Análise de Algoritmos e Estruturas de Dados
2º semestre de 2024
Lista de exercícios 2

1. Explique o princípio de funcionamento do algoritmo de ordenação *selection sort*. Mostre que sua complexidade de tempo de execução é $\Theta(n^2)$.
2. Explique o princípio de funcionamento do algoritmo de ordenação *insertion sort*. Mostre que sua complexidade de tempo de execução é $\Omega(n)$ e $O(n^2)$.
3. Explique o princípio de funcionamento do algoritmo de ordenação *bubble sort*. Mostre que sua complexidade de tempo de execução é $\Omega(n)$ e $O(n^2)$.
4. Explique o princípio de funcionamento do algoritmo de ordenação *merge sort*. Mostre que sua complexidade de tempo de execução é $\Theta(nlgn)$.
5. Explique o princípio de funcionamento do algoritmo de ordenação *quick sort*. Mostre que sua complexidade de tempo de execução é $\Omega(nlgn)$ e $O(n^2)$.
6. Explique o princípio de funcionamento do algoritmo de ordenação *heap sort*. Mostre que sua complexidade de tempo de execução é $O(nlgn)$.
7. O que podemos observar ao comparar os algoritmos de ordenação *selection sort* e *heap sort*?
8. Simule a execução dos algoritmos de ordenação *selection sort*, *quick sort* e *merge sort* para os seguintes valores: $7, 5, 2, 3, 1, 6, 8, 4$.
9. Considere a seguinte variação do algoritmo de ordenação *merge sort*:

```
void merge_sort(int * a, int ini, int fim){

  if(ini < fim){

    int q1 = ini + (fim - ini) / 4;

    merge_sort(a, ini, q1);
    merge_sort(a, q1 + 1, fim);
    merge(a, ini, q1, fim);
  }
}
```

a) O desempenho do função *merge* é afetado devido às modificações desta versão? Justifique.

b) E quanto ao desempenho do *merge sort* como um todo, há alguma alteração? Justifique.

10. Considere uma versão alternativa do algoritmo de ordenação *Quicksort* que utiliza a versão da função **particiona** apresentada a seguir:

```
int particiona(int * a, int ini, int fim){

  int i, min, max, x, prox, q, tam_b;
  int * b;

  tam_b = fim - ini + 1;
  b = calloc(tam_b, sizeof(int));

  min = a[ini];
  max = a[ini];

  for(i = ini + 1; i <= fim; i++){
    if(a[i] < min) min = a[i];
    if(a[i] > max) max = a[i];
  }

  x = (min + max) / 2;
  prox = 0;

  for(i = ini; i <= fim; i++){
    if(a[i] <= x){
      b[prox] = a[i]; prox++;
    }
  }

  q = ini + prox - 1;

  for(i = ini; i <= fim; i++){
    if(a[i] > x){
      b[prox] = a[i]; prox++;
    }
  }

  for(i = 0; i < tam_b; i++){
    a[ini + i] = b[i];
  }

  return q;
}
```

A partir do código, e assumindo que todo vetor a ser ordenado **não possui valores repetidos**, responda:

a) O *Quicksort* funcionará corretamente com a versão do **particiona** apresentada? Justifique.

b) Esta versão do **particiona** previne o cenário de pior caso que ocorre na versão clássica do *Quicksort*? Justifique.

c) Esta versão do **particiona** altera o comportamento do *Quicksort* em relação ao **tempo de execução**? Justifique.

11. Suponha uma versão hipotética da função **particiona**, que sempre garante um particionamento uniforme, a um custo $\Theta(n\sqrt{n})$. Quais os impactos que o uso desta versão hipotética traria ao *Quicksort* no que diz respeito ao tempo de execução? Justifique.

12. Considere um vetor de valores inteiros, organizado da seguinte forma:

$$\{a_1,\ b_1,\ c_1,\ a_2,\ b_2,\ c_2,\ ...,\ a_m,\ b_m,\ c_m\}$$

onde $a_i > b_i > c_i$ e $c_{i+1} > a_i$. Dado que o vetor possui um total de n valores (observe que $n = 3m$), determine o limite assintótico justo (Θ) da função $T(n)$ que descreve o tempo de execução do *insertion sort* ao ordenar os elementos deste vetor em ordem crescente. Justifique sua resposta.

13. Considere as seguintes implementações de estruturas do tipo lista: listas sequenciais (usam um vetor - ou *array* - como espaço de armazenamento dos seus elementos) e listas ligadas (cada elemento é armazenado em uma estrutura denominada nó, e a lista é representada pelo encadeamento de diversos nós). Comparando estas duas abordagens para implementação de listas, o que considerações podemos fazer em relação a(o):

a) Consumo de memória em cada uma das abordagens.

b) Acesso a um elemento da lista a partir de um índice.

c) Eficiência da operação de inserção (discuta também se faz diferença a inserção ser baseada em posição, ou baseada em ordem de valor dos elementos).

d) Eficiência da operação de remoção.

14. São denominadas listas ordenadas, aquelas listas em que a inserção ocorre de modo que os elementos sejam mantidos em ordem de valor. A organização ordenada dos elementos pode acelerar a busca por um elemento na lista? Em caso positivo explique como ocorre essa melhoria, e se ela se aplica tanto a listas sequenciais quanto listas ligadas.

15. Tomando como base as implementação de listas ligada disponibilizada no arquivo **lista_ligada.c**, implemente as seguintes funções:

a) Escreva uma função chamada **inverte_lista** que recebe o endereço de uma lista ligada como parâmetro, e devolve o endereço de uma outra lista ligada contendo os mesmos elementos, porém em ordem invertida.

b) Escreva uma função chamada **divide_lista**, que recebe o ponteiro de uma lista, e a divida em duas partes de igual tamanho (se a quantidade de elementos for ímpar, faça com que a primeira parte tenha 1 elemento a mais do que a segunda). A lista passada como parâmetro deve ser modificada para corresponder à primeira metade da divisão, enquanto o endereço da segunda lista deve ser devolvido como retorno da função.

c) Escreva uma função chamada **junta_listas** que recebe o ponteiro para duas listas, junta o conteúdo das duas listas em uma única lista, e devolve o endereço da lista resultante.

d) Proponha uma nova versão da função **junta_listas** que tenha complexidade de tempo $\Theta(1)$. Para isso você deve propor também alguma modificação na forma como a lista é representada.

16. Explique como funciona a busca em uma árvore binária (não necessariamente de busca). Explique por que sua complexidade de tempo é $O(n)$, onde n é a quantidade de elementos contidos na árvore, no caso em que a função de busca recebe dois parâmetros: a árvore e o elemento a ser buscado.

17. Explique como funciona a inserção em uma árvore binária (não necessariamente de busca). Explique por que sua complexidade de tempo é $O(1)$, no caso em que a função de inserção recebe quatro parâmetros: a árvore, o elemento a ser inserido, o ponteiro do nó que será o pai do elemento a ser inserido (este parâmetro pode ser nulo caso se deseje inserir o elemento na raiz), e o lado em que o elemento será "pendurado" no pai.

18. Explique como funciona a remoção em uma árvore binária (não necessariamente de busca). Explique por que sua complexidade de tempo é $O(h)$, onde h é a altura da árvore, no caso em que a função de remoção recebe dois parâmetros: a árvore e o ponteiro do nó a ser removido.

19. Explique como funcionam os percursos *in-ordem*, *pré-ordem* e *pós-ordem* de uma árvore binária.

20. Quais propriedades devem ser satisfeitas para uma árvore binária ser considerada uma árvore binária de busca?

21. Qual o tipo de percurso que passa pelos elementos de uma árvore binária de busca em ordem crescente de valor?

22. Explique como funciona a busca em uma árvore binária de busca, e explique por que sua complexidade de tempo é $O(h)$, onde h é a altura da árvore. Considere que a função de busca recebe como parâmetros a árvore e o elemento a ser buscado.

23. Explique como funciona a inserção em uma árvore binária de busca, e explique por que sua complexidade de tempo é $O(h)$, onde h é a altura da árvore. Considere que a função de inserção recebe como parâmetros a árvore e o elemento a ser inserido.

24. Explique como funciona a remoção em uma árvore binária de busca, e explique por que sua complexidade de tempo é $O(h)$, onde h é a altura da árvore. Considere que a função de remoção recebe como parâmetros a árvore e nó a ser removido.

25. Seja n a quantidade de elementos armazenados em um árvore binária, e h a sua altura. Explique por que $h = \Omega(log_2(n))$ e $h = O(n)$ (ou, reformulando a mesma pergunta de um modo mais

informal: explique por que o valor mínimo que h pode assumir é aproximadamente $log_2(n)$ e o valor máximo aproximadamente n).

26. Quais condições devem ser atendidas para que uma árvore binária de busca apresente o melhor desempenho possível em suas operações?

27. Explique por que uma árvore binária de busca é, em condições ideais, uma estrutura melhor do que uma lista sequencial ordenada (que também permite a realização de buscas de forma eficiente).

28. Explique o que são árvores AVL, e qual a propriedade que deve ser satisfeita por toda árvore deste tipo.

29. Explique qual o papel das rotações no funcionamento de uma árvore AVL.

30. Indique qual será o estado final de uma árvore binária de busca após a inserção de cada um dos seguintes valores, nesta ordem: 8, 1, 9, 2, 3, 7, 4, 6, 5.

31. Indique quais serão as comparações realizadas em uma busca binária quando o valor 5 for buscado na árvore resultante da sequência de inserções descritas na questão anterior.

32. Indique qual será o estado final de uma árvore binária de busca após a inserção de cada um dos seguintes valores, nesta ordem: 4, 8, 2, 6, 7, 3, 5, 1, 9.

33. Indique quais serão as comparações realizadas em uma busca binária quando o valor 7 for buscado na árvore resultante da sequência de inserções descritas na questão anterior.

34. A partir das suas respostas nas quatro questões anteriores, e baseado no seu conhecimento sobre árvores binárias de busca, o que podemos argumentar a respeito da complexidade assintótica (referente ao tempo de execução) da busca neste tipo de arvóre? Como este desempenho se compara em relação ao desempenho da busca em listas lineares (com e sem ordenação)?

35. Indique os estados de uma árvore AVL após a inserção de cada um dos seguintes valores, nesta ordem: 8, 1, 9, 2, 3, 7, 5, 4. Quando após a inserção de um determinado valor a árvore se tornar desbalanceada, indique dois sub-estados: imediatamente após a inserção (antes da rotação que irá restaurar o balanceamento) e após a rotação que restaura o balanceamento.

36. Tomando como base as implementações de árvores disponibilizadas no arquivo **arvore_binaria.c**, implemente as seguintes funções:

    a) Escreva a função **conta_elementos** que recebe um ponteiro para uma árvore binária (que pode ser ou não de busca, é indiferente para esta função), e devolve a quantidade de elementos armazenados na árvore.

    b) Escreva a função **verifica_arvore_de_busca** que recebe um ponteiro para uma árvore binária (que pode ou não ser de busca), e devolve um valor booleano indicando se a árvore binária é uma árvore de busca ou não.

    c) Escreva a função **max** que recebe um ponteiro para uma árvore (não necessariamente de busca), e devolve o maior armazenado nela.

    d) Escreva a função **min** que recebe um ponteiro para uma árvore (não necessariamente de busca), e devolve o menor armazenado nela.

e) Faça uma nova versão da função **max**, desta vez assumindo que a árvore recebida como parâmetro é uma árvore binária de busca.

f) Faça uma nova versão da função **min**, desta vez assumindo que a árvore recebida como parâmetro é uma árvore binária de busca.

g) Escreva a função **imprime_decrescente** que recebe um ponteiro para uma árvore binária de busca, e imprime seus elementos em ordem decrescente de valor.

h) Escreva a função **altura** que recebe um ponteiro para uma árvore binária (que pode ou ser de busca) e devolva sua altura.

i) Escreva a função **verifica_AVL** que recebe um ponteiro para uma árvore binária de busca, e devolva um Boolean indicando se a árvore é uma AVL ou não.

j) Escreva a função **junta_arvores** que recebe o ponteiro para duas árvores (que não são de busca) e junta o conteúdo das duas árvores em uma única árvore, devolvendo o ponteiro da árvore que contém a união dos elementos. Você pode alterar o conteúdo de uma das árvores para que ela incorpore os elementos presentes na outra, mas sua função deve ser implementada de modo a minimizar o número de acertos de ponteiros que devem ser feitos (ou seja, não vale pegar todos os elementos de uma árvore e, um a um, inserir na outra). Faça uma implementação que aumente a altura da árvore resultante em no máximo uma unidade.