UNIDADE I

Análise de Algoritmos

Prof. Me. Roberto Leminski

Análise de Algoritmos

- A disciplina Análise de Algoritmos tem como objetivo fornecer uma visão dos conceitos de complexidade computacional e algoritmos, bem como apresentar técnicas algorítmicas empregadas para a resolução de diferentes problemas práticos em diferentes áreas.

Trata-se de uma disciplina mais avançada, que apresenta a aplicação e análise de diferentes tipos de algoritmos, assumindo que o estudante possua diversos conhecimentos prévios na área de computação:

- Programação Estruturada;
- Programação Orientada a Objetos;
- Estrutura de Dados;
- Teoria dos Grafos;
- Fundamentos de Inteligência Artificial.

Algoritmos

- Um algoritmo é usualmente definido como sendo “uma sequência finita e ordenada de passos para resolver um problema ou executar uma tarefa”.
- Muitos autores reportam o primeiro algoritmo computacional à Ada Lovelace (1815 - 1852), que elaborou um conjunto de instruções para operar a Máquina Diferencial de Charles, em meados do século XIX.
- Pode causar surpresa descobrir que, quase dois séculos depois, ainda existem problemas matemáticos para os quais não se possui um algoritmo específico para sua resolução.
- Na verdade, existem problemas para os quais não se sabe se pode existir ou não uma solução algorítmica específica.

Fonte: https://commons.wikimedia.org/wiki/File:Ada_lovelace.jpg

Complexidade de algoritmos

Quando vamos implementar um algoritmo na forma de um programa de computador, algumas perguntas surgem:

- Esse algoritmo é o melhor para resolver o problema?
- Existem outras opções de algoritmo melhor?
- Qual o tempo de execução desse algoritmo?
- Quanta memória será consumida por esse algoritmo em execução?

- Do ponto de vista de tempo de execução, é impossível dar uma resposta absoluta de quanto tempo um algoritmo consome: a execução do mesmo algoritmo, sobre a mesma base de dados, será diferente de computador para computador.

Complexidade de algoritmos

- Podemos comparar dois algoritmos em como eles trabalham com uma mesma quantidade n de entradas.
- O algoritmo que realizar menos etapas de processamento para esse conjunto de entradas irá consumir menos tempo.
- O número de etapas realizadas em função do número de entradas que o algoritmo recebe é denominada Complexidade de Tempo do algoritmo.
- De forma análoga, o consumo de memória em função do número de entradas é denominado Complexidade de Espaço.
    - Dessa forma, podemos descrever o funcionamento de um algoritmo por meio de uma função matemática.

Redução assintótica

- Suponhamos uma função matemática com diversos parâmetros, por exemplo, uma função polinomial.
- Para valores suficientemente grandes da variável, o termo correspondente ao maior expoente dessa função será tão maior que os demais, que poderemos desconsiderar os termos menores.
- Por exemplo, a função $f(x) = x^3 + 3x^2 + 5x + 4$, para x suficientemente grande, pode ser aproximada como sendo $f(x) = x^3$
    - Na matemática, usamos o termo "assintótico" para indicar "para todos os valores suficientemente grandes" ou "tendendo ao infinito". Aqui usaremos para indicar um número muito grande de entradas em um algoritmo.
    - Assim, representamos a função de complexidade de um algoritmo por meio do termo mais significativo da sua função. Esse processo se chama redução assintótica.

Notação Grande-O

- Utilizando a redução assintótica, a notação para indicar a complexidade de tempo ou de espaço de um algoritmo é a notação Grande-O (ou “ozão”, ou ainda, em inglês, “big O”).
- A notação Grande-O permite que se compare a ordem de grandeza das complexidades de um algoritmo com as de outro algoritmo.
- Porém, em alguns casos, não é possível obter uma função matemática objetiva para a complexidade de um algoritmo.
- Assim, por exemplo, se dizemos que um algoritmo tem complexidade de tempo $O(n^3)$, isso significa que o número de operações desse algoritmo será da mesma ordem de grandeza que o cubo do número de entradas n.
    - A notação Grande-O pode ser usada comparativamente para relacionar dois algoritmos.
    - Por exemplo, relacionar dois algoritmos com funções A1 e A2 na forma A1 = O(A2) significa dizer que a função A1 não cresce mais rápido que a função A2, para n assintótico.

Notação Ômega e Theta

- Como visto no tópico anterior, dizer que f = O(g) significa que a função f(n) tem um consumo de tempo ou de memória menor que a função g(n).
- Para representarmos a ideia oposta, ou seja, indicar que um algoritmo tem uma complexidade maior que outro, usamos a notação Ômega.
- f = Ω(g) significa que a execução de f(n) irá realizar um número de operações (ou ter um consumo de memória) maior ou igual que g(n).
- A notação Ômega, diferente da notação Grande-O, não é utilizada para indicar a complexidade de um algoritmo individual, mas exclusivamente para estabelecer uma comparação entre dois algoritmos.
    - De forma semelhante, podemos utilizar a notação Theta para indicar que dois algoritmos possuem mesma ordem de complexidade.
    - f1 = Θ(f2) significa que f1 = O(f2) e f2 = O(f1), simultaneamente (e vice-versa).

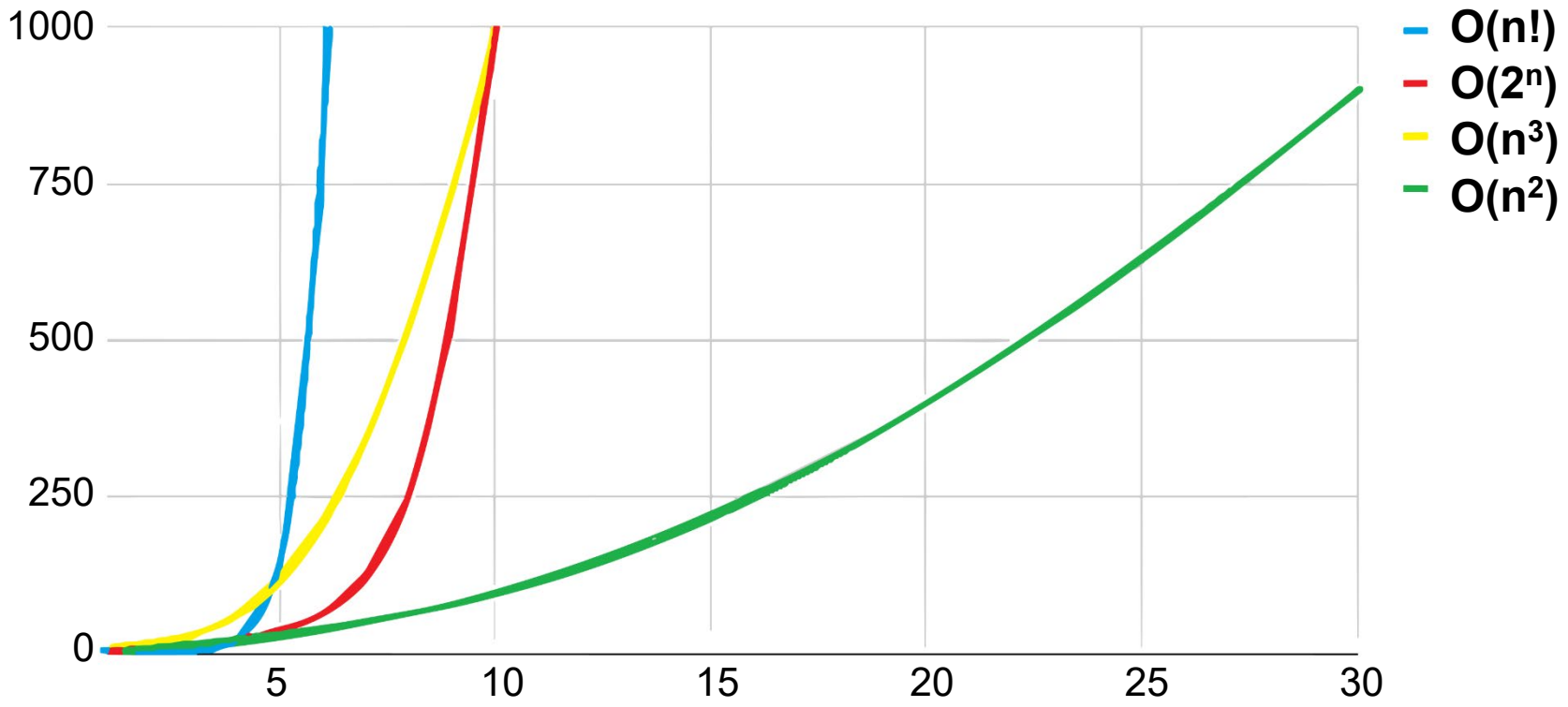
Principais funções de complexidade

- Algumas funções matemáticas são recorrentes ao analisarmos a complexidade de algoritmos.

Das piores (maior complexidade) para as melhores (menor complexidade), temos:

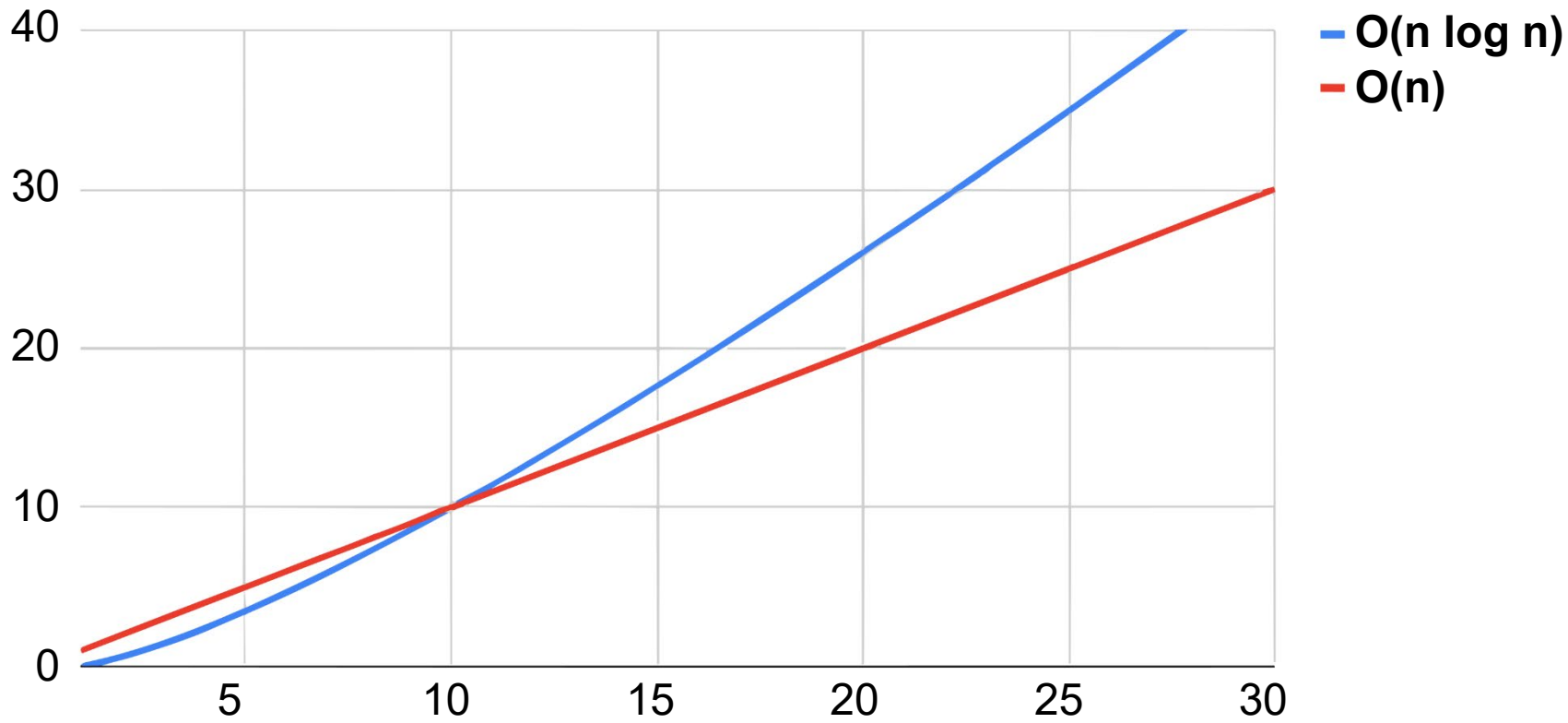
- Função Fatorial $O(n!)$
- Função Exponencial $O(2^n)$
- Função Polinomial $O(n^x)$
- Função n logaritmo de n $O(n \log n)$
- Função Linear $O(n)$
- Função Logarítmica $O(\log n)$
- Função Constante $O(1)$

Principais funções de complexidade



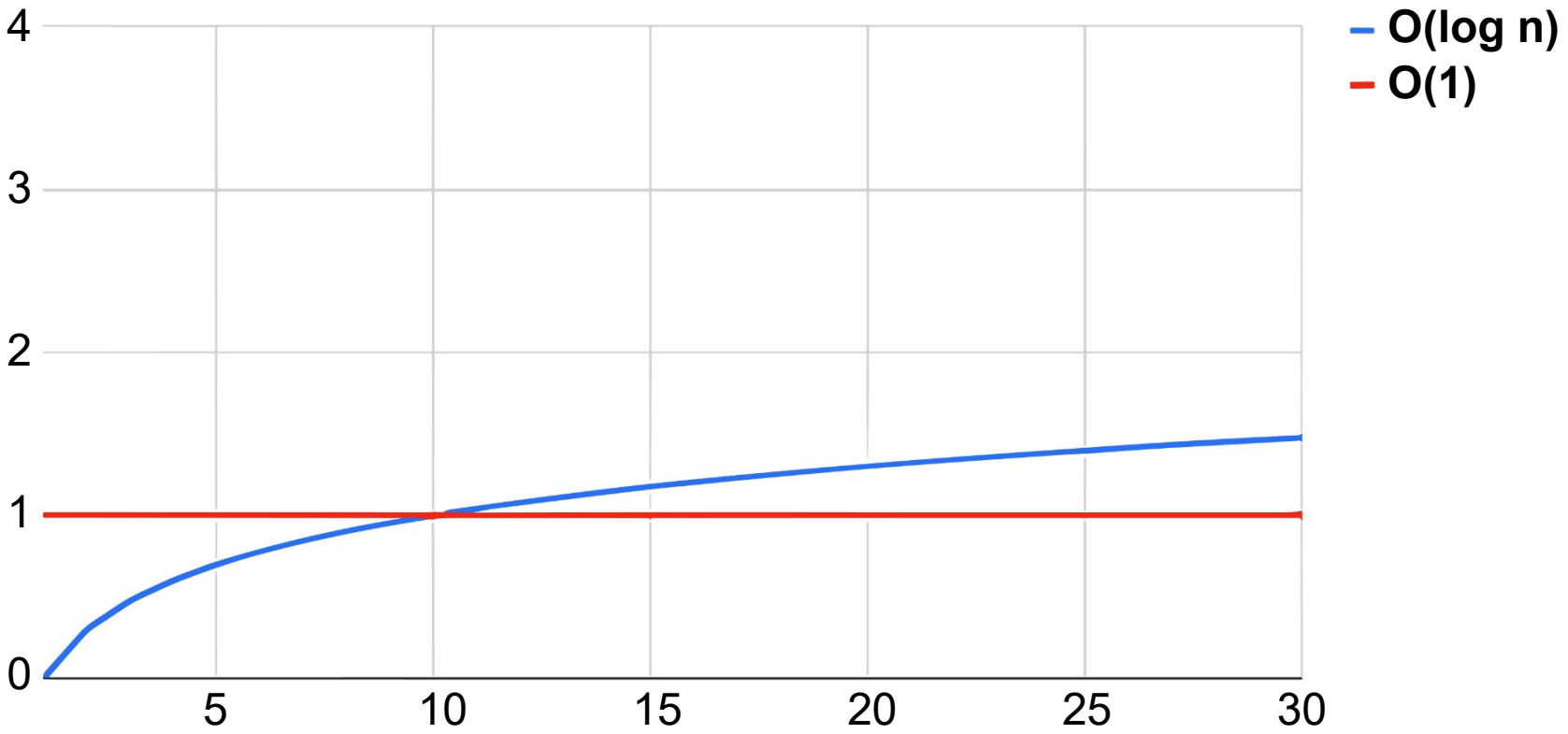
Fonte: livro-texto

Principais funções de complexidade



Fonte: livro-texto

Principais funções de complexidade



Fonte: livro-texto

Interatividade

Um algoritmo recebe dois vetores numéricos como entrada, um com m elementos e outro com n elementos (sendo m > n), e produz todos os pares possíveis compostos por um elemento de m seguido por um elemento de n. Qual a complexidade de espaço desse algoritmo?

a) O(m).
b) $O(m^2)$.
c) O(m + n).
d) O(mn).
e) O(n).

Resposta

Um algoritmo recebe dois vetores numéricos como entrada, um com m elementos e outro com n elementos (sendo m > n), e produz todos os pares possíveis compostos por um elemento de m seguido por um elemento de n. Qual a complexidade de espaço desse algoritmo?

a) O(m).
b) $O(m^2)$.
c) O(m + n).
d) O(mn).
e) O(n).

Espaço = m + n + mn

Algoritmos recursivos

- Em muitas situações, mais de um algoritmo pode ser empregado para resolver um determinado problema.
- Esses diferentes algoritmos muitas vezes apresentam complexidades diferentes (de tempo ou de espaço) para chegar no seu resultado.
- Uma situação bastante comum disso é uso de algoritmos iterativos versus algoritmos recursivos.
- Um algoritmo recursivo consiste em reduzir, sucessivamente, um problema em um problema menor ou mais simples, até que seja possível resolver alguma parte do problema reduzido de forma direta.
    - Podemos dizer que um algoritmo recursivo instancia a si mesmo diversas vezes para chegar até a solução final do problema.
    - Todo algoritmo recursivo possui uma versão equivalente iterativa.

Algoritmos recursivos

Exemplo: duas funções em Python para o cálculo de fatorial:

- def fat_iterativo (N):
  fat = 1
  for i in range (1,N+1):
  fat*=i
  return fat

- def fat_recursivo (N):
  if N <=1:
  return 1
  else:
  return N * fat_recursivo (N-1)

Algoritmos recursivos

- Ambas funções vão realizar a mesma quantidade de multiplicações; assim, a complexidade de tempo para a realização das operações é a mesma O(n).
- A função recursiva vai produzir uma pilha de dados: o valor de n será armazenado enquanto é calculado o valor do fatorial de n-1; o valor de n-1 terá que também ser empilhado enquanto se calcula o fatorial de (n -1) -1, e assim por diante.
- Essa pilha também significa um maior consumo de memória: a complexidade de espaço da função iterativa será, assim O(1), enquanto a complexidade de espaço da função recursiva será O(n).
- Funções recursivas geram códigos menores, o que é um ganho para o desenvolvedor.
    - Porém, o preço do uso dessas funções pode ser um aumento das complexidades de tempo e espaço do algoritmo.

Estratégia da divisão e conquista

Uma estratégia muita empregada para solucionar problema matemáticos e de algoritmos é a estratégia de divisão e conquista, que possui três etapas:

- Divide-se o problema original em subproblemas menores, que são instâncias menores do mesmo problema.
- Esses subproblemas são resolvidos, usualmente de forma recursiva.
- As respostas dos subproblemas são adequadamente combinadas para se obter a resposta do problema original.

- Essa metodologia acaba executando três frentes de trabalho, coordenadas pela estrutura recursiva central do algoritmo: divisão do problema, resolução das partes e concatenação das respostas parciais.

Estratégia da divisão e conquista

- Um exemplo dessa metodologia é a busca binária em um arquivo ordenado com n registro; primeiro, dividimos o arquivo em duas partes de tamanho n/2. Em seguida, reaplicamos recursivamente o algoritmo na metade adequada.
- Assim, o algoritmo é reaplicado em uma estrutura cada vez menor.
- A complexidade de tempo desse algoritmo é O(log n).

0	2	6	7	10	12	15	20	25	30	45

Procurando: 3

Estratégia da divisão e conquista

- Um exemplo dessa metodologia é a busca binária em um arquivo ordenado com n registro; primeiro, dividimos o arquivo em duas partes de tamanho n/2. Em seguida, reaplicamos recursivamente o algoritmo na metade adequada.
- Assim, o algoritmo é reaplicado em uma estrutura cada vez menor.
- A complexidade de tempo desse algoritmo é O(log n).

0	2	6	7	10	12	15	20	25	30	45

Procurando: 3

Estratégia da divisão e conquista

- Um exemplo dessa metodologia é a busca binária em um arquivo ordenado com n registro; primeiro, dividimos o arquivo em duas partes de tamanho n/2. Em seguida, reaplicamos recursivamente o algoritmo na metade adequada.
- Assim, o algoritmo é reaplicado em uma estrutura cada vez menor.
- A complexidade de tempo desse algoritmo é O(log n).

0	2	6	7	10	12	15	20	25	30	45

Procurando: 3

Programação dinâmica

- Programação dinâmica é uma metodologia para a construção de algoritmos que tem como foco encontrar, sempre que possível, a solução melhor (com menos complexidade de tempo, normalmente) para situações envolvendo as diferentes combinações possíveis de um conjunto.

Simplificando, se temos uma série de operações que precisam ser feitas, qual seria a melhor ordem para realizá-las?

- Existem na programação dinâmica duas abordagens: Top-Down e Bottom-up.

Programação dinâmica

- A abordagem Top-Down, como diz o nome, parte de cima para baixo, realizando um processo recursivo. A função recursiva para o cálculo do fatorial apresentada é um exemplo dessa abordagem.

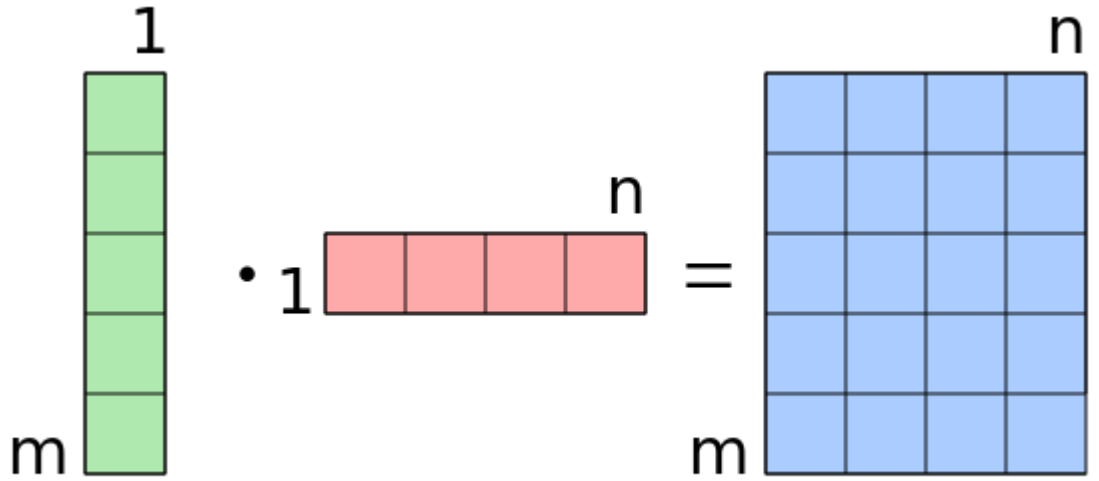
- Na abordagem Bottom-up, (de baixo para cima), o problema é dividido em subproblemas menores, que vão sendo calculados do menor para o maior. Nessa abordagem, sabemos que cada iteração anterior já foi resolvida, logo não precisamos ficar armazenando os resultados parciais toda vez como na abordagem Top-Down. A função iterativa apresentada anteriormente para o cálculo do fatorial é um exemplo dessa abordagem.

Programação dinâmica

- Um exemplo clássico de Programação Dinâmica utilizando a abordagem Bottom-Up é o Problema da Multiplicação Sequencial de Cadeias de Matrizes (ou Matrix Chain Ordering Problem, MCOP).
- Esse é um problema de otimização no qual se busca determinar a melhor forma de realizar o produto de uma dada sequência de matrizes.
- O problema aqui não é realizar a multiplicação em si, mas determinar a melhor sequência em que essa operação deve ser realizada.

Programação dinâmica

- Um exemplo clássico de Programação Dinâmica utilizando a abordagem Bottom-Up é o Problema da Multiplicação Sequencial de Cadeias de Matrizes (ou Matrix Chain Ordering Problem, MCOP).
- Esse é um problema de otimização no qual se busca determinar a melhor forma de realizar o produto de uma dada sequência de matrizes.
- O problema aqui não é realizar a multiplicação em si, mas determinar a melhor sequência em que essa operação deve ser realizada.



Fonte: https://commons.wikimedia.org/wiki/File:Matrix_multiplication_qtl3.svg

Programação dinâmica

- Exemplo: Consideremos as seguintes matrizes e que desejamos calcular A x B x C x D

- A_{5x10}
- B_{10x15}
- C_{15x20}
- D_{20x30}

- Temos cinco possibilidades de associar essas matrizes para realizar essa multiplicação.

Programação dinâmica

- ((AB)C)D: (5 x 10 x 15) + (5 x 15 x 20) + (5 x 20 x 30) = 5.250 operações
- ((A(BC))D): (10 x 15 x 20) + (5 x 10 x 20) + (5 x 20 x 30) = 7.000 operações
- (AB)(CD): (5 x 10 x 15) + (15 x 20 x 30) + (5 x 15 x 30) = 12.000 operações
- A((BC)D): (10 x 15 x 20) + (10 x 20 x 30) + (5 x 10 x 30) = 10.500 operações
- A(B(CD)): (15 x 20 x 30) + (10 x 15 x 30) + (5 x 10 x 30) = 15.000 operações

- O pior caso executa quase três vezes mais operações que o segundo.
- Assim, é interessante executar um algoritmo antes do cálculo para verificar qual a melhor forma de realizar o cálculo.

Interatividade

Qual das características a seguir **não** se aplica à implementação de algoritmos recursivos?

a) Pode ser utilizada quando não existe um algoritmo iterativo equivalente.
b) Pode resultar em um código com maior complexidade de tempo.
c) Pode resultar em um código com maior complexidade de espaço.
d) É empregada em metodologias de Programação Dinâmica.
e) É utilizada muitas vezes para reduzir o tamanho do código.

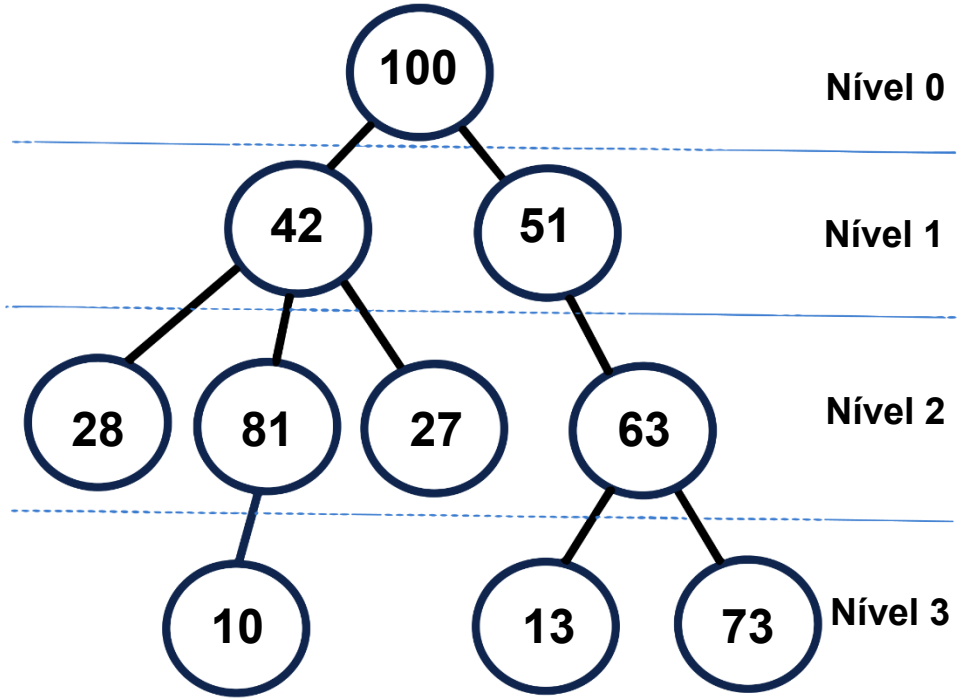
Resposta

Qual das características a seguir **não** se aplica à implementação de algoritmos recursivos?

a) Pode ser utilizada quando não existe um algoritmo iterativo equivalente.
b) Pode resultar em um código com maior complexidade de tempo.
c) Pode resultar em um código com maior complexidade de espaço.
d) É empregada em metodologias de Programação Dinâmica.
e) É utilizada muitas vezes para reduzir o tamanho do código.

Árvores

- Em estruturas de dados, árvores são uma estrutura não linear para armazenar dados de forma hierárquica e sequencial.
- O primeiro elemento, também chamado de nó, é denominado raiz, sendo o único elemento da estrutura que não possui nó anterior.
- O nó imediatamente anterior a um nó é chamado de nó pai, e quaisquer nós que sejam imediatamente subsequentes a ele são denominados de nós filhos.
- Um nó sem filhos é chamado de folha.



Fonte: livro-texto

Árvores binárias

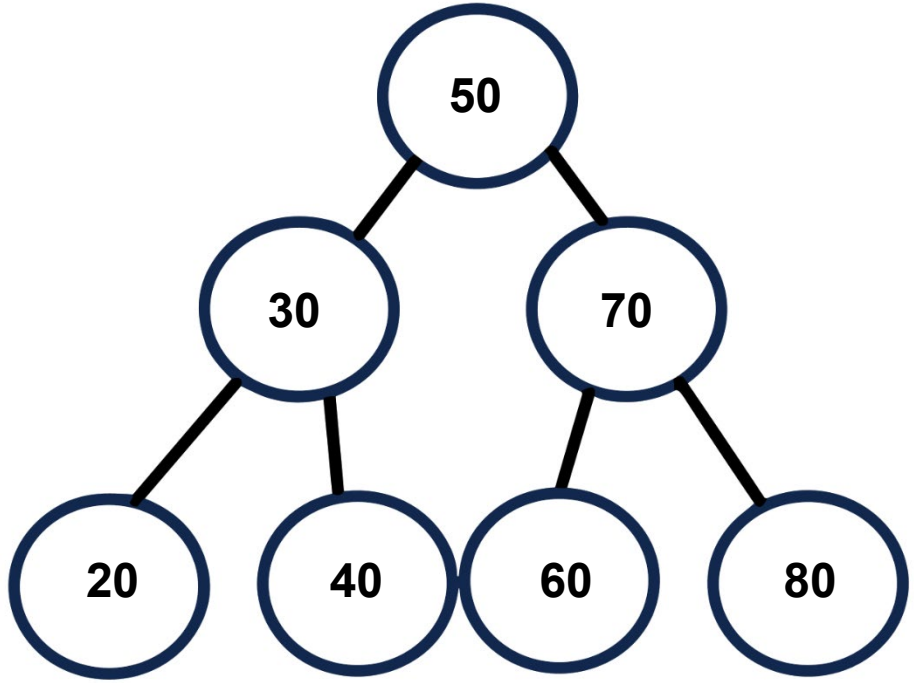
- As diferentes linguagens de programação oferecem suas ferramentas para construirmos e trabalharmos com essa estrutura de dados.
- Dentre essas estruturas, existe um subtipo denominado árvore binária.
- Esse tipo de árvore apresenta como única característica o fato de cada nó poder ter, no máximo, dois nós filhos.

Uma árvore binária de busca é uma estrutura de dados na forma de uma árvore binária na qual o posicionamento dos seus elementos se baseia nas seguintes regras:

- O filho à esquerda de um nó sempre será menor que o seu pai.
- O filho à direita de um nó será sempre maior que seu pai.

Árvores binárias de busca

- Uma árvore binária de busca não necessita ser completa ou balanceada.
- O objetivo da construção dessa estrutura é facilitar o processo de busca binária em um conjunto de dados.
- A busca binária é um algoritmo para encontrar um elemento em uma estrutura de dados ordenada.



Fonte: livro-texto

Algoritmos de busca em árvores

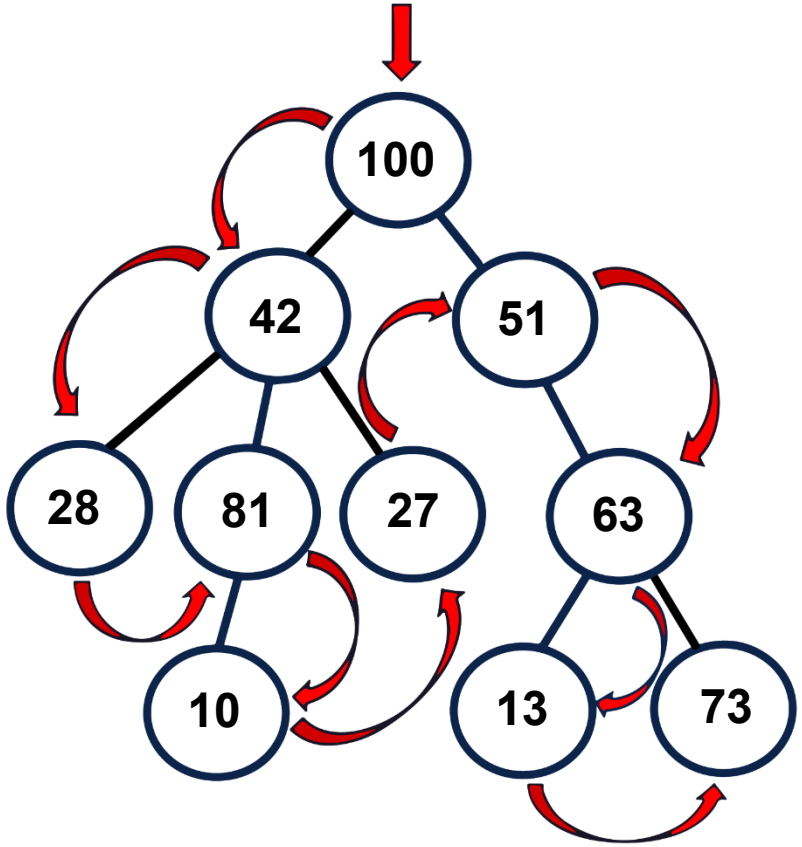
- Existem dois algoritmos principais de busca não heurística em árvores: busca em profundidade e busca em largura.

- A busca em profundidade realiza uma busca não orientada que percorre a árvore da raiz e se aprofunda até chegar em uma folha.
- Então, o algoritmo retrocede até encontrar uma nova ramificação e continua a busca.
- Em termos de complexidade de tempo, devido ao fato do algoritmo retroceder ao longo da estrutura, passamos mais de uma vez por vários nós. Sendo n o número de nós, a complexidade de tempo pode ser definida como sendo O(2n).
    - Porém, o uso desse algoritmo de busca implica a criação de uma lista que fará o registro dos nós já visitados. Dessa forma, a complexidade de espaço desse algoritmo é, no pior caso, O(n).

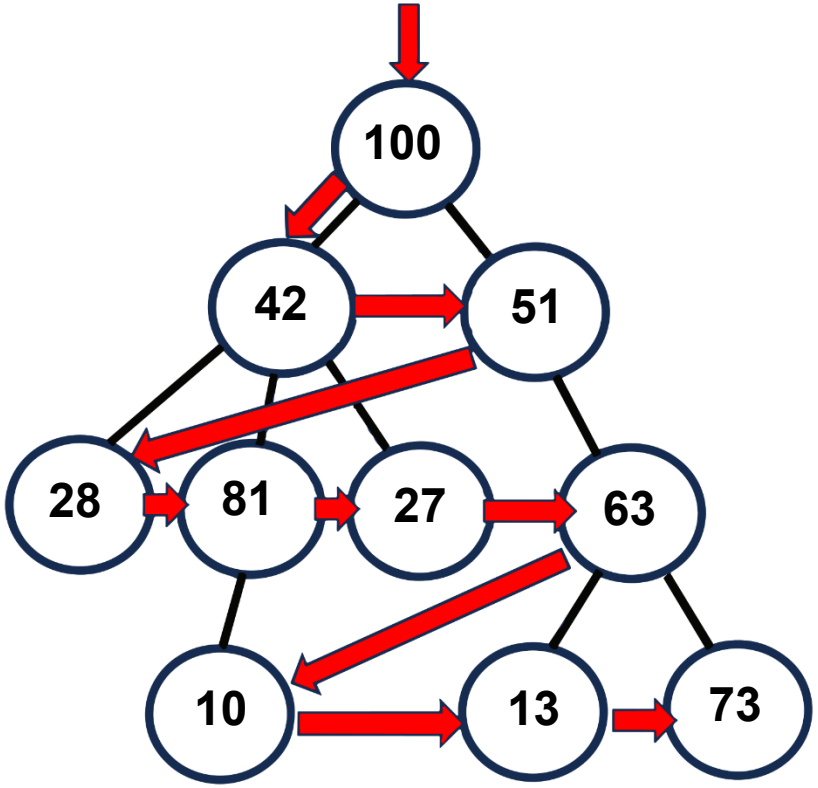
Algoritmos de busca em árvores

- A busca em largura também realiza uma busca não orientada que percorre a árvore.
- Porém, a árvore é percorrida por níveis: primeiro o nível 0, depois todos os nós do nível 1, depois do nível 2, e assim por diante até chegar aos nós do último nível da árvore.
- Em termos de complexidade de tempo, não temos um retrocesso na estrutura, como na busca em profundidade. Assim, cada nó será percorrido apenas uma vez; dessa forma a complexidade de tempo será O(n).
- O uso desse algoritmo de busca implica a criação de uma fila que irá definir a ordem em que os nós serão visitados; essa lista conterá todos os nós presentes na estrutura. Assim, a complexidade de espaço desse algoritmo é O(n).

Algoritmos de busca em árvores



Trajeto: 100 42 28 81 10 27 51 63 13 73



Trajeto: 100 42 51 28 81 27 63 10 13 73

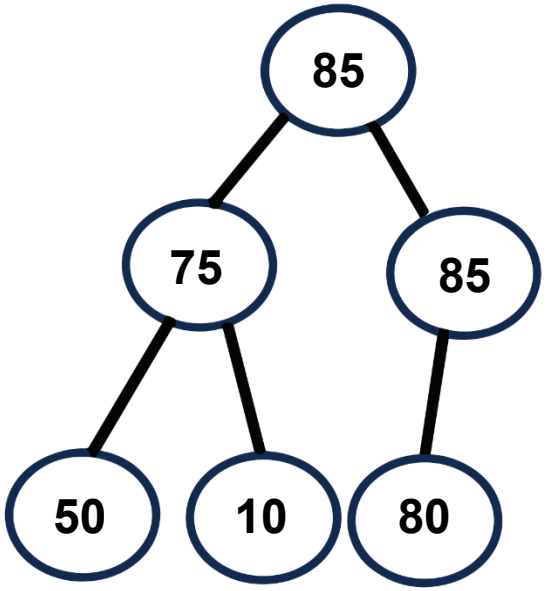
Filas de prioridades e Heaps

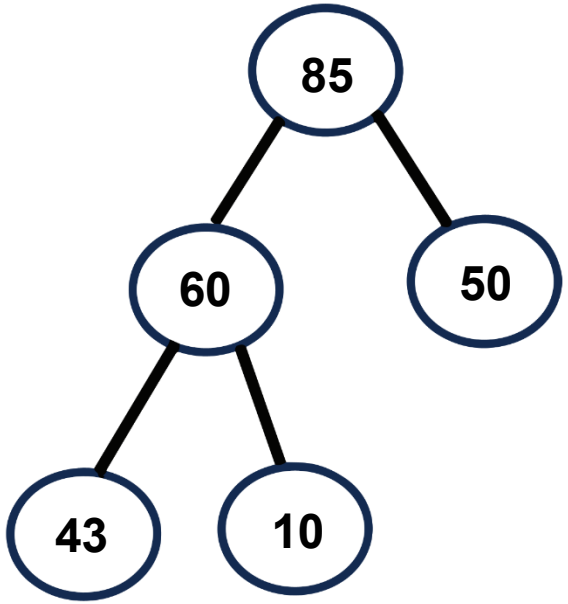
- Em uma fila de prioridades, cada elemento possui uma prioridade e, ao retirarmos um elemento da estrutura, deve-se sempre retirar primeiro o elemento de maior prioridade.
- Se pensarmos nessa situação como uma lista comum, temos duas possibilidades de se lidar com ela.
- Assim, teremos uma estrutura com complexidade de tempo constante para inserir o elemento e linear para remover (ou o oposto, dependendo da abordagem).
- Uma solução para esse tipo de situação é a estrutura chamada Heap.
    - Heap, em inglês, significa "amontoar“: os elementos são “amontoados” em uma árvore binária, com os elementos de maior prioridade (primeiros a serem retirados) mais próximos à raiz.
    - Essa estrutura possibilita que a inserção de novos elementos e a extração do elemento de maior prioridade sejam realizadas com complexidade de tempo O(log n).

Filas de prioridades e Heaps

Um heap é uma árvore binária completa, mas também possui duas outras características necessárias:

- O valor de um determinado nó da árvore sempre será maior ou igual aos valores de seus dois filhos;
- Um heap é uma árvore binária completa ou quase completa da esquerda para a direita, o que significa que existirá uma ordem específica para a inserção dos elementos na estrutura: sempre serão inseridos filhos no nó mais à esquerda possível, só depois passando para o próximo nó imediatamente à direita.

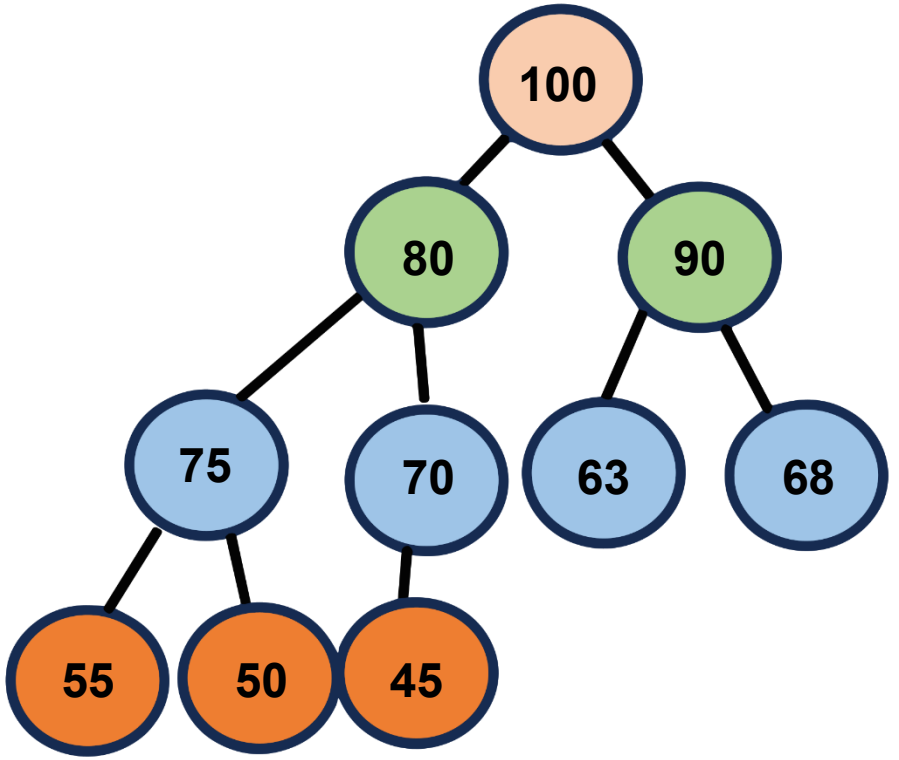




Fonte: livro-texto

Filas de prioridades e Heaps

- O fato do heap ser uma árvore completa também possibilita que usemos uma estrutura de dados linear (vetor ou lista) para representá-lo.
- Cada elemento do heap vai ocupar uma posição no vetor como se fosse resultado de uma busca em largura.



Heap =

100	80	90	75	70	63	68	55	50	45

Fonte: livro-texto

Filas de prioridades e Heaps

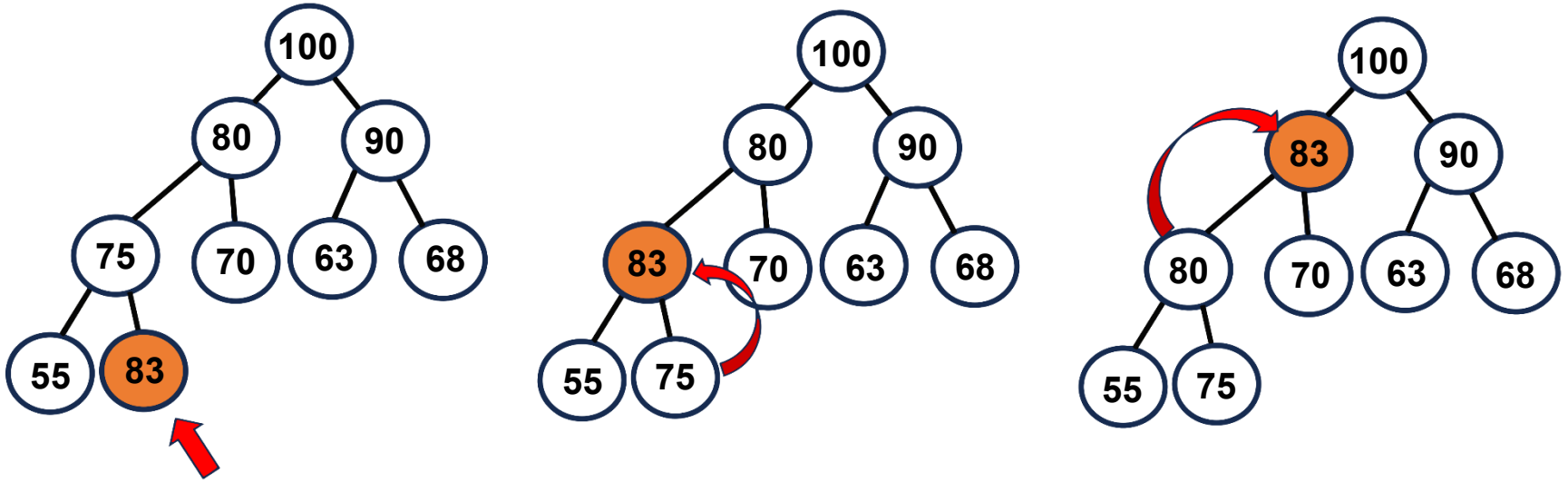
- Para podermos navegar em uma estrutura em árvore, é necessário que, a partir de cada nó, seja possível acessar seu nó pai e seus nós filhos.

Ao representarmos isso na forma de um vetor, é necessário estabelecer uma relação entre o índice do nó atual e os índices dos nós que se conectam com ele. Essa relação é a seguinte:

- O filho à esquerda de um nó será dado pela fórmula 2 x índice + 1;
- O filho à direita de um nó será dado pela fórmula 2 x (índice + 1);
- O pai de um nó será dado pela fórmula (índice − 1) / 2, arredondado para baixo, se fracionário.

Inserção de elemento em um Heap

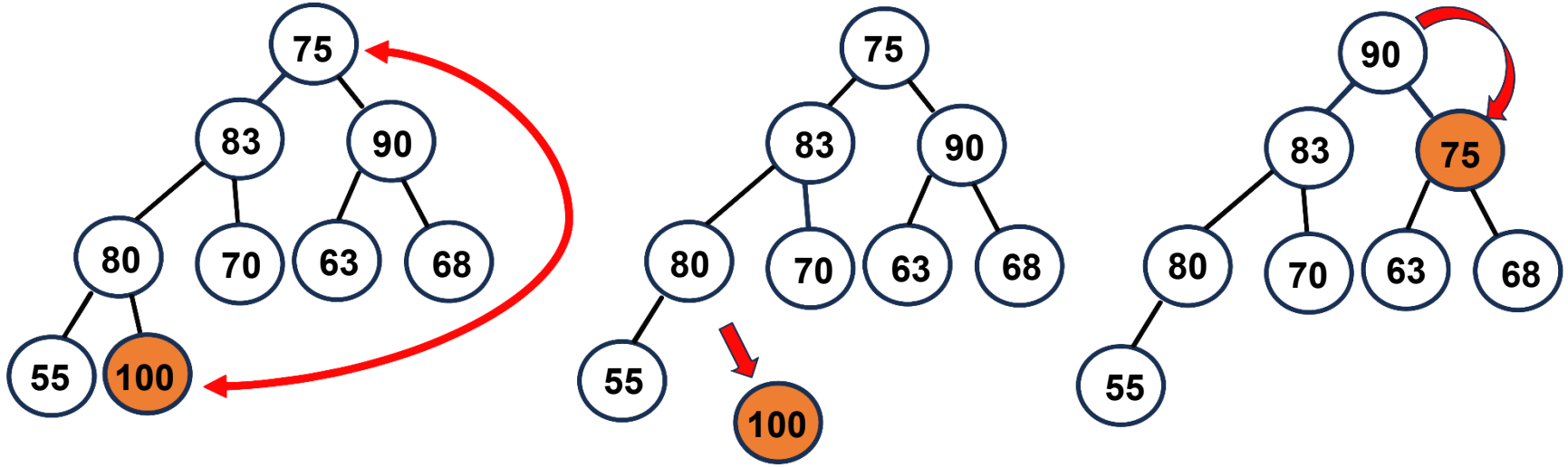
- A inserção de um elemento na estrutura, como foi dito, será sempre na primeira posição livre da árvore, o que significa a última posição do vetor.
- Porém, a regra de que um nó deve sempre ser menor ou igual ao seu pai deve ser respeitada para que continuemos com a estrutura do heap.
- Assim, o elemento que foi inserido deve ser comparado com o elemento imediatamente acima (seu nó pai): se ele for maior, devem trocar de posição.



Fonte: livro-texto

Remoção de elemento em um Heap

- Já a remoção será sempre do elemento da raiz, que é o elemento de maior prioridade.
- Porém, a simples remoção do elemento irá distorcer totalmente a estrutura.
- A metodologia utilizada para resolver esse problema é a seguinte: o elemento da raiz será trocado de posição com o último elemento da estrutura.
- Assim, apenas o elemento presente na raiz estará fora de posição e será reposicionado.
- O método de reposicionamento do elemento da raiz é denominado heapify.



Fonte: livro-texto

Heaps – Considerações finais

- Para transformarmos um vetor qualquer em um heap, aplicaríamos o método heapify sucessivamente em todos os nós começando pelo nó pai da última folha e seguindo em direção à raiz do heap (primeiro elemento do vetor).
- Heaps e filas de prioridades são partes integrantes de outros algoritmos, principalmente para algoritmos envolvendo grafos.
- Linguagens como o Python e o Java apresentam métodos específicos para a construção e trabalho com heaps.

Interatividade

Em relação a um heap, qual afirmação está correta?

a) Por se tratar de uma estrutura em árvore, necessariamente fará uso de ponteiros.
b) Por possuir uma complexidade de tempo de O(log n) para inserção e remoção de elementos, possui maior tempo de execução do que uma lista convencional.
c) A remoção de um elemento do heap implica no reposicionamento de outros elementos para manter o formato da estrutura de dados.
d) A inserção de um elemento sempre se dará já na posição que o elemento ocupará.
e) Embora seja uma árvore completa, não precisa ser necessariamente uma árvore binária.

Resposta

Em relação a um heap, qual afirmação está correta?

a) Por se tratar de uma estrutura em árvore, necessariamente fará uso de ponteiros.
b) Por possuir uma complexidade de tempo de O(log n) para inserção e remoção de elementos, possui maior tempo de execução do que uma lista convencional.
c) A remoção de um elemento do heap implica no reposicionamento de outros elementos para manter o formato da estrutura de dados.
d) A inserção de um elemento sempre se dará já na posição que o elemento ocupará.
e) Embora seja uma árvore completa, não precisa ser necessariamente uma árvore binária.

Algoritmos sobre cadeias de dados

- Por cadeias de dados entende-se qualquer estrutura de dados sequencial e linear, tais como vetores, listas, pilhas e filas.

Independentemente do tipo de cadeia de dados, algumas operações são recorrentes em todas elas. Em termos de complexidade de tempo:

- Preenchimento de uma cadeia inicialmente vazia: O(n).
- Inserção/remoção de elemento na extremidade da cadeia: O(1).
- Inserção/remoção de elemento em estrutura ordenada: O(log n).
- Substituir um elemento de posição conhecida: O(1).
    - Busca de elemento em estrutura não ordenada: O(n).
    - Busca de elemento em estrutura ordenada: O(log n).
    - Copiar uma cadeia: O(n).
    - Concatenar duas cadeias: O(n), sendo n a dimensão da cadeia que está sendo inserida.

Algoritmos sobre cadeias de dados

- Outras operações podem ser realizadas em uma cadeia, tais como: trocar a posição de dois elementos da cadeia, inserir ou remover um trecho de uma cadeia etc., mas para mantermos a lista mais objetiva, reduzimos às operações básicas envolvendo inserir, remover, copiar, substituir e localizar.
- Como uma cadeia tem um tamanho constante (seja ela estática, como um vetor, ou dinâmica, como no caso de uma lista), a complexidade de espaço envolvida nessas operações será sempre O(n).
- Pilhas e filas, apesar de seu funcionamento diferente entre si (não iremos descrever detalhadamente aqui o funcionamento dessas estruturas), possuem as complexidades das suas operações equivalentes às descritas.

Algoritmos de ordenação

- Um algoritmo de ordenação é um algoritmo que organiza os dados de uma cadeia, de forma crescente ou decrescente.
- Todos os algoritmos de ordenação se baseiam em dois elementos: comparações e movimentações (trocas de posições) entre elementos das cadeias.

Assim, teremos duas complexidades envolvidas nesses algoritmos, sendo n o número de registros na cadeia que será ordenada:

- Número de comparações C(n) entre chaves.
- Número de movimentações M(n) dos elementos.

Algoritmos de ordenação

- Um aspecto importante sobre esses algoritmos é sua estabilidade.
- Um algoritmo de ordenação é estável se os valores iguais mantêm suas posições originais relativas após o processo de ordenação.
- Caso essa ordem seja modificada, o algoritmo é chamado de instável.

12	99	10	53	10	82	75	10	67	39

Dados Originais

10	10	10	12	39	53	67	75	82	99

Algoritmo de ordenação estável

10	10	10	12	39	53	67	75	82	99

Algoritmo de ordenação instável

Fonte: livro-texto

Algoritmos de ordenação

- A tabela a seguir ilustra alguns algoritmos de ordenação e suas complexidades.
- Vale uma observação sobre o algoritmo Shell Sort: criado por Donald Shell em 1959, é uma variante do algoritmo de ordenação por inserção. Ainda hoje, a função de complexidade de tempo do algoritmo é desconhecida: estima-se que seja algo entre a linear e a quadrática sua função de complexidade.

Algoritmo	Comparações		Movimentações		Estabilidade	Espaço
	Melhor caso	Pior caso	Melhor caso	Pior caso		
Bubble Sort	$O(n^2)$		$O(n^2)$		Estável	O(1)
Insertion Sort	O(n)	$O(n^2)$	O(n)	$O(n^2)$	Estável	O(1)
Selection Sort	$O(n^2)$		O(n)		Instável	O(1)
Quick Sort	O(n log n)	$O(n^2)$	NA		Instável	O(n)
Merge Sort	O(n log n)		NA		Estável	O(n)
Shell Sort	$O(n^{1.25})$ ou $O(n\ (\ln n)^2)$ *		NA		Instável	O(n)

NA: Não aplicável
* : estimado

Fonte: livro-texto

Algoritmos de localização de cadeias

- Um problema comum ao trabalharmos com cadeias, principalmente com cadeias de textos, é localizar a ocorrência (ou não) de uma cadeia menor dentro de uma maior.
- O caso mais comum é localizar uma palavra ou frase em um texto maior.
- O algoritmo de busca mais simples possível consiste em percorrer uma cadeia T (o string no qual procuramos o texto) em busca de uma cadeia chamada de padrão P (sendo P menor que T).
- Na versão mais simples dessa busca, percorremos T caractere a caractere até achar uma correspondência com o primeiro caractere de P.

Algoritmos de localização de cadeias

- O processo de percorrer T é interrompido, e checa-se a correspondência dos caracteres subsequentes de P com os subsequentes de T.
- Caso a correspondência seja total, encontrou-se uma ocorrência de P; caso contrário, reinicia-se o processo de percorrer T do ponto onde foi interrompido.
- Esse algoritmo terá uma complexidade de tempo O(pt), em que p é o número de caracteres de P e t é o número de caracteres de T. Essa metodologia é chamada por alguns autores de busca ingênua.

Algoritmos de localização de cadeias

- Um dos principais algoritmos de busca de expressões é o Algoritmo de Boyer-Moore, que recebe esse nome em função de seus criadores Robert S. Boyer e J Strother Moore, que o desenvolveram em 1977.
- O algoritmo de Boyer-Moore realiza uma busca mais eficiente, realizando saltos ao longo de T baseados no pré-tratamento de P.
- O processo de execução do algoritmo começa com o alinhamento dos caracteres iniciais de P e T.
- Se houver uma coincidência entre o último caractere de P e um caractere de T, verificam-se a correspondência entre os caracteres anteriores de P e T, de trás para frente.

Algoritmos de localização de cadeias

Essa verificação nesse algoritmo se baseia em dois princípios:

- Regra do Sufixo Correto: supondo que haja uma correspondência do último caractere de P com um de T, como foi dito, iremos percorrer P até confirmarmos que P está contido em T ou até que ocorra um caractere diferente. Os caracteres que corresponderam serão denominados de sufixo correto.
- Regra do Caractere Ruim: se no caso anterior houver uma correspondência parcial do final de P, mas ao avançar em direção ao começo ocorrer um caractere discrepante, o caractere de P que não correspondeu é chamado de caractere ruim.
    - Caso ocorra o caractere ruim, verifica-se se há a existência do sufixo correto não precedido pelo caractere ruim em algum ponto anterior de P.
    - Se houver, realiza-se um salto de P para que haja a correspondência desse novo sufixo; se não, o salto se dá de forma que o primeiro caractere de P alinhe-se com o caractere imediatamente ao que o último estava alinhado.

Algoritmos de localização de cadeias

- Se o padrão não ocorrer no texto, a complexidade de tempo do algoritmo é O(pt) no pior caso, que seria a ocorrência de sufixo longo, o que significa que os saltos serão menores.
- Já no caso de ocorrência de P em T, a complexidade cai para O(p + t).
- Algo interessante sobre esse algoritmo é que quanto maior for P, maiores serão os saltos e mais rápida será execução do algoritmo.

Algoritmos de localização de cadeias

- Um outro algoritmo para realizar essa busca é o Algoritmo de Knuth–Morris–Pratt (chamado de KMP).
- Esse algoritmo se utiliza da seguinte técnica: quando se encontra parte de P em T, mas verifica-se que não há uma correspondência completa, a palavra procurada contém em si a informação necessária para determinar onde começar a próxima comparação.
- Esse algoritmo foi inventado por Donald Knuth e Vaughan Pratt e, independentemente, por James Morris em 1977. Os três se juntaram e publicaram em conjunto, daí o algoritmo levar o nome dos três autores.
    - A principal característica desse algoritmo está no pré-processamento da cadeia, que irá fornecer a informação necessária para definirem em que ponto a busca recomeça em caso de incompatibilidade.
    - Assim, o algoritmo não reexamina os caracteres que já foram vistos anteriormente, reduzindo o número de comparações.

Algoritmos de localização de cadeias

- O algoritmo KMP possui uma complexidade de tempo O(n).
- Porém, a complexidade envolvida no pré-processamento e a memória consumida fazem com que na prática esse algoritmo seja mais eficiente para buscas em situações com alfabetos menores.
- Esse algoritmo tem tido como principal aplicação prática a busca de sequências específicas em base de DNA, já que estamos lidando com alfabeto de apenas quatro letras (A, C, G e T).

Interatividade

A inserção e a remoção de um elemento em uma estrutura de dados do tipo fila possuem complexidade de tempo, respectivamente de:

a) O(n) e O(n).
b) O(1) e O(1).
c) O(1) e O(log n).
d) O(log n) e O(n).
e) O(n) e O(1).

Resposta

A inserção e a remoção de um elemento em uma estrutura de dados do tipo fila possuem complexidade de tempo, respectivamente de:

a) O(n) e O(n).
b) O(1) e O(1).
c) O(1) e O(log n).
d) O(log n) e O(n).
e) O(n) e O(1).

Referências

- DASGUPTA, S.; PAPADIMITRIOU, C.; VAZIRANI, U. *Algoritmos*. Porto Alegre: McGraw-Hill, 2009.
- CORMEN, T. *Algoritmos*: teoria e prática. Rio de Janeiro: LTC, 2012.
- CORMEN, T. *Desmistificando algoritmos*. Rio de Janeiro: Campus, 2013.

ATÉ A PRÓXIMA!