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 comeventuais erros envolvidos com semântica das instruções, erros esses detectados na análise semântica dpiladores.

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 minha Conta. 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 assícronos, 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 o 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 determinada 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 nas 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 se acessar seus dados de modo mais eficiente.

Métodos De Ordenação De Vetoresb

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 ≥ 0, x1, x2, ..., xn, organizados de forma a refletir a posição relativa dos mesmos. Se N ≥ 0, então x1 é o primeiro nó. Para 1 < k < n, o nó xk é precedido pelo nó xk-1 e seguido pelo nó xk+1 e xn 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;

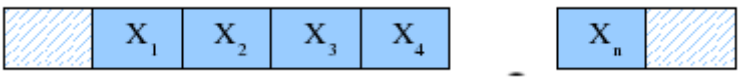
Cartas de baralho.

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.

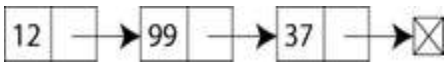
Filas

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.

Deques

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.

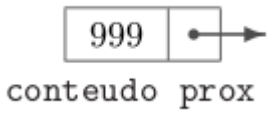
Cada elemento numa lista possui um índice, um número que identifica cada elemento da lista. Usando o índice de um elemento da lista é possível buscá-lo ou removê-lo.

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, boleanos etc... */

 int chave:

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

 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 = {s1, s2, ..., sn} um conjunto de chaves tais que s1 < s2 ... sn. Seja k um valor dados. Deseja-

se verificar se k {\displaystyle \in } S e identificar o índice i tal que k = si.

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 sj {\displaystyle \in } S e tem como rótulo o valor r(u) = sj.

Para cada nó v de T r(v1) < r(v) e r(v2) > r(v), onde v1 pertence à subárvore esquerda de v e v2 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.

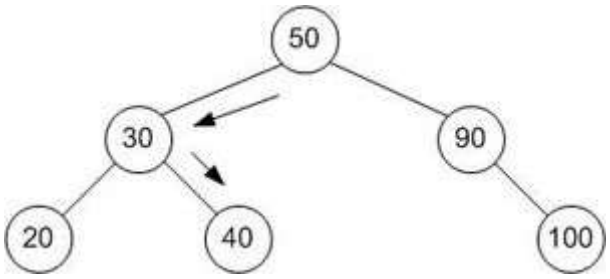
Uma árvore binária de busca com chaves aleatórias uniformemente distribuídas tem altura O(log n).

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 ziig-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))
```

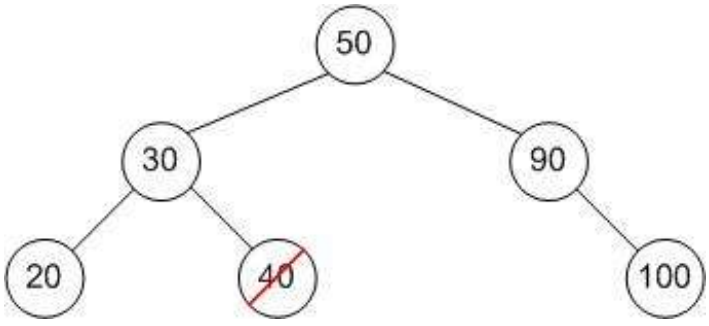
Esta operação requer O (log n) vezes para o caso médio e necessita de O (n) no pior caso. A fim de introduzir um nó novo na árvore, seu valor é primeiro comparado com o valor da raiz. Se seu valor for menor que a raiz, é comparado então com o valor do filho da esquerda da raiz. Se seu valor for maior, 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:

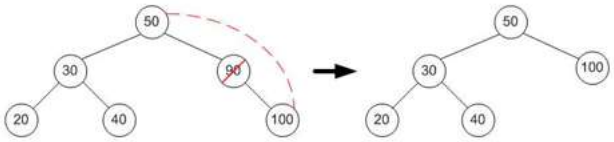
Remoção Na Folha

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



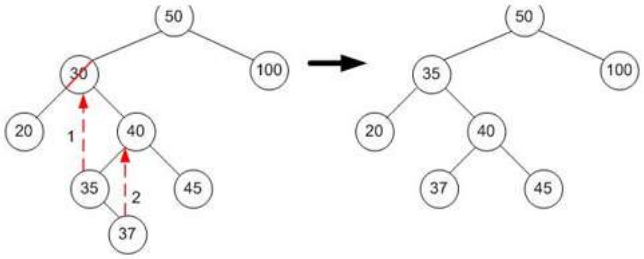
Remoção De Nó Com Um Filho

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

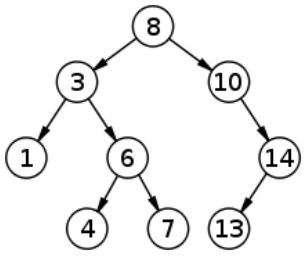
Percorre a subárvore direita em ordem simétrica

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(n2) no pior caso. A geração de um vetor de chaves ordenadas tem complexidade O(n). O algoritmo de ordenação terá complexidade final O(n2) 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		
Algoritimo	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 {\displaystyle O(\logn)} (no qual {\displaystyle 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.

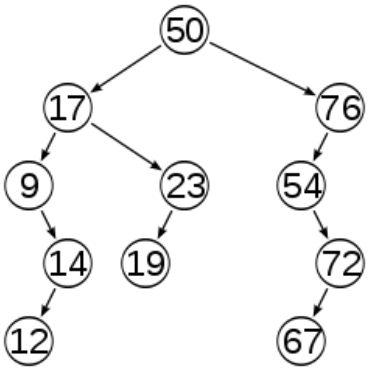
No presente artigo serão apresentados: os conceitos básicos, incluindo uma proposta de estrutura; apresentação das operações busca, inserção e remoção, todas com complexidade {\displaystyle O(\log n)}.

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.log (n) operações, onde n é o número de elementos contido na árvore. Tal estrutura foi a primeira árvore binária balanceada criada.

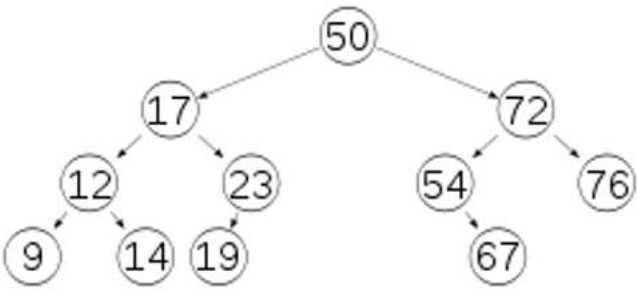
Conceitos Básicos

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)| ≤ 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 o 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 {\displaystyle \varphi } 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->chave (valor chave armazenado no nó u), então a busca foi bem sucedida. Caso contrário, se K < pt_u->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->chave então retornar Verdadeiro;

senão se K < pt_u->chave então

 retornar busca_AVL(K, u->esq);

 senão retornar busca_AVL(K, u->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 Tp passa de altura 0 para altura 1, então Tp mudou de altura.

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

Caso 3: fb=1 ou -1 antes da inserção, passou a ter valor 0, então a subárvore Tp 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 Tp 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) {
        BTNode<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);
        } else {
                        this.root = (BSTNode<T>) noDireito;
protected void rightRotation(BSTNode<T> no) {
        BTNode<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().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 inicio // tem apenas uma subárvore
se p^.esq<>Nulo então
inicio
p^.chave := p^.esq^.chave;
delete p^.esq;
p^.esq := Nulo;
```

```
 fim
 senão inicio
 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
inicio
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 Auxilares 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 = no.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".

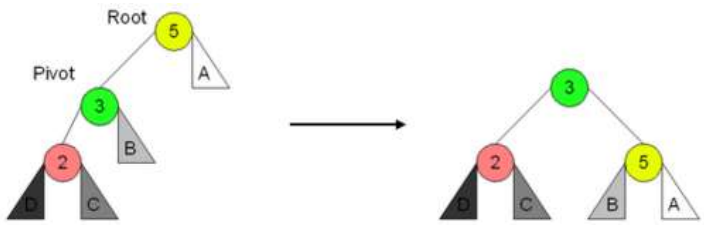
Para garantirmos as propriedades da árvore AVL rotações devem ser feitas conforme necessário após operações de remoção ou inserção. Seja P o nó pai, FE o filho da esquerda de P e FD o filho da direita de P podemos definir 4 tipos diferentes de rotação:

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 Pdeve 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

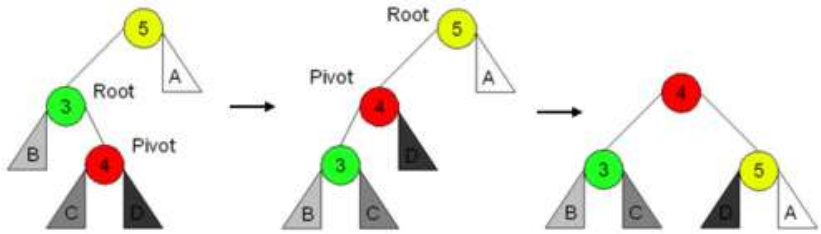
Torne o filho à esquerda de Y o filho à direita de X.

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.

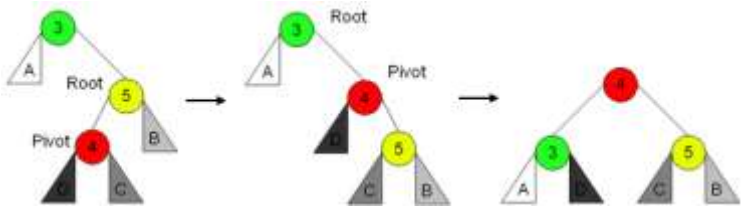


Caso 1.2 - Rotação dupla à direita

Deve ser observado que as 3 possíveis combinações de alturas da subárvores T2 e T3 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 := c^.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;
```

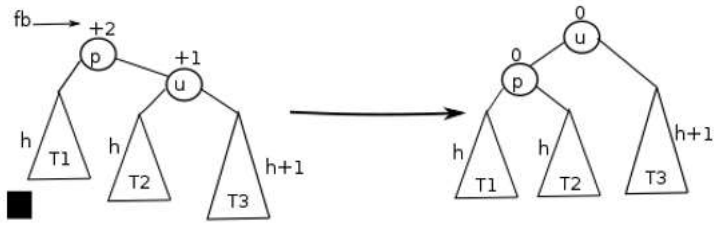
Cabe ressaltar, apesar de não estar relatado na literatura, que na remoção poderá ocorre a situação na qual u^.fb=0. Neste caso deverá ser aplicada rotação simples e o fator de balanço dos nós u e p serão respectivamente 1 e -1.

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.

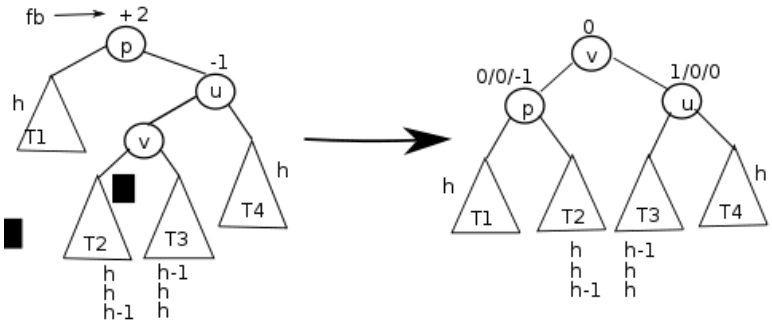
Para identificar se qual rotação aplicar, bastará analisar o fator de balanço do nó u, raiz da subárvore direita do nó p.

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);

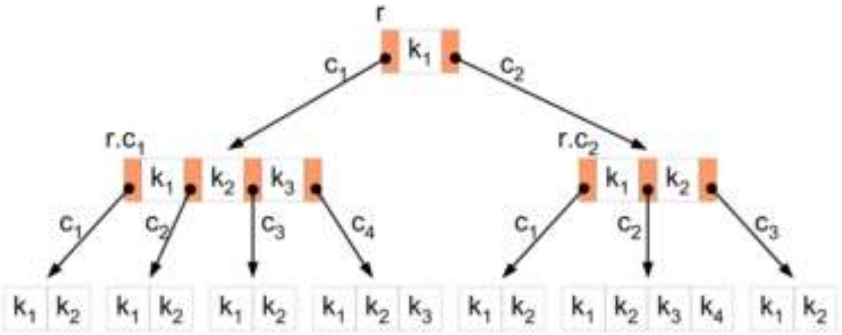
Pertence - O(log n);

União - O(n.log n);

Interseção - O(n.log n).

Árvore B

Árvore B		
Tipo	Árvore	
Ano	1971	
Inventado por	Rudolf Bayer, Edward Meyers McCreight	
Complexidade de Tempo em Notação big O		
Algoritimo	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.

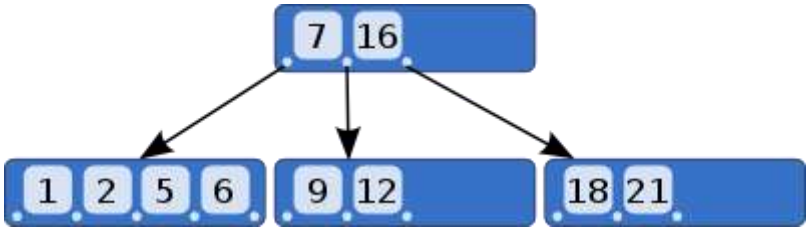
As árvores B são semelhantes as árvores preto e vermelho, mas são melhores para minimizar operações de E/S de disco. Muitos sistemas de bancos de dados usam árvores B ou variações da mesma para armazenar informações. Dentre suas propriedades ela permite a inserção, remoção e busca de chaves numa complexidade 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{\displaystyle \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 contém chaves no nível mais baixo da árvore. Já Knuth define o termo como as páginas que estão abaixo do ultimo nível da árvore, ou seja, páginas que não contém 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 ⌈d/2⌉ 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 contem k-1 chaves

Uma página folha contém pelo menos ⌈d/2⌉-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 ⌈d/2⌉-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 (⌈d/2⌉-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.

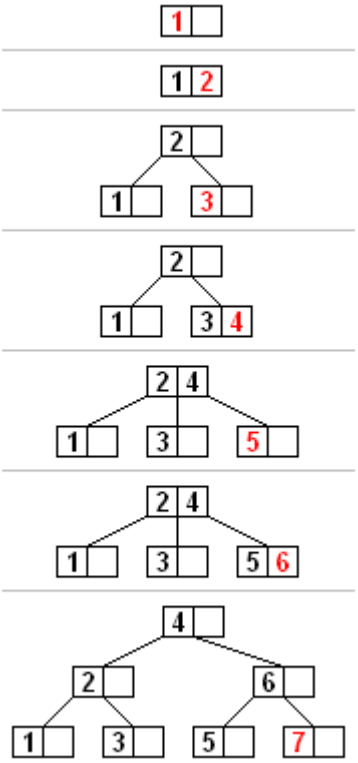
Estrutura Da Página

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 intermediario 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 irmão 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 (⌈d/2⌉-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:

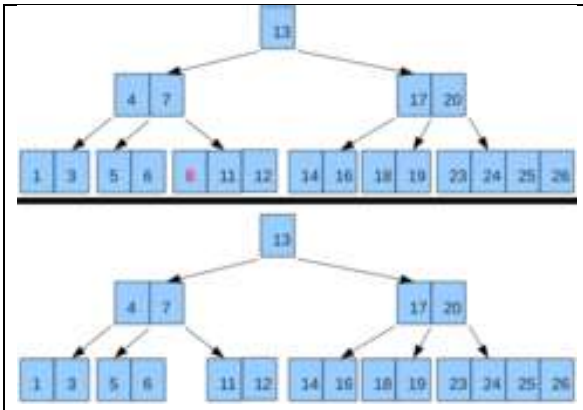


Figura 1: Remoção da chave 8 e posterior reorganização da estrutura

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.

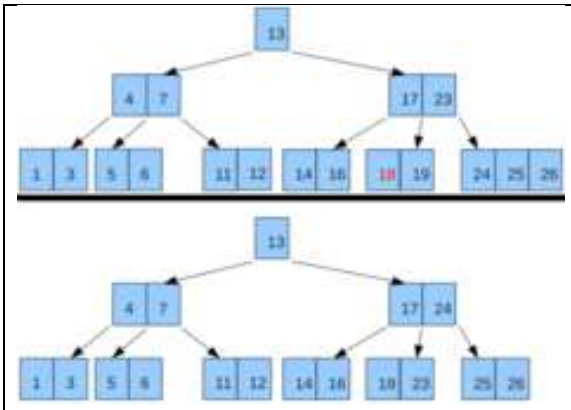


Figura 2: Remoção da chave 18 e posterior redistribuição das chaves

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.

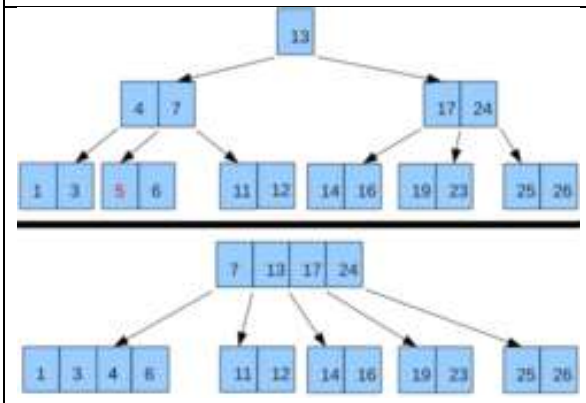


Figura 3: Remoção da chave 5 e posterior concatenação com página irmã à esquerda

Caso da figura 3: Nesta figura foi removido a chave 5, como não foi possivel 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.

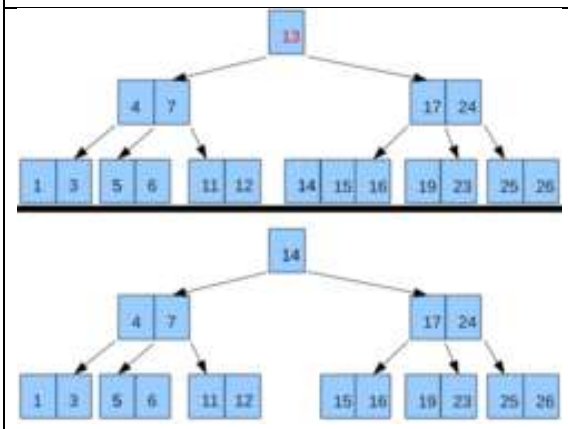


Figura 4: Remoção da chave 13 e promoção da menor chave da subárvore à direita de 13

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.

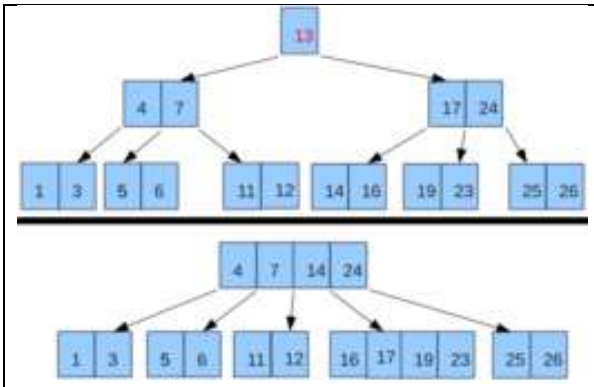


Figura 5: Chave 14 é promovida para a raiz o que causa underflow em sua página

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 possivel 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.

```
Inserçao(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 = Inserçao(posicao, key, chavePromovida)//procedimento recursivo
se(flagRetorno indica que nao houve promoçao 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 promoçao 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

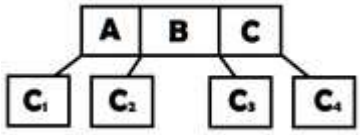
Algoritmo split em Java

```
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(mediana);
if (parent.size() >= maxChildren) {
parent.split();
}
}
protected void promote(int mid) {
T element = elements.get(mid);
this.parent.addElement(element);
```

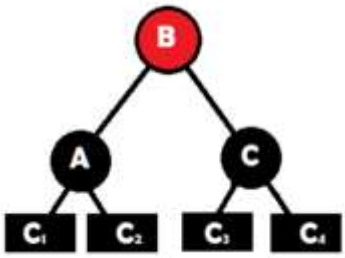
```
 }
Imprimir em Ordem em C
void emOrdem (tpaginaB raiz) {
 if(raiz==NULL)
 return;
 for(int i=0;i<raiz.n,i++)
 emOrdem(raiz→pont[i]);
 printf("%i",raiz→chv[i]);
 }
 emOrdem(raiz→pont[raiz.n]);
}
Algoritmo Split Dentro Da Classe Node Em Java
protected void split() {
                T mediana = this.getElementAt(elements.size() / 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, consequentemente, 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, consequentemente, 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 ligálo 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.

As árvores B+ possuem seus dados armazenados somente em seus nós folha e, seus nós internos e raiz, são apenas referências para as chaves que estão em nós folha. Assim é possivel manter ponteiros em seus nós folha para um acesso sequencial ordenado das chaves contidas no arquivo.

Árvores B* diferem das árvores B em relação ao particionamento de suas páginas. A estratégia dessa variação é realizar o particionamento de duas páginas irmãs somente quando estas estiverem completamente cheias e, claro, isso somente é possível através da redistribuição de chaves entre estas páginas filhas. Estando completamente cheias, as chaves são redistribuídas entre três páginas diferentes que são as duas irmãs anteriores e uma nova criada.

Comparação Com as Variações

Se compararmos as árvores B com suas variações podemos enumerar algumas características importantes para a escolha de implementação destas:

Árvores B+:

A principal característica proporcionada por esta variação é o fato de permitir o acesso sequencial das chaves por meio de seu sequence set de maneira mais eficiente do que o realizado em árvores B .

Além do mais, as páginas utilizadas em seu index set podem conter mais apontadores para páginas filha permitindo reduzir a altura da árvore.

Árvores B*:

A principal vantagem decorrente dessa variação é o fato desta apresentar suas páginas com no mínimo 2/3 do número máximo de chaves, ou seja, esta variação apresenta no pior caso um índice de utilização do arquivo de 66%, enquanto em árvores B esse índice de pior caso cai para 50%.

Em Sistema De Arquivos

Além de sua utilização em bancos de dados, uma Árvore B (ou uma variante) também é usada em sistemas de arquivos para permitir acesso aleatório rápido a um bloco arbitrário em um arquivo particular. O problema básico consiste em transformar o bloco de arquivos {\displaystyle i} em um endereço de bloco de disco (ou talvez para um cilindro-cabeça-sector).

O sistema de arquivos da Apple HFS+, NTFS da Microsoft, AIX (JFS2) e alguns sistemas de arquivos Linux, como btrfs e Ext4, usam árvores B.

Árvores B* são usadas nos sistemas de arquivos HFS e Reiser4.