Universidade Federal do Ceará - Campus de Crateús
Bacharelado em Ciência da Computação
2ª Lista de Exercícios de Estrutura de Dados Avançada
04/05/2019
Prof. Lívio Freire

1. Insira as chaves $5, 28, 19, 15, 20, 33, 12, 17, 10$, com $m = 9$, em um *hashing*. Por simplicidade, assume-se que $hash(key) = key$. Mostre o resultado da inserção para as estratégias de resolução de conflito por encadeamento separado e sondagem linear.

2. Escreva um programa para implementar a seguinte estratégia para multiplicar dois polinômios esparsos P_1 e P_2, com tamanhos M e N, respectivamente, e gerar o polinômio P_3. Cada polinômio é representado por uma lista encadeada de objetos, cujas células armazenam o coeficiente e o expoente dos termos e são ordenadas decrescentemente pelo expoente. Cada termo de P_1 é multiplicado por um termo em P_2 e o resultado é inserido numa lista encadeada provisória L, resultando num total de MN operações. Para gerar P_3, pode-se ordenar os termos de L e combinar aqueles que apresentam o mesmo expoente. Isso requer a ordenação de MN registros, o que pode ser computacionalmente caro, especialmente em ambientes com pouca memória. Alternativamente, pode-se usar uma estratégia de *hashing* para combinar os termos com o mesmo expoente após a multiplicação, gerar os termos de P_3 e ordenar o resultado.

   (a) Escreve um programa para implementar a estratégia alternativa.

   (b) Como o polinômio resultante tem no máximo $M + N$ termos, qual o custo computacional de ambos os métodos?

3. Deseja-se encontrar a primeira ocorrência da string $P = P_1P_2 \dots P_k$ em uma longa string $A = A_1A_2 \dots A_N$. Pode-se resolver esse problema ao gerar o *código hash* da string P, obtendo-se o valor H_P, e comparar esse valor com o valor do *código hash* das substrings dadas por $A_1A_2 \dots A_k$, $A_2A_3 \dots A_{k+1}$, $A_3A_4 \dots A_{k+2}$, até $A_{N-k+1}A_{N-k+2} \dots A_N$. Se existir correspondência entre os valores do *código hash* de P e uma substring, compara-se as duas strings para verificar se são iguais. Retorna-se a posição em A quando a ocorrência é encontrada ou -1 se não existe.

   (a) Mostre que se o valor do *código hash* de $A_iA_{i+1} \dots A_{i+k-1}$ é conhecido, então o *código hash* de $A_{i+1}A_{i+2} \dots A_{i+k}$ pode ser computado em tempo constante.

   (b) Mostre que o custo em relação a tempo, no pior caso, é dado por $k + N$ mais o tempo gasto para refutar correspondências falsas.

   (c) Escreva um programa para implementar esse algoritmo.

4. (Entregar) Implemente um verificador ortográfico usando uma tabela hash. Assuma que o dicionário venha de duas fontes: um arquivo dicionário pré-existente e um segundo arquivo que contém um dicionário pessoal. Mostre todos as palavras com erro ortográfico e a posição no texto onde ocorrem. Além disso, para cada palavra com erro ortográfico, liste as palavras no dicionário que são obtidas por uma das seguintes regras:

(a) Adição de um caractere.

(b) Remoção de um caractere.

(c) Inversão de posição de dois caracteres adjacentes.

5. Considere um mecanismo de autenticação que armazena no base de dados para cada usuário o *login* e, por motivo de segurança, o valor *hash* da senha. Quando um usuário tenta conectar num sistema que usa esse mecanismo, a senha recebida é transformada pela mesma função *hash* usada para salvar as senhas no base de dados. Caso o par (login, *hash(senha)*) exista na base de dados, a autenticação é bem sucedida.

   (a) No caso da base de dados do mecanismo de autenticação ser roubada e quem roubou conhecer a função *hash* que gerou as senhas, é possível ter acesso às contas dos usuários no sistema?

   (b) Suponha que o login de um usuário seja conhecido, assim como uma senha diferente da senha do usuário, mas com o mesmo valor *hash*. Com base nesses dados, é possível ter acesso à conta do usuário?

6. Escreva um programa para encontrar valores de a e M, com M o menor possível, tais que a função de hashing $(a*k)\%M$, que transforma a k-ésima letra do alfabeto em um valor hash, não produza colisões quando aplicada às chaves S E A R C H X M P L. Isso é conhecido como função de hashing perfeita.

7. Acrescente um método *delete()* à classe SeparateChainingHashST. O seu método deve ser ansioso: ele deve remover o chave solicitada imediatamente (e não simplesmente marcá-la com *null* para remoção posterior).

8. Acrescente um método *delete()* à classe LinearProbingHashST. Para remover uma chave da tabela, o método deve atribuir *null* ao valor associado à chave e adiar a efetiva remoção da chave (e do valor associado) até o momento em que a tabela for redimensionada. (Observação: Você deve sobrescrever o valor *null* se uma operação *put()* subsequente associar um novo valor à chave.) O desafio é decidir quando *resize()* deve ser chamado. Para tomar a decisão de redimensionar, seu método deve levar em conta não só o número de posições vagas da tabela mas também o número de posições "mortas".

9. É fácil implementar operações como *min()*, *max()*, *floor()* e *ceiling()* em tabelas de símbolos implementadas com tabelas de hash?

10. (Entregar) Escreva um programa que insira N chaves inteiras (números int) aleatórias em uma tabela de tamanho $N/100$ usando hashing com encadeamento e depois determine o comprimento da lista mais curta e da mais longa. Faça isso para N igual a 10^3, 10^4, 10^5, 10^6. (Faça um pequeno relatório sobre os resultados e acrescente o relatório, como comentário, ao final do seu programa.)

11. Adicione o método *keysThatMatch()* nas implementações de árvores digitais. O método recebe uma string s e imprime todas as chaves que casam com s quando os caracteres '.' em s são interpretados como curingas, ou seja, casam com qualquer caractere. Por exemplo, no nosso exemplo padrão a resposta de keysThatMatch(.he) é she e the. A resposta de keysThatMatch(s..) é sea e she.

12. Adicione as operações *floor()*, *ceiling()*, *rank()* e *select()* nas implementações de árvores digitais.

13. Escreva um método que receba um texto e conte o número de substrings de comprimento L distintas que o texto contém. Por exemplo, o texto cgcgggcgcg tem cinco substrings de comprimento 3 distintas: cgc, cgg, gcg, ggc, e ggg. Dica: Insira cada substring de comprimento L em uma Trie. Qual o custo computacional em relação ao tempo?

14. Escreva um método que receba um texto e conte o número ocorrências de cada substrings de comprimento L que o texto contém.

15. Escreva um método para remover uma chave de uma TST.

16. (Entregar) Qual era mesmo o título daquele filme? Escreva um programa que receba uma string s e devolva os títulos de filmes que se aproximam de s. Mas precisamente, o programa deve devolver (1) os títulos dos filmes que tenham s como prefixo, (2) o mais longo prefixo de s que seja um título de filme, e (3) os títulos dos filmes que casam com s quando os caracteres '.' em s são interpretados como curingas. Detalhes técnicos: Extraia apenas os títulos de filmes do arquivo movies.txt, já que contém no arquivo também o ano e o elenco do filme. Não queremos nos atrapalhar com letras maiúsculas, letras acentuadas, pontuação, etc. Assim, todos os títulos devem ser convertidos para o alfabeto a b ... y z ? 0 1 ... 8 9 mais o espaço em branco (total de 38 caracteres). O alfabeto de consulta tem um caractere adicional, o ponto, que funciona como curinga. Converta cada um dos 256 caracteres do ASCII estendido no correspondente caractere do nosso alfabeto. Por exemplo, A e ä serão levados em a e Ç será levado em c. Todos os caracteres sem um correspondente natural devem ser levados em ?. Armazene tudo numa trie para responder consultas rapidamente. As consultas devem ser feitas pela linha de comando e a string de consulta deve ser embrulhada em aspas simples (caracteres ‘ ou ’) para que a string possa conter espaços em branco. (Portanto a string de consulta não pode conter aspas simples.) Tanto quanto possível, use as classes Java vistas em sala sem alteração. Faça testes. No fim do seu programa, coloque o resultado de testes interessantes (o que você digitou no terminal e qual foi a resposta do programa) em um relatório.

Divirta-se!