

# Exercícios Avançados - Árvores Binárias e Iteradores MySet

## Lista de 40 Questões Detalhadas

---

### Estrutura da Avaliação

**Seção 1: Implementações de funções na própria classe MySet (10 questões)**

**Seção 2: Implementação fora da classe MySet usando iteradores (10 questões)**

**Seção 3: Análise de código com pegadinhas (10 questões)**

**Seção 4: Operadores de iteradores e análise de complexidade (10 questões)**

---

### **SEÇÃO 1: Implementações de funções na própria classe MySet**

#### S1Q1

Implemente o método `size()` na classe MySet que retorna o número de elementos na árvore binária de busca. Considere que cada nó possui ponteiros `left`, `right` e um valor `data`. O método deve ter complexidade  $O(n)$ .

cpp

```
class MySet {  
private:  
    struct Node {  
        int data;  
        Node* left;  
        Node* right;  
        Node(int val) : data(val), left(nullptr), right(nullptr) {}  
    };  
    Node* root;  
  
public:  
    int size() const {  
        // SEU CÓDIGO AQUI  
    }  
};
```

#### S1Q2

Implemente o método `height()` que retorna a altura da árvore. A altura de uma árvore vazia é -1, e a altura de uma árvore com apenas a raiz é 0.

cpp

```
int height() const {
    // SEU CÓDIGO AQUI - implemente recursivamente
}
```

## S1Q3

Implemente o método `findMin()` que retorna um ponteiro para o nó com o menor valor na árvore. Se a árvore estiver vazia, retorne `nullptr`.

cpp

```
Node* findMin() const {
    // SEU CÓDIGO AQUI
}
```

## S1Q4

Implemente o método `findMax()` que retorna um ponteiro para o nó com o maior valor na árvore. Implemente de forma iterativa (não recursiva).

cpp

```
Node* findMax() const {
    // SEU CÓDIGO AQUI - versão iterativa
}
```

## S1Q5

Implemente o método `contains(int value)` que verifica se um valor existe na árvore. Retorne `true` se existir, `false` caso contrário.

cpp

```
bool contains(int value) const {
    // SEU CÓDIGO AQUI
}
```

## S1Q6

Implemente o método `insert(int value)` que insere um novo valor na árvore mantendo a propriedade de BST. Se o valor já existir, não faça nada.

cpp

```
void insert(int value) {
    // SEU CÓDIGO AQUI - implemente recursivamente
}
```

## S1Q7

Implemente o método `remove(int value)` que remove um valor da árvore. Considere todos os casos: nó folha, nó com um filho, nó with dois filhos.

cpp

```
void remove(int value) {
    // SEU CÓDIGO AQUI - método completo de remoção
}
```

## S1Q8

Implemente o método `clear()` que remove todos os elementos da árvore liberando a memória adequadamente.

cpp

```
void clear() {
    // SEU CÓDIGO AQUI
}
```

## S1Q9

Implemente o método `isBalanced()` que verifica se a árvore está balanceada (diferença de altura entre subárvores  $\leq 1$  para todos os nós).

cpp

```
bool isBalanced() const {
    // SEU CÓDIGO AQUI
}
```

## S1Q10

Implemente o método `countLeaves()` que conta o número de nós folha na árvore.

cpp

```
int countLeaves() const {
    // SEU CÓDIGO AQUI
}
```

---

## SEÇÃO 2: Implementação fora da classe MySet usando iteradores

### S2Q1

Implemente uma função `printInOrder(MySet& set)` que imprime todos os elementos em ordem crescente usando iteradores.

cpp

```
void printInOrder(MySet& set) {
    // SEU CÓDIGO AQUI - use begin() e end()
}
```

### S2Q2

Implemente uma função `findElement(MySet& set, int value)` que retorna um iterador para o elemento procurado, ou `set.end()` se não encontrar.

cpp

```
MySet::iterator findElement(MySet& set, int value) {
    // SEU CÓDIGO AQUI
}
```

### S2Q3

Implemente uma função `countRange(MySet& set, int min, int max)` que conta quantos elementos estão no intervalo  $[min, max]$  usando iteradores.

cpp

```
int countRange(MySet& set, int min, int max) {
    // SEU CÓDIGO AQUI
}
```

### S2Q4

Implemente uma função `copyEven(MySet& source, MySet& destination)` que copia apenas os números pares de `source` para `destination`.

cpp

```
void copyEven(MySet& source, MySet& destination) {  
    // SEU CÓDIGO AQUI  
}
```

## S2Q5

Implemente uma função `mergeSets(MySet& set1, MySet& set2, MySet& result)` que merge dois conjuntos ordenados em um terceiro conjunto.

cpp

```
void mergeSets(MySet& set1, MySet& set2, MySet& result) {  
    // SEU CÓDIGO AQUI - use iteradores de ambos os sets  
}
```

## S2Q6

Implemente uma função `intersection(MySet& set1, MySet& set2, MySet& result)` que encontra a interseção de dois conjuntos.

cpp

```
void intersection(MySet& set1, MySet& set2, MySet& result) {  
    // SEU CÓDIGO AQUI  
}
```

## S2Q7

Implemente uma função `difference(MySet& set1, MySet& set2, MySet& result)` que encontra elementos que estão em set1 mas não em set2.

cpp

```
void difference(MySet& set1, MySet& set2, MySet& result) {  
    // SEU CÓDIGO AQUI  
}
```

## S2Q8

Implemente uma função `isSubset(MySet& subset, MySet& superset)` que verifica se subset é subconjunto de superset.

cpp

```
bool isSubset(MySet& subset, MySet& superset) {
    // SEU CÓDIGO AQUI
}
```

## S2Q9

Implemente uma função `reverseOrder(MySet& set)` que imprime os elementos em ordem decrescente usando iterador reverso.

cpp

```
void reverseOrder(MySet& set) {
    // SEU CÓDIGO AQUI - use rbegin() e rend()
}
```

## S2Q10

Implemente uma função `sumRange(MySet& set, int start, int end)` que soma todos os elementos entre start e end (inclusive) usando iteradores.

cpp

```
int sumRange(MySet& set, int start, int end) {
    // SEU CÓDIGO AQUI
}
```

---

## SEÇÃO 3: Análise de código com pegadinhas

### S3Q1

Analise o código abaixo e determine a saída:

cpp

```
MySet set;
set.insert(5);
set.insert(3);
set.insert(7);
set.insert(1);

auto it = set.begin();
cout << *it << " ";
++it;
cout << *it << " ";
it++;
cout << *it << " ";
++it;
cout << *it << endl;
```

**Qual será a saída?**

### S3Q2

Determine o que acontece com este código:

cpp

```
MySet set;
set.insert(10);
set.insert(5);
set.insert(15);

auto it1 = set.find(5);
auto it2 = set.find(15);

set.erase(it1);
cout << *it2 << endl; // Linha problemática?
```

**O código tem algum problema? Se sim, qual?**

### S3Q3

Analise e determine a saída:

cpp

```
MySet set1, set2;
set1.insert(1); set1.insert(3); set1.insert(5);
set2.insert(2); set2.insert(3); set2.insert(4);

auto it1 = set1.begin();
auto it2 = set2.begin();

while(it1 != set1.end() && it2 != set2.end()) {
    if(*it1 < *it2) {
        cout << *it1 << " ";
        ++it1;
    } else if(*it2 < *it1) {
        cout << *it2 << " ";
        ++it2;
    } else {
        cout << *it1 << " ";
        ++it1; ++it2;
    }
}
```

**Qual será a saída?**

### S3Q4

Analise este código com swap:

cpp

```
MySet set1, set2;
set1.insert(1); set1.insert(2);
set2.insert(3); set2.insert(4);

auto it1 = set1.begin();
auto it2 = set2.begin();

set1.swap(set2);

cout << *it1 << " " << *it2 << endl;
```

**Qual será a saída? Os iteradores ainda são válidos?**

### S3Q5

Determine a saída deste código:

cpp

```
MySet set;
set.insert(8); set.insert(4); set.insert(12);
set.insert(2); set.insert(6); set.insert(10);

auto it = set.begin();
advance(it, 3);
cout << *it << " ";

it = set.end();
--it;
cout << *it << " ";

it = set.begin();
cout << *it << endl;
```

**Qual será a saída completa?**

### S3Q6

Analise este código com iteradores inválidos:

cpp

```
MySet set;
set.insert(5); set.insert(3); set.insert(7);

auto it = set.find(5);
set.clear();

if(it != set.end()) { // Linha problemática?
    cout << *it << endl;
}
```

**Este código tem comportamento undefined? Por quê?**

### S3Q7

Determine a saída:

cpp

```
MySet set;
set.insert(20); set.insert(10); set.insert(30);
set.insert(5); set.insert(15); set.insert(25);

auto it1 = set.lower_bound(12);
auto it2 = set.upper_bound(18);

while(it1 != it2) {
    cout << *it1 << " ";
    ++it1;
}
cout << endl;
```

**Qual será a saída?**

### S3Q8

Analise este código com comparação de iteradores:

cpp

```
MySet set1, set2;
set1.insert(1); set1.insert(2);
set2.insert(1); set2.insert(2);

auto it1 = set1.begin();
auto it2 = set2.begin();

if(it1 == it2) {
    cout << "Iguais" << endl;
} else {
    cout << "Diferentes" << endl;
}
```

**Qual será a saída? Por quê?**

### S3Q9

Determine o resultado:

cpp

```
MySet set;
set.insert(6); set.insert(4); set.insert(8);
set.insert(2); set.insert(5); set.insert(7);

auto it = set.begin();
++it; ++it; ++it; // it aponta para qual elemento?

set.erase(5); // Remove o elemento 5

cout << *it << endl; // Ainda válido?
```

**Qual será a saída? O iterador ainda é válido?**

### S3Q10

Analise este código complexo:

cpp

```
MySet set;
for(int i = 0; i < 10; i += 2) {
    set.insert(i);
}

auto it = set.begin();
while(it != set.end()) {
    if(*it % 4 == 0) {
        it = set.erase(it);
    } else {
        ++it;
    }
}

for(auto& elem : set) {
    cout << elem << " ";
}
cout << endl;
```

**Qual será a saída final?**

---

## SEÇÃO 4: Operadores de iteradores e análise de complexidade

### S4Q1

Implemente o operador `--` (pré-decremento) para o iterador da classe MySet. Considere que o iterador implementa percurso in-order.

cpp

```
class MySetIterator {  
private:  
    Node* current;  
    stack<Node*> nodeStack;  
  
public:  
    MySetIterator& operator--() {  
        // SEU CÓDIGO AQUI  
        return *this;  
    }  
};
```

**Qual é a complexidade deste operador no pior caso?**

#### S4Q2

Implemente o operador `+` que avança o iterador n posições:

cpp

```
MySetIterator operator+(int n) const {  
    // SEU CÓDIGO AQUI  
}
```

**Qual é a complexidade desta operação? É eficiente para árvores binárias?**

#### S4Q3

Analise a complexidade das seguintes operações em uma árvore binária de busca平衡ada com n elementos:

- a) `distance(set.begin(), set.end())` b) `advance(it, n)` onde `it = set.begin()` c)  
`set.lower_bound(value)` d) `set.upper_bound(value)`

**Justifique cada resposta.**

#### S4Q4

Implemente um iterador bidirecional personalizado que percorre a árvore em pré-ordem:

cpp

```
class PreOrderIterator {  
private:  
    stack<Node*> nodeStack;  
  
public:  
    PreOrderIterator& operator++() {  
        // SEU CÓDIGO AQUI para pré-ordem  
        return *this;  
    }  
  
    PreOrderIterator& operator--() {  
        // SEU CÓDIGO AQUI para voltar em pré-ordem  
        return *this;  
    }  
};
```

## S4Q5

Compare a complexidade de iterar sobre todos os elementos usando:

- a) Recursão in-order tradicional
- b) Iterador implementado com stack
- c) Morris Traversal (sem stack)

**Analise complexidade de tempo e espaço para cada abordagem.**

## S4Q6

Implemente o operador `[]` para MySet que retorna o n-ésimo elemento em ordem:

cpp

```
int& operator[](size_t index) {  
    // SEU CÓDIGO AQUI  
}
```

**Qual é a complexidade desta operação? Como pode ser otimizada?**

## S4Q7

Analise este código e determine sua complexidade:

cpp

```
void functionComplexity(MySet& set) {
    auto it1 = set.begin();
    auto it2 = set.end();

    while(it1 != it2) {
        auto temp = set.find(*it1);
        if(temp != set.end()) {
            ++it1;
        }
    }
}
```

**Qual é a complexidade total desta função?**

#### S4Q8

Implemente um comparador personalizado para iteradores que compare por valor absoluto:

cpp

```
struct AbsoluteComparator {
    bool operator()(const MySetIterator& a, const MySetIterator& b) const {
        // SEU CÓDIGO AQUI
    }
};
```

**Como isso afetaria a complexidade de operações de busca?**

#### S4Q9

Analise a complexidade desta função que merge duas árvores:

cpp

```
MySet mergeTrees(const MySet& set1, const MySet& set2) {
    MySet result;
    auto it1 = set1.begin();
    auto it2 = set2.begin();

    while(it1 != set1.end() && it2 != set2.end()) {
        if(*it1 <= *it2) {
            result.insert(*it1);
            ++it1;
        } else {
            result.insert(*it2);
            ++it2;
        }
    }

    while(it1 != set1.end()) {
        result.insert(*it1);
        ++it1;
    }

    while(it2 != set2.end()) {
        result.insert(*it2);
        ++it2;
    }

    return result;
}
```

**Qual é a complexidade desta função em termos de  $|set1|$  e  $|set2|$ ?**

#### S4Q10

Implemente uma função que verifica se uma árvore é um espelho de outra usando iteradores:

cpp

```
bool isMirror(const MySet& set1, const MySet& set2) {
    // SEU CÓDIGO AQUI
    // Use iteradores normal e reverso
}
```

**Qual é a complexidade desta verificação? É possível otimizar?**

---

#### GABARITO COMPLETO

## SEÇÃO 1 - GABARITO

### S1Q1 - Resposta:

```
cpp

int size() const {
    return sizeHelper(root);
}

private:
int sizeHelper(Node* node) const {
    if (node == nullptr) return 0;
    return 1 + sizeHelper(node->left) + sizeHelper(node->right);
}
```

**Complexidade:**  $O(n)$  - visita todos os nós uma vez.

### S1Q2 - Resposta:

```
cpp

int height() const {
    return heightHelper(root);
}

private:
int heightHelper(Node* node) const {
    if (node == nullptr) return -1;
    return 1 + max(heightHelper(node->left), heightHelper(node->right));
}
```

**Complexidade:**  $O(n)$  - visita todos os nós.

### S1Q3 - Resposta:

```
cpp

Node* findMin() const {
    if (root == nullptr) return nullptr;
    Node* current = root;
    while (current->left != nullptr) {
        current = current->left;
    }
    return current;
}
```

**Complexidade:** O(h) onde h é a altura da árvore.

#### S1Q4 - Resposta:

cpp

```
Node* findMax() const {
    if (root == nullptr) return nullptr;
    Node* current = root;
    while (current->right != nullptr) {
        current = current->right;
    }
    return current;
}
```

**Complexidade:** O(h) iterativa.

#### S1Q5 - Resposta:

cpp

```
bool contains(int value) const {
    return containsHelper(root, value);
}

private:
bool containsHelper(Node* node, int value) const {
    if (node == nullptr) return false;
    if (node->data == value) return true;
    if (value < node->data) return containsHelper(node->left, value);
    return containsHelper(node->right, value);
}
```

#### S1Q6 - Resposta:

cpp

```
void insert(int value) {
    root = insertHelper(root, value);
}

private:
Node* insertHelper(Node* node, int value) {
    if (node == nullptr) return new Node(value);
    if (value < node->data) {
        node->left = insertHelper(node->left, value);
    } else if (value > node->data) {
        node->right = insertHelper(node->right, value);
    }
    return node;
}
```

**S1Q7 - Resposta:**

cpp

```
void remove(int value) {
    root = removeHelper(root, value);
}

private:
Node* removeHelper(Node* node, int value) {
    if (node == nullptr) return nullptr;

    if (value < node->data) {
        node->left = removeHelper(node->left, value);
    } else if (value > node->data) {
        node->right = removeHelper(node->right, value);
    } else {
        // Nó encontrado
        if (node->left == nullptr) {
            Node* temp = node->right;
            delete node;
            return temp;
        } else if (node->right == nullptr) {
            Node* temp = node->left;
            delete node;
            return temp;
        }
    }

    // Nó com dois filhos
    Node* successor = findMin(node->right);
    node->data = successor->data;
    node->right = removeHelper(node->right, successor->data);
}
return node;
}
```

**S1Q8 - Resposta:**

cpp

```
void clear() {
    clearHelper(root);
    root = nullptr;
}

private:
void clearHelper(Node* node) {
    if (node != nullptr) {
        clearHelper(node->left);
        clearHelper(node->right);
        delete node;
    }
}
```

### S1Q9 - Resposta:

cpp

```
bool isBalanced() const {
    return isBalancedHelper(root) != -1;
}

private:
int isBalancedHelper(Node* node) const {
    if (node == nullptr) return 0;

    int leftHeight = isBalancedHelper(node->left);
    if (leftHeight == -1) return -1;

    int rightHeight = isBalancedHelper(node->right);
    if (rightHeight == -1) return -1;

    if (abs(leftHeight - rightHeight) > 1) return -1;

    return 1 + max(leftHeight, rightHeight);
}
```

### S1Q10 - Resposta:

cpp

```
int countLeaves() const {
    return countLeavesHelper(root);
}

private:
int countLeavesHelper(Node* node) const {
    if (node == nullptr) return 0;
    if (node->left == nullptr && node->right == nullptr) return 1;
    return countLeavesHelper(node->left) + countLeavesHelper(node->right);
}
```

## SEÇÃO 2 - GABARITO

### S2Q1 - Resposta:

cpp

```
void printInOrder(MySet& set) {
    for (auto it = set.begin(); it != set.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;
}
```

### S2Q2 - Resposta:

cpp

```
MySet::iterator findElement(MySet& set, int value) {
    for (auto it = set.begin(); it != set.end(); ++it) {
        if (*it == value) return it;
    }
    return set.end();
}
```

**Nota:** Melhor usar `set.find(value)` que é  $O(\log n)$ .

### S2Q3 - Resposta:

cpp

```
int countRange(MySet& set, int min, int max) {
    int count = 0;
    for (auto it = set.begin(); it != set.end(); ++it) {
        if (*it >= min && *it <= max) {
            count++;
        }
    }
    return count;
}
```

#### S2Q4 - Resposta:

cpp

```
void copyEven(MySet& source, MySet& destination) {
    for (auto it = source.begin(); it != source.end(); ++it) {
        if (*it % 2 == 0) {
            destination.insert(*it);
        }
    }
}
```

#### S2Q5 - Resposta:

cpp

```
void mergeSets(MySet& set1, MySet& set2, MySet& result) {  
    auto it1 = set1.begin();  
    auto it2 = set2.begin();  
  
    while (it1 != set1.end() && it2 != set2.end()) {  
        if (*it1 <= *it2) {  
            result.insert(*it1);  
            ++it1;  
        } else {  
            result.insert(*it2);  
            ++it2;  
        }  
    }  
  
    while (it1 != set1.end()) {  
        result.insert(*it1);  
        ++it1;  
    }  
  
    while (it2 != set2.end()) {  
        result.insert(*it2);  
        ++it2;  
    }  
}
```

## S2Q6 - Resposta:

cpp

```
void intersection(MySet& set1, MySet& set2, MySet& result) {  
    for (auto it = set1.begin(); it != set1.end(); ++it) {  
        if (set2.find(*it) != set2.end()) {  
            result.insert(*it);  
        }  
    }  
}
```

## S2Q7 - Resposta:

cpp

```
void difference(MySet& set1, MySet& set2, MySet& result) {
    for (auto it = set1.begin(); it != set1.end(); ++it) {
        if (set2.find(*it) == set2.end()) {
            result.insert(*it);
        }
    }
}
```

## S2Q8 - Resposta:

cpp

```
bool isSubset(MySet& subset, MySet& superset) {
    for (auto it = subset.begin(); it != subset.end(); ++it) {
        if (superset.find(*it) == superset.end()) {
            return false;
        }
    }
    return true;
}
```

## S2Q9 - Resposta:

cpp

```
void reverseOrder(MySet& set) {
    for (auto it = set.rbegin(); it != set.rend(); ++it) {
        cout << *it << " ";
    }
    cout << endl;
}
```

## S2Q10 - Resposta:

cpp

```
int sumRange(MySet& set, int start, int end) {
    int sum = 0;
    auto it = set.lower_bound(start);
    auto endIt = set.upper_bound(end);

    while (it != endIt) {
        sum += *it;
        ++it;
    }
    return sum;
}
```

## SEÇÃO 3 - GABARITO

### S3Q1 - Resposta:

**Saída:** `1 3 5 7` **Explicação:** Os elementos são inseridos mantendo a propriedade BST, o percurso in-order resulta em ordem crescente.

### S3Q2 - Resposta:

**Problema:** Sim, comportamento indefinido. Após `erase(it1)`, o iterador `it2` pode ser invalidado dependendo da implementação da remoção. **Correção:** Verificar se `it2` ainda é válido após a remoção.

### S3Q3 - Resposta:

**Saída:** `1 2 3 4 5` **Explicação:** Algoritmo de merge de duas sequências ordenadas, imprime elementos em ordem crescente de ambos os sets.

### S3Q4 - Resposta:

**Comportamento indefinido:** Os iteradores `it1` e `it2` tornam-se inválidos após `swap()`. **Resultado:** Comportamento indefinido ao desreferenciar iteradores inválidos.

### S3Q5 - Resposta:

**Saída:** `10 12 2` **Explicação:**

- `advance(it, 3)` move para o 4º elemento (10)
- `--it` do `end()` vai para o último elemento (12)
- `begin()` retorna o primeiro elemento (2)

### S3Q6 - Resposta:

**Sim, comportamento indefinido.** Após `clear()`, todos os iteradores tornam-se inválidos. Usar `it` após `clear()` é undefined behavior.

### S3Q7 - Resposta:

**Saída:** `15` **Explicação:**

- `lower_bound(12)` retorna iterador para 15 (primeiro  $\geq 12$ )
- `upper_bound(18)` retorna iterador para 20 (primeiro  $> 18$ )
- Range contém apenas 15

### S3Q8 - Resposta:

**Saída:** `Diferentes` **Explicação:** Iteradores de containers diferentes nunca são iguais, mesmo que apontem para valores idênticos.

### S3Q9 - Resposta:

**Comportamento depende da implementação.** Se o iterador apontava para 5, após `erase(5)` ele se torna inválido. Se apontava para outro elemento (6), pode ainda ser válido.

### S3Q10 - Resposta:

**Saída:** `2 6` **Explicação:**

- Insere: 0, 2, 4, 6, 8
- Remove elementos divisíveis por 4: 0, 4, 8
- Sobram: 2, 6

## SEÇÃO 4 - GABARITO

### S4Q1 - Resposta:

cpp

```
MySetIterator& operator--() {
    if (nodeStack.empty()) {
        // Caso especial: iterator em end()
        current = findMax(root);
        // Reconstrói stack para current
    } else {
        nodeStack.pop();
        if (!nodeStack.empty()) {
            current = nodeStack.top();
        } else {
            // Voltar para predecessor
            current = findPredecessor(current);
        }
    }
    return *this;
}
```

**Complexidade:**  $O(h)$  no pior caso, onde  $h$  é a altura da árvore.

#### S4Q2 - Resposta:

cpp

```
MySetIterator operator+(int n) const {
    MySetIterator temp = *this;
    for (int i = 0; i < n; ++i) {
        ++temp;
    }
    return temp;
}
```

**Complexidade:**  $O(n \times h)$  - não é eficiente para árvores binárias. Árvores não suportam acesso aleatório eficiente.

#### S4Q3 - Respostas:

- a) `distance(set.begin(), set.end())` -  **$O(n)$**  - precisa percorrer todos os elementos
- b) `advance(it, n)` -  **$O(n \times h)$**  -  $n$  incrementos, cada um custando  $O(h)$
- c) `set.lower_bound(value)` -  **$O(\log n)$**  - busca binária na árvore
- d) `set.upper_bound(value)` -  **$O(\log n)$**  - busca binária na árvore

#### S4Q4 - Resposta:

cpp

```
PreOrderIterator& operator++() {
    if (!nodeStack.empty()) {
        Node* current = nodeStack.top();
        nodeStack.pop();

        // Adiciona filhos (direito primeiro para m
```