3ª Avaliação Parcial (2ª chamada)
Curso: Engenharia de Computação
Disciplina: Estruturas de Dados
Prof. Jarbas Joaci de Mesquita Sá Junior
Universidade Federal do Ceará – UFC/Sobral

Nome: _ Data ___/___/2018

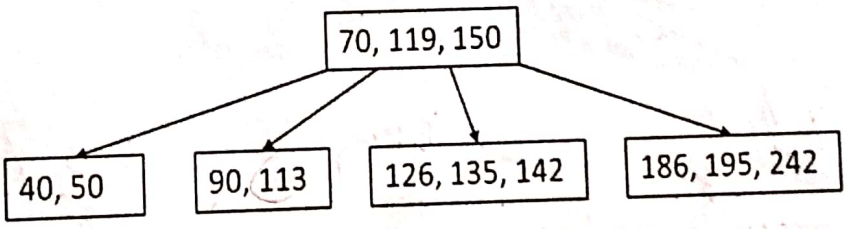
1ª) Explique detalhadamente como funciona o algoritmo *heapsort*. (2,0 pontos)

2ª) Explique detalhadamente como funciona o algoritmo *mergesort*. (2,0 pontos)

3ª) Explique detalhadamente as formas de tratar "colisões" na tabela de dispersão. (2,0 pontos)

4ª) Construa uma árvore B de ordem $t = 3$ para a seguinte sequência de chaves: 10, 2, 3, 4, 5, 11, 12, 13, 14, 15, 6, 7, 8, 9, 1, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26. (2,0 pontos)

5ª) Retire da árvore B de ordem $t = 3$ abaixo a seguinte sequência de elementos: 113, 40, 50, 70, 142. (2,0 pontos)



Obs.

1. todos os nós podem armazenar no **máximo** $2t$-1 elementos. A raiz pode armazenar no **mínimo** 1 elemento e os outros nós no **mínimo** t-1 elementos.
2. se a retirada ocorrer em um nó interno, o substituto da "chave" deverá ser buscado na subárvore esquerda;
3. se determinado nó em déficit devido a uma remoção tiver dois nós irmãos e estes puderem emprestar uma chave, o nó da esquerda deverá ser escolhido para o *empréstimo*;
4. se determinado nó em déficit devido a uma remoção tiver dois nós irmãos e estes **não** puderem emprestar uma chave, o nó da esquerda deverá ser escolhido para a *concatenação*.

Q1) A estrutura de dados Heap é um arranjo A que pode ser visto como uma árvore binária quase completa (as folhas não precisam estar completas)

- Um Heap é representado por dois atributos
  - Comprimento (A)
  - Tamanho-do-Heap(A), que pode ser menor ou igual ao Comprimento (A)
- Um Heap pode ser de máximo ou de mínimo. No Heap de Máximo o pai (i) é sempre maior que seus filhos direito (2i+1) e esquerdo (2i)
- O algoritimo Heapsort pode ser implementado com as seguinte funções
  - Max-Heapify (A, i)
  - Build-Max-Heap (A)
  - Heapsort (A)

Em pseudo código podemos escrever as funções da seguinte forma

Max-Heapify (A, i)

```
l ← left(i)
r ← right(i)
if (l ≤ tamanho-do-heap(A) e A[l] > A[i])
    then maior ← l
    else maior ← i
if (r ≤ tamanho-do-heap(A) e A[r] > A[maior])
    then maior ← r
if maior ≠ i
    then trocar A[maior] ↔ A[i]
         Max-Heapify (A, maior)
```

01 Continuação ---

A função Max-Heapify recebe como parametros um índice i do vetor e testa se o elemento desse índice é maior que seus filhos right(i) e left(i) se um dos filhos for maior que o pai eles são trocados de lugar e o algorítimo é aplicado recursivamente no índice do que antes era o maior elemento (onde agora esta o elemento que antes era o pai), faz isso até que o elemento esteja satisfazendo a condição de Heap de máximo

```
Buid-Max-Heap (A)
    tamanho-do-heap(A) ← comprimento(A)
    for (i ← comprimento(A)/2  downto 1)
        Max-Heapify(A, i)
```

A função Build-Max-Heap, como o nome já diz constroi um Heap de máximo, aplicando a função Max-Heapify partindo da metade até o primeiro elemento, ao final todo o vetor estará satisfazendo a condição de Heap de máximo e o primeiro elemento A[1] será o maior elemento do vetor

↳ tamanho-do-heap(A) recebe o comprimento(A) para ser usado na função Heapsort(A)

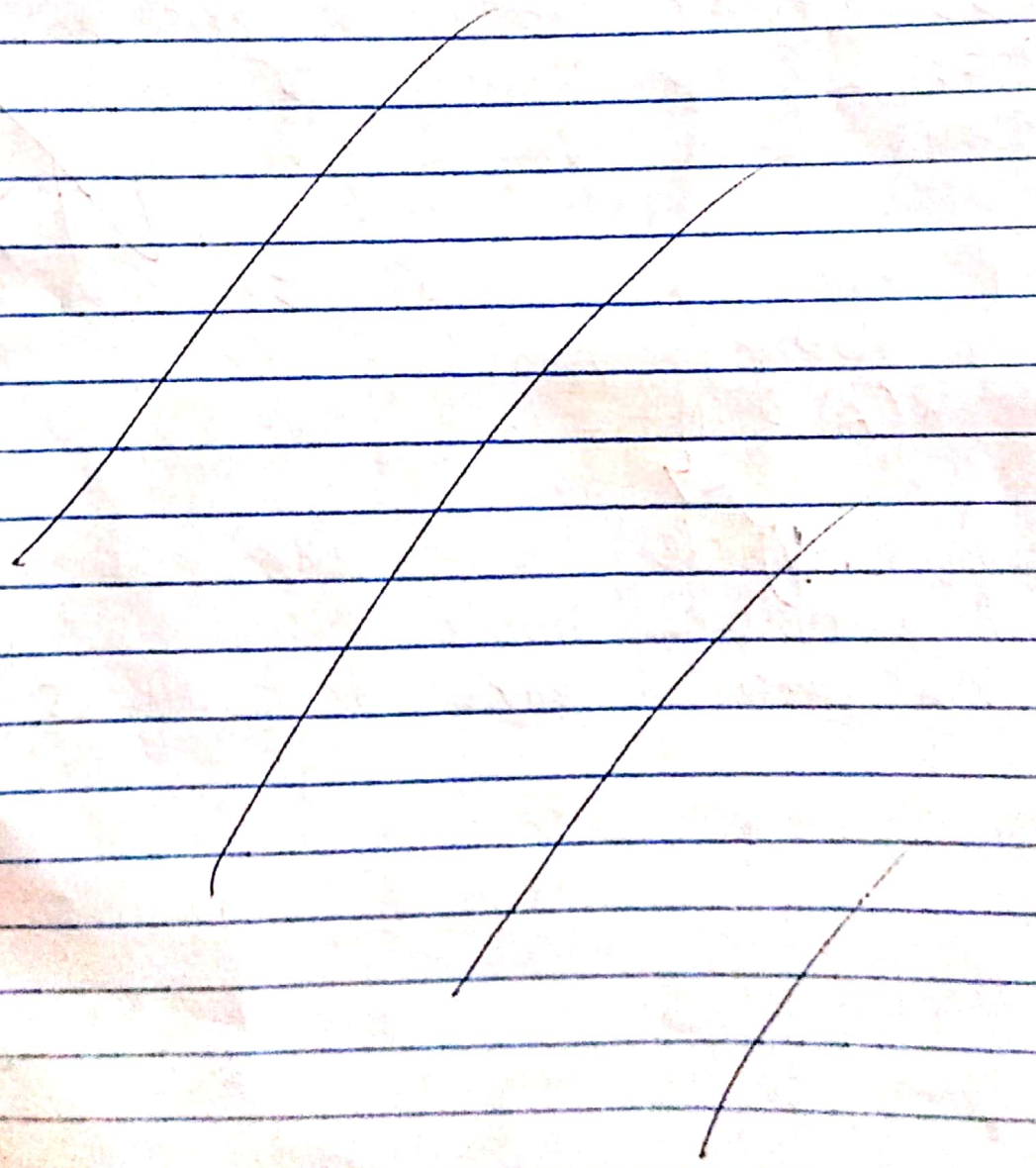
1) continuação II...

```
Heapsort (A)
    Build-Max-Heap (A)
    for (i ← Comprimento (A)  downto 2)
        trocar A[1] ↔ A[i]
        tamanho-do-heap (A) ← tamanho-do-heap(A)-1
        Max-Heapify (A, 1)
```

A função Heapsort começa chamando a função Build-Max-Heap(A) para transformar o vetor em um Heap de máximo, depois disso sabemos que o maior elemento do vetor é A[1], colocamos esse elemento na ultima posição e diminuímos o tamanho do Heap (para ignorarmos o ultimo elemento, pois ele já está na posição certa) ao fazermos isso nós também colocamos, o que antes era, o último elemento do vetor na primeira posição, quebrando a nossa condição de Heap de Maximo, mas como a mudança é pequena, pois foi feita em apenas um elemento a função Max-Heapify(A, 1) resolve o problema

Ao final do laço for o vetor estará ordenado.

(2) MergeSort segue o princípio "dividir para conquistar", onde o vetor é dividido em pedaços cada vez menores até que só reste vetores com 1 elemento ordenamos esses vetore pequenos recursivamente (do maior para o menor) elemento por elemento até que só reste o vetor principal ordenado



3

As formas de tratar colisões em tabelas de dispersão são

1º - Através da próxima posição consecutiva, onde o algoritimo testa se a casa do vetor está ocupada, se estiver testa a próxima e assim por diante até encontrar uma casa vazia.

2º - Através de uma segunda função de dispersão, dada por

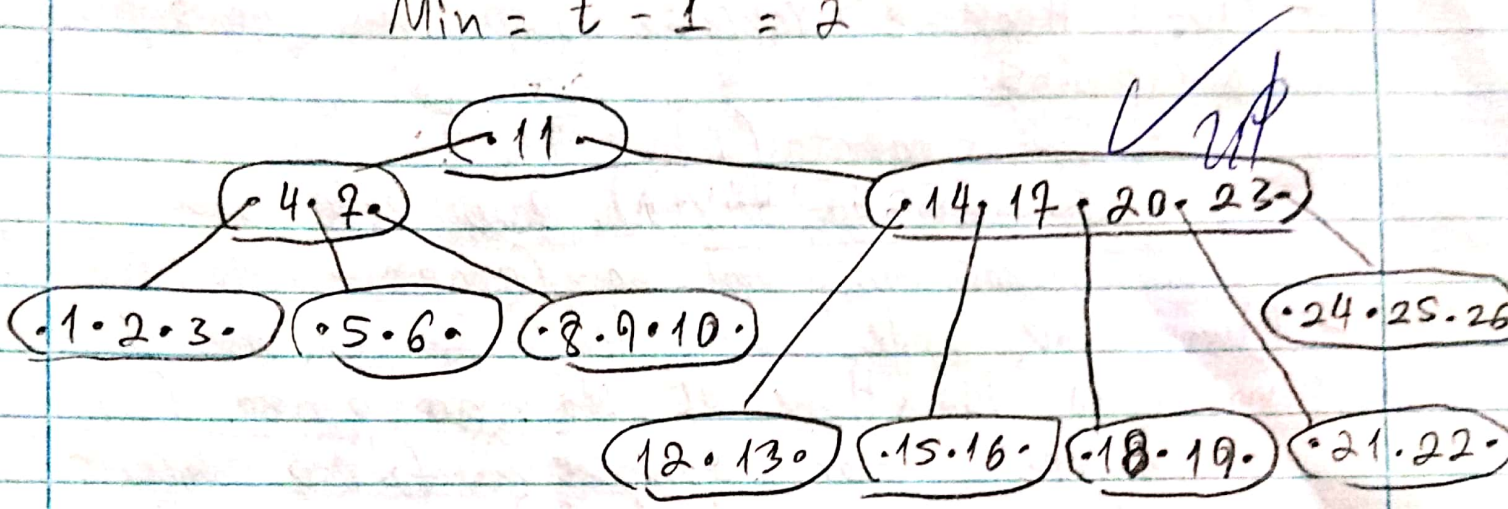
$h'(x) = N - 2 - x \bmod (N-2)$

3º - Através de listas encadeados, onde A estrutura hash passa a conter um ponteiro para estruturas

Q4

$t = 3$; $Max = 2t - 1 = 5$

$Min = t - 1 = 2$



Q5