Exercício-Programa: Árvore Binária de Busca (BST) com Operações Avançadas

Universidade Estadual de Londrina (UEL)
Professor Anderson Ávila

30 de janeiro de 2025

1 Introdução

Neste exercício-programa (EP), você deverá gerenciar uma **Árvore Binária de Busca (BST)** que armazena pares (chave, contador), permitindo a existência de múltiplas cópias (frequência) de uma mesma chave. Além das operações básicas de **inserção**, **busca** e **remoção**, serão implementadas funções mais avançadas, como:

- Remover apenas **uma** ocorrência de uma chave (decrementando o contador).
- Remover **todas** as ocorrências de uma chave (caso o contador seja maior que 1).
- Contar quantos nós (ou chaves distintas) existem na árvore.
- Encontrar o **k-ésimo menor** (ou maior) elemento considerando frequência.
- Imprimir todas as chaves dentro de um **intervalo** $[min, max]$.
- Encontrar o **Lowest Common Ancestor (LCA)** de duas chaves (opcional).

A BST seguirá a regra tradicional:

- Para qualquer nó N, todos os valores (chaves) na subárvore esquerda são menores que N->chave.
- Todos os valores na subárvore direita são maiores que N->chave.

Quando inserirmos uma chave já existente, em vez de criar um nó duplicado, **incrementaremos o contador** daquela chave.

2 Estrutura de Dados

Usaremos a seguinte estrutura para cada nó da BST:

```
typedef struct no {
    int chave;
    int contador;          // n mero de c pias (frequ ncia)
       dessa chave
    struct no* esq;        // ponteiro para a sub rvore
       esquerda
    struct no* dir;        // ponteiro para a sub rvore direita
} NO, *PONT;
```

- `chave`: valor inteiro que identifica o nó.
- `contador`: indica quantas vezes essa chave aparece na árvore.
- `esq` e `dir`: apontam para as subárvores esquerda e direita, respectivamente.

A **raiz** é do tipo `PONT`, inicialmente `NULL` quando a árvore está vazia.

3 Funções a Implementar

Abaixo, relacionamos as **funções obrigatórias** que você deve desenvolver ou completar. Algumas podem estar parcialmente fornecidas em um arquivo-base (`completeERenomeie.c`), e você precisará **complementar** ou **corrigir**.

3.1 Funções Básicas

1. `void inicializar(PONT* raiz)`
   - Recebe um ponteiro para a raiz da BST.
   - Define `*raiz = NULL`, deixando a árvore vazia.
2. `PONT criarNo(int valor)`
   - Aloca memória para um novo nó, preenche:
     - `chave` com `valor`,
     - `contador` com 1,
     - `esq` e `dir` com `NULL`.
   - Retorna o ponteiro para o novo nó.
3. `PONT buscar(PONT raiz, int valor)`
   - Retorna o *ponteiro* para o nó que contenha `valor` ou `NULL` caso não exista.
   - Lógica (pode ser recursiva ou iterativa):

- Se `raiz == NULL`, retorne `NULL`.
- Se `valor == raiz->chave`, retorne `raiz`.
- Se `valor < raiz->chave`, busca na subárvore esquerda.
- Se `valor > raiz->chave`, busca na subárvore direita.

3.2 Inserção e Remoção (com contador)

4. `PONT inserir(PONT raiz, int valor)`

   - Se a árvore estiver vazia (`raiz == NULL`), cria um novo nó e retorna.
   - Caso contrário, segue a regra da BST:
     - Se `valor < raiz->chave`, insira na subárvore esquerda.
     - Se `valor > raiz->chave`, insira na subárvore direita.
     - Se `valor == raiz->chave`, **incrementa** `raiz->contador` e não cria novo nó.
   - Retorna a nova raiz (importante para a recursão).

5. `PONT removerUmaOcorrencia(PONT raiz, int valor)`

   - Remove apenas **1 cópia** do valor:
     - Busca o nó que contenha `valor`.
     - Se não encontrado, não faz nada (retorna a raiz).
     - Se encontrado e `contador > 1`, apenas decrementa `contador` em 1.
     - Se encontrado e `contador == 1`, faz a remoção tradicional do nó na BST (tratando 0, 1 ou 2 filhos).
   - Retorna a (possível nova) raiz resultante.

6. `PONT removerTodasOcorrencias(PONT raiz, int valor)`

   - Remove **todas as cópias** de `valor`.
   - Busca o nó correspondente e, se `contador > 0`, remove-o por completo.
   - Se não encontrado, não faz nada.
   - Retorna a nova raiz.
   - A remoção do nó segue a lógica clássica de BST:
     - Nó sem filhos: libera e retorna `NULL`.
     - Nó com 1 filho: retorna o filho no lugar do nó removido.
     - Nó com 2 filhos: substitui a `chave` (e `contador`) pelo sucessor (ou antecessor), removendo recursivamente esse sucessor (ou antecessor).

3.3 Percursos e Impressões

7. `void exibirInOrder(PONT raiz)`

   - Imprime as chaves em ordem crescente.
   - Lógica recursiva:
     - Chama `exibirInOrder(raiz->esq)`.
     - Imprime `raiz->chave` (e o `contador`, se desejado).
     - Chama `exibirInOrder(raiz->dir)`.

8. `void exibirPreOrder(PONT raiz)` (opcional)
   `void exibirPostOrder(PONT raiz)` (opcional)

   - Podem ser úteis para depuração e verificação.
   - **Pré-ordem**: Raiz, Esquerda, Direita.
   - **Pós-ordem**: Esquerda, Direita, Raiz.

3.4 Funções de Contagem e Estatística

9. `int contarNos(PONT raiz)`

   - Retorna o **número de nós distintos** na árvore (não soma `contador`).
   - Se `raiz == NULL`, retorna 0.
   - Caso contrário, `1 + contarNos(raiz->esq) + contarNos(raiz->dir)`.

10. `int contarTotalElementos(PONT raiz)`

    - Retorna a soma dos `contador` de todos os nós. Ex.: se há nós $\{10(c = 3), 15(c = 2), 20(c = 5)\}$, total = 3+2+5 = 10.
    - Se `raiz == NULL`, retorna 0.
    - Caso contrário, `raiz->contador + contarTotalElementos(raiz->esq) + contarTotalElementos(raiz->dir)`.

3.5 Funções Avançadas

11. `int kEsimoMenor(PONT raiz, int k)`

    - Retorna a chave do **k-ésimo menor elemento** considerando as frequências ou `-1` (ou algo similar) se não existir.
    - Exemplo: BST com $\{5(c = 2), 7(c = 1), 9(c = 4)\}$ resulta na sequência ordenada $[5, 5, 7, 9, 9, 9, 9]$.
      - O 1º menor é 5, o 2º é 5, o 3º é 7, etc.
    - Use funções auxiliares para contar quantos elementos existem na subárvore esquerda para decidir se o k-ésimo está à esquerda, no próprio nó ou na direita.

12. `void imprimirIntervalo(PONT raiz, int min, int max)`

    - Imprime todas as chaves no intervalo $[min, max]$, considerando cada cópia (contador).
    - Se `raiz == NULL`, não faz nada.
    - Se `raiz->chave` $< min$, só busca à direita.
    - Se `raiz->chave` $> max$, só busca à esquerda.
    - Caso contrário, imprime o `raiz->chave` (`contador` vezes) e chama recursivamente esquerda e direita.

13. `PONT lowestCommonAncestor(PONT raiz, int val1, int val2)`

    - Retorna o nó que é o **ancestral comum** mais próximo das chaves `val1` e `val2`.
    - Usando a propriedade da BST:
        - Se `val1` $< raiz- > chave$ e `val2` $< raiz- > chave$, o LCA está na subárvore esquerda.
        - Se `val1` $> raiz- > chave$ e `val2` $> raiz- > chave$, o LCA está na subárvore direita.
        - Caso contrário, `raiz` é o LCA (presumindo que val1 e val2 existam).
    - Se não houver ambas as chaves na árvore, a função pode retornar `NULL` ou algo similar.

4 Arquivo Base e Regras

4.1 Arquivo Fornecido

Você receberá um arquivo (por exemplo, `completeERenomeie.c`) contendo estruturas, protótipos e possivelmente um `main()` de teste. Exemplo:

```c
#include <stdio.h>
#include <stdlib.h>

// Estrutura
typedef struct no {
    int chave;
    int contador;
    struct no* esq;
    struct no* dir;
} NO, *PONT;

// Prototipos das funcoes
void inicializar(PONT* raiz);
PONT criarNo(int valor);
PONT inserir(PONT raiz, int valor);
```

```c
PONT removerUmaOcorrencia(PONT raiz, int valor);
PONT removerTodasOcorrencias(PONT raiz, int valor);
PONT buscar(PONT raiz, int valor);
void exibirInOrder(PONT raiz);
int contarNos(PONT raiz);
int contarTotalElementos(PONT raiz);
int kEsimoMenor(PONT raiz, int k);
void imprimirIntervalo(PONT raiz, int min, int max);
// (opcional) PONT lowestCommonAncestor(PONT raiz, int val1,
    int val2);

int main(){
    PONT raiz;
    inicializar(&raiz);

    // Exemplos e testes...
    return 0;
}
```

Você deve completar as funções destacadas, **sem alterar** suas assinaturas.

4.2 Considerações Importantes

- **Valores repetidos**: quando a função `inserir` encontra uma chave já existente, ela deve simplesmente **incrementar** o campo `contador`.
- **Remover e buscar**: se o valor não existir, não faz nada (no caso da remoção) ou retorna `NULL` (no caso de busca).
- `kEsimoMenor` deve considerar `contador`.
- `imprimirIntervalo` imprime cada valor `contador` vezes.
- `lowestCommonAncestor` (opcional) retorna o ponteiro para o nó que for o LCA. Se alguma das chaves não existir, pode retornar `NULL` ou um código especial.

5 Entrega e Avaliação

- Você deve enviar um arquivo `.c` com seu código, renomeado conforme as instruções do professor.
- **Não** altere as assinaturas das funções fornecidas.
- As funções podem ser testadas individualmente.
- Critérios de avaliação:
  1. **Corretude** das operações de BST (regra de busca, inserção e remoção).
  2. **Lógica** das funções adicionais (`kEsimoMenor`, `imprimirIntervalo`, etc.).

3. **Documentação** (comentários claros nas principais funções).
4. **Qualidade** do código (indentação, clareza, ausência de vazamentos de memória etc.).

5.1 Observações Finais

- O trabalho é **individual**.
- Plágio não será tolerado.
- Exercícios com erro de compilação receberão nota zero.
- Mantenha o código **bem comentado**.

6 Exemplos de Teste

Exemplo de fluxo básico

1. Inserir: [10, 5, 15, 10, 5, 5, 18][1]
2. Agora, a árvore conteria:
   - Nó com chave 10, `contador=2`
   - Nó com chave 5, `contador=3`
   - Nó com chave 15, `contador=1`
   - Nó com chave 18, `contador=1`
3. `exibirInOrder` deve imprimir: 5, 5, 5, 10, 10, 15, 18 (cada valor repetido `contador` vezes).
4. `buscar(raiz, 5)` retorna ponteiro com `contador=3`.
5. `removerUmaOcorrencia(raiz, 5)` faz `contador=2` do nó com chave=5.
6. `removerTodasOcorrencias(raiz, 10)` remove por completo o nó 10 (contadores incluídos).
7. `contarNos(raiz)` retorna 2 (pois restam nós 5, 15 e 18; no caso, 3 nós).
8. `contarTotalElementos(raiz)` retorna a soma dos contadores.
9. `kEsimoMenor(raiz, 3)` se a ordem fosse $[5, 5, 15, 18]$, o 3^{o} seria 15.
10. `imprimirIntervalo(raiz, 6, 18)` imprime as chaves ≥ 6 e ≤ 18. Ex: 15, 18.
11. `lowestCommonAncestor(raiz, 5, 18)` (se houver) retorna o nó que seja ancestral comum mais próximo.

[1] onde chaves repetidas incrementam `contador`.