**Resumo de Estrutura de dados**

**3ª unidade**

* **Árvores binárias de busca (BST):**

Usadas para buscas mais eficientes (O(logn)).

Todos os elementos à esquerda de um nó pai são menores do que ele e todos os a direta são maiores. Ou seja, para encontrarmos um elemento, precisamos apenas percorrer um dos lados da árvore. Isso se chama método **find().**

Para fazermos o método **insert(),** basta fazermos os mesmos passos do find até encontrarmos uma posição vazia de uma folha, então adicionamos o novo nó.

A **remoção** não é tão fácil. Se o nó removido for uma folha, então é só removê-lo. Se o nó removido tem apenas um filho, basta removermos o nó e relinkarmos o filho deste nó ao nó pai. O problema é quando o nó removido tem dois filhos, neste caso, temos que verificar os filhos dos seus filhos até encontrar um valor em que encaixe na nova posição, então relinkamos. No último caso, a complexidade deixa de ser O(logn) para O(n)!!!

* **Árvores de Adelson-Velskii & Landis (AVL):**

São árvores balanceadas com elementos à direita e à esquerda. Para balancearmos as árvores, temos de **rotacioná-las**. As rotações acontecem nas função de inserção e remoção.

* **Heaps e filas de prioridade:**

Uma Heap binária é uma estrutura similar a uma árvore que oferece:

Insert: O(log |Q|),

O(1): get max,

O(log |Q|): deletar e capturar max;

Heap property: Arvore em que todo nó armazena um elemento de maior prioridade que seus filhos. (**NÃO É BST/AVL)**.

Sheap property: todos níveis preenchidos, com exceção (talvez) do ultimo. O último nível é preenchido, obrigatoriamente, da esquerda para direita.

Na **inserção heap** , o nó é adicionado ao último nível mais a esquerda, então é comparado com seu pai, se sua prioridade for maior, então troca. E esse ciclo se repete. (O (logn)).

Na **remoção heap** da raiz, trocamos a raiz pelo útlimo elemento à direita da árvore e então vamos comparando com seus filhos e trocando. O(logn)

* **Árvore B:**

Árvores usadas em HDs ou memórias secundárias. Tentam pegar o máximo de informação em uma interação. Todo nó (página) tem quatro campos. **NÃO É ÁRVORE BINÁRIA ! POSSUI MAIS DE DOIS FILHOS.**

**GRAFOS E DÍGRAFOS**

Dígrafos possuem arestas dirigidas (---> ou 🡨), além de poder ter realimentações de arestas, enquanto grafos possuem apenas arestas sem sentido e sem realimentação.

Um **subgrafo** é uma parte de um grafo completo.

Um **corte** são as arestas de um subgrafo.

Um grafo **conexo** é um grafo completamente interligado.

Um grafo **completo** é um grafo em que todos os vértices possui todas as possíveis arestas.

Um grafo **complementar** é um grafo com apenas arestas não existentes de um outro grafo.

Uma **floresta** é um grafo sem ciclos.

Uma **árvore** é uma floresta conexa.

**MENORES CAMINHOS EM GRAFOS**

Temos que considerar que o grafo é conexo e sem custos negativos.

* **Algorítmo de Dijstrka:**

No pior caso, complexidade de O(n²), em dígrafos gerais O(m + nlogn).

Utiliza de arrays de distância e de predecessores.

Nós temos de acomodar todos os nós, atualizando sempre o array de distâncias e predecessores (caso a distância seja menor do que a outra que existia antes). Depois de testarmos todos os nós, o algoritmo termina e finalizamos os arrays completos e prontos para uso com o menor caminho possível.

* **Algoritmo de Floyd-Warshall’s:**

Sem ciclos negativos.

Duas matrizes n x m, para armazenar predecessores e distâncias.

No pior caso, O(n³).