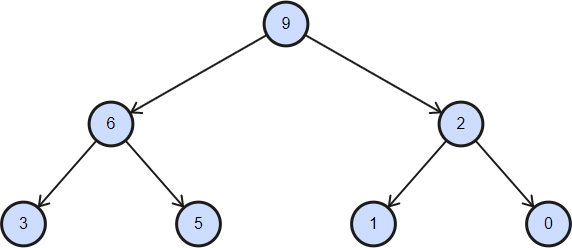
# Exercícios para P1 – Algoritmos e Estrutura de Dados II

1. Utilizando a notação Big-O a complexidade de uma busca sequencial ou linear é, no pior caso:
   1. O(n2) b) O(n) c) O(log n) d) O(1) e) O(n log n)
2. A complexidade de busca ao elemento prioritário em um heap máximo é, no pior caso:
   1. O(n2) b) O(n) c) O(log n) d) O(1) e) O(n log n)
3. Dado um array unidimensional X, contendo milhares de números inteiros não ordenados, a complexidade de um algoritmo que faz a contagem de números iguais a zero presentes em X é:
   1. O(n2) b) O(n) c) O(log n) d) O(1) e) O(n log n)
4. Mariana precisa armazenar dados referentes a cerca de um milhão de clientes. Cada cliente possui um código único que não se repete. Utilizando uma tabela Hash é possível conseguir acessar os dados de um cliente com complexidade de:
   1. O(n2) b) O(n) c) O(log n) d) O(1) e) O(n log n)
5. A complexidade do algoritmo de ordenação Bubble Sort no pior caso é:
   1. O(n2) b) O(n) c) O(log n) d) O(1) e) O(n log n)
6. Qual dos algoritmos de ordenação de arrays unidimensionais abaixo apresenta complexidade no melhor e pior caso O(*n* log *n*)?
   1. Bubble Sort b) Insertion Sort c) Merge Sort d) Quick Sort e) Random Sort
7. Um heap máximo é uma árvore binária (completa ou quase completa) da esquerda para a direita, onde a chave de cada nodo pai não pode ser menor que a chave dos filhos.

Por exemplo:



Considerando a árvore binária acima que representa um heap máximo, responda aos itens abaixo:

* 1. Apresente o array correspondente a esse heap:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 9 | 6 | 2 | 3 | 5 | 1 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |

* 1. Remova o elemento mais prioritário e apresente o heap final.

Desenho de relógio

Descrição gerada automaticamente com confiança média

1. Implemente o método inserir para inserir uma chave na tabela hash abaixo, utilizando o tratamento de colisões por endereçamento aberto. Caso o usuário tente inserir uma quantidade de chaves maior que o tamanho da tabela então deve retornar uma mensagem que não há mais espaço disponível.

Função de hash: **k mod 7**

# k é o valor da chave que pode ser qualquer número inteiro de 0 até 1000. O valor -1 na tabela indica que o a posição está vazia.

|  |  |
| --- | --- |
| **h(k)** | **k** |
| 0 | -1 |
| 1 | -1 |
| 2 | -1 |
| 3 | -1 |
| 4 | -1 |
| 5 | -1 |
| 6 | -1 |

Tabela:

public void inserir(int chave) throws Exception {

     if (!isValid(chave)) return;

    int hashIndex = chave % 7;

    int valueInIndex = hashTable[hashIndex];

    boolean free = valueInIndex == -1;

    if (free) {

        hashTable[hashIndex] = chave;

        System.out.printf("Chave %s inserida com sucesso na posição %d%n", chave, hashIndex);

        return;

    }

    if (valueInIndex == chave) {

        System.out.println("Chave já existe");

        return;

    }

    for (int i = 0; i < hashTable.length; i++)

        if (hashTable[i] == -1) {

            hashTable[i] = chave;

            return;

        }

    throw new Exception("Não há mais espaço disponível");

}

public boolean isValid(int value) {

    return (value >= 0 && value <= 1000);

}

1. Mostre a codificação de compressão RLE e a taxa de compressão para as cadeias de bits abaixo, conforme visto em aula. Utilize 4 bits para armazenar a contagem de repetições.
   1. 00000000011111100000011111

9x0’s + 6x1's + 6x0’s + 5x1's = 26 bits

1001 0110 0110 0101 --> 16/26 = 61%

* 1. 00000011111100000000111100

6x0's + 6x1's + 8x0's + 4x1's + 2x0's = 26 bits

0110 0110 1000 0100 0010 --> 20/26 = 76%

* 1. 11111111111111111110000000

15x1's + 0x0's + 4x1's + 7x0's = 26 bits

1111 0000 0100 0111 --> 16/26 = 61%

1. Apresente tabela de código e árvore Trie binária utilizando a codificação de Huffman correspondente às strings abaixo:
2. AMAR
3. ABRACADABRA! (resposta no slide)
4. COMPRESSAO

Exemplo:

|  |  |  |
| --- | --- | --- |
| Tabela Códigos | | |
| Char | Frequência | Código |
| **A** |  |  |
| **M** |  |  |
| **R** |  |  |

1. Construa uma trie para armazenar as seguintes strings, inseridas nessa ordema:

* AMORA
* AMOR
* AMEIXA
* MORANGO
* MANGA

1. Observe a implementação da estrutura de dados **Trie** abaixo, que armazena caracteres de A-Z e implemente o método Adicionar.

public class Trie {

private static final int *TAMANHO\_ALFABETO* = 26; /*/apenas letras maiusculas*

private class Nodo { private char valor; private int nivel; private Nodo[] proximo;

public Nodo() {

this.proximo = new Nodo[*TAMANHO\_ALFABETO*]; this.nivel=0;

}

}

private Nodo raiz; private int quantidade;

public Trie() { this.quantidade = 0; this.raiz = new Nodo();

}

public void adicionar(String chave){

*//****IMPLEMENTAR***

}

1. Considerando um plano cartesiano com limites (0,0) até (100,100) construa a árvore 2D-Tree e Quadtree de pontos para os pontos abaixo, na ordem em que estão listados:

A (30,30)

B (76,80)

C (15,80)

D (55,60)

E (10,10)

F (25,25)

G (85,05)

H (90,45)

I (65,15)

1. Observe a classe abaixo que implementa uma estrutura de dados 2D Tree para armazenar e pesquisar pontos em um plano cartesiano x, y.

Com base nessas definições implemente o método Adicionar da 2DTree conforme visto em aula.

public class KdTree { private class Nodo {

Ponto ponto; Nodo esquerda; Nodo direita; Ponto pai;

public Nodo(String rotulo, int x, int y) { this.ponto = new Ponto(rotulo, x, y);

}

public Nodo(Ponto p) { this.ponto = p;

}

}

private Nodo raiz;

public void adicionar(String rotulo, int x, int y) { Nodo novoNodo = new Nodo(rotulo, x, y);

**....implementar...**

**}**

}