DCC/ICEx/UFMG **Lista de Exercícios 2**
DCC 205 - Estruturas de Dados
1º semestre de 2024

---

1. Indique se as afirmações são verdadeiras ou falsas e justifique.

(a) No pior caso, a pesquisa binária e a pesquisa sequencial têm a mesma ordem de complexidade.

Solução: Falso
Em um arranjo com n elementos, a ordem de complexidade no pior caso da busca binária é $O(log(n))$, enquanto da busca sequencial é $O(n)$.

(b) O melhor caso de uma pesquisa em tabela hash será O(1), independente do tratamento de colisão adotado.

Solução: Verdadeiro
Considerando o melhor caso nas implementações de tratamento de colisão vistas em sala:

- Utilizando encadeamento no melhor caso a lista que o elemento será inserido está vazia.
- Utilizando endereçamento aberto no melhor caso a posição retornada pela função de *hashing* está vazia.

Ambas operações podem ser realizadas em tempo constante.

(c) O algoritmo de pesquisa em uma árvore AVL é o mesmo de uma árvore binária de pesquisa.

Solução: Verdadeiro
A árvore AVL é essencialmente uma árvore binária de busca. Ela difere de uma árvore comum pois faz o autobalanceamento nas funções de inserção e remoção, que são as funções que interferem na estrutura da árvore. Como a função de pesquisa não mexe na estrutura da árvore, ela é idêntica a de uma árvore binária de busca.

(d) A complexidade de inserção em uma árvore AVL tem o pior caso igual a O(n).

Solução: Falso
O processo de inserção consiste em localizar a posição onde o novo elemento deve ser inserido e realizar uma quantidade constante de rotações. Localizar o elemento pode ser feito em tempo $O(log(n))$ e cada rotação gasta tempo constante.

(e) Considere o seguinte algoritmo de ordenação que recebe como entrada um vetor v com n inteiros positivos.

- Crie uma árvore AVL T vazia. Insira em T os elementos de v na ordem em que eles aparecem.
- Re-escreva os elementos em v fazendo uma leitura in-fixa da árvore.

Este algoritmo possui complexidade de tempo $O(nlog(n))$ no pior caso.

Verdadeiro!
A complexidade de inserção na AVL no pior caso é $O(log(n))$. São feitas n inserções.
A varredura in-fixa tem custo $O(n)$
O custo total fica:
$O(nlog(n)) + O(n)$.

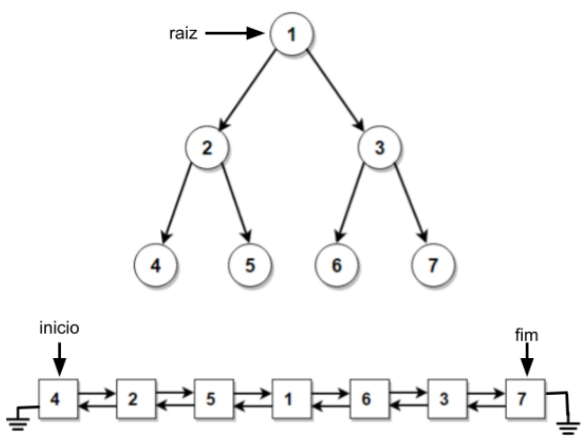
(f) O tempo de execução do quicksort externo independe da disposição das chaves no arquivo.

Falso!
Dado a forma como o pivô é "escolhido", a disposição dos elementos pode fazer com que a divisão dos arquivos levem ao pior caso quadrático.

2. O algoritmo recebe a raiz de uma árvore e retorna a soma das chaves em seus nós. Entretanto algumas linhas de código foram apagadas, sabendo o que esse método deve fazer e usando seus conhecimentos em estruturas de dados, reescreva esse algoritmo.

```
int somaArvore(No* t) {
 int x,y;
 if (t == NULL) return 0;
     x = SomaArvore(t->esq);
     y = SomaArvore(t->dir);
 return t->chave + x + y;
}
```

3. Uma lista duplamente encadeada é uma estrutura de dados lista que permite caminhar em ambas as direções, ou seja, você pode precorrê-la do início para o fim (como acontece na lista encadeada simples) ou do fim para o início. Você recebe uma árvore binária e precisa convertê-la em uma lista duplamente encadeada, mas deve utilizar apenas a estrutura de dados da própria árvore, ou seja, não existe estrutura de dados auxiliar além dos apontadores de início e fim da lista duplamente encadeada. Por exemplo, a árvore binária apresentada é convertida na lista duplamente encadeada, a qual vai estar ordenada de acordo com o caminhamento in-ordem da árvore. Complete o código apresentado de forma a executar a conversão:




```
typedef struct node {
        int data;
        struct node* esq;
        struct node* dir;
} node_t;


node_t* converte(node_t* raiz) {
     if (raiz == NULL) return raiz;
     node_t* inicio = NULL;
     node_t* fim = NULL;
     arvbin2lista(raiz, &inicio, &fim);
     return inicio;
}
```

```
void arvbin2lista(node_t* raiz, node_t** inicio, node_t** fim) {
     if (raiz == NULL) return;
     arvbin2lista(raiz->esq, inicio, fim);
     if (*inicio == NULL) *inicio = raiz;
     raiz->esq = *fim;
     if (*fim != NULL) (*fim)->dir = raiz;
     *fim = raiz;
     arvbin2lista(raiz->dir, inicio, fim);
}
```

4. Dada uma matriz contendo 0s e 1s, uma ilha é definida como um grupo de 1s (que sejam vizinhos horizontal ou verticalmente) que é cercado por apenas 0s em todos os quatro lados (excluindo as diagonais). Assuma que nenhum 1 está nas bordas da matriz fornecida. Por exemplo, a matriz abaixo tem duas ilhas:

   00000000
   01111000
   01010000
   01111010
   00000000

   Considere o tipo abstrato de dados para conjuntos disjuntos (union-find) abaixo:

```
typedef struct subset {
    int parent;
    int rank;
}subset;

int find(subset sets[], int i){
  if (sets[i].parent == i){
    return i;
  } else {
    return find(sets,sets[i].parent);
  }
}
```

```
void Union(subset sets[], int xroot, int yroot){
  if (sets[xroot].rank < sets[yroot].rank)
       sets[xroot].parent = yroot;
  else if (sets[xroot].rank>sets[yroot].rank)
       sets[yroot].parent = xroot;
     else {
       sets[yroot].parent = xroot;
       sets[xroot].rank++;
     }
}
```

   Complemente o código a seguir que calcula o número de ilhas a partir de uma matriz usando o TAD acima:

```
int countislands(int map[DIMY][DIMX]){
  subset setv[MAXCEL];
  for (int i=0; i<MAXCEL; i++){ setv[i].parent = i; setv[i].rank = 0; }
  for (int i=0; i<DIMY; i++){
     for (int j=0; j<DIMX; j++){
        if (map[i][j]){
          int pos = i*DIMX+j;
          if (j+1<DIMX){
            if (map[i][j+1]){
                 int dir = i*DIMX+j+1;
                Union(setv,find(setv, pos), find(setv,dir));
            }
          }
          if (i+1<DIMY){
            if (map[i+1][j]){
                 int baixo = (i+1)*DIMX+j;
                Union(setv,find(setv, pos), find(setv,baixo));
            }
          }
        }
     }
  }
  int numislands=0;
  for (int i=0; i<DIMY; i++){
    for (int j=0; j<DIMX; j++){
       if (map[i][j]){
         int pos = i*DIMX+j;
         if (find(setv,pos) == pos) numislands++;
       }
    }
  }
  return numislands;
}
```

5. Escreva um algoritmo que calcule o piso de $\sqrt{n}$ para qualquer número natural $n \geq 2$. O seu algoritmo só pode usar as operações de soma, subtração, multiplicação e divisão e deve deve consumir tempo $O(log(n))$.

```
int BINSORT(int n, int i){
    j = i/2;
    if(j*j <= n){
        if((j+1)*(j+1) >= n){
            return j;
        }
        else{
            return BINSQRT(n, i/2);
        }
    }
    else{
        if((j-1)*(j-1) <= n){
            return j;
        }
        else{
            return BINSQRT(n, (3*i)/4);
        }
    }
}
```

6. Seja a busca ternária uma extensão do algoritmo de busca binária onde o vetor é dividido em 3 partes ao invés de 2. Tendo isso em mente:

a) Complete o código a seguir que implementa a busca ternária:

```
int ternaria(int x, int * T, int Esq, int Dir){
   int j = (Esq+Dir)/3; if (j<Esq) j=Esq;
   int k = 2*(Esq+Dir)/3; if (k>Dir) k=Dir;
   if (Esq==Dir) return Esq;
   if (Dir-Esq>=2){
        if (T[j] >= x){
        if (T[j] == x) return j;
        else return ternaria(x,T,Esq,j);
        } else {
          if (T[k] >= x){
        if (T[k] == x) return k;
        else return ternaria(x,T,j+1,k);
          } else return ternaria(x,T,k+1,Dir);
        }
   } else {
        if (T[Esq] == x) return Esq;
        if (T[Dir] == x) return Dir;
        return -1;
   }
}
```

b) Qual é a complexidade do algoritmo da letra a? Ele é assintoticamente melhor que a busca binária comum?
Solução:
A complexidade assintótica é $O(log(n))$. A mudança é apenas na base do logaritmo, sendo assim diferem apenas por um fator constante, logo são assintoticamente equivalentes.

7. Escreva um procedimento que recebe dois vetores de tamanho n com inteiros positivos como entrada e imprime na tela a interseção dos dois em ordem não-decrescente. Seu algoritmo **deve** ter complexidade $O(nlog(n))$. Você não precisa implementar TADs ou métodos que foram vistos em sala, mas deve explicitar seu uso.

```
void Intersec(int* V, int* W, int n){
  MergeSort(V, n);
  MergeSort(W, n);
  for(int i = 0; i < n; i++){
    if(BinSearch(W, n, V[i]) != -1){
        cout << V[i] << " ";
    }
  }
  cout << "\n";
}
```

Se a condição de imprimir em ordem não-decrescente fosse removida, ou seja, você pode imprimir a interseção em qualquer ordem, explique como você poderia desenvolver uma estratégia para resolver este problema em tempo $O(n)$.

Solução:
Seja v e w os vetores em questão. Insira os elementos de w em uma Tabela Hash T e depois para cada elemento v_i de v, imprima v_i caso ele esteja presente em T.

8. O algoritmo Heapsort tem como premissa a construção e utilização de um Heap, que é uma árvore binária em um vetor. Para referência, considere a seguinte implementação do HeapSort usando um heap binário.

```
void heapify(int * A, int l, int r){
  int i, j;
  int x;
  i = l;
  j = i * 2;
  x = A[i];
  while (j <= r){
    if (j < r){
       if (A[j] < A[j+1]) j++;
    }
    if (x >= A[j]) break;
    A[i] = A[j];
    i = j;
    j = i *2;
  }
  A[i] = x;
}
```

```
void buildheap(int *A, int n) {
  int l = n / 2;
  while (l > 0) {
    l--;
    heapify(A, l, n);
  }
}
void heapSort(int *A, int n) {
  int l, r;
  int x;
  buildheap(A, n);
  l = 1; r = n;
  while (r > 1){
    x=A[1];A[1]=A[r];A[r]=x;
    r--;
    heapify(A,l,r);
  }
}
```

Essa estratégia pode ser generalizada em um heap n-ário, onde n é o número de filhos de cada nó interno do heap. Um exemplo dessa estratégia é um heap ternário, ou seja, um heap onde cada nó interno tem até 3 filhos. Complete o código a seguir para implementar um heap ternário. Interessante notar que o código da função heapSort não precisa ser alterado.

```
void heapify(int * A, int l, int r){
  int x, i, k, m, n, maior;
  i = l;
  k = 3*i+1; // filho 1 de i
  m = 3*i+2; // filho 2 de i
  n = 3*i+3; // filho 3 de i
  maior = k;
  x = A[i];
  while (k <= r){
    if (k<=r && A[k] > A[maior]) maior = k;
    if (m<=r && A[m] > A[maior]) maior = m;
    if (n<=r && A[n] > A[maior]) maior = n;
    if (x >= A[maior]) break;
    A[i] = A[maior];
    i = maior;
    k = 3*i+1;
    m = 3*i+2;
    n = 3*i+3;
    maior = k;
  }
  A[i] = x;
}
void buildheap(int *A, int n) {
  int l;
  l = round((float)n/3);
  while (l > 0) {
    l--;
    heapify(A, l, n);
  }
}
```

9. Considere um hash de endereçamento aberto cuja função seja dada por h(i) = i mod 13. As colisões são tratadas de forma sequencial com passo igual a 1 (ou seja, considera a partir da próxima entrada da tabela a partir da posição indicada pelo hash). Diferencie as entradas não utilizadas daquelas onde houve remoção. Seja X seu número de matrícula % 100. *A correção em sala será feita utilizando X=71.*

1. Insira as chaves X, 88, 42, 19, 3 e 87. Qual a configuração da tabela hash?

0	-1
1	-1
2	-1
3	42
4	3
5	-1
6	71
7	19
8	-1
9	87
10	88
11	-1
12	-1

2. Insira as chaves 68, 22, 24, 65 e 38. Qual a configuração da tabela hash?

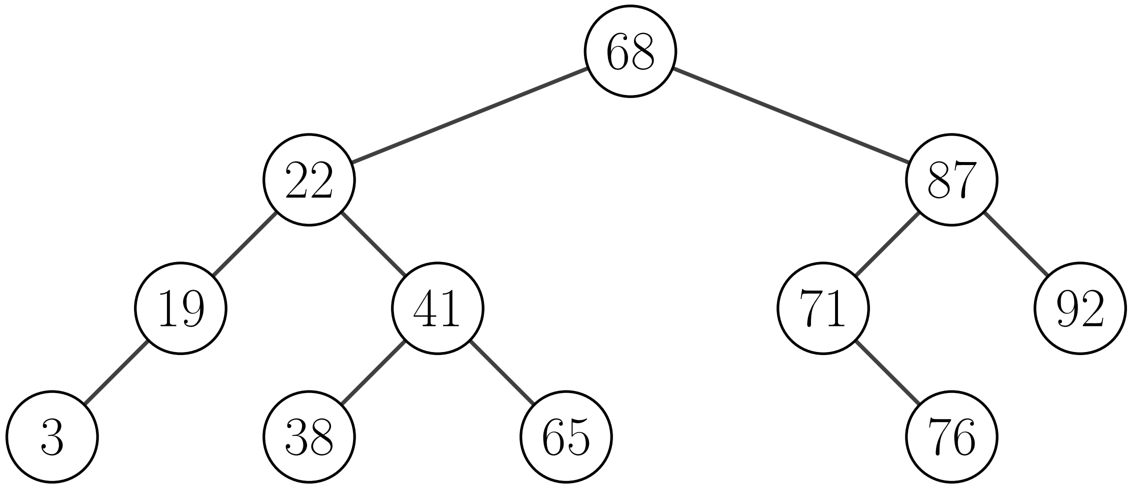
0	65
1	38
2	-1
3	42
4	3
5	68
6	71
7	19
8	-1
9	87
10	88
11	22
12	24

3. Remova as chaves 87, 22 e 24. Qual a configuração da tabela hash?

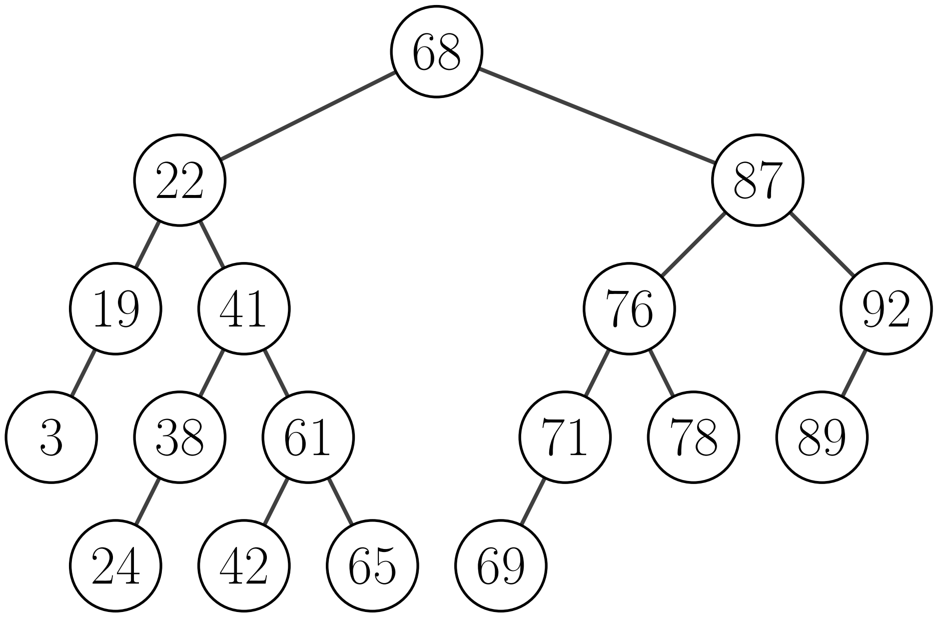
0	65
1	38
2	-1
3	42
4	3
5	68
6	71
7	19
8	-1
9	-2
10	88
11	-2
12	-2

10. Considere uma árvore AVL, onde remoções de chaves internas levam em consideração apenas o antecessor. Seja X seu número de matrícula % 100 *A correção em sala será feita utilizando X=71.*

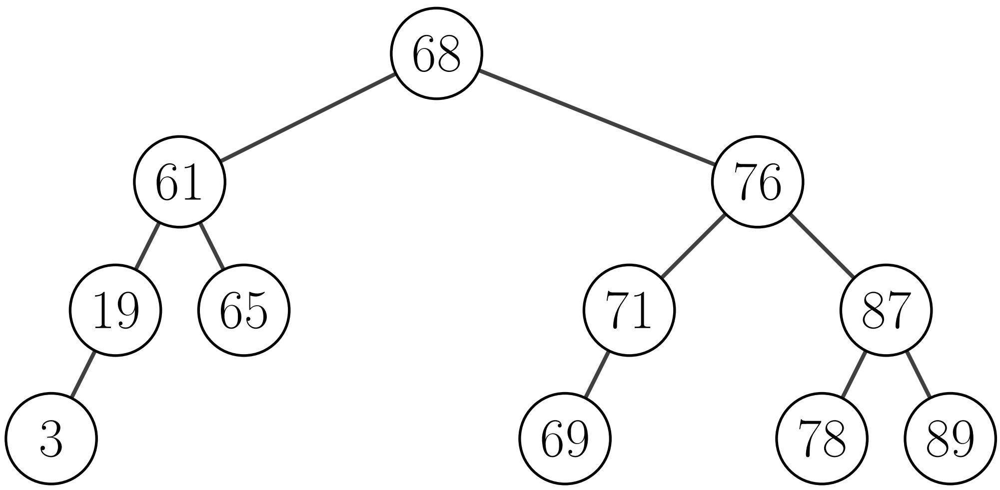
1. Insira as chaves X, 65, 68, 41, 19, 87, 92, 22, 76, 3 e 38 na árvore. Qual a árvore resultante?



2. Insira as chaves 24, 42, 78, 61, 89 e 69 na árvore. Qual a árvore resultante?

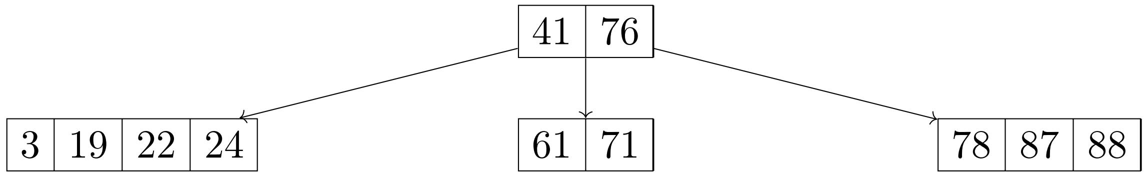


3. Remova as chaves 41, 92, 22, 38, 42 e 24 da árvore. Qual a árvore resultante?

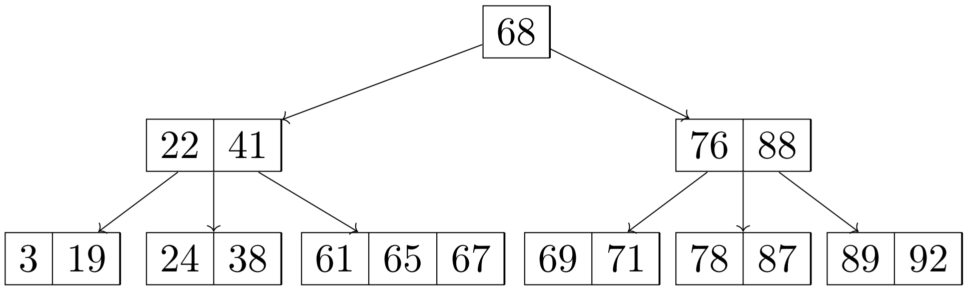


11. Considere uma árvore B com até 4 chaves por nó, onde remoções de chaves internas levam em consideração apenas o antecessor. Empréstimos de registros na remoção apenas dos "irmãos", ou seja, nós que tenham o mesmo "pai". Seja X seu número de matrícula % 100. *A correção em sala será feita utilizando X=71.*

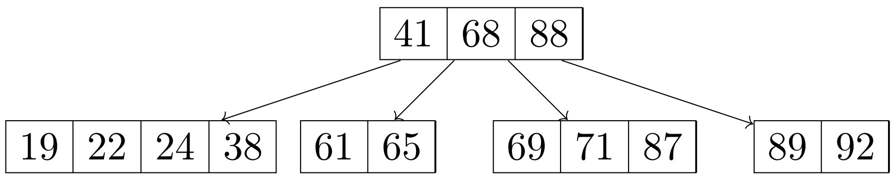
    1. Insira as chaves X, 24, 19, 41, 76, 3, 61, 78, 88, 87 e 22 na árvore e mostre o resultado.



    2. Insira as chaves 89, 92, 69, 65, 38, 68 e 67 na árvore e mostre o resultado.



    3. Remova as chaves 76, 3, 78 e 67 da árvore e mostre o resultado.



12. Você foi encarregado de realizar uma ordenação de um arquivo cujo conteúdo não cabe em memória primária usando ordenação polifásica com 4 "fitas"no total. Seja X seu número de matrícula % 100 *(Neste exercício, tome X=24)*. Após uma primeira passada usando seleção por substituição, você obteve X+33 blocos para serem intercalados. Indique quantos blocos cada uma das fitas deve receber para que o arquivo ordenado seja posicionado na fita 1. Justifique a sua resposta.

Fita 1	Fita 2	Fita 3	Fita 4	Total
13	0	24	20	57
0	13	11	7	31
7	6	4	0	17
3	2	0	4	9
1	0	2	2	5
0	1	1	1	3
1	0	0	0	1