UNIVERSIDADE SALVADOR (UNIFACS)

CURSO DE CIÊNCIA DA COMPUTAÇÃO

ESTRUTURA DE DADOS

PILHA APLICADA AO JOGO DE CLASSIFICAÇÃO POR BOLAS COLORIDAS

HENRIQUE CAVALCANTI ROCHA - 12722117519

MARCOS VINICIUS LIMA RIBEIRO - 12723116626

PEDRO MARTINS CAIRES - 12722124034

RAFAEL RODRIGUES FIGUEIREDO - 12722130532

Salvador - BA

2025

1. Resumo

Este trabalho tem como objetivo apresentar o desenvolvimento de um jogo computacional fundamentado na estrutura de dados do tipo pilha, aplicado como ferramenta didática para o ensino de conceitos de Estrutura de Dados. O jogo, denominado *StackColor*, consiste na simulação de sete pilhas (ou tubos), cada uma com capacidade máxima de sete elementos, representando bolas de diferentes cores, sendo que uma das pilhas inicia-se completamente vazia para possibilitar a reorganização das demais.

A estrutura de pilha adotada no jogo segue o princípio LIFO (Last In, First Out), em que o último elemento inserido é o primeiro a ser removido, o que influencia diretamente na lógica das movimentações. A mecânica do jogo exige que o jogador reorganize as bolas de modo que cada pilha contenha apenas bolas de uma única cor, obedecendo a regras específicas, como a movimentação de apenas uma bola por vez e a impossibilidade de empilhar bolas de cores diferentes em uma mesma pilha.

A implementação do jogo foi realizada utilizando a linguagem Java, explorando as operações fundamentais de pilhas, como stack (empilhar), unstack (desempilhar) e showTop (mostrar o topo). Além do desenvolvimento do sistema, este artigo discute os algoritmos empregados na manipulação e validação das jogadas, a aplicação prática da estrutura pilha no contexto do jogo, bem como os benefícios pedagógicos da abordagem lúdica no processo de aprendizagem de algoritmos e estruturas de dados. São também sugeridas possíveis extensões e melhorias na aplicação, visando maior complexidade, interação e aplicabilidade em ambientes educacionais.

2. Introdução

O estudo de estruturas de dados é um dos pilares fundamentais na formação de profissionais da área de Ciência da Computação. Compreender como organizar, manipular e acessar dados de forma eficiente é essencial para o desenvolvimento de algoritmos robustos e otimizados. Dentre as estruturas básicas, a pilha destaca-se por sua simplicidade e ampla aplicabilidade em contextos como reversão de operações, análise de expressões, gerenciamento de memória e navegação em interfaces.

Apesar da importância conceitual, muitos estudantes enfrentam dificuldades na internalização dos princípios que regem o funcionamento das pilha. Nesse cenário, o uso de ferramentas lúdicas e interativas tem se mostrado eficaz no processo de ensino, permitindo que os alunos visualizem de forma concreta o impacto das operações realizadas sobre esse tipo abstrato de dados.

Com esse propósito, foi desenvolvido o jogo *StackColor*, que simula o empilhamento e a organização de bolas coloridas em tubos, representando pilhas de dados. O desafio proposto ao jogador é reorganizar essas bolas de modo que cada pilha contenha exclusivamente bolas de uma única cor, respeitando regras que impõem restrições às movimentações, de forma a refletir as limitações reais das pilhas. A implementação do jogo utiliza a linguagem Java e foi pensada como ferramenta didática complementar, visando reforçar, por meio da prática, os principais conceitos teóricos relacionados a essa estrutura.

Este artigo descreve detalhadamente o funcionamento do jogo, os algoritmos utilizados, as operações de pilha envolvidas, e discute os benefícios pedagógicos da aplicação, além de propor possíveis extensões futuras para ampliar sua aplicabilidade e complexidade.

3. O que é uma Pilha?

A pilha é um tipo abstrato de dados linear, dinâmico ou estático, amplamente utilizado na computação. Seu funcionamento é baseado no princípio LIFO (Last In, First Out), que significa “o último a entrar é o primeiro a sair”. Esse comportamento implica que o último elemento inserido na pilha será o primeiro a ser removido, o que torna a pilha ideal para situações em que é necessário controlar a ordem reversa das operações.

Conceitualmente, uma pilha pode ser comparada a uma pilha de pratos: um prato só pode ser retirado se estiver no topo da pilha, ou seja, não é possível acessar diretamente os elementos que estão abaixo sem antes remover os que estão acima. Assim, o acesso aos dados é restrito ao topo da pilha, permitindo operações específicas como inserção e remoção.

Pilhas são frequentemente utilizadas em problemas computacionais como controle de chamadas de funções (pilha de execução), verificação de sintaxe em linguagens de programação, e conversão de expressões aritméticas. Devido à sua simplicidade e utilidade, elas constituem uma das primeiras estruturas estudadas em cursos de algoritmos e estruturas de dados.

No contexto do jogo *StackColor*, a pilha é representada por tubos que armazenam bolas coloridas. Cada tubo permite empilhar e desempilhar bolas seguindo rigorosamente o comportamento LIFO, o que oferece ao jogador um ambiente prático para compreender como as pilhas operam em cenários reais e restritos.

4. Principais Operações de uma Pilha

Um tipo abstrato de dados pilha deve possuir alguns métodos básicos, são eles:

- Insere() - Inserir um objeto no topo da pilha. Possui como entrada um objeto e não possui nenhuma saída;
- Remove() - Retirar um objeto do topo da pilha. Possui como saída um objeto e não possui nenhuma entrada.

Esses métodos básicos são responsáveis por manipular diretamente o elemento no topo da pilha e regem o comportamento principal dessa estrutura. Ambos seguem o princípio LIFO (Last In, First Out) e são essenciais para a lógica de funcionamento da pilha.

Além disso, a pilha inclui métodos auxiliares que facilitam a verificação de estado e consulta sem modificação, como:

- Tamanho() - Retorna o número de objetos da pilha. Possui como saída um número inteiro e nenhuma entrada;
- Vazio() - Retorna um valor booleano indicando se a pilha está vazia ou não. Possui como saída um valor booleano e nenhuma entrada;
- Cheio() - Retorna um valor booleano indicando se a pilha está cheia ou não. Possui como saída um valor booleano e nenhuma entrada;
- Topo() - Retorna o objeto que está no topo da pilha, sem retirá-lo. Possui como saída um objeto e nenhuma entrada.

Esses métodos auxiliares não alteram a pilha, mas fornecem informações importantes sobre seu estado. Eles permitem validar ações, prevenir erros de execução e oferecer maior controle sobre o comportamento da estrutura. Em aplicações como jogos ou algoritmos, esses métodos são indispensáveis para garantir a integridade e previsibilidade do sistema.

5. Regras do Jogo: Classificação por Bolas Coloridas

O jogo *StackColor* tem como objetivo organizar bolas coloridas em pilhas de forma que cada pilha contenha bolas de uma única cor. O tabuleiro é composto por sete pilhas, sendo que uma delas inicia vazia para possibilitar as movimentações necessárias durante a partida. Cada pilha pode conter, no máximo, sete bolas, respeitando os limites definidos pela estrutura de dados.

As movimentações devem ser realizadas uma bola por vez, sempre a partir do topo de uma pilha. Não é permitido empilhar bolas de cores diferentes na mesma pilha, o que exige planejamento estratégico por parte do jogador. O jogo simula fielmente o comportamento de uma pilha, obedecendo ao princípio LIFO (Last In, First Out), ou seja, apenas a última bola inserida pode ser removida ou movida.

A vitória é alcançada quando todas as pilhas, com exceção da pilha vazia, estiverem organizadas com bolas da mesma cor do topo até a base. A simplicidade das regras permite fácil assimilação, ao mesmo tempo em que promove o raciocínio lógico, tornando o *StackColor* uma ferramenta educativa eficaz para o aprendizado de estruturas de dados.

6. A Estrutura Pilha no Jogo

Na aplicação desenvolvida, a pilha é implementada por meio da classe Pilha, que utiliza uma estrutura encadeada de objetos do tipo Ball. Cada instância da classe Ball representa uma bola colorida e contém dois atributos principais: um ponteiro (next) que referencia a próxima bola abaixo dela na pilha, e uma variável color, que define a cor da bola com base no enum ColorBall. O topo da pilha é representado por um ponteiro chamado “top”, que sempre aponta para a bola que foi inserida mais recentemente. O atributo “currentPosition” é utilizado para controlar a quantidade atual de elementos empilhados, respeitando o limite máximo de sete bolas por pilha.

A lógica de encadeamento é fundamental para o funcionamento da estrutura, permitindo que as bolas sejam ligadas entre si dinamicamente, sem a necessidade de utilizar estruturas de arrays fixos. Cada nova bola empilhada recebe uma

referência para o elemento que estava anteriormente no topo, formando uma cadeia de objetos conectados. O último elemento da pilha (ou seja, aquele mais ao fundo) é identificado por ter seu ponteiro “next” definido como null, indicando que não há nenhum elemento abaixo dele. Essa organização permite operações eficientes de empilhamento e desempilhamento, além de facilitar a verificação de estados como pilha cheia, vazia ou o acesso ao elemento do topo.

Os métodos stack() e unstack() da classe Pilha implementam, respectivamente, as operações de inserção e remoção de bolas. A inserção (stack) adiciona uma nova bola no topo da pilha, desde que ela não esteja cheia. Caso a pilha esteja vazia, a bola se torna o topo diretamente; caso contrário, a nova bola aponta para a bola que estava no topo anterior e se torna a bola do topo. Já a remoção (unstack) retira a bola do topo da pilha, atualizando o topo para a próxima bola da sequência. Essas operações são a base da mecânica de movimentação das bolas no jogo.

Já os métodos auxiliares fornecem suporte às operações principais de inserção e remoção, permitindo maior controle e interatividade durante o jogo. O método isEmpty() verifica se a pilha está vazia, retornando verdadeiro quando o ponteiro do topo está nulo, ou seja, quando não há nenhuma bola empilhada. Já o método isFull() serve para identificar se a pilha atingiu seu limite máximo de sete bolas, com base na contagem da variável currentPosition. Esses dois métodos são fundamentais para garantir que a lógica de empilhamento e desempilhamento respeite as regras do jogo, evitando inserções além da capacidade ou retiradas de uma pilha vazia.

O método showTop() retorna a bola que está no topo da pilha, sem removê-la, permitindo a visualização do último elemento inserido. Já o método showStack() percorre toda a pilha a partir do topo, exibindo no console as cores de todas as bolas empilhadas. Essa funcionalidade é útil para acompanhar o estado atual do jogo e tomar decisões estratégicas ao mover bolas entre as pilhas.

O método clear tem como função esvaziar completamente a pilha. Ele percorre todos os elementos encadeados e redefine o ponteiro do topo, além de zerar a contagem de elementos. É um recurso útil para reiniciar a pilha ou limpar

uma jogada. Já o método winnerStack() verifica se a pilha está completamente preenchida com bolas da mesma cor, retornando verdadeiro apenas se todas as sete posições forem ocupadas por bolas iguais. Essa verificação é essencial para determinar se o jogador alcançou a condição de vitória.

Por fim, o método getColorIndex() permite acessar a cor de uma bola em uma posição específica da pilha. Ele percorre a pilha a partir do topo até o índice informado e retorna o nome da cor correspondente. Caso a posição seja inválida ou não haja bola naquela posição, o método retorna um caractere padrão, representando a ausência de uma bola. Esse recurso pode ser útil em representações visuais da pilha ou no controle de lógica do jogo.

7. Algoritmos Relacionados

7.1. Inicialização das Pilhas

O jogo tem início com a execução do método initStacks(), responsável pela criação e preenchimento das pilhas que compõem o cenário principal. Ao todo, são sete pilhas instanciadas, por meio do loop-for que percorre o array pilhas, atribuindo a cada posição uma nova instância da classe Pilha. No entanto, apenas seis dessas pilhas são preenchidas no início; a sétima permanece vazia propositalmente, servindo como espaço auxiliar estratégico durante a jogabilidade.

Antes de iniciar o preenchimento, os contadores de bolas de cada cor são zerados para garantir uma contagem correta e limpa. Em seguida, o algoritmo entra em um laço que se repete seis vezes, correspondente às seis pilhas que devem ser preenchidas com bolas coloridas.

Para cada uma dessas seis pilhas, utiliza-se um laço do-while que garante que a pilha atual seja válida antes de prosseguir. Dentro deste laço, a pilha é limpa com o método clear() para evitar resíduos de execuções anteriores, e em seguida é preenchida com sete bolas, cada uma criada com uma cor gerada aleatoriamente por meio do método randomColor(). Esse método retorna um número inteiro entre 1 e 6, representando as seis cores possíveis no jogo.

Após o preenchimento da pilha, são feitas duas verificações fundamentais. A primeira assegura que a cor do topo da pilha atual (currentColor), obtida com pilhas[i].showTop().getColor(), não seja igual à do topo da pilha anterior (lastColor). Essa condição é ignorada apenas na primeira pilha. A segunda verificação, mais crítica, ocorre por meio da chamada ao método numberColors(pilhas[i]).

O método numberColors() tem como objetivo validar se a inclusão das bolas daquela pilha não ultrapassará o limite de sete bolas por cor no jogo todo. Para isso, ele simula a contagem das bolas daquela pilha sobre os contadores atuais. Caso qualquer cor exceda o limite de sete instâncias, o método retorna false, invalidando a pilha, que será descartada e sorteada novamente. Essa lógica evita configurações iniciais impossíveis ou desequilibradas.

Somente quando ambas as condições são satisfeitas — topo diferente da pilha anterior (quando necessário) e quantidade de cores válida — a pilha é aceita e seus dados são oficializados. Nesse momento, o método updateColorCounts(pilhas[i]) é invocado para atualizar os contadores globais de cores com base nas bolas daquela pilha. Esse processo se repete até que todas as seis pilhas estejam preenchidas de forma válida, equilibrada e aleatória.

Assim, o método initStacks() garante que a partida comece com uma distribuição justa das bolas coloridas, sem repetições indesejadas no topo das pilhas iniciais e respeitando as limitações de quantidade por cor. Essa lógica é essencial para assegurar um ponto de partida bem estruturado e jogável, evitando cenários insolúveis logo na inicialização.

7.2. Alterar posição da bola

Durante a execução do jogo, o jogador realiza suas jogadas por meio da movimentação de bolas entre as pilhas, buscando organizar as cores de forma correta até atingir a condição de vitória. Esse processo de movimentação é conduzido dentro de um laço while, que permanece ativo enquanto o jogo não for resolvido — condição essa verificada constantemente pelo método verifyWinner().

A cada iteração do laço, o console exibe o estado atual das pilhas. Em seguida, o sistema solicita duas entradas numéricas: a primeira identifica a pilha de origem (de onde a bola será retirada), e a segunda a pilha de destino (onde a bola será colocada). Ambas as entradas são lidas como String, permitindo ao jogador também digitar "exit" para abandonar a partida e retornar ao menu principal. Caso contrário, os valores são convertidos para inteiros e ajustados para trabalhar com índices iniciando do zero (conforme a lógica interna do array de pilhas).

Com as posições em mãos, o jogo chama o método changeBall(), que é o responsável por executar a lógica de movimentação da bola entre as pilhas selecionadas. Esse método inclui diversas validações essenciais para garantir a integridade das jogadas.

A primeira verificação impede que o jogador tente mover uma bola para a mesma pilha de onde ela foi retirada, lançando uma exceção com uma mensagem clara: *"Você não pode tirar uma bola para colocar na mesma pilha"*. Em seguida, é feita uma validação de faixa para garantir que os valores das pilhas estejam entre 0 e 6 (equivalente ao intervalo 1-7 para o jogador). Se o número digitado estiver fora dessa faixa, uma exceção é lançada informando que o valor é inválido.

Outras duas condições críticas também são verificadas: se a pilha de origem estiver vazia (isEmpty()), não é possível remover uma bola e uma exceção é disparada com a mensagem apropriada. Do mesmo modo, se a pilha de destino estiver cheia (isFull()), o sistema impede a jogada e avisa o usuário com um erro informando que a pilha está cheia.

Somente após todas essas validações serem aprovadas, a movimentação da bola é efetivamente realizada. A bola do topo da pilha de origem é removida com o método unstack() e armazenada temporariamente. Em seguida, seu ponteiro de referência (next) é anulado com temp.setNext(null), uma medida importante para evitar possíveis links indevidos entre elementos. Finalmente, essa bola é empilhada no topo da pilha de destino por meio do método stack(temp).

Esse processo de movimentação é repetido continuamente até que todas as pilhas estejam organizadas corretamente, momento em que o método verifyWinner() detecta a condição de vitória e o jogo é encerrado. A estrutura do código assegura

que apenas jogadas válidas sejam realizadas, promovendo uma experiência fluida e desafiadora, mas também segura do ponto de vista lógico.

7.3. Método mostrar Menu

A classe ShowMenu tem como principal objetivo facilitar a interação do jogador com o jogo "Organizar Pilhas de Bolas" por meio do console. Ela não lida com a lógica do jogo em si, mas sim com a apresentação visual e textual das informações necessárias para que o jogador entenda como jogar, o que está acontecendo no momento e quais ações estão disponíveis. Toda a comunicação com o jogador passa por essa classe, tornando-a essencial para uma boa experiência de uso.

O método showMenu() é responsável por exibir a tela inicial do jogo, onde o jogador pode escolher entre iniciar uma nova partida, visualizar as regras ou sair do jogo. Além disso, há uma observação que informa como retornar ao menu principal durante o jogo, bastando digitar "exit". Essa tela de menu serve como ponto de partida para todas as ações que o jogador pode realizar.

Outro método importante é o showRules(), que imprime no console as regras do jogo de maneira clara e objetiva. Ele explica o funcionamento das pilhas, a forma de movimentar as bolas e qual é o objetivo final do jogo. Ao fim da leitura, o jogador pode digitar qualquer número para retornar ao menu, permitindo uma navegação simples e intuitiva.

A classe também conta com o método cleanConsole(), que imprime linhas em branco no console para simular uma limpeza de tela. Embora não limpe de fato o console, essa abordagem ajuda a manter a tela visualmente organizada entre as ações do jogador, evitando confusão com informações antigas que poderiam permanecer visíveis.

Por fim, há dois métodos que exibem o estado atual do jogo: showGameStacks() e showGameStacks2(). Ambos mostram as pilhas de bolas e suas cores em tempo real, mas de maneiras diferentes. O primeiro utiliza um laço de repetição para tornar a exibição mais dinâmica e fácil de manter, enquanto o

segundo faz a impressão linha por linha, proporcionando um visual mais alinhado e refinado, embora com menor flexibilidade. Ambos os métodos cumprem a função de manter o jogador informado sobre a situação atual do jogo, o que é essencial para tomar decisões estratégicas.

Em resumo, a classe ShowMenu centraliza todas as responsabilidades relacionadas à exibição e interação com o jogador, contribuindo para que o jogo tenha uma interface amigável, compreensível e organizada. Ela garante que o jogador saiba o que fazer, como jogar e qual é o estado atual das pilhas, mesmo sem precisar entender os detalhes técnicos do código.

7.4. Verificação de Vitória

O algoritmo de verificação de vitória no *StackColor* é composto por dois métodos: verifyWinner() e winnerStack(). Juntos, eles avaliam se o jogador conseguiu organizar as bolas de maneira correta, atendendo ao objetivo principal do jogo, que é separar as bolas por cor em pilhas distintas.

O método verifyWinner() é responsável por verificar o estado de todas as pilhas do jogo, com o intuito de determinar se a condição de vitória foi alcançada. Ele percorre as sete pilhas existentes e, para cada uma, chama o método winnerStack(), que retorna true caso a pilha esteja corretamente preenchida com bolas da mesma cor. A cada pilha que satisfaz essa condição, um contador é incrementado. Ao final da verificação, se exatamente seis das sete pilhas estiverem completas e uniformes em cor, o método retorna true, indicando que o jogador venceu o jogo. Caso contrário, o retorno é false, e o jogo continua.

O método winnerStack() realiza a checagem individual de uma pilha para determinar se ela está em um estado vencedor. Primeiramente, ele verifica se a pilha está vazia (top == null); nesse caso, já retorna false. Se houver bolas na pilha, ele armazena a cor da primeira bola e percorre a pilha inteira, comparando a cor de cada bola subsequente com a cor da primeira. Se for encontrada alguma bola de cor diferente, o método retorna false imediatamente. Além disso, é feita uma contagem da quantidade de bolas na pilha, e a pilha só será considerada vencedora se

contiver exatamente sete bolas da mesma cor. Assim, o método garante não apenas a uniformidade de cor, mas também a completude da pilha.

Em conjunto, esses dois métodos formam a lógica que determina se o jogador cumpriu com sucesso o desafio do jogo. O uso do número "6" como critério no verifyWinner() sugere que uma das sete pilhas é usada como pilha auxiliar ou de movimentação, e não precisa estar completa para que o jogo seja considerado vencido. Essa abordagem torna o algoritmo robusto e coerente com as regras do jogo, assegurando que a verificação da vitória seja feita de forma precisa e eficiente.

7.5. Controle de Exceções

A classe StackExeption, localizada nos pacotes do *StackColor*, é uma extensão da classe Exception da linguagem Java e foi criada com o objetivo de representar exceções específicas relacionadas a operações inválidas em pilhas dentro da lógica do jogo ou aplicação que a utiliza. Ela permite lançar erros mais descritivos e contextualizados sempre que uma ação proibida ou incorreta for tentada sobre as estruturas de pilha usadas na aplicação.

Essa classe é estruturada com um atributo adicional chamado type, que armazena uma informação específica sobre o tipo de erro ocorrido, além da tradicional mensagem descritiva (message). O construtor principal recebe esses dois parâmetros — a mensagem e o tipo da exceção — e passa a mensagem para o construtor da superclasse Exception, enquanto armazena o tipo para uso posterior. O campo serialVersionUID é utilizado por questões de compatibilidade em processos de serialização, uma prática recomendada quando se trabalha com exceções personalizadas.

A classe também define quatro métodos estáticos auxiliares que servem como fábricas de exceções (factory methods). Cada um desses métodos representa um tipo específico de erro: stackIsEmpty para pilha vazia, stackItsFull para pilha cheia, operationInvalid para operações inválidas em geral, e operatorInvalid para operadores inválidos. Esses métodos facilitam a criação e o lançamento de exceções específicas, tornando o código mais limpo e expressivo ao invés de

instanciar manualmente novos objetos da classe com strings fixas espalhadas pelo código.

Por fim, o método getType() oferece uma forma de acessar o tipo da exceção externamente, permitindo que o tratamento de erro possa ser feito de maneira mais refinada, como exibir mensagens específicas ao usuário ou registrar logs diferenciados com base no tipo de erro. Assim, a classe StackExeption não só centraliza e organiza o tratamento de erros, como também torna o sistema mais robusto, legível e manutenível, ao separar claramente os tipos de falhas que podem ocorrer no contexto das operações com pilhas.

8. O que é uma Fila Heap?

A Fila Heap, também chamada de fila de prioridade, é um tipo especial de estrutura de dados que organiza os elementos com base em suas prioridades, e não apenas na ordem em que foram adicionados, como ocorre nas filas comuns. Isso significa que, ao invés de sempre atender o primeiro item que entrou, a Fila Heap sempre entrega primeiro o item mais importante — ou seja, o de maior ou menor valor, dependendo da forma como foi configurada.

Para organizar esses elementos, a Fila Heap utiliza uma estrutura parecida com uma árvore, mas que é armazenada dentro de um vetor (ou lista). Essa estrutura é organizada de maneira que o item de maior prioridade (por exemplo, o número mais alto) fique sempre no topo. Quando um novo item é adicionado, ele é colocado no fim da fila e depois comparado com os outros até encontrar o lugar certo, mantendo a ordem correta da fila. O mesmo acontece quando o item de maior prioridade é removido: a estrutura se reorganiza automaticamente para que o próximo item mais importante vá para o topo.

A Fila Heap pode funcionar de duas formas principais: como Max Heap ou como Min Heap. No Max Heap, o maior valor sempre fica no topo da estrutura, ou seja, é o primeiro a ser retirado quando necessário. Esse tipo é útil em situações onde se deseja sempre acessar o item mais importante ou com maior prioridade. Já no Min Heap, o menor valor é quem ocupa o topo da estrutura, sendo o primeiro a

ser atendido. Esse modelo é ideal quando se quer sempre tratar o item menos custoso ou com menor valor numérico primeiro. Ambos os tipos mantêm sua organização automaticamente a cada inserção ou remoção de elementos, garantindo que a ordem de prioridade seja respeitada o tempo todo.

Essa estrutura é muito útil porque consegue manter a ordem de prioridade de maneira rápida e eficiente, mesmo quando os elementos estão sendo constantemente adicionados ou removidos. Por isso, a Fila Heap é uma ferramenta importante no desenvolvimento de programas que precisam lidar com decisões rápidas e organizadas.

9. Diferenças entre Pilha e Fila Heap

A Pilha e a Fila Heap são estruturas de dados utilizadas para organizar e gerenciar informações de formas distintas, cada uma com propósitos específicos. A principal diferença entre ambas está no princípio de organização dos elementos. A Pilha segue a lógica LIFO (Last In, First Out), ou seja, o último elemento inserido é o primeiro a ser removido. Já a Fila Heap é baseada em um sistema de prioridades, onde os elementos são removidos de acordo com seu valor, e não necessariamente na ordem em que foram inseridos.

Na estrutura de Pilha, os métodos principais são push, que insere um elemento no topo da pilha, e pop, que remove o elemento do topo. A pilha é ideal para problemas que envolvem ordem inversa de execução, como chamadas de função, desfazer operações ou verificação de expressões aninhadas.

Por outro lado, a Fila Heap é uma estrutura baseada em árvore binária completa, geralmente implementada como um array, onde os elementos são organizados de modo a sempre manter o maior (no caso de uma max heap) ou o menor (no caso de uma min heap) valor na raiz. Os principais métodos da heap são insert, que adiciona um novo valor mantendo a propriedade de heap, e remove (ou extract), que remove o valor da raiz (máximo ou mínimo) e reorganiza os demais.

Enquanto a Pilha é linear e trabalha com a posição mais recente, a Fila Heap é hierárquica e trabalha com a prioridade dos elementos. Em termos de desempenho, a Pilha realiza inserções e remoções em tempo constante, enquanto a Fila Heap realiza essas operações em tempo logarítmico, devido à necessidade de reorganizar os elementos para manter a propriedade da heap.

10. Principais Aplicações da Pilha e Fila Heap

A estrutura de dados Pilha é amplamente utilizada em diversas situações do mundo real onde há necessidade de controle de ordem reversa ou de execução organizada em etapas. Um exemplo clássico é o controle de chamadas de funções em linguagens de programação: cada nova função chamada é empilhada e, ao ser finalizada, é removida da pilha. Outra aplicação bastante comum ocorre na navegação por páginas na internet, em que o navegador mantém uma pilha para permitir que o usuário volte à página anterior. Pilhas também são utilizadas em algoritmos de backtracking, como na resolução de labirintos ou quebra-cabeças, além de desempenharem papel importante em verificações de sintaxe, como o balanceamento de parênteses em expressões matemáticas e códigos-fonte.

Já a Fila Heap, por sua vez, aparece em cenários onde é necessário gerenciar prioridades. Um exemplo real está em sistemas operacionais, nos quais o escalonador de processos utiliza filas de prioridade para decidir qual tarefa será executada em seguida, dando preferência a processos mais urgentes. Outro uso relevante é nos algoritmos de caminho mínimo, como o algoritmo de Dijkstra, utilizado em aplicações de GPS e mapas para encontrar a rota mais curta entre dois pontos. Filas heap também são essenciais em sistemas de simulação de eventos, como em jogos ou modelagens de tráfego, onde os eventos mais próximos de ocorrer precisam ser processados primeiro. Além disso, elas são utilizadas em compressão de dados, como no algoritmo de Huffman, para organizar os símbolos com base em suas frequências de uso.

11. Benefícios Educacionais

O desenvolvimento do jogo *StackColor* apresenta inúmeros benefícios educacionais, especialmente no contexto do ensino de estruturas de dados. Ao simular o funcionamento de pilhas por meio de uma mecânica lúdica e interativa, o jogo permite que os alunos compreendam, de forma prática, o princípio LIFO (Last In, First Out) e a lógica por trás dessa estrutura. A visualização e manipulação das bolas coloridas reforçam a compreensão de como os dados são armazenados, acessados e removidos na pilha, o que fortalece a internalização do conceito de estrutura linear e suas operações básicas. Esse tipo de abordagem promove uma aprendizagem mais significativa, pois conecta a teoria com uma aplicação concreta e acessível.

Além disso, o jogo incentiva o raciocínio lógico e a formulação de estratégias diante de restrições claras, como mover apenas uma bola por vez e não empilhar cores diferentes. Isso leva os alunos a pensarem de maneira algorítmica, buscando sequências eficientes de ações para atingir o objetivo final — que, na prática, se assemelha à resolução de problemas de ordenação. No aspecto técnico, o desenvolvimento e o uso do jogo também favorecem o exercício da programação orientada a objetos, visto que os elementos do jogo são representados por classes, atributos e métodos, estimulando boas práticas de encapsulamento e modularização do código. Assim, *StackColor* se consolida como uma ferramenta didática eficaz, unindo conceitos fundamentais de programação com uma experiência prática e envolvente.

12. Conclusão

A construção deste projeto proporcionou uma rica oportunidade de aprendizado e aprofundamento nos conceitos de estruturas de dados, especialmente no uso de pilhas, além de reforçar conhecimentos de orientação a objetos e tratamento de exceções. Ao longo do desenvolvimento da aplicação, diversos desafios foram enfrentados — desde o planejamento da lógica do jogo até a implementação dos comportamentos que simulam as regras de movimentação entre as pilhas de bolas coloridas.

Um dos principais pontos de atenção foi garantir que a manipulação das pilhas respeitasse as restrições lógicas do jogo, como a movimentação apenas quando válida e a verificação constante do estado atual para avaliar condições de vitória. Isso exigiu um raciocínio cuidadoso na construção de métodos que fossem ao mesmo tempo eficientes e confiáveis. A definição das condições de vitória, por exemplo, exigiu uma lógica minuciosa que envolvesse a análise de todos os elementos de uma pilha, além da sua uniformidade de cor e tamanho adequado.

O tratamento de exceções foi outro aspecto importante, pois permitiu lidar com situações irregulares, como tentativas de manipular pilhas vazias ou cheias, de forma controlada e elegante, garantindo maior robustez e segurança para a aplicação. A criação de uma classe específica para essas exceções tornou o código mais legível, organizado e preparado para futuras expansões ou ajustes.

Apesar das dificuldades técnicas enfrentadas, como o ajuste de métodos recursivos e a validação da lógica de jogo em diferentes cenários, a aplicação final alcançou seu objetivo: oferecer uma experiência interativa e desafiadora, que alia entretenimento ao raciocínio lógico. Além disso, o projeto serviu como excelente exercício de prática e fixação de conceitos fundamentais da programação em Java.

Concluímos, portanto, que o desenvolvimento deste jogo não apenas resultou em uma aplicação funcional e lúdica, mas também contribuiu significativamente para a formação técnica e lógica envolvida no processo. Superar os obstáculos e ver o sistema em pleno funcionamento evidencia a importância da persistência, do planejamento bem estruturado e da análise detalhada em projetos de desenvolvimento de software.

13. Referências

Projeto pessoal baseado em jogos simples e exercícios de programação em Java, 2025.

— Desenvolvimento do jogo de organização de pilhas coloridas, aplicando conceitos de estruturas de dados aprendidos em sala de aula com os professores Lucas Santos e Luciano Savio.