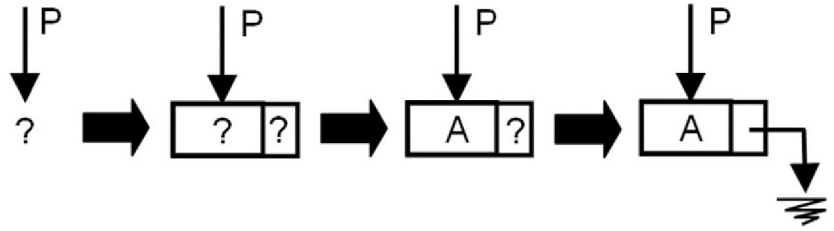
Capítulo 5

Listas Encadeadas com Alocação Dinâmica



Seus objetivos neste capítulo

- Entender o que é Alocação Dinâmica de Memória, no contexto do armazenamento temporário de conjuntos de elementos.
- Entender que a Alocação Encadeada e a Alocação Dinâmica são conceitos independentes que, quando combinados, formam uma técnica flexível e poderosa para armazenamento temporário de conjuntos de elementos.
- Desenvolver habilidade para implementar estruturas encadeadas com Alocação Dinâmica de Memória.
- Fazer uma reflexão visando a escolher a técnica de armazenamento mais adequada aos jogos que você está desenvolvendo.

5.1 Alocação Dinâmica de Memória para um conjunto de elementos

Conforme estudamos no Capítulo 2, na Alocação Estática de Memória definimos previamente o tamanho máximo do conjunto e reservamos memória para todos os seus elementos. O espaço reservado não pode crescer ou diminuir ao longo da execução do programa. Mesmo se a quantidade de elementos no conjunto for menor que o seu tamanho máximo, o espaço para todos os elementos permanecerá reservado. Por exemplo, suponha que reservamos espaço para armazenar 1.000 elementos. No decorrer da execução do programa, apenas 150 elementos entram no conjunto. Mas o espaço para os outros 850 elementos permanecerá reservado durante toda a execução.

Na Alocação Dinâmica, o espaço de memória pode ser alocado *no decorrer* da execução do programa, quando for efetivamente necessário. Com a Alocação Dinâmica, podemos alocar memória para um elemento de cada vez: quando um novo elemento entrar no conjunto, reservamos memória para armazená-lo. Se um único elemento entrou no conjunto, teremos alocado espaço para um único elemento; se entraram 150, teremos alocado espaço para 150 elementos.

Definição: Alocação Dinâmica de Memória para um conjunto de elementos
Na Alocação Dinâmica de Memória para um conjunto de elementos: • Espaços de memória podem ser alocados no decorrer da execução do programa quando forem efetivamente necessários. • É possível alocar espaço para um elemento de cada vez. • Espaços de memória também podem ser liberados no decorrer da execução do programa quando não forem mais necessários. • Também é possível liberar espaço de um elemento de cada vez.

Figura 5.1 Alocação Dinâmica para um conjunto de elementos.

5.2 Alocação Dinâmica nas linguagens C e C++

A Figura 5.2 apresenta um conjunto de comandos das linguagens C e C++. Na linha 1 da Figura 5.2 estamos declarando duas variáveis do tipo Inteiro, denominadas X e Y. Na linha 2, declaramos duas variáveis do tipo Ponteiro para Inteiro, P1 e P2. Isso significa que P1 e P2 podem armazenar o endereço da variável X, o endereço da variável Y ou o endereço de outras variáveis do tipo Inteiro.

Através dos comandos das linhas 1 e 2, espaços de memória para as variáveis X, Y, P1 e P2 são reservados através de Alocação Estática. Ou seja, os espaços de memória alocados para X, Y, P1 e P2 permanecem reservados durante toda a execução do programa ou módulo em que esses comandos estiverem inseridos.

Os diagramas da Figura 5.3 mostram uma representação de X, Y, P1 e P2 alocados na memória: a variável X está alocada na posição 920 da memória, Y na posição 923, P1 na posição 914 e P2 na posição 908. A representação da Figura 5.3 considera que cada variável do tipo Inteiro ocupa duas unidades de memória, e cada variável do tipo Ponteiro para Inteiro ocupa quatro unidades de memória. Essa mesma consideração é adotada nas figuras seguintes.

	C++	C
1	int X, Y;	int X, Y;
2	int *P1, *P2;	int *P1, *P2;
3	X = Y;	X = Y;
4	X = *P2;	X = *P2;
5	*P1 = *P2;	*P1 = *P2;
6	P1 = P2;	P1 = P2;
7	P1 = &X;	P1 = &X;
8	P1 = new int;	P1 = (int *) malloc((unsigned) (sizeof(int)));
9	delete P1;	free((char *) P1);

Figura 5.2 Comandos nas linguagens C e C++.

P1, que é do tipo Ponteiro para Inteiro, está armazenando o endereço de memória da variável X, que é 920; P2, também do tipo Ponteiro para Inteiro, está armazenando o endereço da variável Y, 923. Podemos interpretar que P1 e P2 "apontam" para os endereços de X e Y, respectivamente, conforme mostram as setas pontilhadas. Na parte inferior da Figura 5.3 há outra representação, destacada com o fundo cinza, que deixa essa interpretação ainda mais evidente.

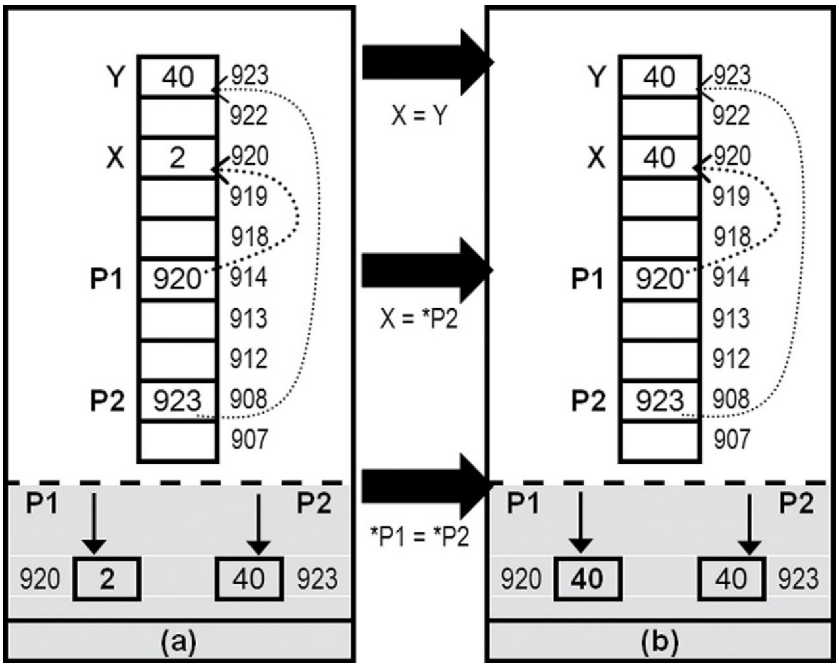


Figura 5.3 Representação de X, Y, P1 e P2 — execução dos comandos X = Y ou X = *P2 ou *P1 = *P2.

A Figura 5.3a mostra uma situação inicial, e a Figura 5.3b mostra essa situação inicial modificada pela execução do comando da linha 3 da Figura 5.2 (**X = Y**). Com a execução desse comando, X passa a ter o valor da variável Y, que é 40. Note que o valor armazenado em P1 (920) não muda. O que muda é o valor armazenado no espaço de memória para onde P1 aponta, que armazenava o valor 2, e passa a armazenar o valor 40.

O comando da linha 4 da Figura 5.2 (**X = *P2**) deve ser lido como "X recebe o conteúdo apontado por P2". Na Figura 5.3a, o valor do conteúdo apontado por P2 é 40. Assim, se executado sobre a situação da Figura 5.3a, o comando X = *P2 resultaria na situação da Figura 5.3b, onde o valor da variável X é 40. O comando da linha 5 da Figura 5.2 (***P1 = *P2**) pode ser lido como "o conteúdo apontado por P1 recebe o conteúdo apontado por P2". Na Figura 5.3a, o conteúdo apontado por P1 é o conteúdo de X, e o conteúdo apontado por P2 é o conteúdo de Y. Logo, os comandos das linhas 3 (X = Y), 4 (X = *P2) e 5 (*P1 = *P2) da Figura 5.2 produzem exatamente o mesmo efeito na Figura 5.3a, resultando na situação da Figura 5.3b.

A Figura 5.4 mostra a alteração de uma situação inicial (Figura 5.4a) pela execução do comando da linha 6 da Figura 5.2 (**P1 = P2**). A partir da execução dessa operação, o ponteiro P1 passa a ter o mesmo valor do ponteiro P2. Na Figura 5.4a, P1 armazena o valor 920; já na Figura 5.4b, P1 armazena o valor 522, que é o mesmo valor armazenado em P2. Em uma interpretação mais visual, P1 passa a apontar para o mesmo lugar para onde aponta P2. A representação na parte inferior da figura mostra que, com o comando P1 = P2, estamos "movendo" o ponteiro P1 para onde está o ponteiro P2.

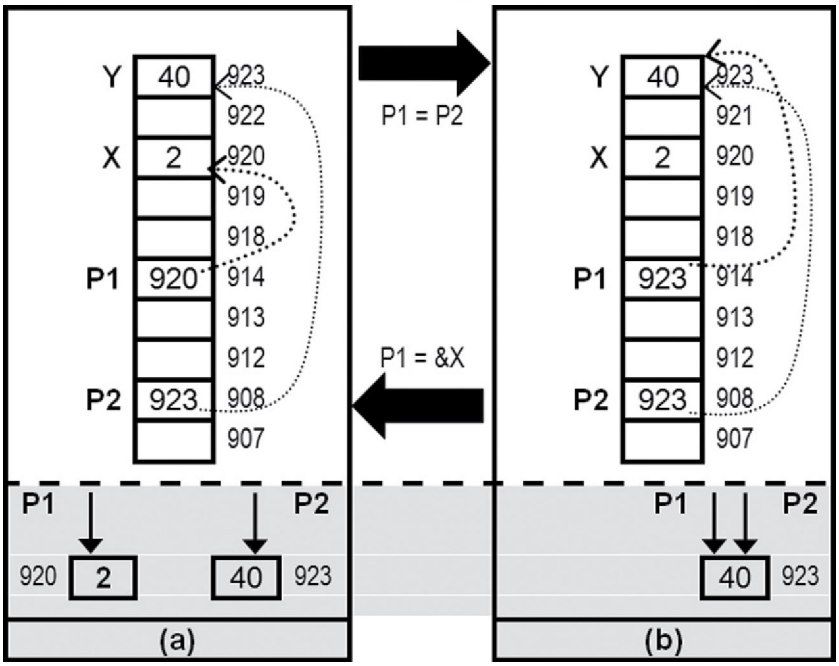


Figura 5.4 Representação de X, Y, P1 e P2 — execução dos comandos *P1 = P2* e *P1 = &X*.

O comando da linha 7 da Figura 5.2 (**P1 = &X**) pode ser lido como "P1 recebe o endereço da variável X". Se tomarmos como ponto de partida a situação da Figura 5.4b, a execução desse comando resultará na situação ilustrada na Figura 5.4a. Ou seja, P1 voltará a armazenar o valor 920, que é o endereço de memória da variável X. Visualmente, estaríamos movendo P1 de volta para a posição em que se encontrava anteriormente: apontando para a posição de memória 920.

Preste bastante atenção à diferença de significado entre o valor de P1 e o valor do conteúdo apontado por P1. Na situação da Figura 5.4a, o valor de P1 é 920 e o valor do conteúdo apontado por P1 é 2. Interpretando essa situação: P1 está apontando para a posição de memória 920 e nessa posição apontada por P1 está sendo armazenado um Inteiro, de valor 2.

Alocando Memória Dinamicamente

Através dos comandos das linhas 1 e 2 da Figura 5.2, as variáveis X, Y, P1 e P2 foram alocadas estaticamente. A linha 8 da Figura 5.2 traz comandos de Alocação Dinâmica de Memória. Em C++, o comando utilizado é **new**; em C, o comando é **malloc**.

Com a execução da linha 8 da Figura 5.2 (em C + +: **P1 = new int**), estamos alocando dinamicamente espaço de memória para armazenar uma variável do tipo Inteiro, e o endereço desse espaço de memória é atribuído à variável P1. Conforme mostra a Figura 5.5, na situação inicial — Figura 5.5a — a variável P1 já está alocada e possui um

valor qualquer, desconhecido. Ou seja, a variável P1 já existe e aponta para uma posição de memória indefinida. Após a execução do comando **P1 = new int**, um novo espaço de memória é alocado (no exemplo, posição 924), e P1 passa a apontar para essa posição de memória. Em uma representação mais visual, na parte de baixo da figura, em cinza, P1 deixou de apontar para uma posição qualquer, desconhecida, e passou a apontar para a posição 924 (Figura 5.5b).

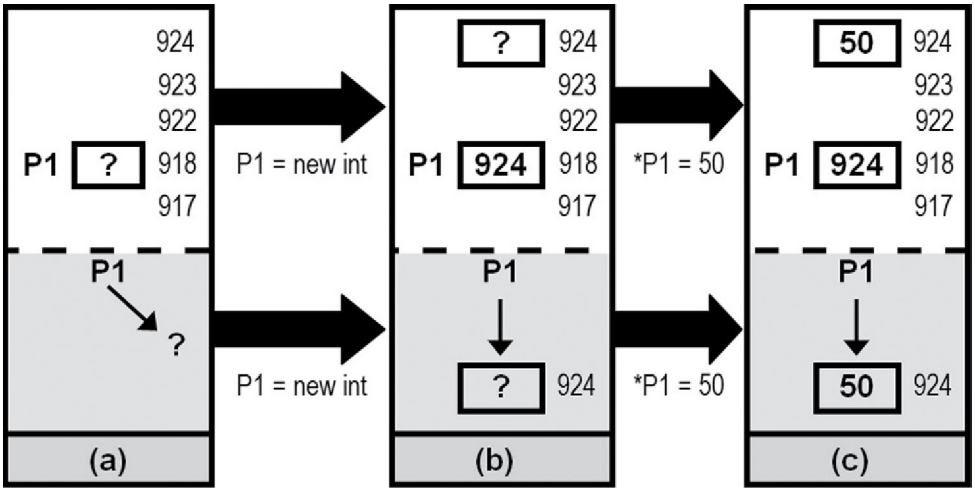


Figura 5.5 **Alocando memória dinamicamente: execução dos comandos *P1 = new int* e **P1 = 50*.**

É possível verificar o sucesso de uma operação de alocação dinâmica de memória: se não houvesse memória disponível para ser alocada, em vez de apontar para a posição de memória recentemente alocada (924), P1 estaria apontando para NULL.

Na situação da Figura 5.4b, o valor armazenado na posição 924 ainda é desconhecido. Para alterar essa situação podemos utilizar o comando ***P1 = 50**, por exemplo. A interpretação desse comando seria: "O conteúdo apontado por P1 recebe o valor 50." A partir da execução desse comando, a posição de memória 924 passaria a armazenar o valor 50, como mostra a Figura 5.5c.

Preste atenção ao seguinte ponto: P1 é uma variável do tipo Ponteiro para Inteiro. O comando P1 = new int aloca espaço de memória para armazenar valores do tipo Inteiro (ou seja, espaço equivalente a duas unidades de memória) e não valores do tipo Ponteiro para Inteiro (o que demandaria quatro unidades de memória).

Se olharmos para a parte superior da Figura 5.5, será preciso certo grau de concentração para entender o que está acontecendo. Contudo, se observarmos a parte inferior da Figura 5.5, com fundo cinza, a interpretação será muito simples: da situação *a* para a situação *b* simplesmente "aparece do nada" um espaço de memória, e P1 está apontando para ele; da situação *b* para a situação *c*, esse espaço de memória que "apareceu do nada" passa a ter um valor.

Desalocando memória dinamicamente

Para desalocar memória dinamicamente, na linguagem C++ o comando utilizado é **delete**. Na linguagem C, o comando utilizado é **free**. Na linha 9 da Figura 5.2 temos o comando (em C++) **delete P1**. Se tomarmos como ponto de partida a situação da Figura 5.6a, o comando delete P1 libera o espaço de memória apontado pela variável P1, resultando na situação da Figura 5.6b. Note que P1 é uma variável do tipo Ponteiro para Inteiro. Ao executarmos o comando delete P1, estamos desalocando o espaço de memória referente a um Inteiro (duas unidades de memória).

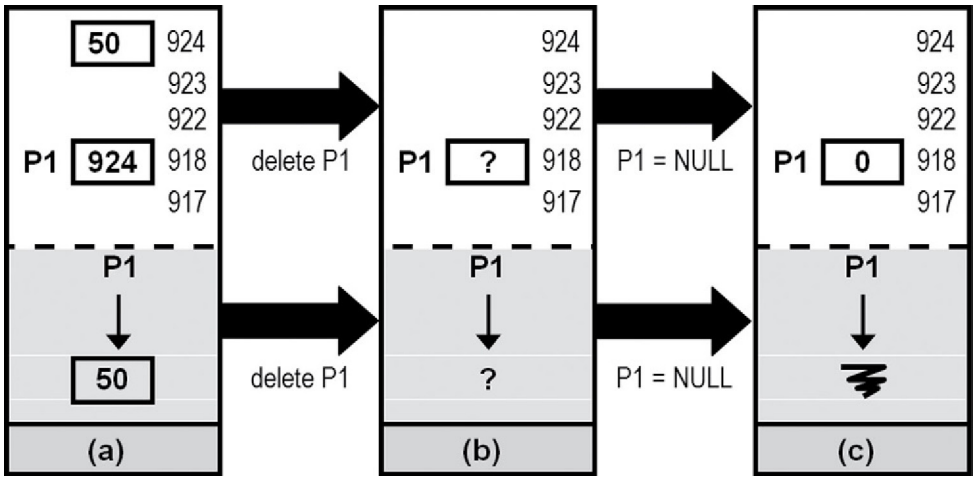


Figura 5.6 **Desalocando memória dinamicamente: execução dos comandos *delete P1* e *P1 = NULL*.**

Na situação da Figura 5.6b, conceitualmente não podemos mais manipular o conteúdo apontado por P1, pois o espaço de memória para o qual P1 estava apontando foi desalocado. Precisamos então aplicar um comando como P1 = NULL (que equivale ao comando P1 = 0) para atribuir um valor bem definido e seguro para P1, resultando na situação da Figura 5.6c.

Se você se concentrar na parte inferior da Figura 5.6, com fundo cinza, a interpretação visual da situação será muito simples. Da situação *a* para a situação *b*, o espaço de armazenamento apontado por P1 simplesmente "desaparece do nada", deixando P1 em uma situação conceitualmente confusa. Da situação *b* para a situação *c*, P1 passa a apontar para uma posição bem definida, conhecida como NULL.

5.3 Nós de uma Lista Encadeada alocados dinamicamente

Até o momento utilizamos variáveis do tipo Inteiro e do tipo Ponteiro para Inteiro para ilustrar a Alocação Dinâmica de Memória e a manipulação de valores armazenados em espaços de memória alocados dinamicamente. Para alocar dinamicamente Nós de uma Lista Encadeada, em vez de Inteiros e Ponteiros para Inteiros, basta trabalharmos, analogamente, com Nós e Ponteiros para Nós.

Na Figura 5.7 ilustramos na linguagem de programação C++ as definições conceituais que fizemos no Capítulo 4 (Figura 4.4). Na linha 1 definimos o tipo Node, e na linha 2, o tipo NodePtr.

```
1   struct Node {
        char Info;
        struct node *Next;
    };
2   typedef struct Node *NodePtr;
3   NodePtr P; PAux;
4   Int X;
5   P = new Node;
6   P->Info = 50;
7   P->Next = NULL;
8   X = P->Info;
9   PAux = P->Next;
10  Delete P;
11  P = NULL;
```

Figura 5.7 Implementação de Listas Encadeadas com alocação dinâmica: comandos em C++ .

Cada Nó da Lista Encadeada possui os campos Info (utilizado para armazenar informação) e Next (utilizado para apontar o próximo elemento da lista). Conforme mencionamos no Capítulo 4, essa nomenclatura foi definida de modo a manter certa compatibilidade com a adotada em parte da literatura sobre estruturas de dados (como Drozdek [2002] e também Langsam, Augenstein e Tenenbaum [1996]).

Na linha 3 declaramos as variáveis P e PAux do tipo Ponteiro para Nó, e na linha 4 declaramos X do tipo inteiro. Na linha 5 encontramos o comando **P = new Node**, que implementa a operação conceitual **P = NewNode** que utilizamos nos algoritmos do Capítulo 4 (veja a Figura 4.4). Através do comando P = new Node estamos alocando dinamicamente espaço de memória para armazenar valores do tipo Node. Ou seja, estamos alocando um Nó da Lista Encadeada, e a variável P estará apontando para esse Nó. Note que, na situação da Figura 5.8a, simplesmente "aparece do nada" um novo Nó, e a variável P passa a apontar para esse novo Nó (Figura 5.8b).

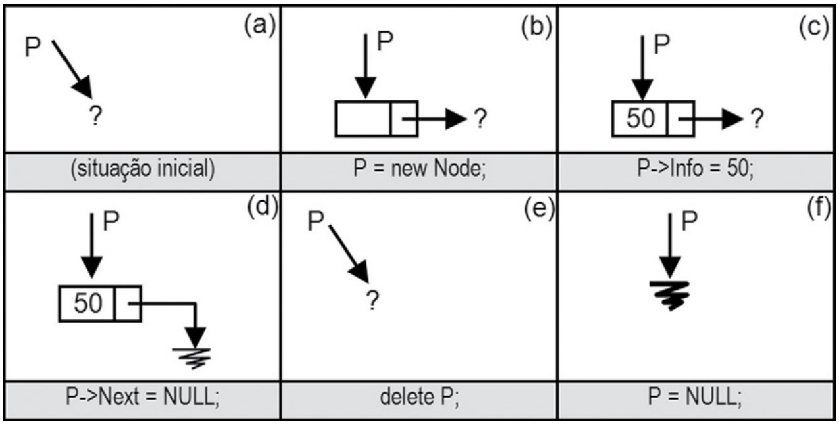


Figura 5.8 Manipulando uma Lista Encadeada alocada dinamicamente: comandos em C++.

O comando da linha 6 da Figura 5.7, **P- > Info = 50** pode ser lido da forma: "A porção Info do Nó apontado por P recebe o valor 50." Um modo alternativo de realizar essa leitura

seria: "Info de P recebe 50." Na situação da Figura 5.8b, o Nó apontado por P ainda não possui valor em seus campos Info e Next. Após executarmos o comando P- > Info = 50, teremos como resultado a situação da Figura 5.8c, na qual o campo Info do Nó apontado por P possui valor 50. Analogamente, se executarmos em seguida o comando da linha 7, **P- > Next = NULL**, chegaremos à situação da Figura 5.8d. O acesso aos campos Info e Next de um Nó apontado por P é exemplificado pelas linhas 8 e 9 da Figura 5.7.

Podemos exemplificar a operação de desalocar um Nó da Lista Encadeada com o comando da linha 10 da Figura 5.7: **delete P**, que levaria a situação da Figura 5.8d à situação da Figura 5.8e. O Nó apontado por P simplesmente "desaparece"! O comando delete P implementa a operação conceitual **DeleteNode (P)**, que utilizamos nos algoritmos do Capítulo 4 (veja a Figura 4.4).

Finalmente, para não deixar o ponteiro P apontando para uma posição indefinida, aplicamos o comando **P = NULL** (linha 11 da Figura 5.7) e chegamos à situação da Figura 5.8f.

Dica importante: desenhe!

Ao elaborar e testar algoritmos sobre Listas Encadeadas, utilize sempre um conjunto de diagramas. Desenhe a execução do seu algoritmo! Desenhe passo a passo! A representação visual simplifica a compreensão e evita erros.

Exercício 5.1 Revisar comandos da Operação Empilha

No Capítulo 4 implementamos uma Pilha como uma Lista Encadeada e utilizamos uma notação conceitual para a elaboração dos algoritmos. A Figura 5.9 é uma reprodução da Figura 4.11, e suas imagens ilustram a execução passo a passo da operação Empilha, para uma situação inicial com a Pilha vazia. Implemente em C++ cada um dos comandos expressos em linguagem conceitual na parte cinza de cada diagrama. Consulte a Figura 5.7 se tiver dúvidas.

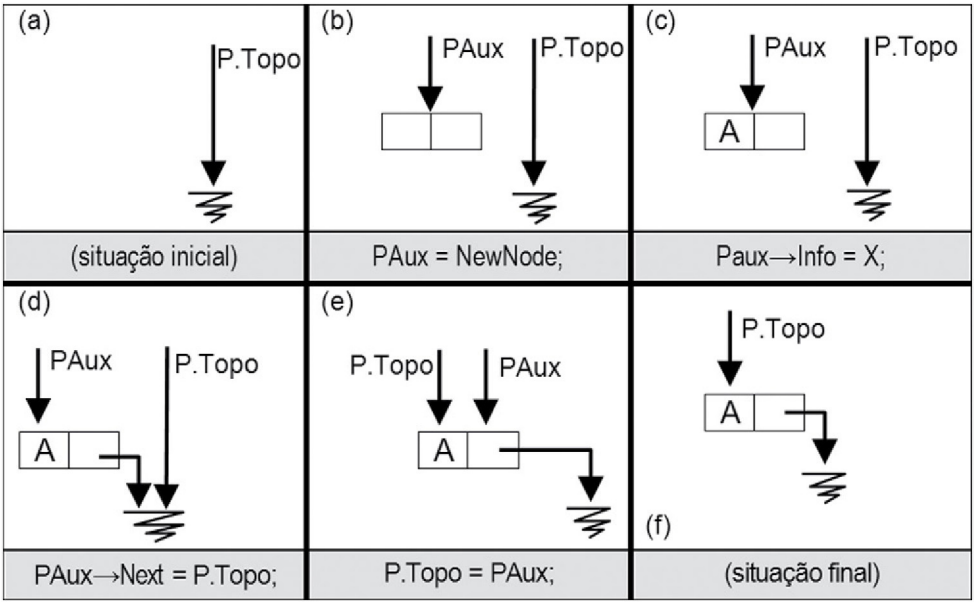


Figura 5.9 Operação Empilha partindo de Pilha Vazia.

Na linguagem C++, a implementação do comando conceitual PAux = NewNode é PAux = new Node (Figura 5.9b). O comando conceitual da Figura 5.9c, Paux→Info = X pode ser implementado por PAux- > Info = X. PAux→Next = P.Topo (Figura 5.9d) pode ser implementado por PAux- > Next = P.Topo. Na Figura 5.9e, a notação conceitual é idêntica à implementação em C++, ou seja: P.Topo = PAux.

Exercício 5.2 Revisar comandos da Operação Desempilha

No Capítulo 4, implementamos uma Pilha como uma Lista Encadeada e utilizamos uma notação conceitual para a elaboração dos algoritmos. A Figura 5.10 é uma reprodução da Figura 4.13, e suas imagens ilustram a execução passo a passo da operação Desempilha, para uma situação inicial com a Pilha com dois elementos. Implemente em C++ cada um dos comandos expressos em linguagem conceitual na parte cinza de cada diagrama. Consulte a Figura 5.7 se tiver dúvidas.

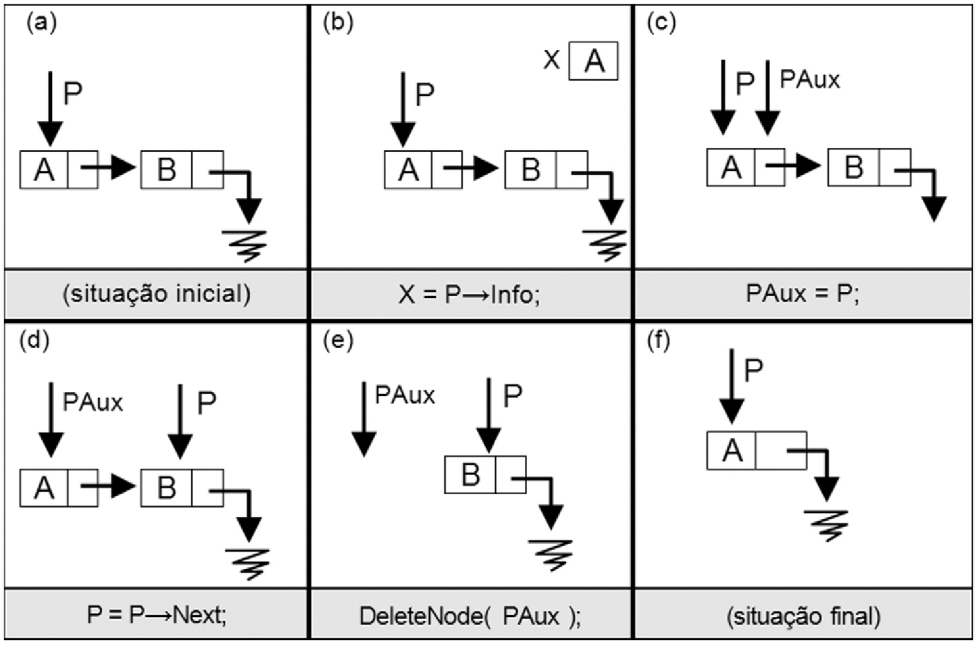


Figura 5.10 Operação Desempilha — situação inicial de Pilha com dois elementos.

Exercício 5.3 Implemente uma Pilha em C++ com Alocação Encadeada e Dinâmica de Memória

No Capítulo 4, implementamos uma Pilha como uma Lista Encadeada e utilizamos uma notação conceitual para a elaboração dos algoritmos. Implemente, na linguagem de programação C++, uma Pilha como uma Lista Encadeada e com Alocação Dinâmica de Memória. Implemente as Operações Primitivas Empilha, Desempilha, Cria, Vazia e Cheia. Tome como base os algoritmos elaborados nos Exercícios 4.1 a 4.6 (Figuras 4.10 e 4.12). Em um arquivo separado, faça um programa para testar o funcionamento das operações da Pilha.

Exercício 5.4 Implemente uma Fila em C++ com Alocação Encadeada e Dinâmica de Memória

No Capítulo 4 implementamos uma Fila como uma Lista Encadeada e utilizamos uma notação conceitual para a elaboração dos algoritmos. Implemente, na linguagem de programação C++, uma Fila como uma Lista Encadeada e com Alocação Dinâmica de Memória. Implemente as Operações Primitivas Retira, Insere, Cria, Vazia e Cheia. Tome como base a especificação e os algoritmos elaborados nos Exercícios 4.8 a 4.12. Em um arquivo separado, faça um programa para testar o funcionamento das operações da Fila.

5.4 Alocação Sequencial e Estática ou Encadeada e Dinâmica?

Nos Capítulos 2 e 3, utilizamos o conceito de Alocação Sequencial juntamente com o conceito de Alocação Estática para implementar estruturas do tipo Pilha e Fila. Alocação Sequencial é um conceito independente do conceito de Alocação Estática. Os conceitos foram combinados.

Analogamente, a Alocação Encadeada de Memória é um conceito independente do conceito de Alocação Dinâmica de Memória. Mas são conceitos que se encaixam perfeitamente conforme mostram as implementações discutidas nos Capítulos 4 e 5.

Comparando Sequencial-Estática com Encadeada-Dinâmica

Na técnica de Alocação Sequencial e Estática, utilizamos um vetor e indicamos previamente a quantidade máxima de elementos que podem entrar no conjunto. Reservamos memória para a quantidade máxima de elementos, mesmo que na prática o número de elementos seja menor.

Na técnica de Alocação Encadeada e Dinâmica, não é necessário indicar previamente quantos elementos poderão fazer parte do conjunto. Novos elementos podem ser agregados ao conjunto na medida da necessidade. Na prática, na Alocação Encadeada e Dinâmica, o limite para o crescimento de um conjunto é a disponibilidade de memória do computador ou dispositivo em que o programa está sendo executado. Assim, a Alocação Encadeada e Dinâmica é uma técnica bastante flexível em relação ao número de elementos e possibilita que a memória seja compartilhada com eficiência entre diversas estruturas de armazenamento.

Outra característica da Alocação Encadeada e Dinâmica é a facilidade para modelar diferentes situações. É bastante simples ajustar uma lista encadeada para que seja circular ou não, para que tenha um indicador para o final da lista ou não, e assim por diante. Nos próximos capítulos implementaremos diversas outras variações das Listas Encadeadas, evidenciando ainda mais sua flexibilidade.

Para uma leitura complementar sobre alocação de memória no contexto de conjuntos de elementos, consulte Pereira (1996, p. 9-14, 83-93), Langsam, Augenstein e Tenenbaum (1996, p. 203-207]), e Celes, Cerqueira e Rangel (2004, p. 64-72).

Exercício 5.5 Avanço de Projeto: refletir sobre a aplicação FreeCell *e definir a técnica mais adequada para implementação da Pilha*

Você já implementou uma Pilha com duas técnicas diferentes: com Alocação Sequencial e Estática (Capítulo 2) e com Alocação Encadeada e Dinâmica (Capítulos 4 e 5). Considerando a aplicação em que será utilizada a Pilha, qual dessas duas técnicas de implementação você considera mais adequada?

Exercício 5.6 Avanço de Projeto: refletir sobre a aplicação Snake *e definir a técnica mais adequada para implementação da Fila*

Você já implementou uma Fila com duas técnicas diferentes: com Alocação Sequencial e Estática (Capítulo 3) e com Alocação Encadeada e Dinâmica (Capítulos 4 e 5). Considerando a aplicação em que será utilizada a Fila, qual dessas duas técnicas de implementação você considera mais adequada?

Comparação entre Alocação Sequencial e Estática e Alocação Encadeada e Dinâmica

A Alocação Sequencial e Estática é uma técnica simples e adequada a situações em que a quantidade de elementos que poderão entrar no conjunto é previsível, com pequena margem de variação. A Alocação Encadeada e Dinâmica é flexível com relação à quantidade de elementos e pode ser facilmente adaptada para modelar diferentes necessidades; é uma técnica poderosa e muito utilizada para o armazenamento temporário de conjuntos de elementos.

Consulte nos Materiais Complementares

Vídeos sobre Alocação Dinâmica nas Linguagens C e C++

http://www.elsevier.com.br/edcomjogos

Exercícios de fixação

Exercício 5.7 Diferença entre Alocação Estática e Alocação Dinâmica de Memória. Qual a diferença entre Alocação Estática e Alocação Dinâmica de Memória, no contexto do armazenamento temporário de conjuntos de elementos?

Exercício 5.8 Diferença entre Alocação Sequencial e Alocação Encadeada de Memória. Qual a diferença entre Alocação Sequencial e Alocação Encadeada de Memória, no contexto do armazenamento temporário de conjuntos de elementos?

Exercício 5.9 Conceitos independentes. Explique por que a Alocação Encadeada e a Alocação Dinâmica são conceitos independentes que, quando combinados, formam uma técnica poderosa.

Exercício 5.10 Diagrama da implementação de Pilha — Alocação Sequencial e Estática de Memória. Como seria implementar uma Pilha com Alocação Sequencial e Estática de Memória? Faça um diagrama e explique resumidamente o funcionamento dessa técnica de implementação.

Exercício 5.11 Diagrama da implementação de Pilha — Alocação Encadeada e Dinâmica de Memória. Como seria implementar uma Pilha com Alocação Encadeada e Dinâmica de Memória? Faça um diagrama e explique resumidamente o funcionamento dessa técnica de implementação.

Exercício 5.12 Vantagens e desvantagens. Quais são as vantagens e desvantagens das técnicas de Alocação Sequencial e Estática e Alocação Encadeada e Dinâmica? Sugestão de uso acadêmico: discuta com os colegas as vantagens e desvantagens dessas técnicas.

Exercício 5.13 Implemente uma classe Node em C++. Revise a definição dos tipos Node e NodePtr realizada na Figura 5.7 e implemente uma classe Node. A classe Node deve definir um Nó com os campos Info e Next, e o tipo NodePtr. Defina e implemente as operações: NewNode (para alocar um Nó), DeleteNode (liberar um Nó), GetInfo (acesso ao valor da informação armazenada no Nó), SetInfo (atualização da informação armazenada no Nó), GetNext (acesso à indicação do próximo elemento da lista, armazenada no Nó) e SetNext (atualização da indicação do próximo elemento da lista, armazenada no Nó). Altere a solução do Exercício 5.3 ou do Exercício 5.4, para que utilize a classe Node. Realize alguns testes para se certificar do correto funcionamento.

Exercício 5.14 Testar e aprimorar a portabilidade e a reusabilidade das soluções com Pilha. No Exercício 5.3, você implementou um TAD Pilha com Alocação Encadeada e Dinâmica e, em um arquivo separado, fez um Programa Teste, para verificar o funcionamento do TAD Pilha. Avalie o grau de portabilidade e o potencial de reusabilidade de sua solução da seguinte forma: pegue o mesmo Programa Teste que utilizou para avaliar o funcionamento do TAD Pilha implementado com Alocação Encadeada e Dinâmica, e utilize para testar o TAD Pilha que você implementou nos Exercícios 2.12 ou 2.13, com Alocação Sequencial e Estática. Idealmente, o seu Programa Teste deve executar com ambas as versões do TAD Pilha, sem necessidade de qualquer alteração. Se, ao trocar a implementação da Pilha, você tiver que fazer alguma alteração no Programa Teste, as soluções devem ser aprimoradas até que o Programa Teste execute com ambas as implementações de Pilha, sem necessidade de qualquer alteração.

Exercício 5.15 Testar e aprimorar a portabilidade e a reusabilidade das soluções com Fila. Analogamente ao Exercício 5.14, teste e aprimore a portabilidade das soluções com Fila.

Exercício 5.16 Avanço de Projeto: avaliar a portabilidade das soluções com Pilha e Fila de seus jogos. Você desenvolveu ou está desenvolvendo jogos que utilizam Pilhas e Filas. Suas soluções estão adequadamente portáveis e reutilizáveis? Analogamente ao realizado nos Exercícios 5.14 e 5.15, avalie o grau de portabilidade e o potencial para reusabilidade das soluções com Pilhas e Filas em seus jogos. Aprimore as soluções onde for necessário.

Exercício 5.17 Conclusões sobre portabilidade e reusabilidade. Que conclusões adicionais sobre portabilidade e reusabilidade de software você tirou após a realização dos Exercícios 5.14, 5.15 e 5.16?

Soluções para alguns dos exercícios

Exercício 5.2 Revisar comandos da Operação Desempilha

```
b) X = P->Info;
c) PAux = P;
d) P = P->Next;
e) delete PAux;
```

Exercício 5.4 Fila Encadeada e Dinâmica em C++

```
/* arquivo FilaEncDinamc.h - implementa um TAD Fila */
#include<conio.h>
#include<stdio.h>

struct Node {
   char Info;
   Node *Next;
};

typedef struct Node *NodePtr;

struct Fila{
   NodePtr Primeiro;
   NodePtr Ultimo;
};

void Cria(Fila *F){
     F->Primeiro = NULL;
     F->Ultimo = NULL;
  }

bool Vazia(Fila *F){
 if(F->Primeiro ==NULL)  // ou F->Ultimo==NULL
    return true;
  else
    return false;
}

bool Cheia(Fila *F){
    return false;   // veja teste na operação Insere, após alocação de memória
}

void Insere(Fila *F, char X, bool *DeuCerto){
   NodePtr PAux = new Node;
   if (PAux == NULL)
   *DeuCerto = false;   // se PAux retornar NULL, não há mais memória - fila cheia
   else { *DeuCerto = true;
   PAux->Info = X;
            PAux->Next = NULL;
            if(Vazia(F))
            F->Primeiro = PAux;
            else F->Ultimo->Next = PAux;
            F->Ultimo = PAux;
           } // else
   } // Insere

   void Retira(Fila *F, char *X, bool *DeuCerto){
      NodePtr PAux;
      if(Vazia(F)) {
      DeuCerto=false;}
      else { *DeuCerto = true;
           *X = F->Primeiro->Info;
           PAux = F->Primeiro;
           F->Primeiro = F->Primeiro->Next;
           if(F->Primeiro==NULL){   // a fila ficara vazia
           F->Ultimo = NULL; };
           delete(PAux);
          }; // else
    }; // retira
```

```
void Destroi(Fila *F){   // remove todos os nós da Fila
  char X;
  bool Ok;
  while (Vazia(F)==false) {
       Retira(F, &X, &Ok);
       } // while
} // Destroi

/* arquivo FilaEncDinamc.cpp - testa o TAD Fila */
#include "FilaEncDinamc.h"
#include<iostream>

using namespace std;

void Imprime(Fila *F){  // imprime sem abrir a TV
   char X;
   Fila *FAux = new Fila;
   bool Ok;
   Cria(FAux);
   while (Vazia(F)==false) {
        Retira(F, &X, &Ok);
        if (Ok) {
              Insere(FAux, X, &Ok);
              } // if
        } // while
   printf("\n F.Primeiro --> ");
   while (Vazia(FAux)==false) {
        Retira(FAux, &X, &Ok);
        if (Ok){
              printf("%c ", X);
              Insere(F, X, &Ok);
              } // if
        } // while
   printf(" <-- F.Ultimo \n");
}

int main(){
Fila *F = new Fila;
Cria(F);
bool Ok;
char Valor;
char Op = 't';
while (Op != 's') {
        cout << "digite: (i)inserir,(r)retirar, (s)sair [enter]" << endl;
        cin >> Op;
        switch (Op) {
         case 'i' : cout << "digite um UNICO CARACTER para inserir [enter]" << endl;
                cin >> Valor;
                Insere(F,Valor,&Ok);
                if (Ok==true) cout << "> valor inserido" << endl;
                else cout << "> nao conseguiu inserir" << endl;
                break;
         case 'r' : Retira (F,&Valor,&Ok);
                  if (Ok==true) cout << "valor retirado= " << Valor << endl;
                  else cout << "nao conseguiu retirar" << endl;
                  break;
         default : cout << "saindo... " << endl; Op = 's'; break;
         }; // case
         Imprime (F);
```

```
  } // while
    Destroi(F);       // retira todos os elementos da fila
    cout <<"a fila apos a operacao destroi... " << endl;
    Imprime(F);
    cout <<"pressione uma tecla... " << endl;
    getch();
    return(0);
}
```

Referências e leitura adicional

Celes, W.; Cerqueira, R.; Rangel, J. L. *Introdução a estruturas de dados*. Rio de Janeiro: Elsevier, 2004.
Drozdek, A. *Estrutura de dados e algoritmos em C++*. São Paulo: Thomson, 2002.
Langsam, Y.; Augenstein, M.J; Tenenbaum, A. M. *Data Structures Using C and C++*. 2nd ed. New Jersey: Prentice Hall, 1996. Upper Saddle River.
Pereira, S. L. *Estruturas de dados fundamentais: conceitos e aplicações*. São Paulo: Érica, 1996.