos vários elementos da árvore.

Síntese do capítulo

Neste capítulo foram apresentados os conceitos fundamentais sobre árvores e árvores binárias. O TAD Árvore Binária de Busca foi definido e implementado, e alguns exemplos de utilização foram ilustrados, assim como os diferentes percursos possíveis sobre a estrutura.

Adicionalmente, a conceituação sobre árvores binárias balanceadas, e sua importância em relação ao aumento no desempenho das operações realizadas foi relatada. A propriedade de balanceamento e as estratégias para restabelecer o equilibrio na árvore foram detalhados e ilustrados.

Referências

AHO, J.E. Hopcroft, and J.D. Ullman. **Data structures and algorithms**. Addison-Wesley, Reading, Mass., 1983.

CORMEN T. H., LEISERSON C. E., RIVEST R. L., STEIN C. (2001). **Introduction to Algorithms**. McGraw-Hill e The Mit Press.

HOROWITZ and S. Sahni (1987). **Fundamentals of data structures**, Computer Science Press. Editora Campus.

KNUTH D. E. (1968). **The Art of Computer Programming**, Vol. 1: Fundamental Algorithms. Addison-Wesley.

SZWARCFITER J. I., MARKENZON L. (2010). **Estruturas de Dados e Seus Algoritmos**. 3ª. Edição. LTC.

ZIVIANI N. (2005). **Projeto de Algoritmos com implementações em Pascal e C**, 2da. Edição. Thomson.

Capítulo 7

Busca avançada

Objetivos

- A eficiência alcançada para resolver o problema da busca ou recuperação de informação a partir de uma estrutura de dados é um indicador da eficiência da estrutura. Nesta parte são apresentados métodos específicos de busca que objetivam reduzir a complexidade em relação aos métodos de busca padrão apresentados em partes anteriores. Em particular, é apresentado o uso de tabelas de índices que possibilitam o acesso direto à informação, idealmente. No domínio especifico de tratamento de cadeias a busca digital é apresentada como solução para o problema de casamento de padrões. Finalmente, o funcionamento de estruturas auto-ajustáveis é descrito.
-

1. Tabela de dispersão

O armazenamento e recuperação da informação são possivelmente as funcionalidades mais importantes requeridas de um computador. De forma geral, a informação é organizada em estruturas que possibilitam que os dados possam ser recuperados da memória e interpretados quando necessário.

A pesquisa por um determinado dado requer que seja estabelecido um critério, que geralmente é baseado na existência de uma *chave de pesquisa*[15] que possibilita que ocorrências da informação sejam corretamente identificadas.

Diversas estratégias podem ser utilizadas para realizar uma pesquisa por registros em uma tabela. A escolha pela mais adequada depende principalmente das necessidades e características da aplicação específica. Os dois principais fatores que influenciam nesta escolha são o tamanho da entrada, ou seja a quantidade de elementos a serem processados e as operações mais frequentemente executadas sobre a estrutura.

[15] A chave de pesquisa é o campo do registro a partir do qual o registro pode ser referenciado de forma unívoca. Cada registro no conjunto possui um campo chave único o que possibilita a sua identificação a partir dele.

Existem diversos métodos de pesquisa amplamente utilizados. Dentre eles, a pesquisa sequencial é o método mais simples onde, a partir do primeiro registro, a pesquisa é realizada em forma sequencial seguindo a ordem apresentada pelos elementos. O processo continua até que a chave for encontrada ou a tabela for percorrida completamente sem sucesso. Esta estratégia foi utilizada no método de busca implementado no TAD Lista. O esforço requerido nesta busca é *O(n),* uma vez que no pior cenário, a lista precisará ser percorrida na sua totalidade.

A árvore de busca e suas variantes abordada na Parte 4 é uma estrutura de dados muito eficiente para armazenamento e recuperação de informação. Neste caso, o custo demandará em média *O(log n).*

Tanto a pesquisa sequencial como a aplicada em árvores de busca, são baseadas na comparação entre chaves. Diferentemente, a técnica baseada em transformação de chave ou *hashing* utiliza uma função de transformação aritmética a partir da qual uma chave é mapeada para um endereço de memória utilizando as chamadas *tabelas de dispersão* ou *tabelas de hash*. A seguir o funcionamento da tabela de dispersão é apresentado.

Seja, por exemplo, a distribuição de expedientes de funcionários de uma empresa ao longo de um arquivo de pastas, onde cada pasta indica a inicial do sobrenome do funcionário. A principio é considerado que não existe ordem para a colocação dos expedientes dentro de cada pasta do arquivo. Nesse caso o sobrenome seria a chave e a inicial o endereço de armazenamento.

Em ciência da computação a tabela de dispersão (de *hashing*, no inglês), é uma estrutura de dados especial, que associa chaves de pesquisa a valores. Seu objetivo é, a partir de uma chave simples, fazer uma busca rápida e obter o valor desejado em tempo constante.

A utilização de tabela de dispersão para o armazenamento e recuperação da informação visa tornar estas operações mais eficientes em termos de esforço, em relação aos outros mecanismos de busca estudados, alcançando uma complexidade média de *O(1).* O ganho com relação a outras estruturas associativas (como um vetor simples) passa a ser maior conforme a quantidade de dados aumenta. Em contrapartida, em uma tabela de dispersão é virtualmente impossível estabelecer uma ordem para os elementos. Em outras palavras, a função de dispersão estabelece uma indexação sobre as chaves mas não preserva a ordem entre elas.

O método de pesquisa com uso de transformação de chave envolve duas etapas.

1. Desenvolver e aplicar a função de transformação aritmética do valor da chave para um endereço na memória.
2. Dependendo da quantidade de chaves e da eficiência da função de transformação na geração de endereços, pode ser necessário elaborar uma estratégia de tratamento para colisões[16] para tratar os casos em que duas chaves gerem o mesmo endereço.

Considere a existência de *n* chaves a serem armazenadas na tabela *T*, de dimensão *m*. A tabela é considerada uma estrutura sequêncial, portanto as posições ou endereços se encontram no intervalo [0, *m – 1*]. Se o número de chaves *n* for igual ao número de posições na tabela, *m*, e, além disso, os valores das chaves forem de 0 a *m – 1*, então, cada chave *x* poderia ser armazenada no endereço *x* correspondente.

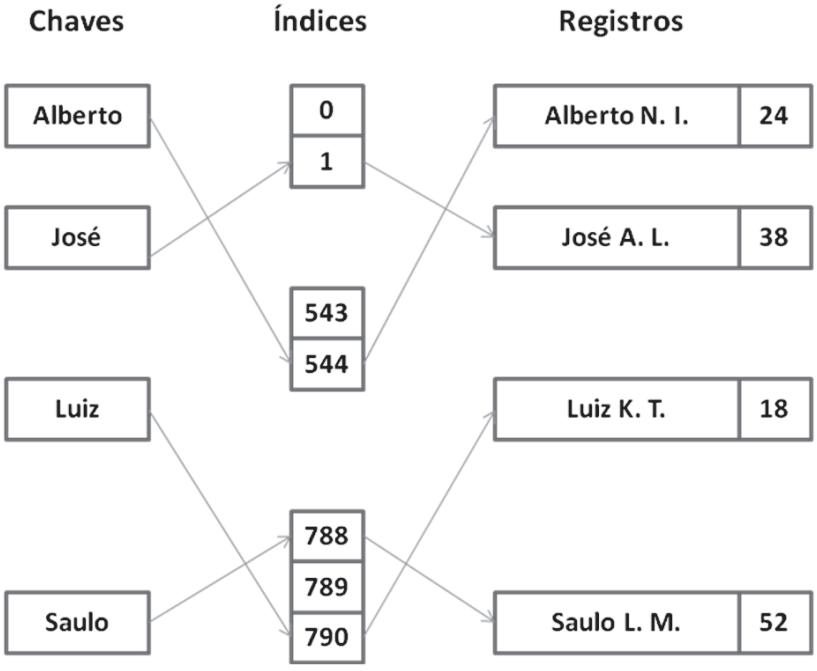
Tabelas de dispersão são tipicamente utilizadas para indexação de grandes volumes de informação, tais como bases de dados. Outros exemplos de uso das tabelas de dispersão são as tabelas de transposição em jogos de xadrez para computador.

[16] A ocorrência de colisões pode ser abordada do ponto de vista probabilistico. O paradoxo do aniversário estabelece que, se tomadas aleatoriamente 50 pessoas, pelo menos duas pessoas teriam a mesma data de aniversário ou colisão.

1.1. A função de dispersão

A função de dispersão é a responsável por gerar um índice a partir de determinada chave. A definição da função é de fundamental importância, uma vez que se não for adequada para o tratamento dos dados em questão, a manipulação da tabela terá um mau desempenho.

O ideal para a função de dispersão é que sejam sempre fornecidos índices únicos para as chaves de entrada. A função perfeita seria a que, para quaisquer entradas A e B, sendo A diferente de B, fornecesse saídas diferentes. Quando as entradas A e B são diferentes e, passando pela função de dispersão, geram a mesma saída, acontece uma *colisão*. De forma simplificada temos que: *Pos (elemento) = H (elemento, n),* onde *H* é a função de dispersão aplicada ao elemento, considerando o tamanho *n* da tabela.



No exemplo, a função de dispersão[17] é aplicada sobre a chave nome, para obter um índice de acesso ao vetor onde o registro é armazenado. O registro pode agregar vários campos de informação, além do campo chave.

No entanto, na prática, é difícil encontrar uma função de dispersão perfeita que consiga espalhar de forma uniformemente esparsa as chaves ao longo da estrutura. Dando sequência ao exemplo acima, a aplicação da função de dispersão para o nome *Luiza* pode vir a gerar o mesmo índice do que *Luiz* (*790*), provocando uma colisão. Esta situação é indesejável uma vez que reduz o desempenho do sistema. No entanto é muito comum acontecer, é por isso que diversas técnicas de tratamento de colisões tem sido propostas na literatura.

[17] Na prática, funções de dispersão perfeitas ou quase perfeitas são encontradas apenas onde a colisão é intolerável, como por exemplo em aplicações de criptografia, ou quando o conteúdo da tabela armazenada é conhecido previamente.

Exemplo de função de dispersão

Uma função de dispersão muito simples envolve transformar um caracter em um valor numérico. Isto, na linguagem C, poderia ser feito da seguinte forma:

```
int hashExemplo(char *chave)        {
        return (chave[0]-65);
}
```

Dada sua simplicidade esta função causaria muitas colisões, no entanto pode ser utilizada como parte de uma função mais complexa que possibilite um melhor espalhamento dos dados.

Atividades de avaliação

1. Defina o conceito de tabela de dispersão e descreva o processo envolvido na aplicação desta técnica.
2. Estabeleça vantagens e desvantagens em relação a outros mecanismos de armazenamento e recuperação de informação.
3. Pesquise na literatura sobre funções de dispersão comumente utilizadas e que tem demostrado desempenho aceitável.

a) Método da divisão

b) Método da dobra

4. Apresente um exemplo prático de geração de uma tabela de dispersão utilizando os métodos pesquisados no item anterior.

O tratamento das colisões envolve geralmente a utilização de alguma outra estrutura de dados em conjunção com as tabelas de dispersão, tal como uma lista encadeada ou até mesmo árvore árvores balanceadas (AVL). Em outras oportunidades a colisão é solucionada dentro da própria tabela.

1.2. Estratégias para resolução de colisões

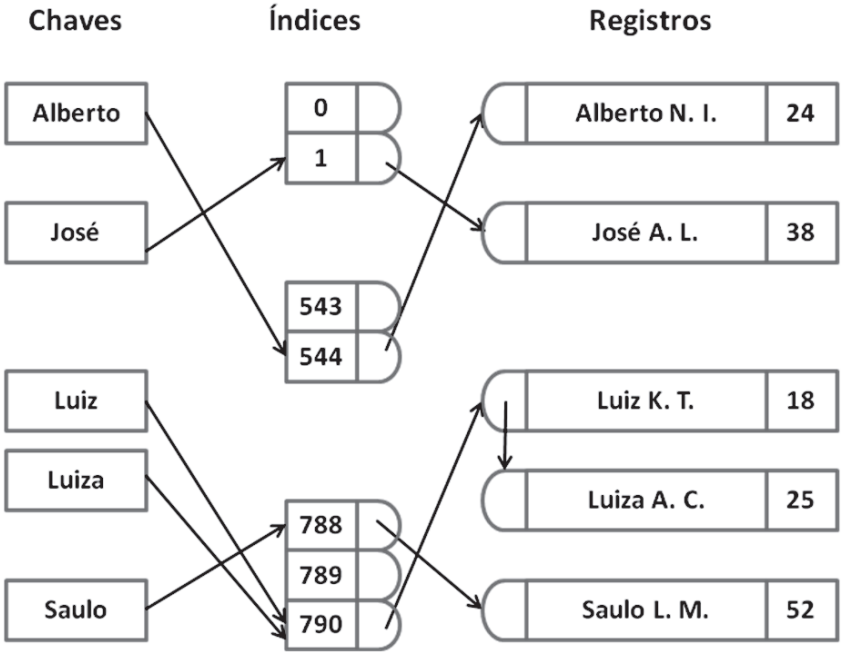
Considerando que a ocorrência de colisões é praticamente inevitável, um bom método de resolução de colisões é essencial, independentemente da qualidade da função de dispersão utilizada.

Há diversos algoritmos de resolução de colisão, mas os mais conhecidos são os de encadeamento e sondagem.

a) Encadeamento

A utilização de listas encadeadas é a solução mais simples para o tratamento de colisões. Neste caso, a partir do índice em conflito é mantido um ponteiro para uma lista encadeada onde são armazenados os registros em conflito. A inserção na tabela requer e inserção dentro da lista encadeada. Analogamente, a remoção requer atualizar os índices dentro da lista. O TAD lista na sua versão encadeada através de ponteiros foi apresentado na Parte 3.

Graficamente, a solução de colisões através de listas encadeadas é representada a seguir. A situação de colisão entre as chaves *Luiz* e *Luiza* seria resolvida adicionando o registro correspondente à chave repetida na lista associada ao índice.



Esta solução adiciona um nível de indireção às operações de inserção e recuperação da informação, uma vez que o registro não mais é acessado diretamente a partir da chave, no entanto, o custo de acesso continua se mantendo baixo. Estruturas de dados alternativas podem ser utilizadas no lugar das listas encadeadas. Por exemplo, a partir da utilização de árvores binárias balanceadas (AVL) é possível melhorar o tempo médio de acesso da tabela dispersão para *O(log n)* ao invés de *O(n)* demandado no caso da utilização de listas.

b) Técnicas de sondagem

Em contrapartida à técnica de encadeamento, as técnicas de sondagem para o tratamento de colisões não fazem uso de nenhuma estrutura auxiliar para o armazenamento da informação. Em caso de colisão, os registros em conflito são armazenados dentro da própria tabela, utilizando buscas padronizadas até encontrar um registro vazio ou o registro buscado.

Outras formas mais complexas de implementar a técnica de sondagem consiste em determinar a posição do novo elemento em colisão a partir de uma função quadrática, incrementando o índice exponencialmente. Desta forma, caso a chave procurada não se encontre na posição 10, em uma segunda tentativa será procurada na posição 100, 1000, e assim por diante.

Uma terceira possibilidade envolve a aplicação de uma nova função de dispersão (também chamado de *double hashing*), cuja chave de entrada será o valor gerado pela função anterior. Esta solução pode ser útil em casos muito específicos, com enormes quantidades de dados, no entanto a sobrecarga no sistema nem sempre justifica a experiência.

Atividades de avaliação

1. Explique o conceito de colisão e por que acontecem.
2. Pesquise as diferentes formas de resolução de colisão e analise suas vantagens e desvantagens. Exemplifique.

2. Busca digital

O problema de busca geralmente considera a comparação entre uma chave desejada e as chaves que compõem um conjunto, que pode ser estruturado de formas convenientes no intuito de melhorar o desempenho das operações. Diferentemente, no caso da busca digital, a chave é constituida de um conjunto de caracteres ou dígitos pertencentes a um alfabeto[18] apropriado. Neste caso específico, a comparação entre chaves é realizada dígito a dígito, individualmente.

[18] As cadeias aparecem no processamento de texto em linguagem natural, dicionários, sequenciamento de DNA, processamento de imagens, etc. Em cada domínio, um alfabeto específico é utilizado. Exemplos de alfabetos: {0, 1}, {a, b, c, ..., z}, {0, 1, ... 9}.

A busca digital funciona de forma similar à busca em dicionários, decompondo a palavra letra a letra (caracter ou dígito), onde a primeira letra da palavra determina um índice de página onde se encontram as palavras iniciadas por aquela letra.

Os métodos de busca digital são particularmente úteis quando as chaves são grandes e de tamanho variável. A partir desta pesquisa é possível localizar todas as ocorrências de uma determinada cadeia em um texto, com tempo de resposta $O(\log n)$ em relação ao tamanho do texto. Este problema é conhecido como **casamento de cadeias**, no contexto de **processamento de cadeias de caracteres**.

O **processamento de cadeias de caracteres** envolve duas classes de problemas: casamento de cadeias (do inglês, *pattern matching*) e compressão de cadeias.

O problema de casamento de cadeias envolve a procura pela ocorrência de um determinado padrão em um texto que está sendo editado. Formalmente, o texto T é considerado um vetor de tamanho n e o padrão P um vetor de tamanho m, com $m <= n$. Os elementos que compõem T e

P pertencem a um alfabeto finito de tamanho c. Dadas duas cadeias *T* e *P*, deseja-se saber as ocorrências de *P* em *T*.

A compressão de texto está relacionada com a representação de um texto original de forma a ocupar menos espaço, ou seja utilizando um número menor de bits.

Métodos mais modernos de compressão de cadeias possibilitam o acesso direto a texto comprimido sem necessidade de descomprimir o texto, permitindo melhorar a eficiência de sistemas de recuperação de informação e aumentar a economia de espaço.

No, método de busca digital, tanto o padrão procurado quanto o texto são pré-processados a partir da construção de índices de forma a reduzir a complexidade das operações para um custo *O(log n)*. No entanto, o tempo de pré-processamento é compensado por muitas operações de busca.

A seguir são apresentados brevemente os tipos de índices mais conhecidos para o pré-processamento de cadeias de forma a agilizar o desempenho das buscas.

2.1. Árvore digital

A estrutura mais apropriada para realizar a busca digital é através da árvore digital.

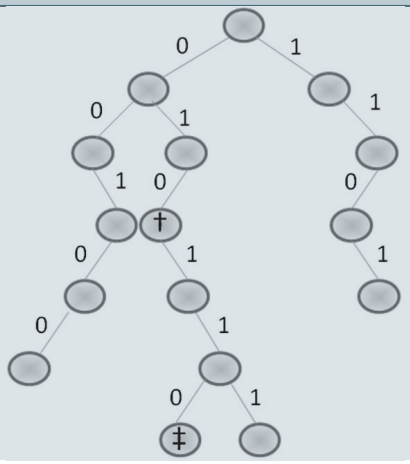
Formalmente, uma sequência $S = \{s_1, ..., s_n\}$ um conjunto de *n* chaves em que cada s_i é formada por uma sequência de elementos d_j denominados dígitos. Supõe-se que existe em *S* o total de *m* dígitos distintos que determinam o alfabeto ordenado de *S*. Os primeiros *p* dígitos de uma chave compõem o prefixo de tamanho *p* da chave.

Uma árvore digital para S é uma árvore m-ária *T*, não vazia, tal que:

3. Se o nó *v* é o j-éssimo filho de seu pai, então *v* corresponde ao dígito d_j do alfabeto *S*, $1 <= j <= m$.
4. Para cada nó *v*, a sequência de dígitos definida pelo caminho desde a raiz de *T* até *v* corresponde a um prefixo de alguma chave de *S*.

Na análise de um texto em linguagem natural, *S* seria o conjunto de frases do texto, onde s_i é cada frase que pode ser buscada e *n* o número de frases, e $m = 26$.

Em um caso específico da árvore digital, no entanto o mais utilizado, temos a *árvore digital binária*, onde o grau da árvore é $m = 2$. O alfabeto considerado neste caso é {0, 1}. A partir da construção da árvore digital com estas características, as chaves a serem consideradas nas buscas envolvem sequências binárias.

Chaves	Árvore digital binária
00 0010 010 010111 010110 00100 11 110 1101	

A partir da árvore gerada é possível observar que algumas chaves binárias são prefixos de outras chaves pertencentes à mesma coleção. Por exemplo, o caminho da raiz até o nó com conteúdo † correspondente à chave 010 que é prefixo da chave 010110 correspondente ao caminho até o nó ‡.

Cada nó na árvore pertence a uma chave (ou mais) do conjunto. Por exemplo o nó com † corresponde à chave 010, e faz parte (prefixo) das chaves 010111 e também 010110. De acordo com a implementação da árvore digital, cada nó na árvore envolve um campo adicional contendo um ponteiro para a informação correspondende à chave detectada. Por exemplo, no caso do nó † existiria um ponteiro de forma a localizar no texto a chave 010. Os ponteiros de nós que não correspondem ao último dígito de uma chave válida seriam nulos (NULL).

Uma vez criada a árvore, a busca acontece como resultado do percurso a partir da raiz: o filho esquerdo representa o dígito 0 e o filhor à direita representa o dígito 1. Note que a busca localiza não somente uma chave válida completa, mas também prefixos. A busca por prefixos pode ser de grande utilidade dependendo da aplicação, como por exemplo linguística.

Atividades de avaliação

1. Explique como funciona a busca digital e seu papel no contexto de processamento de cadeias. Mencione áreas de aplicação deste tipo de busca.
2. Pesquise sobre compressão de textos em linguagem natural.
3. Crie graficamente uma árvore digital binária a partir do seguinte conjunto de chaves: 00, 0000, 00010, 00011, 0101100, 0101101, 10, 101, 1010.

3. Estruturas autoajustáveis

As operações de inserção e remoção de elementos aplicadas sobre uma determinada estrutura de dado afetam necessariamente a forma da estrutura. Já a operação de busca é inocua no sentido em que, a principio, não produz nenhuma alteração na forma da estrutura. No caso de estruturas autoajustáveis, a operação de busca pode alterar a forma da estrutura de dado objetivando melhorar o desempenho em buscas futuras. Por exemplo, no caso de ser detectada certa frequência na procura por um determinado componente em uma lista ou árvore, a posição ou nível do componente na lista ou na árvore, respectivamente, pode ser alterado, de forma a agilizar as futuras buscas pelo mesmo componente.

A partir deste comportamento, a complexidade ordinária de uma operação calculada individualmente, e de forma independente, para o pior caso na sua execução não é mais adequada. Em contrapartida, a *complexidade amortizada* considera a configurações da estrutura ao longo de uma sequência de operações executadas, avaliando as consequências de cada execução, de forma acumulada.

O conceito de autoajuste pode ser aplicado a diversas estruturas de dados, tais como listas, conjuntos e árvores.

3.1. Listas autoajustáveis

O TAD Lista foi abordado na Parte 3. Como foi visto, o tipo lista pode ser implementado utilizando a abordagem de alocação de memória estática (vetores) ou dinámica (ponteiros). Considerando que na sua versão mais simples não é estabelecido um critério de ordenação entre os elementos, alguns elementos na lista podem ser requisitados mais frequentemente do que outros. A partir desta constatação, uma lista autoajustável implementa estratégias para reduzir o tempo de acesso em operações subsequentes. A estratégia geral consiste em posicionar os nós mais procurados mais próximos do inicio da lista, de forma que em futuras buscas possam ser alcançados mais rapidamente. Esta estratégia pode ser implementada utilizando diversos métodos.

O método de *mover para frente* transfere o nó procurado para o inicio da lista. Repare que dependendo da implementação utilizada para implementar a lista, esta operação pode ser custosa. Por outro lado, nós com baixa probabilidade de acesso podem ser eventualmente acessados, tornando mais custosas as buscar por nós com maior probabilidade.

No método de *transposição* uma vez acessado o nó procurado, este é transferido para a posição imediatamente anterior. Na medida em que o nó for acessado, mais próximo do inicio será posicionado.

A implementação do método de *contador de frequências* envolve a incorporação de um campo adicional na estrutura do nó da lista, responsável por manter o número de acessos efetuados ao respectivo nó. Este campo é utilizado como chave para a ordenação decrescente dos elementos na lista. Ou seja que os nós mas frequentemente acessados são localizados no inicio, e portanto mais rapidamente alcançados.

Alguns métodos híbridos envolvem a combinação dos métodos anteriores de forma a tirar vantagem dos beneficios e reduzir as desvantagens dos métodos no seu formato individual.

Atividades de avaliação

1. Explique o funcionamento das estruturas de dados autoajustáveis.
2. Quais seriam as adaptações requeridas para que o TAD Lista apresentado na Parte 3 se torne TAD Lista Autoajustável?

a) Defina a nova estrutura de dados.

b) Implemente ou adapte as operações necessárias.

3. Pesquise sobre uma outra estrutura estudada ao longo desta disciplina que possa se tornar autoajustável. Descreva as características de funcionamento e as adaptações necessárias.

Síntese do capítulo

Nesta parte foram apresentadas diversas estratégias para armazenamento e recuperação de informação. Inicialmente foram apresentadas as tabelas de dispersão ou hash, como uma solução para o acesso direto ou semidireto à informação a partir da geração de índices, obtidos como resultado de um processo de transformação tendo como base o valor chave do registro. Mecanismos de tratamento de colisões no caso de geração de índices repetidos foram indicados.

Em relação ao processamento de cadeias de caracteres, foi apresentada a teoria sobre busca digital, que possibilita estabelecer o casamento de chaves, constituídas por caracteres ou dígitos, e texto. Com este objetivo, a estrutura de árvore digital foi apresentada.

Finalmente, o funcionamento de estruturas autoajustáveis, com foco na sua exemplificação através de listas foi descrito, indicando as estratégias de implementação a serem seguidas.

De forma geral, todos estes mecanismos e estratégias objetivam tornar mais eficiente o armazenamento e posterior recuperação da informação, de forma a tornar os algoritmos mais acessíveis e capazes de lidar melhor com a complexidade cada vez maior requerida pelas aplicações atuais.

Referências

AHO, J.E. Hopcroft, and J.D. Ullman. **Data structures and algorithms**. Addison-Wesley, Reading, Mass., 1983.

CORMEN T. H., LEISERSON C. E., RIVEST R. L., STEIN C. (2001). **Introduction to Algorithms**. McGraw-Hill e The Mit Press.

HOROWITZ and S. Sahni (1987). **Fundamentals of data structures**, Computer Science Press. Editora Campus.

KNUTH D. E. (1968). **The Art of Computer Programming**, Vol. 1: Fundamental Algorithms. Addison-Wesley.

SZWARCFITER J. I., MARKENZON L. (2010). **Estruturas de Dados e Seus Algoritmos**. 3ª. Edição. LTC.

ZIVIANI N. (2005). **Projeto de Algoritmos com implementações em Pascal e C**, 2da. Edição. Thomson.

Sobre a autora

Mariela Inés Cortés: É doutora em Informática pela Pontifícia Universidade Católica do Rio de Janeiro (2003) e mestre em Sistemas de Computação pelo Intituto Militar de Engenharia do Rio de Janeiro (1999). Sua alma mater é a Universidade Nacional de La Plata, onde completou os estudos de graduação em Ciências da Computação. Especialista na área de Engenharia de Software, atualmente é professora adjunta na Universidade Estadual do Ceará, vinculada ao Curso de Ciências da Computação, onde ministra dentre outras, a disciplina de Estrutura de Dados. Adicionalmente, coordena o Laboratório de Qualidade e Padrões de Software (LAPAQ) e lidera o Grupo de Engenharia de Software e Sistemas Inteligentes (GESSI).