TIPOS ABSTRATOS DE DADOS

ALGORITMO

Um **algoritmo** pode ser visto como uma seqüência de ações expressas em termos de uma linguagem de programação, constituindo parte da solução de um tipo determinado de problema. Ou seja, um algoritmo corresponde à descrição do padrão de comportamento associado aos **elementos funcionais ativos** de um processamento e deve ser expresso em termos de um conjunto finito de ações especificadas por meio de uma linguagem de programação.

ESTRUTURAS DE DADOS

Na linguagem de programação adotada, **estruturas de dados** dão suporte à descrição dos **elementos funcionais passivos** do padrão de comportamento acima referido, complementando o **algoritmo** que constitui parte da solução do problema considerado. Ambos, algoritmo e estruturas de dados, compõem o **programa** a ser executado pelo computador.

Não é útil estudar estruturas de dados sem considerar os algoritmos básicos a elas associados. Da mesma forma, a escolha de um algoritmo depende, em geral, da estrutura de dados associada.

PROGRAMA

Programar é, basicamente, estruturar dados e construir algoritmos.

Um **programa** é uma formulação concreta, em termos de uma linguagem de programação, de um **procedimento** abstrato que atua sobre um **modelo de dados** também abstrato. Ambos, procedimento e modelo de dados são representações abstratas de algoritmo e estruturas de dados, respectivamente, e **não** devem ser expressos em termos de uma linguagem de programação.

TIPOS DE DADOS

Em uma linguagem de programação, é importante **classificar** constantes, variáveis e valores gerados por expressões/funções de acordo com o seu **tipo de dados**.

Um tipo de dados deve caracterizar o conjunto de valores a que uma constante pertence ou o conjunto de valores que pode ser assumido por uma variável ou gerado por uma expressão/função.

Um tipo de dados **elementar** (ou **simples**) é caracterizado por um conjunto -**domínio -** de valores indivisíveis.

Um tipo de dados **estruturado** (ou **complexo**) define, em geral, uma coleção homogênea (de mesmo tipo) de valores elementares/estruturados **ou** um agregado de valores de tipos diferentes. Um exemplo é o tipo *lista*.

TIPO ABSTRATO DE DADOS

Abstraída qualquer linguagem de programação, um **tipo abstrato de dados** (*TAD*) pode ser visto como um modelo matemático que encapsula um **modelo de dados** e um conjunto de **procedimentos** que atuam com exclusividade sobre os dados encapsulados. Em nível de abstração mais baixo, associado à implementação, esses procedimentos são implementados por subprogramas denominados *operações*, *métodos* ou *serviços*.

Qualquer processamento a ser realizada sobre os dados encapsulados em um *TAD* só poderá ser executada por intermédio dos procedimentos definidos no modelo matemático do *TAD*, sendo esta restrição a característica operacional mais útil dessa estrutura.

Nesses casos, um programa baseado em *TAD* deverá conter algoritmos e estruturas de dados que implementem, em termos da linguagem de programação adotada, os procedimentos e os modelos de dados dos *TADs* utilizados pelo programa.

Assim, a implementação de cada *TAD* pode ocupar porções bem definidas do programa: uma para a definição das estruturas de dados e outra para a definição do conjunto de algoritmos.

Nessas condições, quaisquer alterações realizadas na estrutura de um dado *TAD* não afetarão as partes do programa que utilizam esse *TAD*.

NÍVEIS DE ABSTRAÇÃO

Uma coleção de atividades, tais como *inserir*, *suprimir* e *consultar*, encapsulada junto com uma estrutura passiva, como um *dicionário* (conjunto de verbetes), pode ser considerada um **tipo abstrato de dados** (*TAD*).

Definido dessa forma, o *TAD DICIONÁRIO* fica representado no nível de abstração mais alto possível: o **nível conceitual**.

Em um nível de abstração mais baixo, denominado **nível de design**, a estrutura passiva deve ser representada por um **modelo de dados** (por exemplo: seqüência ou árvore binária de busca) e as operações devem ser especificadas através de **procedimentos** cuja representação não dependa de uma linguagem de programação.

Em um nível de abstração ainda mais baixo, denominado **nível de implementação**, deve-se tomar como base o design do *TAD* e estabelecer representações concretas para os elementos de sua estrutura em termos de uma linguagem de programação específica.

No exemplo do *TAD DICIONÁRIO*, se o design escolhido para a estrutura passiva for a árvore binária de busca, será possível implementar esse modelo de dados, por exemplo, em termos de uma estrutura de dados do tipo *lista* correspondendo à representação prefixada *(raiz sub-árvore-esquerda subárvore-direita)*. Os procedimentos serão implementados por meio dos algoritmos apropriados às operações
raiz, *esquerda*, *direita* etc. utilizando-se os recursos disponíveis para codificar estruturas ativas (operações e controle). As duas estruturas, a ativa e a passiva, do *TAD* poderão eventualmente constituir um encapsulamento (módulo) específico e identificável no programa construído.

MOTIVAÇÃO

Uma razão importante para programar em termos de *TAD* é o fato de que os elementos da estrutura passiva do *TAD* são acessíveis somente através dos elementos da estrutura ativa.

No caso do *TAD DICIONÁRIO*, por exemplo, o acesso a qualquer elemento da estrutura de dados pode ocorrer apenas via os algoritmos correspondentes às operações *inserir*, *eliminar* e *consultar*.

Essa restrição conduz a uma forma eficiente de **programação defensiva**, protegendo os dados encapsulados no *TAD* contra manipulações inesperadas por parte de outros algoritmos.

Uma segunda razão, também importante, para programar em termos de *TAD* é o fato de que seu uso permite introduzir alterações nas estruturas definidas no nível de implementação - visando, por exemplo, aumento de eficiência - livre da preocupação de gerar erros no restante do programa. Tais erros não ocorrem porque a única conexão entre o *TAD* e o restante do programa é aquela constituída pela interface **imutável** dos algoritmos que implementam a estrutura ativa do *TAD*.

Por fim, pode-se dizer que um *TAD* bem construído pode tornar-se uma porção de código confiável e genérica, permitindo e aconselhando seu **reuso** em outros programas. Dessa forma, aumenta-se a produtividade na construção de programas e, sobretudo, garante-se a qualidade dos produtos gerados.

COMPLEXIDADE DE ALGORITMOS

Teoria da Complexidade Computacional: consiste em estudar o custo computacional para resolver problemas interessantes, medindo-se a quantidade de recursos necessários: tempo e espaço em memória. A partir desse estudo pode-se determinar se há algoritmos mais rápidos ou não para o problema.

Análise de Algoritmos: consiste na análise de recursos usados por um determinado algoritmo. Custos de ordem exponencial não são desejáveis. Algoritmos de ordem polinomial são os melhores.

Metodologia para análise de algoritmos

Umas das formas de se analisar a eficiência de um algoritmo é implementá-lo e testá-lo com conjuntos de dados de entrada de tamanhos variados, observando e registrando o tempo gasto pelo programa para cada conjunto. O objetivo é determinar a dependência existente entre o tempo de execução e o tamanho do conjunto de entrada fornecido.

Para analisar esses resultados podemos plotar o desempenho de cada função, relacionando o tamanho da entrada de dados (x) com o tempo gasto no processamento (y). Nesse tipo de análise, é importante que sejam escolhidos conjuntos de dados de entrada relevantes e que sejam realizados testes em número suficiente para que as conclusões obtidas sejam consideradas válidas do ponto de vista estatístico.

Geralmente, o tempo de execução de um algoritmo cresce à medida que aumenta o tamanho do conjunto de dados fornecido como entrada. No entanto, esse crescimento pode acontecer de formas diferentes. Há diversos fatores que podem influenciar no desempenho de um algoritmo:

- o problema a ser resolvido;
- o sistema operacional da máquina onde o programa será executado;
- a linguagem de programação escolhida;
- o compilador (interpretador) usado;

- o hardware da máquina (processador, freqüência, memória, disco, etc) no qual o programa será executado;
- a habilidade e eficiência do programador;
- o algoritmo desenvolvido.

Um algoritmo possui duas medidas de complexidade:
- Espacial: quantidade de memória que ele utiliza durante sua execução;
- Temporal: é, aproximadamente, o número de instruções que ele executa.
Ambas as complexidades são em função do número de entradas.

Para medir o custo de execução de um algoritmo é comum definir uma função de custo ou **função de complexidade f**.

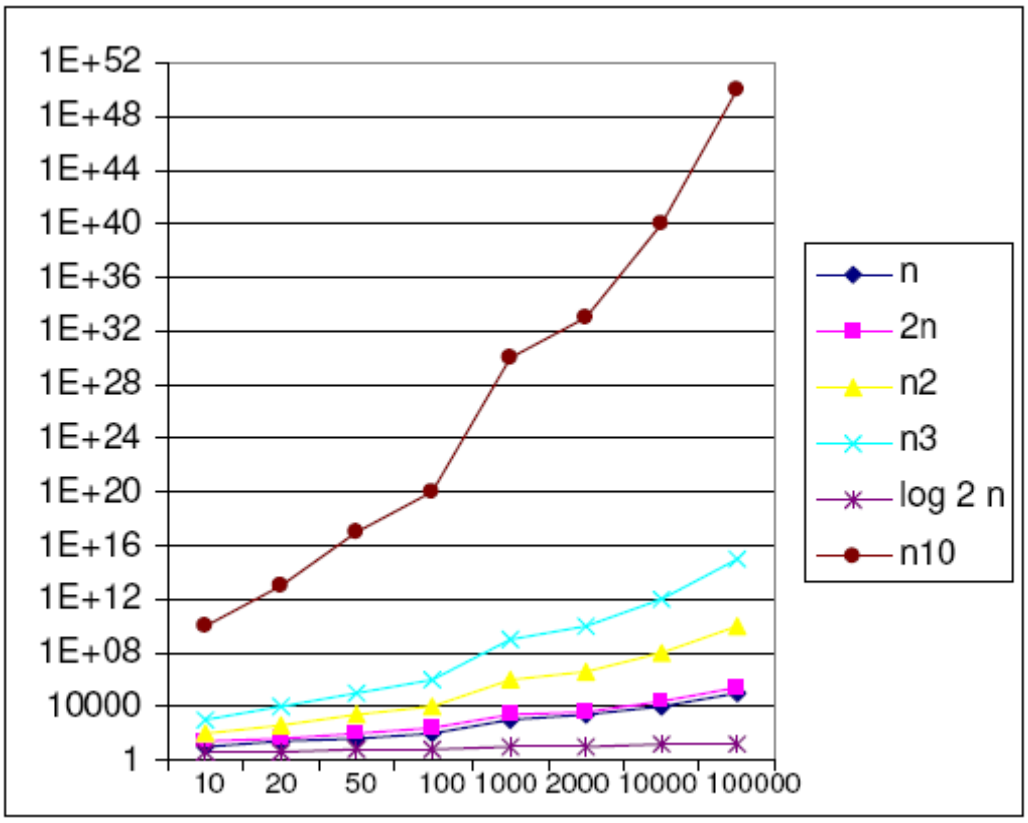
- f(n) é a medida do tempo necessário para executar um algoritmo para um problema de tamanho n.

Função de complexidade de tempo: f(n) mede o tempo necessário para executar para executar um algoritmo em um problema de tamanho n.

Função de Complexidade de Espaço f(n): mede a memória necessária para executar um algoritmo em um problema de tamanho n.

Quando dizemos que uma função f(n) define o desempenho de um algoritmo, queremos dizer que esse algoritmo é executado em um tempo proporcional a n, ou seja, que qualquer entrada de tamanho n nunca excede cn, onde c é uma constante que depende do hardware e do software usado no experimento.

O gráfico abaixo exibe o desempenho das funções n, 2n, n^2, n^3, $\log_2 n$, n^{10} para tamanho de entrada como 10, 20, 50, 100, 1000, 2000, 10000, 100000.



Para usar uma metodologia analítica devemos considerar os seguintes elementos:

- Uma linguagem para a descrição dos algoritmos;
- Um modelo computacional para execução desses algoritmos;
- Uma métrica para medir o tempo de execução desses algoritmos;
- Uma abordagem para caracterizar os tempos de execução, incluindo os algoritmos recursivos.

Para que possamos analisar os algoritmos devemos definir os critérios de avaliação desses algoritmos, ou seja, o conjunto de operações primitivas que permitem medir a complexidade dos algoritmos. As operações primitivas consideradas são as seguintes:

- Atribuição;
- Chamadas de subprogramas;
- Operações aritméticas;
- Operações lógicas;
- Acesso a um elemento de um vetor ou matriz;
- Referência a um dado na memória (uso de apontadores);
- Retorno de um subprograma.

Ao invés de determinar o tempo de execução de cada operação primitiva, vamos apenas contá-las.

Essa abordagem simples dá origem a um modelo computacional conhecido como Máquina de Acesso Aleatório (*RAM - Random Acess Machine*). Nesse modelo pressume-se que a máquina pode executar qualquer operação primitiva em um número constante de passos que não depende do tamanho da entrada. Assim, o número de operações primitivas que o algoritmo realiza corresponde diretamente ao tempo de execução desse algoritmo no modelo RAM.

Contagem de operações primitivas:

```
Algoritmo MaiorElemento(int A[],int n) : int
{
      // Entrada: Vetor A de inteiros de comprimento n
      // Saída: maior valor do vetor

      int maior = A[1]

      para int i = 2 até n faça{
            se A[i] > maior então maior = A[i]
      }
      retorna maior
}
```

Vamos analisar o algoritmo MaiorElemento e determinar o número de operações que ele realiza.

```
int maior = A[1]
```

- O trecho de inicialização acima corresponde a **duas operações primitivas**: acesso a um elemento do vetor e atribuição de valor a uma variável. Como a operação não é executada dentro de um laço de repetição, totaliza apenas **2 unidades na contagem.**

```
para int i = 2 até n faça
```

- O índice i é inicializado com 2. Isso corresponde a **uma operação primitiva** (atribuição de valor a uma variável).
- Antes de iniciar a repetição a comparação i<=n é verificada, isso corresponde a uma operação primitiva (expressão lógica). Como o contador é inicializado em 2 e incrementado em uma unidade ao final de cada iteração, a operação i<=n é executada n vezes. Portanto, essa instrução contabiliza ***n* operações primitivas**. Totalizando, então **n+1 unidades na contagem**.

```
se A[i] > maior então maior = A[i]
```

- Essa linha corresponde a quatro instruções primitivas (acesso a um elemento do vetor, expressão lógica, novamente acesso a um elemento do vetor e atribuição).
- A cada iteração o contador é incrementado, isso corresponde a duas instruções primitivas (operação aritmética de adição e atribuição).
- Como a execução da atribuição à variável *maior* depende do resultado da expressão lógica, a cada iteração podem ser executadas 4 ou 6 instruções primitivas.
- Como o bloco do laço de repetição será executado ***n-1*** vezes, o número de instruções executadas ficará entre ***4(n-1)*** e ***6(n-1)***.

```
retorna maior
```

- A instrução de retorno corresponde a **uma operação primitiva.**

Desta forma, o número de instruções primitivas será no mínimo:

$$2 + (n + 1) + 4(n-1) + 1 = 4 + n + 4n - 4 = \mathbf{5n}$$

E no máximo:

$$2 + (n + 1) + 6(n-1) + 1 = 4 + n + 6n - 6 = \mathbf{7n-2}$$

O melhor caso (**5n**) ocorre quando a variável maior não tem seu valor alterado, pois o maior elemento está na primeira posição do vetor.

O pior caso (**7n-2**) ocorre quando o vetor está em ordem crescente e assim a variável maior é atualizada a cada iteração.

Exercício: Conte o número de instruções executadas pelos algoritmos abaixo:

1)
```
Algoritmo SomaElementos(int A[],int n) : int
{
      // Entrada: Vetor A de inteiros de comprimento n
      // Saída: soma dos elementos do vetor

      int soma = 0

      para int i = 1 até n faça{
            soma = soma + A[i]
      }
      retorna soma
}
```

2)
```
Algoritmo Pesquisa(int A[],int n, int valor) : int
{
      // Entrada: Vetor A de inteiros de comprimento n e um valor
      // Saída: o índice do elemento procurado, se for encontrado.
      //        -1, em caso contrário

      para int i = 1 até n faça{
            se A[i] == valor então Retorna i
      }
      retorna -1
}
```