

NIVEL SUPERIOR



Quem Somos

A Domina Concursos, especialista no desenvolvimento e comercialização de apostilas digitais e impressas para Concurso Públicos, tem como foco tornar simples e eficaz a forma de estudo. Com visão de futuro, agilidade e dinamismo em inovações, se consolida com reconhecimento no segmento de desenvolvimento de materiais para concursos públicos. É uma empresa comprometida com o bem-estar do cliente. Atua com concursos públicos federais, estaduais e municipais. Em nossa trajetória, já comercializamos milhares de apostilas, sendo digitais e impressas. E esse número continua aumentando.

MISSÃO

Otimizar a forma de estudo, provendo apostilas de excelência, baseados nas informações de editais dos concursos públicos, para incorporar as melhores práticas, com soluções inovadoras, flexíveis e de simples utilização e entendimento.

VISÃO

Ser uma empresa de Classe Nacional em Desenvolvimento de Apostilas para Concursos Públicos, com paixão e garra em tudo que fazemos.

VALORES

- Respeito ao talento humano
- Foco no cliente
- Integridade no relacionamento
- Equipe comprometida
- Evolução tecnológica permanente
- Ambiente diferenciado
- Responsabilidade social



HABILITADA P/ IMPRESSÃO



PROIBIDO CÓPIA

Não é permitida a revenda, rateio, cópia total ou parcial sem autorização da Domina Concursos, seja ela cópia virtual ou impressa. Independente de manter os créditos ou não, não importando o meio pelo qual seja disponibilizado: link de download, Correios, etc...

Caso houver descumprimento, o autor do fato poderá ser indiciado conforme art. 184 do CP, serão buscadas as informações do responsável em nosso banco de dados e repassadas para as autoridades responsáveis.



Conhecimentos específicos

*"Camuflar um erro seu é
anular a busca pelo
conhecimento. Aprenda
com eles e faça novamente
de forma correta."*

Nara Nubia Alencar

Algoritmos e Estruturas de Dados

Em ciência da computação tipos de variáveis ou dados é uma combinação de valores e de operações que uma variável pode executar, o que pode variar conforme o sistema operacional e a linguagem de computador. São utilizados para indicar ao compilador ou interpretador as conversões necessárias para obter os valores em memória durante a construção do programa.

O tipo de dado ajuda também o programador a detectar os eventuais erros envolvidos com semântica das instruções, erros esses detectados na análise semântica dos programas.

Dependendo da linguagem de programação, o tipo de um dado é verificado diferentemente, de acordo com a análise léxica, sintática e semântica do compilador ou interpretador da linguagem. Os tipos têm geralmente associações com valores na memória ou com objetos (para uma linguagem orientada a objeto) ou variáveis.

Tipo Estático e Dinâmico

A verificação do tipo de um dado é feita de forma estática em tempo de compilação ou de forma dinâmica em tempo de execução. Em C, C++, Java e Haskell os tipos são estáticos, em Scheme, Lisp, Smalltalk, Perl, PHP, Visual Basic, Ruby e Python são dinâmicos.

Em C uma definição estática do tipo de uma variável ficaria assim:

```
printf ("O tipo char ocupa %lld bytes\n", sizeof(char));
```

Tipo Forte e Fraco

Linguagens implementadas com tipificação forte (linguagem fortemente tipificada), tais como Java e Pascal, exigem que o tipo de dado de um valor seja do mesmo tipo da variável ao qual este valor será atribuído. Exemplo:

(Sintaxe genérica)

1. Declarar Variáveis
2. TEXTO nome
3. INTEIRO idade
5. Atribuições
6. nome = "Helisson"
7. idade = 13.1

Ocorrerá um erro ao compilar a linha 7, pois o valor "13.1", que é do tipo REAL, precisa ser convertido para o tipo de dado INTEIRO.

Em linguagens com tipos de dados fracos, tais como PHP e VBScript, a conversão não se faz necessária, sendo realizada implicitamente pelo compilador ou interpretador.

Tipo Primitivo e Composto

Um tipo primitivo (também conhecido por nativo ou básico) é fornecido por uma linguagem de programação como um bloco de construção básico. Dependendo da implementação da linguagem, os tipos primitivos podem ou não possuir correspondência direta com objetos na memória.

Um tipo composto pode ser construído em uma linguagem de programação a partir de tipos primitivos e de outros tipos compostos, em um processo chamado composição.

Em C, cadeias de caracteres são tipos compostos, enquanto em dialetos modernos de Basic e em JavaScript esse tipo é nativo da linguagem.

Tipos primitivos típicos incluem caractere, inteiro (representa um subconjunto dos números inteiros, com largura dependente do sistema; pode possuir sinal ou não), ponto flutuante (representa o conjunto dos números racionais), booleano (lógica booleana, verdadeiro ou falso) e algum tipo de referência (como ponteiro ou handles).

Tipos primitivos mais sofisticados incluem tuplas, listas ligadas, números complexos, números racionais e tabela hash, presente sobretudo em linguagens funcionais.

Espera-se que operações envolvendo tipos primitivos sejam as construções mais rápidas da linguagem. Por exemplo, a adição de inteiros pode ser feita com somente uma instrução de máquina, e mesmo algumas CPUs oferecem instruções específicas para processar sequências de caracteres com uma única instrução.

A maioria das linguagens não permite que o comportamento de um tipo nativo seja modificado por programas. Como exceção, Smalltalk permite que tipos nativos sejam estendidos, adicionando-se operações e também redefinindo operações nativas.

Uma estrutura em C e C++ é um tipo composto de um conjunto determinado de campos e membros. O tamanho total da estrutura para o tipo composto corresponde a soma dos requerimentos de cada campo da estrutura, além de um possível espaço para alinhamento de bits. Por exemplo:

```
struct Conta {  
  
    int numero;  
  
    char *nome;  
  
    char *sobrenome;  
  
    float balanco;};
```

Define um tipo composto chamado Conta. A partir de uma variável minhaConta do tipo acima, pode-se acessar o número da conta através de minhaConta.numero.

Algoritmos Para Pesquisa E Ordenação

Em ciência da computação, um algoritmo de busca, em termos gerais é um algoritmo que toma um problema como entrada e retorna a solução para o problema, geralmente após resolver um número possível de soluções.

Uma solução, no aspecto de função intermediária, é um método o qual um algoritmo externo, ou mais abrangente, utilizará para solucionar um determinado problema.

Esta solução é representada por elementos de um espaço de busca, definido por uma fórmula matemática ou um procedimento, tal como as raízes de uma equação com números inteiros variáveis, ou uma combinação dos dois, como os circuitos hamiltonianos de um grafo.

Já pelo aspecto de uma estrutura de dados, sendo o modelo de explanação inicial do assunto, a busca é um algoritmo projetado para encontrar um item com propriedades especificadas em uma coleção de itens. Os itens podem ser armazenadas individualmente, como registros em um banco de dados.

A maioria dos algoritmos estudados por cientistas da computação que resolvem problemas são algoritmos de busca.

Os algoritmos de busca têm como base o método de procura de qualquer elemento dentro de um conjunto de elementos com determinadas propriedades. Que podiam ser livros nas bibliotecas, ou dados cifrados, usados principalmente durante as duas grandes guerras.

Seus formatos em linguagem computacional vieram a se desenvolver juntamente com a construção dos primeiros computadores.

Sendo que a maioria de suas publicações conhecidas começa a surgir a partir da década de 1970. Atualmente os algoritmos de busca são a base de motores de buscas da Internet

Classes de Algoritmos de Busca

Algoritmos para a busca de espaços virtuais são usados em problema de satisfação de restrição, onde o objetivo é encontrar um conjunto de atribuições de valores para certas variáveis que irão satisfazer específicas equações e inequações matemáticas.

Eles também são utilizados quando o objetivo é encontrar uma atribuição de variável que irá maximizar ou minimizar uma determinada função dessas variáveis. Algoritmos para estes problemas incluem a base de busca por força bruta (também chamado de "ingênua" ou busca "desinformada"), e uma variedade de heurísticas que tentam explorar o conhecimento parcial sobre a estrutura do espaço, como relaxamento linear, geração de restrição, e propagação de restrições.

Algumas subclasses importantes são os métodos de busca local, que vêem os elementos do espaço de busca como os vértices de um grafo, com arestas definidas por um conjunto de heurísticas aplicáveis ao caso, e fazem a varredura do espaço, movendo-se de item para item ao longo das bordas, por exemplo de acordo com o declive máximo ou com o critério da melhor escolha, ou em uma busca estocástica. Esta categoria inclui uma grande variedade de métodos metaheurísticos gerais, como arrefecimento simulado, pesquisa tabu, times assíncronos, e programação genética, que combinam heurísticas arbitrárias de maneiras específicas.

Esta classe também inclui vários algoritmos de árvore de busca, que vêem os elementos como vértices de uma Árvore (teoria dos grafos) árvore, e atravessam-na em alguma ordem especial. Exemplos disso incluem os métodos exaustivos, como em busca em profundidade e em busca em largura, bem como vários métodos de busca por poda de árvore baseados em heurística como retrocesso e ramo e encadernado. Ao contrário das metaheurísticas gerais, que trabalham melhor apenas no sentido probabilístico, muitos destes métodos de árvore de busca têm a garantia de encontrar a solução exata ou ideal, se for dado tempo suficiente.

Outra importante sub-classe consiste de algoritmos para explorar a árvore de jogo de jogos para múltiplos participantes (multiplayer), como xadrez ou gamão, cujos nós consistem em todas as situações de jogo possíveis que poderiam resultar da situação atual. O objetivo desses problemas é encontrar o movimento que oferece a melhor chance de uma vitória, tendo em conta todos os movimentos possíveis do(s) adversário(s). Problemas similares ocorrem quando as pessoas, ou máquinas, têm de tomar decisões sucessivas cujos resultados não estão totalmente sob seu controle, como em um robô, ou na orientação de marketing, financeira ou de planejamento estratégico militar. Este tipo de problema - pesquisa combinatória - tem sido extensivamente estudado no contexto da inteligência artificial. Exemplos de algoritmos para esta classe são o algoritmo minimax, poda alfa-beta e o algoritmo A*.

Para Subestruturas de Uma Dada Estrutura

O nome de pesquisa combinatória é geralmente usado para os algoritmos que procuram uma subestrutura específica de uma dada estrutura discreta, tais como um grafo, uma cadeia de caracteres, um grupo (matemática) finito, e assim por diante. O termo otimização combinatória é normalmente utilizado quando o objetivo é encontrar uma subestrutura com um valor máximo (ou mínimo) de algum parâmetro. (Uma vez que a subestrutura normalmente é representada no computador por um conjunto de variáveis de inteiros com restrições, estes problemas podem ser vistos como casos especiais de satisfação restrita ou otimização discreta, mas eles geralmente são formulados e resolvidos em um ambiente mais abstrato onde a representação interna não é explicitamente mencionada.)

Uma subclasse importante e extensivamente estudada são os algoritmos de grafos, em particular algoritmos de travessia de grafo, para encontrar determinadas subestruturas em um dado grafo - como subgrafos, caminhos, circuitos, e assim por diante. Exemplos incluem o algoritmo de Dijkstra, algoritmo de Kruskal, o algoritmo do vizinho mais próximo, e algoritmo de Prim.

Outra subclasse importante desta categoria são os algoritmos de busca de cadeia de caracteres, que busca de padrões dentro de expressões. Dois exemplos famosos são os algoritmos Boyer-Moore e Knuth-Morris-Pratt, e vários algoritmos baseados na estrutura de dados árvore de sufixo.

Busca Pelo Máximo De Uma Função

Em 1953, Kiefer concebeu a busca de Fibonacci, que pode ser utilizada para encontrar o máximo de uma função unimodal e tem muitas outras aplicações em ciências da computação.

Para Computadores Quânticos

Há também métodos de busca projetados para computadores quânticos, como o algoritmo de Grover, que são teoricamente mais rápidos do que a busca linear ou força bruta mesmo sem a ajuda de estruturas de dados ou heurísticas.

Algoritmo De Ordenação

Algoritmo de ordenação em ciência da computação é um algoritmo que coloca os elementos de uma dada sequência em uma certa ordem -- em outras palavras, efetua sua ordenação completa ou parcial. As ordens mais usadas são a numérica e a lexicográfica.

Existem várias razões para se ordenar uma sequência. Uma delas é a possibilidade de acessar seus dados de modo mais eficiente.

Métodos De Ordenação De Vetores

Métodos Simples

Insertion sort

Selection sort

Bubble sort

Comb sort

Métodos Sofisticados

Merge sort

Heapsort

Shell sort

Radix sort

Gnome sort

Counting sort

Bucket sort

Cocktail sort

Timsort

Quick sort

Métodos De Pesquisa

Pesquisa binária

Busca linear

BogoBusca

Listas Lineares e Suas Generalizações

Lista linear é uma estrutura de dados na qual elementos de um mesmo tipo de dado estão organizados de maneira sequencial. Não necessariamente, estes elementos estão fisicamente em sequência, mas a idéia é que exista uma ordem lógica entre eles.

Um exemplo disto seria um consultório médico: as pessoas na sala de espera estão sentadas em qualquer lugar, porém sabe-se quem é o próximo a ser atendido, e o seguinte, e assim por diante. Assim, é importante ressaltar que uma lista linear permite representar um conjunto de dados afins (de um mesmo tipo) de forma a preservar a relação de ordem entre seus elementos. Cada elemento da lista é chamado de nó, ou nodo.

Definição:

Conjunto de N nós, onde $N \geq 0$, x_1, x_2, \dots, x_n , organizados de forma a refletir a posição relativa dos mesmos. Se $N \geq 0$, então x_1 é o primeiro nó. Para $1 < k < n$, o nó x_k é precedido pelo nó x_{k-1} e seguido pelo nó x_{k+1} e x_n é o último nó. Quando $N = 0$, diz-se que a lista está vazia. Exemplos de listas lineares:

Pessoas na fila de um banco;

Letras em uma palavra;

Relação de notas dos alunos de uma turma;

Itens em estoque em uma empresa;

Dias da semana;

Vagões de um trem;

Pilha de pratos;

Cartas de baralho.

Alocação De Uma Lista

Quanto a forma de alocar memória para armazenamento de seus elementos, uma lista pode ser:

Sequencial ou Contígua

Numa lista linear contígua, os nós além de estarem em uma sequência lógica, estão também fisicamente em sequência. A maneira mais simples de acomodar uma lista linear em um computador é através da utilização de um vetor.



A representação por vetor explora a sequencialidade da memória de tal forma que os nós de uma lista sejam armazenados em endereços contíguos.

Encadeada

Os elementos não estão necessariamente armazenados sequencialmente na memória, porém a ordem lógica entre os elementos que compõem a lista deve ser mantida. Listas lineares encadeadas são discutida na aula 11.

Operações Com Listas

As operações comumente realizadas com listas são:

Criação de uma lista

Remoção de uma lista

Inserção de um elemento da lista

Remoção de um elemento da lista

Acesso de um elemento da lista

Alteração de um elemento da lista

Combinação de duas ou mais listas

Classificação da lista

Cópia da lista

Localizar nodo através de info

Tipos de Listas Lineares

Os tipos mais comuns de listas lineares são as:

Pilhas

Uma pilha é uma lista linear do tipo LIFO - Last In First Out, o último elemento que entrou, é o primeiro a sair. Ela possui apenas uma entrada, chamada de topo, a partir da qual os dados entram e saem dela. Exemplos de pilhas são: pilha de pratos, pilha de livros, pilha de alocação de variáveis da memória, etc.

Filas

Uma fila é uma lista linear do tipo FIFO - First In First Out, o primeiro elemento a entrar será o primeiro a sair. Na fila os elementos entram por um lado ("por trás") e saem por outro ("pela frente"). Exemplos de filas são: a fila de caixa de banco, a fila do INSS, etc.

Deque

Um deque - Double-Ended QUEUE) é uma lista linear na qual os elementos entram e saem tanto pela "pela frente" quanto "por trás". Pode ser considerada uma generalização da fila.

Em ciência da computação, uma lista ou sequência é uma estrutura de dados abstrata que implementa uma coleção ordenada de valores, onde o mesmo valor pode ocorrer mais de uma vez. Uma instância de uma lista é uma representação computacional do conceito matemático de uma sequência finita, que é, uma tupla. Cada instância de um valor na lista normalmente é chamado de um item, entrada ou elemento da lista. Se o mesmo valor ocorrer várias vezes, cada ocorrência é considerada um item distinto.



Uma estrutura de lista encadeada isoladamente, implementando uma lista com 3 elementos inteiros.

O nome lista também é usado para várias estruturas de dados concretas que podem ser usadas para implementar listas abstratas, especialmente listas encadeadas.

As chamadas estruturas de lista estática' permitem apenas a verificação e enumeração dos valores. Uma lista mutável ou dinâmica pode permitir que itens sejam inseridos, substituídos ou excluídos durante a existência da lista.

Muitas linguagens de programação fornecem suporte para tipos de dados lista e possuem sintaxe e semântica especial para listas e operações com listas. Uma lista pode frequentemente ser construída escrevendo-se itens em sequência, separados por vírgulas, ponto e vírgulas ou espaços, dentro de um par de delimitadores como parênteses '()', colchetes '[]', chaves '{}' ou chevrons '<>'. Algumas linguagens podem permitir que tipos lista sejam indexados ou cortados como os tipos vetor.

Em linguagens de programação orientada a objetos, listas normalmente são fornecidas como instâncias ou subclasses de uma classe "lista" genérica. Tipos de dado lista são frequentemente implementados usando arrays ou listas encadeadas de algum tipo, mas outras estruturas de dados podem ser mais apropriadas para algumas aplicações. Em alguns contextos, como em programação Lisp, o termo lista pode se referir especificamente à lista encadeada em vez de um array.

É forma de organização através da enumeração de dados para melhor visualização da informação. Em informática, o conceito expande-se para uma estrutura de dados dinâmica, em oposição aos vetores, que são estruturas de dados estáticas. Assim, uma lista terá virtualmente infinitos elementos.

Numa lista encadeada existem dois campos. Um campo reservado para colocar o dado a ser armazenado e outro campo para apontar para o próximo elemento da lista. Normalmente a implementação é feita com ponteiros.

Existem vários tipos de implementação de listas como estruturas de dados:

Listas duplamente ligadas

Listas FIFO, ou filas (First In First Out - primeiro a entrar, primeiro a sair).

Listas LIFO, ou pilhas (Last In First Out - último a entrar, primeiro a sair).

Características

Listas possuem as seguintes características:

Tamanho da lista significa o número de elementos presentes na lista. Listas encadeadas tem a vantagem de ter um tamanho variável, novos itens podem ser adicionados, o que aumentando seu tamanho.

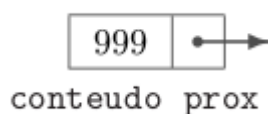
Cada elemento numa lista possui um índice, um número que identifica cada elemento da lista. Usando o índice de um elemento da lista é possível buscá-lo ou removê-lo.

Uma lista encadeada é uma representação de uma sequência de objetos, todos do mesmo tipo, na memória RAM (= random access memory) do computador. Cada elemento da sequência é armazenado em uma célula da lista: o primeiro elemento na primeira célula, o segundo na segunda e assim por diante.

Estrutura de Uma Lista Encadeada

Uma lista encadeada (= linked list = lista ligada) é uma sequência de células; cada célula contém um objeto de algum tipo e o endereço da célula seguinte. Suporemos neste capítulo que os objetos armazenados nas células são do tipo int. Cada célula é um registro que pode ser definido assim:

```
struct reg {  
    int conteudo;  
    struct reg *prox;};
```



É conveniente tratar as células como um novo tipo-de-dados e atribuir um nome a esse novo tipo:

```
typedef struct reg celula; // célula
```

Uma célula c e um ponteiro p para uma célula podem ser declarados assim:

```
celula c;  
celula *p;
```

Se c é uma célula então c.conteudo é o conteúdo da célula e c.prox é o endereço da próxima célula. Se p é o endereço de uma célula, então p->conteudo é o conteúdo da célula e p->prox é o endereço da próxima célula. Se p é o endereço da última célula da lista então p->prox vale NULL .



(A figura pode dar a falsa impressão de que as células da lista ocupam posições consecutivas na memória. Na realidade, as células estão tipicamente espalhadas pela memória de maneira imprevisível.)

Árvores E Suas Generalizações

Árvore binária é uma estrutura de dados caracterizada por:

Ou não tem elemento algum (árvore vazia).

Ou tem um elemento distinto, denominado raiz, com dois apontamentos para duas estruturas diferentes, denominadas sub-árvore esquerda e sub-árvore direita.

Perceba que a definição é recursiva e, devido a isso, muitas operações sobre árvores binárias utilizam recursão. É o tipo de árvore mais utilizado na computação. A principal utilização de árvores binárias são as árvores de busca.

Implementação de um Nó de uma árvore binária em C++

```
class No
```

```
private:
```

```
/* Aqui vão as informações do nó, no nosso caso usaremos apenas um inteiro como chave,
```

```
mas poderíamos usar qualquer outra informação junto com as chaves como strings, booleanos etc... */
```

```
int chave;
```

```
// Ponteiros do tipo Nó para as sub-arvores direitas e esquerdas respectivamente
```

```
No *esq;
```

```
No *dir;
```

```
public:
```

```
/* Aqui criaremos um construtor para o nó que seta o valor da chave e inicializa os ponteiros
```

```
das sub-arvores como null. OBS: Note que usarei nullptr em vez de NULL, se não utilizar o c++11 essa keyword
```

```
não funcionará. */
```

```
No ( int chave )
```

```
{ this->chave = chave; // seta a chave
```

```
esq = nullptr; // inicializa a sub-arvore esquerda como null.
```

```
dir = nullptr; // inicializa a sub-arvore direita como null.}
```

```
// METODOS GETS E SETTERS.
```

```
int getChave()
```

```
{ return chave; }
```

```
No* getEsq()
```

```
{ return esq; }
```

```
No* getDir()
{ return dir; }

void setEsq( No* esq )
{ this->esq = esq; }

void setDir( No* dir )
{ this->dir = dir; }
```

Definições Para Árvores Binárias

Os nós de uma árvore binária possuem graus zero, um ou dois. Um nó de grau zero é denominado folha.

Uma árvore binária é considerada estritamente binária se cada nó da árvore possui grau zero ou dois.

A profundidade de um nó é a distância deste nó até a raiz. Um conjunto de nós com a mesma profundidade é denominado nível da árvore. A maior profundidade de um nó, é a altura da árvore.

Uma árvore é dita completa se todas as folhas da árvore estão no mesmo nível da árvore.

Definições Em Teoria Dos Grafos

Em teoria dos grafos, uma árvore binária é definida como um grafo acíclico, conexo, dirigido e que cada nó não tem grau maior que 3. Assim sendo, só existe um caminho entre dois nós distintos.

E cada ramo da árvore é um vértice dirigido, sem peso, que parte do pai e vai o filho.

Árvore Binária De Busca

Em Ciência da computação, uma árvore binária de busca (ou árvore binária de pesquisa) é uma estrutura de dados de árvore binária baseada em nós, onde todos os nós da subárvore esquerda possuem um valor numérico inferior ao nó raiz e todos os nós da subárvore direita possuem um valor superior ao nó raiz (esta é a forma padrão, podendo as subárvores serem invertidas, dependendo da aplicação).

O objetivo desta árvore é estruturar os dados de forma a permitir busca binária.

Seja $S = \{s_1, s_2, \dots, s_n\}$ um conjunto de chaves tais que $s_1 < s_2 \dots s_n$. Seja k um valor dados. Deseja-se verificar se $k \in S$ e identificar o índice i tal que $k = s_i$.

A Árvore Binária de Busca (ABB) resolve os problemas propostos. A figura ilustra uma ABB.

Uma ABB é uma árvore binária rotulada T com as seguintes propriedades:

T possui n nós. Cada nó u armazena uma chave distinta $s_j \in S$ e tem como rótulo o valor $r(u) = s_j$.

Para cada nó v de T $r(v_1) < r(v)$ e $r(v_2) > r(v)$, onde v_1 pertence à subárvore esquerda de v e v_2 pertence à subárvore direita de v .

Dado o conjunto S com mais de um elemento, existem várias ABB que resolvem o problema.

Elementos

Nós - são todos os itens guardados na árvore

Raiz - é o nó do topo da árvore (no caso da figura acima, a raiz é o nó 8)

Filhos - são os nós que vem depois dos outros nós (no caso da figura acima, o nó 6 é filho do 3)

Pais - são os nós que vem antes dos outros nós (no caso da figura acima, o nó 10 é pai do 14)

Folhas - são os nós que não têm filhos; são os últimos nós da árvore (no caso da figura acima, as folhas são 1, 4, 7 e 13)

Complexidade

A complexidade das operações sobre ABB depende diretamente da altura da árvore.

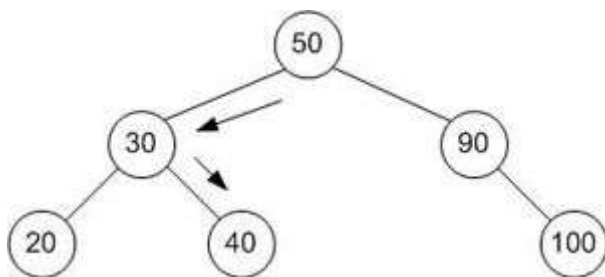
Uma árvore binária de busca com chaves aleatórias uniformemente distribuídas tem altura $O(\log n)$.

No pior caso, uma ABB poderá ter altura $O(n)$. Neste caso a árvore é chamada de árvore zig-zag e corresponde a uma degeneração da árvore em lista encadeada.

Em função da observação anterior, a árvore binária de busca é de pouca utilidade para ser aplicada em problemas de busca em geral. Daí o interesse em árvores balanceadas, cuja altura seja $O(\log n)$ no pior caso

Operações

Busca



Buscando um valor na árvore binária.

A busca em uma árvore binária por um valor específico pode ser um processo recursivo ou iterativo. Será apresentado um método recursivo.

A busca começa examinando o nó raiz. Se a árvore está vazia, o valor procurado não pode existir na árvore. Caso contrário, se o valor é igual a raiz, a busca foi bem-sucedida. Se o valor é menor do que a raiz, a busca segue pela subárvore esquerda. Similarmente, se o valor é maior do que a raiz, a busca segue pela subárvore direita. Esse processo é repetido até o valor ser encontrado ou a subárvore ser nula (vazia). Se o valor não for encontrado até a busca chegar na subárvore nula, então o valor não deve estar presente na árvore.

Segue abaixo o algoritmo de busca implementado na linguagem Python:

'no' refere-se ao nó-pai, neste caso

```
def arvore_binaria_buscar(no, valor):
```

```
    if no is None:
```

```
        # valor não encontrado
```

```
        return None
```

```
    else:
```

```
        if valor == no.valor:
```

```
            # valor encontrado
```

```
            return no.valor
```

```
elif valor < no.valor:  
  
    # busca na subárvore esquerda  
  
    return arvore_binaria_buscar(no.filho_esquerdo, valor)  
  
elif valor > no.valor:  
  
    # busca na subárvore direita  
  
    return arvore_binaria_buscar(no.filho_direito, valor)
```

Essa operação poderá ser $O(\log n)$ em algumas situações, mas necessita $O(n)$ de tempo no pior caso, quando a árvore assumir a forma de lista ligada (árvore zig-zag).

Inserção

A inserção começa com uma busca, procurando pelo valor, mas se não for encontrado, procuram-se as subárvores da esquerda ou direita, como na busca. Eventualmente, alcança-se a folha, inserindo-se então o valor nesta posição. Ou seja, a raiz é examinada e introduz-se um nó novo na subárvore da esquerda se o valor novo for menor do que a raiz, ou na subárvore da direita se o valor novo for maior do que a raiz. Abaixo, um algoritmo de inserção em Python:

```
def arvore_binaria_inserir(no, chave, valor):  
  
    if no is None:  
  
        return TreeNode(None, chave, valor, None)  
  
    if chave == no.chave:  
  
        return TreeNode(no.filho_esquerdo, chave, valor, no.filho_direito)  
  
    if chave < no.chave:  
  
        return TreeNode(arvore_binaria_inserir(no.filho_esquerdo, chave, valor), no.chave, no.valor, no.filho_direito)  
  
    else:  
  
        return TreeNode(no.filho_esquerdo, no.chave, no.valor, arvore_binaria_inserir(no.filho_direito, chave, valor))
```

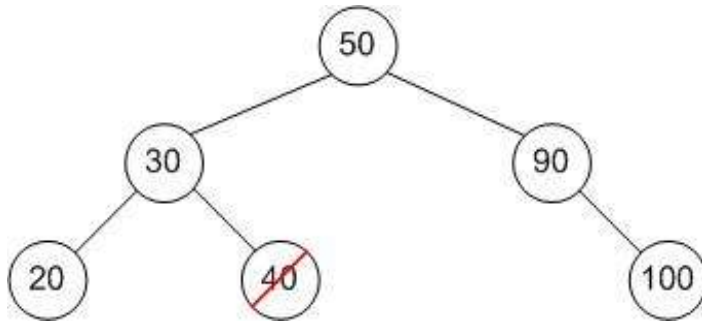
Esta operação requer $O(\log n)$ vezes para o caso médio e necessita de $O(n)$ no pior caso. A fim de introduzir um nó novo na árvore, seu valor é primeiro comparado com o valor da raiz. Se seu valor for menor que a raiz, é comparado então com o valor do filho da esquerda da raiz. Se seu valor for maior, está comparado com o filho da direita da raiz. Este processo continua até que o nó novo esteja comparado com um nó da folha, e então adiciona-se o filho da direita ou esquerda, dependendo de seu valor.

Remoção

A exclusão de um nó é um processo mais complexo. Para excluir um nó de uma árvore binária de busca, há de se considerar três casos distintos para a exclusão:

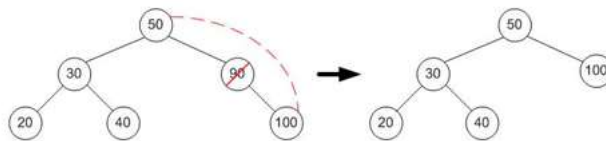
Remoção Na Folha

A exclusão na folha é a mais simples, basta removê-lo da árvore.



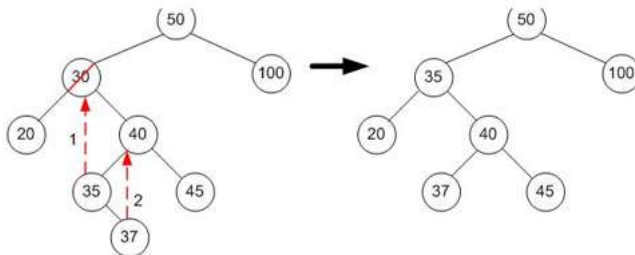
Remoção De Nó Com Um Filho

Excluindo-o, o filho sobe para a posição do pai.



Remoção de Nó com Dois Filhos

Neste caso, pode-se operar de duas maneiras diferentes. Pode-se substituir o valor do nó a ser retirado pelo valor sucessor (o nó mais à esquerda da subárvore direita) ou pelo valor antecessor (o nó mais à direita da subárvore esquerda), removendo-se aí o nó sucessor (ou antecessor).



No exemplo acima, o nó de valor 30 está para ser removido, e possui como sucessor imediato o valor 35 (nó mais à esquerda da sua sub-árvore direita). Assim sendo, na exclusão, o valor 35 será promovido no lugar do nó a ser excluído, enquanto a sua sub-árvore (direita) será promovida para sub-árvore esquerda do 40, como pode ser visto na figura.

Exemplo De Algoritmo De Exclusão Em Python:

```
def exclusao_em_arvore_binaria(nó_arvore, valor):
    if nó_arvore is None: return None # Valor não encontrado

    esquerda, nó_valor, direita = nó_arvore.esquerda, nó_arvore.valor, nó_arvore.direita

    if nó_valor == valor:
        if esquerda is None:
            return direita
        elif direita is None:
            return esquerda
        else:
```

```
valor_max, novo_esquerda = busca_max(esquerda)
return TreeNode(novo_esquerda, valor_max, direita)

elif valor < nó_valor:
return TreeNode(exclusao_em_arvore_binaria(esquerda, valor), nó_valor, direita)

else:
return TreeNode(esquerda, nó_valor, exclusao_em_arvore_binaria(direita, valor))

def busca_max(nó_arvore):
    esquerda, nó_valor, direita = nó_arvore.esquerda, nó_arvore.valor, nó_arvore.direita
    if direita is None: return (nó_valor, esquerda)
    else:
        (valor_max, novo_direita) = busca_max(direita)
        return (valor_max, (esquerda, nó_valor, novo_direita))
```

Embora esta operação não percorra sempre a árvore até uma folha, esta é sempre uma possibilidade; assim, no pior caso, requer o tempo proporcional à altura da árvore, visitando-se cada nó somente uma única vez.

Aplicações

Percursos Em ABB

Em uma árvore binária de busca podem-se fazer os três percursos que se fazem para qualquer árvore binária (percursos em inordem, pré-ordem e pós-ordem). É interessante notar que, quando se faz um percurso em ordem em uma árvore binária de busca, os valores dos nós aparecem em ordem crescente. A operação "Percorre" tem como objetivo percorrer a árvore numa dada ordem, enumerando os seus nós. Quando um nó é enumerado, diz-se que ele foi "visitado".

Pré-ordem (ou profundidade):

Visita a raiz

Percorre a subárvore esquerda em pré-ordem

Percorre a subárvore direita em pré-ordem

Ordem Simétrica:

Percorre a subárvore esquerda em ordem simétrica

Visita a raiz

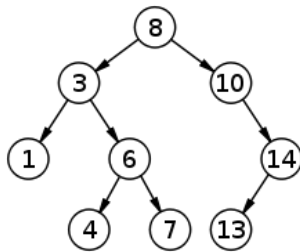
Percorre a subárvore direita em ordem simétrica

Pós-ordem:

Percorre a subárvore esquerda em pós-ordem

Percorre a subárvore direita em pós-ordem

Visita a raiz



Árvore Binária De Busca

Usando a ABB acima, os resultados serão os seguintes:

Pré-ordem => 8, 3, 1, 6, 4, 7, 10, 14, 13

Ordem simétrica => 1, 3, 4, 6, 7, 8, 10, 13, 14 (chaves ordenadas)

Pós-ordem => 1, 4, 7, 6, 3, 13, 14, 10, 8

Ordenação

Uma árvore binária de busca pode ser usada para ordenação de chaves. Para fazer isto, basta inserir todos os valores desejados na ABB e executar o percurso em ordem simétrica.

```
def criar_arvore_binaria(valor):
```

```
    arvore = None
```

```
    for v in valor:
```

```
        arvore = arvore_binaria_de_insercao(arvore, v)
```

```
    return arvore
```

```
def arvore_binaria_transversal(nó_arvore):
```

```
    if nó_arvore is None: return []
```

```
    else:
```

```
        esquerda, valor, direita = nó_arvore
```

```
        return (arvore_binaria_transversal(esquerda) + [valor] + arvore_binaria_transversal(direita))
```

Criar ABB tem complexidade $O(n^2)$ no pior caso. A geração de um vetor de chaves ordenadas tem complexidade $O(n)$. O algoritmo de ordenação terá complexidade final $O(n^2)$ no pior caso.

Cabe observar que há algoritmos de ordenação dedicados com complexidade $O(n \log n)$, com desempenho superior ao proposto neste tópico.

Árvore AVL

Árvore AVL	
Tipo	Árvore
Ano	1962
Inventado por	Georgy Adelson-Velsky e Yevgeniy Landis
Complexidade de Tempo em Notação big O	

Algoritmo	Caso Médio	Pior Caso
Espaço	$O(n)$	$O(n)$
Busca	$O(\log n)[1]$	$O(\log n)[1]$
Inserção	$O(\log n)[1]$	$O(\log n)[1]$
Remoção	$O(\log n)[1]$	$O(\log n)[1]$

Árvore AVL é uma árvore binária de busca balanceada, ou seja, uma árvore balanceada (árvore completa) são as árvores que minimizam o número de comparações efetuadas no pior caso para uma busca com chaves de probabilidades de ocorrências idênticas. Contudo, para garantir essa propriedade em aplicações dinâmicas, é preciso reconstruir a árvore para seu estado ideal a cada operação sobre seus nós (inclusão ou exclusão), para ser alcançado um custo de algoritmo com o tempo de pesquisa tendendo a $O(\log N)$.

As operações de busca, inserção e remoção de elementos possuem complexidade $O(\log n)$ (no qual n é o número de elementos da árvore), que são aplicados a árvore de busca binária.

O nome AVL vem de seus criadores soviéticos Adelson Velsky e Landis, e sua primeira referência encontra-se no documento "Algoritmos para organização da informação" de 1962.

Nessa estrutura de dados cada elemento é chamado de nó. Cada nó armazena uma chave e dois ponteiros, uma para a subárvore esquerda e outro para a subárvore direita.

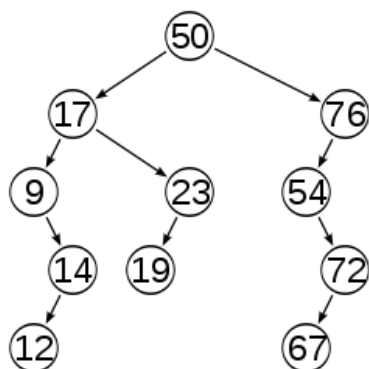
No presente artigo serão apresentados: os conceitos básicos, incluindo uma proposta de estrutura; apresentação das operações busca, inserção e remoção, todas com complexidade $O(\log n)$.

História

Esta estrutura foi criada em 1962 pelos soviéticos Adelson Velsky e Landis que a criaram para que fosse possível inserir e buscar um elemento em tempo $c \cdot \log(n)$ operações, onde n é o número de elementos contido na árvore. Tal estrutura foi a primeira árvore binária balanceada criada.

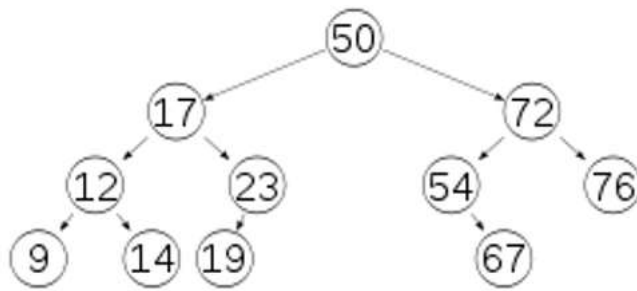
Conceitos Básicos

Definição



Uma Árvore Não AVL

Uma árvore binária T é denominada AVL quando, para qualquer nó de T , as alturas de suas duas subárvores, esquerda e direita, diferem em módulo de até uma unidade.



Uma árvore AVL

Pela definição fica estabelecido que todos os nós de uma árvore AVL devem respeitar a seguinte propriedade:

$|hd(u) - he(u)| \leq 1$, onde $hd(u)$ é a altura da subárvore direita do nó u e $he(u)$ é a altura da subárvore esquerda do nó u .

O valor $hd(u) - he(u)$ é denominado fator de balanço do nó. Quando um nó possui fator de balanço com valor -1, 0 ou 1 então o mesmo é um nó regulado. Todos os nós de uma árvore AVL são regulados, caso contrário a árvore não é AVL.

Estrutura

Proposta de estrutura dos nós de uma árvore AVL básica, com chave do tipo inteiro:

tipo No_AVL = registro

chave: inteiro;

fb: inteiro; // Fator de Balanço

esq: ^No_AVL; // aponta subárvore esquerda

dir: ^No_AVL; // aponta subárvore direita

fim;

O campo chave armazena o valor da chave. Os campos esq e dir são ponteiros para as subárvores esquerda e direita, respectivamente. O campo fb armazena o fator de balanço.

Definição da estrutura da árvore:

tipo Arvore_AVL = registro

raiz: ^No_AVL;

// definição de outros campos de interesse

fim;

Balanceamento

Toda árvore AVL é balanceada, isto é, sua altura é $O(\log n)$.

A vantagem do balanceamento é possibilitar que a busca seja de complexidade $O(\log n)$. Entretanto, as operações de inserção e remoção devem possuir custo similar. No caso da árvore AVL, a inserção e remoção têm custo $O(\log n)$.

Por definição, todos os nós da AVL devem ter $fb = -1, 0$ ou 1 .

Para garantir essa propriedade, a cada inserção ou remoção o fator de balanço deve ser atualizado a partir do pai do nó inserido até a raiz da árvore. Na inserção basta encontrar o primeiro nó desregulado ($fb = -2$ ou $fb = 2$), aplicar a operação de rotação necessária, não havendo necessidade de verificar os demais nós. Na remoção a verificação deverá prosseguir até a raiz, podendo requerer mais de uma rotação.

Uma árvore AVL sempre terá um tamanho menor que:

Onde n é o número de elementos da árvore e φ é a proporção áurea.

Complexidade

A árvore AVL tem complexidade $O(\log n)$ para todas operações e ocupa espaço n , onde n é o número de nós da árvore.

	Média	Pior Caso
Espaço	$O(n)$	$O(n)$
Busca	$O(\log n)$	$O(\log n)$
Inserção	$O(\log n)$	$O(\log n)$
Deleção	$O(\log n)$	$O(\log n)$

Complexidade da árvore AVL em notação O .

Operações

Busca

A busca é a mesma utilizada em árvore binária de busca.

A busca pela chave de valor K inicia sempre pelo nó raiz da árvore.

Seja pt_u um ponteiro para o nó u sendo verificado. Caso o pt_u seja nulo então a busca não foi bem sucedida (K não está na árvore ou árvore vazia). Verificar se a chave K igual $pt_u \rightarrow chave$ (valor chave armazenado no nó u), então a busca foi bem sucedida. Caso contrário, se $K < pt_u \rightarrow chave$ então a busca segue pela subárvore esquerda; caso contrário, a busca segue pela subárvore direita.

Algoritmo De Busca

```
busca_AVL(@pt_u:^no_AVL, K:inteiro):logico;
```

inicio

se pt_u é NULO então retornar Falso;

se $K = pt_u \rightarrow chave$ então retornar Verdadeiro;

senão se $K < pt_u \rightarrow chave$ então

retornar $busca_AVL(K, u \rightarrow esq)$;

senão retornar $busca_AVL(K, u \rightarrow dir)$;

fim.

Exemplo De Algoritmo De Busca Em Java.

// O método de procura numa AVL é semelhante ao busca binária de uma árvore binária de busca comum.

```
public BSTNode<T> search(T element) {  
    return search(element, this.root);  
}  
// Método auxiliar à recursão.  
private BSTNode<T> search(T element, BSTNode<T> node) {  
    if (element == null || node.isEmpty()) {  
        return new BSTNode<T>();  
    }  
    if (node.isEmpty() || node.getData().equals(element)) {  
        return node;  
    } else if (node.getData().compareTo(element) > 0) {  
        return search(element, node.getLeft());  
    } else {  
        return search(element, node.getRight());  
    }  
}
```

Inserção

Para inserir um novo nó de valor K em uma árvore AVL é necessária uma busca por K nesta mesma árvore. Após a busca o local correto para a inserção do nó K será em uma subárvore vazia de uma folha da árvore. Depois de inserido o nó, a altura do nó pai e de todos os nós acima deve ser atualizada. Em seguida o algoritmo de rotação simples ou dupla deve ser acionado para o primeiro nó pai desregulado.

Algoritmo Recursivo

```
inserir_AVL(@p: ^no_AVL, K: inteiro, @mudou_h: logico): logico;  
  
var aux: logico; // variável local auxiliar;  
  
inicio  
    se p = NULO então  
        inicio // não achou chave, inserir neste local  
            p := novo(no_AVL); // aloca novo nó dinamicamente, diretamente na subárvore do nó pai  
            p^.chave := K;  
            p^.esq := NULO;  
            p^.dir := NULO;  
            mudou_h := Verdadeiro; // sinalizar que a altura da subárvore mudou 1 unidade  
            retornar Verdadeiro;  
        fim;  
    se K < p^.chave então // inserir recursivamente na subárvore esquerda  
        se inserir_AVL(p^.esq, K, mudou_h) então // ocorreu inserção na subárvore esquerda  
            inicio
```

```
se mudou_h então
inicio // mudou altura da subárvore esquerda de p
p^.fb := p^.fb - 1; // fator de balanço decrementa 1 unidade
caso p^.fb
-2: p := rotacao_direita(p); mudou_h := Falso;
0: mudou_h := Falso; // não mudou a altura da subárvore de raiz p
// -1: mudou_h := Verdadeiro; // desnecessário pois mudou_h já é Verdadeiro
fim
retornar Verdadeiro;
fim
senão
se K > p^.chave então // ocorreu inserção na subárvore direita
se inserir_AVL(p^.dir, K, mudou_h) então // ocorreu inserção
inicio
se mudou_h então
inicio // mudou altura da subárvore esquerda de p
p^.fb := p^.fb + 1; // fator de balanço incrementa 1 unidade
caso p^.fb
2: p := rotacao_esquerda(p); mudou_h := Falso;
0: mudou_h := Falso; // não mudou a altura da subárvore de raiz p
// 1: mudou_h := Verdadeiro; // desnecessário pois mudou_h já é Verdadeiro
fim
retornar Verdadeiro;
fim
retornar Falso;
fim.
```

Os parâmetros p e mudou_h são passados por referência. O ponteiro p aponta para o nó atual. O parâmetro mudou_h é do tipo lógico e informa ao chamador se a subárvore apontada por p mudou sua altura.

Como Identificar Mudança De Altura?

Considerar que o nó p é raiz da subárvore Tp e houve inserção em uma de suas subárvores.

Caso a subárvore Tp tenha mudado de altura, decrementar fb (inserção na subárvore esquerda) ou incrementar fb (inserção na subárvore direita).

Caso 1: Ao inserir um nó folha, a subárvore T_p passa de altura 0 para altura 1, então T_p mudou de altura.

Caso 2: $fb=0$ antes da inserção foi alterado para 1 ou -1, então a subárvore T_p mudou de altura.

Caso 3: $fb=1$ ou -1 antes da inserção, passou a ter valor 0, então a subárvore T_p não mudou de altura.

Caso 4: O fb passou a ter valor -2 ou 2 após a inserção, então há necessidade de aplicação de alguma operação de rotação. Após a rotação, a subárvore T_p terá a mesma altura anterior à inserção.

Exemplo De Algoritmo De Inserção Em Java

/* Por definição, a árvore AVL é uma árvore binária de busca (BST).

* Por este motivo utiliza-se aqui a mesma definição (classe) de Nós que uma BST simples.

```
public void insert(T element) {
    insertAux(element);

    BSTNode<T> node = search(element); // Pode-se utilizar o mesmo search exemplificado acima.
    rebalanceUp(node);
}

private void insertAux(T element) {
    if (element == null) return;
    insert(element, this.root);
}

private void insert(T element, BSTNode<T> node) {
    if (node.isEmpty()) {
        node.setData(element);
        node.setLeft(new BSTNode<T>());
        node.setRight(new BSTNode<T>());
        node.getLeft().setParent(node);
        node.getRight().setParent(node);
    } else {
        if (node.getData().compareTo(element) < 0) {
            insert(element, node.getRight());
        } else if (node.getData().compareTo(element) > 0) {
            insert(element, node.getLeft());
        }
    }
}
```

Algoritmos De Complemento À Inserção E/Ou Algoritmos Para Identificar Desbalanceamento Em Java

```
protected void rebalanceUp(BSTNode<T> node) {
    if (node == null || node.isEmpty()) return;
    rebalance(node);
    if (node.getParent() != null) {
```

```
rebalanceUp(node.getParent());  
protected int calculateBalance(BSTNode<T> node) {  
    if (node == null || node.isEmpty()) return 0;  
    return height(node.getRight()) - height(node.getLeft());  
protected void rebalance(BSTNode<T> node) {  
    int balanceOfNode = calculateBalance(node);  
    if (balanceOfNode < -1) {  
        if (calculateBalance(node.getLeft()) > 0) {  
            leftRotation(node.getLeft());  
            rightRotation(node);  
        } else if (balanceOfNode > 1) {  
            if (calculateBalance(node.getRight()) < 0) {  
                rightRotation(node.getRight());  
                leftRotation(node);  
            }  
        }  
    }  
}
```

Rotação Para Direita E Para Esquerda Em Java

```
protected void leftRotation(BSTNode<T> no) {  
    BSTNode<T> noDireito = no.getRight();  
    no.setRight(noDireito.getLeft());  
    noDireito.getLeft().setParent(no);  
    noDireito.setLeft(no);  
    noDireito.setParent(no.getParent());  
    no.setParent(noDireito);  
    if (no != this.getRoot()) {  
        if (noDireito.getParent().getLeft() == no) {  
            noDireito.getParent().setLeft(noDireito);  
        } else {  
            noDireito.getParent().setRight(noDireito);  
        }  
    }  
    this.root = (BSTNode<T>) noDireito;  
protected void rightRotation(BSTNode<T> no) {  
    BSTNode<T> noEsquerdo = no.getLeft();  
    no.setLeft(noEsquerdo.getRight());  
    noEsquerdo.getRight().setParent(no);  
    noEsquerdo.setRight(no);  
    noEsquerdo.setParent(no.getParent());  
    no.setParent(noEsquerdo);  
    if (no != this.getRoot()) {  
        if (noEsquerdo.getParent().getRight() == no) {  
            noEsquerdo.getParent().setRight(noEsquerdo);  
        } else {  
            noEsquerdo.getParent().setLeft(noEsquerdo);  
        }  
    }  
    this.root = (BSTNode<T>) noEsquerdo;  
}
```

```
noEsquerdo.setRight(no);  
noEsquerdo.setParent(no.getParent());  
no.setParent(noEsquerdo);  
if (no != this.getRoot()) {  
if (noEsquerdo.getParent().getLeft() == no) {  
noEsquerdo.getParent().setLeft(noEsquerdo);  
} else {  
noEsquerdo.getParent().setRight(noEsquerdo);  
} else {  
this.root = (BSTNode<T>) noEsquerdo;
```

Remoção

O primeiro passo para remover uma chave K consiste em realizar uma busca binária a partir do nó raiz. Caso a busca encerre em uma subárvore vazia, então a chave não está na árvore e a remoção não pode ser realizada. Caso a busca encerre em um nó u o nó que contenha a chave então a remoção poderá ser realizada da seguinte forma:

Caso 1: O nó u é uma folha da árvore, apenas exclui-lo.

Caso 2: O nó u tem apenas uma subárvore, necessariamente composta de um nó folha, basta apontar o nó pai de u para a única subárvore e excluir o nó u.

Caso 3: O nó u tem duas subárvores: localizar o nó v predecessor ou sucessor de K, que sempre será um nó folha ou possuirá apenas uma subárvore; copiar a chave de v para o nó u; excluir o nó v a partir da respectiva subárvore de u.

O último passo consiste em verificar a desregulagem de todos nós a partir do pai do nó excluído até o nó raiz da árvore. Aplicar rotação simples ou dupla em cada nó desregulado.

Algoritmo Recursivo

```
remover_AVL(@p:^no_AVL, K:inteiro, @mudou_h:logico):logico;  
var q:^No_AVL; // ponteiro auxiliar para nó  
inicio  
se p = NULO então  
retornar Falso; // não achou a chave K a ser removida  
se K < p^.chave então // remover recursivamente na subárvore esquerda  
se remover_AVL(p^.esq, K, mudou_h) então // ocorreu remoção na subárvore esquerda  
inicio  
se mudou_h então  
inicio // mudou altura da subárvore esquerda de p  
p^.fb := p^.fb + 1; // fator de balanço incrementa 1 unidade  
caso p^.fb
```

```
2: p := rotacao_esquerda(p);
se (p->fb=1) então mudou_h := Falso;
1: mudou_h := Falso; // não mudou a altura da subárvore de raiz p
// 0: mudou_h := Verdadeiro; // desnecessário pois mudou_h já é Verdadeiro
fim
retornar Verdadeiro;
fim
senão
se K > p^.chave então
se remover_AVL(p^.dir, K, mudou_h) então // ocorreu remoção na s.a. direita
inicio
se mudou_h então
inicio // mudou altura da subárvore direita de p
p^.fb := p^.fb - 1; // fator de balanço decrementa 1 unidade
caso p^.fb
-2: p := rotacao_direita(p);
se (p->fb = -1) então mudou_h := Falso;
-1: mudou_h := Falso; // não mudou a altura da subárvore de raiz p
// 0: mudou_h := Verdadeiro; // desnecessário pois mudou_h já é Verdadeiro
fim
retornar Verdadeiro;
fim
senão inicio
se K = p^.chave então // achou a chave K
inicio
se p^.esq=Nulo e p^.dir=Nulo então
inicio // nó folha
delete p;
p := Nulo; // Aterra a subárvore respectiva do nó pai
mudou_h := Verdadeiro;
fim
senão se p^.esq<>Nulo e p^.dir<>Nulo então
inicio // nó tem duas subárvores
```

```
q := Predecessor(p); // retorna nó com chave predecessora
p^.chave := q^.chave;
remover(p^.esq, p^.chave, mudou_h);
fim

senão início // tem apenas uma subárvore
se p^.esq <> Nulo então
início
p^.chave := p^.esq^.chave;
delete p^.esq;
p^.esq := Nulo;
fim
senão início
p^.chave := p^.dir^.chave;
delete(p^.dir);
p^.dir := Nulo;
fim;
mudou_h = Verdadeiro;
fim;
retornar Verdadeiro;
fim
fim
retornar Falso;
fim;

Predecessor(u: ^No_AVL): ^No_AVL // retorna nó contendo chave predecessora
início
u = u^.esq; // aponta para a raiz da subárvore esquerda
enquanto(u^.dir <> Nulo) faça // procura a maior chave da subárvore esquerda
u := u^.dir;
retornar u; // retorna o predecessor
fim;

Exemplo De Algoritmo De Remoção Em Java
public void remover(int valor) {
    removerAVL(this.raiz, valor);
```

```
private void removerAVL(No atual, int valor) {
    if (atual != null) {
        if (atual.getChave() > valor) {
            removerAVL(atual.getEsquerda(), valor);
        } else if (atual.getChave() < valor) {
            removerAVL(atual.getDireita(), valor);
        } else if (atual.getChave() == valor) {
            removerNoEncontrado(atual);
        }
    }
}

private void removerNoEncontrado(No noARemover) {
    No no;

    if (noARemover.getEsquerda() == null || noARemover.getDireita() == null) {
        if (noARemover.getPai() == null) {
            this.raiz = null;
            noARemover = null;
            return;
        }
        no = noARemover;
    } else {
        no = sucessor(noARemover);
        noARemover.setChave(no.getChave());
    }

    No no2;

    if (no.getEsquerda() != null) {
        no2 = no.getEsquerda();
    } else {
        no2 = no.getDireita();
    }

    if (no2 != null) {
        no2.setPai(no.getPai());
    }

    if (no.getPai() == null) {
        this.raiz = no2;
    } else {
        if (no == no.getPai().getEsquerda()) {
            no.getPai().setEsquerda(no2);
        } else {
            no.getPai().setDireita(no2);
        }
    }
}
```

```
        verificarBalanceamento(no.getPai());
```

```
    no = null;
```

Algoritmos Auxiliares Na Remoção

```
public No sucessor(No no) {
```

```
    if (no.getDireita() != null) {
```

```
        No noDireita = no.getDireita();
```

```
        while (noDireita.getEsquerda() != null) {
```

```
            noDireita = noDireita.getEsquerda();
```

```
        return noDireita;
```

```
    } else {
```

```
        No noPai = no.getPai();
```

```
        while (noPai != null && no == noPai.getDireita()) {
```

```
            no = noPai;
```

```
            noPai = noPai.getPai();
```

```
        return noPai;
```

```
public void verificarBalanceamento(No atual) {
```

```
    setBalanceamento(atual);
```

```
    int balanceamento = atual.getBalanceamento();
```

```
    if (balanceamento == -2) {
```

```
        if (altura(atual.getEsquerda().getEsquerda()) >= altura(atual.getEsquerda().ge-  
tDireita())) {
```

```
            atual = rotacaoDireita(atual);
```

```
        } else {
```

```
            atual = duplaRotacaoEsquerdaDireita(atual);
```

```
        }
```

```
    } else if (balanceamento == 2) {
```

```
        if (altura(atual.getDireita().getDireita()) >= altura(atual.getDireita().getEs-  
querda())) {
```

```
            atual = rotacaoEsquerda(atual);
```

```
        } else {
```

```
            atual = duplaRotacaoDireitaEsquerda(atual);
```

```
        }
```

```
    }
```



```
        if (atual.getPai() != null) {
            verificarBalanceamento(atual.getPai());
        } else {
            this.raiz = atual;
        }
    }

    public No rotacaoEsquerda(No inicial) {
        No direita = inicial.getDireita();
        direita.setPai(inicial.getPai());
        inicial.setDireita(direita.getEsquerda());
        if (inicial.getDireita() != null) {
            inicial.getDireita().setPai(inicial);
        }
        direita.setEsquerda(inicial);
        inicial.setPai(direita);
        if (direita.getPai() != null) {
            if (direita.getPai().getDireita() == inicial) {
                direita.getPai().setDireita(direita);
            }
            else if (direita.getPai().getEsquerda() == inicial) {
                direita.getPai().setEsquerda(direita);
            }
        }
    }

    setBalanceamento(inicial);
    setBalanceamento(direita);
    return direita;
}

    public No rotacaoDireita(No inicial) {
        No esquerda = inicial.getEsquerda();
        esquerda.setPai(inicial.getPai());
        inicial.setEsquerda(esquerda.getDireita());
        if (inicial.getEsquerda() != null) {
```

```
        inicial.getEsquerda().setPai(inicial);
    }
    esquerda.setDireita(inicial);
    inicial.setPai(esquerda);
    if (esquerda.getPai() != null) {
        if (esquerda.getPai().getDireita() == inicial) {
            esquerda.getPai().setDireita(esquerda);

        } else if (esquerda.getPai().getEsquerda() == inicial) {
            esquerda.getPai().setEsquerda(esquerda);
        }
    }
    setBalanceamento(inicial);
    setBalanceamento(esquerda);
    return esquerda;
}

public No duplaRotacaoEsquerdaDireita(No inicial) {
    inicial.setEsquerda(rotacaoEsquerda(inicial.getEsquerda()));
    return rotacaoDireita(inicial);
}

public No duplaRotacaoDireitaEsquerda(No inicial) {
    inicial.setDireita(rotacaoDireita(inicial.getDireita()));
    return rotacaoEsquerda(inicial);
}

private void setBalanceamento(No no) {
    no.setBalanceamento(altura(no.getDireita()) - altura(no.getEsquerda()));
}

private int altura(No atual) {
    if (atual == null) {
        return -1;
    }

    if (atual.getEsquerda() == null && atual.getDireita() == null) {
```

```
        return 0;

    } else if (atual.getEsquerda() == null) {
        return 1 + altura(atual.getDireita());

    } else if (atual.getDireita() == null) {
        return 1 + altura(atual.getEsquerda());

    } else {
        return 1 + Math.max(altura(atual.getEsquerda()), altura(atual.getDireita()));
    }
}
```

Como Identificar Mudança De Altura Na Remoção ?

Considerar que o nó p é raiz da subárvore Tp e houve remoção em uma de suas subárvores.

Caso a subárvore Tp tenha mudado de altura, incrementar fb (remoção na subárvore esquerda) ou decrementar fb (remoção na subárvore direita).

Caso 1: Ao remover um nó folha, a subárvore Tp passa de altura 1 para altura 0, então Tp mudou de altura.

Caso 2: fb=0 antes da remoção foi alterado para 1 (remoção à esquerda) ou -1 (remoção à direita), então a subárvore Tp não mudou de altura.

Caso 3: fb=1 ou -1 antes da remoção à direita e à esquerda, respectivamente, passando a ter valor 0, então a subárvore Tp mudou de altura.

Caso 4: fb passou a ter valor -2 ou 2 após a remoção, então houve necessidade de aplicação de rotação. Após a rotação dupla Tp muda de altura, exceto quando u=0 (antes da remoção- caso não relatado na literatura).

Rotação

A operação básica em uma árvore AVL geralmente envolve os mesmos algoritmos de uma árvore de busca binária desbalanceada. A rotação na árvore AVL ocorre devido ao seu desbalanceamento, uma rotação simples ocorre quando um nó está desbalanceado e seu filho estiver no mesmo sentido da inclinação, formando uma linha reta. Uma rotação-dupla ocorre quando um nó estiver desbalanceado e seu filho estiver inclinado no sentido inverso ao pai, formando um "joelho".

Para garantirmos as propriedades da árvore AVL rotações devem ser feitas conforme necessário após operações de remoção ou inserção. Seja P o nó pai, FE o filho da esquerda de P e FD o filho da direita de P podemos definir 4 tipos diferentes de rotação:

Deve ser efetuada quando a diferença das alturas h dos filhos de P é igual a 2 e a diferença das alturas h dos filhos de FE é igual a 1. O nó FE deve tornar o novo pai e o nó P deve se tornar o filho da direita de FE. Segue pseudocódigo:

Seja Y o filho à esquerda de X

Torne o filho à direita de Y o filho à esquerda de X.

Torne X o filho à direita de Y



Caso 1.1 - Rotação Simples à Direita

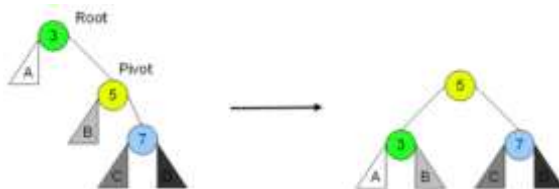
Rotação à Esquerda

Deve ser efetuada quando a diferença das alturas h dos filhos de P é igual a -2 e a diferença das alturas h dos filhos de FD é igual a -1 . O nó FD deve tornar o novo pai e o nó P deve se tornar o filho da esquerda de FD . Segue pseudocódigo:

Seja Y o filho à direita de X

Torne o filho à esquerda de Y o filho à direita de X .

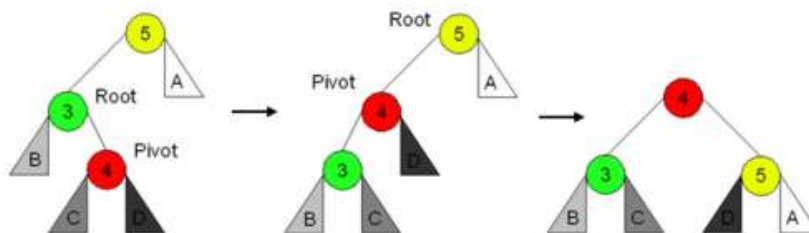
Torne X filho à esquerda de Y



Rotação Simples A Esquerda

Rotação Dupla À Direita

Deve ser efetuada quando a diferença das alturas h dos filhos de P é igual a 2 e a diferença das alturas h dos filhos de FE é igual a -1 . Nesse caso devemos aplicar uma rotação à esquerda no nó FE e, em seguida, uma rotação à direita no nó P .

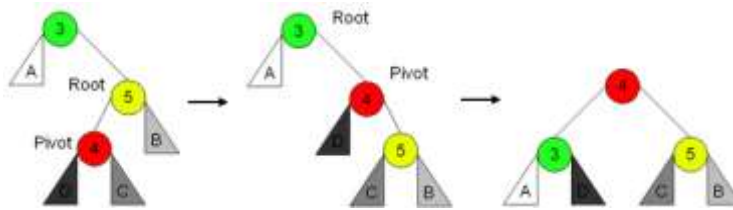


Caso 1.2 - Rotação dupla à direita

Deve ser observado que as 3 possíveis combinações de alturas das subárvores T_2 e T_3 constam da figura $(h, h-1; h, h; e h-1, h)$, implicando em nos respectivos balanços do nó u $(0/0/-1)$ e do nó p $(1/0/0)$.

Rotação Dupla À Esquerda

Deve ser efetuada quando a diferença das alturas h dos filhos de P é igual a -2 e a diferença das alturas h dos filhos de FD é igual a 1 . Nesse caso devemos aplicar uma rotação à direita no nó FD e, em seguida, uma rotação à esquerda no nó P .



Rotação dupla à esquerda

rotaçaoDireita(p:NoAVL):@NoAVL;

inicio

var u,v:@NoAVL;

u := p^.esq;

se u^.fb > 0 então

inicio // rotação dupla, conforme figura Caso 1.2

v := u^.dir;

u^.dir := v^.esq;

p^.esq := v^.dir;

v^.esq := u;

v^.dir := p;

caso v->fb

-1: u^.fb := 0; p^.fb := 1;

0: u^.fb := 0; p^.fb := 0;

1: u^.fb := -1; p^.fb := 0;

v^.fb := 0;

retornar v;

fim;

// rotaçao simples, conforme figura Caso 1.1

p^.esq := u^.dir;

u^.dir := p;

se u^.fb < 0 então

inicio

u^.fb := 0;

p^.fb := 0;

fim;

senão inicio // ocorre apenas na remoção - não relatado na literatura

$u^{fb} := 1;$

$p^{fb} := -1;$

fim;

retornar u;

fim;

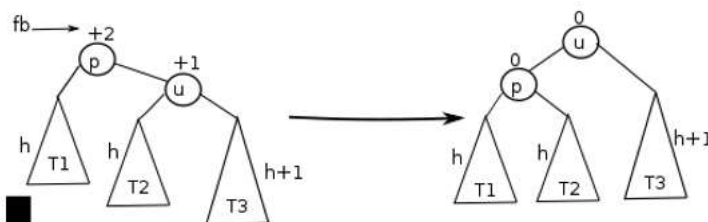
Cabe ressaltar, apesar de não estar relatado na literatura, que na remoção poderá ocorrer a situação na qual $u^{fb}=0$. Neste caso deverá ser aplicada rotação simples e o fator de balanço dos nós u e p serão respectivamente 1 e -1.

Rotação À Esquerda

Caso o fator de balanço do nó p tenha valor +2, então haverá necessidade de aplicar rotação simples ou dupla à esquerda.

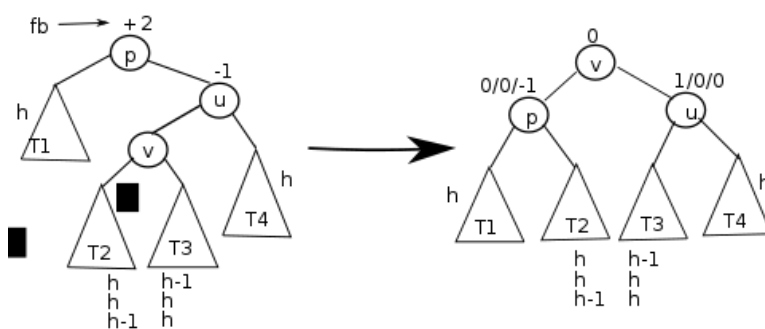
Para identificar se qual rotação aplicar, bastará analisar o fator de balanço do nó u, raiz da subárvore direita do nó p.

Caso o fator de balanço do nó u seja +1, então deverá ser aplicada uma rotação simples à esquerda. A figura a seguir mostra a configuração da árvore antes e depois da rotação.



Caso 2.1 - Rotação Simples à Esquerda

Caso o fator de balanço do nó u seja -1, então deverá ser aplicada uma rotação dupla à esquerda. A figura a seguir mostra a configuração da árvore antes e depois da rotação.



Caso 2.2 - Rotação Dupla à Direita

Aplicações

A árvore AVL é muito útil pois executa as operações de inserção, busca e remoção em tempo $O(\log n)$ sendo inclusive mais rápida que a árvore rubro-negra para aplicações que fazem uma quantidade excessiva de buscas, porém esta estrutura é um pouco mais lenta para inserção e remoção. Isso se deve ao fato de as árvores AVL serem mais rigidamente balanceadas.

Dicionários

Árvore AVL pode ser usada para formar um dicionário de uma linguagem ou de programas, como os opcodes de um assembler ou um interpretador.

Geometria Computacional

Árvore AVL pode ser usada também na geometria computacional por ser uma estrutura muito rápida. Sem uma estrutura com complexidade $O(\log n)$ alguns algoritmos da geometria computacional poderiam demorar dias para serem executados.

Conjuntos

Árvore AVL podem ser empregadas na implementação de conjuntos, principalmente aqueles cujas chave não são números inteiros.

A complexidade das principais operações de conjuntos usando árvore AVL:

Inserir - $O(\log n)$;

Remover - $O(\log n)$;

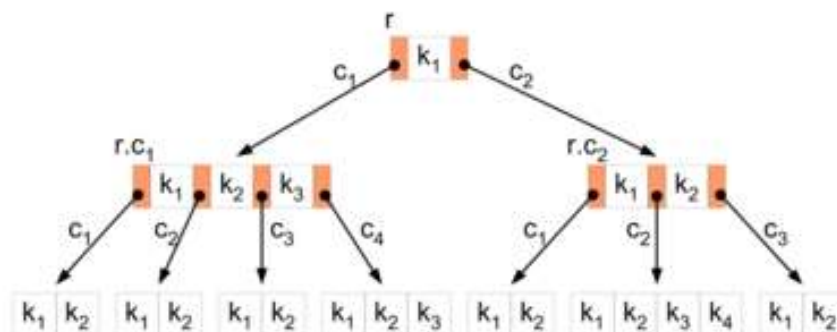
Pertence - $O(\log n)$;

União - $O(n \cdot \log n)$;

Interseção - $O(n \cdot \log n)$.

Árvore B

Árvore B		
Tipo	Árvore	
Ano	1971	
Inventado por	Rudolf Bayer, Edward Meyers McCreight	
Complexidade de Tempo em Notação big O		
Algoritmo	Caso Médio	Pior Caso
Espaço	O(n)	O(n)
Busca	O(log n)	O(log n)
Inserção	O(log n)	O(log n)
Remoção	O(log n)	O(log n)



Exemplo de Árvore B

Na ciência da computação uma árvore B é uma estrutura de dados projetada para funcionar especialmente em memória secundária como um disco magnético ou outros dispositivos de armazenamento secundário.

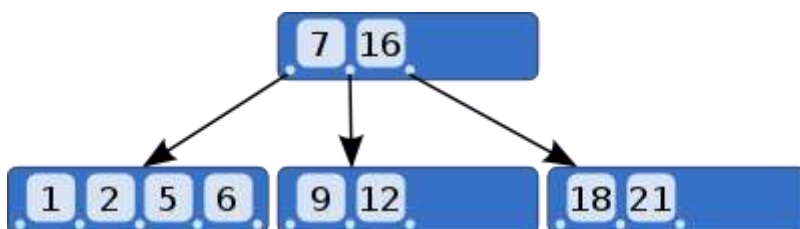
As árvores B são semelhantes as árvores preto e vermelho, mas são melhores para minimizar operações de E/S de disco. Muitos sistemas de bancos de dados usam árvores B ou variações da mesma para armazenar informações. Dentre suas propriedades ela permite a inserção, remoção e busca de chaves numa complexidade de tempo logarítmica e, por esse motivo, é muito empregada em aplicações que necessitam manipular grandes quantidades de informação tais como um banco de dados ou um sistema de arquivos.

Inventada por Rudolf Bayer e Edward Meyers McCreight em 1971 enquanto trabalhavam no Boeing Scientific Research Labs, a origem do nome (árvore B) não foi definida por estes.

Especula-se que o B venha da palavra balanceamento, do nome de um de seus inventores Bayer ou de Boeing, nome da empresa.

Árvores B são uma generalização das árvores binária de busca, pois cada nó de uma árvore binária armazena uma única chave de busca, enquanto as árvores B armazenam um número maior do que um de chaves de busca em cada nó, ou no termo mais usual para essa árvore, em cada página. Como a ideia principal das árvores B é trabalhar com dispositivos de memória secundária, quanto menos acessos a disco a estrutura de dados proporcionar, melhor será o desempenho do sistema na operação de busca sobre os dados manipulados.

Visão Geral



Árvore-B de ordem 2(Bayer & McCreight 1972) ou ordem 5 (Knuth 1998).

Os dispositivos de memória de um computador consistem na memória principal e secundária, sendo cada uma delas com suas características. A memória primária é mais conhecida como memória volátil de endereçamento direto (RAM), esta por sua vez apresenta baixo tempo de acesso, porém armazena um volume relativamente pequeno de informação e altos custos.

Já a memória secundária, possui um endereçamento indireto, armazena um grande volume de informação e possui um acesso (seek) muito lento quando comparada com a memória primária. A árvore B é uma solução para cenários em que o volume de informação é alto (e este não pode ser armazenado diretamente em memória primária) e, portanto, apenas algumas páginas da árvore podem ser carregadas em memória primária.

As árvores B são organizadas por nós, tais como os das árvores binárias de busca, mas estes apresentam um conjunto de chaves maior do que um e são usualmente chamados de páginas. As chaves em cada página são, no momento da inserção, ordenadas de forma crescente e para cada chave há dois endereços para páginas filhas, sendo que, o endereço à esquerda é para uma página filha com um conjunto de chaves menor e o à direita para uma página filha com um conjunto de chaves maior.

A figura acima demonstra essa organização de dados característica. Se um nó interno x contém $n[x]$ chaves, então x tem $n[x] + 1$ filhos. as chaves do nó x são usadas como pontos de divisão que separam o intervalo de chaves manipuladas por x em $x[x]$ subintervalos, cada qual manipulado por um filho de x .

Vale lembrar que todo este endereçamento está gravado em arquivo (memória secundária) e que um acesso a uma posição do arquivo é uma operação muito lenta. Através da paginação é possível carregar em memória primária uma grande quantidade de registros contidos numa única página e assim decidir qual a próxima página que o algoritmo de busca irá carregar em memória primária caso esta chave buscada não esteja na primeira página carregada. Após carregada uma página em memória primária, a busca de chave pode ser realizada linearmente sobre o conjunto de chaves ou através de busca binária.

Definição

Nó Ou Página

Um nó ou página, geralmente é representado por um conjunto de elementos apontando para seus filhos. Alguns autores consideram a ordem de uma árvore B como sendo a quantidade de registros que a página pode suportar. Outros consideram a ordem como a quantidade de campos apontadores. Todo nó da árvore tem um mínimo de registros definido pela metade da ordem, arredondando-se para baixo, caso a árvore seja de ordem ímpar, exceto a raiz da árvore, que pode ter um mínimo de um registro.

Por exemplo, os nós de uma árvore de ordem 5, podem ter, no mínimo $\lfloor 5/2 \rfloor$ registros, ou seja, dois registros. A quantidade de filhos que um nó pode ter é sempre a quantidade de registros do nó mais 1 ($V+1$). Por exemplo, se um nó tem 4 registros, este nó terá obrigatoriamente 5 apontamentos para os nós filhos.

Para definir uma árvore B devemos esclarecer os conceitos de ordem e página folha de acordo com cada autor Bayer e McCreight, comer, dentre outros, definem a ordem como sendo o número mínimo de chaves que uma página pode conter, ou seja, com exceção da raiz todas devem conter esse número mínimo de chaves, mas essa definição pode causar ambiguidades quando se quer armazenar um número máximo ímpar de chaves. Por exemplo, se uma árvore B é de ordem 3, uma página estará cheia quando tiver 6 ou 7 chaves? Ou ainda, se quisermos armazenar no máximo 7 chaves em cada página qual será a ordem da árvore, uma vez que, o mínimo de chaves é k e o máximo $2k$?

Knuth propôs que a ordem de uma árvore B fosse o número máximo de páginas filhas que toda página pode conter. Dessa forma, o número máximo de chaves por página ficou estabelecido como a ordem menos um.

O termo página folha também é inconsistente, pois é referenciado diferentemente por vários autores. Bayer e McCreight referem-se a estas como as páginas mais distantes da raiz, ou aquelas que contêm chaves no nível mais baixo da árvore. Já Knuth define o termo como as páginas que estão abaixo do último nível da árvore, ou seja, páginas que não contêm nenhuma chave.

De acordo com a definição de Knuth de ordem e página folha de Bayer e McCreight, uma árvore B de ordem d (número máximo de páginas filhas para uma página pai) deve satisfazer as seguintes propriedades:

Cada página contém no máximo d páginas filhas

Cada página, exceto a raiz e as folhas, tem pelo menos $\lfloor d/2 \rfloor$ páginas filhas

A página raiz tem ao menos duas páginas filhas (ao menos que ela seja uma folha)

Toda página folha possui a mesma profundidade, na qual é equivalente à altura da árvore

Uma página não folha com k páginas filha contém $k-1$ chaves

Uma página folha contém pelo menos $\lfloor d/2 \rfloor - 1$ chaves e no máximo $d-1$ chaves

Página Raiz

A página raiz das árvores B possuem o limite superior de $d-1$ chaves armazenadas, mas não apresentam um número mínimo de chaves, ou seja, elas podem ter um número inferior a $\lfloor d/2 \rfloor - 1$ de chaves. Na figura acima, essa página é representada pelo nó que possui o registro 7 e 16.

Páginas Internas

As páginas internas são as páginas em que não são folhas e nem raiz, estas devem conter o número mínimo $\lceil d/2 \rceil - 1$ e máximo $(d-1)$ de chaves.

Páginas Folha

Estes são os nós que possuem a mesma restrição de máximo e mínimo de chaves das páginas internas, mas estes não possuem apontadores para páginas filhas. Na figura acima são todos os demais nós exceto a raiz.

Estrutura Da Página

Uma possível estrutura de dados para uma página de árvore B na linguagem C:

```
# define D 5 //árvore de ordem 5

typedef struct BTPage{

    //armazena numero de chaves na pagina

    short int totalChaves;

    //vetor de chaves

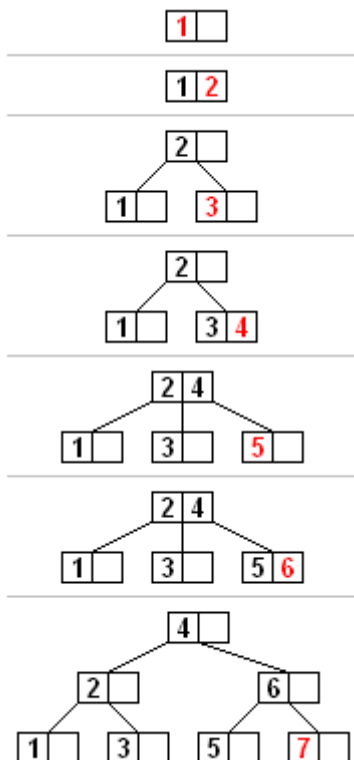
    int chaves[D-1];

    //Ponteiros das paginas filhas, -1 aponta para NULL

    struct BTPage filha[D];

}Page;
```

Operações Básicas



Exemplo de inserção em árvore B da sequência de 1 a 7. Os nós dessa árvore possuem no máximo 3 filhos

Altura De Uma Árvore B

O número de acessos ao disco exigidos para a maioria das operações em uma árvore B é proporcional a altura da árvore B.

Como Criar Uma Árvore Vazia

Para construir uma árvore B, primeiro criamos um nó raiz vazio, e depois inserimos novas chaves. Esses procedimentos alocam uma página de disco para ser usada como um novo nó no tempo $O(1)$

As Operações Básicas Sobre Um Árvore B São A Busca, Inserção E Remoção De Chaves.

Busca

A busca de uma chave k em uma árvore B é muito parecido com uma busca em árvore binária, exceto pelo fato de que, em vez de tomar uma decisão de ramificação binária ou de “duas vias” em cada nó, tomamos uma decisão de ramificação de várias vias, de acordo com o número de filhos do nó. Em cada nó interno x , tomamos uma decisão de ramificação de $(n[x] + 1)$ vias.

Esse método toma como entrada um ponteiro para o nó de raiz de uma subárvore e uma chave k a ser pesquisada.

Inserção

A operação de inserção, inicialmente com a árvore vazia, deve garantir que o nó raiz será criado. Criado o nó raiz, a inserção das próximas chaves seguem o mesmo procedimento: busca-se a posição correta da chave em um nó folha e insere a chave garantindo a ordenação destas. Após feito isso, considerando a abordagem de inserção de baixo para cima (Bottom-up) na árvore B, podem ocorrer duas situações:

Página folha está com um número menor de chaves do que o máximo permitido ($d-1$): Nesse caso apenas inserimos a chave de maneira ordenada na página

Página folha completa ou com o número máximo de chaves permitido ($d-1$): Nesse caso ocorre o overflow da página em questão e é necessário a operação de split para manter o balanceamento da árvore.

Primeiramente escolhe-se um valor intermediário na sequência ordenada de chaves da página incluindo-se a nova chave que deveria ser inserida. Este valor é exatamente uma chave que ordenada com as chaves da página estará no meio da sequência.

Cria-se uma nova página e os valores maiores do que a chave intermediária são armazenados nessa nova página e os menores continuam na página anterior (operação de split).

Esta chave intermediária escolhida deverá ser inserido na página pai, na qual poderá também sofrer overflow ou deverá ser criada caso em que é criada uma nova página raiz. Esta série de overflows pode se propagar para toda a árvore B, o que garante o seu balanceamento na inserção de chaves.

Uma abordagem melhorada para a inserção é a de cima para baixo (Top-down) que utiliza uma estratégia parecida com a inserção de baixo para cima, a lógica para a inserção das próximas chaves (levando em consideração que a raiz já está criada) é a seguinte: busca-se a posição correta da chave em um nó, porém durante a busca da posição correta todo nó que estiver com o número máximo de chaves ($d-1$) é feita a operação de split, adicionando o elemento intermediário na sequência ordenada de chaves da página no pai e separando os elementos da página em outras duas novas páginas, onde uma vai conter os elementos menores que o elemento intermediário e a outra os elementos maiores que ele, a inserção será feita em um nó folha somente após todo o processo de split e insere a chave garantindo a ordenação destas. Esta abordagem melhorada previne de ter que ficar fazendo chamadas sucessivas ao pai do nó, o que pode ser caro se o pai estiver na memória secundária.

Split

Trecho de uma árvore que tem ordem $m = 3$, sendo 10 o valor_central no nó que sofre o split.

A função do split é dividir o nó em duas partes e "subir" o valor central do nó para um nó acima ou, caso o nó que sofreu o split seja a raiz, criar uma nova raiz com um novo nó. O que ocorre quando é feito um split:

Primeiramente calcula-se qual a mediana dos valores do nó, no caso o valor central do nó. Sendo tamanho = quantidade de elementos no nó, mediana = tamanho/2 e usamos a mediana para acessar o elemento que se encontra no centro do nó, no caso valor_central = valores[mediana];

É testado se o nó que sofreu split tem pai, caso não, cria-se um novo nó apenas com o valor valor_central e o seta como a nova raiz. São criados mais dois nós, cada um irá conter os valores do nó que estavam antes da mediana e depois da mediana. Um nó terá os valores menores que o valor_central e ficará na primeira posição dos filhos da nova raiz, e o outro nó terá os valores maiores que o valor_central e ficará na segunda posição dos filhos da nova raiz;

Caso o nó tenha pai, adicionamos o valor_central ao nó pai. Caso o nó pai já esteja cheio, este também vai sofrer split após a inserção do valor nele. E da mesma forma que criamos dois nós para o caso do nó não ter pai, criaremos dois nós que conterão os valores menores e maiores que o valor_central. O nó com os menores valores ficará posicionado como filho do lado esquerdo do valor_central e o nó com os maiores valores ficará posicionado como filho do lado direito do valor_central. Por exemplo: Caso o valor_central seja inserido na posição 0 do array de valores do nó pai, o nó filho com os menores valores ficará na posição 0 do array de filhos, e o nó com os maiores valores ficará na posição 1 do array de filhos.

Remoção

A remoção é análoga a inserção, o algoritmo de remoção de uma árvore B deve garantir que as propriedades da árvore sejam mantidas, pois uma chave pode ser eliminada de qualquer página e não apenas de páginas folha. A remoção de um nó interno, exige que os filhos do nó sejam reorganizados. Como na inserção devemos nos resguardar contra a possibilidade da eliminação produzir uma árvore cuja estrutura viole as propriedades de árvores B. Da mesma maneira que tivemos de assegurar que um nó não ficará pequeno demais durante a eliminação (a não ser pelo fato da raiz pode ter essa pequena quantidade de filhos).

Se o método para remover a chave k da subárvore com raiz em x. Este método tem que está estruturado para garantir que quando ele for chamado recursivamente em um nó x, o número de chaves em x seja pelo menos o grau mínimo t. Essa condição exige uma chave além do mínimo exigido pelas condições normais da árvore B, de forma que, quando necessário, uma chave seja movida para dentro do nó filho.

Descrição de como a eliminação funciona:

Se a chave k está no nó x e x é uma folha, elimine a chave k de x.

Se a chave k está no nó x e x é um nó interno:

Se o filho y que precede k no nó x tem pelo menos t chaves, então encontre o predecessor k' de k na subárvore com raiz em y. Elimine recursivamente k', e substitua k por k' em x.

Simetricamente, se o filho z que segue k no nó x tem pelo menos t chaves, então encontre o sucessor k' de k na subárvore com raiz em z. Elimine recursivamente k' e substitua k por k' em x.

Caso contrário, se tanto y quanto z tem apenas t-1 chaves, faça a intercalação de k e todos os seus itens z em y, de modo que x perca tanto k quanto o ponteiro para z, e y contenha agora 2t-1 chaves.

Se a chave k não estiver presente no nó interno x, determine a raiz c[x] da subárvore apropriada que deve conter k, se k estiver absolutamente na árvore. Se c[x] tiver somente t - 1 chaves:

Se c[x] tiver somente t-1 chaves, mas tiver um irmão com t chaves, forneça a c[x] uma chave extra, movendo uma chave de x para baixo até c[x], movendo uma chave do irmão esquerdo ou direito imediato de c[x] para dentro de x, e movendo o ponteiro do filho apropriado do irmão para c[x]

Se $c[x]$ e todos os irmãos de $c[x]$ têm $t-1$ chaves, faça a intercalação de $c[x]$ com um único irmão, o que envolve mover uma chave de x para baixo até o novo nó intercalado, a fim de se tornar a chave mediana para esse nó

Nessas operações podem ocorrer underflows nas páginas, ou seja, quando há um número abaixo do mínimo permitido ($\lceil d/2 \rceil - 1$) de chaves em uma página.

Na remoção há vários casos a se analisar, as seguintes figuras apresentam alguns casos numa árvore de ordem 5:

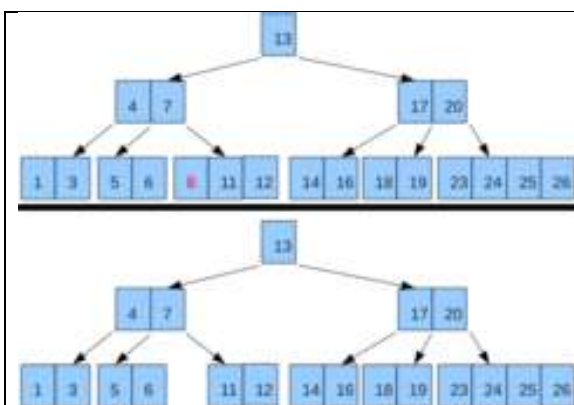


Figura 1: Remoção da chave 8 e posterior reorganização da estrutura

Caso da figura 1: Neste caso a remoção da chave 8 não causa o underflow na página folha em que ela está, portanto ela é simplesmente apagada e as outras chaves são reorganizadas mantendo sua ordenação.

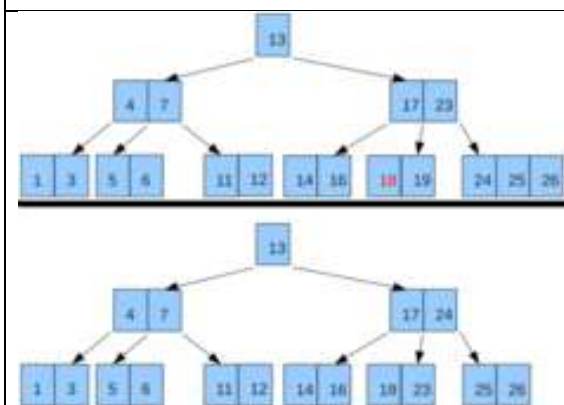


Figura 2: Remoção da chave 18 e posterior redistribuição das chaves

Caso da figura 2: O caso da figura 2 é apresentado a técnica de redistribuição de chaves. Na remoção da chave 18, a página que contém essa chave possui uma página irmã à direita com um número superior ao mínimo de chaves (página com chaves 24, 25 e 26) e, portanto, estas podem ser redistribuídas entre elas de maneira que no final nenhuma delas tenha um número inferior ao mínimo permitido.

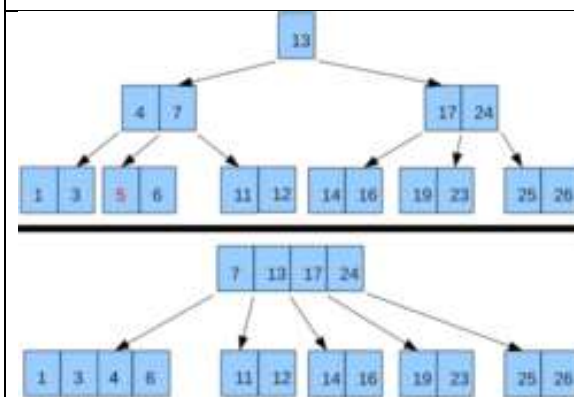


Figura 3: Remoção da chave 5 e posterior concatenação com página irmã à esquerda

Caso da figura 3: Nesta figura foi removido a chave 5, como não foi possível utilizar a técnica de redistribuição, pois as páginas irmãs possuem o número mínimo de chaves, então foi necessário concatenar o conteúdo da página que continha a chave 5 com sua página irmã à esquerda e a chave separadora pai. Ao final do processo a página pai fica com uma única chave (underflow) e é necessário diminuir a altura da árvore de maneira que o conteúdo da página pai e sua irmã, juntamente com a raiz, sejam concatenados para formar uma página única.

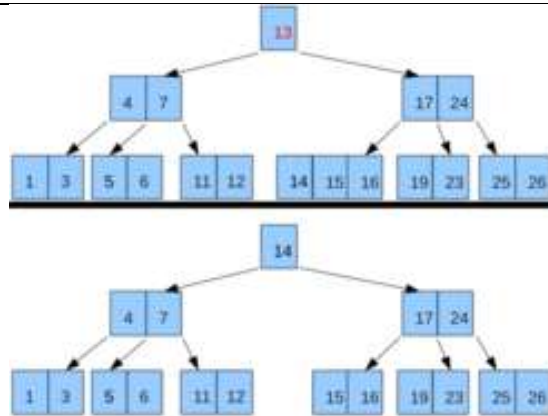


Figura 4: Remoção da chave 13 e promoção da menor chave da subárvore à direita de 13

Caso da figura 4: A remoção da chave 13 nesse caso foi realizado com a substituição do 13 pelo menor número da subárvore à direita de 13 que era o 14. Essa troca não causou o underflow da página em que estava o 14 e, portanto não gerou grandes alterações na árvore.

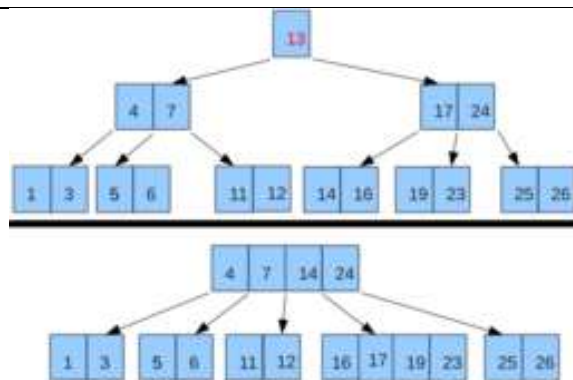


Figura 5: Chave 14 é promovida para a raiz o que causa underflow em sua página

Caso da figura 5: Caso semelhante ao anterior, mas esse ocorre o underflow da página que contém a menor chave da subárvore à direita de 13. Com isso, como não é possível a redistribuição, concatena-se o conteúdo dessa página com sua irmã à direita o que gera também underflow da página pai. O underflow da página pai também é resolvido com a concatenação com sua irmã e a raiz, resultando na diminuição da altura da árvore.

Algoritmos

Busca

Neste algoritmo recursivo os parâmetros recebidos inicialmente devem ser a chave buscada e um ponteiro para a página raiz da árvore B.

Busca(k, ponteiroRaiz)

```
{
se(ponteiroRaiz == -1)
{
return (chave nao encontrada)
}
senao
{
carrega em memoria primaria pagina apontado por ponteiroRaiz
procura k na pagina carregada
se(k foi encontrada)
```



```
{  
    return (chave encontrada)  
}  
  
senao  
{  
    ponteiro = ponteiro para a próxima página da possível ocorrência de k  
    return (Busca (k, ponteiro))  
}  
}  
}
```

Algoritmo De Busca Em Java

```
public BNodePosition<T> search(T element) {  
    return searchAux(root, element);  
}  
  
private BNodePosition<T> searchAux(BNode<T> node, T element) {  
    int i = 0;  
    BNodePosition<T> nodePosition = new BNodePosition<T>();  
    while (i <= node.elements.size() && element.compareTo(node.elements.get(i)) > 0) {  
        i++;  
    }  
    if (i <= node.elements.size() && element.equals(node.elements.get(i))) {  
        nodePosition.position = i;  
        nodePosition.node = node;  
        return nodePosition;  
    }  
    if (node.isLeaf()) {  
        return new BNodePosition<T>();  
    }  
    return searchAux(node.children.get(i), element);  
}
```

Inserção

O algoritmo de inserção em árvore B é um procedimento recursivo que inicialmente ponteiroRaiz aponta para a raiz da árvore em arquivo, key é a chave a ser inserida e chavePromovida representa a chave promovida após um split de uma página qualquer.

Inserção(ponteiroRaiz, key, chavePromovida)

```
{
    se(ponteiroRaiz == -1)//se ponteiroRaiz nao aponta para nenhuma pagina
    {
        chavePromovida = key
        return(flag que indica que houve promoção de chave)
    }
    senao
    {
        carregue a página P apontada por ponteiroRaiz em memoria primária
        busque por key nessa página P
        posicao = página no qual key poderia estar
    }
    se(key foi encontrada)
    {
        //chave ja esta na arvore, retorne uma flag de erro
        return(flag de erro)
    }
    flagRetorno = Insercao(posicao, key, chavePromovida)//procedimento recursivo
    se(flagRetorno indica que nao houve promoção de chave ou que ocorreu um erro)
    {
        return(conteudo de flagRetorno)
    }
    senao se(há espaço na página P para chavePromovida)
    {
        insere chavePromovida na página P
        escreve página P em arquivo
        return(flag que indica que nao houve promocao de chave)
    }
    senao //nao ha espaço em P para key
```

```
{  
    realize operação de split em P  
    escreva em arquivo a nova página e a página P  
    return(flag que indica que houve promoção de chave)  
}  
}
```

Inserção Recursiva

```
public void insertRec(BNode<T> node, T element) {  
    if (node.isLeaf()) {  
        node.addElement(element);  
        if (node.elements.size() > node.getMaxKeys()) {  
            node.split();  
        }  
    } else {  
        int position = searchPositionInParent(node.getElements(), element);  
        insertRec(node.getChildren().get(position), element);  
    }  
}  
  
// insert abordagem top-down  
public void insert(BNode<T> node, T element) {  
    if(node.isFull()) {  
        node = split(node); // Troque a referencia para o novo node.  
        // split retorna a referencia do no que contem a mediana do no anterior.  
    }  
    if(node.isLeaf()) {  
        node.addElement(element);  
        this.size++;  
    } else {  
        int i = 0;  
        while (i < node.size() && node.getElementAt(i).compareTo(element) < 0) {  
            i++;  
        }  
        insert(node.getChildren().get(i), element);  
    }  
}
```

```
}
```

```
}
```

Remoção

- 1 Busque a chave k
- 2 Busque a menor chave M na página folha da sub-árvore à direita de k
- 3 Se a chave k não está numa folha
- 4 {
- 5 Substitua k por M
- 6 }
- 7 Apague a chave k ou M da página folha
- 8 Se a página folha não sofrer underflow
- 9 {
- 10 fim do algoritmo
- 11 }
- 12 Se a página folha sofrer underflow, verifique as páginas irmãs da página folha
- 13 {
- 14 Se uma das páginas tiver um número maior do que o mínimo redistribua as chaves
- 15 Senão concatene as páginas com uma de suas irmãs e a chave pai separadora
- 16 }
- 17 Se ocorrer concatenação de páginas aplique o trecho das linhas 8 até 17 para a página pai da folha

Algoritmo split em Java

```
protected void split() {  
    int mediana = (size()) / 2;  
    BNode<T> leftChildren = this.copyLeftChildren(mediana);  
    BNode<T> rightChildren = this.copyRightChildren(mediana);  
    if (parent == null) {  
        parent = new BNode<T>(maxChildren);  
        parent.children.addFirst(this);  
    }  
    BNode<T> parent = this.parent;  
    int index = parent.indexOfChild(this);  
    parent.removeChild(this);  
    parent.addChild(index, leftChildren);
```

```
parent.addChild(index + 1, rightChildren);  
leftChildren.setParent(parent);  
rightChildren.setParent(parent);  
this.promote(media);  
if (parent.size() >= maxChildren) {  
    parent.split();  
}  
}
```

```
protected void promote(int mid) {  
    T element = elements.get(mid);  
    this.parent.addElement(element);  
}
```

Imprimir em Ordem em C

```
void emOrdem (tpaginaB raiz) {  
    if(raiz==NULL)  
        return;  
    for(int i=0;i<raiz.n,i++)  
        emOrdem(raiz->pont[i]);  
    printf("%i",raiz->chv[i]);  
}  
emOrdem(raiz->pont[raiz.n]);  
}
```

Algoritmo Split Dentro Da Classe Node Em Java

```
protected void split() {  
    T mediana = this.getElementAt(elements.size() / 2);  
    int posicao, esquerda, direita;  
    BNode<T> maior = new BNode<>(this.getMaxChildren());  
    BNode<T> menor = new BNode<>(this.getMaxChildren());  
    LinkedList<BNode<T>> criancas = new LinkedList<BNode<T>>();  
    this.armazenaElementos(mediana, maior, menor);  
    if (this.getParent() == null && this.isLeaf()) {  
        this.setElements(new LinkedList<T>());  
        this.addElement(mediana);  
    }  
}
```

```
        this.addChild(0, menor);
        this.addChild(1, maior);
    }
    else if (this.getParent() == null && !isLeaf()) {
        criancas = this.getChildren();
        this.setElements(new LinkedList<T>());
        this.addElement(mediana);
        this.setChildren(new LinkedList<BNode<T>>());
        this.addChild(0, menor);
        this.addChild(1, maior);
        this.reajustaFilhos(criancas, menor, 0, menor.size() + 1);
        this.reajustaFilhos(criancas, maior, maior.size() + 1, criancas.size());
    }
    else if (this.isLeaf()) {
        BNode<T> promote = new BNode<>(this.getMaxChildren());
        promote.getElements().add(mediana);
        promote.parent = this.getParent();
        menor.parent = this.getParent();
        maior.parent = this.getParent();
        posicao = buscaPosicaoNoPai(promote.getParent().getElements(), mediana);
        esquerda = posicao;
        direita = posicao + 1;
        this.getParent().getChildren().set(esquerda, menor);
        this.getParent().getChildren().add(direita, maior);
        promote.promote();
    }
    else {
        criancas = this.getChildren();
        BNode<T> paraPromote = new BNode<>(this.getMaxChildren());
        paraPromote.getElements().add(mediana);
        paraPromote.parent = this.getParent();
        menor.parent = this.getParent();
        maior.parent = this.getParent();
    }
}
```

```

posicao = buscaPosicaoNoPai(paraPromote.getElements(), mediana);

esquerda = posicao;

direita = posicao + 1;

this.getParent().getChildren().add(esquerda, menor);

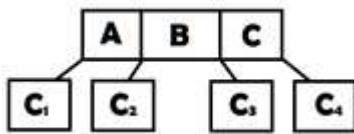
this.getParent().getChildren().add(direita, maior);

    }

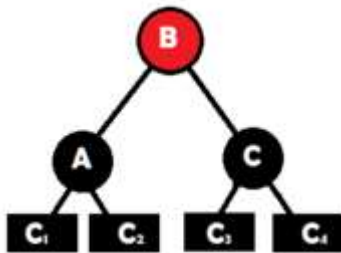
}

```

Árvores 2-3-4



Representação genérica de árvore B com três chaves e, conseqüentemente, quatro filhos.



Árvore preta e vermelha resultante de uma transformação de uma árvore B.

Árvores 2-3-4 são um tipo de árvore B que possuem uma, duas ou três chaves. E, conseqüentemente, dois, três ou quatro filhos. São utilizadas na implementação de dicionários. Além disso, servem como base para o desenvolvimento do código de árvores preto e vermelho.

Existem três situações na mudança de árvore B para árvore preto-vermelho:

Caso o nó só possua uma chave, basta transformá-lo num nó de cor preta e ligá-lo aos seus filhos correspondentes.

Caso o nó possua duas chaves, a chave mais à esquerda será transformada num nó preto e a mais à direita, num nó vermelho. O nó preto terá como filho da esquerda o primeiro nó filho da antiga árvore e como filho da direita o novo nó vermelho. Este, por sua vez, terá como filhos os dois filhos restantes da lista de filhos da árvore original.

Caso o nó possua três chaves, a chave do meio será transformada num nó vermelho e terá como filhos as antigas chaves adjacentes que serão nós pretos com os antigos filhos da árvore B.

Aplicando essas situações, deve-se checar se as propriedades de árvores preto-vermelho são mantidas como o valor do nó da esquerda ser menor que o nó atual.

Variações

As árvores B não são as únicas estruturas de dados usadas em aplicações que demandam a manipulação de grande volume de dados, também existem variações desta que proporcionam determinadas características como as árvores B+ e B*. Estas, por sua vez, se assemelham muito com as árvores B, mas possuem propriedades diferentes.

Estrutura de Controle

Em ciência da computação, estrutura de controle (ou fluxo de controle) refere-se à ordem em que instruções, expressões e chamadas de função são executadas ou avaliadas em programas de computador sob programação imperativa ou funcional.

Os tipos de estruturas de controle disponíveis diferem de linguagem para linguagem, mas podem ser cruamente caracterizados por seus efeitos. O primeiro é a continuação da execução em uma outra instrução, como na estrutura sequencial ou em uma instrução jump.

O segundo é a execução de um bloco de código somente se uma condição é verdadeira, uma estrutura de seleção. O terceiro é a execução de um bloco de código enquanto uma condição é verdadeira ou de forma a iterar uma coleção de dados, uma estrutura de repetição.

O quarto é a execução de instruções distantes entre si, em que o controle de fluxo possivelmente volte para a posição original posteriormente, como chamadas de subrotinas e corotinas. O quinto é a parada do programa de computador.

Interrupções e sinais são mecanismos de baixo nível que podem alterar o fluxo de controle de forma similar a uma sub-rotina, mas geralmente em resposta a algum estímulo externo ou um evento ao invés de uma estrutura de controle em uma linguagem.

Em nível de linguagem de máquina, as instruções de estruturas de controle geralmente funcionam ao alterar o contador de programa. Para algumas CPUs, as únicas instruções de estruturas de controle disponíveis são os diversos tipos de jump condicional.

Primitivas

Etiquetas

Uma etiqueta é um nome ou um número explícito atribuído a uma posição fixa no código fonte, e pode ser referenciada pelas instruções de fluxo de controle. Um exemplo é a atribuição de um número para cada linha do código fonte, frequente em linguagem de máquina.

Outras linguagens como C definem uma etiqueta como um identificador, geralmente aparecendo no início da linha, como por exemplo:

```
Sucesso:  
printf ("The operation was successful.\n");
```

Goto

O goto é a forma mais simples de transferência de controle incondicional, e seu efeito é fazer com que a próxima instrução executada seja aquela imediatamente após a etiqueta indicada.

O notável cientista da computação Edsger Dijkstra tinha posição contrária ao comando, até então um recurso bastante comum na programação da época, o que culminou no artigo de 1968 "A Case against the GO TO Statement".

Dijkstra alegava que o artifício era motivo para vários erros de programação. O artigo é considerado como um grande passo para a depreciação do comando em prol de estruturas de controle como o laço de repetição. O título mais famoso para o artigo, "Go To Statement Considered Harmful", foi um trabalho não de Dijkstra mas de Niklaus Wirth, então editor da Communications of the ACM, onde o artigo foi publicado.

Sub-Rotinas

A terminologia de sub-rotina varia, pois ela pode ser conhecida como rotina, procedimento, função ou método. Durante a década de 1950, a memória disponível em computadores era pequena, e as sub-rotinas reduziam o tamanho dos programas ao resumir somente uma vez alguma rotina que poderia ser usada em diversas partes do código. Atualmente elas são usadas para ajudar na estruturação dos

programas, isolando algoritmos ou ainda encapsulando algum método.

Estruturas

Estrutura Sequencial

Uma estrutura sequencial realiza um conjunto predeterminado de comandos de forma sequencial, na ordem em que foram declarados no código fonte. A cada instrução, o contador de programa é incrementado.

Estrutura de Seleção

Também chamada de expressão condicional ou ainda construção condicional, a estrutura de seleção realiza diferentes computações ou ações dependendo se a seleção (ou condição) é verdadeira ou falsa. A condição é uma expressão processada e transformada em um valor booleano.

Estrutura de Repetição

Uma estrutura de repetição realiza e repete diferentes computações ou ações dependendo se uma condição é verdadeira ou falsa, condição essa que é uma expressão processada e transformada em um valor booleano.

Está associado a ela além da condição (também chamada "expressão de controle" ou "condição de parada") o bloco de código: verifica-se a condição, e caso seja verdadeira, o bloco é executado. Após o final da execução do bloco, a condição é verificada novamente, e caso ela ainda seja verdadeira, o código é executado novamente.

Deve-se observar que, caso o bloco de código nunca modificar o estado da condição, a estrutura será executada para sempre, uma situação chamada laço infinito. Da mesma forma, é possível especificar uma estrutura em que o bloco de código modifica o estado da condição, mas esta é sempre verdadeira.

Algumas linguagens de programação especificam ainda uma palavra reservada para sair da estrutura de repetição de dentro do bloco de código, "quebrando" a estrutura. Também é oferecido por algumas linguagens uma palavra reservada para terminar uma iteração específica do bloco de código, forçando nova verificação da condição.

Comandos de Repetição

O VisuAlg implementa as três estruturas de repetição usuais nas linguagens de programação: o laço contado para...ate...faça (similar ao for...to...do do Pascal), e os laços condicionados enquanto...faça (similar ao while...do) e repita...ate (similar ao repeat...until).

A sintaxe destes comandos é explicada a seguir.

Para ... faça

Esta estrutura repete uma sequência de comandos um determinado número de vezes.

para <variável> de <valor-inicial> ate <valor-limite> [passo <incremento>] faça
<seqüência-de-comandos>
fimpara

<variável >	É a variável contadora que controla o número de repetições do laço. Na versão atual, deve ser necessariamente uma variável do tipo inteiro, como todas as expressões deste comando.
<valor-inicial>	É uma expressão que especifica o valor de inicialização da variável contadora antes da primeira repetição do laço.
<valor-limite >	É uma expressão que especifica o valor máximo que a variável contadora pode alcançar.

<incremento >	É opcional. Quando presente, precedida pela palavra passo, é uma expressão que especifica o incremento que será acrescentado à variável contadora em cada repetição do laço. Quando esta opção não é utilizada, o valor padrão de <incremento> é 1. Vale a pena ter em conta que também é possível especificar valores negativos para <incremento>. Por outro lado, se a avaliação da expressão <incremento > resultar em valor nulo, a execução do algoritmo será interrompida, com a impressão de uma mensagem de erro.
fimpara	Indica o fim da seqüência de comandos a serem repetidos. Cada vez que o programa chega neste ponto, é acrescentado à variável contadora o valor de <incremento >, e comparado a <valor-limite >. Se for menor ou igual (ou maior ou igual, quando <incremento > for negativo), a seqüência de comandos será executada mais uma vez; caso contrário, a execução prosseguirá a partir do primeiro comando que esteja após o fimpara.

<valor-inicial >, <valor-limite > e <incremento > são avaliados uma única vez antes da execução da primeira repetição, e não se alteram durante a execução do laço, mesmo que variáveis eventualmente presentes nessas expressões tenham seus valores alterados.

No exemplo a seguir, os números de 1 a 10 são exibidos em ordem crescente.

```
algoritmo "Números de 1 a 10"
var j: inteiro
inicio
para j de 1 ate 10 faca
  escreva (j:3)
fimpara
fimalgoritmo
```

Importante: Se, logo no início da primeira repetição, <valor-inicial > for maior que <valor-limite > (ou menor, quando <incremento> for negativo), o laço não será executado nenhuma vez. O exemplo a seguir não imprime nada.

```
algoritmo "Numeros de 10 a 1 (não funciona)"
var j: inteiro
inicio
para j de 10 ate 1 faca
  escreva (j:3)
fimpara
fimalgoritmo
```

Este outro exemplo, no entanto, funcionará por causa do passo -1:

```
algoritmo "Numeros de 10 a 1 (este funciona)"
var j: inteiro
inicio
para j de 10 ate 1 passo -1 faca
  escreva (j:3)
fimpara
fimalgoritmo
```

Enquanto ... faça

Esta estrutura repete uma seqüência de comandos enquanto uma determinada condição (especificada através de uma expressão lógica) for satisfeita.

```
enquanto <expressão-lógica> faca
  <seqüência-de-comandos>
fimenquanto
```

<expressão-lógica>	Esta expressão que é avaliada antes de cada repetição do laço. Quando seu resultado for VERDADEIRO, <seqüência-de-comandos> é executada.
fimenquanto	Indica o fim da <seqüência-de-comandos> que será repetida. Cada vez que a execução atinge este ponto, volta-se ao início do laço para que <expressão-lógica> seja avaliada novamente. Se o resultado desta avaliação for VERDADEIRO, a <seqüência-de-comandos> será executada mais uma vez; caso contrário, a execução prosseguirá a partir do primeiro comando após fimenquanto.

O mesmo exemplo anterior pode ser resolvido com esta estrutura de repetição:

```

algoritmo "Números de 1 a 10 (com enquanto...faca)"
var j: inteiro
inicio
j <- 1
enquanto j <= 10 faca
  escreva (j:3)
  j <- j + 1
fimenquanto
fimalgoritmo
  
```

Importante: Como o laço enquanto...faca testa sua condição de parada antes de executar sua seqüência de comandos, esta seqüência poderá ser executada zero ou mais vezes.

Repita ... até

Esta estrutura repete uma seqüência de comandos até que uma determinada condição (especificada através de uma expressão lógica) seja satisfeita.

```

repita
  <seqüência-de-comandos>
ate <expressão-lógica>
  
```

repita	Indica o início do laço.
ate <expressão-lógica>	Indica o fim da <seqüência-de-comandos> a serem repetidos. Cada vez que o programa chega neste ponto, <expressão-lógica> é avaliada: se seu resultado for FALSO, os comandos presentes entre esta linha e a linha repita são executados; caso contrário, a execução prosseguirá a partir do primeiro comando após esta linha.

Considerando ainda o mesmo exemplo:

```

algoritmo "Números de 1 a 10 (com repita)"
var j: inteiro
inicio
j <- 1
repita
  escreva (j:3)
  j <- j + 1
ate j > 10
fimalgoritmo
  
```

Importante: Como o laço repita...ate testa sua condição de parada depois de executar sua seqüência de comandos, esta seqüência poderá ser executada uma ou mais vezes.

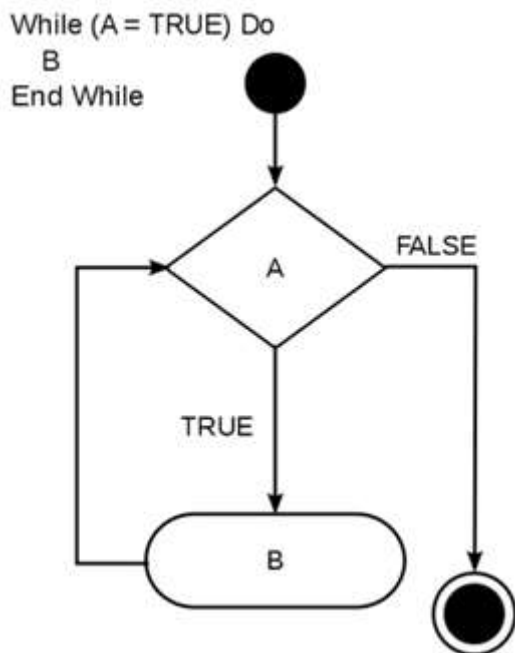
O que é estrutura de repetição?

Dentro da lógica de programação é uma estrutura que permite executar mais de uma vez o mesmo comando ou conjunto de comandos, de acordo com uma condição ou com um contador.

São utilizadas, por exemplo, para repetir ações semelhantes que são executadas para todos os elementos de uma lista de dados, ou simplesmente para repetir um mesmo processamento até que a condição seja satisfeita.

Existem 4 estruturas de repetição básica para praticamente todas as linguagens de programação, seja C ou javascript.

While (enquanto)



É dentre as 3 a mais simples.

Repete um bloco de código enquanto uma condição permanecer verdadeira

Caso a condição seja falsa, os comandos dentro do while não serão executados e a execução continuará com os comandos após o while

A repetição do while é controlada por uma condição que verifica alguma variável. Porém para que o while funcione corretamente é importante que essa variável sofra alteração dentro do while. Ex: um contador.

Após entrar dentro da repetição, o bloco de comandos sempre será executado, mesmo que dentro do bloco a variável que está controlando a execução seja alterada.

Exemplo de código:

```
numero = 42;
```

```
divisor = 1;
```

```
while (divisor <= numero) {
```

```
    resto = numero % divisor;
```

```
    if (resto == 0) {
```

```
        printf("Divisor encontrado: %d \n", divisor);
```

```

}

divisor = divisor + 1;

}

```

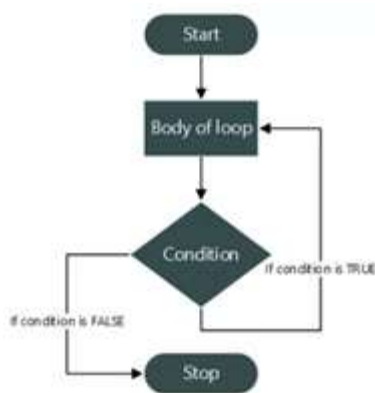
Quando Utilizar?

Não há necessidade de inicializar ou atualizar variáveis contadoras.

As etapas de inicialização ou atualização requerem muitas instruções e não caberiam elegantemente numa única linha do for.

As informações necessárias para avaliar a condição não dependem de uma variável contadora ou são obtidas durante a execução do bloco.

Do... While (faça enquanto)



Muito parecido com o while, porém tem uma diferença crucial: condição é verificada após executar o bloco de comandos.

Há uma bloco de comandos e logo depois uma verificação. Assim caso a variável condicional for alterada dentro do bloco de comandos, isso afetará a validação da condição.

A escolha entre while e do while é mínima, então dependerá do bom senso do programador, que optará pela estrutura que deixar o algoritmo mais simples e legível.

```

numeroA = 42;
numeroB = 2;

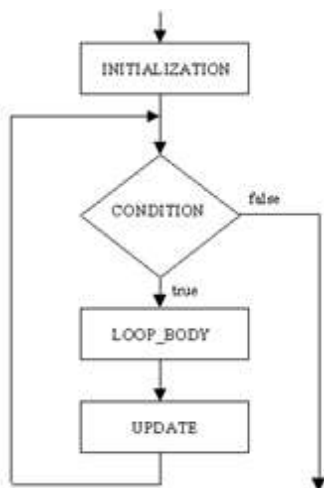
do {
    resto = numeroB % numeroA;
    numeroB = numeroA;
    numeroA = resto;
} while (numeroA > 0);

```

Quando utilizar?

Quando é necessário executar um bloco pelo menos uma vez para obter as informações necessárias para avaliar a condição.

For (para)



O For é utilizado para executar um conjunto de comandos executado por um número X de vezes.

É passada uma situação inicial, uma condição e uma ação a ser executada a cada repetição.

Uma variável é inicializada com uma valor inicial.

Essa variável é utilizada para controlar a quantidade de vezes em que o conjunto de comandos será executado.

E ao final do conjunto de comandos a variável sempre sofrerá uma alteração, aumentando ou diminuindo de acordo com a lógica utilizada.

```

for($contador = 0; $contador < 10; $contador++){
    echo $contador;
}
  
```

Quando utilizar?

O número de repetições é controlado por uma variável controladora.

Há necessidade de inicialização e atualização, mas que sejam simples o suficiente para que sejam acomodadas na linha do for. Para casos mais complexos, é melhor usar o comando while.

A avaliação da condição não depende de dados obtidos na execução do bloco.

O FOREACH é uma simplificação do operador FOR.

Permite acessar cada elemento individualmente iterando sobre toda a coleção sem a necessidade de informação de índices.

```

$vetor = array(1 => 'a', 2 => 'b', 3 => 'c', 4 => 'd', 5 => 'e');

foreach($vetor as $key => $item){
    echo $key . ' = ' . $item;
}
  
```

Quando utilizar?

Quando há uma coleção e você precisa acessar o valor ou dado que está no índice.

Análise da Complexidade de Algoritmos

Ao trabalhar com algoritmos, é comum que se encontre problemas ou gargalos de eficiência à medida que ele escala em tamanho ou complexidade. Entender a complexidade do algoritmo que se está criando ou aplicando é fundamental para planejar o trabalho com qualidade e estabilidade.

Apesar de ser um tema comum nos cursos superiores em Ciência da Computação, muitos desenvolvedores carecem de conhecimentos em Análise de Complexidade de Algoritmos.

Mas esse assunto não precisa ser motivo de estresse. Confira este artigo aprofundado e entenda melhor os fatores que tornam um algoritmo simples ou complexo.

Como tornar seu algoritmo mais escalável que as tretas do Twitter?

Algum tempo após começar a programar, há um momento em que se tem aquele “clique”. Você começa a pensar nos problemas do ponto de vista de um programador. Não tem mais volta: você estará eternamente fadado a olhar para um interruptor e pensar:

```
if interruptor == up {  
    turnLightOn()  
} else {  
    turnLightOff()  
}
```

Parece que o cérebro funciona com uma lógica booleana, mas não é isso. Você levou um bom tempo para que, em determinado momento, pudesse olhar para as coisas dessa forma. Isso é resultado de muito esforço, e ao mesmo tempo parece um “clique” porque não percebemos a nós mesmos com muita clareza.

Um desses “cliques” é o momento em que você deixa de pensar “Como posso resolver isso?” para pensar “Já sei como resolver, mas essa solução é suficiente?” ou “a solução é escalável ou legível?”.

Quando iniciantes, até conseguimos resolver problemas. Mas todo desafio proposto parece complicado, e nossa única preocupação é resolver da forma que for possível.

Isso muda conforme ganhamos experiência na solução de problemas. Agora, levamos menos tempo e temos o privilégio de pensar em uma implementação diferente.

Uma solução não-escalável é perfeitamente aceitável quando proposta ou encontrada por um iniciante. Ninguém deve cobrar de um estagiário, por exemplo, desenvolvimento de soluções sofisticadas ou escaláveis. Tampouco se deve encarregá-lo de pontos críticos de um sistema.

Não ter conhecimentos profundos em Algoritmos e Estrutura de Dados não torna ninguém um mau programador, mas estes conhecimentos certamente proporcionam um daqueles “cliques” que mencionamos.

Complexidade de Algoritmos

A complexidade de um algoritmo é analisada em termos de tempo e espaço. Normalmente, o algoritmo terá um desempenho diferente com base no processador, disco, memória e outros parâmetros de hardware.

A complexidade é usada para medir a velocidade de um algoritmo. Sendo o algoritmo um agrupamento de etapas para se executar uma tarefa, o tempo que leva para um algoritmo ser executado é baseado no número de passos. Digamos que um algoritmo percorre um array de dez posições somando o índice das posições a 200. A complexidade seria de $10 \times t$, sendo que t representa o tempo necessário para atualizar cada elemento do array com a operação de soma. Cada computador pode ser muito diferente: o tempo varia de acordo com o hardware, a linguagem utilizada e o sistema operacional.

Output:

```
indice[0] = 200  
indice[1] = 201  
indice[2] = 202  
indice[3] = 203  
indice[4] = 204  
indice[5] = 205  
indice[6] = 206  
indice[7] = 207  
indice[8] = 208  
indice[9] = 209
```

Temos 10 operações acontecendo, aparentemente muito simples. Com uma Notação é possível calcular a complexidade de um algoritmo sem precisar levar em consideração parâmetros de hardware.

Big O Notation

A Big O Notation é uma notação especial que indica a complexidade de um algoritmo. Na realidade, existem várias notações: big e small O, big e small omega e theta.

A notação do O é sobre um “limite por cima”; a omega, “limite por baixo”; e a theta é a combinação de ambos. ‘Small’ Notations representam afirmações mais rígidas sobre a complexidade do que ‘Big’ Notations.

Geralmente a Big O é mais utilizada porque o interesse está em verificar, sem tanta rigidez, o máximo de recursos que o algoritmo vai utilizar, e tentar reduzir isso quando possível.

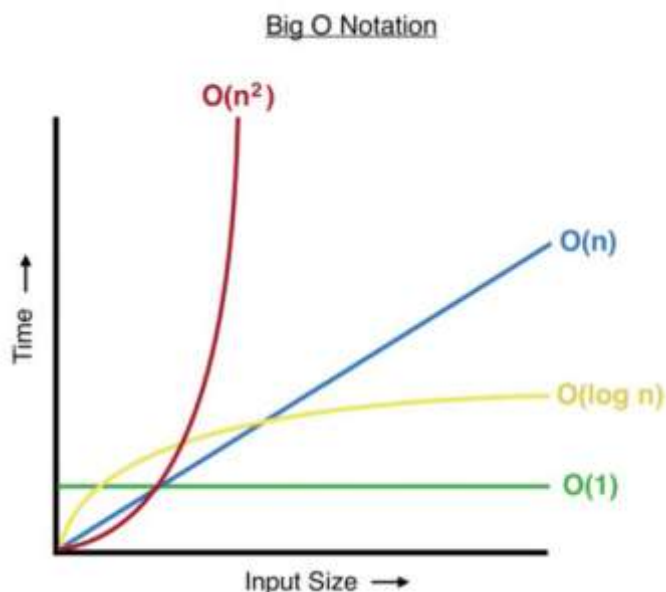
A função de tempo $T(n)$ representa a complexidade do algoritmo, tal que $T(n) = O(n)$ afirmando que um algoritmo tem uma complexidade linear de tempo, pois o TEMPO é relacionado a N, isto é, o tamanho de input do algoritmo.

Boa parte do material relacionado a Big O Notation é excessivamente formal. Isso talvez cause certo receio em algumas pessoas. Apesar disso, com alguma dedicação se torna fácil enxergar os padrões.

Em Big O Notation as complexidades de tempo linear, logarítmica, cúbica e quadrática são representações de complexidades diferentes de relação entre T e N em um algoritmo. A Big O Notation também é usada para determinar quanto espaço é consumido pelo algoritmo.

Você pode visualizar de forma gráfica o crescimento de tempo e de tamanho do input e como eles se relacionam. No final, é sobre isso: a relação entre um ou mais aspectos da entrada e o tempo de execução do algoritmo.

Importante: nem sempre o desempenho do algoritmo é afetado apenas pelo tamanho da entrada. O algoritmo de ordenação counting sort, por exemplo, é afetado também pelo maior número na lista.



Começaremos pelas principais funções de complexidade, com ênfase na complexidade de tempo. Ao longo da explicação, assuntos como recursão serão explicados conforme necessário.

Finalmente, abordemos a questão que trouxe você até aqui: como calcular a complexidade de um algoritmo?

Constantes $O(1)$

Fazemos isso contando as diferentes operações. Observe o exemplo:

```
package main

func main() {

}

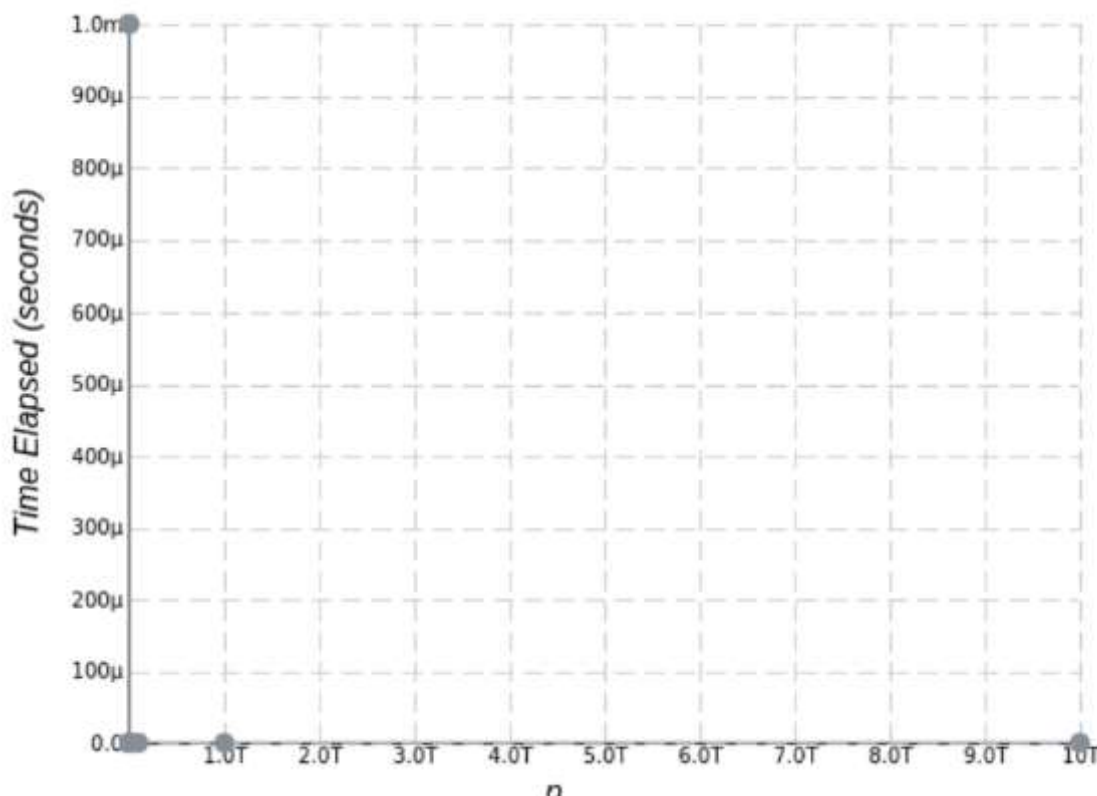
func constante(n int) int {
    return n * (n + 1) / 2
}
```

Aqui, temos uma multiplicação, uma adição e uma divisão – três operações. Não importa se N é igual a 2 ou 1 bilhão, o número de operações é o mesmo: três. Essa é uma complexidade de $O(3)$. Isto é o que se chama de complexidade constante, pois o número de operações não muda – mesmo quando o input varia.

Mas a notação Big O tem regras que tornam dispensável que você fique somando cada operação. $O(3)$ ou $O(200)$, no final o tempo constante é sempre $O(1)$.

Ademais, as constantes de um algoritmo são normalmente ignoradas. Isto porque a notação Big O se importa com o comportamento do algoritmo à medida que a entrada cresce, e não com os detalhes exatos para cada tamanho.

Quanto maior a entrada fica, menos importantes se tornam as constantes. Por isso, todo algoritmo com número de operações constante tem tempo de execução $O(1)$.



Esse gráfico demonstra como uma complexidade constante se comporta. No eixo vertical, temos a relação de tempo; no horizontal, a de N (input).

Na extremidade direita do eixo horizontal, temos um input de 10 TB sobre o algoritmo. Nessa simulação, ele gasta menos de um milésimo de execução. Mais à frente, analisaremos o mesmo gráfico em perspectiva a diferentes complexidades.

Complexidade Linear $O(n)$

Toda complexidade diferente da constante se dá em relação ao número de itens que a sua função recebe. Logo, quanto maior o input, maior o tempo de execução do algoritmo.

Apesar disso, na complexidade linear a diferença é bem proporcional ao tamanho da entrada, em vez de ser gritante. Em Big O Notation, complexidade linear é apresentada como $O(n)$.

Algoritmos de String Matching, como Boyer-Moore (usando a regra de Galil) e Ukkonen têm complexidade linear.

Mas a proposta não é pegar algoritmos conhecidos e procurar no Google a complexidade deles. O objetivo é que você enxergue os padrões nesses algoritmos, e entenda-os a ponto de flagrar a complexidade no seu código e no de outras pessoas.

A pergunta a ser feita não é “esse algoritmo é $O(n)$?”, mas sim “por que esse algoritmo é $O(n)$?”. Se ao final desta leitura você puder responder a essa última pergunta, você está começando a entender Big O Notation.

A complexidade linear $O(n)$ é demonstrada em um algoritmo da seguinte forma:

```
package main

import (
    "fmt"
)

func linear(n int) int {
    total := 0
    for i := 1; i <= n; i++ {
        total++
    }
    return total
}

func main() {
    fmt.Println(linear(10))
}
```

Aqui temos uma operação apenas – e não três, como anteriormente. Nesse caso, é `total++`, que é o mesmo que: `total = total + 1`. Porém, a complexidade é diferente, pois o input também define quantas vezes `i` será incrementado no `for` statement.

Então quanto maior for o input `(n)` maior o número de operações que serão feitas em `total`.

Se no lugar de `(i <= n)`, colocássemos `(i <= 10)` teríamos um tempo constante $O(1)$. Mas como o input interfere no `for`, essa complexidade é $O(n)$.

Ao analisar cada pedaço do código para encontrar o número de operações:

`total := 0` (Uma operação)

`i := 0` (Uma operação)

`i++` (N operações)

`total++` (N operações)

Para analisar a complexidade do algoritmo não é necessário contar todas as operações. Na maioria das vezes, você vai acabar percebendo de forma intuitiva.

Nesse caso, como temos operações relacionadas a N (que é o número de inputs), não há por que se preocupar com as que são constantes. Isso pois a parte linear já dominará a notação de complexidade, então se deixarmos de lado as constantes sobram apenas as $O(n)$.

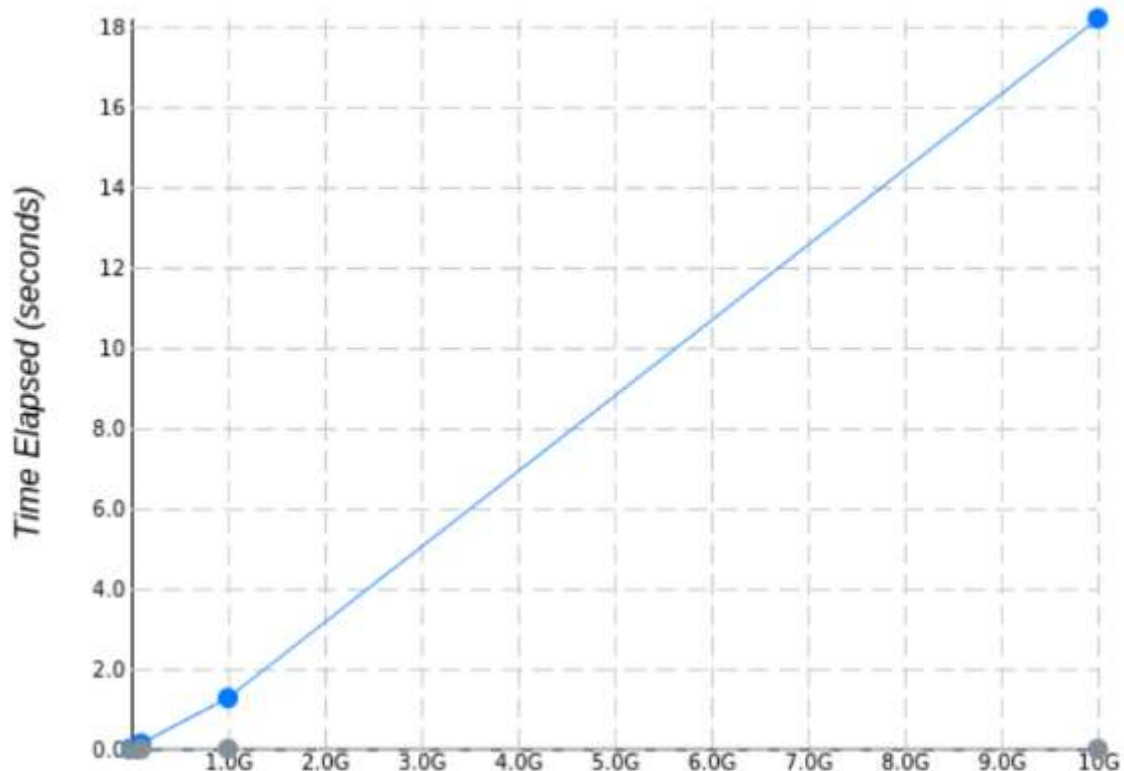
E se você tiver dois **for** statement?

```
package main
```

```
import "fmt"
```

```
func elevador(n) {  
    for i := 0 ; i < n; i++{  
        fmt.Println(i)  
    }  
  
    for j := n - 1 ; j >= 0; j--{  
        fmt.Println(j)  
    }  
}
```

Ainda assim, a complexidade será $O(n)$ e não $O(2n)$, porque Big O é sobre o comportamento quando a entrada cresce muito. Quanto maior fica a entrada, menos importa se é n ou $2n$.



Agora temos em perspectiva dois tipos de complexidade: a constante, em cinza, e a linear, em azul. Agora fica muito óbvio o porquê do nome.

Repare que, desta vez, o input é de 10 GB, e não 10 TB. Usar um input de 10 TB em um algoritmo de complexidade $O(n)$ consumiria bem mais tempo de execução.

A linha de $O(1)$ permanece plana – leva menos de 1 segundo de runtime. A azul, com um input de 10 GB, leva 18 segundos.

Ainda que o objetivo não seja comparar tempos de execução, sua diferença para cada input é bastante expressiva.

Complexidade Logarítmica $O(\log n)$

É necessário entender um pouco sobre logaritmos para entender a próxima notação. Basicamente, o que você precisa ter em mente é que logaritmos são o inverso de exponenciais.

Se você está estudando por conta própria e acha que saber sobre complexidade e estrutura de dados não vai fazer diferença, tome cuidado!

Busque compensar estas lacunas pois, quando você fizer um teste, possivelmente alguém que faz ciências da computação vai disputar a vaga com você. E esta pessoa já conhece isso por padrão.

Isso é um logaritmo:

$$\log_2(8) = 3$$

Lê-se: log na base 2 de 8 é igual a 3. E porque é igual a 3? Porque a pergunta desse logaritmo é:

“a qual potência o número 2 deve ser elevado para resultar em 8?”

A resposta será 3, pois:

$$2^3 = 2 * 2 * 2$$

e

$$2 * 2 * 2 = 8$$

Voltando aos algoritmos, através do logaritmo de N podemos encontrar o número de operações realizadas durante o runtime:

$$\log_2(n) = x$$

Infelizmente nem tudo são flores, e nem todo logaritmo é na base de 2. Mas esse cálculo não é a parte mais importante. Repare que, até agora, falamos sobre as complexidades que têm exemplos matemáticos, como a exponenciação, mas não fizemos cálculos para chegar à notação do algoritmo.

Então, para uma lista de 8 números, você teria que verificar 3 números no máximo.

Para uma lista de 1.024 elementos:

$$\log_2 1.024 = 10$$

porque

$$2^{10} = 1.024.$$

Então, para uma lista de 1.024 números, você tem que verificar 10 números no máximo.

Um exemplo de algoritmo com complexidade $O(\log n)$ é uma busca binária em uma lista já ordenada.


```
package main

import "fmt"

func main() {
    arr := []int{2, 3, 4, 10, 40}
    item := 9
    busca := buscaBinaria(arr, 0, len(arr), item)
    fmt.Println(busca)
}

func buscaBinaria(arr []int, esquerda, direita, item int) bool {
    for esquerda <= direita {
        meio := esquerda + (direita-esquerda)/2

        if arr[meio] == item {
            return true
        }

        if arr[meio] < item {
            esquerda = meio + 1
        }
    }
}
```

```
package main

import "fmt"

func main() {
    arr := []int{2, 3, 4, 10, 40}
    item := 9
    busca := buscaBinaria(arr, 0, len(arr), item)
    fmt.Println(busca)
}

func buscaBinaria(arr []int, esquerda, direita, item int) bool {
    for esquerda <= direita {
        meio := esquerda + (direita-esquerda)/2

        if arr[meio] == item {
            return true
        }

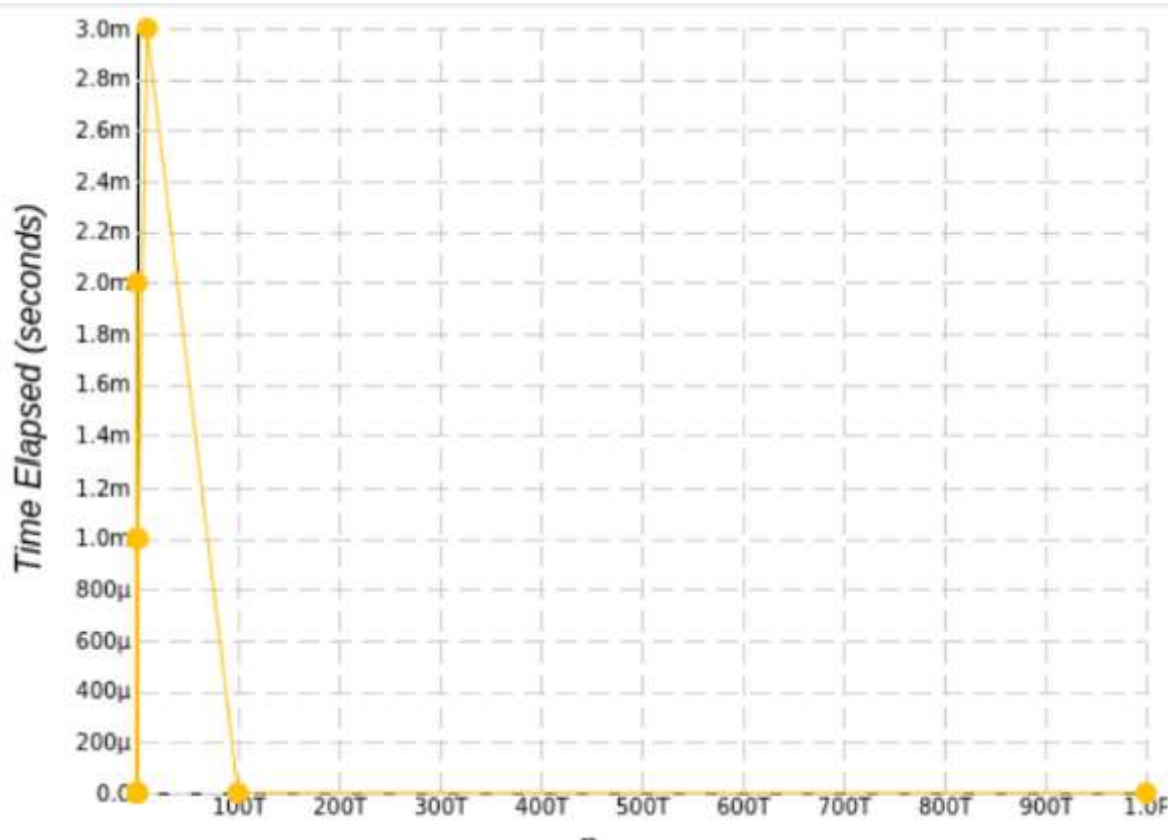
        if arr[meio] < item {
            esquerda = meio + 1
        }
    }
}
```



```
    } else {  
        direita = meio - 1  
    }  
}  
return false  
}
```

Esse tipo de algoritmo é bem simples: você parte o input ao meio e compara para verificar se o item a ser buscado é menor ou maior que o item no meio do array. Quando isso acontece, você descarta metade da lista – ficando com uma parte menor.

Esse processo é repetido até que se encontre o item da busca, diminuindo cada vez mais o processamento. Por isso ele é o inverso do exponencial: você diminui o N toda vez que um processamento é feito.



No exemplo do gráfico é possível observar que o número varia muito, porém o tempo é irrelevante em relação a outras complexidades. O input de 1 PB tem um tempo de execução de 3 milésimos de segundo.

O gráfico mostra como N aumenta e, logo em seguida, se torna quase uma constante. Isso acontece porque, mesmo que N duplique, o algoritmo estará fatiando N pela metade repetidas vezes, até que encontre o resultado.

Esse exemplo da busca binária só funciona quando se tem uma lista ordenada. Caso contrário, o algoritmo não poderia garantir que o elemento procurado está de fato em uma metade ou outra da lista, sendo necessária outra abordagem.

Complexidade Quadrática $O(N^2)$

```
package main

import (
    "fmt"
)

func printaTodosOsPares(n int) {
    for i := 0; i < n; i++ {
        for j := 0; j < n; j++ {
            fmt.Println(i, j)
        }
    }
}
```

Acabamos de falar sobre um caso em que se tem dois for e a complexidade é $O(n)$. Mas esse caso é diferente, pois o for é aninhado; não é como se fosse um caso de N^2 , mas sim N^2 . Se N fosse 10, o retorno seria 100. A estas proporções, o tempo de execução é muito maior.

Enquanto a complexidade linear sobe em uma linha reta, a complexidade quadrática desenha uma curva (parábola) em relação ao eixo de tempo para cada vez que N aumenta.

Nesse caso, a regra de pensar quem domina quem também é válida. Se você tiver dois for aninhados e um solto dentro da função, você não tem um $O(n + n^2)$ porque o N linear é insignificante perto do quanto N^2 cresce. Então, no final, conte apenas com a maior função: $O(n^2)$.

Até certo ponto, um algoritmo linear pode ser pior que um quadrático – justamente por causa das constantes. Em algum momento o quadrático vai ficar mais lento, mas e se isso só acontecer quando a entrada tiver mais de 500 terabytes? Vale a pena usar o linear?

Não se preocupe: para algoritmos simples e na maior parte dos casos de uso no mercado de software, o ponto onde o $O(n^2)$ fica mais lento é bem cedo.

Output:

```
00
01
10
11
```

Com um parâmetro de 2, esse seria o output, porque o índice i varia de 0 a n para cada um dos valores de j . O número de cálculos vai ser uma multiplicação deles, em vez da soma. Ou seja, muito mais passos para parâmetros maiores.

Há ainda a Complexidade Cúbica $O(n^3)$ – três loops aninhados – e a lógica é a mesma, mas não abordaremos o tema neste texto. Para exemplificar, entretanto:

```
package main

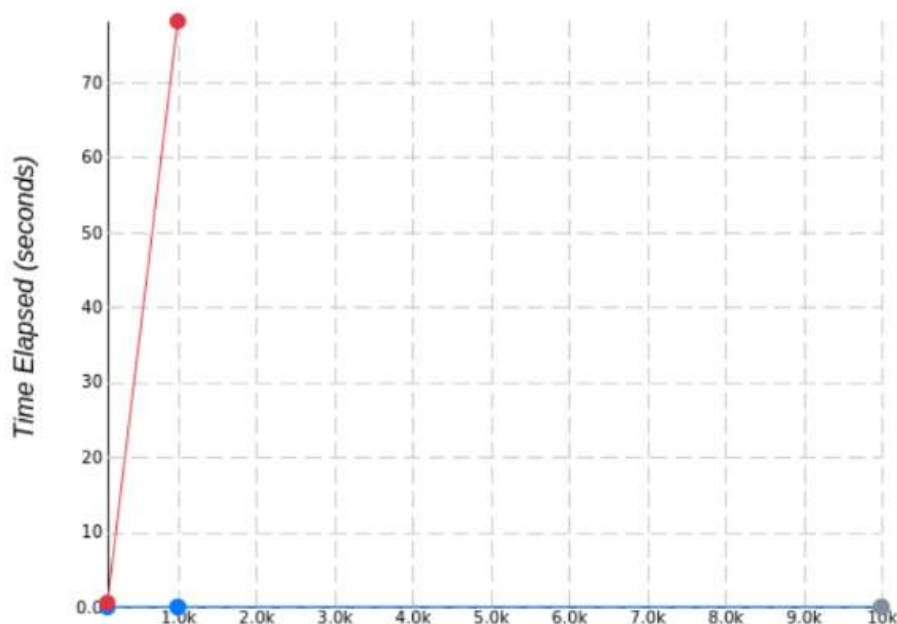
import (
    "fmt"
)

func main() {
    var k, l, m int

    var arr [10][10][10]int

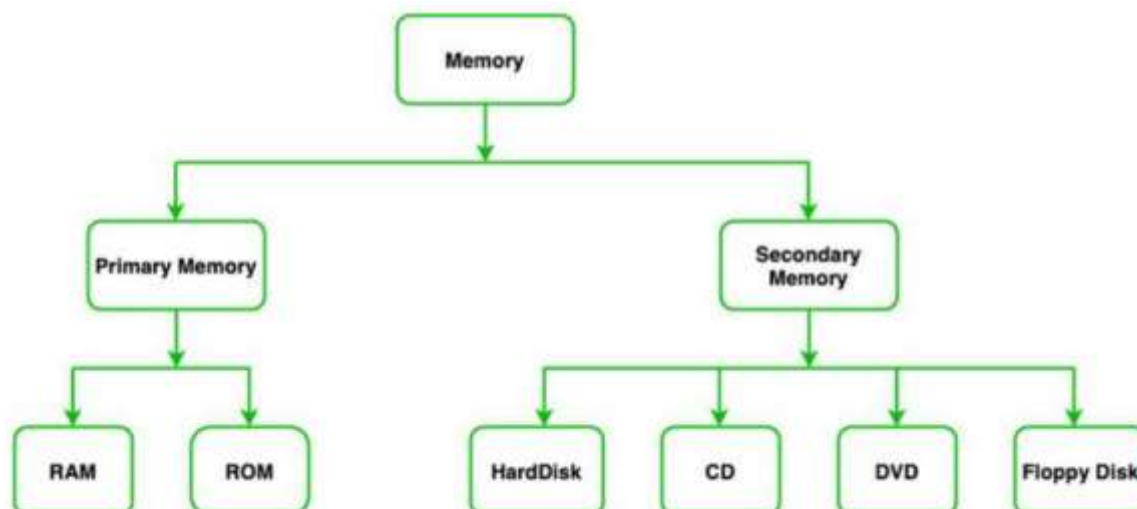
    for k = 0; k < 10; k++ {
        for l = 0; l < 10; l++ {
            for m = 0; m < 10; m++ {
                arr[k][l][m] = 1
                fmt.Println("Valor do elemento", k, l, m, " é", arr[k][l][m])
            }
        }
    }
}
```

Temos três loops percorrendo um array multidimensional. A complexidade aumenta tanto que, para percorrer esse array com três loops, são necessárias mil operações (pois $10 \times 10 \times 10 = 1000$). Se o tamanho do array aumenta, também cresce o número de operações.



Memória Secundária

Em um computador, a memória se refere aos dispositivos físicos usados para armazenar programas ou dados de forma temporária ou permanente. É um grupo de registros. A memória é de dois tipos (i) memória primária, (ii) memória secundária. A memória primária é composta de semicondutores, também é dividida em dois tipos, memória somente leitura (ROM) e memória de acesso aleatório (RAM). A memória secundária é um dispositivo físico para o armazenamento permanente de programas e dados (disco rígido, disco compacto, pen drive, etc.).



Memória Primária

A memória primária é composta por semicondutores e é a memória principal do sistema de computador. Geralmente é usado para armazenar dados ou informações em que o computador está trabalhando no momento, portanto, podemos dizer que é usado para armazenar dados temporariamente. Dados ou informações são perdidos quando os sistemas estão desligados. Também é dividido em dois tipos:

(eu). Memória somente leitura (ROM)

(ii). Memória de acesso aleatório (RAM).

1. Memória de acesso aleatório: a memória primária também é chamada de memória interna. Esta é a área principal de um computador onde os dados, instruções e informações são armazenados. Qualquer local de armazenamento nesta memória pode ser acessado diretamente pela Unidade Central de Processamento. Como a CPU pode acessar aleatoriamente qualquer local de armazenamento nesta memória, ela também é chamada de Memória de Acesso Aleatório ou RAM. A CPU pode acessar dados da RAM enquanto o computador estiver ligado. Assim que o computador é desligado, os dados e instruções armazenados desaparecem da RAM. Esse tipo de memória é conhecido como memória volátil. A RAM também é chamada de memória de leitura / gravação.

2. Memória somente leitura: a memória somente leitura (ROM) é um tipo de memória primária da qual as informações só podem ser lidas. Portanto, também é conhecido como memória somente leitura. O ROM pode ser acessado diretamente pela Unidade Central de Processamento. Porém, os dados e instruções armazenados na ROM são retidos mesmo quando o computador é desligado OU podemos dizer que ele mantém os dados após ser desligado. Esse tipo de memória é conhecido como memória não volátil.

Memória Secundária

Nós lemos até agora que a memória primária é volátil e tem capacidade limitada. Portanto, é importante ter outra forma de memória que tenha maior capacidade de armazenamento e da qual dados e programas não sejam perdidos ao desligar o computador. Esse tipo de memória é chamado de memória secundária. Na memória secundária, programas e dados são armazenados. Também é chamada de memória auxiliar.

É diferente da memória primária porque não é diretamente acessível através da CPU e não é volátil. Os dispositivos de armazenamento secundário ou externo têm uma capacidade de armazenamento muito maior e o custo da memória secundária é menor em comparação com a memória primária.

Uso de Memória Secundária

A memória secundária é usada para finalidades diferentes, mas as principais finalidades do uso da memória secundária são:

Armazenamento permanente: Como sabemos que a memória primária armazena dados apenas quando a fonte de alimentação está ligada, ela perde dados quando a energia é desligada. Portanto, precisamos de uma memória secundária para armazenar dados permanentemente, mesmo se a fonte de alimentação estiver desligada.

Grande armazenamento: a memória secundária fornece grande espaço de armazenamento para que possamos armazenar grandes dados como vídeos, imagens, áudios, arquivos, etc. permanentemente.

Portátil: alguns dispositivos secundários são removíveis. Assim, podemos facilmente armazenar ou transferir dados de um computador ou dispositivo para outro.

Tipos de Memória Secundária

A memória secundária é de dois tipos:

1. Armazenamento fixo

Na memória secundária, um armazenamento fixo é um dispositivo de mídia interno usado para armazenar dados em um sistema de computador. O armazenamento fixo é geralmente conhecido como unidades de disco fixas ou unidades de disco rígido.

Geralmente, os dados do sistema de computador são armazenados em um dispositivo de armazenamento fixo embutido.

Armazenamento fixo não significa que você não possa removê-los do sistema do computador, você pode remover o dispositivo de armazenamento fixo para reparo, atualização ou manutenção, etc. com a ajuda de um especialista ou engenheiro.

Tipos de armazenamento fixo:

A seguir estão os tipos de armazenamento fixo:

Memória flash interna (rara)

SSD (disco de estado sólido)

Unidades de disco rígido (HDD)

2. Armazenamento removível

Na memória secundária, o armazenamento removível é um dispositivo de mídia externa usado para armazenar dados em um sistema de computador.

O armazenamento removível é geralmente conhecido como unidades de disco ou unidades externas. É um dispositivo de armazenamento que pode ser inserido ou removido do computador de acordo com nossos requisitos.

Podemos removê-los facilmente do sistema do computador enquanto o sistema do computador está em execução.

Os dispositivos de armazenamento removíveis são portáteis para que possamos facilmente transferir dados de um computador para outro. Além disso, os dispositivos de armazenamento removíveis fornecem as taxas de transferência de dados rápidas associadas a redes de área de armazenamento (SANs).

Tipos de armazenamento removível:

Discos óticos (como CDs, DVDs, discos Blu-ray, etc.)

Cartões de memória

Disquetes

Fitas magnéticas

Pacotes de disco

Armazenamento de papel (como fitas perfuradas, cartões perfurados, etc.)

Dispositivos de Memória Secundária

A seguir estão os dispositivos de memória secundária comumente usados:

1. Disquete: Um disquete consiste em um disco magnético em uma caixa plástica quadrada. É usado para armazenar dados e transferir dados de um dispositivo para outro. Os disquetes estão disponíveis em dois tamanhos (a) Tamanho: 3,5 polegadas, a capacidade de armazenamento de 1,44 MB (b) Tamanho: 5,25 polegadas, a capacidade de armazenamento de 1,2 MB. Para usar um disquete, nosso computador precisa ter uma unidade de disquete. Este dispositivo de armazenamento se tornou obsoleto agora e foi substituído por CDs, DVDs e drives flash.

2. Compact Disc: Um Compact Disc (CD) é um dispositivo de armazenamento secundário comumente usado. Ele contém trilhas e setores em sua superfície. Seu formato é circular e é feito de plástico polycarbonato. A capacidade de armazenamento do CD é de até 700 MB de dados. Um CD também pode ser chamado de CD-ROM (Compact Disc Read-Only Memory), neste computador pode ler os dados presentes em um CD-ROM, mas não pode gravar novos dados nele. Para um CD-ROM, exigimos um CD-ROM. O CD é de dois tipos:

CD-R (disco compacto gravável): Uma vez que os dados foram gravados nele não podem ser apagados, eles podem apenas ser lidos.

CD-RW (disco compacto regravável): É um tipo especial de CD no qual os dados podem ser apagados e regravados quantas vezes quisermos. Também é chamado de CD apagável.

3. Digital Versatile Disc: Um Digital Versatile Disc também conhecido como DVD tem a aparência de um CD, mas a capacidade de armazenamento é maior em comparação com o CD, ele armazena até 4,7 GB de dados. A unidade de DVD-ROM é necessária para usar o DVD em um computador. Os arquivos de vídeo, como filmes ou gravações de vídeo, etc., geralmente são armazenados em DVD e você pode executar o DVD usando o DVD player. O DVD é de três tipos:

DVD-ROM (Digital Versatile Disc Readonly): No DVD-ROM, o fabricante grava os dados nele e o usuário só pode ler esses dados, não pode gravar novos dados nele. Por exemplo, um DVD de filme, um DVD de filme já gravado pelo fabricante, só podemos assistir ao filme, mas não podemos gravar novos dados nele.

DVD-R (disco digital versátil gravável): No DVD-R, você pode gravar os dados, mas apenas uma vez. Uma vez que os dados foram gravados nele não podem ser apagados, eles podem apenas ser lidos.

DVD-RW (Disco Digital Versátil Regravável e Apagável): É um tipo especial de DVD no qual os dados podem ser apagados e regravados quantas vezes quisermos. Também é chamado de DVD apagável.

4. Disco Blu-ray: Um disco Blu-ray se parece com um CD ou DVD, mas pode armazenar dados ou informações de até 25 GB de dados. Se quiser usar um disco Blu-ray, você precisa de um leitor de Blu-ray. O nome Blu-ray deriva da tecnologia usada para ler o disco 'Blu' do laser azul-violeta e 'ray' de um raio óptico.

5. Disco rígido: um disco rígido é uma parte de uma unidade chamada unidade de disco rígido. É usado para armazenar uma grande quantidade de dados. Os discos rígidos ou unidades de disco rígido vêm em diferentes capacidades de armazenamento (como 256 GB, 500 GB, 1 TB e 2 TB, etc.).

Métodos De Ordeção

Imagine como seria buscar um número em um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética? Seria muito complicado. A ordenação ou classificação de registros consiste em organizá-los em ordem crescente ou decrescente e assim facilitar a recuperação desses dados. A ordenação tem como objetivo facilitar as buscas e pesquisas de ocorrências de determinado elemento em um conjunto ordenado.

Como preza a estratégia algorítmica: "Primeiro coloque os números em ordem. Depois decidimos o que fazer."

Na computação existe uma série de algoritmos que utilizam diferentes técnicas de ordenação para organizar um conjunto de dados, eles são conhecidos como Métodos de Ordenação ou Algoritmos de Ordenação. Vamos conhecer um pouco mais sobre eles.

Os métodos de ordenação se classificam em:

- Ordenação Interna: onde todos os elementos a serem ordenados cabem na memória principal e qualquer registro pode ser imediatamente acessado.
- Ordenação Externa: onde os elementos a serem ordenados não cabem na memória principal e os registros são acessados sequencialmente ou em grandes blocos.

Hoje veremos apenas os métodos de ordenação interna.

Dentro da ordenação interna temos os Métodos Simples e os Métodos Eficientes:

Métodos Simples

Os métodos simples são adequados para pequenos vetores, são programas pequenos e fáceis de entender. Possuem complexidade $C(n) = O(n^2)$, ou seja, requerem $O(n^2)$ comparações. Exemplos: Insertion Sort, Selection Sort, Bubble Sort, Comb Sort.

Dica: Veja uma breve introdução à análise de algoritmos

Nos algoritmos de ordenação as medidas de complexidade relevantes são:

- Número de comparações $C(n)$ entre chaves.
- Número de movimentações $M(n)$ dos registros dos vetores.

Onde n é o número de registros.

Insertion Sort

Insertion Sort ou ordenação por inserção é o método que percorre um vetor de elementos da esquerda para a direita e à medida que avança vai ordenando os elementos à esquerda. Possui complexidade $C(n) = O(n)$ no melhor caso e $C(n) = O(n^2)$ no caso médio e pior caso. É considerado um método de ordenação estável.

Um método de ordenação é estável se a ordem relativa dos itens iguais não se altera durante a ordenação.

O funcionamento do algoritmo é bem simples: consiste em cada passo a partir do segundo elemento selecionar o próximo item da sequência e colocá-lo no local apropriado de acordo com o critério de ordenação.

Vejam os a implementação:

```
void insercao (int vet, int tam){
    int i, j, x;
    for (i=2; i<=tam; i++){
        x = vet[i];
        j=i-1;
        vet[0] = x;
        while (x < vet[j]){
            vet[j+1] = vet[j];
            j--;
        }
        vet[j+1] = x;
    }
}
```

Selection Sort

A ordenação por seleção ou selection sort consiste em selecionar o menor item e colocar na primeira posição, selecionar o segundo menor item e colocar na segunda posição, segue estes passos até que reste um único elemento. Para todos os casos (melhor, médio e pior caso) possui complexidade $C(n) = O(n^2)$ e não é um algoritmo estável.

```
void selecao (int vet, int tam){
    int i, j, min, x;
    for (i=1; i<=n-1; i++){
        min = i;
        for (j=i+1; j<=n; j++){
            if (vet[j] < vet[min])
                min = j;
        }
        x = vet[min];
        vet[min] = vet[i];
        vet[i] = x;
    }
}
```

Métodos Eficientes

Os métodos eficientes são mais complexos nos detalhes, requerem um número menor de comparações. São projetados para trabalhar com uma quantidade maior de dados e possuem complexidade $C(n) = O(n \log n)$. Exemplos: Quick sort, Merge sort, Shell sort, Heap sort, Radix sort, Gnome sort, Count sort, Bucket sort, Cocktail sort, Timsort.

Quick Sort

O Algoritmo Quicksort, criado por C. A. R. Hoare em 1960, é o método de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.

Provavelmente é o mais utilizado. Possui complexidade $C(n) = O(n^2)$ no pior caso e $C(n) = O(n \log n)$ no melhor e médio caso e não é um algoritmo estável.

É um algoritmo de comparação que emprega a estratégia de "divisão e conquista". A ideia básica é dividir o problema de ordenar um conjunto com n itens em dois problemas menores. Os problemas menores são ordenados independentemente e os resultados são combinados para produzir a solução final.

Basicamente a operação do algoritmo pode ser resumida na seguinte estratégia: divide sua lista de entrada em duas sub-listas a partir de um pivô, para em seguida realizar o mesmo procedimento nas duas listas menores até uma lista unitária.

Funcionamento do algoritmo:

- Escolhe um elemento da lista chamado pivô.

- Reorganiza a lista de forma que os elementos menores que o pivô fiquem de um lado, e os maiores fiquem de outro. Esta operação é chamada de "particionamento".
- Recursivamente ordena a sub-lista abaixo e acima do pivô.

Dica: Conheça um pouco sobre algoritmos recursivos

```
void quick(int vet[], int esq, int dir){
    int pivo = esq, i, ch, j;
    for(i=esq+1; i<=dir; i++){
        j = i;
        if(vet[j] < vet[pivo]){
            ch = vet[j];
            while(j > pivo){
                vet[j] = vet[j-1];
                j--;
            }
            vet[j] = ch;
            pivo++;
        }
    }
    if(pivo-1 >= esq){
        quick(vet, esq, pivo-1);
    }
    if(pivo+1 <= dir){
        quick(vet, pivo+1, dir);
    }
}
```

A principal desvantagem deste método é que ele possui uma implementação difícil e delicada, um pequeno engano pode gerar efeitos inesperados para determinadas entradas de dados.

Mergesort

Criado em 1945 pelo matemático americano John Von Neumann o Mergesort é um exemplo de algoritmo de ordenação que faz uso da estratégia "dividir para conquistar" para resolver problemas. É um método estável e possui complexidade $C(n) = O(n \log n)$ para todos os casos.

Esse algoritmo divide o problema em pedaços menores, resolve cada pedaço e depois junta (merge) os resultados. O vetor será dividido em duas partes iguais, que serão cada uma divididas em duas partes, e assim até ficar um ou dois elementos cuja ordenação é trivial.

Para juntar as partes ordenadas os dois elementos de cada parte são separados e o menor deles é selecionado e retirado de sua parte. Em seguida os menores entre os restantes são comparados e assim se prossegue até juntar as partes.

```
void mergeSort(int *vetor, int posicaoInicio, int posicaoFim) {
    int i, j, k, metadeTamanho, *vetorTemp;
    if(posicaoInicio == posicaoFim) return;
    metadeTamanho = (posicaoInicio + posicaoFim) / 2;
    mergeSort(vetor, posicaoInicio, metadeTamanho);
    mergeSort(vetor, metadeTamanho + 1, posicaoFim);
    i = posicaoInicio;
    j = metadeTamanho + 1;
    k = 0;
    vetorTemp = (int *) malloc(sizeof(int) * (posicaoFim - posicaoInicio + 1));
```

```
while(i < metadeTamanho + 1 || j < posicaoFim + 1) {  
    if (i == metadeTamanho + 1) {  
        vetorTemp[k] = vetor[j];  
        j++;  
        k++;  
    }  
    else {  
        if (j == posicaoFim + 1) {  
            vetorTemp[k] = vetor[i];  
            i++;  
            k++;  
        }  
        else {  
            if (vetor[i] < vetor[j]) {  
                vetorTemp[k] = vetor[i];  
                i++;  
                k++;  
            }  
            else {  
                vetorTemp[k] = vetor[j];  
                j++;  
                k++;  
            }  
        }  
    }  
    for(i = posicaoInicio; i <= posicaoFim; i++) {  
        vetor[i] = vetorTemp[i - posicaoInicio];  
    }  
    free(vetorTemp);  
}
```

Shell Sort

Criado por Donald Shell em 1959, o método Shell Sort é uma extensão do algoritmo de ordenação

por inserção. Ele permite a troca de registros distantes um do outro, diferente do algoritmo de ordenação por inserção que possui a troca de itens adjacentes para determinar o ponto de inserção. A complexidade do algoritmo é desconhecida, ninguém ainda foi capaz de encontrar uma fórmula fechada para sua função de complexidade e o método não é estável.

Os itens separados de h posições (itens distantes) são ordenados: o elemento na posição x é comparado e trocado (caso satisfaça a condição de ordenação) com o elemento na posição $x-h$. Este processo repete até $h=1$, quando esta condição é satisfeita o algoritmo é equivalente ao método de inserção.

A escolha do salto h pode ser qualquer sequência terminando com $h=1$. Um exemplo é a sequência abaixo:

```
h(s) = 1, para s = 1  
h(s) = 3h(s - 1) + 1, para s > 1
```

A sequência corresponde a 1, 4, 13, 40, 121, ...

Knuth (1973) mostrou experimentalmente que esta sequência é difícil de ser batida por mais de 20% em eficiência.

```
void shellSort(int *vet, int size) {  
    int i, j, value;  
    int gap = 1;  
    while(gap < size) {  
        gap = 3*gap+1;  
    }  
    while ( gap > 1) {  
        gap /= 3;  
        for(i = gap; i < size; i++) {  
            value = vet[i];  
            j = i - gap;  
            while (j >= 0 && value < vet[j]) {  
                vet [j + gap] = vet[j];  
                j -= gap;  
            }  
            vet [j + gap] = value;  
        }  
    }  
}
```

Conclusão

Neste artigo foi apresentado os principais métodos de ordenação com foco no conceito e funcionamento de cada um deles.

Você pode ver na tabela abaixo a comparação entre eles:

Algoritmo	Comparações			Movimentações			Espaço	Estável	In situ
	Melhor	Médio	Pior	Melhor	Médio	Pior			
Bubble	$O(n^2)$			$O(n^2)$			$O(1)$	Sim	Sim
Selection	$O(n^2)$			$O(n)$			$O(1)$	Não*	Sim
Insertion	$O(n)$	$O(n^2)$		$O(n)$	$O(n^2)$		$O(1)$	Sim	Sim
Merge	$O(n \log n)$			–			$O(n)$	Sim	Não
Quick	$O(n \log n)$		$O(n^2)$	–			$O(n)$	Não*	Sim
Shell	$O(n^{1.25})$ ou $O(n (\ln n)^2)$			–			$O(1)$	Não	Sim

* Existem versões estáveis.

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.

Linguagens de Programação

JavaScript (abreviado como "JS") é uma linguagem de programação dinâmica cheia de recursos que quando aplicada em um documento HTML, pode fornecer interatividade dinâmica em sites. Foi inventada por Brendan Eich, co-fundador do projeto Mozilla, da Fundação Mozilla e da Corporação Mozilla.

JavaScript é incrivelmente versátil. Você pode começar pequeno, com carrosséis, galerias de imagens, layouts flutuantes e respostas a cliques de botão. Com mais experiência, você poderá criar jogos, gráficos 2D e 3D animados, aplicativos abrangentes baseados em bancos de dados e muito mais!

JavaScript em si é bastante compacto, mas muito flexível. Os desenvolvedores escreveram uma grande variedade de ferramentas sobre a linguagem JavaScript principal, desbloqueando uma grande quantidade de funcionalidades extras com o mínimo de esforço. Essas incluem:

- Interfaces de programação de aplicativos no navegador (APIs) - APIs integradas em navegadores da Web, fornecendo funcionalidade como criar dinamicamente HTML e definir estilos CSS, coletar e manipular um fluxo de vídeo da webcam do usuário ou gerando gráficos 3D e amostras de áudio.
- APIs de terceiros — permitem que os desenvolvedores incorporem funcionalidades em seus sites de outros provedores de conteúdo, como o Twitter ou o Facebook.
- Estruturas e bibliotecas de terceiros — você pode aplicá-las ao seu HTML para permitir que você crie rapidamente sites e aplicativos.

Como este artigo deve ser apenas uma introdução ao JavaScript, não vamos confundir você neste estágio, falando em detalhes sobre qual é a diferença entre a linguagem JavaScript principal e as diferentes ferramentas listadas acima. Você pode aprender tudo isso em detalhes mais tarde, em nossa área de aprendizado de JavaScript e no restante do MDN.

Um exemplo "Olá Mundo"Seção

A seção acima pode parecer realmente empolgante, e assim deve ser — o JavaScript é uma das tecnologias mais ativas da Web e, à medida que você começa a usá-lo bem, seus sites entrarão em uma nova dimensão de poder e criatividade.

A introdução acima parece ser empolgante, e de fato é — JavaScript é uma tecnologia incrível, e quando você começa a pegar o jeito, seus sites vão entrar em um novo patamar de criatividade e interação.

No entanto, ficar confortável com o JavaScript é um pouco mais difícil do que ficar confortável com HTML e CSS. Você pode ter que começar pequeno e continuar trabalhando em pequenos passos consistentes. Para começar, mostraremos como adicionar alguns JavaScript básicos à sua página, criando um exemplo de "olá mundo!" (o padrão em exemplos básicos de programação).

Importante: Se você não acompanhou o restante de nosso curso, faça o download desse código exemplo e use-o como um ponto de partida.

1. Primeiro, vá para o seu site de teste e crie uma nova pasta chamada scripts. Em seguida, dentro da nova pasta de scripts que você acabou de criar, crie um novo arquivo chamado main.js. Salve na sua pasta de scripts.

2. Em seguida, no seu arquivo index.html, insira o seguinte elemento em uma nova linha logo antes da tag de fechamento </body>:

```
<script src="scripts/main.js"></script>
```

3. Isso é basicamente a mesma coisa que o elemento <link> para o CSS — ele aplica o JavaScript à página, para que ele tenha efeito no HTML (junto com o CSS e qualquer outra coisa na página).

4. Agora adicione o seguinte código no arquivo main.js:

```
5. var myHeading = document.querySelector('h1');
```

```
myHeading.textContent = 'Hello world!';
```

6. Por fim, verifique se os arquivos HTML e Javascript estão salvos e, em seguida, carregue o index.html no navegador. Você deve ver algo do tipo:

Nota: A razão pela qual colocamos o elemento <script> perto da parte inferior do arquivo HTML, é que o HTML é carregado pelo navegador na ordem em que ele aparece no arquivo. Se o JavaScript é carregado primeiro ele pode afetar o HTML abaixo dele, mas as vezes isso pode não funcionar, já que o JavaScript seria carregado antes do HTML em que ele deveria trabalhar. Portanto, colocar o JavaScript próximo à parte inferior da página HTML geralmente é a melhor estratégia.

Seção

Seu texto de título foi alterado para "Hello world!" usando JavaScript. Você fez isso primeiro usando uma função chamada `querySelector()` para obter uma referência ao título e armazená-lo em uma variável chamada `myHeading`. Isso é muito parecido ao que fizemos usando seletores CSS. Quando queremos fazer algo em um elemento, primeiro você precisa selecioná-lo.

Depois disso, você define o valor da propriedade `textContent` para "Hello world!", na variável `myHeading` (que representa o conteúdo do título).

jQuery

Considerando que você queira incorporar o jQuery Core + jQuery UI nas suas páginas web, os trechos representados pela **Listagem 1** precisam ser acrescentados dentro da tag head do seu HTML:

Listagem 1. Importando arquivos do jQuery + UI

```
<!-- CSS jQuery UI -->
<link href="css/jquery-ui.min.css" rel="stylesheet">
<!-- CSS jQuery UI Default Theme -->
<link href="css/jquery-ui.theme.min.css" rel="stylesheet">
<!-- JS jQuery Core -->
<script src="js/jquery-3.1.1.min.js"></script>
<!-- JS jQuery UI -->
<script src="js/jquery-ui.min.js"></script>
```

Os downloads dos referidos arquivos podem ser efetuados nos sites oficiais de cada biblioteca (vide seção **Links**). Feita essa configuração, é possível utilizar qualquer função da biblioteca. Por exemplo, podemos efetuar um teste para visualização da versão do jQuery. Segue o trecho de código em que é possível visualizar a mesma através de um alert do browser:

```
$(document).ready(function() {
    alert("Você está executando a versão do jQuery: " + jQuery.fn.jquery);
});
```

A função `ready()` é amplamente usada pelo jQuery para efetuar ações assim que a página tiver seu carregamento finalizado pelo navegador.

Seletores

Ao desenvolver com JavaScript, uma tarefa bem comum é a leitura e modificação do conteúdo de uma página HTML, algo extremamente custoso de se fazer já que a linguagem disponibiliza apenas alguns

métodos específicos para busca: pelo identificador do elemento (`getElementById`), pelo nome (atributo `name`) do elemento (`getElementsByName`), etc. O jQuery, por outro lado, pode buscar elementos na página através de seu id, classes CSS, tipos, atributos, valores de atributos, entre muitos outros. E tudo que ele faz é encapsular os seletores do CSS (as regras que criamos para instruir ao CSS onde imputar seus estilos) para atingir esse objetivo.

Em vista disso, analisando que teremos de fazer tal tarefa com demasiada frequência, o construtor do jQuery é sempre usado para receber a tal regra (o seletor) e buscar, no objeto do DOM, todos os elementos que atendem à mesma. Você pode instanciar um novo objeto do jQuery usando o seguinte trecho de código: `jQuery()`, ou simplesmente `$()` (a versão reduzida). Os termos `$` e `jQuery`, nesse formato, são reservados para uso exclusivo da biblioteca. Vejamos alguns exemplos:

- Obtendo um elemento pelo id, considerando uma DIV com atributo `id="header"`:

```
$('#header')
```

- Obtendo elementos pela classe, considerando DIVS com atributo `class="header"`:

```
$('.header')
```

É interessante ressaltar que esse tipo de seleção pode retornar tanto um objeto quanto uma lista deles, logo devemos nos preocupar em tratar sempre o retorno em nosso código.

É possível também filtrar pelo seletor mais de uma classe ou tipo de objeto:

```
$('.header', '.footer')
```

Da mesma forma que é possível para elementos do DOM:

```
$('h1', 'p', 'div')
```

Outra possibilidade é obter elementos pelo nome de um de seus atributos e de seu respectivo valor. Poderíamos, por exemplo, obter os elementos cujos atributos `HREF` possuam o valor `"default.html"` por meio do código abaixo:

```
$("[href='default.html']")
```

JavaSE

Começando com a base

Apesar das linguagens de programação serem parecidas, todas elas têm suas peculiaridades, Java não é diferente. Por isso, comecei estudando o básico da linguagem, conhecendo as estruturas condicionais e de loops.

Além disso, estudar orientação a objetos e aprender conceitos como herança, polimorfismo e composição, ajudam bastante no dia a dia. Esses são os conceitos básicos. Mas como já foi dito, cada linguagem é única.

Logo, o próximo passo é conhecer as particularidades da linguagem. Aprender sobre como funcionam as String, as funcionalidades de entrada e saída de dados, as coleções, como listas, mapas e conjuntos, além de aprender sobre as novidades da linguagem são algo de muito valor, já que agrega muito no desenvolvimento.

Legal! Com isso temos uma forte base na linguagem Java, mas será que isso é suficiente para entrar no mercado? Em muitos casos sim, principalmente quando estamos procurando estágios. Porém, se estamos almejando um cargo como júnior, é interessante conhecer mais sobre outros locais do uso da linguagem.

Começando no mundo web

Hoje em dia, muitos serviços funcionam na internet. A web está em muitos lugares, por isso, conhecer esse mundo é importante para quem pensa em ingressar no mercado de trabalho. Logo, se vamos utilizar Java para a web, precisamos conhecer o que a linguagem nos oferece para tal.

E, no Java, a base para aplicações web é conhecida como Servlets. Entender o ciclo da web e como as Servlets trabalham com ele é um passo muito importante.

Praticamente tudo que funciona no mundo Java para web, roda em cima de uma Servlet. Utilizando apenas Servlets, já conseguimos escrever sistemas que podem ser acessados de um navegador, criar páginas dinâmicas além de muitas outras coisas.

Além da web, é muito comum que no dia a dia salvar informações como dados de usuários, informações de acesso, entre muitos outros. Portanto, precisamos de um banco de dados para armazenar as informações. Mas como podemos realizar a comunicação do banco de dados com o Java?

Se comunicando com um banco de dados

As Servlets são uma especificação. Isto é, elas definem um contrato de como uma aplicação que usa Java na web deve funcionar. De uma forma análoga as Servlets, temos uma especificação que é utilizada para realizar a comunicação com bancos de dados, a JPA.

Uma especificação é um contrato. Por isso, precisamos de alguma coisa que coloque o que está neste contrato em prática, ou seja, o implemente.

No caso da JPA, a implementação mais utilizada é o Hibernate. Logo, estudar Hibernate como a implementação da JPA é algo muito valioso para quem quer entrar no mercado. Essas tecnologias são, praticamente, o padrão que a indústria usa no dia a dia quando falamos do mundo Java.

Geralmente, um dos maiores gargalos de uma aplicação é na comunicação como banco de dados.

Logo, conhecer bem o Hibernate e a JPA para realizar otimizações é algo bem valioso, principalmente, quando pensamos em performance da aplicação. Bacana! Já conhecemos Hibernate, Servlets, sabemos muitas coisas sobre a linguagem Java.

Conseguimos escrever aplicações bem legais já com essas tecnologias, mas precisa ser tão trabalhoso? Quando trabalhamos com Servlets puras, realmente é um pouco trabalhoso criar e manter a aplicação.

Claro que pode acontecer de encontrarmos aplicações assim no dia a dia, as chamadas aplicações legado. Mas, no dia a dia, não costumamos trabalhar com Servlets puras.

Utilizamos algo que se comunica com a Servlet, dessa forma, não precisamos trabalhar diretamente com o código da Servlet, mas ainda utilizamos de seu poder. Mas como podemos trabalhar com Servlets, sem escrever Servlets?

Conhecendo o Spring Framework

Escrever código de Servlets é algo comum a muitas aplicações. E, como é algo trabalhoso, foi criada ferramentas para melhorar esse processo.

Uma dessas ferramentas é o Spring. Um conjunto de códigos e projetos, que chamamos de framework, que nos auxiliam no dia a dia do desenvolvimento.

O Spring é um framework muito robusto e utilizado por grandes empresas e tecnologia, como a Netflix. Com ele, conseguimos agilizar muito no processo de escrita e códigos Java. Vimos que para desenvolver em Java, seguimos um passo a passo, mas quando estudamos, as vezes, sentimos falta da prática. Por isso, aqui na Alura, pensamos em criar as formações.

Conhecendo a Formação Java

Uma formação nada mais é do que uma trilha de cursos onde podemos nos tornar proficiente em uma tecnologia. No caso, criamos a Formação Java. Nela, você começará desde o básico com Java.

Conhecendo a linguagem, conhecendo a orientação a objetos, as APIs mais comuns, além de começar com o mundo web, passando por Servlets e Spring, e no mundo de persistência de dados com JPA e Hibernate.

A formação não é formada apenas por cursos, mas sim por diversos outros conteúdos, como podcasts, posts e lives. Cada conteúdo tem uma parte no aprendizado e eles vão se complementando e integrando conforme avançamos na formação. Ao final da formação, você realizará um projeto.

Isto é, irá colocar em prática tudo que aprendeu nos cursos que estudou. Esse projeto será revisado por um instrutor ou instrutora aqui da Alura que te passará um feedback sobre o código que escreveu.

Java EE

As aplicações Web de hoje em dia já possuem regras de negócio bastante complicadas. Codificar essas muitas regras já representam um grande trabalho. Além dessas regras, conhecidas como requisitos funcionais de uma aplicação, existem outros requisitos que precisam ser atingidos através da nossa infraestrutura: persistência em banco de dados, transação, acesso remoto, web services, gerenciamento de threads, gerenciamento de conexões HTTP, cache de objetos, gerenciamento da sessão web, balanceamento de carga, entre outros. São chamados de requisitos não-funcionais.

Se formos também os responsáveis por escrever código que trate desses outros requisitos, teríamos muito mais trabalho a fazer. Tendo isso em vista, a Sun criou uma série de especificações que, quando implementadas, podem ser usadas por desenvolvedores para tirar proveito e reutilizar toda essa infraestrutura já pronta.

O Java EE (Java Enterprise Edition) consiste de uma série de especificações bem detalhadas, dando uma receita de como deve ser implementado um software que faz cada um desses serviços de infraestrutura.

Veremos no decorrer desse curso vários desses serviços e como utilizá-los, focando no ambiente de desenvolvimento web através do Java EE. Veremos também conceitos muito importantes, para depois conceituar termos fundamentais como servidor de aplicação e containers.

Porque a Sun faz isso? A ideia é que você possa criar uma aplicação que utilize esses serviços. Como esses serviços são bem complicados, você não perderá tempo implementando essa parte do sistema. Existem implementações tanto open source quanto pagas, ambas de boa qualidade.

Algum dia, você pode querer trocar essa implementação atual por uma que é mais rápida em determinados pontos, que use menos memória, etc.

Fazendo essa mudança de implementação você não precisará alterar seu software, já que o Java EE é uma especificação muito bem determinada. O que muda é a implementação da especificação: você tem essa liberdade, não está preso a um código e a especificação garante que sua aplicação funcionará com a implementação de outro fabricante. Esse é um atrativo muito grande para grandes empresas e governos, que querem sempre evitar o vendor lock-in: expressão usada quando você está preso sempre nas mãos de um único fabricante.

Servidor de Aplicação

Como vimos, o Java EE é um grande conjunto de especificações. Essas especificações, quando implementadas, vão auxiliar bastante o desenvolvimento da sua aplicação, pois você não precisará se preocupar com grande parte de código de infraestrutura, que demandaria muito trabalho.

Como fazer o "download do Java EE"? O Java EE é apenas um grande PDF, uma especificação, detalhando quais especificações fazem parte deste. Para usarmos o software, é necessário fazer o download de uma implementação dessas especificações.

Existem diversas dessas implementações. Já que esse software tem papel de servir sua aplicação para auxiliá-la com serviços de infraestrutura, esse software ganha o nome de servidor de aplicação. A própria Sun/Oracle desenvolve uma dessas implementações, o Glassfish que é open source e gratuito, porém não é o líder de mercado apesar de ganhar força nos últimos tempos.

Existem diversos servidores de aplicação famosos compatíveis com a especificação do J2EE 1.4, Java EE 5 e alguns já do Java EE 6. O JBoss é um dos líderes do mercado e tem a vantagem de ser gratuito e open source. Alguns softwares implementam apenas uma parte dessas especificações do Java EE, como o Apache Tomcat, que só implementa JSP e Servlets (como dissemos, duas das principais especificações), portanto não é totalmente correto chamá-lo de servidor de aplicação. A partir do Java EE 6, existe o termo "application server web profile", para poder se referenciar a servidores que não oferecem tudo, mas um grupo menor de especificações, consideradas essenciais para o desenvolvimento web.

Servlet Container

O Java EE possui várias especificações, entre elas, algumas específicas para lidar com o desenvolvimento de uma aplicação Web:

- Servlet
- JSP
- JSTL
- JSF

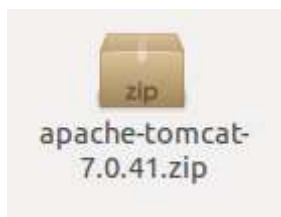
Um Servlet Container é um servidor que suporta essas funcionalidades, mas não necessariamente o Java EE Web Profile nem o Java EE completo. É indicado a quem não precisa de tudo do Java EE e está interessado apenas na parte web (boa parte das aplicações de médio porte encaixam-se nessa categoria).

Há alguns servlet containers famosos no mercado. O mais famoso é o Apache Tomcat, mas há outros como o Jetty, que nós da Caelum usamos muito em projetos e o Google usa em seu cloud Google App Engine.

Exercícios: Preparando o Tomcat

Para preparar o Tomcat na Caelum, siga os seguintes passos:

1. Entre na pasta 21 do Desktop e selecione o arquivo do apache-tomcat-7.x;



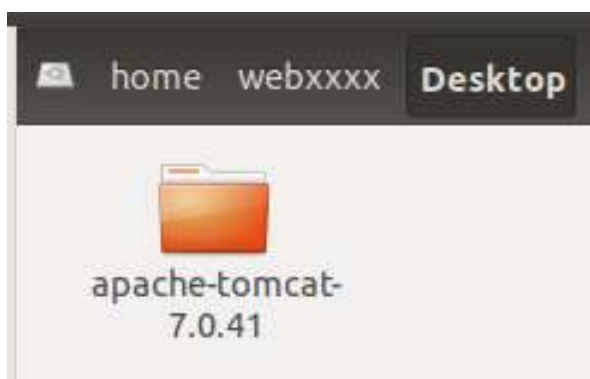
2. Dê dois cliques para abrir o Archive Manager do Linux;



3. Clique em Extract;
4. Escolha o seu Desktop e clique em extract;



5. O resultado é uma pasta chamada apache-tomcat-7.x. Pronto, o tomcat já está instalado.



Preparando o Tomcat em casa

Baixe o Tomcat 7 em <http://tomcat.apache.org/> na página de downloads da versão que escolher, você precisa de uma "Binary Distribution". Mesmo no windows, dê preferência a versão .zip, para você entender melhor o processo de inicialização do servidor. A versão executável é apenas um wrapper para executar a JVM, já que o Tomcat é 100% Java.

O Tomcat foi por muito tempo considerado implementação padrão e referência das novas versões da API de servlets. Ele também é o servlet container padrão utilizado pelo JBoss. Ele continua em primeira posição no mercado, mas hoje tem esse lugar disputado pelo Jetty e pelo Grizzly (esse último é o servlet container que faz parte do servidor de aplicação da Oracle/Sun, o Glassfish).

Entre no diretório de instalação e execute o script startup.sh:

```
cd apache-tomcat<TAB>/bin  
./startup.sh
```

Entre no diretório de instalação do tomcat e rode o programa shutdown.sh:

```
cd apache-tomcat<TAB>/bin  
./shutdown.sh
```

Aprenderemos futuramente como iniciar o container de dentro do próprio Eclipse, por comodidade e para facilitar o uso do debug.

REFERÊNCIAS

Os links citados abaixo servem apenas como referência. Nos termos da lei brasileira (lei no 9.610/98, art. 8o), não possuem proteção de direitos de autor: As ideias, procedimentos normativos, sistemas, métodos, projetos ou conceitos matemáticos como tais; Os esquemas, planos ou regras para realizar atos mentais, jogos ou negócios; Os formulários em branco para serem preenchidos por qualquer tipo de informação, científica ou não, e suas instruções; Os textos de tratados ou convenções, leis, decretos, regulamentos, decisões judiciais e demais atos oficiais; As informações de uso comum tais como calendários, agendas, cadastros ou legendas; Os nomes e títulos isolados; O aproveitamento industrial ou comercial das ideias contidas nas obras.

Caso não concorde com algum item do material entre em contato com a Domina Concursos para que seja feita uma análise e retificação se necessário

A Domina Concursos não possui vínculo com nenhuma banca de concursos, muito menos garante a vaga ou inscrição do candidato em concurso. O material é apenas um preparatório, é de responsabilidade do candidato estar atento aos prazos dos concursos.

A Domina Concursos reserva-se o direito de efetuar apenas uma devolução parcial do conteúdo, tendo em vista que as apostilas são digitais, isso, [e, não há como efetuar devolução do material.

A Domina Concursos se preocupa com a qualidade do material, por isso todo conteúdo é revisado por profissionais especializados antes de ser publicado.



Prezado cliente,

É com imensa satisfação que expressamos nossa profunda gratidão pela sua escolha em adquirir suas apostilas de estudos conosco. A preferência pelo nosso serviço é motivo de grande alegria e reforça nosso compromisso em fornecer materiais de alta qualidade para contribuir efetivamente em seu caminho educacional.

Aqui na nossa loja, dedicamo-nos diariamente para oferecer produtos que atendam não apenas às suas necessidades de aprendizado, mas que também superem suas expectativas. Cada compra realizada é um voto de confiança em nossa equipe, e estamos comprometidos em corresponder a essa confiança através de excelência em produtos e atendimento.

Saiba que sua decisão de confiar em nós para sua jornada de estudos é valorizada e respeitada. Estamos sempre empenhados em aprimorar nossos serviços para garantir que sua experiência seja positiva e produtiva. Se houver algo específico que possamos fazer para melhor atendê-lo, por favor, não hesite em nos informar.

Agradecemos por fazer parte da nossa comunidade de clientes e por escolher a qualidade e confiabilidade das nossas apostilas. Estamos ansiosos para continuar a servi-lo com dedicação e comprometimento.

Atenciosamente, Domina Concursos.



contato@dominaconcursos.com.br



WhatsApp (48) 9.9695-9070



Rua Aracatuba, nº 45,
Centro, Criciúma/SC - CEP
88810-230