UNISENAIPR
GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

MONITORIA DA MATÉRIA DE PROJETO DE SISTEMAS ORIENTADOS A OBJETOS

IMPLEMENTAÇÕES DE LISTAS COM POO

MATERIAL DE APOIO

CURITIBA
2025

MONITORIA DA MATÉRIA DE PROJETO DE SISTEMAS ORIENTADOS A OBJETOS

IMPLEMENTAÇÕES DE LISTAS COM POO

MATERIAL DE APOIO

Material de apoio desenvolvido pelos alunos da monitoria da disciplina de Projeto de Sistemas Orientados a Objetos do terceiro semestre de Engenharia de Software da UniSenaiPR, referente à matéria ministrada pela professora Danielle Forbeci Suzuki.

CURITIBA

2025

SUMÁRIO

1 INTRODUÇÃO.....3

2 ARRAYS E LISTAS.....4

2.1 LISTA (LIST).....7

2.2 LISTA ENCADEADA (LINKEDLIST).....9

2.3 CONJUNTO DE HASH (HASHSET).....12

2.4 MAPA DE HASH (HASHMAP).....14

3 EXERCÍCIOS.....17

3.1 DETALHES E DICAS.....17

3.2 CRITÉRIOS DE AVALIAÇÃO E ENTREGA.....19

3.3 DESAFIO BÔNUS.....19

4 CONCLUSÃO.....21

5 REFERÊNCIAS.....22

1 INTRODUÇÃO

Ao trabalhar com estruturas de dados, compreender o funcionamento interno das listas é muito importante para escrever código mais eficiente e adequado a diferentes cenários. Neste material, exploraremos em detalhes cinco tipos de listas fundamentais: *List*, *LinkedList*, *HashSet* e *HashMap.*

O funcionamento dessas estruturas está diretamente relacionado à forma como os dados são armazenados na memória, impactando a eficiência das operações de inserção, remoção e busca. Enquanto *arrays* oferecem acesso direto aos elementos, sua rigidez em relação ao tamanho pode ser uma limitação. Listas dinâmicas contornam essa restrição, mas podem introduzir novos desafios, como o uso adicional de memória para ponteiros ou a necessidade de cálculos *hash* para localização de elementos.

A análise de desempenho dessas estruturas é realizada por meio da notação *Big-O*, que expressa a complexidade computacional das operações fundamentais. Escolher a estrutura adequada para cada situação requer compreender não apenas a sintaxe de implementação, mas também o comportamento interno dessas listas e suas implicações no desempenho do *software*.

2 *ARRAYS* E LISTAS

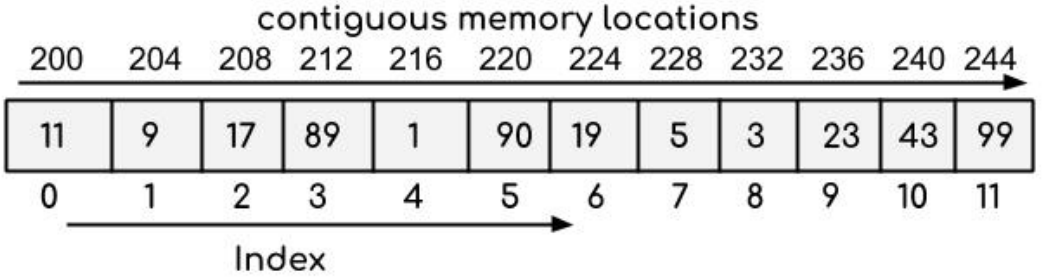
As estruturas de dados são essenciais para o armazenamento e a manipulação eficiente de informações. Entre as mais utilizadas estão as *arrays* e as listas, que possuem características distintas e aplicações específicas. Enquanto *arrays* são estruturas de tamanho fixo que armazenam elementos do mesmo tipo em posições contíguas de memória, listas oferecem maior flexibilidade ao permitir o armazenamento dinâmico de elementos sem a necessidade de definir previamente um tamanho fixo. A escolha entre essas estruturas depende do contexto de uso e das operações predominantes em uma aplicação.

Uma *array* é uma estrutura de dados estática composta por uma sequência ordenada de elementos do mesmo tipo. Os elementos são armazenados em posições seguidas da memória, permitindo acesso direto por meio de um índice. O endereço de qualquer elemento dentro da *array* pode ser calculado a partir do endereço base da estrutura usando a seguinte fórmula:

$$Endereço\ do\ elemento\ i\ =\ Endereço\ base\ +\ (i\ \times\ Tamanho\ do\ tipo\ do\ dado)$$

Essa característica garante que a busca por um elemento específico seja realizada em tempo constante $O(1)$, pois o cálculo do endereço é feito diretamente sem a necessidade de percorrer a estrutura (SINGH, 2024).

Figura 1 - Indexagem de *arrays* e alocação da memória.



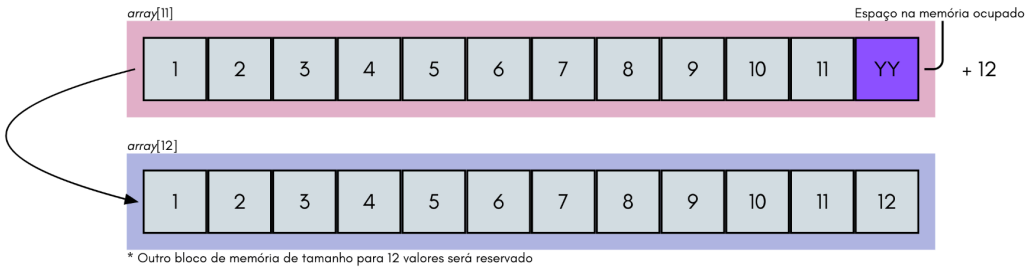
Fonte: SINGH, 2018.

A alocação de memória em *arrays* é realizada de forma contígua, ou seja, os elementos ocupam um bloco único e ininterrupto na memória RAM. Esse comportamento proporciona um acesso eficiente, pois os processadores utilizam a hierarquia de *cache* para

otimizar a recuperação dos dados. No entanto, essa abordagem também traz limitações, pois exige que o sistema operacional encontre um espaço de memória suficientemente grande para armazenar a estrutura sem fragmentação.

Quando uma *array* é criada, um bloco de memória de tamanho fixo é reservado. Caso seja necessário expandir a capacidade da estrutura, um novo bloco de tamanho maior precisa ser alocado e os dados existentes devem ser copiados para essa nova área de memória, o que pode ter um custo computacional elevado de $O(n)$.

Figura 2 -Realocação de memória para redimensionar *arrays*.



Fonte: Ilustração do autor.

As principais vantagens das *arrays* incluem acesso rápido aos elementos, eficiência no uso de *cache* e menor sobrecarga de memória, pois não necessitam de ponteiros adicionais para armazenar referências entre os elementos. No entanto, apresentam desvantagens como tamanho fixo, inserções e remoções custosas e fragmentação de memória, pois exigem um bloco ligado de memória. Essas limitações tornam *arrays* pouco flexíveis quando há necessidade de operações dinâmicas como adição e remoção frequente de elementos.

Diferente das *arrays*, as listas são estruturas dinâmicas que permitem o armazenamento de elementos sem a necessidade de um tamanho fixo predefinido. Elas podem ser implementadas de diferentes formas, sendo as mais comuns as listas encadeadas, listas duplamente encadeadas e listas baseadas em arrays dinâmicos. Além disso, podem ser classificadas em listas estáticas e listas dinâmicas. Listas estáticas são implementadas sobre arrays, mantendo um tamanho fixo, mas com mecanismos internos que permitem a realocação de memória quando necessário, como vetores dinâmicos que dobram de tamanho ao atingir sua capacidade máxima. Já listas dinâmicas são implementadas por meio de nós conectados

por ponteiros, permitindo que novos elementos sejam adicionados ou removidos sem necessidade de realocação de toda a estrutura.

As principais vantagens das listas dinâmicas incluem flexibilidade no tamanho, facilidade de inserção e remoção e menor impacto da fragmentação de memória, pois os elementos são armazenados em nós dispersos pela memória. No entanto, apresentam desvantagens como acesso sequencial aos elementos, maior uso de memória devido ao armazenamento de ponteiros e menor eficiência em *cache*, já que os elementos não estão armazenados de forma contígua. A escolha entre *arrays* e listas depende das necessidades da aplicação e do perfil das operações realizadas. Se o objetivo é acessar elementos de maneira rápida e previsível, *arrays* são mais adequadas devido ao seu acesso indexado em tempo constante. Por outro lado, se a estrutura precisa suportar inserções e remoções frequentes sem alto custo de realocação, listas dinâmicas são uma alternativa mais viável.

A partir desta análise inicial sobre *arrays* e listas, serão abordados os diferentes tipos de listas, tanto estáticas quanto dinâmicas, explorando seu funcionamento interno, alocação de memória, eficiência computacional e principais aplicações. Cada estrutura será analisada detalhadamente, destacando suas vantagens, desvantagens e contextos de uso mais adequados.

2.1 LISTA *(LIST)*

A Lista *(List)* é uma das estruturas de dados mais utilizada na programação orientada a objetos (POO), pois oferece uma maneira flexível e eficiente de armazenar e manipular coleções de elementos. Diferentemente das *arrays* tradicionais, que possuem um tamanho fixo e exigem realocação para expandir sua capacidade, as listas são dinâmicas e podem crescer ou diminuir conforme necessário.

Em uma implementação baseada em *arrays* dinâmicas, a lista mantém um *buffer* interno de elementos, quando a capacidade máxima é atingida e um novo elemento precisa ser adicionado, ocorre um processo de redimensionamento. Esse processo envolve a alocação de um novo espaço de memória com o dobro do tamanho anterior e a cópia dos elementos para essa nova área. Esse crescimento exponencial ajuda a reduzir a quantidade de realocações ao longo do tempo, tornando o desempenho mais previsível.

A lista também armazena internamente um contador de elementos, que permite determinar rapidamente o número de itens contidos na estrutura sem precisar percorrer toda a coleção. Sua eficiência depende das operações realizadas sobre ela, abaixo estão as principais operações e suas respectivas complexidades:

- **Acesso a um elemento por índice:** $O(1)$ - Como a lista utiliza uma *array* interna, o acesso a um elemento específico por índice é feito de maneira direta, sem necessidade de percorrer a estrutura.
- **Inserção no final:** $O(1)$ (amortizado) – Se houver espaço disponível no *buffer*, a inserção ocorre em tempo constante. Caso contrário, quando o *buffer* precisa ser redimensionado, o tempo de inserção é $O(n)$ devido à cópia dos elementos para uma nova *array*, mas, ao longo do tempo, essa operação tem um custo amortizado de $O(1)$.
- **Inserção em uma posição intermediária:** $O(n)$ – Todos os elementos subsequentes precisam ser deslocados para abrir espaço para o novo elemento.
- **Remoção no final:** $O(1)$ – O último elemento pode ser removido sem a necessidade de deslocamentos.
- **Remoção em uma posição intermediária:** $O(n)$ – Todos os elementos posteriores precisam ser deslocados para preencher o espaço deixado pelo elemento removido.
- **Busca por um elemento (pesquisa linear):** $O(n)$ – No pior caso, é necessário percorrer toda a lista para encontrar um elemento específico.

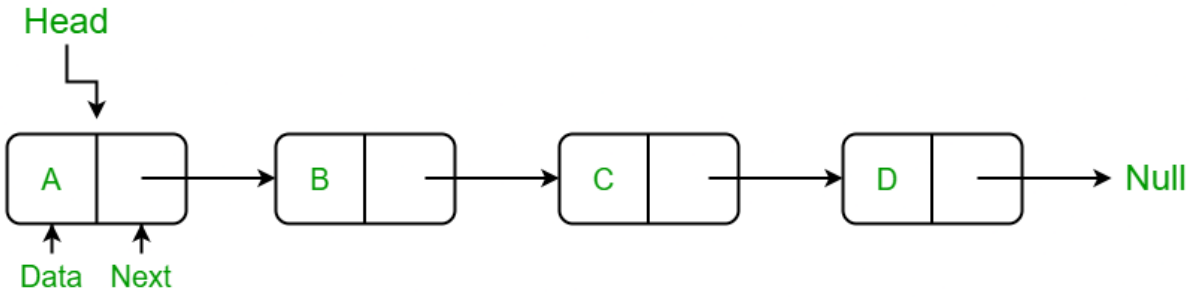
As listas são muito utilizadas devido à sua flexibilidade e eficiência para várias operações. No entanto, sua escolha deve levar em consideração o tipo de operação mais frequente, a lista é adequada para cenários onde há necessidade de acesso rápido por índice e crescimento dinâmico da estrutura.

No *.NET,* existe uma implementação pronta de lista que segue esse modelo, denominada *List<T>*. Essa implementação utiliza *arrays* dinâmicas internamente e oferece diversas funcionalidades otimizadas para operações comuns em coleções.

2.2 LISTA ENCADEADA *(LINKEDLIST)*

A Lista Encadeada *(Linked List)* é uma estrutura de dados fundamental na programação orientada a objetos (POO), que oferece uma maneira eficiente de armazenar e manipular coleções de elementos de forma dinâmica. Diferente da lista baseada em *array*, onde os elementos são armazenados em posições contíguas de memória, uma lista encadeada armazena os elementos de forma não contígua. Cada elemento da lista é um nó que contém, no mínimo, dois campos: um valor (o dado propriamente dito) e uma referência (ou ponteiro) para o próximo nó da lista.

Figura 3 - Ilustração de Lista Encadeada.



Fonte: GEEKSFORGEEKS, 2023.

Em uma lista encadeada, cada nó é alocado dinamicamente e está "ligado" ao próximo nó por meio de uma referência, o que permite que a lista cresça e encolha conforme necessário, sem a necessidade de redimensionar a estrutura como acontece em *arrays* dinâmicas. Além disso, cada nó pode estar em qualquer posição da memória, o que elimina o problema de fragmentação de memória que ocorre com *arrays*.

A lista encadeada é composta por um primeiro nó, chamado de cabeça (head), que serve como ponto de entrada para a lista. A partir daí, cada nó contém um ponteiro para o próximo nó, formando uma cadeia de elementos. O último nó, por sua vez, possui um ponteiro nulo (ou referência nula), indicando o final da lista.

A complexidade das operações depende da posição onde elas ocorrem, já que para realizar muitas dessas operações, é necessário percorrer a lista até o nó de interesse.

- **Acesso a um elemento por índice:** $O(n)$ – Ao contrário das listas baseadas em *arrays*, onde o acesso a um elemento por índice é feito diretamente, nas listas

encadeadas é necessário percorrer a lista a partir do primeiro nó até o índice desejado, o que resulta em uma complexidade linear.

- **Inserção no início:** $O(1)$ – A inserção de um elemento no início da lista é uma operação de tempo constante, pois basta alterar o ponteiro da cabeça para apontar para o novo nó, e o novo nó passa a ser a cabeça da lista.
- **Inserção no final:** $O(n)$ – Se a lista não tiver uma referência do último nó, é necessário percorrer toda a lista até o último nó para inserir um elemento no final. Após encontrar o último nó, a inserção é feita alterando o ponteiro do último nó para apontar para o novo nó.
- **Inserção em uma posição intermediária:** $O(n)$ – Inserir um elemento em uma posição intermediária requer a navegação até o nó na posição desejada e o ajuste de ponteiros para inserir o novo nó. Essa operação também possui complexidade linear.
- **Remoção no início:** $O(1)$ – Para remover um elemento no início da lista, basta alterar a cabeça para apontar para o próximo nó, descartando o primeiro nó da lista.
- **Remoção no final:** $O(n)$ – A remoção do último elemento envolve a busca pelo penúltimo nó (que deve ter seu ponteiro ajustado para nulo), o que torna essa operação de tempo linear.
- **Busca por um elemento:** $O(n)$ – Para encontrar um elemento em uma lista encadeada, é necessário percorrer a lista desde o início até encontrar o elemento desejado ou até chegar ao final da lista, o que resulta em uma busca linear.

A principal vantagem das listas encadeadas é sua flexibilidade, elas permitem inserções e remoções eficientes, especialmente no início da lista. Além disso, como os elementos não precisam ser armazenados de maneira contígua na memória, as listas encadeadas não enfrentam problemas de fragmentação de memória como as *arrays*.

Porém, as listas encadeadas têm algumas desvantagens. Como cada nó armazena um ponteiro, elas requerem mais memória do que uma lista baseada em *array*. Além disso, o acesso aos elementos é mais lento, já que é necessário percorrer a lista sequencialmente.

As listas encadeadas são mais adequadas em cenários onde as operações de inserção e remoção, especialmente no início da lista, são mais frequentes do que as de acesso a elementos por índice. Elas são ideais quando a estrutura precisa crescer ou diminuir dinamicamente sem a sobrecarga de redimensionamento de *arrays*. Também são úteis quando é necessário garantir que a memória seja alocada de maneira flexível e sem fragmentação.

No *.NET*, existe uma implementação pronta de lista encadeada denominada *LinkedList<T>*. Essa implementação utiliza nós duplamente encadeados internamente (apontam para o próximo nó e para o anterior), permitindo inserções e remoções eficientes em qualquer posição da lista. Além disso, oferece métodos otimizados para manipulação de elementos, sendo uma alternativa eficiente para cenários onde operações frequentes de inserção e remoção são necessárias.

2.3 CONJUNTO DE *HASH (HASHSET)*

O Conjunto de *Hash (HashSet)* é uma estrutura de dados utilizada para armazenar conjuntos de elementos únicos, sem ordem definida, e com operações eficientes de inserção, remoção e busca. Diferente de listas ou *arrays*, que permitem elementos duplicados e acessos por índice, o *HashSet* utiliza uma tabela de dispersão *(hash table)* para organizar os dados, garantindo que cada elemento seja armazenado de forma única e permitindo operações de busca e inserção em tempo médio $O(1)$.

A base do *HashSet* é uma função *hash*, que transforma um elemento em um valor numérico *(hashcode)*, utilizado para determinar em qual posição da estrutura esse elemento será armazenado. Essa abordagem minimiza a necessidade de percorrer a estrutura inteira para encontrar um item, tornando a busca extremamente eficiente.

No entanto, em alguns casos, diferentes elementos podem gerar o mesmo valor de *hash*, causando o que é chamado de colisão. Para resolver colisões, a estrutura pode utilizar técnicas como encadeamento (listas ligadas em cada posição da tabela) ou endereçamento aberto (realocação do elemento em outra posição disponível).

A eficiência do *HashSet* está diretamente ligada à qualidade da função *hash* e à carga da tabela (razão entre o número de elementos armazenados e o tamanho da tabela). Se houver muitas colisões, a busca e a inserção podem se degradar para $O(n)$ no pior caso, mas, em condições normais, essas operações são executadas em tempo $O(1)$.

- **Inserção:** $O(1)$ (médio) – O elemento é inserido diretamente na posição determinada pela função *hash*. Se houver colisão, o tempo pode aumentar.
- **Remoção:** $O(1)$ (médio) – Sem necessidade de deslocamento, apenas a posição da tabela *hash* é liberada.
- **Busca por um elemento:** $O(1)$ (médio) – A posição do elemento é determinada diretamente pela função hash. No pior caso, pode ser $O(n)$ se houver muitas colisões.
- **Iteração sobre os elementos:** $O(n)$ – Como a estrutura não mantém uma ordem específica, a iteração percorre toda a tabela.

O *HashSet* é ideal para situações onde a singularidade dos elementos é crucial e onde a ordem de inserção não é importante, como no armazenamento de chaves únicas, remoção rápida de duplicatas ou otimização de buscas.

No *.NET*, existe uma implementação otimizada dessa estrutura chamada *HashSet<T>*, que utiliza uma tabela de dispersão para armazenar elementos únicos e oferece métodos eficientes para operações comuns de conjuntos, como união, interseção e diferença.

2.4 MAPA DE *HASH (HASHMAP)*

O Mapa de Hash *(HashMap)* é uma estrutura de dados que implementa um dicionário chave-valor, permitindo armazenar, recuperar e manipular pares de dados de forma eficiente. Diferente de listas ou *arrays*, que armazenam elementos individualmente, o *HashMap* permite a associação entre uma chave única e um valor correspondente, possibilitando consultas rápidas.

A base do *HashMap* é a tabela de dispersão *(hash table)*, onde cada chave passa por uma função *hash* para determinar a posição onde seu valor correspondente será armazenado. Essa abordagem garante que a busca, inserção e remoção sejam realizadas, em média, em tempo $O(1)$.

Entretanto, como em qualquer estrutura baseada em funções *hash*, pode ocorrer uma colisão, que acontece quando duas chaves diferentes geram o mesmo valor de *hash* e tentam ocupar a mesma posição na tabela. Para resolver esse problema, o *HashMap* pode utilizar técnicas como:

- Encadeamento *(Separate Chaining)* – Cada posição da tabela armazena uma lista ligada para acomodar múltiplos valores.
- Endereçamento aberto *(Open Addressing)* – Quando ocorre uma colisão, o *HashMap* busca outra posição disponível dentro da tabela para armazenar o elemento.

A eficiência do *HashMap* depende da função *hash* utilizada e do fator de carga da tabela *(load factor),* que define a relação entre o número de elementos e o tamanho da tabela. Quando esse fator ultrapassa um determinado limite, a estrutura pode redimensionar sua capacidade, realocando os elementos para uma nova tabela de tamanho maior.

- **Inserção:** $O(1)$ (médio) – A chave é processada pela função *hash* e seu valor é armazenado na posição correspondente. Em caso de colisão, a operação pode levar mais tempo.
- **Remoção:** $O(1)$ (médio) – A chave é processada e o valor correspondente é removido da tabela. No pior caso, pode ser $O(n)$ se houver muitas colisões.
- **Busca por uma chave:** $O(1)$ (médio) – A função *hash* permite localizar a chave diretamente. No pior caso, pode ser $O(n)$ devido a colisões.
- **Iteração sobre os pares chave-valor:** $O(n)$ – A tabela de dispersão precisa ser percorrida integralmente para acessar todos os elementos.

O *HashMap* é ideal para aplicações que exigem armazenamento e recuperação eficiente de informações associadas a chaves, como tabelas de banco de dados, cache de dados e indexação de documentos.

No *.NET*, existe uma implementação otimizada dessa estrutura chamada *Dictionary<TKey, TValue>*, que utiliza uma tabela de dispersão para mapear chaves a valores e oferece métodos eficientes para manipulação de dados.

A escolha entre *arrays* e listas dinâmicas depende diretamente do contexto de uso e das operações mais frequentes na aplicação. *Arrays* são ideais para cenários onde o tamanho da coleção é conhecido previamente e o acesso é realizado principalmente por índices, devido à sua eficiência em tempo constante $O(1)$ para leituras. No entanto, sua rigidez em relação ao tamanho pode ser um obstáculo, especialmente quando é necessário redimensionar a estrutura.

Por outro lado, listas encadeadas oferecem flexibilidade ao permitir inserções e remoções dinâmicas sem a necessidade de deslocar elementos, tornando-as mais eficientes para modificações frequentes. Entretanto, o custo de armazenamento adicional com ponteiros e o acesso sequencial tornam a busca por elementos menos eficiente.

Outras estruturas como *HashSet* e *HashMap* são mais adequadas para casos onde a ênfase está na busca rápida de elementos, utilizando funções de *hash* para localizar dados em tempo próximo de $O(1)$, mas sem garantir ordenação.

A tabela a seguir resume as principais características das estruturas analisadas, considerando suas operações principais e suas respectivas complexidades em notação *Big-O*:

Estrutura de Dados	Acesso por índice	Inserção / Remoção no Final	Inserção / Remoção no Início	Inserção / Remoção no Meio	Busca
Array	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
List	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Linked List	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
HashSet	-	$O(1)$	$O(1)$	$O(1)$	$O(1)$
HashMap	-	$O(1)$	$O(1)$	$O(1)$	$O(1)$

3 EXERCÍCIOS

Agora que foi esclarecido o funcionamento de diferentes tipos de listas, implemente quatro estruturas de dados utilizando Programação Orientada a Objetos (POO): *List*, *LinkedList*, *HashSet* e *HashMap*. Cada uma dessas classes deve ser implementada do zero, sem utilizar as implementações nativas do C#, respeitando os conceitos de encapsulamento, polimorfismo e boas práticas de código.

Deve criar quatro classes correspondentes às estruturas mencionadas, garantindo que cada uma delas possua os seguintes métodos e propriedades:

- Construtores: Um construtor vazio e um que aceite valores iniciais.
- Métodos de Manipulação:
    - Append(item): Adiciona um elemento ao final da estrutura.
    - Prepend(item): Adiciona um elemento no início.
    - Insert(index, item): Insere um elemento em uma posição específica.
    - Remove(item): Remove um elemento da estrutura.
    - RemoveAt(index): Remove um elemento em um índice específico (quando aplicável).
    - Contains(item): Verifica se o elemento existe na estrutura.
    - IndexOf(item): Retorna o índice do elemento, se existir (quando aplicável).
    - Clear(): Remove todos os elementos da estrutura.
- Atributos essenciais:
    - First: Retorna o primeiro elemento da estrutura (quando aplicável).
    - Last: Retorna o último elemento da estrutura (quando aplicável).
    - Count: Retorna a quantidade de elementos armazenados.
- Sobrescrita do método ToString() para uma lista tradicional.

Para as classes de lista e lista encadeada implementar um método para retornar o elemento pelo index, e no *HashMap* um para retornar por chave.

3.1 DETALHES E DICAS

A *List* (Lista Dinâmica) deve funcionar como uma *array* dinâmica, expandindo-se conforme novos elementos são adicionados. Para isso, utilize uma *array* interna que seja redimensionada automaticamente quando atingir sua capacidade máxima, dobrando de

tamanho para otimizar realocações. A remoção de elementos deve deslocar os itens seguintes para evitar buracos na estrutura. O método *Insert(index, item)* deve mover os elementos necessários para abrir espaço no índice desejado. Para melhorar o desempenho, é importante controlar a quantidade real de elementos com um atributo *Count*, evitando operações desnecessárias.

Já a *LinkedList* (Lista Encadeada) difere da lista dinâmica por armazenar seus elementos em nós independentes, que se conectam entre si. Para isso, é necessário criar duas classes: *Node<T>*, que representa cada nó e contém referências para o próximo *(Next)* e, no caso de uma lista duplamente encadeada, também para o anterior *(Previous)*; e *LinkedList<T>*, que gerencia os nós e mantém referências para o primeiro *(First)* e último *(Last)* elemento, além do total de itens *(Count)*. Métodos como *Append* e Prepend devem apenas ajustar os ponteiros, tornando essas operações rápidas ($O(1)$), enquanto buscas *(IndexOf, Contains)* exigem percorrer a lista sequencialmente ($O(n)$). Ao inserir *(Insert)* ou remover *(Remove, RemoveAt)* um elemento, os ponteiros dos nós vizinhos devem ser corretamente ajustados para evitar a quebra da cadeia.

O *HashSet* (Conjunto Baseado em Hash) deve garantir que não existam elementos repetidos, utilizando uma tabela *hash* para armazenar os dados de forma eficiente. Para isso, utilize uma *array* de listas *(buckets)*, onde a posição de armazenamento é determinada pelo *hash* do elemento, caso ocorra colisão — quando dois elementos caem no mesmo *bucket* — implemente um sistema de encadeamento, onde múltiplos valores podem ser armazenados dentro de uma mesma posição. O método *Contains* deve calcular o *hash* do elemento, acessar o índice correto e verificar se ele já está presente. Para manter o desempenho ideal, redimensione a tabela quando uma determinada taxa de ocupação for atingida, garantindo um balanceamento adequado.

Por fim, o *HashMap* (Dicionário Chave-Valor) deve seguir o mesmo princípio do *HashSet*, mas permitindo armazenar pares chave-valor. A inserção *(Insert ou Add)* deve calcular a posição da chave e armazenar o par correspondente. Em caso de colisão, use listas encadeadas dentro dos *buckets* ou um sistema de realocação (endereçamento aberto). O método *GetValue(key)* deve buscar a chave correta e retornar o valor associado. Assim como no *HashSet*, a estrutura deve ser redimensionada automaticamente quando um limite de ocupação for atingido, garantindo eficiência no armazenamento e recuperação dos dados.

Todas as implementações devem seguir os princípios da programação orientada a objetos, encapsulando os detalhes internos de cada estrutura.

3.2 CRITÉRIOS DE AVALIAÇÃO E ENTREGA

Seu código será avaliado com base nos seguintes critérios:

- Aplicação correta dos princípios de POO.
- Implementação funcional e eficiente dos métodos requeridos.
- Clareza e organização do código.

A entrega do trabalho deve ser feita de uma das seguintes formas:

- Envio do *link* para o repositório no GitHub contendo o projeto.
- Envio de um arquivo *.zip* com o projeto.

3.3 DESAFIO BÔNUS

Com as quatro estruturas principais implementadas, o desafio é criar uma estrutura híbrida entre uma *LinkedList* e uma *array*, combinando as vantagens de ambas. Nesta estrutura, cada nó da lista encadeada conterá um *array* de tamanho fixo/máximo, armazenando múltiplos elementos dentro do mesmo nó antes de encadear para o próximo.

- Cada nó conterá uma *array* de tamanho fixo (por exemplo, capacidade = 10).
- Quando um nó atingir sua capacidade máxima, um novo nó será criado e encadeado ao final da estrutura.
- A estrutura deve funcionar de forma transparente para o usuário, parecendo uma lista única e contínua.
- O acesso a elementos por índice deve ser suportado, convertendo o índice para o nó correto e a posição dentro do array.
- Deve implementar os mesmos métodos básicos e atributos essenciais das listas anteriores.

A implementação dessa estrutura híbrida traz diversos benefícios. Primeiramente, ela otimiza o uso de memória ao evitar a alocação individual de pequenos nós, reduzindo a sobrecarga de ponteiros e aproveitando melhor o espaço disponível. Além disso, melhora o desempenho, pois operações sequenciais dentro de um mesmo nó são mais rápidas do que acessar múltiplos nós individuais dispersos na memória. Esse modelo também introduz o conceito de estruturas híbridas, que combinam diferentes abordagens para maximizar

eficiência e flexibilidade, sendo amplamente utilizado em cenários que exigem um equilíbrio entre velocidade de acesso e economia de recursos.

Para implementá-la de maneira eficiente, algumas estratégias podem ser adotadas. O acesso a elementos por índice pode ser otimizado calculando diretamente em qual nó está localizado o índice, utilizando a divisão pelo tamanho máximo da *array* interna de cada nó. Na inserção e remoção de elementos, pode ser necessário redistribuir os valores entre nós vizinhos, garantindo que a estrutura permaneça balanceada e sem fragmentação excessiva. Além disso, a operação de remoção deve ser projetada para fundir nós caso um deles fique parcialmente vazio, evitando desperdício de memória e mantendo a lista eficiente ao longo do tempo.

4 CONCLUSÃO

Este material de apoio apresentou os fundamentos e as implementações de diversas estruturas de dados essenciais na Programação Orientada a Objetos. A partir da análise das *arrays* e das listas, ficou evidente como o armazenamento contíguo em *arrays* permite acesso rápido por índice, enquanto as listas dinâmicas oferecem maior flexibilidade para operações de inserção e remoção.

A comparação entre essas estruturas concluiu que a escolha da implementação ideal depende diretamente do contexto e dos requisitos específicos da aplicação, considerando fatores como a frequência de inserções, remoções e buscas, bem como a utilização de memória. Essa compreensão é fundamental para o desenvolvimento de sistemas eficientes e escaláveis.

Por fim, os exercícios propostos incentivam a aplicação prática dos conceitos abordados, estimulando aprimoramento na implementação de estruturas de dados em C#. Dessa forma, espera-se que este material contribua significativamente para o aprimoramento dos conhecimentos em Programação Orientada a Objetos e estrutura de dados.

5 REFERÊNCIAS

Applications of linked list data structure. **GeeksforGeeks**, 2023. Disponível em: https://www.geeksforgeeks.org/applications-of-linked-list-data-structure/. Acesso em: 31 mar. 2025.

Array Data Structure Guide. **GeeksforGeeks**, 2025. Disponível em: https://www.geeksforgeeks.org/array-data-structure-guide/. Acesso em: 31 mar. 2025.

Dictionary<TKey,TValue> Classe. **Microsoft Learn**,. Disponível em: https://learn.microsoft.com/pt-br/dotnet/api/system.collections.generic.dictionary-2?view=net-9.0. Acesso em: 01 abr. 2025.

HashSet<T> Classe. **Microsoft Learn**,. Disponível em: https://learn.microsoft.com/pt-br/dotnet/api/system.collections.generic.hashset-1?view=net-8.0. Acesso em: 01 abr. 2025.

HENRIQUE, João . POO: o que é programação orientada a objetos?. **Alura**, 2023. Disponível em: https://www.alura.com.br/artigos/poo-programacao-orientada-a-objetos?srsltid=AfmBOorFWUWbaFju-E4Ivhw11NongXHuXqwB_OLOs8ZW656gBuDcBxcV. Acesso em: 01 abr. 2025.

LinkedList<T> Classe. **Microsoft Learn**,. Disponível em: https://learn.microsoft.com/pt-br/dotnet/api/system.collections.generic.linkedlist-1?view=net-8.0. Acesso em: 01 abr. 2025.

List<T> Classe. **Microsoft Learn**. Disponível em: https://learn.microsoft.com/pt-br/dotnet/api/system.collections.generic.list-1?view=net-8.0. Acesso em: 31 mar. 2025.

SINGH, Chaitanya. Data Structure – Array. **BeginnersBook**, 2018. Disponível em: https://beginnersbook.com/2018/10/data-structure-array/. Acesso em: 31 mar. 2025.

TSIBULEVSKIY, Oleg. Arrays, behind the scenes: What is an array? How is an array stored in memory? Review Push and Pop operations.. **Medium**, 2020. Disponível em: https://medium.com/justeattakeaway-tech/array-behind-the-scene-e098f42d4623. Acesso em: 31 mar. 2025.

What is Array?. **GeeksforGeeks**, 2024. Disponível em: https://www.geeksforgeeks.org/what-is-array/?ref=lbp. Acesso em: 31 mar. 2025.