RELATÓRIO DE ALGORITMOS E ESTRUTURAS DE DADOS

Projeto 2 - SET

Integrantes:

Frederico Scheffel Oliveira - 15452718

Pedro Henrique Perez Dias - 15484075

INTRODUÇÃO

O intuito do projeto é implementar um TAD Conjunto que permita ao usuário escolher entre dois outros TADs, que no caso de nosso projeto, foram escolhidos a Árvore AVL e a Lista Sequencial Ordenada. Os TADs escolhidos devem realizar as operações básicas, sendo elas: **Criar um conjunto, apagar um conjunto, inserir um elemento no conjunto, remover um elemento do conjunto e imprimir os elementos de um conjunto.** Além de realizar as operações específicas do conjunto, sendo elas: **Verificar se um elemento está presente no conjunto, realizar a união entre dois conjuntos e realizar a interseção entre dois conjuntos.** Sendo que os TADs implementados possuem a menor complexidade computacional possível, considerando que foram pensadas 2 estruturas diferentes, a fim de gerar maior grau de comparação.

TADS IMPLEMENTADOS

- **ÁRVORE AVL**

  Foi escolhida a árvore AVL pois ela é uma árvore binária de busca, ou seja, é possível utilizar a busca binária para realizar a busca de elemento dentro de um certo conjunto, sendo bastante eficiente, uma vez que possui uma complexidade de O(log n), podendo se utilizar essa busca para inserção, remoção e verificação se um certo elemento pertence ao conjunto.

❖ **COMPLEXIDADE DAS OPERAÇÕES BÁSICAS**

➢ **avl_criar:** Realiza a alocação de memória necessária para a árvore AVL. Como é realizada uma vez por chamada, possui **complexidade geral de O(1).**

➢ **avl_apagar:** Realiza a liberação da memória ocupada por cada nó da árvore AVL. Como a função percorre todos os nós da

árvore, **sua complexidade geral é O(n)** na função auxiliar, que predomina sobre a principal.

- ➢ **avl_inserir:** Insere um nó na árvore AVL. Realiza uma busca pela posição que o nó estaria na árvore e o insere, realizando rotações se necessário. **Sua complexidade geral é de O(log n)** devido à natureza da árvore binária
- ➢ **avl_remover:** Remove um nó específico da árvore AVL. Devido à natureza da árvore AVL, qualquer busca procura o valor em um espaço com metade do tamanho do anterior. Por isso podemos afirmar que **sua complexidade é O(log n).**
- ➢ **avl_imprimir:** Escreva na tela todos os valores dos nós da árvore AVL. Possui 1 operação na função principal (verificação) e 2 operações na função auxiliar (verificação e print). Como a função percorre todos os nós da árvore, **sua complexidade geral é de O(n)**.

❖ COMPLEXIDADE DAS OPERAÇÕES ESPECÍFICAS

- ➢ **avl_pertence:** Realiza uma busca na árvore AVL. Funciona de maneira análoga à remoção, procurando em espaços do vetor cada vez menores. Por isso, podemos afirmar que a **complexidade geral é de O(log n)**
- ➢ **avl_uniao:** Realiza a inserção de cada chave das duas árvores em uma terceira árvore de união, respeitando o conceito de união de conjunto, o qual junta todos os elementos de outros dois conjuntos, porém não repetindo valores caso certo elemento apareça em ambos os conjuntos. Em seu pior caso, realiza todas as inserções de A e todas as inserções de B na árvore nova, que passaria a ter o tamanho das duas somadas.

A complexidade pode ser calculada como O((nA + nB) log (nA + nB)), sendo nA o total de nós na árvore A e nB o total em B. Isso se deve ao fato de a complexidade da inserção ser log(n), para o tamanho máximo da árvore e de estarem sendo inseridos nA + nB na mesma.

➢ **avl_interseccao:** Realiza a inserção de cada chave das duas árvores em uma terceira árvore de intersecção, respeitando o conceito de intersecção de conjunto, a qual procura por elementos iguais entre dois conjuntos diferentes. Para cada nó da árvore A, são realizadas uma busca na árvore B (de complexidade O(log nB)) e uma inserção de complexidade O(log nA). Sua complexidade é dada por O((nA * log nB) + (nA * lognA)), que pode ser simplificada para **O(nA * log nB)**.

- **LISTA SEQUENCIAL ORDENADA**

Foi escolhida a lista sequencial ordenada, uma vez que por já estar ordenada, suas operações ficam muito mais simples, devido a possibilidade de utilizar a busca binária para filtrar o conjunto, agilizando na busca, inserção e remoção de elementos.

❖ **COMPLEXIDADE DAS OPERAÇÕES BÁSICAS**

➢ **lista_criar:** Simplesmente aloca a memória para a lista, inicializando as variáveis início, fim e tamanho, portanto sua **complexidade final é apenas O(1)**.

➢ **lista_apagar:** Verifica se a lista é nula e libera a memória alocada, sendo a verificação O(1) e a liberação O(1), portanto sua **complexidade final é apenas O(1)**.

➢ **lista_inserir:** Verifica se a lista está cheia, com complexidade

de O(1) e se o elemento já está presente usando lista_pertence, que possui complexidade de O(log n), após isso, encontra a posição correta para inserir o valor por meio de uma busca binária O(log n), e por fim desloca os elementos para que o novo elemento seja inserido O(n) e insere o elemento O(1). Como O(n) é dominante, a **complexidade final dessa função é de O(n)**.

- ➢ **lista_remover:** Verifica as condições iniciais com O(1) e se o elemento pertence ao conjunto usando o lista_pertence, de complexidade O(log n), caso ele esteja no conjunto ele procura exatamente a posição desse elemento por meio de uma busca binária O(log n), após isso, desloca os elementos para esquerda, com fim de preencher o espaço que foi removido, com complexidade O(n) e atualiza os índices com O(1). Como o deslocamento possui complexidade dominante de O(n), a **complexidade final é 0(n)**.
- ➢ **lista_imprimir:** Imprime os valores, iterando pela lista inteira para imprimir cada um deles, com complexidade de O(n), portanto sua **complexidade final é de O(n)**.

❖ **COMPLEXIDADE DAS OPERAÇÕES ESPECÍFICAS**

- ➢ **lista_pertence:** Realiza uma busca binária para verificar se um elemento pertence ou não ao conjunto, a cada iteração, divide o espaço de busca pela metade, ficando proporcional a O(log n), sendo n o número de elementos da lista, resultando assim em uma **complexidade final de O(log n)**.
- ➢ **lista_uniao:** Itera de maneira simultânea os dois conjuntos com dois índices diferentes, sendo n o tamanho da primeiro conjunto passado como parâmetro e m sendo o tamanho do segundo conjunto, verificando qual dos dois conjuntos possui o

menor valor para ser inserido primeiro no conjunto da união, tendo complexidade O(n + m). Ao fim, caso ainda haja elementos não percorridos em algum dos conjuntos, se adiciona ao fim da união o conjunto que sobrou, podendo ser O(n), ou O(m). Como 0(n + m) é dominante, **sua complexidade final é O(n + m)**.

- **lista_interseccao:** Assim como na união, se itera simultaneamente os dois conjuntos com índices diferentes, enquanto verifica se algum dos elementos são iguais, para assim adicioná-lo ao conjunto interseção, sendo percorrido até que um dos dois conjuntos chegue ao seu fim, possuindo uma complexidade de O(n + m) novamente, sendo n a quantidade de elementos do primeiro conjunto e m a quantidade do segundo conjunto. Como as verificações são O(1), 0(n + m) é dominante, portanto a **complexidade final é O(n + m)**.