Lista de exercícios – Organização de Dados II – GABARITO SUGERIDO
Prof. Vinícius Gusmão

1) Simule a execução de uma busca interpolada pelo elemento 82 no array abaixo, indicando as posições do array que foram lidas, a cada passo.

2	5	7	10	25	27	28	33	50	52	60	61	80	88	99

M = L + ceil{(R – L) * (82 – A[L]) / (A[R] – A[L])}

início:

L = 0, R = 14
A[0] = 2, **A[14]** = 99

primeiro passo:

M = 0 + ceil{(14 – 0) * (82 – 2) / (99 – 2)} = 12
A[12] = 80 < 82
L = 12, R = 14

segundo passo:

M = 12 + ceil{(15 – 12) * (82 – 80) / (99 – 80)} = 13
A[13] = 88 > 82
L = 12, R = 13

R – L = 1 indica que não há nenhuma posição não-consultada entre elas, portanto 82 não pertence ao array. O algoritmo para após dois passos além do passo inicial, num total de quatro leituras no array (indicadas em negrito): as duas iniciais + uma por passo.

2) Suponha que o array acima passe a ser utilizado por uma aplicação que emprega busca sequencial auto-organizável. Suponha que a seguinte sequência de consultas seja efetuada repetidas vezes:

27 88 27 88 27 88 27 88 5 99
27 88 27 88 27 88 27 88 5 99
...

a) Qual o número médio de leituras e de escritas no array (por consulta) quando o número de consultas tende a infinito, quando a política de *move-to-front* é utilizada?

Situação do array após a *k*-ésima consulta a um "99", para *k* muito grande:

99	5	88	27	2	7	10	25	28	33	50	52	60	61	80

Sejam agora as consultas efetuadas até o próximo "99" (perfazendo um ciclo completo de consultas):

consulta ao "27": 4 leituras (99, 5, 88, 27) + 4 escritas (os 4 primeiros elementos do array precisam ser re-escritos), deixando assim o array:

27 99 5 88 2 7 10 25 28 33 50 52 60 61 80

consulta ao "88": 4 leituras (27, 99, 5, 88) + 4 escritas (idem)

88 27 99 5 2 7 10 25 28 33 50 52 60 61 80

consulta ao "27": 2 leituras (88, 27) + 2 escritas

27 88 99 5 2 7 10 25 28 33 50 52 60 61 80

consulta ao "88": 2 leituras (27, 88) + 2 escritas

88 27 99 5 2 7 10 25 28 33 50 52 60 61 80

consulta ao "27": 2 leituras (88, 27) + 2 escritas

27 88 99 5 2 7 10 25 28 33 50 52 60 61 80

consulta ao "88": 2 leituras (27, 88) + 2 escritas

88 27 99 5 2 7 10 25 28 33 50 52 60 61 80

consulta ao "27": 2 leituras (88, 27) + 2 escritas

27 88 99 5 2 7 10 25 28 33 50 52 60 61 80

consulta ao "88": 2 leituras (27, 88) + 2 escritas

88 27 99 5 2 7 10 25 28 33 50 52 60 61 80

consulta ao "5": 4 leituras (88, 27, 99, 5) + 4 escritas

5 88 27 99 2 7 10 25 28 33 50 52 60 61 80

consulta ao "99": 4 leituras (5, 88, 27, 99, 5) + 4 escritas

99 5 88 27 2 7 10 25 28 33 50 52 60 61 80

...e tudo recomeça.

O número médio de leituras (e escritas) por consulta tenderá, portanto, a

(4 + 4 + 2 + 2 + 2 + 2 + 2 + 2 + 4 + 4) / 10 = **2,8.**

b) Qual seria o número médio de leituras no array (por consulta) caso fosse utilizada uma busca binária?

Com busca binária,

o "27" demanda 3 leituras,
o "88" demanda 4 leituras,
o "5" demanda 3 leituras, e
o "2" demanda 4 leituras,

para um número médio de **3,5** leituras por consulta (e zero escritas).

3) Num esquema de hashing com resolução de colisões por encadeamento externo, que relação tem o fator de carga da tabela com o tempo esperado para o lookup de uma chave presente na tabela? O que é preciso para que o tempo esperado para esse lookup seja O(1)?

O fator de carga *alfa* = n/m corresponde ao tamanho médio de uma lista de chaves sinônimas (supondo um espalhamento uniforme da função hash utilizada). Portanto, o tempo médio para acesso a uma chave que *não está* na tabela é O(1 + *alfa*), pois primeiro localiza-se o bucket, depois percorre-se toda a lista. Tomando-se a média considerando apenas chaves *presentes* na tabela, o raciocínio é um pouco mais elaborado (intuição: buckets com mais chaves sinônimas são também mais frequentemente acessados, pois mais buscas de chaves presentes na tabela recairão naqueles buckets); no entanto, é possível provar que vale ainda aquela mesma expressão. Para que o tempo esperado seja constante, basta dimensionar o array subjacente à tabela hash (isto é, o valor de m) como sendo proporcional ao número de chaves que serão armazenadas, isto é, m = Theta(n).

4) O que é hashing universal?

Hahing universal: uma família $H = \{h_i\}$, para i de 1 a t, t > 0, de funções hash com contradomínio de tamanho m, e com a propriedade de que, para qualquer par de chaves x, y do conjunto de chaves possíveis, x e y são sinônimas segundo no máximo t/m das funções de H. Com isso, escolher aleatoria e uniformemente uma função de H para distribuir as chaves equivale a utilizar uma função hash "muito boa", que mapeasse uniformemente as chaves pelos m elementos do contradomínio.

5) Seja a seguinte função que converte letras minúsculas (ou espaços) em números inteiros:

$f(c)$	=	0,	se c pertence a {' ', 'a', 'e', 'i', 'o', 'u'};
		1,	se c pertence a {'b', 'c', 'd'};
		2,	se c pertence a {'f', 'g', 'h'};
		3,	se c pertence a {'j', 'k', 'l', 'm', 'n'};
		4,	se c pertence a {'p', 'q', 'r', 's', 't'};
		5,	se c pertence a {'v', 'w', 'x', 'y', 'z'}.

Seja agora a seguinte função hash para pares de letras minúsculas:

$h(XY) = f(X)*6 + f(Y)$. (Exemplo: h("th") = 26.)

a) Cite duas características ruins da função hash acima.

A divisão de caracteres pelos "grupos" foi nitidamente arbitrária, não se baseando em qualquer estudo estatístico sobre a frequência de

ocorrência daqueles caracteres numa língua qualquer. As vogais, por exemplo, estão todas no mesmo grupo. Dificilmente a distribuição que advirá dessa função hash será próxima à uniforme. Outra característica ruim é a má utilização do espaço, pois o contradomínio de *h* corresponde ao intervalo [0,35], isto é, são necessários no mínimo 36 bits para cada "filtro" (no contexto de assinaturas de linhas de texto). Mas, para termos 36 bits, acabaremos utilizando 64 bits por palavra (por exemplo o long long int do C). Ou seja, quase metade do tamanho de cada filtro será desperdiçado.

b) Suponha que você irá criar a assinatura de uma linha de texto utilizando uma palavra de 40 bits, onde os bits '1' estarão nas posições que são imagem de algum par de letras consecutivas daquela linha de texto segundo a função *h*. Qual seria a assinatura da linha de texto 'testando assinatura'?

h(te) = 24 + 0 = 24
h(es) = 0 + 4 = 4
h(st) = 24 + 4 = 28
h(ta) = 24 + 0 = 24
h(an) = 0 + 3 = 3
h(nd) = 18 + 1 = 19
h(do) = 6 + 0 = 6
h(o) = 0 + 0 = 0
h(a) = 0 + 0 = 0
h(as) = 0 + 4 = 4
h(ss) = 16 + 4 = 20
h(si) = 16 + 0 = 16
h(in) = 0 + 3 = 3
h(na) = 18 + 0 = 18
h(at) = 0 + 4 = 4
h(tu) = 24 + 0 = 24
h(ur) = 0 + 4 = 4
h(ra) = 24 + 0 = 24

(Notem o número escessivo de colisões, em razão da função hash pobre que estamos utilizando.)

A assinatura seria:

L = 0001101000000000101110001000100000000000

c) A consulta pela palavra 'erro' na linha 'testando assinatura' demandaria uma busca lenta naquela linha, ou seria eliminada de primeira quando testada contra o filtro (assinatura) da linha?

h(er) = 0 + 4 = 4
h(rr) = 24 + 4 = 28
h(ro) = 24 + 0 = 24

Assinatura da palavra buscada:
B = 00001000000000000000000010001000000000000

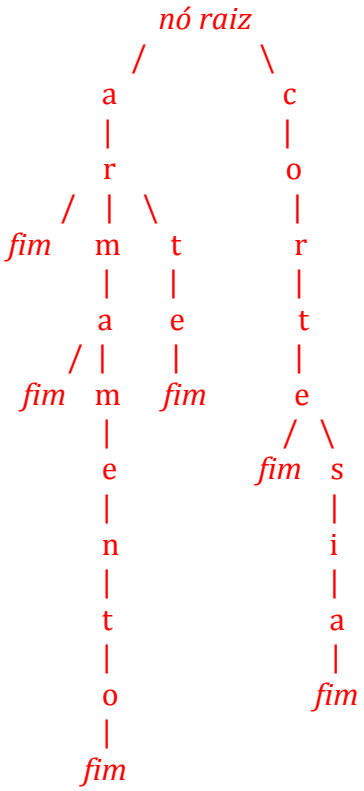
Comparando com a assinatura *L* da linha, temos:
B and not L = **00000000000000000000000000000000000000000**

Tudo zero. Ou seja, todos os bits "requisitados" pela palavra buscada estão presentes na assinatura daquela linha, de forma que uma busca lenta teria que ser feita (é possível, a princípio, que a palavra buscada se encontre naquela linha). Nesse caso, sabemos ter sido um falso positivo, pois a palavra de fato *não se encontra* naquela linha.

6) Que string demandaria um maior número de buscas lentas nas linhas de um texto com assinaturas por linha obtidas de funções hash aplicadas a pares de caracteres: "arara" ou "araruama"? Esse maior número seria necessariamente observado? Justifique suas respostas.

A palavra "arara" requisitará bits '1' correspondendo às imagens dos seguintes pares: "ar", "ra". A palavra "araruama" requisitará, *provavelmente*, mais bits, pois considerará, além dos pares "ar", "ra," também "ru", "ua", "am", "ma". Sendo assim, é de se esperar que a palavra "araruama" demande um número *menor* de buscas lentas. Isso deve ocorrer na maioria dos casos. No entanto, é perfeitamente possível que: (1) a *função hash* utilizada seja tal que o conjunto formado pelas imagens de todos os pares de "araruama" seja idêntico ao conjunto formado pelas imagens de todos os pares de "arara", e, assim sendo, para qualquer texto, as buscas lentas demandadas pelas duas palavras seriam exatamente as mesmas; ou (2) o *texto* seja tal que todas as linhas que atendam às requisições de "arara" atendam também às requisições de "araruama", e, dessa forma, as quantidades de buscas lentas seriam as mesmas para as duas palavras.

7) Represente graficamente a trie (árvore de prefixo) referente às palavras ‘ar’, ‘arma’, ‘armamento’, ‘arte’, ‘artefato’, ‘corte’ e ‘cortesia’.



8) O que é a propriedade da estabilidade de um algoritmo de ordenação? Em que situação ela pode ser desejável?

É a propriedade de manter a ordem relativa de elementos que empatem segundo o critério de ordenação utlizado. Desejável, por exemplo, em situações em que diversos critérios consecutivos de ordenação são aplicados. Uma vez que os ítens estejam ordenados segundo um critério c_1, ordenar segundo um novo critério c_2 não destruirá a ordem original, dada por c_1, dos elementos empatados segundo c_2. Ex.: e-mails, músicas no iTunes (ou coisa que o valha) etc.

9) Dados k arrays grandes de inteiros (com tamanhos n_i, para i de 1 a k, e com conteúdos aleatórios sobre os quais nada se sabe), como você faria para retornar os elementos que estão em todos os arrays? Obs.: Queremos minimizar em primeiro lugar o tempo médio dessa operação, mas também estamos interessados em utilizar a memória de forma inteligente.

Uma maneira eficiente (em tempo) é a seguinte: armazene os elementos de todas as listas, exceto a menor das listas, em tabelas hash (*hash sets*), uma tabela por lista. Agora, para cada elemento daquela menor lista,

procure se ele está em cada uma das tabelas hash. Se ele não estiver em alguma delas, não é necessário procurá-lo nas demais; se estiver em todas, ele faz parte da interseção. Essa estratégia roda em tempo esperado linear O(n1 + n2 + ...) no tamanho das listas, pois esse é o tempo necessário para popular as tabelas. O tempo necessário para os lookups é no pior caso igual ao número de tabelas hash vezes o tamanho da menor lista, portanto não excede a quantidade total de elementos das listas.

O problema com essa solução é que ela exige um espaço extra enorme. Se as listas estiverem, por exemplo, em disco, ocupando arquivos imensos, precisaremos de outro tanto de disco apenas para as tabelas. Alternativa melhor seria criar *uma* tabela hash apenas, inicialmente contendo os elementos da primeira lista. Agora, para cada elemento da segunda lista, faça um lookup por ele na tabela hash com os elementos da primeira. Cada elemento cujo lookup for bem-sucedido será armazenado numa lista temporária. Depois que todos os elementos da segunda lista tiverem sido computados, a lista temporária populará uma segunda tabela hash, com os elementos da interseção das duas primeiras listas. Note que a primeira tabela hash pode ser, agora, destruída. Além disso, essa segunda tabela só precisará ser dimensionada com tamanho *proporcional ao tamanho da interseção das duas primeiras listas* (ou seja, ao tamanho da lista temporária). Repete-se agora o processo para a terceira lista. Para cada um de seus elementos, lookup na tabela hash com a interseção parcial. Se bem-sucedido, coloque numa lista temporária. Ao fim, dimensione nova tabela hash para os elementos sobreviventes (interseção das três primeiras listas) e a popule, destruindo a tabela anterior. E assim sucessivamente até a última lista. Tempo continua linear no total de elementos de todas as listas; e, começando da menor lista, o espaço extra necessário é apenas proporcional ao tamanho dessa menor lista. Além disso, *ao longo da execução do algoritmo*, é possível devolver a memória ocupada por tabelas grandes, criando tabelas progressivamente menores. Isto é, se estamos falando de listas realmente grandes, de forma que o algoritmo, ainda que linear, demore tempo não-desprezível rodando, sua memória (ou seu disco, caso seus dados não caibam na RAM) não ficará ocupada excessivamente durante todo o tempo, sendo possível, por exemplo, que outros processos disponham dela antes da finalização do algoritmo de interseção de listas.

10) Quais as vantagens e desvantagens de se implementar uma trie utilizando, para cada nó, um array de filhos, ao invés de utilizar, para cada nó, uma lista encadeada de filhos? Se o alfabeto a ser considerado for muito grande, qual dessas maneiras soa mais adequada? Se o alfabeto for *realmente* grande, você pensaria em alguma outra implementação?

Vantagens do array: simplicidade, eficiência (é imediato verificar se certo caracter existe como filho de algum outro caracter). Desvantagem: gasta espaço igual ao tamanho do alfabeto por nó da trie. Por outro lado,

utilizar uma lista encadeada faz como que seja necessário percorrer eventualmente toda a lista para pesquisar a existência de um filho. Em compensação, listar todos os filhos é mais rápido, pois é feito em tempo proporcional ao número de filhos, e não ao tamanho do alfabeto; e gasta (muito) menos espaço, sobretudo em tries esparsas. Para alfabetos grandes, *deve ser* mais interessante o uso de listas encadeadas, para evitar o uso perdulário de espaço. Mas, evidentemente, tudo depende do que você pretende conseguir em termos de eficiência, e também do espaço que você dispõe. Para alfabetos realmente grandes (e aqui precisamos saber interpretar esse "realmente grandes" como algo que torna inviável o espaço demandado pela implementação via arrays, e torna muito lenta a consulta "x é filho de y" pela implementação via listas, uma alternativa é perdermos em simplicidade mas resolvermos a questão utilizando tabelas hash de filhos por nó da trie. Cada nó da trie, portanto, seria um par (chave, valor) armazenado numa tabela hash, onde chave = caracter, e valor = tabela hash contendo seus nós-filhos.