ICMC - Instituto de Ciências Matemáticas e da Computação

Disciplina: SCC0202 - Algoritmo e Estrutura de Dados I

Projeto 2

Aluno: Alec Campos Aoki (15436800)
Aluno: Jhonatan Barboza da Silva (15645049)

Descrição do projeto:

Conjuntos, conceito fundamental da Teoria dos Conjuntos, representam coleções de objetos chamados elementos, cuja relação de pertinência define se pertencem ou não ao conjunto. Com ampla aplicação em sistemas computacionais, os conjuntos aparecem em diversas linguagens de programação como estruturas de dados e são utilizados em problemas matemáticos, de otimização e estatística.

Objetivo:

O objetivo do projeto foi implementar três TADs, visando desenvolver tanto as operações básicas de cada TAD quanto às operações específicas do conjunto, como pertinência, união e interseção, otimizando ao máximo a complexidade computacional. Para isso, escolhemos implementar os TADs utilizando uma lista com busca binária, uma árvore binária de busca AVL e um TAD Conjunto, responsável por gerenciar as operações durante a execução.

Justificativa:

A escolha de implementar o TAD com uma lista utilizando busca binária deve-se ao fato de que a operação de busca pode ser realizada em O(\log n). Apesar de as operações de inserção e remoção apresentarem complexidade O(n), no geral, o desempenho é satisfatório em cenários onde a busca é predominante. Já a árvore binária de busca AVL foi escolhida por sua eficiência consistente, com complexidade O(\log n) para as operações de busca, inserção e remoção, garantindo bom desempenho mesmo no pior caso.

Principais operações:

Main:

A função main gerencia as operações e o fluxo do programa. Inicialmente, ela solicita ao usuário que escolha a estrutura de dados a ser utilizada (0 para lista e 1 para ABB AVL).

Com base nessa escolha, dois conjuntos (conjA e conjB) são criados utilizando a estrutura selecionada.

Em seguida, o programa lê o tamanho dos conjuntos A e B. Caso ambos os tamanhos sejam zero, uma mensagem informando que os conjuntos estão vazios é exibida, e o programa é encerrado. Caso contrário, os elementos de cada conjunto são lidos e inseridos nas respectivas estruturas de dados por meio da função conjunto_inserir.

Posteriormente, o programa solicita ao usuário que selecione a operação a ser realizada entre os conjuntos. As operações disponíveis são: verificar a pertinência de um elemento a um conjunto (PERTENCE), realizar a união dos conjuntos (UNIAO) ou calcular a interseção dos conjuntos (INTERSEC). Com base na operação escolhida, a função correspondente é executada para obter e exibir o resultado.

Ao final do programa, os conjuntos criados são apagados da memória utilizando a função conjunto_apagar, e o programa é encerrado com sucesso.

TAD Conjunto

O código importa os arquivos list.h, avl.h e conjunto.h e define a estrutura conjunto. Essa estrutura contém o campo TAD, que indica o tipo de estrutura de dados utilizada (lista ou árvore binária), o tamanho do conjunto e dois ponteiros: um para a lista (conjuntoLista) e outro para a árvore binária (conjuntoAVL). Apenas um desses ponteiros será utilizado, dependendo da estrutura escolhida.

A ideia principal das funções é gerenciar as chamadas para as funções específicas da lista ou da árvore binária, garantindo que as operações sejam executadas de acordo com a estrutura de dados escolhida.

Descrição das Funções

1. **conjunto_criar**: Aloca espaço na memória para criar um conjunto, inicializando-o conforme o tipo de estrutura especificado (lista ou árvore binária).

2. **conjunto_apagar**: Libera a memória do conjunto, utilizando as funções específicas da estrutura escolhida.

3. **conjunto_inserir**: Insere um elemento no conjunto e incrementa seu tamanho.

4. **conjunto_remover**: Remove um elemento específico do conjunto, decrementa seu tamanho e retorna o elemento removido.

5. **conjunto_imprimir**: Imprime os elementos do conjunto utilizando a função correspondente à estrutura de dados.

6. **conjunto_pertence**: Verifica se um elemento pertence ao conjunto. Retorna true se o elemento for encontrado e false caso contrário.

7. **conjunto_uniao**: Realiza a união de dois conjuntos.

    - Verifica se ambos não estão vazios.
    - Cria um novo conjunto e insere os elementos do conjunto A nele.
    - Para listas, insere diretamente os elementos de B, pois a função lista_inserir evita duplicatas.
    - Para árvores, é necessário ajustar a lógica para lidar com a remoção e inserção de elementos (essa parte está pendente no código).

8. **conjunto_interseccao**: Calcula a interseção de dois conjuntos.

    - Verifica se ambos não estão vazios.
    - Cria um novo conjunto e insere elementos que pertencem simultaneamente aos dois conjuntos originais.
    - Para árvores, segue a mesma lógica de manipulação de cópias utilizada na união.

9. **conjunto_copiar**: Cria uma cópia de um conjunto, alocando um novo espaço na memória e copiando os elementos da estrutura original.

TAD Lista Sequencial

O objetivo do arquivo lista.c é implementar uma lista sequencial ordenada com suporte a busca binária. Ele contém as principais funções relacionadas à manipulação da lista, como inserção, remoção, busca binária e outras operações úteis. Abaixo segue uma breve descrição de cada função:

Descrição das Funções

1. **lista_criar**

    - Aloca espaço na memória para um ponteiro do tipo lista, inicializa o tamanho como 0 e retorna a lista criada.

2. **lista_apagar**

    - Libera a memória da lista passada como argumento e define o ponteiro para NULL.

3. **lista_inserir**

    - Insere elementos em ordem crescente. Por se tratar de uma lista ordenada, essa função não permite a inserção de elementos repetidos. Após a inserção, incrementa o tamanho da lista e retorna `true` se a operação for bem-sucedida.

4. **lista_remover**

- Utiliza a busca binária para localizar o elemento a ser removido. Após encontrar o elemento, remove-o e desloca os elementos subsequentes para a esquerda. No final, decrementa o tamanho da lista e retorna o elemento removido.

5. **lista_imprimir**

    - Imprime os elementos da lista passada como parâmetro.

6. **lista_busca**

    - Chama a função buscaBinaria para encontrar a chave especificada. A busca binária funciona de forma recursiva, dividindo a lista ao meio sucessivamente até localizar o elemento. Caso a busca seja bem-sucedida, retorna o elemento; caso contrário, retorna um erro.

7. **lista_copiar**

    - Faz uma cópia da lista passada como parâmetro e retorna a nova lista criada.

8. **lista_consulta**

    - Permite consultar elementos pelo índice, em vez da chave. Retorna o elemento localizado no índice especificado como parâmetro.

Essa implementação prioriza eficiência, especialmente nas operações de busca e inserção, aproveitando as características de uma lista ordenada e a busca binária.

TAD Árvore Binária de Busca AVL

O objetivo do arquivo AVL.c é implementar uma Árvore Binária de Busca AVL (uma árvore de busca balanceada). A característica principal dessa estrutura é o balanceamento automático, que garante que a diferença entre as alturas das subárvores esquerda e direita de qualquer nó nunca seja maior que 1. Caso essa condição seja violada após uma inserção ou remoção, a árvore realiza rotações para corrigir o balanceamento.

A implementação inclui definições auxiliares para criar, remover, balancear e acessar os nós, além das funções principais do TAD AVL.

Descrição das Funções

1. **avl_criar**

    - Aloca espaço na memória para uma árvore AVL, inicializa a raiz como `NULL` e o tamanho como 0, e retorna a estrutura criada.

2. **avl_apagar**

- Remove todos os nós da árvore de forma recursiva em pós-ordem, utilizando a função auxiliar no_apagar_recursivo, e desaloca a estrutura AVL.

3. **avl_inserir**

    - Insere um elemento na árvore AVL, mantendo o balanceamento. Cria um novo nó com a chave a ser inserida e utiliza a função auxiliar avl_inserir_no para realizar a inserção como em uma ABB e corrigir o balanceamento através de rotações, se necessário.

4. **avl_remover**

    - Remove um nó que contém a chave especificada. Lida com os três casos de remoção (nó folha, nó com uma subárvore e nó com duas subárvores) usando a função auxiliar avl_remover_no, que também realiza as rotações necessárias para restaurar o balanceamento.

5. **avl_get_altura**

    - Calcula e retorna a altura da árvore ou de uma subárvore específica.

6. **avl_get_tamanho**

    - Retorna o número total de nós na árvore.

7. **avl_copiar**

    - Cria uma cópia profunda da árvore AVL, duplicando todos os nós enquanto mantém a mesma estrutura, com o auxílio da função recursiva no_copiar_recursivo.

8. **avl_imprimir**

    - Imprime os elementos da árvore em ordem crescente (percurso in-ordem), utilizando a função auxiliar avl_imprimir_arv.

9. **avl_busca**

    - Tenta encontrar a chave especificada na árvore, chamando a função auxiliar busca_binaria_avl, que realiza uma busca binária recursiva até encontrar o elemento ou retornar um erro.

10. **avl_get_chave_raiz**

    - Retorna a chave armazenada na raiz da árvore ou um erro, caso a árvore esteja vazia.

A implementação assegura que as operações de busca, inserção e remoção sejam executadas de forma eficiente, com complexidade garantida de O(\log n) no pior caso, graças ao balanceamento automático proporcionado pelas rotações AVL.

Complexidade das operação:

O objetivo é analisar separadamente a complexidade de cada função implementada, considerando apenas o código presente em cada uma delas. Para funções que realizam chamadas a TADs externos, a análise se limitará à lógica interna da função, desconsiderando a complexidade das funções chamadas. Ao final, será feito um levantamento geral para determinar a complexidade predominante de cada operação.

Main:

Na função main, as operações realizadas têm complexidade diretamente relacionada ao número de elementos nos conjuntos e ao tipo de estrutura escolhida. Segue a análise:

1. **Leitura do TAD**:

    - A escolha da estrutura de dados é uma simples leitura de entrada e tem complexidade O(1).

2. **Leitura dos Tamanhos dos Conjuntos (tamA e tamB):**

    - São realizadas duas operações de leitura, cada uma com complexidade O(1).

3. **Inserção de Elementos nos Conjuntos (conjunto_inserir):**

    - A main executa dois laços de leitura e inserção:
        - Para tamA elementos no conjunto A, a complexidade é O(tamA + C inserir).
        - Para tamB elementos no conjunto B, a complexidade é O(tamB + C inserir).
    - A complexidade final depende da implementação da função conjunto_inserir.

4. **Escolha da Operação e Execução**:

    - A leitura da operação tem complexidade O(1).
    - A execução da operação (pertence, união ou interseção) depende do código interno de cada função chamada, sendo detalhada posteriormente.

5. **Liberação dos Conjuntos (conjunto_apagar)**:

- São chamadas duas vezes para liberar os conjuntos A e B. A complexidade depende do número de elementos nos conjuntos e da implementação da função conjunto_apagar.

Resumo Geral da Main:
A complexidade da função main é dominada pelo custo da leitura e inserção dos elementos, além das operações executadas sobre os conjuntos. Estes detalhes são dependentes das implementações específicas das funções chamadas.

TAD Conjunto:

A análise a seguir considera apenas a lógica implementada no arquivo conjunto.c, sem detalhar as funções chamadas (como lista_inserir, avl_inserir, entre outras). A complexidade destas funções será especificada posteriormente.

1. **conjunto_criar**

    - **Descrição**: Aloca memória para o conjunto e inicializa o ponteiro correspondente à estrutura escolhida (lista ou árvore).
    - **Operações principais**:
        - Alocação de memória: O(1).
        - Inicialização de ponteiros: O(1).
    - **Complexidade Total**: O(1).

2. **conjunto_apagar**

    - **Descrição**: Libera a memória ocupada pelo conjunto.
    - **Operações principais**:
        - Liberação do conjunto com chamada à função específica (detalhada posteriormente).
        - Liberação da memória do próprio conjunto.
    - **Complexidade Total**: O(1), desconsiderando a complexidade das funções chamadas.

3. **conjunto_inserir**

    - **Descrição**: Insere um elemento no conjunto e incrementa o tamanho.
    - **Operações principais**:
        - Verifica o tipo de estrutura e chama a função correspondente para inserção.
        - Incrementa o tamanho do conjunto.
    - **Complexidade Total**: O(1), desconsiderando a complexidade da função chamada.

4. **conjunto_remover**

    - **Descrição**: Remove um elemento do conjunto e decrementa o tamanho.

- **Operações principais**:
    - Verifica o tipo de estrutura e chama a função correspondente para remoção.
    - Atualiza o tamanho do conjunto.
- **Complexidade Total**: O(1), desconsiderando a complexidade da função chamada.

5. **conjunto_imprimir**

    - **Descrição**: Imprime os elementos do conjunto.
    - **Operações principais**:
        - Verifica o tipo de estrutura e chama a função de impressão correspondente.
    - **Complexidade Total**: O(1), desconsiderando a complexidade da função chamada.

6. **conjunto_pertence**

    - **Descrição**: Verifica se um elemento pertence ao conjunto.
    - **Operações principais**:
        - Chama a função de busca correspondente à estrutura escolhida.
        - Retorna o resultado da busca.
    - **Complexidade Total**: O(1), desconsiderando a complexidade da função chamada.

7. **conjunto_uniao**

    - **Descrição**: Realiza a união de dois conjuntos.
    - **Operações principais**:
        - Copia o conjunto A.
        - Chama as funções de inserção para adicionar os elementos do conjunto B.
    - **Complexidade Total**: O(1), desconsiderando a complexidade das funções chamadas.

8. **conjunto_interseccao**

    - **Descrição**: Calcula a interseção de dois conjuntos.
    - **Operações principais**:
        - Cria um novo conjunto.
        - Chama a função de busca para verificar a presença de elementos nos dois conjuntos.
        - Insere os elementos encontrados na interseção.
    - **Complexidade Total**: O(1), desconsiderando a complexidade das funções chamadas.

9. **conjunto_copiar**

- **Descrição**: Cria uma cópia de um conjunto.
- **Operações principais**:
  - Cria um novo conjunto.
  - Chama a função de cópia correspondente à estrutura utilizada.
- **Complexidade Total**: O(1), desconsiderando a complexidade das funções chamadas.

Resumo Geral:
Todas as funções do arquivo conjunto.c apresentam complexidade constante O(1) com relação à lógica interna do código. A complexidade completa dependerá diretamente das funções chamadas (lista_inserir, avl_remover, etc.), cujas complexidades serão analisadas separadamente.

TAD Lista Sequencial

A análise a seguir considera apenas a lógica implementada no código fornecido, desconsiderando as complexidades de funções externas chamadas.

1. **lista_criar**

   - **Descrição**: Aloca memória para uma estrutura de lista e inicializa o tamanho como 0.
   - **Operações principais**:
     - Alocação de memória: O(1).
     - Inicialização do tamanho: O(1).
   - **Complexidade Total**: O(1).

2. **lista_apagar**

   - **Descrição**: Libera a memória da lista e define o ponteiro como `NULL`.
   - **Operações principais**:
     - Liberação de memória: O(1).
     - Atribuição de NULL: O(1).
   - **Complexidade Total**: O(1).

3. **lista_inserir**

   - **Descrição**: Insere um elemento na lista em ordem crescente, deslocando elementos para abrir espaço.
   - **Operações principais**:
     - Busca pela posição correta: O(n) no pior caso (percorre toda a lista).
     - Verificação de duplicatas (chamada a lista_busca): O(1).
     - Deslocamento de elementos para abrir espaço: O(n) no pior caso (inserção no início).
   - **Complexidade Total**: O(n).

4. **lista_remover**

    - **Descrição**: Remove um elemento da lista, deslocando elementos subsequentes para preencher o espaço.
    - **Operações principais**:
        - Busca binária para encontrar a posição do elemento: O(\log n).
        - Deslocamento de elementos para preencher o espaço: O(n) no pior caso (remoção do primeiro elemento).
    - **Complexidade Total**: O(n).

5. **lista_imprimir**

    - **Descrição**: Imprime todos os elementos válidos da lista em ordem.
    - **Operações principais**:
        - Iteração pelos elementos da lista: O(n).
    - **Complexidade Total**: O(n).

6. **lista_busca**

    - **Descrição**: Busca um elemento na lista utilizando a função buscaBinariaLista.
    - **Operações principais**:
        - Busca binária: O(\log n).
    - **Complexidade Total**: O(\log n).

7. **lista_copiar**

    - **Descrição**: Cria uma cópia da lista original, inserindo cada elemento na nova lista.
    - **Operações principais**:
        - Criação de uma nova lista: O(1).
        - Iteração para copiar elementos: O(n).
        - Inserção de cada elemento (considerada O(1) neste contexto): O(n).
    - **Complexidade Total**: O(n).

8. **lista_consultar**

    - **Descrição**: Retorna o elemento armazenado em um índice específico da lista.
    - **Operações principais**:
        - Verificação de limites: O(1).
        - Acesso ao elemento pelo índice: O(1).
    - **Complexidade Total**: O(1).

9. **buscaBinariaLista**

- **Descrição**: Realiza busca binária recursiva para encontrar um elemento no vetor.
- **Operações principais**:
    - Divisão recursiva do vetor em cada chamada: O(\log n) no pior caso.
- **Complexidade Total**: O(\log n).

Resumo Geral:

- Operações de criação, remoção de memória e consulta direta possuem complexidade O(1).
- Operações de inserção e remoção, devido ao deslocamento de elementos, possuem complexidade O(n).
- Operações de busca utilizam busca binária, com complexidade O(\log n).
- A operação de copiar a lista tem complexidade O(n), pois envolve iteração e inserção de elementos.

TAD Árvore Binária de Busca AVL

A análise a seguir considera apenas a lógica implementada no arquivo `AVL.c`.

1. **avl_criar**

    - **Descrição**: Aloca memória para uma estrutura AVL, inicializa a raiz como NULL e o tamanho como 0.
    - **Operações principais**:
        - Alocação de memória: O(1).
        - Inicialização de ponteiros e variáveis: O(1).
    - **Complexidade Total**: O(1).

2. **avl_apagar**

    - **Descrição**: Apaga todos os nós da árvore recursivamente utilizando a função auxiliar no_apagar_recursivo.
    - **Operações principais**:
        - Chamada recursiva para percorrer todos os nós em pós-ordem: O(n), onde n é o número de nós.
        - Liberação de memória de cada nó: O(1) por nó.
    - **Complexidade Total**: O(n).

3. **avl_inserir**

    - **Descrição**: Insere um novo nó na árvore AVL e mantém o balanceamento.
    - **Operações principais**:
        - Criação de um novo nó: O(1).
        - Chamada à função auxiliar avl_inserir_no para percorrer a árvore e realizar a inserção e rotações: O(h), onde h é a altura da árvore.

- **Complexidade Total**: O(h) como se trata de uma árvore balanceada: O(\log n).

4. **avl_remover**

    - **Descrição**: Remove um nó da árvore AVL e mantém o balanceamento.
    - **Operações principais**:
        - Busca do nó a ser removido: O(h), onde h é a altura da árvore.
        - Chamada à função auxiliar avl_remover_no para tratar os casos de remoção e realizar rotações: O(h).
    - **Complexidade Total**: O(h) como se trata de uma árvore balanceada: O(\log n).

5. **avl_get_altura**

    - **Descrição**: Calcula a altura da árvore utilizando a função auxiliar no_get_altura.
    - **Operações principais**:
        - Cálculo recursivo da altura para cada nó: O(h).
    - **Complexidade Total**: O(h) como se trata de uma árvore balanceada: O(\log n).

6. **avl_get_tamanho**

    - **Descrição**: Retorna o tamanho da árvore armazenado na estrutura AVL.
    - **Operações principais**:
        - Acesso direto a uma variável: O(1).
    - **Complexidade Total**: O(1).

7. **avl_copiar**

    - **Descrição**: Cria uma cópia da árvore, duplicando todos os nós com a função auxiliar no_copiar_recursivo.
    - **Operações principais**:
        - Chamada recursiva para percorrer e copiar cada nó: O(n), onde n é o número de nós.
    - **Complexidade Total**: O(n).

8. **avl_imprimir**

    - **Descrição**: Imprime os elementos da árvore em ordem crescente (in-ordem) utilizando a função auxiliar avl_imprimir_arv.
    - **Operações principais**:
        - Percurso in-ordem para visitar todos os nós: O(n), onde n é o número de nós.
    - **Complexidade Total**: O(n).

9. **avl_busca**

    - **Descrição**: Busca uma chave na árvore AVL usando a função auxiliar busca_binaria_avl.
    - **Operações principais**:
        - Busca binária recursiva até encontrar o nó ou determinar sua ausência: O(h), onde h é a altura da árvore.
    - **Complexidade Total**: O(h) como se trata de uma árvore balanceada: O(\log n).

10. **avl_get_chave_raiz**

    - **Descrição**: Retorna a chave armazenada na raiz da árvore ou erro se estiver vazia.
    - **Operações principais**:
        - Acesso direto ao ponteiro da raiz: O(1).
    - **Complexidade Total**: O(1).

Resumo Geral:

- Operações básicas como criação, acesso ao tamanho e chave da raiz possuem complexidade O(1).
- Operações dependentes da altura (como inserção, remoção e busca) possuem complexidade O(h) como se trata de uma árvore balanceada: O(\log n).
- Operações que percorrem toda a árvore (como apagar, copiar e imprimir) possuem complexidade O(n).