Árvores B e B+
Pesquisa Teórica

I Introdução a Árvores B e B+

I.I A Árvore B

Há mais de 50 anos, em 1971, seria inventada a estrutura de dados revolucionária que viria a se tornar a mais utilizada em sistemas de bancos de dados pelo mundo. Criada por Rudolf Bayer e Edward M. McCreight, dois cientistas dos Laboratórios de Pesquisa da Boeing, a árvore B mudou a forma como se armazenava dados, reduzindo enormemente os custos de processamento em pesquisas, deleções e inserções de dados em bancos.
Antes da invenção da árvore B, os dados eram organizados em tabelas de índice, onde quanto mais dados fossem guardados, mais "alta" seria a tabela. O problema dessa forma de organização é que, para uma grande quantidade de dados - ou seja, para uma tabela mais "alta" - eram necessários muitos acessos à memória até achar a informação desejada. Uma forma de resolver isso seria criar uma estrutura com dados ligados a intervalos de chaves que direcionassem a busca de modo mais eficiente.
Assim surgiu a árvore B, uma árvore m-ária onde cada nó consiste em chaves de índice com ponteiros em seus intervalos apontando para blocos de dados, podendo estes ser outros nós internos ou folhas. Foi essa estrutura tão única e inovadora que reduziu consideravelmente o número de acessos à memória.

I.II Uma alternativa ainda melhor

Com a popularização do uso das árvores B em bancos de dados, estudos foram realizados sobre como aumentar ainda mais sua eficiência. Não existe um registro formal sobre como, ou através de quem, a árvore B+ surgiu, só se sabe que seu primeiro uso foi no Virtual Storage Access Method (VSAM) da IBM, além de ser citada em um artigo de 1973, publicado pela mesma empresa. Mas afinal, qual a diferença entre a árvore B e a B+?
Assim como sua sucessora, a árvore B+ é uma árvore m-ária com chaves e ponteiros em seus intervalos apontando para blocos de dados. Porém, diferentemente da árvore B, a árvore B+ encara as chaves de seus nós internos apenas como valores de referência, não como dados a serem acessados. Por causa disso, a árvore B+ precisa de cópias dessas chaves nas suas folhas, que são encaradas como dados válidos para o acesso. Esse comportamento é muito bem utilizado em contextos de armazenamento em blocos, como sistemas de arquivos.

II Estrutura e Funcionamento de Árvores B

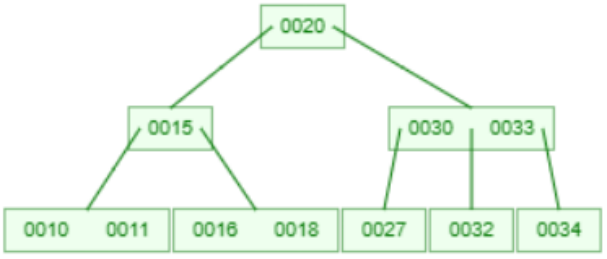
Como dito anteriormente, árvores B são árvores m-árias onde cada nó consiste em chaves de índice com ponteiros em seus intervalos apontando para blocos de dados, podendo estes ser outros nós internos ou folhas. Esta descrição resumida pode dar uma boa base sobre o

funcionamento dessa estrutura, mas é preciso aprofundar-se para entender de fato o porquê desta ser uma das estruturas mais utilizadas na programação.
Antes de analisar em detalhes as operações da árvore B, porém, é preciso ter em mente as seguintes propriedades da estrutura:

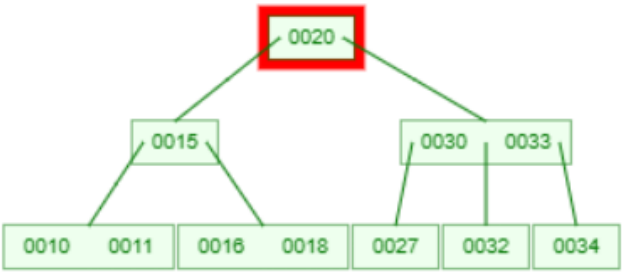
- **Todo nó *x* tem:**
  - *x.n*: refere-se ao número de chaves armazenadas no nó.
  - *x.chave[i]*: é a i-ésima chave armazenada, sendo $x.chave[1] < x.chave[2] < ... < x.chave[x.\ n]$.
  - *x.folha*: indica se *x* é uma folha ou não.
- **Grau mínimo (constante *t*):**
  - Todo nó, exceto a raiz, precisa ter pelo menos $t-1$ chaves. Ou seja, cada nó interno tem pelo menos *t* filhos. A raiz, por outro lado, pode ter apenas uma chave.
  - Todo nó tem no máximo $2t-1$ chaves. Ou seja, cada nó interno tem no máximo *2t* filhos.

II.I Busca na Árvore B

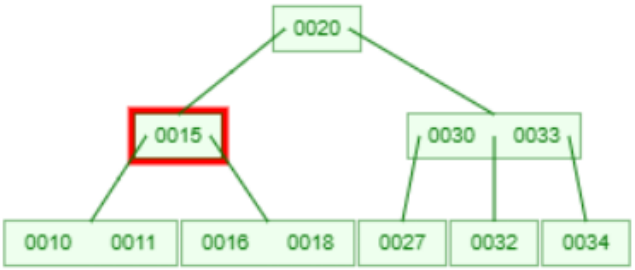
É possível comparar a busca na árvore B com a busca em uma árvore binária - claro, já que essa foi a base para sua criação - onde para cada nó analisado, caminha-se para os filhos da esquerda ou da direita. A grande diferença entre as estruturas surge quando vê-se que, ao invés de cada nó armazenar apenas uma informação e levar a outros dois nós filhos, é possível colocar *n* (igual a no mínimo *t-1* e no máximo *2t-1*) valores em um mesmo nó, que levam a *n+1* (igual a no mínimo *t* e no máximo *2t*) nós filhos, também com no mínimo *t-1* e no máximo *2t-1* valores.
A busca começa sempre pela raiz, onde é realizada uma busca binária para tentar achar o valor desejado. Caso ele seja encontrado, a operação termina, caso contrário, caminha-se para o nó filho apontado pelo intervalo de valores em que o valor desejado pertence. O processo então se repete até encontrar o valor.
Por exemplo, ao tentar encontrar o valor 11 na árvore B com $t = 2$ a seguir, o processo de busca será:



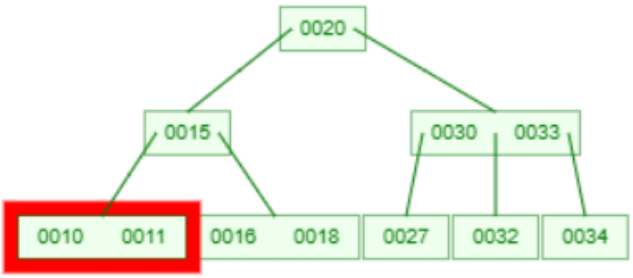
1. Primeiro, verifica-se a raiz. Caso o valor seja encontrado, a operação termina, caso contrário, parte-se para o próximo passo.



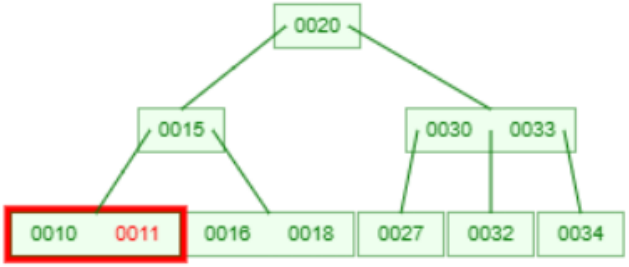
2. $20 \neq 11$, portanto o próximo passo é caminhar até o nó filho apontado pelo intervalo de valores em que o valor desejado pertence, nesse caso $11 < 20$, portanto percorre-se para o nó à esquerda.



3. Novamente, verifica-se se o valor desejado pertence ao nó atual. Como $15 \neq 11$, repete-se o passo anterior, movendo-se para o nó à esquerda ($11 < 15$).



4. Finalmente, realiza-se uma busca binária para verificar se 11 pertence ao nó atual, o que é verdadeiro, encerrando o processo.



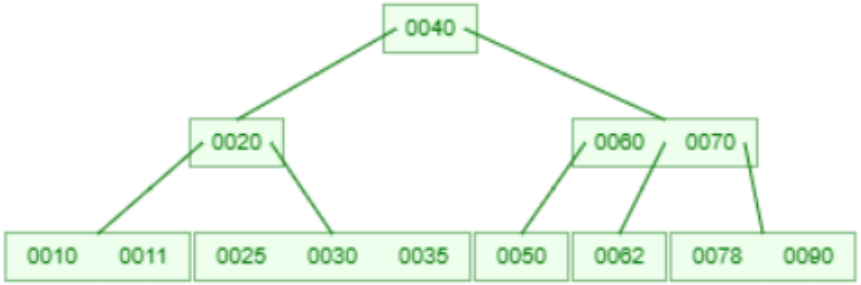
Vale destacar que, graças à propriedade de auto balanceamento da árvore B e o uso da busca binária dentro dos nós, é garantida uma complexidade de O(log n) em todos os casos de busca da estrutura, independente de seu tamanho ou grau mínimo.

II.II Inserção na Árvore B

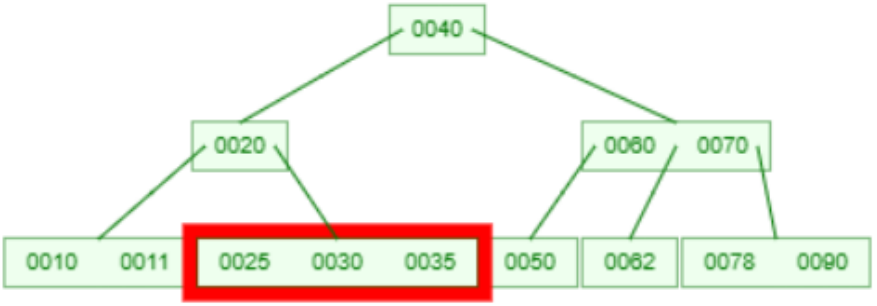
Assim como a busca, a inserção na árvore B tem certa semelhança com a inserção da árvore binária. Nas duas estruturas, começa-se pelo básico: uma operação de busca para saber onde adicionar o novo valor. Feito isso, insere-se o elemento em ordem no nó correto e pronto, certo? Na árvore binária, provavelmente, mas na árvore B é necessário levar em conta seu grau mínimo.

Como visto antes, o grau mínimo da árvore B define quantos elementos e quantos filhos um nó pode ter. Portanto, se um elemento for adicionado a um nó "cheio", o balanço estabelecido pelo grau mínimo seria perturbado, criando problemas na estrutura. Para resolver essa questão, foi criada a função de **split**, que, caso um nó fique com mais de *2t-1* elementos na inserção, divide ele em dois. Vejamos um exemplo para entender melhor esse conceito.

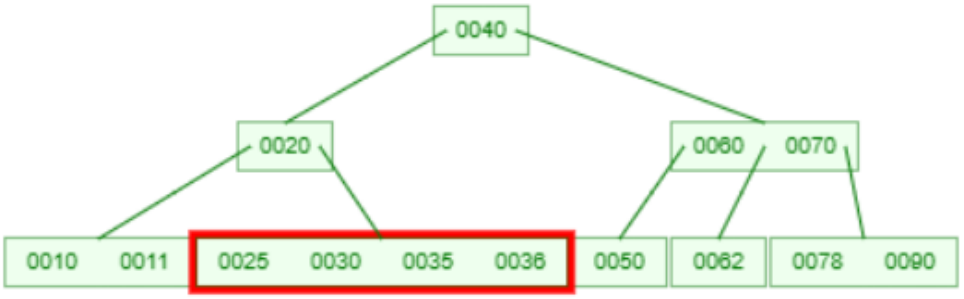
Dada a árvore B a seguir com $t = 2$, vamos inserir o valor 36:



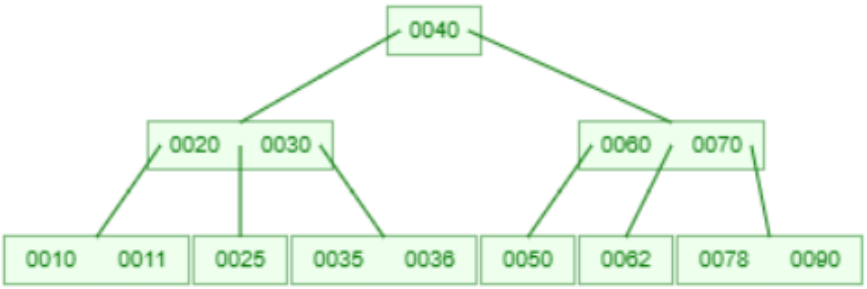
1. Primeiro realiza-se uma função de busca para saber onde adicionar o novo elemento.



2. Encontrado o nó correto, insere-se o valor.



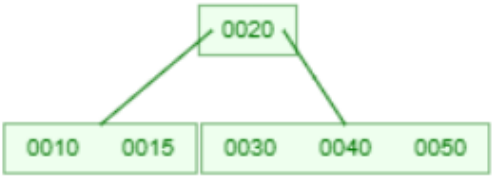
3. Como $t = 2$, o máximo de chaves contidas em um mesmo nó deve ser $2t-1 = 3$. Como o nó agora apresenta 4 valores, realiza-se a operação de split.



Note que, com o split realizado, o valor médio do nó cheio “sobe” para o nó pai e outro filho é criado para armazenar o(s) elemento(s) menores que o valor médio, enquanto os maiores continuam no nó pré-existente. Caso o pai também ficasse com mais de *2t-1* valores após o split, o processo seria repetido nos níveis superiores até não ser mais necessário.

II.III Remoção na Árvore B

Na remoção, também é preciso obedecer às regras estabelecidas pelo grau mínimo, nesse caso a que estabelece que cada nó não raiz precisa ter no mínimo *t-1* valores. Por causa disso, existem alguns casos de exclusão, dependendo do tipo de nó em que a chave a ser excluída está e a quantidade de chaves no nó.
O primeiro caso acontece quando a chave a ser removida está em um nó folha com mais do que a quantidade mínima de valores. Aqui, como o nó tem mais do que a quantidade mínima de chaves, é possível apenas remover o valor sem a necessidade de ajustes.
O segundo caso ocorre também quando o nó é folha, mas dessa vez quando a quantidade de chaves presentes nele é igual a quantidade mínima. Nesse contexto, não se pode apenas remover a chave escolhida, já que isso feriria a lei estabelecida pelo grau mínimo. Deve-se, então, após a remoção, ajustar o nó através de um empréstimo (rotação) - tentar “subir” o valor mais próximo de um nó irmão que tenha mais do que a quantidade mínima e descer uma chave do nó pai - ou de um merge - combinar os irmãos, caso eles também tenham a quantidade mínima de nós. Para entender melhor esses conceitos, vejamos um exemplo:
Dada a árvore B+ com grau mínimo $t = 3$ (ou seja, o mínimo de chaves é $t-1 = 2$), vamos excluir a chave 15:



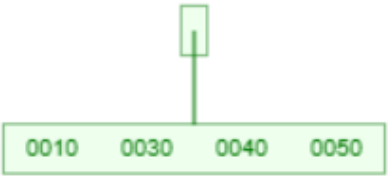
1. Primeiro, realiza-se uma função de busca para achar a chave.

2. Feito isso, exclui-se o valor em questão e realiza-se o ajuste. Como o nó irmão possui mais de 2 chaves, utiliza-se a lei do empréstimo, passando o 30 para o pai e descendo o 20.

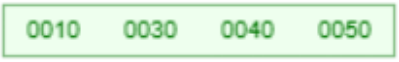
Agora, a partir da árvore resultante, vamos tentar remover o valor 20:

1. Primeiro, realiza-se a função de busca.

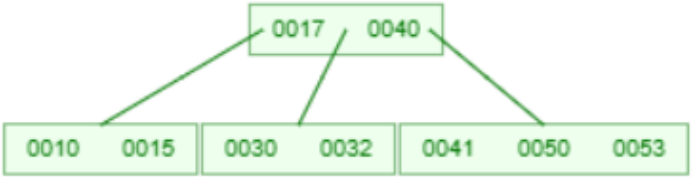
2. Depois, a exclusão e os ajustes são feitos. Note que dessa vez o nó irmão tem exatamente 2 chaves, que é o valor mínimo. Por isso, utiliza-se o merge para agrupar as duas folhas. É importante perceber também que, nesse exemplo, o pai dos nós apresenta apenas uma chave, portanto ele também participa da combinação.



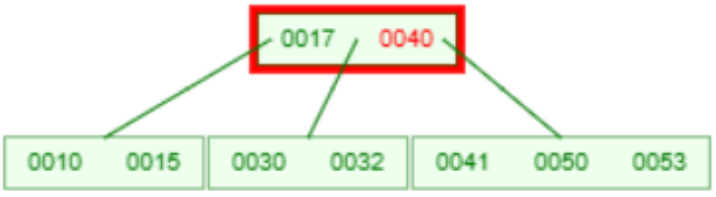
3. Finalmente, ajusta-se a raiz, encerrando o processo de exclusão.



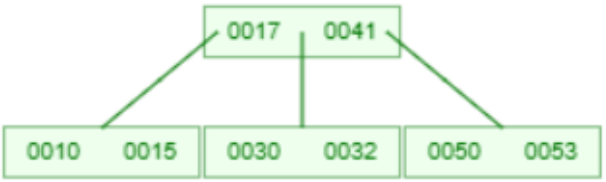
O terceiro caso de exclusão ocorre quando a chave faz parte de um nó interno. Aqui, é sempre necessário substituir a chave excluída, existindo dois caminhos possíveis: substituir pelo predecessor - maior chave na subárvore esquerda - ou pelo sucessor - menor chave na subárvore direita. Vejamos um exemplo:
Dada a árvore B com grau mínimo $t = 3$, vamos excluir a chave 40:



1. Como sempre, primeiro realiza-se uma função de busca.



2. Feito isso, exclui-se a chave e realiza-se a troca pelo sucessor.



Em resumo, temos os seguintes casos na remoção de árvores B:

I. Remoção de chave de nó folha com mais que o mínimo:
   - Remova a chave diretamente.
   - Nenhuma reestruturação necessária.

II. Remoção de chave de nó folha com o mínimo:
   - Tente emprestar uma chave de um irmão.
   - Se não for possível, combine com um irmão e ajuste o nó pai.

III. Remoção de chave de nó interno:
   - Substitua a chave pelo predecessor ou sucessor.
   - Remova a chave substituta do nó folha correspondente.
   - Ajuste a árvore conforme necessário.

III Estrutura e Funcionamento de Árvores B+

Como descrito pelo seu próprio nome, árvores B+ têm uma grande semelhança com árvores B. Ambas são árvores m-árias onde cada nó consiste em chaves de índice com ponteiros em seus intervalos apontando para blocos de dados. A grande diferença entre elas surge porque a árvore B+ encara as chaves de seus nós internos apenas como valores de referência, não como dados a serem acessados, sendo necessário uma cópia desses valores nas folhas - folhas essas que são todas ligadas através de uma lista circular. Graças a essas propriedades, o funcionamento da árvore B+ apresenta diferenças significativas com relação à árvore B, embora ainda tenha muitas semelhanças.

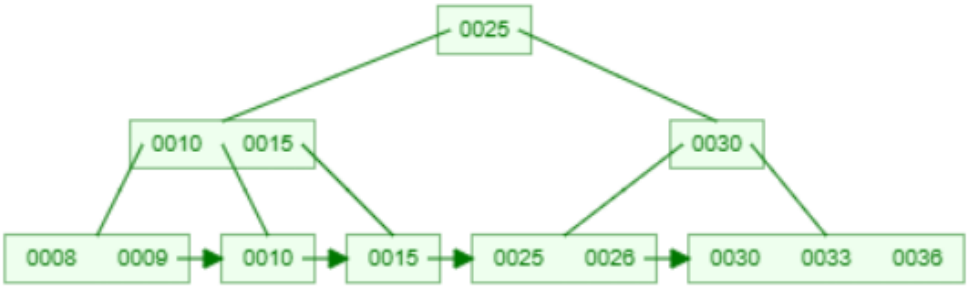
III.I Busca na Árvore B+

A busca em uma árvore B+ segue a mesma lógica inicial da busca em uma árvore B, mas sempre desce até um nó folha, mesmo que a chave seja encontrada em um nó interno, já que os dados de fato estão todos agrupados nas folhas.
O processo, então, é o seguinte: assim como na árvore B, a busca desce a árvore seguindo os nós internos, porém não para se achar a chave desejada, ao invés disso, continua até alcançar um nó folha. Ao chegar em um nó folha, realiza uma busca binária com o intuito de encontrar o valor de fato, encerrando a busca.

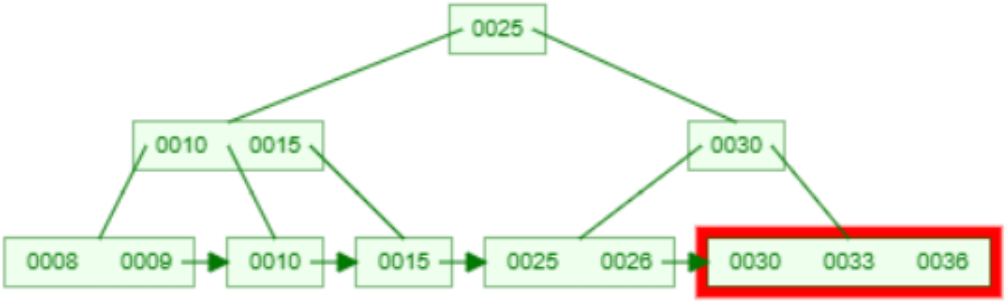
III.II Inserção na Árvore B+

Assim como a busca, a inserção na árvore B+ tem muita semelhança com a da árvore B. Aqui, as regras estabelecidas pelo grau mínimo também se aplicam, então se um elemento for adicionado a um nó cheio, realiza-se uma função de split para manter o balanceamento, repetindo o processo com os nós pais enquanto for necessário. Uma diferença marcante da inserção na árvore B+ é que em casos de divisão de nós folha, a nova chave promovida no split permanece no nó folha, mesmo que apareça no nó pai interno. Vejamos isso com um exemplo.

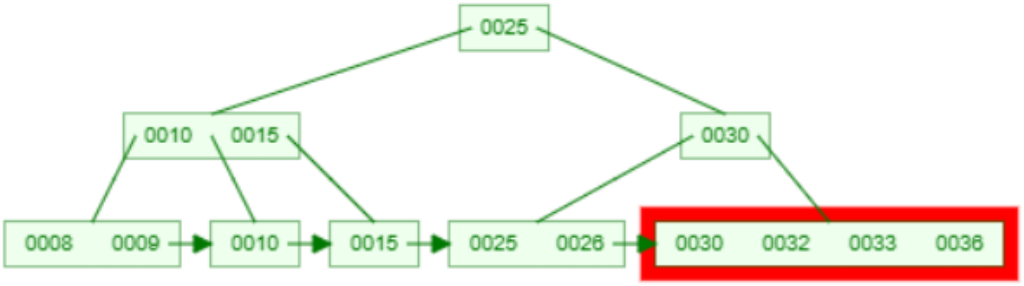
Seja a árvore B+ com grau mínimo $t = 2$ (ou seja, o máximo de chaves é $2t-1 = 3$) a seguir, vamos inserir a chave 32:



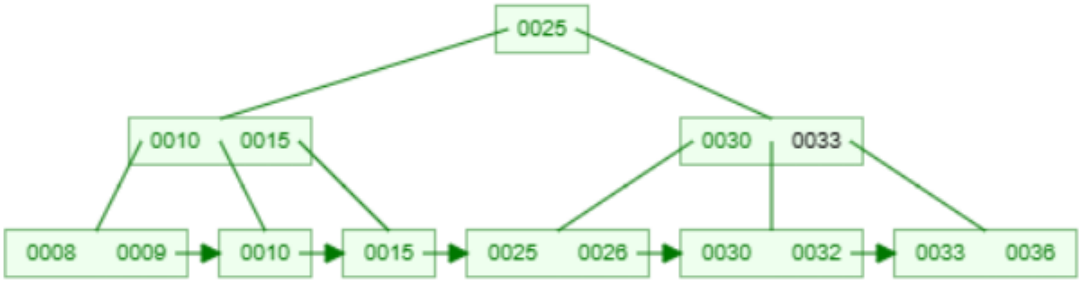
1. Primeiro, realiza-se uma função de busca até as folhas, seguindo os intervalos determinados pelos nós internos.



2. Depois, adiciona-se normalmente o valor no nó folha.



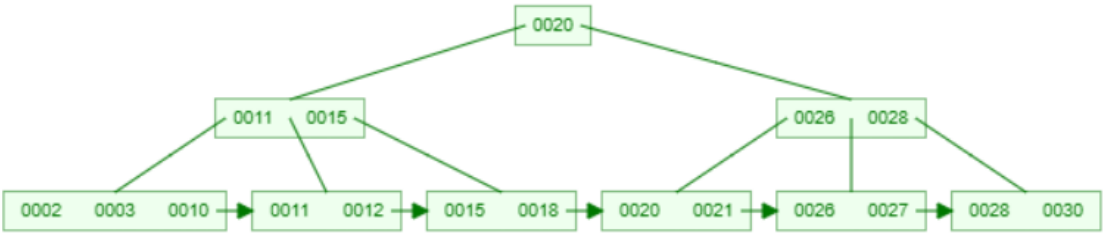
3. Como o nó está com mais do que o máximo possível de chaves, realiza-se um split, passando o valor médio para o pai (nesse caso 33).



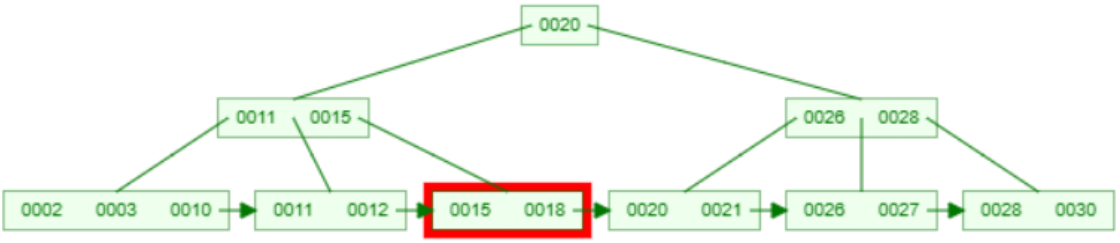
Note que, mesmo sendo promovido, a chave 33 permanece com uma cópia no nó folha, mantendo a regra da árvore B+.

III.III Remoção na Árvore B+

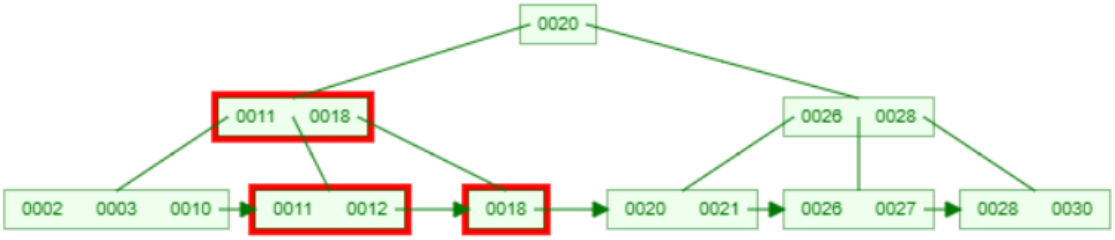
Do mesmo modo que as outras operações, a remoção na árvore B+ segue os mesmos princípios da árvore B com as funções de empréstimo e merge, mas com algumas diferenças importantes devido às características únicas da estrutura. Aqui, como os nós internos são meramente referências para percorrer a árvore, todas as remoções ocorrem nas folhas, existindo então dois casos possíveis: remoção de chave presente apenas nas folhas e de chave com cópia em nó interno.
No primeiro caso, pode-se apenas remover o nó e, se necessário, utilizar o empréstimo ou merge para manter o balanceamento. Já no segundo caso, é preciso, além de excluir a chave na folha, excluí-la no nó interno, também utilizando o empréstimo ou merge para manter o balanceamento. Vejamos um exemplo:
Seja a árvore B+ com $t = 3$ (ou seja, o mínimo de chaves é $t-1 = 2$) a seguir, vamos excluir a chave 15:



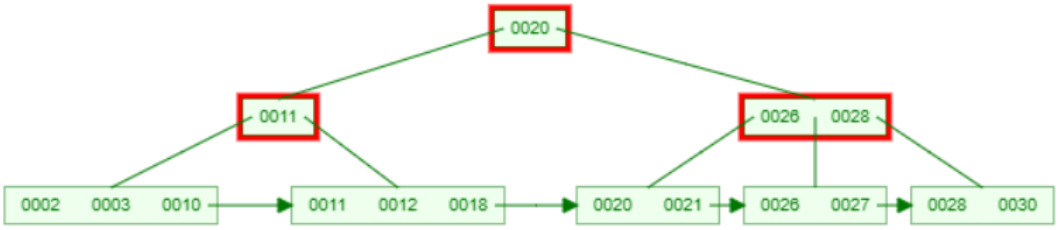
1. Primeiramente, realiza-se a busca.



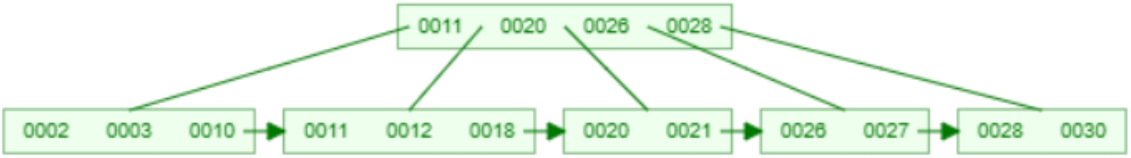
2. Depois, a chave em questão é excluída. Como o 15 também está presente em um nó interno, ele também deve ser excluído. Então, para manter o balanceamento, seu sucessor (18) é promovido para o pai. Porém, vemos que isso cria um problema.



3. Como o nó folha da chave 18 só tem um elemento, é preciso realizar outra operação para manter as propriedades da árvore, dessa vez um merge com as duas folhas. Note como, após a operação, o 18 deixa de ter uma cópia no nó interno, estando presente apenas na folha unificada, o que também causa um problema.



4. Agora, para finalizar a remoção, é necessário realizar outra operação de merge, dessa vez envolvendo os dois nós internos e a raiz da estrutura. A árvore final será então a seguinte:



De modo resumido, a remoção na árvore B+ segue os seguintes passos:

- Remover a chave desejada
- Se ela tiver uma cópia em um nó interno, removê-lo
- Ajustar os nós internos para refletir as mudanças, promovendo ou combinando nós conforme necessário
- Repetir as operações até a árvore ser perfeitamente balanceada

IV Comparação com Outras Estruturas

Antes de comparar as funcionalidades da árvore B+ com outras estruturas e ver porque ela é uma das estruturas mais utilizadas no mundo moderno, é preciso entender tais estruturas, como árvores AVL e Red-Black. Vamos então explorar seus conceitos e funcionalidades.

IV.I Árvores AVL

Criada em 1962 pelos pesquisadores russos Georgy Adelson-Velsky e Evgenii Landis com o propósito de tornar uma árvore binária auto-balanceável, a árvore AVL foi a primeira estrutura a trazer regras e funcionalidades de modo a aumentar a eficiência de árvores na época. Elas atuam de forma análoga a árvores binárias de busca, mas se baseiam em um sistema de pesos, definidos pela diferença de altura nas subárvores, para realizar rotações e tornar a árvore balanceada automaticamente.

Nela, a altura de uma subárvore direita não pode ter mais de dois níveis de diferença da subárvore esquerda e vice-versa. Caso isso ocorra, são realizadas rotações entre os nós até que essa diferença de altura seja, no máximo, igual a 1. Isso resolveu o problema de desempenho em árvores binárias de busca degeneradas. Quando uma árvore binária de busca cresce de forma desbalanceada, as operações podem ter um custo de O(n), onde n é o número de nós. A ideia era criar uma estrutura que mantivesse o balanceamento e garantisse O(log n) em todas as operações.
Em questões de aplicação, graças à sua estrutura, árvores AVL são melhor utilizadas em casos onde são necessárias muitas buscas e poucas inserções e remoções.

IV.II Árvores Red-Black

Introduzidas em 1972 por Rudolf Bayer, sob o nome de "árvores simétricas binárias", as árvores Red-Black - nomenclatura popularizada por Leonidas J. Guibas e Robert Sedgewick em 1978 - propunham-se a fornecer um balanceamento mais flexível que permitisse operações de inserção e remoção ligeiramente mais simples e eficientes em comparação com as árvores AVL. Com essa intenção, a estrutura abandonou o uso de pesos para determinar a diferença de altura entre subárvores, introduzindo o conceito de cores (vermelho e preto) para determinar a necessidade de balanceamentos, reduzindo significativamente o número de rotações necessárias em algumas operações.
As propriedades essenciais da estrutura são:

1. Um nó é vermelho ou preto;
2. A raiz é preta;
3. Todas as folhas (nil) são pretas;
4. Ambos os filhos de todos os nós vermelhos são pretos;
5. Todo caminho de um dado nó para qualquer um de seus nós folha descendentes contém o mesmo número de nós pretos.

Sendo a quinta regra a principal, já que ela estabelece de fato a necessidade de rotações para o balanceamento.
Em termos de aplicação, árvores Red-Black são preferidas em sistemas onde há uma frequência maior de inserções e remoções em relação às buscas, já que a estrutura requer no máximo duas rotações após a inserção ou remoção. Por causa disso, a estrutura é ideal para sistemas de processamento com fluxos contínuos de dados.

IV.III Comparação de Complexidade com Árvores B+

Como dito anteriormente, as árvores B+ estão entre as estruturas mais utilizadas no globo atualmente. Porém, por quais motivos essa estrutura em específico se sobressai com relação a outras estruturas como a árvore AVL ou Red-Black? Através de tabelas de comparação é possível entender um pouco melhor esse motivo.

1. Comparação na Busca:

Tipo	Complexidade Média	Comentários
AVL	O(log n)	Garante operações de busca muito eficientes, devido ao balanceamento estrito
Red-Black	O(log n)	Tem uma busca ligeiramente mais lenta que a AVL graças à sua altura maior (menos estrito)
B+	O(log n) para localizar folha e O(1) nas folhas	Também garante uma busca eficiente graças à estrutura em camadas e ao armazenamento das chaves nas folhas

2. Comparação na Inserção:

Tipo	Complexidade Média	Comentários
AVL	O(log n)	Pode ser mais lenta devido à necessidade de rotações frequentes
Red-Black	O(log n)	Realiza no máximo duas rotações e alterações, garantindo maior desempenho na inserção
B+	O(log n)	Inserções em nós internos podem demandar divisão do nó e atualizações nos nós pai, o que pode atrasar a operação. Porém, é ideal para grandes volumes de dados

3. Comparação na Remoção:

Tipo	Complexidade Média	Comentários
AVL	O(log n)	Exige rebalanceamento completo em muitos casos, com várias rotações, o que deixa o processo lento
Red-Black	O(log n)	Geralmente apresenta menos rotações que a AVL, por conta do balanceamento mais permissivo, o que aumenta o desempenho
B+	O(log n)	Pode necessitar de redistribuição ou mesclagem de nós internos após remoção, porém é muito eficaz em remoções sequenciais

Após a comparação e análise das estruturas, fica clara a maior eficiência da árvore B+ sobre as árvores AVL e Red-Black, na maioria dos casos. Embora a AVL tenha uma busca extremamente rápida e a Red-Black seja ideal para inserções e remoções, essas estruturas encontram suas limitações em ambientes com números exorbitantes de dados. É justamente nesse contexto que a árvore B+ se destaca. Mesmo não sendo tão eficiente quanto suas

concorrentes em pequenos volumes, essa estrutura atua como nenhuma outra em bancos de dados densos e em sistemas baseados em disco, onde são necessárias operações complexas de busca, inserção e remoção.

V Exemplos de Aplicação

Após entender as propriedades e o funcionamento da árvore B+, é possível aprofundar-se em aplicações reais da estrutura, percebendo sua verdadeira importância no mundo moderno.
Árvores B+ são a base de inúmeros sistemas, como no gerenciamento de bancos de dados para implementar indexação, sistemas de arquivos, gerenciamento de memória, bancos de dados NoSQL e NewSQL, mecanismos de pesquisa e até mesmo em sistemas de informação geográfica (GIS). A seguir, serão melhor detalhadas algumas dessas aplicações.

V.I Árvores B+ em Sistemas de Arquivos

Árvores B+ são as estruturas subjacentes de grande parte dos sistemas de arquivos modernos, como NTFS, ext4 e APFS - utilizados no Windows, Linux e MacOS, respectivamente. Graças a sua estabilidade e rapidez, a árvore B+ é ideal para esses sistemas, que necessitam gerenciar grandes volumes de dados em disco com alta eficiência.
A propriedade da estrutura de armazenar todas as chaves em folhas, utilizando os nós internos apenas como índices, facilita o acesso sequencial e o armazenamento otimizado em blocos de disco. Além disso, como as folhas são ligadas entre si por ponteiros, é possível realizar varreduras mais rapidamente, funcionalidade extremamente importante para a leitura de diretórios ou listas de arquivos.
No NTFS, por exemplo, a Master File Table (MFT) utiliza árvores B+ para indexar os arquivos do sistema. Cada nó interno guarda informações de localização e tamanho, enquanto as folhas armazenam referências para os blocos reais do disco.

V.II Árvores B+ em Bancos de Dados NoSQL e NewSQL

Em bancos de dados modernos, especialmente aqueles projetados para escalabilidade e alta disponibilidade, é comum o uso extensivo de árvores B+ como índice primário para o gerenciamento em grande escala. Graças à sua organização única, a estrutura facilita muito a busca e recuperação de blocos grandes de dados, reduzindo a quantidade de acessos aleatórios ao disco. Bancos como o MongoDB (NoSQL) e MySQL (NewSQL), por exemplo, utilizam inclusive a B+ para criar índices nas colunas de suas tabelas.
Outro ponto importante é que, por conta das redistribuições e fusões de nós em casos de remoção ou inserção, sistemas baseados em árvores B+ tendem a se manter balanceados sem a necessidade de reconstruir totalmente seus índices. Isso é crucial para bancos de dados mais robustos - que lidam com milhões de atualizações em tempo real - como o Cassandra, banco de dados criado pela Meta, atualmente utilizado pela Apple, Netflix, BlackRock, Uber e muitas outras empresas.

V.III Árvores B+ em Sistemas de Informação Geográfica (GIS)

Usado para criar, manejar, analisar e mapear uma variedade de dados, o GIS combina informações integrando a localização e todos os tipos de descrição de lugares de interesse em torno do globo. Esse aglomerado de referências provê uma fundação sólida para o mapeamento e análise de dados, muito utilizados por cientistas e indústrias. É justamente pelo GIS ser um sistema de dados tão denso e importante que a utilização de árvores B+ é essencial para armazenar e acessar essas grandes quantidades de dados geoespaciais de forma eficiente.
A B+ ganha notoriedade aqui porque sua estrutura torna o armazenamento de dados geoespaciais, como pontos de latitude e longitude, uma tarefa cabal. Por exemplo, ao indexar objetos como cidades ou rios, cada nó da B+ pode armazenar intervalos de coordenadas, tornando o manuseio dessas informações mais simples. Outro exemplo relevante a ser citado é o do banco de dados geoespacial PostGIS - baseado em PostgreSQL - que utiliza árvores B+ para indexar metadados e dados lineares, como estradas, linhas de ônibus, linhas de trem, etc.

VI Conclusão

Neste trabalho, foi possível explorar de maneira detalhada as árvores B e B+, analisando desde suas origens até suas características estruturais e funcionais. Ao longo do desenvolvimento, ficou evidente o papel essencial das estruturas em sistemas computacionais modernos, especialmente a B+, com sua superioridade em aplicações que demandam eficiência no acesso e manipulação de grandes volumes de dados em disco.
A análise comparativa entre as árvores B+, AVL e Red-Black mostrou não apenas semelhanças em termos de complexidade assintótica, mas também diferenças importantes no comportamento prático das estruturas. Enquanto a AVL e Red-Black se destacam em sistemas com alta dinamicidade, a B+ brilha em cenários onde consultas de intervalo e operações sequenciais são críticas, como sistemas de arquivos, bancos de dados e outros.
Inclusive, ao abordar algumas das aplicações reais da B+, foi possível evidenciar como sua estrutura possibilita atender necessidades cruciais do mundo moderno. Seja em sistemas operacionais, no gerenciamento de índices de bancos de dados NoSQL e NewSQL ou no processamento de informações espaciais em GIS, a B+ demonstrou ser uma escolha fundamental para arquiteturas mais recentes.
Em conclusão, este estudo não apenas reforça a relevância da árvore B+ no contexto teórico, mas também evidencia sua aplicabilidade prática, contribuindo para um entendimento mais profundo de como a escolha de estruturas de dados impacta diretamente o desempenho e a eficiência de sistemas computacionais contemporâneos.

VII Bibliografia

1. COMER, Douglas. **The Ubiquitous B-Tree**. Carlos Proal. Disponível em: The Ubiquitous B-Tree. Acesso em: 12 jan. 2025.
2. OKASAKI, Chris. **Purely Functional Data Structures**. Local de publicação: Cambridge University Press, 1999.

3. RABL, Tilman. B-trees and how this data structure changed the way of storing data and accessing it efficiently. **Hasso Plattner Institut**, 2019. Disponível em: B-trees and how this data structure changed the way of storing data and accessing it efficiently. Acesso em: 12 jan. 2025.
4. LET'S meet B-trees. **Blogs Archive**, 2017. Disponível em: Let's meet B-trees | Blogs Archive. Acesso em: 12 jan. 2025.
5. ABHISHEK, Ujjwal. B Tree and B+ Tree in DBMS | Core Computer Science. **work@tech**, © 2020-2022. Disponível em: B Tree and B+ Tree in DBMS | Core Computer Science. Acesso em: 14 jan. 2025.
6. UNDERSTANDING B+ Trees: A Comprehensive Guide. **Medium**, 2024. Disponível em: Understanding B+ Trees: A Comprehensive Guide | Medium. Acesso em: 15 jan. 2025.
7. B+ Tree in Data Structure (Explained With Examples). **WsCube Tech**, 2024. Disponível em: B+ Tree in Data Structure (Explained With Examples). Acesso em: 15 jan. 2025.
8. WHAT is GIS. **Esri**. Disponível em: What is GIS? | Geographic Information System Mapping Technology. Acesso em: 17 jan. 2025.