PESQUISA, ORDENAÇÃO E TÉCNICAS DE ARMAZENAMENTO

UNIDADE 4 - TÉCNICAS DE PESQUISAS SEQUENCIAL E BINÁRIA

Keila Barbosa Costa

Introdução

A busca por uma palavra-chave ou por elementos, valores ou dados específicos (informações) é a base de muitos aplicativos de computação. Serve para visualizar o saldo de uma conta bancária a partir de um mecanismo de pesquisa na internet, por exemplo, ou para procurar um arquivo no computador. Este, inclusive, lida com muitos dados, por isso, precisamos de algoritmos eficientes para a pesquisa.

No entanto, que tipos de algoritmos podem ser eficientes? Como os computadores precisam buscar informações em coleções de dados muito grandes, é essencial usar o algoritmo certo para pesquisar, mas como definir o algoritmo adequado para cada situação? Será que podemos aplicar métodos de pesquisas binária ou sequencial com os dados organizados de qualquer forma?

A pesquisa é o processo de encontrar o elemento em determinada lista. Neste processo, verificamos que o item está disponível ou não. A pesquisa está em toda parte, desde encontrarmos o endereço de uma pessoa até um número de telefone em uma lista telefônica. Neste contexto, ao longo desta última unidade, iremos explorar algoritmos comuns que são usados para procurar dados em computadores.

Aprenderemos sobre pesquisa binária em vetores e pesquisa sequencial. Veremos a respeito das particularidades de cada uma e como são aplicadas no dia a dia dos profissionais da área. Também estudaremos a respeito das tabelas de dispersão e das árvores de busca.

Vamos, então, nos aprofundar a respeito desses assuntos? Acompanhe!

4.1 Técnicas de pesquisas

Podemos definir como dado um conjunto de elementos, em que cada um é identificado por uma chave. O objetivo da pesquisa é localizar, dentro desse conjunto, o elemento que corresponde a uma chave específica.

Assim, uma tarefa bastante importante, hoje — não só na computação como também em outras áreas —, é a pesquisa de informação contida em coleções de dados. De forma geral, é desejado que o procedimento seja realizado de forma que não tenha a necessidade de se verificar toda a coleção (VIANA; CINTRA; NOBRE 2015).

A busca é muito comum na área da computação, sendo que podemos usar métodos e estruturas de dados para se encontrar o elemento esperado. A busca em vetor, por exemplo, pode ser feita por índice ou valor.

A busca realizada pelo **índice** é considerada uma busca direta, ou seja, vai direto na posição da memória. Podemos tomar como exemplo o quadro a seguir com o vetor “nome”. Passando no índice 2, teríamos o valor “Khyonara”.

0	1	2	3	4
Keila	Arthur	Khyonara	Berenice	Aparecido

Quadro 1 - Exemplo de quadro com vetores

Agora, vamos imaginar que não sabemos onde o **valor** “Khyonara” está. Neste caso, temos que verificar todos os elementos do vetor até encontrá-lo.

Portanto, para realizar a busca por um valor, temos dois métodos a serem considerados: realizamos uma busca sequencial ou uma busca binária.

A **busca sequencial** percorre todas as posições do vetor, verificando uma a uma, até achar o valor desejado ou simplesmente chegar ao final sem achá-lo. Já na **busca binária**, o vetor é dividido ao meio e a busca é realizada apenas em uma das metades.
Vamos entender melhor sobre os dois métodos a partir de agora. No próximo item, ainda mostraremos exemplos de como o processo é realizado dentro da computação.

4.1.1 Busca sequencial

De acordo com Viana, Cintra e Nobre (2015), a busca sequencial é definida como a procura por um valor x em um vetor L. Inspecionando em sequência as posições de L a partir da primeira posição, podemos encontrar o x. Ao encontrar o elemento procurado, podemos dizer que a busca obteve sucesso. Caso contrário, se chegarmos à última posição sem encontrarmos a posição desejada, concluímos que ela não ocorre no vetor L.
Vamos imaginar o mesmo vetor "nome", em que desejamos encontrar o nome "Khyonara". Como isto pode ser realizado? Comparando elemento a elemento, ou seja, o valor que está no índice pelo valor a ser procurado.

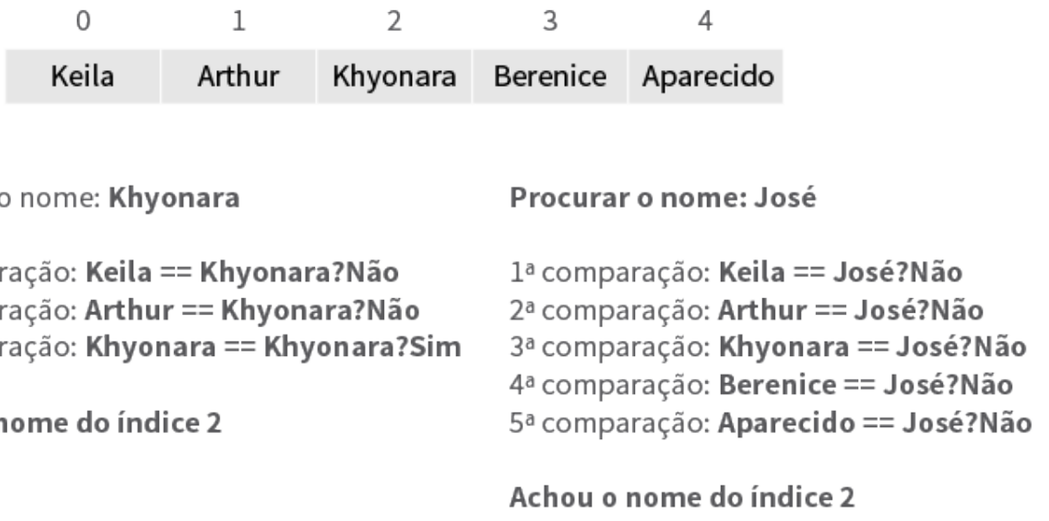


Figura 1 - Exemplo de busca sequencial

De acordo com a figura anterior, temos a primeira comparação realizada. Nela, percorremos o valor que está na posição 0. Ele é igual ao valor que estamos procurando (Khyonara)? Na verdade, não!
Incrementando o índice e realizando a mesma verificação, podemos dizer que o valor que está na posição ou no índice 1 é o mesmo que estamos procurando? Arthur é igual a Khyonara? Também não, certo?
Incrementando novamente o índice e realizando mais uma comparação, podemos dizer que o índice 2 é igual ao valor que estamos procurando? Khyonara é igual a Khyonara? Sim! Assim, encontramos nosso elemento a partir da posição do índice que está no valor buscado.
Agora, imagine que iremos procurar o nome José. Perceba que ele não se encontra no vetor que estamos analisando, por isso, o algoritmo irá parar de executar quando chegar ao final, retornando e informando que não foi encontrado o nome desejado.
No entanto, imaginemos que o vetor tenha mais de um milhão de posições, sendo que o valor desejado não se encontra no vetor. Será necessário, então, realizar um milhão de verificações para se descobrir que o valor não se encontra no vetor a ser verificado. Isto ocorre com frequência na busca sequencial, sendo que, para resolver o problema de o algoritmo ter que verificar elemento por elemento, podemos usar a busca binária.

Na busca binária, o vetor precisa estar ordenado. A mesma lógica serve para números, textos ou letras. Assim, agora que você já conhece o processo de como fazer uma busca sequencial, iremos conhecer como se dá a busca binária. Vejamos!

4.1.2 Busca binária

A busca binária só funciona em vetores que estejam ordenados. Isto porque ela divide sucessivamente o vetor ao meio e procura apenas em uma das metades, ou seja, o algoritmo é executado até encontrar o valor ou posição. Vamos ver um exemplo com a figura a seguir. Observe com atenção!

Figura 2 - Exemplo de busca binária

Imagine o vetor ordenado na figura anterior. Pretendemos procurar o elemento 20, por isso, a primeira coisa que o vetor irá fazer é descobrir a posição inicial (0) e, depois, a posição final (11). Será realizado o seguinte cálculo:
meio = (posiçaoInicial + posicaoFinal) / 2
meio = (0 + 11) / 2
meio = 5.5 (Arredondar para o inteiro mais próximo 5)
Temos, assim, que a quinta posição será a posição do meio, com valor igual a 12.
Encontrada a posição do meio, será realizada a verificação a seguir:
Se (meio = numeroProcurado) (18 = 20)
Achou o valor
Do contrário
Se (numeroProcurado < meio) (20 < 18)
posiçãoFinal = meio - 1
Do contrário
posiçãoInicial = meio + 1 // Realizar novamente a posição inicial.
Dessa forma, podemos verificar o número procurado, ou seja, se o elemento do meio é igual ao número procurado (12 = 20). Como a afirmação é falsa, segue-se, então, para 20 < 12. Como a afirmação também não é verdadeira, o vetor irá andar com a posição inicial.
Como a posição inicial era 0, agora ele vai passar a ser: posiçãoInicial = meio + 1 = 5 + 1 = 6), então, recebe a posição 6. Perceba que a posição 6 tem o valor 14.
A parte do vetor com valor menor que 12 (< posição 5) foi desconsiderada, pois, como o vetor está ordenado, iremos considerar apenas da posição 6 até a 11.

Com a posição inicial 6 e a posição final 11, é realizado um cálculo para obtermos a posição meio, como vemos a seguir, até se alcançar a posição desejada. No nosso caso, trata-se da posição 9. Observe:
meio = (posiçaoInicial + posicaoFinal) / 2
meio = (6 + 11) / 2
meio = 8.5 (inteiro 8) vetor formado por números inteiros.
Realizando a verificação, temos:
Se (meio == numeroProcurado) (12 = 20)
Achou o valor
Do contrário
Se (numeroProcurado < meio) (20 < 12)
posiçãoFinal = meio - 1
Do contrário
posiçãoInicial = meio + 1
Assim, foram necessárias muitas verificações até se encontrar o elemento desejado, porém, se fossemos usar a verificação da busca sequencial, seriam necessárias ainda mais verificações. Isto é, para a busca sequencial, teríamos que realizar 10 verificações, enquanto a busca binária realizou apenas três.

VAMOS PRATICAR?

De acordo com o que vimos, imagine o vetor ordenado a seguir. Procure o elemento 22 usando a busca binaria.

3	7	9	11	12	14	15	22	23	25

Agora que já entendemos a respeito das buscas sequencial e binária, no tópico a seguir, veremos com mais detalhes a respeito das tabelas de dispersão, também conhecidas como tabelas de espalhamento ou tabelas de *hashing*. Acompanhe!

4.2 Tabelas de dispersão

Um paradigma muito comum no processamento de dados envolve o armazenamento de informações em uma tabela e, posteriormente, a recuperação dos dados guardados. Por exemplo, considere um banco de dados de registros de carteira de trabalho: ele contém um registro para cada carteira de trabalho emitida, sendo que, dado o número da carteira de trabalho, podemos procurar as informações associadas a ele.
Normalmente, o banco de dados compreende uma coleção de pares e valores. As informações são recuperadas do banco a partir de uma determinada chave. No caso do banco de dados da carteira de trabalho, a chave é o número da carteira de trabalho. Já no caso de uma tabela de símbolos, por exemplo, a chave é o nome do símbolo.

Viana, Cintra e Nobre (2015) nos explicam que podemos definir as tabelas de dispersão, também conhecidas como tabelas de espalhamento ou de hashing, como aquelas que armazenam uma coleção de valores, sendo que cada valor está associado a uma chave. As chaves precisam ser todas diferentes, pois são usadas para mapear os valores das tabelas.

As vantagens da tabela de dispersão é que ela pode ser usada como índice, porém a grande vantagem está em se ter uma operação cujo acesso é direto. Isto é, não será preciso fazer um percurso em uma árvore ou comparar registro. Ainda de acordo com Viana, Cintra e Nobre (2015), outra vantagem é o curso da operação O(1) acesso, em que a determinação da posição onde o registro será armazenado virá de uma função hashing.

A ideia essencial por trás de uma tabela de dispersão é que todas as informações são armazenadas em uma matriz de tamanho fixo. O hashing é usado para identificar a posição em que um item deve ser armazenado. Quando acontece uma colisão, o item em colisão é armazenado em outro lugar na matriz (VIANA; CINTRA; NOBRE, 2015).

Os tipos de hashings podem ser divididos em. Clique no recurso a seguir.

Hashing fechado	Em que é permitido armazenar um conjunto de informações de tamanho limitado.
Hashing aberto	Em que é permitido armazenar um conjunto de informações de tamanho ilimitado.

A função de *hashing* será responsável por converter a chave em um índice de alocação. Vejamos um exemplo de função de dispersão:

h(chave) endereço

Função: h(x) = x mod 7, em que x é a chave e h(x) é o endereço

h(4315) 615, em que 4315 é a chave e 615 é o endereço

VAMOS PRATICAR?

De acordo com a função de *hashing*, encontre o endereço para a chave h(5426). Adote a fórmula "Função: h(x) = x mod 5".

Assim, temos que a função *hashing* vai ditar o desempenho do algoritmo, trazendo o ponto em que devemos armazenar o registro. O segredo está na qualidade da função *hashing*, ou seja, quanto melhor for a função, mais dispersos os registros serão.

Além disso, o *hashing* funciona da seguinte forma: a chave D1 passa pela função que é convertida no índice 1, enquanto a chave D2 passa pela função que vai ser convertida no índice 2. Veja o esquema a seguir para compreender melhor.

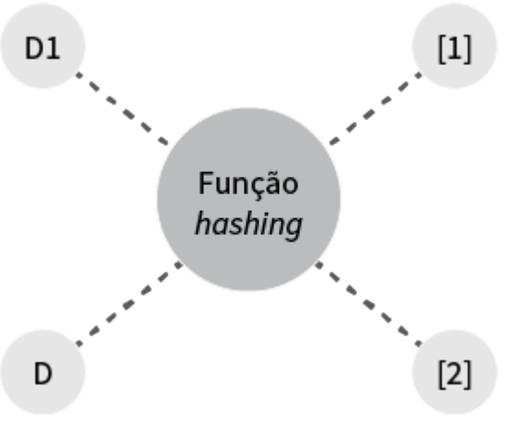


Figura 3 - Função hashing

Agora, vejamos um exemplo de como a função de *hashing* funciona em uma tabela de dados. Observe o quadro na sequência.

Índice	Chave (x)	Valor	Função: h(x) = x mod 7
0			
1			
2	30	Khyonara	h(x) = x mod 7 = 30/7
3	45	Keila	h(x) = x mod 7 = 45/7
4			
5	40	Arthur	h(x) = x mod 7 = 40/7
6			

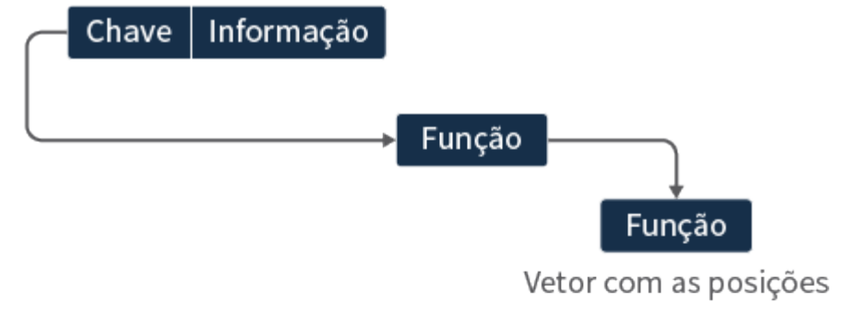
Quadro 2 - Processo da função hashing

Observe que o valor 30 não está na posição 0, mas, sim, na posição 2, devido à fórmula, ou seja, *h(x) = x mod 7 = 30/7 ˜= 4,2* . Como o restante é aproximadamente 2, temos para a chave 30 o lugar de alocação para a posição 2. Da mesma forma procede para as demais chaves.
Agora, digamos que desejamos incluir uma nova chave que também na divisão com 7 gere um resto 2. A este caso damos o nome de colisão ou ***hashing* fechado** (*hash overflow*).

VOCÊ SABIA?

O *hash overflow* é quando tentamos inserir um elemento-chave na tabela de dispersão, mas a tabela já se encontra cheia, invalidando essa inserção. Neste caso, dizemos que aconteceu um estouro de tabela de dispersão.

A colisão acontece quando determinado índice (posição no vetor) gerado por uma chave se encontra alocado por um registro. Isto é, quando o valor de uma chave ganha uma posição que já está cheia. Vejamos um exemplo na figura a seguir.



Chave	Resto	Adote: h(x) = x mod m	84	0
47	5		70*	1
9	9	h(47) = 47 mod 14 = 5	16	2
75	5	h(9) = 9 mod 14 = 9	25*	3
16	2		47	4
24	10	...	75	5
52	10		34*	6
84	0		12*	7
70	0			8
10	10		9	9
25	11		24	10
34	6		52*	11
12	12	h(12) = 12 mod 14 = 12	10*	12

Figura 4 - Processo de hashing fechado

Na figura anterior, temos o esquema de como funciona o *hashing* fechado, em que temos o dado com a chave que é convertida pela função, em um índice no qual será alocada a informação.

Podemos observar que nas chaves 24 e 52 acontece uma colisão, pois as duas têm um resto 10, ou seja, precisam ser alocadas na mesma posição 10. Porém, a posição já tem uma chave alocada que chegou primeiro (24). Neste caso, o que fazemos com a chave que precisa ser alocada, sendo que sua posição já está ocupada por outra chave?

O 24 já está alocando a posição 10, então, a chave 52 vai procurar a próxima posição vazia, que seria a 11. Portanto, a chave 52 ficará na posição 11 com asterisco (*), deixando claro que se trata de uma chave deslocada.

Na implementação real, adotamos que cada entrada H[i] da tabela é uma lista, cujo elemento têm *hash code i*. Para inserir um elemento na tabela, computamos o *hash code i* e inserimos o elemento na lista ligada à H[i].

A inserção é realizada seguindo a ordem de cima para baixo na tabela, sendo que, ao final da lista, retorna-se para o início, seguindo o ciclo novamente.

A chave deslocada só sai da posição também deslocada se for para ir à sua respectiva posição, de acordo com a função.

A técnica de ***hashing* aberto**, por outro lado, denota que, em uma tabela, cada posição possui um ponteiro que dá para uma lista encadeada, sendo que esta contém todas as chaves mapeadas por uma função *hashing* na posição (VIANA; CINTRA; NOBRE, 2015). Aliás, na técnica aberta, não existe limite de chaves que podem ser armazenadas na tabela.

A primeira estratégia, comumente conhecida como *hashing* aberto ou encadeamento separado, é manter uma lista de todos os elementos com *hashing* no mesmo valor. Vejamos um exemplo concreto na tabela a seguir, em que é possível observar na posição 10 duas colisões. Assim, foram adicionados dois campos extras para alocar as posições.

Chave	Resto	Adote: h(x) = x mod m	0	→	84	70	
47	5		1				
9	9	h(47) = 47 mod 14 = 5	2	→	16		
75	5	h(9) = 9 mod 14 = 9	3				
16	2		4				
24	10	...	5	→	47	75	
52	10		6	→	34		
84	0		7				
70	0		8				
10	10		9	→	9		
25	11		10	→	24	52	10
34	6		11	→	25		
12	12	h(12) = 12 mod 14 = 12	12	→	12		

Tabela 1 - Processo de hashing aberto

Para realizar uma localização, usamos a função *hash* para determinar qual lista percorrer. Em seguida, percorremos a lista da maneira normal, retornando à posição em que o item foi encontrado. Para executar uma inserção, percorremos a lista apropriada para verificar se o elemento já está no lugar. Se são esperadas duplicatas, geralmente um campo extra é mantido e este seria incrementado no caso de uma correspondência (VIANA; CINTRA; NOBRE, 2015). Se o elemento for novo, ele será inserido na frente da lista ou no final. Este é um problema fácil de resolver enquanto o código está sendo gravado. Às vezes, novos elementos são inseridos na frente da lista, pois é conveniente e também porque frequentemente acontece que os elementos inseridos recentemente têm maior probabilidade de serem acessados em um futuro próximo.

VAMOS PRATICAR?

Usando a tabela a seguir, realize a localização das posições das chaves usando a função *hash*. Em seguida, percorra a lista da maneira normal, retornando à posição em que o item foi encontrado. Em caso de colisão, adote o processo de *hashing* aberto para resolver o conflito.

Chave	Resto						
50		Adote: h(x) = x mod 3	0				
23			1				
45			2				
17			3				
9			4				
37			5				
			6				

Assim, o *hashing* aberto é uma estratégia em que nenhum dos objetos é realmente armazenado na matriz da tabela de *hashing*. Em vez disto, ele é armazenado em uma lista separada da matriz interna da tabela.

4.3 Árvores de buscas balanceadas

Uma árvore de pesquisa binária balanceada mantém automaticamente sua altura pequena (garantida como logarítmica) para uma sequência de inserções e exclusões. Conforme Viana, Cintra e Nobre (2015), essa estrutura fornece implementações eficientes para dados abstratos, como matrizes associativas.
Diferentemente das listas, em que as informações se encontram em uma sequência; nas árvores, esses dados são representados de forma hierárquica, sendo que cada nó possui um conteúdo (VIANA; CINTRA; NOBRE, 2015).

VOCÊ QUER VER?

Uma árvore binária é composta de nós, sendo que cada nó contém uma referência "esquerda", uma referência "direita" e um elemento de dados. O nó superior da árvore é chamado de “raiz”. Para compreender melhor o funcionamento de uma árvore binária, sugerimos que você assistia ao vídeo disponível no *link*: https://youtu.be/PgZflufXGUU. Com ele, será mais fácil identificar os pontos apresentados.

Nas árvores de busca balanceada, as chaves alocadas são mantidas ordenadas, permitindo que a operação de busca seja realizada, percorrendo-se um ramo da árvore, desde a base até chegar ao início (VIANA; CINTRA; NOBRE, 2015).
As árvores de pesquisa são como um método para armazenar dados, de modo a oferecer suporte para as operações de inserção rápida, pesquisa e exclusão. O principal problema das árvores de busca é o desejo que elas sejam equilibradas para que as pesquisas possam ser executadas rapidamente. No entanto, não é necessário que elas sejam perfeitas, visto que seria muito caro de manter para inserir ou excluir um novo elemento.

Nesse contexto, vários algoritmos foram desenvolvidos para a construção de árvores de busca que permanecem equilibradas.

4.3.1 Árvore AVL

A árvore AVL é uma árvore binária que vai seguir as mesmas regras para inserção, busca e remoção de elementos, bem como adicionar tais regras a métodos para se manter o equilíbrio da árvore. Ela é muito balanceada: em suas inserções e exclusões, procura-se executar uma rotina de balanceamento em que as alturas das sub-árvores sejam próximas.

VOCÊ O CONHECE?

Georgy Adelson era um matemático e cientista de computação. Nascido em Samara, Adelson foi originalmente educado como um matemático puro. Seu primeiro trabalho, com o colega e eventual colaborador de longa data, Alexander Kronrod, em 1945, ganhou um prêmio da Sociedade Matemática de Moscou. Ele começou a trabalhar em inteligência artificial e outros tópicos aplicados no final da década de 1950. Junto com Evgenii Landis, inventou a árvore AVL em 1962. Esta foi a primeira estrutura de dados da árvore de pesquisa binária balanceada já conhecida.

Conforme nos explicam Viana, Cintra e Nobre (2015), as operações básicas, como busca, inserção, remoção e outras, levam um tempo proporcional ao número de níveis da árvore binária de busca. Isto é, a quantidade de operações para busca, inserção e remoção, no pior dos casos, depende de quantos níveis existem na árvore. Dada esta observação, podemos concluir que é desejado manter a árvore sempre com a menor quantidade de níveis possível para que as operações básicas não custem tanto tempo.
Vejamos um exemplo em que a inserção dos números se encontra em sequência (10, 11, 12, 13, 14, 15, 16, 17 e 18), ou seja, foi inserido número por número de forma sequencial na árvore. Observe a figura a seguir.

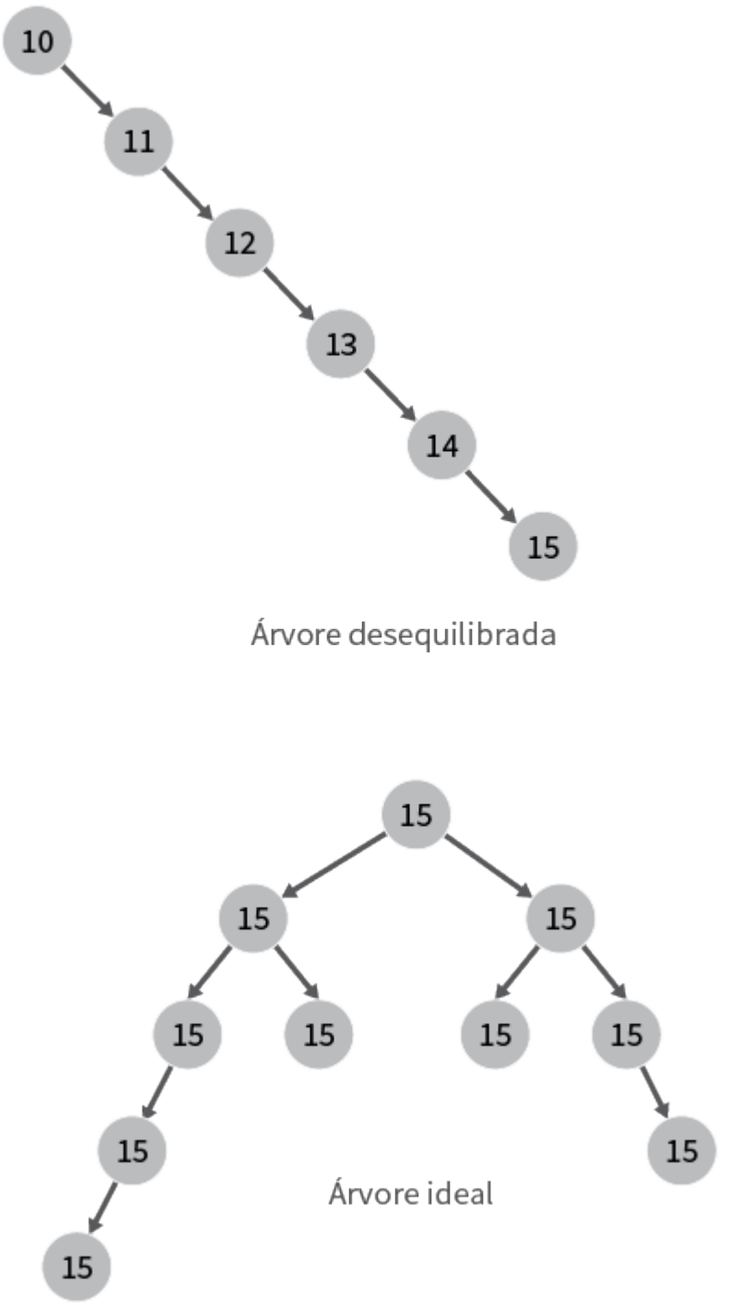


Figura 5 - Distribuição de árvore sequencial

Neste caso, para elementos, a árvore teria níveis, resultando em uma péssima distribuição. O ideal seria que os nós, conforme sejam inseridos, também sejam rearranjados para manter um equilíbrio. Assim, o ideal seria manter a mesma quantidade de níveis na subárvore da direita e na subárvore da esquerda.

A questão é: como fazer para garantir esse equilíbrio? A resposta vai estar nas chamadas rotações. Podemos, inclusive, dividir o problema em duas partes: como detectar o desequilíbrio e como corrigir o desequilíbrio. Vamos começar entendendo como detectar o desequilíbrio.

O equilíbrio de uma árvore de busca é medido subtraindo o número de níveis na subárvore da esquerda do número de níveis na subárvore da direita.

Vejamos um exemplo a seguir, em que temos uma árvore qualquer e podemos observar cada um dos nós, bem como contar quantos níveis existe à esquerda e à direta, subtraindo esse número.

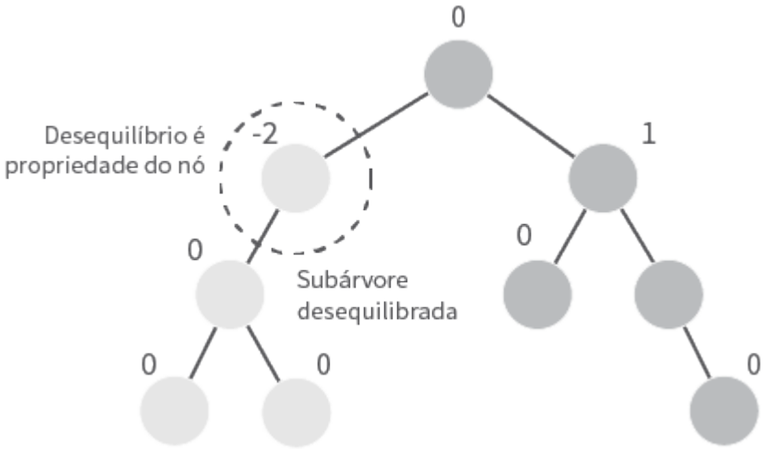


Figura 6 - Desequilíbrio de uma árvore binária

Pode-se dizer que a árvore está desequilibrada apenas quando o número for maior que 1 ou menor que -1. Assim, iremos tolerar números de equilíbrio de 0, 1 e de -1, pois podemos ter um número ímpar de nós, ou seja, em alguns casos, não conseguiremos fazer a árvore ter um equilíbrio completamente nulo.
Uma vez detectado o desequilíbrio na árvore, o próximo passo é entender como corrigi-lo, sendo este feito por meio das rotações.

VOCÊ QUER LER?

Um jogo educacional digital, denominado DEG4Trees (*Digital Educational Game four Trees*), foi construído para apoiar o ensino de estrutura de árvores binárias de buscas e árvores AVL. No jogo, é preciso observar as propriedades das estruturas de dados e interagir com elas para atingir os objetivos. Para saber mais, sugerimos a leitura do artigo "DEG4Trees: um jogo educacional digital de apoio ao ensino de estruturas de dados", deWeider Alves Barbosa, Isabella de Freitas Nunes, Ana Carolina Gondim Inocêncio, Thiago Borges de Oliveira e Paulo Afonso Parreira Júnior, disponível em: https://www.researchgate.net/profile/Weider_Alves_Barbosa2/publication/265786605_DEG4Tree_Um_Jogo_Educacional_Digital_de_Apoio_ao_Ensino_de_Estruturas_de_Dados/links/55fc6c1808ae07629e0d3628.pdf

Existem quatro tipos de rotações, sendo dois tipos primitivos e dois compostos de rotações: rotação à esquerda, rotação à direita, rotação dupla à esquerda e rotação dupla à direita. Clique no recurso a seguir para conhecer a rotação à esquerda e rotação à direita.

- Rotação à esquerda

  A rotação à esquerda é como o próprio nome sugere. Os primeiros nós que estão na subárvore da direita passam para a esquerda, fazendo com que o filho da direita se torne a nova raiz. Também temos o caso particular em que o filho da direita já possui o filho da esquerda, com todos os elementos da subárvore da direita maiores que a raiz (x > 1).

- Rotação à direita

  Já a rotação à direita vai funcionar de forma análoga, seguindo a mesma lógica da rotação à esquerda. Nela, temos uma diagonal para o outro lado, ou seja, o filho da esquerda vira a nova raiz (2 = raiz) e o filho da direita de 2 será o filho da esquerda de 3. Depois, 3 se tornará o filho da direita de 2.

Veja a figura a seguir para entender melhor.

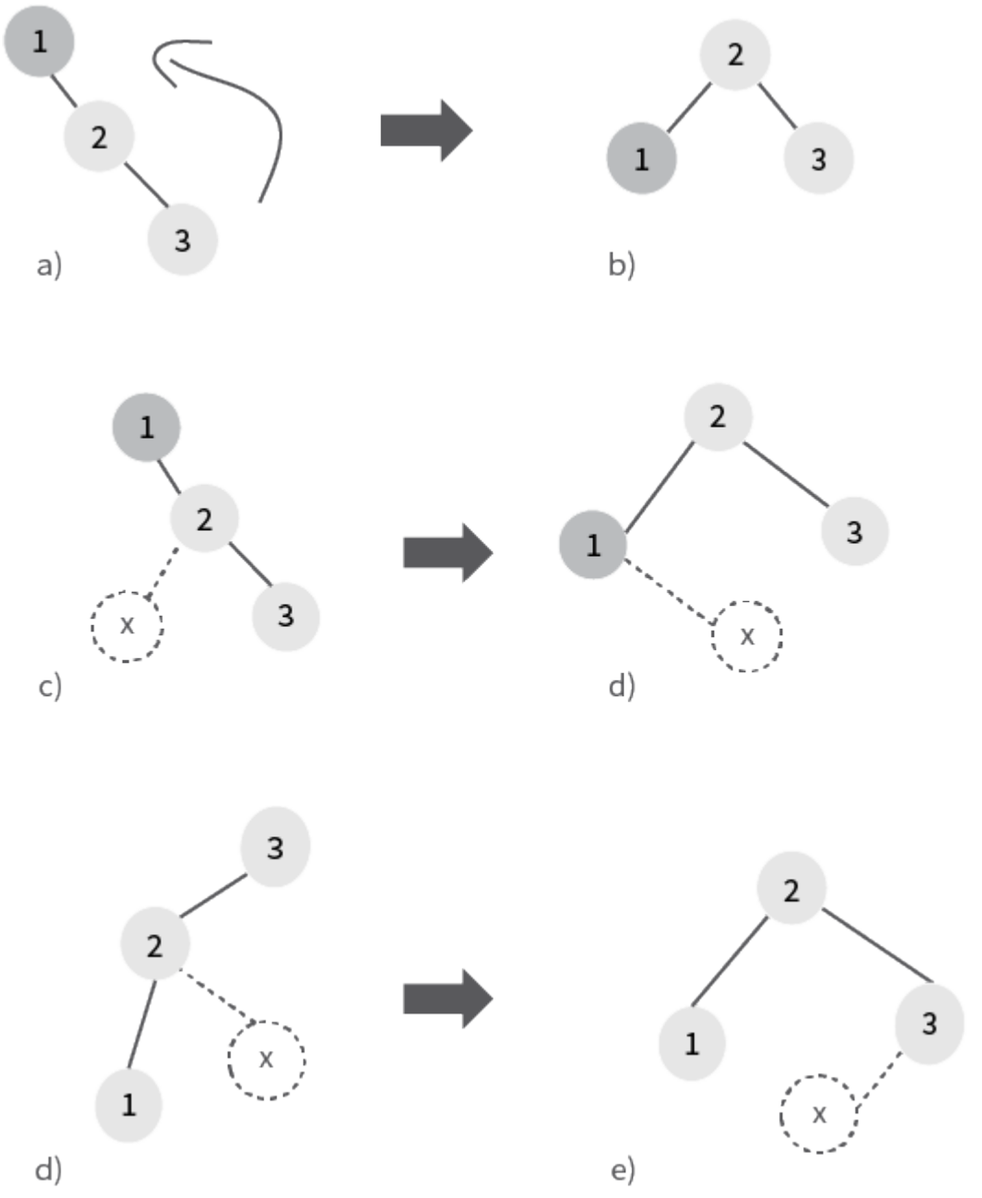


Figura 7 - Tipos de rotação para árvore AVL

Vamos tentar compreender, agora, como funciona a rotação dupla à esquerda e a rotação dupla à direita, que são rotações duplas compostas.
Considere a seguinte situação: a rotação simples à esquerda não resolve o desequilíbrio. Para detectar isto, basta olharmos o desequilíbrio das subárvores de onde a nova raiz vai vir (3). Temos, assim, um desequilíbrio positivo

na raiz original (1), mas na subárvore da direita percebemos um desequilíbrio negativo (3). Para corrigir o problema, podemos adotar como solução uma rotação à direita na subárvore da direita e, em seguida, uma rotação à esquerda na árvore original.

A **rotação dupla à direita** vai ser bem similar à **rotação dupla à esquerda**, conforme vemos na figura na sequência.

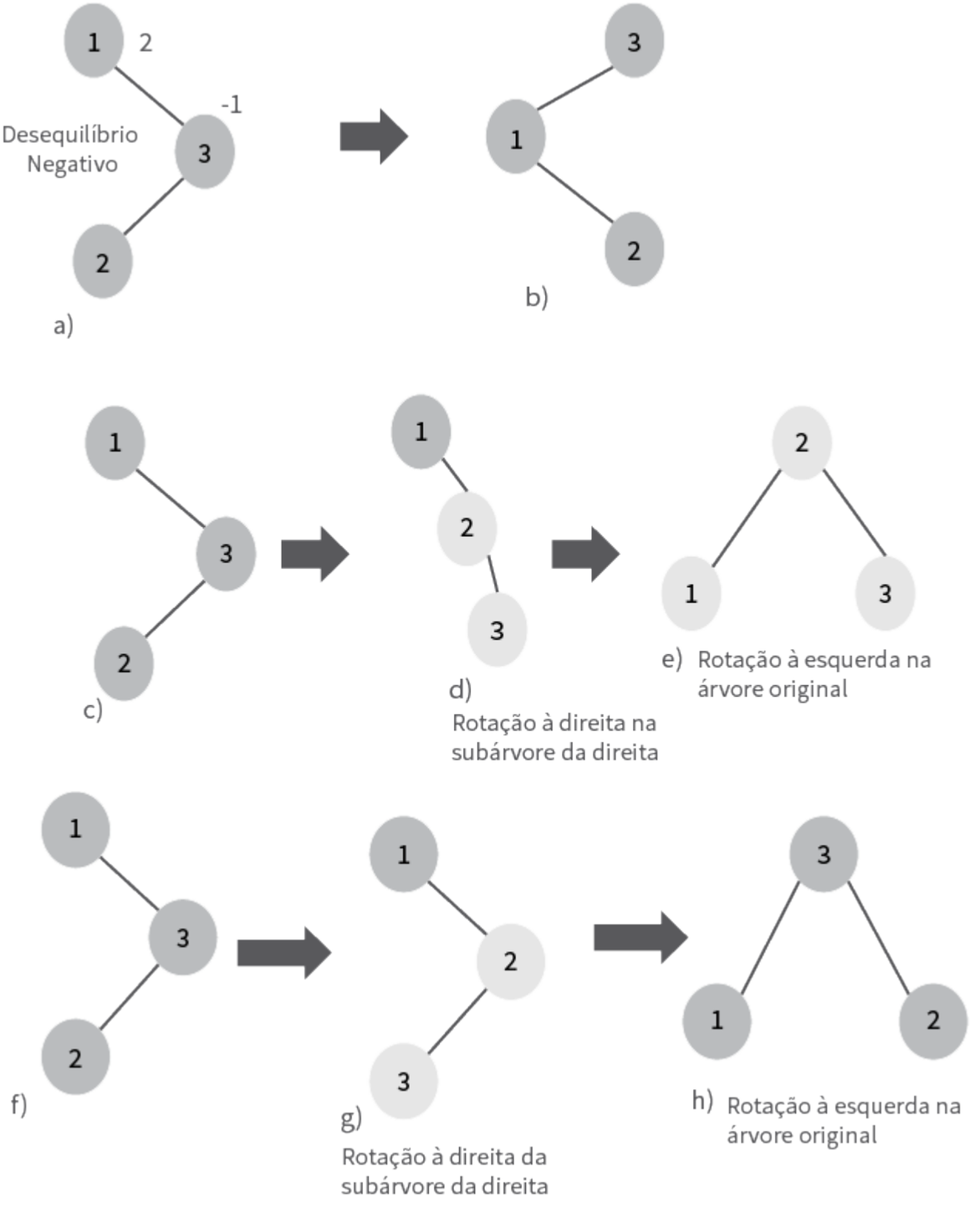


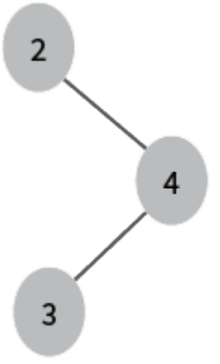
Figura 8 - Processo da rotação dupla à direita e à esquerda, respectivamente

Agora que já conhecemos a rotação dupla à esquerda e a rotação dupla à direita, como decidir qual delas devemos usar? Para tanto, precisamos seguir um passo a passo. Clique nos ícones a seguir.

Calcular o fator de equilíbrio (Q = R - L), em que “R” é o número de níveis à direita e “L” é o número de níveis à esquerda.
Se -1 ≤ Q ≤ 1, a árvore é equilibrada.
Se Q > 1, temos duas opções: se a subárvore da direita tem Q < 0, então a rotação seria dupla à esquerda, mas, do contrário, seria rotação à esquerda; se a subárvore da direita tem Q > 0, então a rotação seria dupla à direita, mas, do contrário, seria rotação à direita.

VAMOS PRATICAR?

Uma vez detectado o desequilíbrio na árvore, o próximo passo é entender como corrigi-lo. Use o processo da rotação dupla para corrigir o desequilíbrio da árvore a seguir.



Como já conhecemos a respeito da árvore AVL, vamos, no próximo item, estudar a respeito das árvores B e B+. Vejamos!

4.3.2 Árvores B e B+

As árvores B são uma forma de árvore de pesquisa equilibrada, baseada em árvores gerais. Um nó da árvore B pode conter vários elementos, em vez de apenas um, como nas árvores de pesquisa binária. Isto é, em vez de ter apenas dois filhos, ela pode ter vários.
De acordo com Viana, Cintra e Nobre (2015), as árvores B são especialmente úteis para estruturas de pesquisa armazenadas em discos. Estes têm características de recuperação diferentes da memória interna (RAM) e, obviamente, seu acesso é muito mais lento.
Na maioria das árvores de pesquisas, como a AVL, supõe-se que tudo esteja na memória principal. Contudo, para entender o uso das árvores B, devemos pensar na quantidade de dados que não cabe na memória principal.

Quando o número de chaves é alto, os dados são lidos do disco na forma de blocos. A ideia principal do uso de árvores B é reduzir o número de acessos ao disco.

Temos, assim, que as árvores B são uma boa combinação para armazenamento e pesquisa em disco, pois é possível escolher o tamanho do nó para corresponder ao tamanho do cilindro. Ao fazer isto, armazenaremos muitos membros de dados em cada nó, tornando a árvore mais plana, para que sejam necessárias menos transições (VIANA; CINTRA; NOBRE, 2015).

Uma estrutura de árvore B é balanceada automaticamente e, geralmente, permite que um nó tenha mais de dois filhos — para manter a árvore larga e, portanto, crescer em altura —, com várias chaves inseridas em um nó.

Normalmente, definimos uma árvore B com “ordem b”, em que “b” é o número mínimo de filhos e “2b” é o número máximo de filhos que qualquer nó pode ter.

A seguir, temos um exemplo de uma árvore B com b = 2.

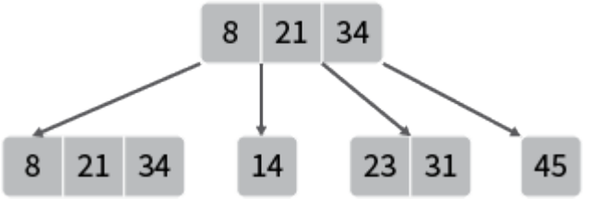


Figura 9 - Exemplo de árvore B

Na árvore da figura anterior, podemos perceber que cada número inteiro é considerado uma chave separada, ou seja, cada um teria seu próprio nó. Assim como em outras árvores, todas as chaves na subárvore à esquerda são menores que a chave atual, bem como todas as chaves na subárvore à direita são maiores. Por exemplo, começando na chave 8, as chaves 3, 4 e 7 são menores, enquanto a chave 14 é maior.

Observe, também, que, como o número máximo de filhos é 2 * 2 = 4, um nó pode armazenar apenas três chaves, pois os indicadores para filhos são armazenados entre as chaves e nas bordas.

Para implementar a indexação dinâmica em vários níveis, geralmente são utilizadas as árvores B e B+. A desvantagem da árvore B usada para indexação é que ela armazena o ponteiro de dados (ponteiro para o bloco de arquivos do disco que contém o valor da chave), correspondente a um valor de chave específico, juntamente com o valor de chave no nó de um B. Esta técnica reduz bastante o número de entradas que podem ser empacotadas em um nó de uma árvore B, contribuindo para o aumento do número de níveis e do tempo de pesquisa de um registro.

A árvore B+ elimina essa desvantagem, armazenando ponteiros de dados apenas nos nós das folhas da árvore. Assim, a estrutura é bem diferente da dos nós internos da árvore B+.

Vejamos algumas propriedades das árvores B e B+. Clique nos ícones a seguir.

Todas as folhas estão no mesmo nível.

Uma árvore B é definida pelo termo grau mínimo “t”, sendo que seu valor depende do tamanho do bloco de disco.

Todo nó, exceto a raiz, deve conter pelo menos chaves “t - 1”.

A raiz pode conter, no mínimo, uma chave.

Todos os nós, incluindo raiz, podem conter, no máximo, “2t - 1” chaves.

O número de filhos de um nó é igual ao número de chaves, mais 1.

Todas as chaves de um nó são classificadas em ordem crescente.

O filho entre duas chaves k1 e k2 contêm todas as chaves no intervalo de k1 e k2.

A árvore B cresce e diminui a partir da raiz, diferente da árvore de pesquisa binária.

Todo nó do tipo folha possui a mesma profundidade entre eles e o nó da raiz.

Como outras árvores de pesquisa binária equilibradas, a complexidade do tempo para pesquisar, inserir e excluir é O(Log n).
Cada nó está 50% cheio, com exceção o nó da raiz.

Na figura a seguir, podemos observar um exemplo de árvore B+.

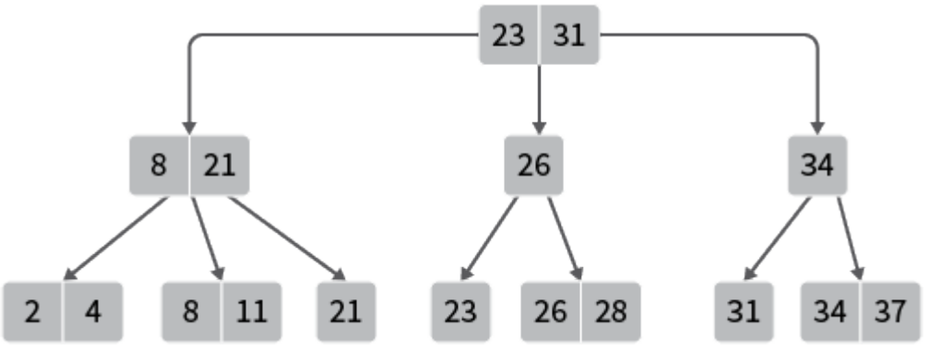


Figura 10 - Árvore B+

Dessa forma, a árvore B+ é muito semelhante a árvore B. As duas se equilibram automaticamente e possuem operações logarítmicas de inserção, localização e exclusão.
Uma árvore B+ pode ser vista como uma árvore B em que cada nó contém apenas chaves (não pares de chave-valor) e um nível adicional é inserido na parte inferior, com folhas vinculadas.

Síntese

Chegamos ao fim de mais uma unidade de estudos! Aqui, pudemos compreender as técnicas de pesquisa sequencial e binária, em que foi possível aprender os conceitos e a aplicabilidade de tabelas de dispersão. Também pudemos conhecer as principais árvores de buscas e suas complexidades.
Nesta unidade, você teve a oportunidade de:

- conhecer as técnicas de pesquisas sequencial e binária;
- entender sobre as tabelas de dispersão;
- compreender a respeito das árvores de buscas balanceadas;
- identificar as técnicas de manutenção das árvores de pesquisas;
- verificar a complexidade dos algoritmos de pesquisa.

Bibliografia

BARBOSA, W. A. *et al*. **DEG4Trees**: um jogo educacional digital de apoio ao ensino de estruturas de dados. Goiás: Universidade Federal de Goiás, 2015. Disponível em: https://www.researchgate.net/profile/Weider_Alves_Barbosa2/publication/265786605_DEG4Tree_Um_Jogo_Educacional_Digital_de_Apoio_ao_Ensino_de_Estruturas_de_Dados/links/55fc6c1808ae07629e0d3628.pdf. Acesso em: 11 out. 2019.

VIANA, G. V. R.; CINTRA, G. F.; NOBRE; R. H. **Pesquisa e ordenação de dados**. 2. ed. Ceará: EdeuECE, 2015.