### 

### **Introdução**

As estruturas de dados constituem um pilar essencial da computação, sendo fundamentais para a organização, o armazenamento e a manipulação eficiente de dados em qualquer sistema. A correta implementação e compreensão dessas estruturas é, portanto, imprescindível para a construção de aplicações bem estruturadas .

Este trabalho propõe uma análise individual e detalhada de como essas estruturas são concebidas e implementadas em duas linguagens de programação distintas: JavaScript e Zig. A escolha recai sobre este par por elas representarem filosofias e ecossistemas radicalmente diferentes. JavaScript, uma linguagem de alto nível amplamente difundida no desenvolvimento web, oferece um paradigma flexível e dinâmico, com gerenciamento automático de memória. Zig, por sua vez, é uma linguagem de sistemas moderna que prioriza o controle de baixo nível, a performance previsível e a simplicidade, exigindo um gerenciamento manual de recursos.

O objetivo não é compará-las diretamente, mas examinar as características de implementação inerentes a cada uma. Será analisada a sintaxe específica, os métodos de organização dos dados, a lógica por trás de cada estrutura e questões de desempenho intrínsecas a cada ambiente de execução. Esta abordagem individual permite um entendimento de como o design de cada linguagem molda a construção e a eficácia de suas estruturas de dados, fornecendo insights valiosos para dominar esses conceitos em diferentes contextos tecnológicos.

### 

### **Árvores Binárias em JavaScript**

### **Conceituação e Estrutura**

Diferente de outras linguagens que são tipadas estaticamente, o JavaScript, com sua natureza dinâmica baseada em protótipos, oferece uma maneira flexível ainda que manual de se implementar estruturas de dados complexas como as árvores binárias. Nesta linguagem, a ausência de ponteiros explicitamente definidos é suprida por uma referência de objetos, onde cada nó é tratado como um objeto independente que armazena seu valor e mantém referências (que podem ser null) para seus filhos sejam à esquerda ou à direita (GRONER, 2019). Esse tipo de modelagem aproveita o sistema de gerenciamento de memória e coleta de lixo do JavaScript, simplificando a alocação e desalocação de nós.

#### **Princípios de Funcionamento e a Implementação com Classes JS**

Em JavaScript a implementação se utiliza da sintaxe ‘Class’ que encapsula o comportamento da árvore e cada um de seus ‘nós’, oferecendo uma abstração clara e organizada. A classe ‘No’ tem a responsabilidade de criar cada elemento da árvore, iniciando com um valor e definindo as referências aos filhos como ‘null’, indicando uma ausência de descendentes.

A operação de inserção, central para a construção da árvore, é comumente implementada usando métodos recursivos. O algoritmo compara o valor do novo nó com o valor do nó atual. Se for menor, a tentativa de inserção é feita na subárvore esquerda; se for maior ou igual, esta tentativa ocorrerá na direita. Esse processo é repetido recursivamente até encontrar uma posição vaga(‘null’), onde o novo nó é efetivamente alocado. É importante notar que, embora poderosa, a recursividade em JavaScript pode levar a erros do tipo stack overflow em árvores muito desbalanceadas e profundas, um risco inerente à escolha do método(CORMEN et al., 2012) .

Para operações de busca, a mesma lógica de comparação e recursão é aplicada. A eficiência média O(log n) é atingida apenas se a árvore estiver balanceada. Caso os valores sejam inseridos em ordem crescente ou decrescente, a árvore degenera em uma lista ligada, e a complexidade de busca se torna O(n), um ponto de atenção crítico para o desempenho em JavaScript, considerando que ele apenas executa uma tarefa por vez (single-threaded.).

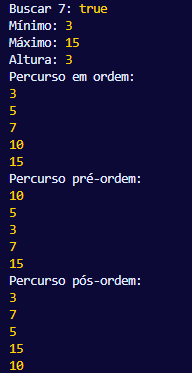
As travessias (em-ordem, pré-ordem e pós-ordem) são implementadas como métodos que também utilizam a recursividade para visitar todos os nós. A travessia ‘em-ordem’ é particularmente valiosa em JavaScript para processar sequências ordenadas de dados sem a necessidade de ordenar explicitamente um array, que teria complexidade O(n log n).

**Exemplo de Aplicação e Características da Implementação JS**

A implementação de referência (disponível no repositório) ilustra uma árvore binária de busca completa. Um aspecto característico do JavaScript nesta implementação é o uso de métodos privados (usando o prefixo #), como ‘#inserirNo’ e ‘#buscarNo’ , para encapsular a lógica recursiva interna, escondendo-a da interface pública da classe e prevenindo seu uso inadvertido.

Outra particularidade é a manipulação de valores. O JavaScript permite que a árvore armazene qualquer tipo de valor (números, strings, até objetos), desde que eles possam ser comparados usando os operadores < e > . Isso pode ser tanto uma vantagem (flexibilidade) quanto uma fonte de bugs, se tipos diferentes forem inseridos inadvertidamente.

O gerenciamento de memória é implícito. Quando um nó não é mais referenciado (por exemplo, ao ser removido de uma estrutura pai), ele se torna elegível para coleta de lixo, liberando o programador da responsabilidade de desalocação manual, comum em linguagens de mais baixo nível.



### **Filas em JavaScript**

#### **Conceituação e Princípio FIFO**

As filas são estruturas de dados lineares do tipo FIFO (First In, First Out), onde o primeiro elemento inserido é o primeiro a ser removido. Essa estrutura é fundamental em JavaScript, não apenas na implementação de diversos algoritmos, como também no próprio funcionamento da linguagem, que utiliza uma *event loop* baseada em filas de tarefas para gerenciar Pilha de Chamadas e a Fila de Retornos.

Em JavaScript, a ausência de uma implementação nativa de fila na biblioteca padrão é contornada utilizando arrays e como também seus métodos embutidos. Por serem flexíveis os arrays em JavaScript com dinamicidade de tamanho e métodos de alta ordem se tornam eficazes para simular os comportamentos de filas, embora com ressalvas sobre seu uso, principalmente se levar em consideração a sua performance para operações em larga escala.

#### **Implementação de Fila Simples com Arrays JS**

A implementação de uma fila simples em JavaScript normalmente envolve a criação de uma classe que encapsula um array e fornece métodos que mapeiam as operações fundamentais:

Enfileirar(elemento): Utiliza o método Array.push( ) para adicionar o elemento ao final do array (o final da fila).

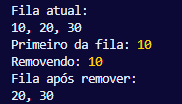
Desenfileirar: Utiliza o método Array.shift( ) para remover e retornar o primeiro elemento do array (a frente da fila).

Frente( ): Acessa o elemento no índice [0] do array para inspecionar o próximo a ser atendido sem removê-lo.

EstaVazia( ): Verifica se o comprimento (length) do array é zero.

Um ponto crítico de performance nesta implementação reside no método shift( ). Como o JavaScript implementa arrays na forma de objetos, a operação shift que remove o primeiro elemento tem complexidade de tempo O(n), pois requer o reindexamento de todos os elementos restantes no array. Isso torna a operação desenfileirar( ) muito custosa para filas grandes, uma limitação a ser considerada no design de sistemas em JavaScript.

A robustez da implementação é aumentada pela inclusão de verificações de estado, como conferir se a fila está vazia antes de tentar remover um elemento, prevenindo comportamentos indefinidos.



**A Fila Dupla (Deque)**

A fila dupla, ou Deque (Double-Ended Queue), estende o conceito da fila simples, permitindo que a inserção e a remoção de elementos ocorram tanto no início quanto no final da estrutura. Essa flexibilidade é implementada em JavaScript combinando os métodos push( ) e pop( ) (para operações no final) com unshift( ) e shift( ) (para operações no início).

A classe Deque herda a mesma base de array da fila simples, mas expõe uma API ampliada:

adicionarFrente(elemento) e adicionarTras(elemento): Usam unshift( ) e push( ), respectivamente

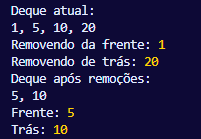
removerFrente() e removerTras(): Usam shift() e pop(), respectivamente.

É crucial notar que, assim como shift, o método unshift também possui complexidade O(n) em engines de JavaScript padrão, pois precisa reindexar o array ao inserir um elemento no início. Portanto, embora o Deque ofereça maior flexibilidade operacional, ele compartilha a mesma potencial limitação de performance para volumes muito altos de dados. Para cenários que exigem performance constante (O(1)) em ambas as extremidades, uma implementação com listas ligadas seria mais adequada, ainda que mais complexa.

#### **Exemplo de Aplicação**

As filas são amplamente utilizadas em JavaScript para gerenciamento de tarefas em sequência, como em:   
 Buffer de comandos em aplicações interativas.   
 Algoritmos de busca em largura (BFS) para grafos e árvores.   
 Processamento de mensagens em sistemas de comunicação.

A implementação de referência (disponível no repositório) demonstra o funcionamento prático de ambas as estruturas. A execução do código exemplifica o comportamento FIFO da fila simples e a natureza bidirecional do Deque.



**Conjuntos em JavaScript**

#### **Conceituação e Propriedades de Unicidade**

Um conjunto é uma coleção de elementos distintos e não ordenados, fundamentado diretamente na teoria matemática dos conjuntos. Em JavaScript, embora exista a estrutura de dados Set nativa, é instrutivo compreender como implementar manualmente essa estrutura para dominar os conceitos subjacentes e para contextos onde o uso de estruturas nativas não é possível ou desejado.

A principal característica de um conjunto é que nenhum elemento pode aparecer mais de uma vez na coleção. Essa propriedade é frequentemente implementada utilizando-se a estrutura de objeto em JavaScript, onde as chaves são únicas por natureza, oferecendo assim uma maneira eficiente de verificar a existência de um elemento.

#### **Implementação Manual com Objetos JS**

A implementação de um conjunto em JavaScript envolve a criação de uma classe que utiliza um objeto interno para armazenar os elementos. A chave e o valor do objeto são geralmente definidos como o próprio elemento, aproveitando o fato de que as chaves de um objeto em JavaScript são sempre strings ou símbolos, o que implica em uma conversão implícita para string para outros tipos de dados. Essa conversão é um ponto de atenção, pois pode levar a comportamentos inesperados.

As operações básicas são implementadas da seguinte forma:

adicionar(elemento): Antes de inserir, verifica-se se a chave já existe no objeto interno usando Object.prototype.hasOwnProperty.call(), um método robusto para evitar conflitos. Se não existir, o elemento é adicionado como uma chave-valor.  
remover(elemento): Utiliza o operador delete para remover a propriedade do objeto interno correspondente ao elemento.

contem(elemento): Verifica a existência da chave no objeto interno, também utilizando a verificação segura com hasOwnProperty.

Uma decisão de projeto importante nesta implementação é o retorno de um booleano nas operações de adição e remoção, informando se a operação foi bem-sucedida (isto é, se o elemento foi de fato adicionado ou removido), o que aumenta a utilidade da classe para o usuário.

#### **Operações de Conjunto e Iteração**

Além das operações básicas, é fundamental implementar as operações clássicas da teoria dos conjuntos:

uniao(outroConjunto): Cria um novo conjunto e adiciona todos os elementos do conjunto atual e do conjunto passado como parâmetro. A propriedade de unicidade é mantida naturalmente pelo método adicionar

interseccao(outroConjunto): Percorre os elementos do conjunto atual e adiciona ao novo conjunto apenas aqueles que também estão presentes no outro conjunto

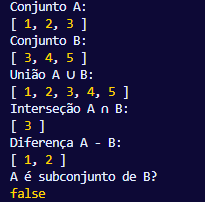
diferenca(outroConjunto): Percorre os elementos do conjunto atual e adiciona ao novo conjunto apenas os que não estão presentes no outro conjunto.

ehSubconjunto(outroConjunto): Utiliza o método every() para verificar se todos os elementos do conjunto atual estão contidos no outro conjunto.

Para permitir a interação e inspeção, os métodos valores() e imprimir() são implementados. O método valores() utiliza Object.values() para retornar um array com todos os elementos, enquanto imprimir() exibe esse array no console. A conversão para array é necessária porque o objeto interno não preserva uma ordem específica, reforçando a natureza não ordenada do conjunto.

#### **Exemplo de Aplicação e Comportamento**

Conjuntos são amplamente utilizados em algoritmos para eliminar duplicatas de listas, verificar a existência de elementos de forma eficiente e realizar comparações lógicas entre coleções. A execução do código de exemplo demonstra o comportamento esperado:



### **PriorityQueue em Zig**

A fila de prioridade (Priority Queue) é uma estrutura de dados de um tipo especial de fila convencional, mas com a característica distintiva de que cada elemento inserido recebe um nível de prioridade. Dessa forma, a remoção não segue estritamente a ordem de chegada (FIFO), e sim o critério de prioridade: o primeiro elemento a ser removido será sempre aquele com maior prioridade (Programiz, s.d.).

Essa estrutura costuma ser bastante eficiente, apresentando, em geral, tempo de inserção e remoção da ordem de O(log n), onde n representa a quantidade de elementos armazenados.

A biblioteca padrão do Zig oferece uma implementação própria dessa estrutura. De acordo com sua documentação oficial, a std.PriorityQueue é definida como uma "priority queue for storing generic data", ou seja, uma fila de prioridade destinada ao armazenamento de dados genéricos. Sua principal função é garantir que o elemento de maior prioridade esteja sempre acessível para consulta ou remoção.

A prioridade dos elementos é determinada por uma função de comparação (compareFn) fornecida no momento da criação da fila. Esse tipo de estrutura é particularmente útil em problemas que demandam o processamento contínuo do item mais importante, como em:

Algoritmos de agendamento de tarefas;

Busca de caminhos (ex.: algoritmo de Dijkstra).

A PriorityQueue em Zig utiliza internamente uma heap binária (binary heap) construída sobre um array. A estrutura conta com os seguintes componentes:

**items: []T** – slice genérico que armazena os elementos. A ordenação é mantida pela lógica de heap;

**cap: usize** – capacidade atual do array, indicando quantos elementos podem ser armazenados antes de ser necessário realocar memória;

**allocator: Allocator** – responsável pela alocação dinâmica de memória, expandindo o array quando necessário.

As principais operações dessa estrutura incluem:

**add** - insere um novo elemento na fila, reorganizando a heap para manter a prioridade.

**remove** - remove e retorna o elemento de maior prioridade.

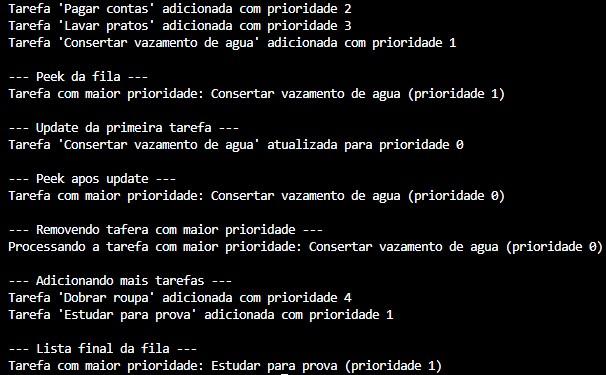
**peek** - retorna o elemento de maior prioridade sem removê-lo.

**deinit** - libera a memória alocada pela fila, evitando vazamentos.

#### **Exemplo Prático em Zig**

Para ilustrar o uso da PriorityQueue na prática, foi criado um programa simples de agendamento de tarefas. Nele, cada tarefa recebe um nível de prioridade, e a fila garante que a tarefa mais urgente seja sempre a primeira a ser processada.

Execução do programa produz a seguinte saída, que demonstra a ordem de processamento baseada na prioridade das tarefas:



### **HashMap em Zig**

Uma tabela de hash (hash table), também chamada de mapa de hash (hash map), é uma estrutura de dados que armazena pares de chave e valor, permitindo operações de inserção, busca e remoção de forma altamente eficiente. Em média, essas operações apresentam complexidade O(1), embora possam se degradar para O(n) em casos de colisões, quando múltiplas chaves são mapeadas para o mesmo índice. Segundo Tapia-Fernández, García-García e García-Hernandez (2022), a estrutura pode ser formalmente entendida como uma correspondência entre dois conjuntos arbitrários, K (chaves) e V (valores), representando pares (k, v).

No Zig, a biblioteca padrão oferece a estrutura std.HashMap, uma implementação genérica de tabela de hash projetada para garantir operações rápidas de consulta (lookup), inserção e remoção. Assim como em outras linguagens, não há garantia de ordem para os elementos, e qualquer modificação na estrutura invalida iteradores previamente criados — regra explicitada no Zig devido ao foco em segurança e previsibilidade na gestão de memória.

Uma característica importante destacada na documentação é que o HashMap suporta um fator de carga (load factor) elevado, chegando a 80% por padrão. Isso significa que o mapa consegue manter boa eficiência mesmo quando bastante preenchido. Essa estrutura é particularmente útil em aplicações que exigem associações rápidas entre chaves e valores, como:

Contagem de frequência de palavras em textos;

Criação de caches, para evitar recomputações desnecessárias;

Tabelas de tradução, como códigos de erro mapeados para mensagens;

Cadastro indexado, por exemplo, matrículas de alunos associadas a dados pessoais.

A implementação do HashMap em Zig é parametrizada por alguns dos componentes a seguir:

**K (Key)** – tipo da chave;

**V (Value)** – tipo do valor;

**Allocator** – gerencia a alocação de memória da tabela.

Além disso, a estrutura utiliza internamente os tipos Entry, KV e Iterator, que permitem representar pares chave-valor e percorrer o mapa.

As operações fundamentais oferecidas pelo HashMap incluem:

**Criação e destruição:**

● init(allocator) – cria um novo mapa associado a um alocador.

● deinit() – libera a memória utilizada, sendo obrigatório chamar essa função para evitar vazamentos.

**Inserção**:

● put(chave, valor) – insere ou substitui o valor associado a uma chave.

● putNoClobber(chave, valor) – insere somente se a chave ainda não existir; caso contrário, retorna erro.

**Busca e acesso**:

● get(chave) – retorna o valor associado a uma chave ou null caso não exista.

● contains(chave) – verifica se determinada chave está presente no mapa.

**Remoção**:

● remove(chave) – remove a entrada associada e retorna true em caso de sucesso.

● fetchRemove(chave) – remove e retorna o par chave-valor caso exista.

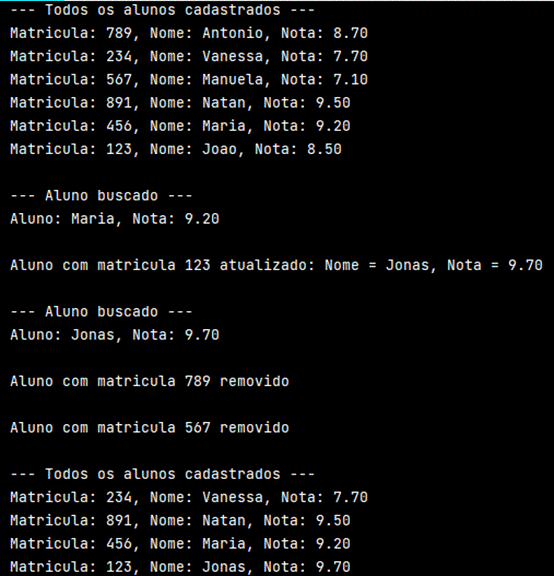
**Iteração:**

● iterator() – fornece um iterador para percorrer os elementos. Importante destacar que modificações no mapa invalidam qualquer iterador ativo, o que evita comportamentos indefinidos.

#### **Exemplo Prático em Zig**

Para ilustrar os conceitos de HashMap na prática, foi elaborado um programa simples de cadastro de alunos. Nele, uma tabela de hash é utilizada para associar a matrícula do aluno (a chave) aos seus nomes e suas notas, (o valor), permitindo operações rápidas de cadastro, busca, atualização e remoção.

A execução do programa resulta na seguinte saída, que mostra como o HashMap gerencia os dados dos alunos:



### Array List em Zig

## **O std.ArrayList(T) e a Gestão Explícita de Memória em Zig**

O std.ArrayList(T) é a estrutura de dados em Zig para se criar uma lista dinâmica — uma coleção contígua de elementos de um tipo T que pode crescer ou encolher em tempo de execução. Em contraste com as arrays de tamanho fixo ([N]T) e as slices de apenas visualização ([]T), o ArrayList gerencia ativamente a memória necessária para armazenar seus elementos, tornando-o essencial para a maioria das aplicações dinâmicas.

O aspecto mais importante do ArrayList é que ele não gerencia a memória por conta própria; ele exige que você forneça um alocador (std.mem.Allocator). Essa é a filosofia central de Zig: a alocação de memória é explícita. O alocador é responsável por reservar mais espaço na heap (memória dinâmica) quando a lista atinge sua capacidade máxima e precisa crescer.

### **O Fluxo de Vida e os Comandos do ArrayList**

O uso correto de um ArrayList segue um ciclo rigoroso de alocação e liberação de recursos, garantido pela palavra-chave defer.

#### **1. Inicialização e Alocação da Memória Base**

Antes de usar o ArrayList, precisamos de uma fonte de memória: o alocador.

● var gpa = std.heap.GeneralPurposeAllocator(.{}){};: Este comando cria uma instância de um alocador de uso geral (GeneralPurposeAllocator ou GPA). Ele é o gerente de memória de alto nível.

● const allocator = gpa.allocator();: Obtemos a interface (Allocator) que passaremos para as estruturas de dados.

● var list = std.ArrayList(T).init(allocator);: Aqui, o ArrayList é inicializado, recebendo o alocador que usará para todas as operações futuras de redimensionamento e crescimento.

#### **2. Adição de Elementos (append e dupe)**

O comando fundamental para adicionar itens é o append. No entanto, quando trabalhamos com tipos que não são copiados por valor (como slices ou strings, representadas por []const u8), a memória deve ser gerenciada com cuidado.

● try list.append(item);: Adiciona um elemento ao final da lista. A operação pode falhar com o erro error.OutOfMemory se o alocador não conseguir encontrar espaço para o crescimento da lista, por isso usamos o prefixo try.

● try allocator.dupe(u8, fatia);: Este comando é crucial no nosso exemplo de lista de tarefas. Como strings em Zig são slices (apenas referências), dupe aloca um novo bloco de memória na heap, copia o conteúdo da fatia original e retorna a propriedade da nova cópia. O ArrayList armazena esta cópia, garantindo que a memória não seja liberada por engano em outro lugar (evitando Use After Free).

#### **3. Acesso e Modificação**

Para acessar os dados, o ArrayList expõe o campo .items, que é uma slice da memória interna:

● list.items: Retorna uma []T (slice de T). É o caminho para ler ou iterar sobre os elementos.

● list.items.len: Retorna o número atual de elementos (usize).

● \_ = list.orderedRemove(index);: Remove o elemento no index especificado. É importante notar que, antes de chamar orderedRemove, devemos liberar manualmente a memória do elemento se ele foi alocado via dupe (como a nossa string). A função orderedRemove mantém a ordem dos elementos restantes.

#### **4. O Passo Mais Crítico: Desinicialização (deinit e defer)**

A alocação deve ser pareada com a desliberação. Em Zig, isso é frequentemente garantido pelo defer:

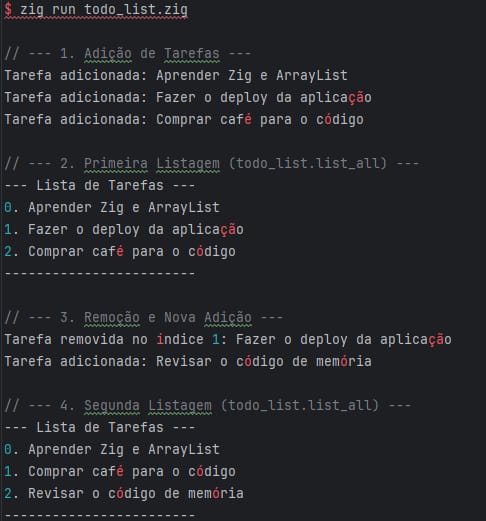
● defer list.deinit();: No final da função, defer garante que a função deinit seja chamada. A list.deinit() libera a memória do array interno que o ArrayList usava para armazenar os elementos.

● A Regra da Duplicação (Cleanup em deinit): Se você usou dupe para alocar memória para os elementos (as strings), a sua função deinit personalizada no TodoList deve primeiro iterar sobre todos os elementos e chamar allocator.free(item) para liberar cada string, antes de chamar list.deinit(). Isso é a essência do gerenciamento explícito de recursos em Zig.

● defer \_ = gpa.deinit();: Finalmente, a desinicialização do GeneralPurposeAllocator libera toda a memória remanescente sob sua responsabilidade.

Em resumo, o std.ArrayList(T) em Zig é uma ferramenta poderosa para coleções dinâmicas, mas exige disciplina com o alocador (Allocator) e a garantia de que cada alocação (init, append, dupe) seja correspondida por uma liberação (deinit, free), geralmente orquestrada pela palavra-chave defer.

#### **Exemplo Prático em Zig:**



### **Linked List em Zig**

## **O conceito de std.SinglyLinkedList(T) em Zig**

A Lista Encadeada Simples (std.SinglyLinkedList(T)) é a estrutura de dados de lista baseada em nós (nodes) na biblioteca padrão de Zig. Ao contrário do ArrayList, que armazena dados em memória contígua e redimensionável, a Linked List armazena dados em blocos discretos (nós), onde cada nó contém o dado e um ponteiro para o próximo nó.

Essa arquitetura tem implicações de performance:

· Vantagem: Inserções e remoções no início (cabeça) são extremamente rápidas (O(1)), pois basta reajustar um único ponteiro.

· Desvantagem: Acessar um elemento no meio da lista exige percorrer todos os nós desde o início (O(N)), e a memória não é contígua, o que pode ser menos eficiente para o cache da CPU.

### **A Filosofia Intrusiva em Zig**

As Linked Lists em std de Zig são geralmente intrusivas (como no exemplo do Histórico Web). Isso significa que os ponteiros de ligação da lista (next) não são estruturas separadas, mas sim um campo embutido dentro da sua própria estrutura de dados (Page no nosso exemplo).

Para isso funcionar, a estrutura que você deseja armazenar (Page) deve conter um campo especial:

Sua Estrutura→Page={dados,link: std.SinglyLinkedList(Page).Node}

### **O Fluxo de Vida e os Comandos Chave da Linked List**

O gerenciamento de memória em uma Linked List é mais granular que no ArrayList, pois cada nó é uma alocação separada na heap.

#### **1. Definição e Alocação dos Nós**

O alocador é usado para criar cada nó individualmente.

● const L = std.SinglyLinkedList(Page);: Define o tipo da nossa lista encadeada, especializada para armazenar o struct Page.

● list: L = L.init();: Inicializa a instância da lista (apenas define o ponteiro first como null).

● try self.allocator.create(Page);: Este comando aloca o espaço de memória para a nova struct Page (que é o nosso nó). O alocador (Allocator) é fundamental, pois ele é quem reserva memória para cada novo nó que entra na lista.

#### **2. Inserção de Elementos (prepend)**

A inserção no início (prepend) é o principal caso de uso para as Linked Lists.

● self.list.prepend(new\_page);: Esta é a operação O(1). Ela insere o ponteiro new\_page na cabeça da lista.

o O ponteiro link.next do new\_page é definido para apontar para a antiga cabeça da lista.

o O ponteiro list.first é atualizado para apontar para new\_page.

● new\_page.\* = .{ .link = .{}, ... };: Antes de usar prepend, precisamos inicializar o conteúdo do novo nó alocado. O campo link: .{} garante que o nó da lista encadeada seja inicializado corretamente.

#### **3. Remoção, Limpeza e Desalocação**

A desinicialização é a parte mais crítica, pois cada nó deve ser desalocado individualmente.

● while (self.list.popFirst()) |node\_ptr|: O loop de limpeza. O método popFirst() remove o primeiro nó da lista e o retorna como um ponteiro opcional (?\*Page). O loop continua até que a lista esteja vazia.

● self.allocator.destroy(node\_ptr);: Após remover o nó da estrutura da lista, este comando libera a memória que foi originalmente alocada para aquela struct Page usando allocator.create. Este é o par obrigatório para a alocação de nós.

● defer history.deinit();: Garantido pelo defer, esta função chama o loop de limpeza acima, liberando todos os nós alocados um por um antes que a memória do GPA (gpa.deinit()) seja liberada.

#### **4. Travessia (Percorrendo a Lista)**

A única maneira de acessar um elemento específico é percorrendo a lista desde o início.

● var current\_node\_ptr = self.list.first;: Começa-se no ponteiro first (a cabeça).

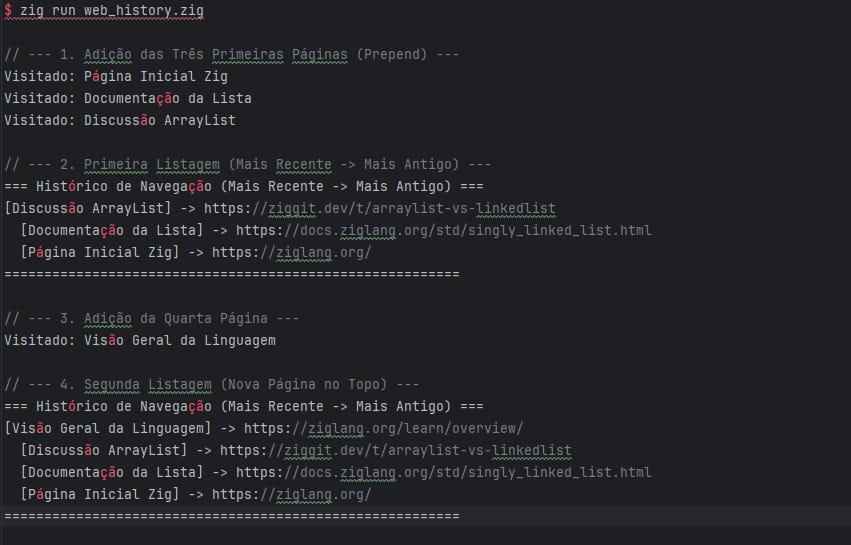
● while (current\_node\_ptr) |page\_ptr| : (current\_node\_ptr = page\_ptr.link.next): O loop percorre a lista. A cada iteração:

o page\_ptr é o ponteiro para o nó atual (\*Page).

o A instrução de continuação (: (...)) avança para o próximo nó, pegando o ponteiro contido no campo de ligação embutido: page\_ptr.link.next.

Em resumo, a Lista Encadeada é a escolha ideal em Zig quando você precisa de inserções e remoções rápidas nas extremidades, aceitando a complexidade de gerenciar a memória de nós individuais e a travessia lenta no acesso aleatório.

### **Exemplo Prático em Zig:**



### **Conclusão**

Através da análise das implementações em JavaScript e Zig, este trabalho evidenciou como as filosofias de design de uma linguagem de programação moldam profundamente a construção e a eficácia de suas estruturas de dados.

JavaScript, com seu paradigma dinâmico e gerenciamento automático de memória, mostrou-se uma linguagem onde a produtividade e a flexibilidade são prioridades. Suas implementações, frequentemente baseadas em arrays e objetos, se beneficiam de uma sintaxe expressiva e de abstrações do alto nível, tornando mais simples o desenvolvimento de estruturas complexas como árvores binárias, filas e conjuntos. No entanto, essa facilidade implica em uma limitação no controle em aspectos de desempenho e alocação de memória, com operações críticas podendo apresentar complexidades inesperadas devido às suas implementações nativas.

Em contrapartida, Zig como a personificação do controle e da previsibilidade. Como linguagem de sistemas, ela exige do desenvolvedor uma gestão manual e explícita dos recursos, o que se traduz em códigos mais verbosos, porém extremamente eficientes e otimizados. As estruturas analisadas, como PriorityQueue, HashMap e ArrayList são construídas sobre alocadores de memória explícitos e oferecem garantias de desempenho sólidas, tornando-as ideais para aplicações onde a performance e a ausência de overhead são críticas.

Portanto, conclui-se que a escolha entre JavaScript e Zig para a implementação de estruturas de dados não é uma questão de superioridade, mas de adequação ao contexto. JavaScript é a ferramenta ideal para o desenvolvimento ágil de aplicações, enquanto Zig destina-se à engenharia de sistemas de alto desempenho, onde cada ciclo de CPU e cada byte de memória tem de ser gerenciado com precisão. Dominar ambas as abordagens permite ao desenvolvedor selecionar a ferramenta mais capaz para transformar requisitos técnicos em soluções robustas e eficientes