**Estrutura de dados em Golang**

João Pedro D’Agostin, Bruno Ramos Leonardi

1Estrutura de Dados – Centro Universitário UniBrasil - 82821-020 – Curitiba – PR – Brasil

**{dagostinjoaopedro@gmail.com, brunoramosleo@gmail.com}**

**Abstract.** This article provides a comparative analysis of binary tree implementations in two widely-used programming languages, Go (Golang) and C/C++. Binary trees are fundamental data structures in computer science with diverse applications in algorithms and systems. We explore how these languages address the creation, manipulation, and management of binary trees, highlighting their differences and similarities.

In Go, the simplicity of syntax and automatic memory management are strong points that make binary tree implementation accessible and safe. Additionally, Go offers concurrency features that can be valuable in parallel processing scenarios.

On the other hand, C/C++ provides greater control over memory allocation, which can be advantageous in high-performance situations, although it requires more detailed manual management.

This article aims to assist developers in making informed decisions when choosing the most suitable language for implementing binary trees in their projects, considering factors such as performance, ease of development, and memory management.

**Resumo.** Este artigo apresenta uma análise comparativa das implementações de árvores binárias em duas linguagens de programação amplamente utilizadas, Go (Golang) e C/C++. Árvores binárias são estruturas de dados fundamentais em ciência da computação e têm diversas aplicações em algoritmos e sistemas. Exploramos como essas linguagens abordam a criação, manipulação e gerenciamento de árvores binárias, destacando suas diferenças e semelhanças.

Em Go, a simplicidade da sintaxe e o gerenciamento automático de memória são pontos fortes que tornam a implementação de árvores binárias acessível e segura. Além disso, Go oferece recursos de concorrência que podem ser úteis em cenários de processamento paralelo.

Por outro lado, C/C++ proporcionam maior controle sobre a alocação de memória, o que pode ser vantajoso em situações de alta performance, embora exija um gerenciamento manual mais detalhado.

Este artigo visa auxiliar desenvolvedores a tomar decisões informadas ao escolher a linguagem mais adequada para implementar árvores binárias em seus projetos, considerando fatores como desempenho, facilidade de desenvolvimento e gerenciamento de memória.

**1. Introdução**

A escolha da linguagem de programação desempenha um papel fundamental no desenvolvimento de software. Ela afeta diretamente a produtividade do desenvolvedor, a eficiência do código e a confiabilidade do sistema resultante. Neste contexto, a linguagem Go (também conhecida como Golang) tem emergido como uma alternativa interessante, oferecendo um conjunto de recursos que a tornam competitiva em relação às linguagens tradicionais como C e C++.

Este artigo tem como objetivo realizar uma análise comparativa entre Go e C/C++ em várias dimensões, incluindo desempenho, facilidade de desenvolvimento, segurança e manutenção. Através desta análise, pretendemos fornecer insights valiosos para desenvolvedores e engenheiros de software que buscam tomar decisões informadas sobre a escolha da linguagem de programação mais adequada para seus projetos.

**2. Justificativa**

A escolha de C/C++ como ponto de comparação com Go é significativa, uma vez que C/C++ são amplamente reconhecidas por sua eficiência e controle de baixo nível, enquanto Go foi projetada com foco na simplicidade e produtividade do desenvolvedor. Essa dicotomia cria uma oportunidade valiosa para avaliar como essas linguagens atendem às necessidades de desenvolvimento de software em cenários variados.

Go foi desenvolvida pelo Google e lançada em 2009 com o objetivo de abordar algumas das deficiências percebidas em linguagens existentes, como a complexidade do C/C++ e os problemas de concorrência. Desde então, Go tem ganhado popularidade em várias áreas, incluindo desenvolvimento de sistemas, aplicativos web e serviços em nuvem.

**3. Fundamentação**

A Estrutura de Dados de Árvore Binária:

As árvores binárias são uma estrutura de dados fundamental em ciência da computação e programação. Essa estrutura hierárquica é composta por nós (ou elementos) que estão conectados de forma hierárquica. Cada nó pode ter até dois filhos: um à esquerda e outro à direita. A raiz da árvore é o nó principal a partir do qual todos os outros nós são acessados. Os nós sem filhos são chamados de folhas. A capacidade de organização hierárquica e eficiente torna as árvores binárias uma escolha comum para uma ampla gama de aplicações, incluindo bancos de dados, sistemas de arquivos, compiladores, algoritmos de pesquisa e muito mais.

Conceitos Fundamentais das Árvores Binárias:

Nós e Folhas: Em uma árvore binária, os nós podem conter informações ou dados, enquanto as folhas são os nós que não têm filhos. A característica distintiva das árvores binárias é que cada nó pode ter no máximo dois filhos.

1. Raiz: A raiz é o nó inicial da árvore, a partir do qual todos os outros nós podem ser acessados. É o nível mais alto da hierarquia da árvore.
2. Subárvore Esquerda e Direita: Cada nó em uma árvore binária possui duas subárvores, uma à esquerda e outra à direita. Cada uma dessas subárvores, por si só, é uma árvore binária, permitindo a construção de estruturas hierárquicas complexas.
3. Altura e Profundidade: A altura de um nó é definida como o número de arestas no caminho mais longo entre ele e uma folha. A profundidade de um nó é o número de arestas no caminho entre ele e a raiz. Entender essas métricas é essencial para a análise de desempenho de algoritmos que usam árvores binárias.
4. Árvore Binária de Busca (BST): Uma variante importante das árvores binárias é a Árvore Binária de Busca, onde as chaves (ou valores) dos nós na subárvore esquerda são menores que o valor do nó raiz, e as chaves na subárvore direita são maiores. Isso permite uma rápida pesquisa e ordenação.

Implementação em C/C++:

A linguagem C/C++ é uma escolha comum para a implementação de estruturas de dados como árvores binárias devido ao seu desempenho e controle de memória. Para criar uma árvore binária, você geralmente usa estruturas de dados personalizadas para representar os nós e implementa funções para adicionar, remover e manipular os nós na árvore. O uso de ponteiros é comum para representar as relações entre os nós.

Aplicações:

As árvores binárias são fundamentais em uma ampla variedade de algoritmos e aplicações, incluindo:

* Pesquisa e classificação eficientes de dados
* Estruturas de diretórios em sistemas de arquivos
* Modelagem de árvores genealógicas e hierarquias organizacionais
* Análise sintática em compiladores
* Expressões aritméticas e avaliação de fórmulas matemáticas
* Algoritmos de roteamento em redes de computadores
* Estruturas de banco de dados, como índices de árvores B

**4. Metodologia**

A pesquisa será conduzida através da análise de casos de estudo, medições de desempenho e comparações práticas entre as linguagens. Além disso, consideraremos fatores como a legibilidade do código, a facilidade de manutenção, a segurança e o ecossistema de bibliotecas disponíveis.

**4.1 Em Go (Golang):**

Em Go, você criaria uma lista ligada usando structs e referências a essas structs. Primeiro, você define uma struct para representar um nó da lista e outra struct (ou um tipo de dado) para representar a própria lista. Em seguida, você usa essas estruturas para criar os nós e vinculá-los para formar a lista. A alocação de memória é tratada automaticamente pelo coletor de lixo, então você não precisa se preocupar com a liberação de memória.

// Definindo a estrutura do nó

type Node struct {

data int

next \*Node

}

// Definindo a estrutura da lista ligada

type LinkedList struct {

head \*Node

}

// Função para adicionar um nó ao final da lista

func (ll \*LinkedList) Append(data int) {

newNode := &Node{data: data, next: nil}

if ll.head == nil {

ll.head = newNode

} else {

current := ll.head

for current.next != nil {

current = current.next

}

current.next = newNode

}

}

**4.2 Em C++:**

Em C++, a criação de uma lista ligada envolve a alocação manual de memória e o uso de ponteiros. Você declara uma classe para representar um nó e, possivelmente, uma classe separada para representar a lista ligada. O programador é responsável pela alocação e liberação de memória para os nós.

// Definindo a classe do nó

class Node {

public:

int data;

Node\* next;

Node(int val) : data(val), next(nullptr) {}

};

// Definindo a classe da lista ligada

class LinkedList {

public:

Node\* head;

LinkedList() : head(nullptr) {}

// Função para adicionar um nó ao final da lista

void append(int val) {

Node\* newNode = new Node(val);

if (head == nullptr) {

head = newNode;

} else {

Node\* current = head;

while (current->next != nullptr) {

current = current->next;

}

current->next = newNode;

}

}

};

Em resumo, em Go, a criação de listas ligadas é mais simples e envolve a definição de structs, enquanto em C++, requer a criação de classes e gerenciamento manual de alocação de memória usando ponteiros. A linguagem Go lida automaticamente com a liberação de memória, simplificando a tarefa, enquanto em C++, é necessário cuidado extra para evitar vazamentos de memória.

**3.2.1 Código desenvolvido em aula:**

Este código em C implementa uma árvore de busca binária (BST) e oferece funções para realizar operações comuns em árvores binárias, como inserção, exclusão e travessia (in-order, pre-order e post-order). Vou explicar o código em partes:

1. Inclusão de Bibliotecas:

Uma imagem contendo Texto

Descrição gerada automaticamente

Aqui, estão incluídas as bibliotecas padrão de entrada/saída (stdio.h) e a biblioteca padrão de alocação dinâmica de memória (stdlib.h), que são necessárias para operações de entrada/saída e alocação de memória.

1. Definição da Estrutura do Nó:

Texto

Descrição gerada automaticamente

Define a estrutura de um nó da árvore binária. Cada nó contém um valor inteiro (valor), um ponteiro para o nó filho esquerdo (esquerda) e um ponteiro para o nó filho direito (direita).

1. Função criarNovoNo:



Essa função cria um novo nó com o valor especificado e retorna um ponteiro para esse novo nó.

1. Função inserirNaBST:



Essa função insere um novo nó com um valor especificado na árvore de busca binária. Ela mantém a propriedade da BST, onde os nós à esquerda são menores e os nós à direita são maiores que o nó pai.

1. Funções de Traversing (Percorrimento):

percorrerEmOrdem: Percorre a árvore em ordem (in-order).

percorrerEmPreOrdem: Percorre a árvore em pré-ordem (pre-order).

percorrerEmProOrdem: Percorre a árvore em pós-ordem (post-order).

Todas essas funções imprimem os valores dos nós da árvore de acordo com o tipo de travessia.

1. Função buscarUmNo:



Essa função procura um nó com um valor específico na árvore e retorna um ponteiro para esse nó, se encontrado.

1. Funções de Exclusão:

excluirUmNoFolha: Exclui um nó folha (sem filhos).

excluirUmNoComUmaFolha: Exclui um nó que tem apenas um filho.

excluirComDoisFilhos: Exclui um nó que tem dois filhos.

Todas essas funções removem um nó com um valor específico da árvore, mantendo a propriedade da BST.

1. Função findMinValueNode:



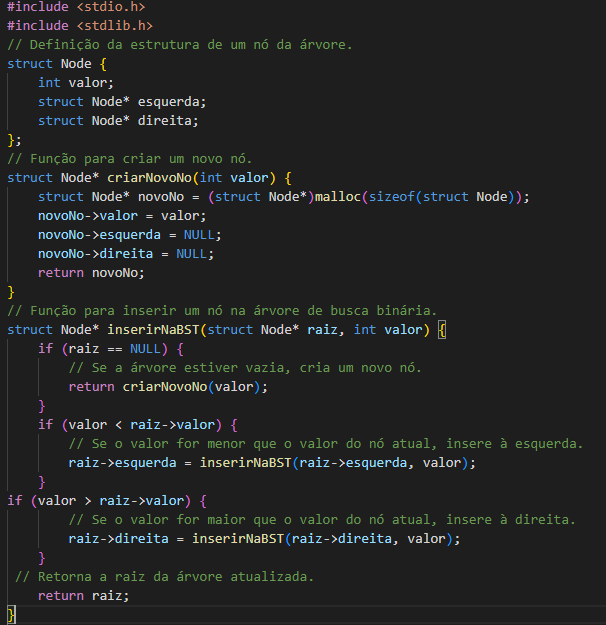
Essa função encontra o nó com o valor mínimo em uma subárvore. Ela é usada para encontrar o nó sucessor na exclusão de um nó com dois filhos.

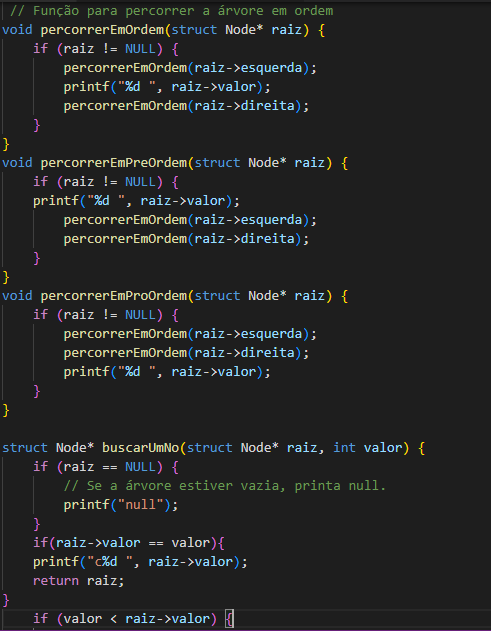
1. Função main:

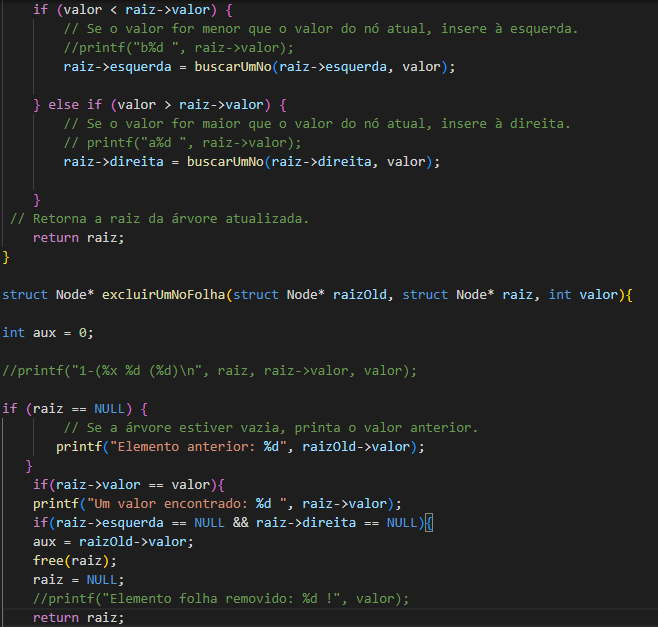
* Cria a raiz da árvore.
* Insere valores na árvore.
* Oferece um menu interativo para realizar operações na árvore, como exclusão e impressão.
* Utiliza um loop para continuar até que o usuário escolha sair.

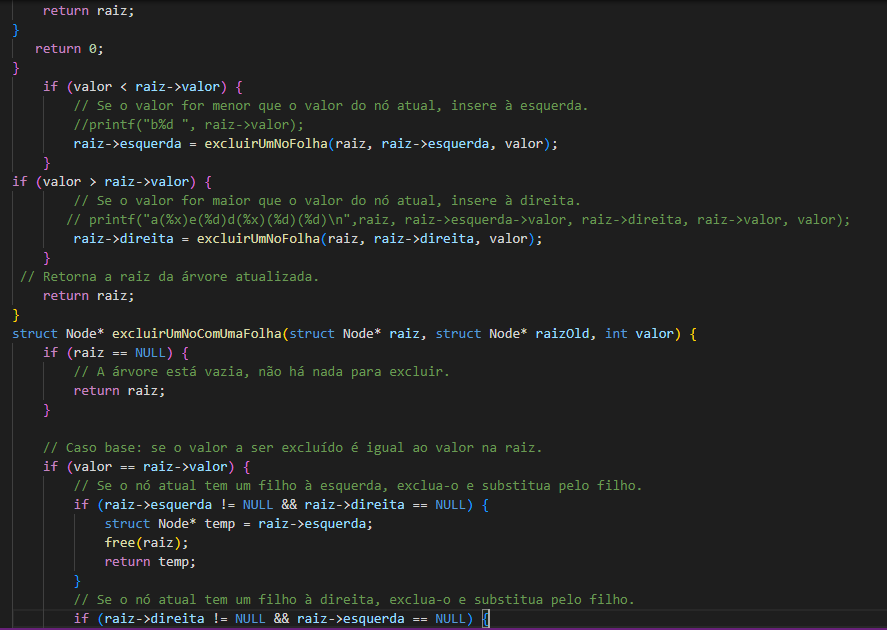
O código tem um menu simples para interagir com a árvore, permitindo ao usuário escolher a operação desejada. É importante notar que há alguns problemas de formatação e de lógica em algumas partes do código que podem precisar de correções, como a exibição dos resultados após a exclusão e a saída do programa.

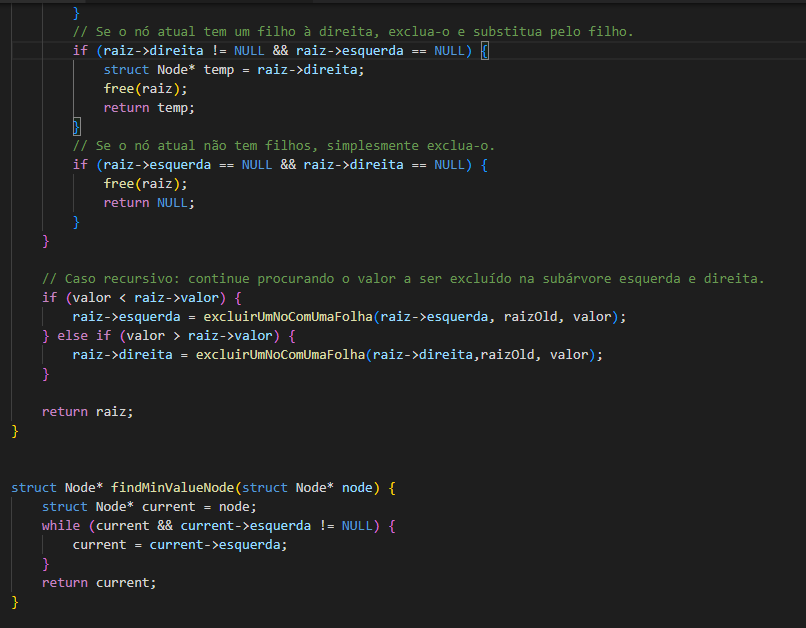
**Segue o código completo logo abaixo.**

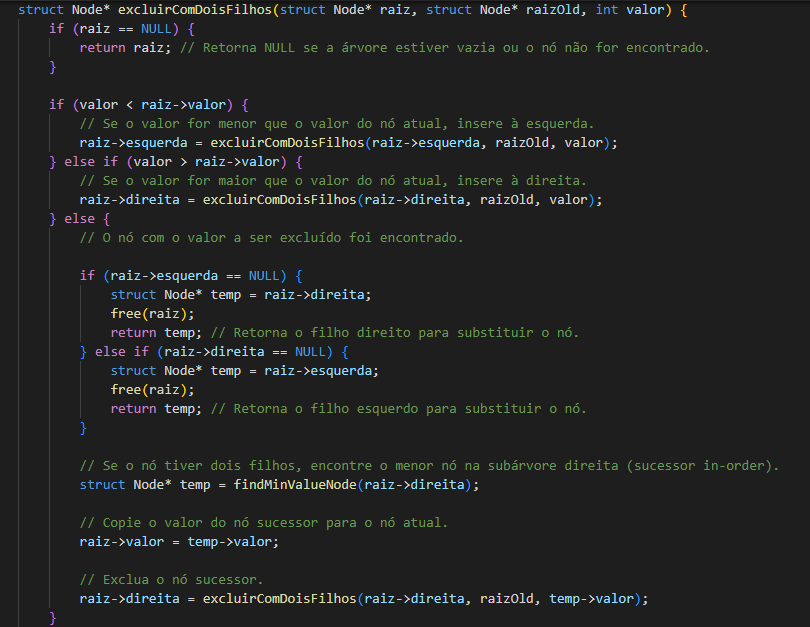


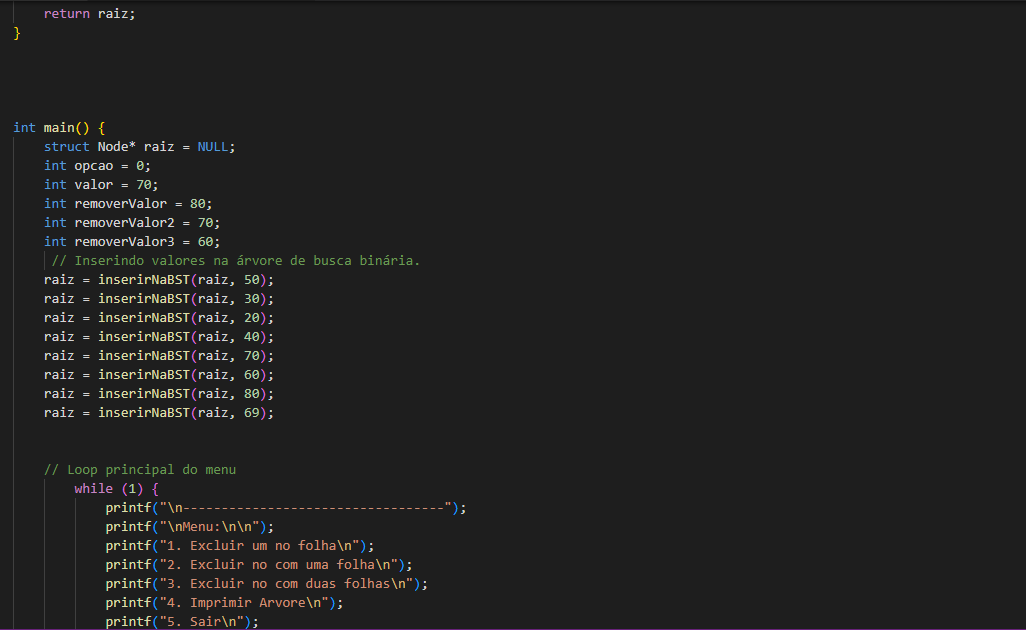


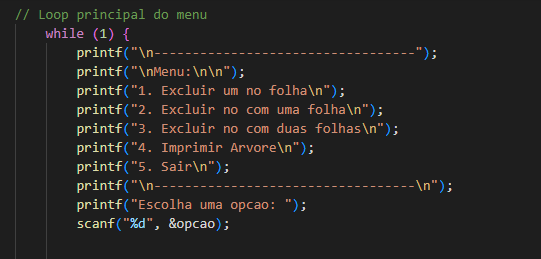


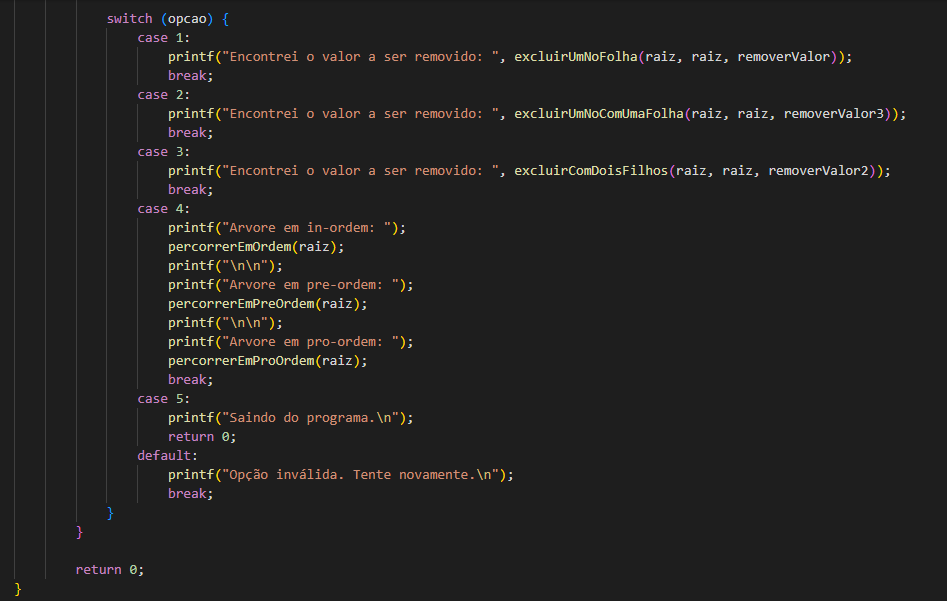






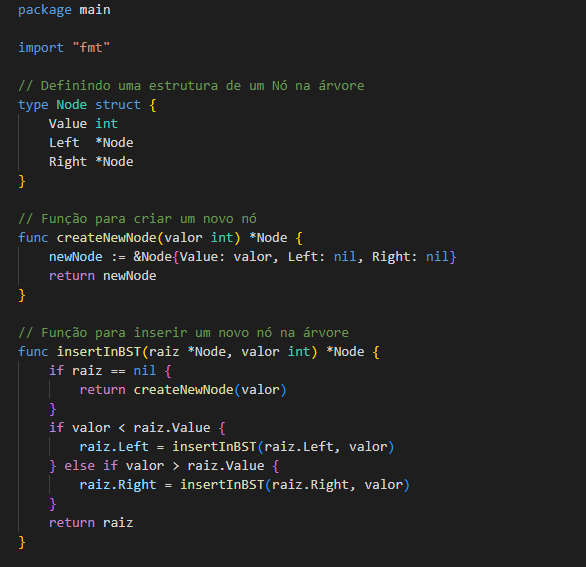


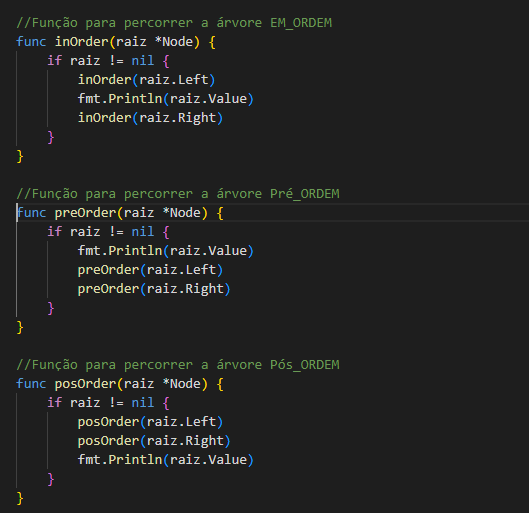


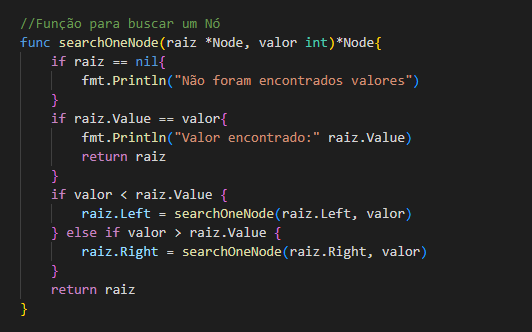


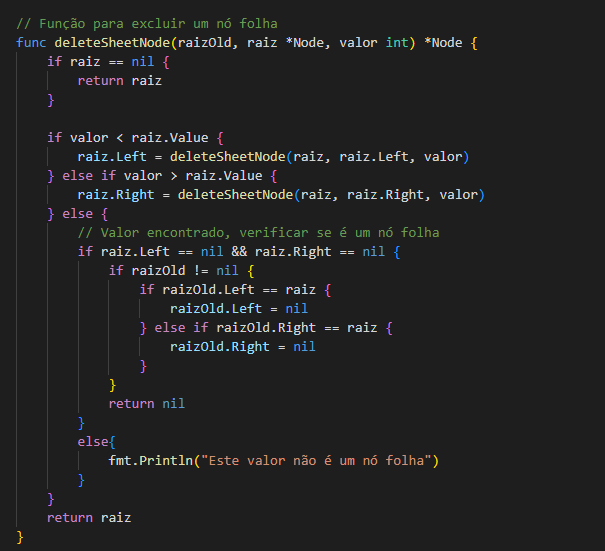
**4.3 Árvore Binária em Go (Golang)**

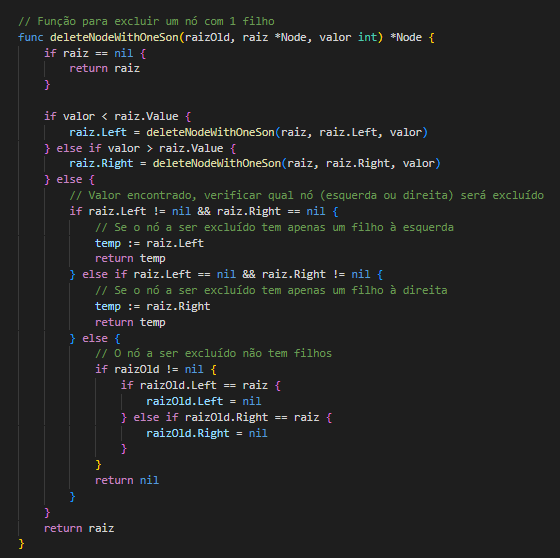
Em Go, a implementação de uma árvore binária geralmente envolve a criação de uma estrutura de nó (node) que contém os dados e as referências para os nós filhos (esquerdo e direito). Aqui está s árvore binária em Go realizada em C/C++ durante o semestre:

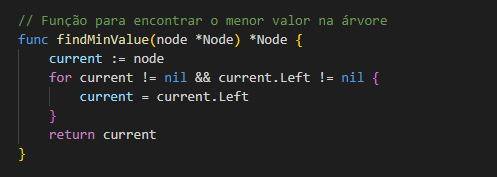


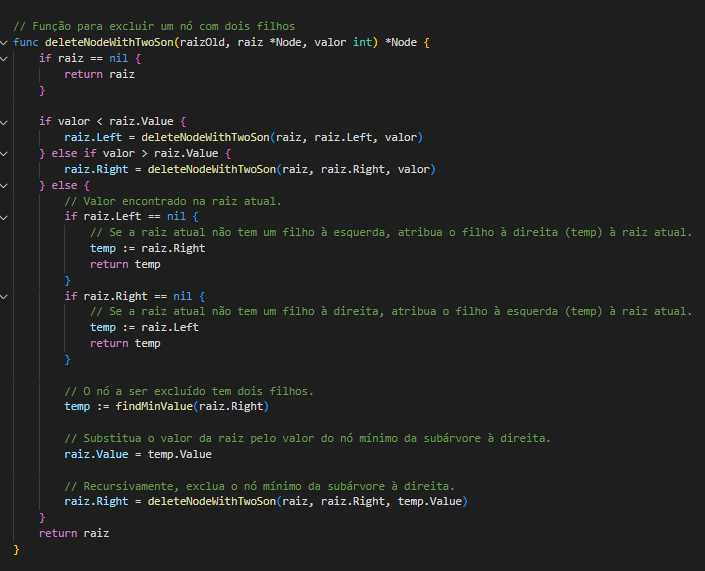




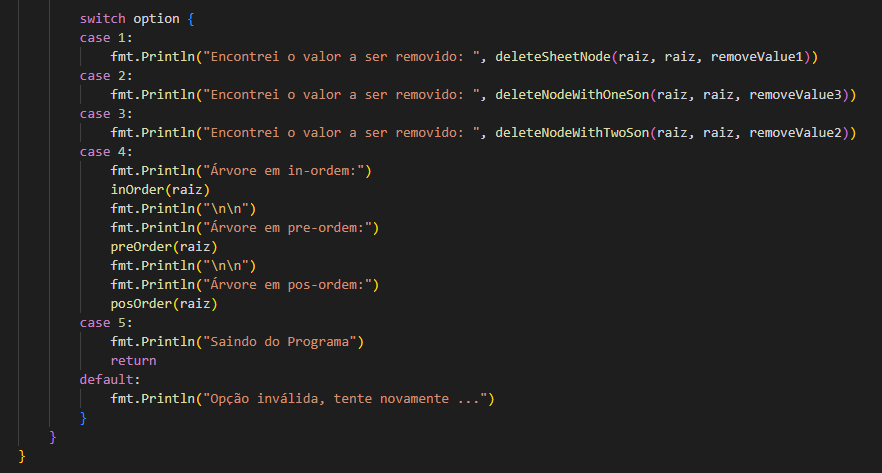












**4.4 Árvore Binária em C/C++**

Em C/C++, a implementação de uma árvore binária também envolve a criação de uma estrutura de nó (node) e a manipulação de ponteiros. Aqui está um exemplo similar em C++:

#include <iostream>

struct TreeNode {

int data;

TreeNode\* left;

TreeNode\* right;

};

int main() {

// Criando uma árvore binária simples

TreeNode\* root = new TreeNode{1, nullptr, nullptr};

root->left = new TreeNode{2, nullptr, nullptr};

root->right = new TreeNode{3, nullptr, nullptr};

// Acesso aos nós e dados

std::cout << "Raiz: " << root->data << std::endl;

std::cout << "Filho esquerdo: " << root->left->data << std::endl;

std::cout << "Filho direito: " << root->right->data << std::endl;

// Lembre-se de liberar a memória alocada

delete root->left;

delete root->right;

delete root;

return 0;

}

Neste exemplo em C++, estamos criando a árvore binária da mesma maneira, usando ponteiros para os nós. É importante lembrar de liberar a memória alocada para evitar vazamentos de memória.

**4.5 Comparação**

Ambas as linguagens, Go e C/C++, permitem a implementação de árvores binárias de forma eficiente. No entanto, há algumas diferenças notáveis:

**1.** Gerenciamento de Memória: Em Go, o gerenciamento de memória é automático (coleta de lixo), o que torna mais fácil evitar vazamentos de memória. Em C/C++, é necessário gerenciar manualmente a alocação e desalocação de memória, como demonstrado no exemplo C++.

**2.** Sintaxe e Convenções: A sintaxe em Go é mais simples e consistente em comparação com C/C++, o que pode tornar o código mais legível e menos propenso a erros.

**3.** Desempenho: Em termos de desempenho puro, C/C++ geralmente oferece maior controle sobre a manipulação de memória, o que pode ser benéfico para estruturas de dados como árvores binárias, especialmente em cenários de alto desempenho.

A escolha entre Go e C/C++ para implementar uma árvore binária dependerá das necessidades específicas do projeto, considerando fatores como desempenho, facilidade de desenvolvimento e gerenciamento de memória. Cada linguagem tem suas vantagens e desvantagens, e a escolha deve ser feita com base nas características do projeto.

**5. Conclusão**

Este artigo se propõe a fornecer uma visão abrangente das vantagens e desvantagens de Go em relação a C/C++. Esperamos que esta análise possa ajudar os desenvolvedores a tomar decisões mais informadas sobre qual linguagem de programação utilizar em seus projetos, considerando as necessidades específicas de cada aplicação.

**6. Referências**

Para embasar nossa pesquisa, recorreremos a fontes confiáveis e estudos anteriores relacionados a Go, C e C++. Alguns dos recursos iniciais incluem:

PIKE, R.; THOMPSON, K.; GRIESEMER, R. The Go Programming Language. ACM SIGPLAN Notices, v. 45, n. 10, 2010, p. 1-2.

STROUSTRUP, Bjarne. The C++ Programming Language. Addison-Wesley Professional, 2013.

WEISS, Mark Allen. Estruturas de Dados e Algoritmos em C++. 2ª ed. Pearson, 2011. 950 p. ISBN: 978-8576058814.

KERNIGHAN, Brian W.; RITCHIE, Dennis M. The C Programming Language. Prentice Hall, 1988.

SINGH, Aahan. Data Structures with Go - Part 1: Linked Lists. Disponível em: https://aahansingh.com/data-structures-with-go-part-1-linked-lists.

Bogotobogo. GoLang Linked List. Disponível em: https://www.bogotobogo.com/GoLang/GoLang\_Linked\_List.php.