Segunda lista de exercícios – Estruturas de Dados / 2019-2
Prof. Vinícius Gusmão

1) Simule a inserção das seguintes chaves, na ordem abaixo, em uma tabela hash implementada sobre um array de 10 posições com resolução de colisões por encadeamento externo. PS.: Considere que uma chave é sempre inserida no início da lista encadeada correspondente à sua posição no array. PS.2.: A função hash utilizada é h(x) = (x (x -1)) % 10.

Chaves: 134, 22, 15, 1466, 11, 900, 9000, 90000, 2341, 433.

a) Indique claramente como estarão organizadas as chaves na estrutura final, depois de todas as inserções.
b) Qual o número médio de comparações realizadas durante a busca por uma chave que pertença à tabela hash, supondo que todas as chaves sejam buscadas com a mesma frequência?
c) Essa parece ser uma boa função hash para a dispersão dessas chaves?
d) Sugira uma função hash facilmente calculável que seja melhor do que essa, *para essas chaves*.

2) Simule a inserção das mesmas chaves da questão anterior (com a mesma função hash), mas agora utilizando uma resolução de colisões diferente: para a inserção da chave x, se a posição h(x) estiver vazia, a chave x será inserida na posição h(x). Do contrário, a chave x será inserida na primeira posição vazia do array, a partir de h(x). Se o array estiver com todas as posições ocupadas de h(x) até sua última posição, então a busca pela posição vazia onde x deverá ser inserida continua a partir do começo do array.

a) Indique claramente como estarão organizadas as chaves na estrutura final, depois de todas as inserções.
b) Como deverá ser o algoritmo de busca numa tabela hash que utilize esse tipo de resolução de colisões?
c) Qual o número médio de comparações realizadas durante a busca por uma chave que pertença à tabela hash, supondo que todas as chaves sejam buscadas com a mesma frequência?

3) Simule a execução do QuickSort e do MergeSort para a sequência abaixo:

60 56 12 45 4 2 9 18 46 1 400 234

PS.: No QuickSort, considere que o pivô escolhido é sempre o primeiro de cada lista.
PS.2: Você deve mostrar, a cada passo, o que é feito pelo algoritmo, não apenas o resultado final.

4) Utilize dois arrays de inteiros com endereçamento direto para implementar UNION/FIND de conjuntos disjuntos de inteiros. Um dos arrays, chamado *ranks,* armazenará na posição *j* o rank do elemento *j*, caso *j* seja o representante do subconjunto que contém *j*; caso contrário, conterá 0. O outro array, chamado *parents*, indicará, na posição *j*, o pai de *j* na árvore cuja raiz é o representante do subconjunto que contém *j*. Um nó terá ele próprio como pai se, e somente se, for o representante do subconjunto que o contém.

Obs.: As implementações podem ser em pseudo-código ou na linguagem que você preferir.

a) Implemente o método CREATE_SET(x), estabelecendo que o rank do representante de um conjunto unitário é 1.

b) Implemente o método FIND(x), que deve retornar o representante do subconjunto que contém *x*.

c) *Union by rank.* Implemente o método UNION(x, y), que deve fazer com que (i) o pai do representante de *x* passe ser o representante de *y,* caso o rank do representante de *x* seja menor ou igual ao rank do representante de *y*, incrementando o rani do representante de *y* caso sejam iguais*;* ou (ii) o pai do representante de *y* passe ser o representante de x*,* caso o rank do representante de y seja menor que o rank do representante de x.

d) *Path compression.* Modifique o método FIND(x) para que não apenas retorne o representante do subconjunto que contém *x*, mas que faça também com que todos os elementos no caminho de *x* até o representante *r* do subconjunto que contém x passem a ter *r* como pai.