

Iteradores em Árvores Binárias de Pesquisa

Um Guia Completo para Compreensão e Implementação

Sumário

1. [Introdução aos Iteradores](#)
 2. [Percursos em Árvores Binárias](#)
 3. [Por que In-Ordem na STL?](#)
 4. [Implementação de Iteradores](#)
 5. [Complexidade e Análise](#)
 6. [Exercícios Práticos](#)
 7. [Considerações Avançadas](#)
-

1. Introdução aos Iteradores

O que são Iteradores?

Um iterador é um objeto que permite percorrer elementos de uma coleção de dados de forma sequencial, abstraindo os detalhes internos da estrutura. Em C++, iteradores seguem o padrão de design Iterator, fornecendo uma interface uniforme para acessar elementos independentemente da estrutura de dados subjacente.

Por que usar Iteradores em ABPs?

Árvores Binárias de Pesquisa são estruturas hierárquicas, não lineares como arrays ou listas. Para torná-las compatíveis com algoritmos que esperam sequências lineares (como os algoritmos da STL), precisamos de uma forma de "linearizar" o acesso aos elementos. É aqui que entram os iteradores.

Características Fundamentais

Um iterador para ABP deve:

- Fornecer acesso sequencial aos elementos
 - Manter uma ordem consistente e útil
 - Ser eficiente em tempo e espaço
 - Seguir as convenções da linguagem (C++)
-

2. Percursos em Árvores Binárias

Tipos de Percurso

Existem várias formas de percorrer uma árvore binária:

2.1 Depth-First Search (DFS)

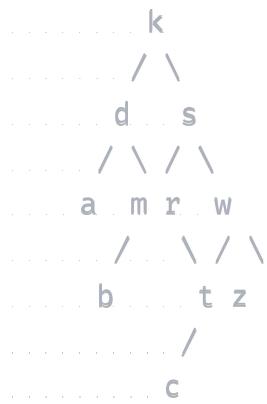
- **Pré-ordem:** Raiz → Esquerda → Direita
- **In-ordem:** Esquerda → Raiz → Direita
- **Pós-ordem:** Esquerda → Direita → Raiz

2.2 Breadth-First Search (BFS)

- **Por níveis:** Percorre nível por nível da árvore

Exemplo Prático

Considerando a árvore:



Pré-ordem: k, d, a, m, b, s, r, t, c, w, z **In-ordem:** a, d, b, m, k, r, c, t, s, w, z **Pós-ordem:** a, b, m, d, c, t, r, z, w, s, k **Por níveis:** k, d, s, a, m, r, w, b, t, z, c

Por que In-Ordem é Especial?

Para uma Árvore Binária de Pesquisa, o percurso in-ordem tem uma propriedade fundamental: **os elementos são visitados em ordem crescente**. Isso acontece porque:

1. Visitamos primeiro toda a subárvore esquerda (elementos menores)
2. Depois visitamos a raiz
3. Por fim, visitamos a subárvore direita (elementos maiores)

No exemplo acima, o percurso in-ordem produz: a, b, c, d, k, m, r, s, t, w, z - uma sequência ordenada!

3. Por que In-Ordem na STL?

Vantagens do Percurso In-Ordem

3.1 Ordem Natural

A principal vantagem é que os elementos são acessados em sua ordem natural (crescente). Isso torna a ABP compatível com algoritmos que esperam dados ordenados.

3.2 Consistência

Mesmo que duas ABPs diferentes contenham os mesmos elementos, o percurso in-ordem sempre produzirá a mesma sequência ordenada. Por exemplo:

Árvore 1:



Árvore 2:



Ambas produzem a sequência in-ordem: 1, 3, 5, 6, 7, 9

3.3 Compatibilidade com Algoritmos

Muitos algoritmos da STL (como `std::lower_bound`, `std::upper_bound`, `std::binary_search`) assumem que os dados estão ordenados. O percurso in-ordem garante essa propriedade.

Desvantagens de Outros Percursos

- **Pré-ordem/Pós-ordem:** Não mantêm ordem dos elementos
- **Por níveis:** Também não mantém ordem, além de ser mais complexo de implementar eficientemente

4. Implementação de Iteradores

4.1 Estrutura Básica do Nô

Para implementar iteradores eficientemente, precisamos armazenar referência ao nó pai:

cpp

```
template<typename T>
struct Node {
    ... T data;
    ... Node* left;
    Node* right;
    Node* parent; // Essencial para iteradores eficientes
    ...
    Node(const T& value) : data(value), left(nullptr),
                           right(nullptr), parent(nullptr) {}
};
```

4.2 Por que Precisamos do Ponteiro Parent?

Sem o ponteiro parent, encontrar o próximo elemento no percurso in-ordem seria muito custoso.

Teríamos que:

1. Armazenar o caminho percorrido em uma pilha
2. Ou fazer uma busca completa a partir da raiz

Com o ponteiro parent, podemos navegar eficientemente pela árvore.

4.3 Algoritmo para Encontrar o Sucessor

Dado um nó atual, como encontrar o próximo nó no percurso in-ordem?

cpp

```
Node* nextInOrder(Node* current) {
    if (current == nullptr) return nullptr;

    // Caso 1: Se há subárvore direita, o sucessor é o menor
    // elemento dessa subárvore
    if (current->right != nullptr) {
        Node* successor = current->right;
        while (successor->left != nullptr) {
            successor = successor->left;
        }
        return successor;
    }

    // Caso 2: Não há subárvore direita
    // Sobe na árvore até encontrar um ancestral onde
    // somos parte da subárvore esquerda
    Node* parent = current->parent;
    while (parent != nullptr && current == parent->right) {
        current = parent;
        parent = parent->parent;
    }

    return parent;
}
```

4.4 Implementação da Classe Iterator

cpp

```

template<typename T>
class BST {
private:
    Node<T>* root;

public:
    class Iterator {
private:
    Node<T>* current;

public:
    Iterator(Node<T>* node) : current(node) {}

        // Operador de desreferenciamento
    T& operator*() {
        return current->data;
    }

        // Operador de incremento (pré-fixo)
    Iterator& operator++() {
        current = nextInOrder(current);
        return *this;
    }

        // Operador de incremento (pós-fixo)
    Iterator operator++(int) {
        Iterator temp = *this;
        current = nextInOrder(current);
        return temp;
    }

        // Operador de comparação
    bool operator==(const Iterator& other) const {
        return current == other.current;
    }

    bool operator!=(const Iterator& other) const {
        return current != other.current;
    }
};

// Função begin() - retorna iterador para o menor elemento
Iterator begin() {
    if (root == nullptr) return Iterator(nullptr);

    Node<T>* leftmost = root;
}

```

```

        while (leftmost->left != nullptr) {
            leftmost = leftmost->left;
        }
        return Iterator(leftmost);
    }

    // Função end() - retorna iterador "one past the end"
    Iterator end() {
        return Iterator(nullptr);
    }
};

```

4.5 Mantendo o Ponteiro Parent Atualizado

Toda operação que modifica a árvore deve atualizar os ponteiros parent:

```

cpp

void insert(const T& value) {
    if (root == nullptr) {
        root = new Node<T>(value);
        return;
    }

    Node<T>* current = root;
    Node<T>* parent = nullptr;

    while (current != nullptr) {
        parent = current;
        if (value < current->data) {
            current = current->left;
        } else {
            current = current->right;
        }
    }

    Node<T>* newNode = new Node<T>(value);
    newNode->parent = parent; // Definir o pai

    if (value < parent->data) {
        parent->left = newNode;
    } else {
        parent->right = newNode;
    }
}

```

5. Complexidade e Análise

5.1 Complexidade do Incremento

Melhor caso: $O(1)$

- Quando o próximo elemento está na subárvore direita do nó atual

Pior caso: $O(h)$, onde h é a altura da árvore

- Quando precisamos subir até próximo da raiz

Caso amortizado: $O(1)$

- Embora um incremento individual possa ser $O(h)$, percorrer toda a árvore continua sendo $O(n)$

5.2 Por que Amortizado é $O(1)$?

Considere percorrer toda a árvore com n nós:

- Cada aresta é percorrida no máximo 2 vezes (uma subindo, uma descendo)
- Uma árvore com n nós tem $n-1$ arestas
- Total de operações: $O(n)$
- Operações por incremento: $O(n)/n = O(1)$

5.3 Utilidades dos Iteradores

1. **Algoritmos da STL:** `std::find`, `std::for_each`, etc.
 2. **Range-based loops:** `for (auto& elem : mySet)`
 3. **Busca por intervalos:** encontrar elementos em um range específico
 4. **Operações set:** união, interseção, diferença
-

6. Exercícios Práticos

6.1 Tipos de Iteradores

A classe MySet (implementação de set usando ABP) pode suportar:

- **Forward Iterator:** Permite incremento (`++`)
- **Bidirectional Iterator:** Permite incremento e decremento (`++, --`)
- **Random Access Iterator:** Não é prático para ABPs devido à estrutura hierárquica

6.2 Busca por Predecessor e Sucessor

cpp

```
// Encontra o maior elemento menor que X e o menor elemento maior que X
pair<Iterator, Iterator> findBounds(const T& X) {
    Iterator predecessor = end();
    Iterator successor = end();

    Node<T>* current = root;

    while (current != nullptr) {
        if (current->data < X) {
            predecessor = Iterator(current);
            current = current->right;
        } else if (current->data > X) {
            successor = Iterator(current);
            current = current->left;
        } else {
            // Elemento encontrado
            predecessor = Iterator(prevInOrder(current));
            successor = Iterator(nextInOrder(current));
            break;
        }
    }

    return make_pair(predecessor, successor);
}
```

Complexidade: $O(\log n)$ para árvore balanceada, $O(n)$ no pior caso.

6.3 Contagem Eficiente

Para contar elementos menores que X eficientemente, podemos:

1. **Modificar a estrutura do nó** para armazenar o tamanho da subárvore:

cpp

```
struct Node {
    T data;
    Node* left;
    Node* right;
    Node* parent;
    int subtree_size; // Tamanho da subárvore com raiz neste nó
};
```

2. **Implementar contagem em $O(\log n)$:**

cpp

```
int countLessThan(const T& X) {
    return countLessThanHelper(root, X);
}

int countLessThanHelper(Node<T>* node, const T& X) {
    if (node == nullptr) return 0;

    if (node->data >= X) {
        // Todos os elementos da subárvore direita são >= X
        return countLessThanHelper(node->left, X);
    } else {
        // Este nó e toda sua subárvore esquerda são < X
        int leftSize = (node->left) ? node->left->subtree_size : 0;
        return 1 + leftSize + countLessThanHelper(node->right, X);
    }
}
```

7. Considerações Avançadas

7.1 Iteradores Reversos

Para percorrer a árvore em ordem decrescente:

cpp

```
class ReverseIterator {
    // Similar ao Iterator, mas usa prevInOrder em vez de nextInOrder
    ReverseIterator& operator++() {
        current = prevInOrder(current);
        return *this;
    }

    // rbegin() retorna iterador para o maior elemento
    ReverseIterator rbegin() {
        if (root == nullptr) return ReverseIterator(nullptr);

        Node<T>* rightmost = root;
        while (rightmost->right != nullptr) {
            rightmost = rightmost->right;
        }
        return ReverseIterator(rightmost);
    }
```

7.2 Iteradores Const

cpp

```
class ConstIterator {
private:
    const Node<T>* current;
public:
    const T& operator*() const {
        return current->data;
    }
    ...
    // Demais operações similares, mas não permitem modificação
};
```

7.3 Invalidação de Iteradores

Cuidado: Operações que modificam a árvore podem invalidar iteradores:

- **Inserção:** Pode invalidar iteradores se houver rotações (em árvores auto-balanceadas)
- **Remoção:** Invalida iteradores que apontam para o nó removido
- **Reestruturação:** Operações como rebalanceamento invalidam todos os iteradores

7.4 Thread Safety

Iteradores em ABPs não são thread-safe por padrão. Para uso em ambiente multi-threaded:

1. Use locks externos
2. Implemente copy-on-write
3. Use estruturas de dados concorrentes especializadas

7.5 Otimizações Avançadas

1. **Cache de iteradores:** Manter cache do último iterador usado
2. **Lazy evaluation:** Calcular próximo elemento apenas quando necessário
3. **Memory pooling:** Usar pool de memória para nós da árvore

Conclusão

Os iteradores em Árvores Binárias de Pesquisa são uma ferramenta poderosa que:

1. **Abstraem a complexidade** da estrutura hierárquica
2. **Mantêm ordem natural** dos elementos (percurso in-ordem)
3. **Fornecem interface uniforme** compatível com algoritmos padrão

4. Oferecem eficiência com complexidade amortizada O(1) por incremento

A implementação requer cuidado especial com:

- Manutenção de ponteiros parent
- Algoritmos para encontrar sucessor/predecessor
- Gerenciamento de invalidação de iteradores

Dominar esses conceitos é fundamental para trabalhar efetivamente com estruturas de dados em C++ e aproveitar todo o poder da Standard Template Library.

Este material serve como base sólida para compreensão e implementação de iteradores em ABPs. Para aprofundamento, consulte a documentação da STL e implemente os exercícios propostos.