Relatório Projeto 2 - Disciplina Algoritmos e Estrutura de Dados

Nome: Luis henrique Ponciano dos Santos NUSP: 15577760
Nome: Gabriel de Araujo Lima NUSP: 14571376

1. TAD CONJUNTO

Para implementação do TAD CONJUNTO, utilizamos as seguintes estruturas de dados: árvore binária balanceada AVL e Left-Leaning Red-Black Tree (LLRBT).

1.1 **JUSTIFICATIVA DAS ESTRUTURAS ESCOLHIDAS**

A escolha da AVL e LLRB foi motivada por 2 principais motivos: uso de memória e complexidade computacional.

Em primeiro lugar, é possível destacar que como as árvores são implementadas dinamicamente através de nós, a memória é alocada de maneira suficiente para a quantidade necessária de elementos de um conjunto. Dessa forma, o desperdício de memória é evitado.

Por outro lado, a estrutura binária das árvores possibilita a busca binária, extremamente importante para garantir buscas eficientes. Assim, na operação pertence, por exemplo, a busca do elemento no conjunto ocorre de maneira eficiente.

Além disso, as operações de união e interseção podem ser implementadas de maneira mais simples, por conta da natureza das árvores. Nesse sentido, na união, por exemplo, é possível utilizar o fato de que em árvores não é possível inserir elementos repetidos, propriedade fundamental da união.

1.2 **COMPLEXIDADE DE TEMPO (Big Oh) de cada uma das operações**

Operações básicas:

<u>Criar (um conjunto)</u>

A operação criar é dada pelo seguinte protótipo no TAD CONJUNTO:

```
CONJUNTO *conjunto_criar(int estrutura);
```

Essa função aloca memória para o conjunto e retorna o ponteiro para ele.

Para determinar a complexidade final, somamos as complexidades de todas as partes relevantes: alocação de memória O(1) , comparações e atribuições O(1) + O(1) + O(1) + O(1), chamada de avl_criar ou llrb_criar: O(1). Assim a complexidade é dada por O(1)+O(1)+O(1)+O(1)+O(1)=O(1)

Apagar (um conjunto)

A operação “apagar” é dada pelo seguinte protótipo no TAD CONJUNTO:

```
void conjunto_apagar(CONJUNTO **conjunto);
```

A função conjunto_apagar tem complexidade O(n), onde n é o número de nós no conjunto representado pela árvore (AVL ou LLRB). Ela verifica inicialmente se o ponteiro do conjunto é válido, o que tem complexidade O(1). Em seguida, dependendo do tipo de conjunto, chama a função avl_apagar ou llrb_destruir. Ambas as funções realizam um percurso pós-ordem para liberar os nós da árvore. No percurso, cada nó é visitado exatamente uma vez, e as operações de liberação de memória possuem custo constante O(1) por nó. Assim, a liberação completa da árvore, seja AVL ou LLRB, possui complexidade O(n). Portanto, a complexidade total da função conjunto_apagar é O(n).

Inserir (um elemento em um conjunto)

A operação “inserir” é dada pelo seguinte protótipo no TAD CONJUNTO:

```
bool conjunto_inserir_elemento(CONJUNTO *conjunto, int chave);
```

No início é verificado qual estrutura de dados é escolhida. Caso seja AVL, é chamada a função de inserir elemento na AVL. Caso seja LLRB, é chamada a função de inserir elemento na LLRB. A inserção na AVL é dada por uma busca para encontrar a posição a ser inserido o elemento e um rebalanceamento após a inserção. Assim a complexidade é dada pela busca (binária). Por sua vez, a inserção na LLRB é dada também por uma busca para encontrar o lugar de inserção, além de um rebalanceamento após a inserção.

Portanto, a complexidade da inserção no conjunto pode ser calculado através da inserção na AVL, dada pela busca (binária) O(log n) somada com o rebalanceamento, no qual é feito cálculos de alturas e balancear, onde em cada nível o custo é O(1), mas como ocorre para cada nível da árvore, então multiplicamos pela altura da árvore O(log n). Por outro lado, a inserção na LLRB, dada pela busca (binária) (O log n) somada com o rebalanceamento, dado pelas rotações à esquerda O(1), direita O(1) e inversão de cores O(1).

Assim, a complexidade total é dada pela maior entre as complexidades inserção entre AVL e LLRB. Nesse caso, a complexidade de AVL demanda maior poder computacional por conta do rebalanceamento e cálculo dos novos fatores de balanceamento. Dessa forma, a complexidade é dada por

O(log n) + O(1) * O (log n) = O (log n)

Remover (um elemento de um conjunto)

A operação “remover” é dada pelo seguinte protótipo no TAD CONJUNTO:

```
bool conjunto_remover_elemento(CONJUNTO *conjunto, int chave);
```

A complexidade da função conjunto_remover_elemento depende do tipo de estrutura de dados utilizado para representar o conjunto. Se a estrutura for uma árvore AVL, a remoção tem complexidade da remoção somado com o rebalanceamento, que resulta em O(log n), onde n é o número de elementos na árvore, pois a busca e as operações de balanceamento ocorrem em tempo logarítmico, por conta da busca (binária). O mesmo se aplica ao caso de uma árvore LLRB, que também tem complexidade O(log n) devido à busca realizada para encontrar o elemento (busca binária), de balanceamento da árvore, que consiste em rotações e inversões de cor, cuja complexidade é dada por O(1). Assim, a complexidade da função como um todo é O(log n), independentemente da estrutura utilizada, considerando que as operações internas de remoção em ambas as árvores seguem o mesmo comportamento assintótico.

Imprimir (os elementos armazenados no conjunto)

A operação "imprimir" é dada pelo seguinte protótipo no TAD CONJUNTO:

```
void conjunto_imprimir(CONJUNTO *conjunto);
```

São realizadas 3 verificações, como a que decide entre AVL e LLRB. Além disso, é chamada função auxiliar para imprimir cada árvore, onde é realizado um percurso, passando por cada um dos n nós da árvore. Assim, tem-se: verificações e decisões: **O(1), O(1), O(1);** percurso: O(n). A complexidade final é

O(1)+O(n)=O(n)O(1) + O(n) = O(n)O(1)+O(n)=O(n)

Operações específicas:

Pertence (um elemento está presente ou não no conjunto)

A operação "pertence" foi implementada através do seguinte protótipo no TAD CONJUNTO:

```
bool conjunto_pertence(CONJUNTO *conjunto, int chave);
```

A função funciona da seguinte forma: o tipo de estrutura de dado utilizado é analisado. Se for AVL, é feita uma busca na AVL com o elemento. Caso seja LLRB, é feita uma busca na LLRB com o elemento.

A complexidade de uma busca em árvores balanceadas, no caso AVL e LLRB é dada pela busca binária, assim O(log n).

Dessa forma, a complexidade total da operação pertence é a comparação para saber o tipo de estrutura, que é constante O(1), somado com a complexidade da busca binária de uma das duas árvores, O(log n). Assim, a complexidade total é

O(1) + O(log n) = O(log n)

União entre dois conjuntos

A operação “união” foi implementada através do seguinte protótipo no TAD CONJUNTO:

```
CONJUNTO *conjunto_uniao(CONJUNTO *conjuntoA, CONJUNTO *conjuntoB);
```

A função funciona da seguinte forma: depois de 3 verificações iniciais de ponteiros para NULL, é analisada qual estrutura de dados foi escolhida. Após essa análise, criamos o conjunto C, que armazenará os valores da união entre os conjuntos A e B. Dependendo da estrutura escolhida, chamamos duas funções (são análogas para as duas árvores). Caso seja AVL, por exemplo, chamamos as duas funções a seguir:

```
avl_transferir_elementos(conjuntoA->avl_conjunto,conjuntoC->avl_conjunto);
avl_transferir_elementos(conjuntoB->avl_conjunto,conjuntoC->avl_conjunto);
```

A função avl_transferir_elementos realiza a união propriamente dita. Nessa função, percorremos em ordem o primeiro conjunto e para cada nó visitado inserimos seu elemento na segunda árvore.

No final teremos um conjunto contendo tanto os elementos da segunda árvore quanto os da primeira árvore, sem repeti-los, por conta da natureza da inserção de árvores, já que elementos repetidos não são inseridos.

Chamamos a função duas vezes para inserir primeiro os elementos de A no conjunto C (união) e depois inserir os elementos de B no conjunto C (união). Portanto, teremos no final o conjunto C contendo tanto os elementos de A quanto de B sem repeti-los.

Essa composição garante uma eficiência maior na união por conta de um motivo principal: natureza da inserção em árvores, já que quando um elemento repetido é visitado, ele não será inserido na árvore da união. Isso evita cálculos computacionais extras para não inserir elementos repetidos ou tirar os repetidos depois de inseri-los.

A complexidade da união é dada pela soma do percurso em ordem da árvore A com cada inserção para cada nó visitado. Por sua vez, a complexidade da inserção se resume à busca (binária) do elemento para achar a posição de inserção. O custo da inserção, após encontrar o lugar, é constante.

Portanto, temos uma inserção (busca) para cada um dos n1 nós visitados na primeira árvore. Considerando que a segunda árvore tem n2 nós, nota-se que a complexidade da busca binária é dada por O(log n2). Assim, a complexidade da união é dada pelo produto de cada nó por uma inserção somado com as 3 verificações iniciais da função. Porém, como chamamos duas vezes a função avl_transfeir_elemento ou sua análoga, llrb_transferir_elemento, temos:

$$O(2 * n1 * \log n2) + 3 * O(1) = O(n1 \log n2)$$

Intersecção entre dois conjuntos

A operação “intersecção” foi implementada através do seguinte protótipo no TAD CONJUNTO:

```
CONJUNTO *conjunto_interseccao(CONJUNTO *conjuntoA, CONJUNTO *conjuntoB);
```

A função funciona da seguinte forma: após 2 verificações iniciais, a estrutura de dados escolhida para o conjunto é analisada. Após a comparação, de maneira análoga (tanto para a AVL como para a LLRB), criamos o conjunto C (intersecção) e chamamos a função a seguir (foi exemplificado caso a estrutura seja AVL):

```
avl_interseccao_elementos(conjuntoA->avl_conjunto,conjuntoB->avl_conjunto,
conjuntoC->avl_conjunto);
```

Essa função realiza a intersecção propriamente dita entre o conjunto A e o conjunto B, sendo a intersecção armazenada no conjunto C, também passado como argumento. Após 3 verificações, essa função chama uma outra, auxiliar, que depois de verificações realiza um percurso em ordem na árvore A, onde para cada nó visitado de A é verificado se ele pertence à árvore B através de uma busca. Assim, se o nó visitado de A também pertencer a B, então ele é inserido em C. Caso contrário, o percurso continua para a subárvore à direita.

Dessa forma, a complexidade da intersecção é dada pela soma de uma série de verificações (constante) e a busca (binária) para cada um dos n1 nós visitados do conjunto A para verificar se ele também pertence à B, realizando a inserção caso pertença. Por fim, cada busca em B possui a complexidade dada pelo tamanho n2 (quantidade de nós) da árvore B. Considerando a árvore A com tamanho n1, temos:

Percorrer: O(n1)

Busca: O(log n2)

Inserção: O(log n2) - A inserção tem complexidade equivalente à busca

Considerando o pior caso, em que cada elemento de A está presente em B, então será realizado uma inserção para cada nó visitado. Assim, a complexidade total é dada por:

$O(n1 * (\log n2 + \log n2)) + O(1) = O(n1 \log n2)$