Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Estruturas de Dados Básicas I • IMD0029
◁ Notas de aula sobre listas encadeadas ▷
27 de outubro de 2015

Objetivos

O objetivo destas notas de aula é motivar e apresentar a estrutura de dados *lista encadeada*, enfatizando sua implementação através de classes em C++. Este documento é dividido em três partes:

- **Seção 1** que motiva o uso de listas encadeadas, comparando-a com a utilização de arranjos (lista sequencial) e demonstrando qual a relação que listas encadeadas tem com apontadores. Esta seção é uma tradução livre e adaptação do documento *"Linked List Basics"* da Universidade de Stanford [2].
- **Seção 2** que aborda os algoritmos básicos de manipulação de listas encadeadas simples com *nó cabeça* (*header*).
- **Seção 3** que descreve a implementação de lista duplamente encadeada, apresentando sua interface e quais classes devem ser construídas.

Após sua correta implementação, espera-se que a lista encadeada possa ser utilizada, por exemplo, na implementação de pilhas, filas e deques, bem como na solução de alguns problemas computacionais que requerem tal estrutura.

Sumário

1 Básico sobre Lista Encadeada

1.1 Estruturas Básicas para Lista e Sua Implementação

Listas encadeadas[1] e *arranjos unidimensionais*[2] são similares em sua funcionalidade, uma vez que ambos armazenam coleções de dados de maneira linear. É correto afirmar que vetores e listas encadeadas armazenam *elementos* em nome do código *cliente*. O tipo específico de elemento que é armazenado não é importante, uma vez que cada uma destas estruturas de dados deve ser capaz de armazenar elemento de qualquer tipo. Uma forma didática de motivar e introduzir o conceito de listas encadeadas é observar como vetores funcionam e, a partir daí, pensar em abordagens alternativas para realizar a mesma tarefa.

Revisão de Vetores

Vetores são, provavelmente, o tipo mais simples de estrutura de dados usado para armazenar uma coleção de elementos. Na maioria das linguagens, vetores são fáceis de declarar e seguem a sintaxe intuitiva de se utilizar `[ ]` e índices para diretamente acessar qualquer elemento do vetor. O exemplo a seguir demonstra um código típico de uso de vetores, juntamente com um desenho esquemático de como tal vetor poderia ser representado na memória. O código aloca um vetor de 100 elementos chamado `score`, atribui sequencialmente os valores 1, 2, 3 para os três primeiros elementos da coleção, deixando o resto do vetor não inicializado.

```
1 void ArrayTest() {
2     int scores[100];
3     // Opera sobre os elementos do vetor score...
4     scores[0] = 1;
5     scores[1] = 2;
6     scores[2] = 3;
7 }
```

Na Figura 1 temos uma desenho esquemático de como o vetor `scores` poderia ser representado na memória. O ponto principal a ser notado é que todo o vetor é alocado em um único bloco de memória. Cada elemento do vetor tem seu próprio espaço reservado. Qualquer elemento pode ser

[1] Também conhecida como *listas ligadas*.
[2] Também conhecido como *vetores*.

diretamente acessado utilizando a sintaxe `[ ]`. Também vale a pena notar que o vetor `scores` existe apenas no espaço de *memória local* (pilha de execução local) da função `ArrayTest` e, consequentemente, será desalocado automaticamente quando a execução desta função terminar.

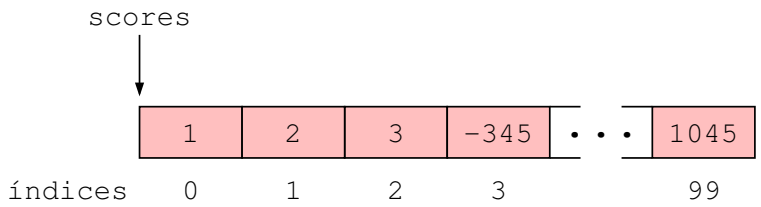


Figura 1: Representação da memória contendo o vetor `scores` de 100 elementos.

Acesso ao vetor com expressões do tipo `scores[i]` é, quase sempre, implementada utilizando aritmética de endereçamento rápido: o endereço de um elemento é computado como um *deslocamento* (*offset*) do início do vetor, o que apenas requer uma multiplicação e uma adição.

As desvantagens do uso de vetor são:

1. O tamanho do vetor é fixo — 100 elemento neste caso. A forma mais comum é especificar o tamanho do vetor em tempo de compilação como o exemplo acima. Com um pouco de esforço por parte do programador, a decisão quanto ao tamanho do vetor pode ser adiada até o vetor ser criado em tempo de execução, mas após a criação seu tamanho permanece fixo.
2. Devido a (1), a forma mais conveniente de trabalho é declarar um vetor "grande o suficiente" para a aplicação em questão. Apesar de conveniente, esta estratégia apresenta duas desvantagens: (a) na maioria das vezes o vetor vai apresentar uma taxa de ocupação média que gira em torno de 20-30%, ou seja, aproximadamente 70% do espaço reservado ao vetor é desperdiçado; (b) se o programa precisar processar mais do que 100 elementos ocorrerá uma falha grave. Um montante surpreendente de código comercial adota este estratégia ingênua de alocação de memória, consumindo mais memória do que necessário na maioria das vezes e, ocasionalmente, falhando em situações especiais que ultrapassam o tamanho original do vetor.
3. Uma consequência menos grave mas que deve ser considerada é que a inserção de novos elemento na frente do vetor. Tal situação representa uma ação computacionalmente cara, uma vez que será necessário deslocar todos os elementos armazenados no vetor para liberar espaço para a inserção dos novos elementos.

Assim como os vetores, as listas encadeadas também possuem vantagens e desvantagens, porém elas apresentam um melhor desempenho (de execução e de uso de memória) nas situações em que os vetores não têm um desempenho satisfatório. As características dos vetores mencionadas anteriormente são consequências diretas da estratégia de alocar espaço para todos os elemento em um único bloco de memória contígua. Listas encadeadas, por sua vez, utilizam uma estratégia totalmente diferente. Como veremos a seguir, listas encadeadas alocam memória para cada elemento separadamente e apenas quando necessário.

Revisão de Apontadores (Ponteiros)

Nesta seção faremos uma rápida revisão da terminologia e de algumas regras relacionadas a *apontadores*[3]. O código exemplo de lista encadeada que virá logo após esta revisão depende diretamente do bom entendimento de apontadores e suas regras de manipulação. Para obter informação mais detalhadas sobre apontadores e seu uso consulte livros de programação em C ou C++ ou então o material eletrônico da Universidade de Stanford [1].

Apontador/Alvo Um *apontador* armazena a referência para uma outra variável, algumas vezes denominada de *alvo*. Alternativamente, um apontador pode receber o valor `nullptr` ou `nullptr` que significa que o apontador, atualmente, não está associado a um alvo.

Desreferenciar A operação de desreferência sobre um apontador acessa seu alvo. Um apontador pode apenas ser desreferenciado depois ter sido associado a um alvo específico através de um comando de atribuição. Um apontador que não aponta para um alvo válido é considerado *inválido* ou *selvagem* e não deve ser desreferenciado.

Atribuição de Apontador Uma operação de atribuição entre dois apontadores na forma `p=q` faz com que os dois apontadores apontem para o mesmo alvo. Este comando não copia (i.e. duplica) a memória do alvo. Depois da atribuição ambos apontadores apontarão para a mesma região de memória correspondente ao alvo, situação essa denominada de *compartilhamento*.

`new` O comando `new` é o comando da linguagem C++ que aloca um bloco de memória no *heap* e retorna um apontador para este novo bloco. Ao contrário de variáveis locais (armazenadas na *pilha* de execução local), a memória *heap* não é automaticamente desalocada quando a função onde a mesma foi solicitada é encerrada. A forma segura de utilizar o `new` é em associação com o comando de tratamento de exceções `try{...} catch(){...}`, e a exceção que se deve tratar, em caso de falha na alocação é a `st::bad_alloc`. Para tanto o código deve incluir o cabeçalho `<new>`, seguido da indicação de uso da classe com `using std::bad_alloc;`.

`delete` O comando `delete` é o comando oposto ao `new`. Uma vez terminado o uso do bloco de memória do *heap*, executa-se o comando `delete` sobre o bloco, desta forma devolvendo tal recurso ao sistema.

Logo abaixo no Código 1 temos um exemplo de um programa em C++ que faz uso dos comandos `new`/`delete` para dinamicamente *alocar*/*desalocar* um vetor de 100 elementos.

Como São as Listas Encadeadas

Um vetor aloca memória para todos seus elemento e um único bloco contíguo de memória. Em contraste, uma lista encadeada aloca espaço (bloco de memória) para cada elemento separadamente, o qual é chamado de *elemento da lista encadeada* ou simplesmente *nó*. A estrutura de ligação da lista é realizada através de conexões implementadas com apontadores que conectam todos os nós da lista como elos em uma corrente.

[3] Também conhecidos como *ponteiros*.

Código 1 Usando os comandos `new` e `delete` para alocação dinâmica.

```
1  #include <iostream>
2  using std::cerr;
3  using std::endl;
4  #include <new> // Para capturar a exceção de falha na alocação de memória.
5  using std::bad_alloc;
6  #include <cassert> // Para usar a funcao assert()
7
8  int main( ) {
9    // Apontador usado para receber o bloco de memória do heap.
10   int *array = nullptr; // É uma boa prática fazer o apontador receber nullptr.
11   // A alocação deve se feita dentro do try{} caso exceção seja lançada
12   try { array = new int [ 100 ]; }
13   catch ( const bad_alloc& exception ) {
14     cerr << "\n[main()]:Erro durante alocacao de array[]!\n";
15     assert ( false ); // Aborta o programa; usado apenas em testes.
16   }
17
18   // Aqui vem o código que utiliza o vetor de alguma forma...
19
20   delete [] array; // Devolver a memória ao sistema.
21
22   // Término de progama normal.
23   return EXIT_SUCCESS;
24 }
```

Cada nó de uma lista encadeada *simples* contém dois campos: um campo `data` para armazenar o tipo de elemento que a lista manipula para o cliente, e um campo `next` que é um apontador usado para ligar o nó atual ao próximo. Cada nó é alocado no *heap* através de uma chamada `new`, de forma que a memória correspondente continua a existir até que seja explicitamente desalocada através de uma chamada ao comando `delete`. A frente da lista é um apontador para o primeiro nó. A Figura 2 apresenta uma representação esquemática de como uma lista contendo os números 1, 2, e 3 é representada na memória.

O desenho da Figura 2 apresenta a lista construída na memória pela função `BuildOneTwoThree()` [4]. O início da lista encadeada é armazenada em um apontador `head`, o qual aponta para o primeiro nó da lista. O primeiro nó, por sua vez, contém um apontador para o segundo nó da lista. O segundo nó contém um apontador para o terceiro nó, e assim sucessivamente. O último nó da lista tem seu campo `next` recebendo o valor `nullptr` como forma de indicar o fim da lista.

Códigos de manipulação podem acessar qualquer nó da lista apenas de forma sequencial, ou seja, começando no *apontador de início de lista* (AIL), como por exemplo `head`, e seguindo através dos apontadores do campo `next` de cada nó. Por conseguinte, operações de manipulações realizadas no início da lista são rápidas se comparadas com operações realizadas no final da lista. Este tipo de acesso, de complexidade *linear*, é mais lento do que o acesso de complexidade *constante* oferecido pelos vetores via operador `[ ]`. Neste aspecto as listas encadeadas são definitivamente menos eficientes do que vetores.

[4] Confira mais adiante em Código 2, página 8, o código fonte desta função.

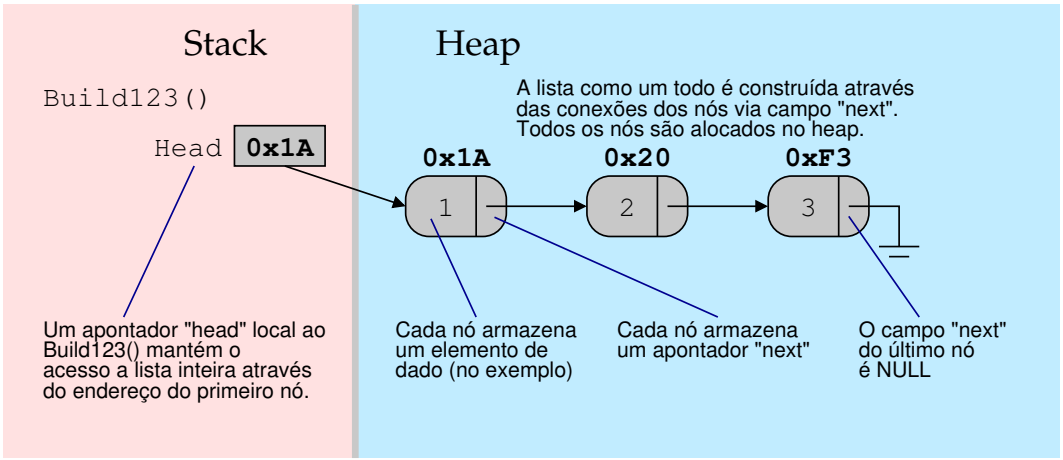


Figura 2: Representação da memória para uma lista encadeada com 3 elementos. Note a distinção entre pilha de memória local (*stack*) e a memória *heap*.

Desenhos como o exibido na Figura 2 são importantes para ajudar a "visualizar" como escrever código envolvendo apontadores, de forma que a maioria dos exemplos deste documento irá associar código com desenho de memória para enfatizar este hábito. No caso citado acima, o apontador `head` é uma variável local ordinária. Por isso `head` é desenhado separadamente do lado esquerdo, para indicar que ele é definido na pilha (*stack*) local. Os nós da lista são desenhados do lado direito para indicar que foram alocados no *heap*.

A Lista Vazia e o `nullptr`

A lista encadeada da Figura 2 apontada por `head` tem *comprimento três*, uma vez que ela é formada por três nós com o último deles apontando para `nullptr`. É necessário que exista uma representação de lista vazia — uma lista com zero nós. A forma mais comum de representar uma lista vazia é fazer o AIL (apontador de início de lista) receber `nullptr`. Esta abordagem cria, como consequência, uma necessidade de verificar se uma lista está vazia antes de efetuar qualquer operação de manipulação, caso contrário poderemos incorrer no erro clássico de *falha de segmentação* (*segmentation fault*). Portanto, ao desenvolver código de manipulação para lista encadeada simples (sem nó cabeça, discutido mais adiante) é um bom hábito lembrar de verificar o caso especial de lista vazia para ter certeza que o algoritmo irá funcionar corretamente em todos os casos. Em algumas situações o caso da lista vazia é resolvido da mesma forma que para os outros casos, mas às vezes ele requer a inclusão de código de caso especial.

Tipos para Listas Encadeadas: Nós e Apontadores

Antes de introduzir o código que constrói a lista da Figura 2, é necessário definir dois tipos de dados:

Nó O tipo que representa os nós que constituirão o corpo da lista encadeada. Cada nó, alocado no

heap, contém um único elemento de dado do cliente e um apontador para o próximo nó da lista.

```
1 struct node {
2     int           data; // Vamos começar com uma lista de inteiros.
3     struct node * next;
4 };
```

Apontador para Nó O tipo que representa os apontadores para nós. Este é o tipo do AIL e dos campos `next` dentro de cada nó. Em C++ nenhuma declaração de tipo separada é necessária, já que o tipo apontador para nó é obtido declarando-se o tipo `node` seguido por um '`*`'. Uma forma de reduzir o tamanho da referida declaração de um apontador, que seria

```
1 struct node* head;
```

é utilizar o comando `typedef` (definição de tipo) da seguinte forma:

```
1 typedef struct node * nodePtr; // Definindo 'apelido' p/ o novo tipo.
2 nodePtr head; // Mesmo que: struct node* head;
```

A única desvantagem de usar `typedef` é que ela "esconde" o '`*`' da declaração, o que pode confundir o programador iniciante fazendo-o acreditar que não se trata de um apontador.
Já em C++, ao declarar uma estrutura (com `struct`) ou uma classe (com `class`), pode-se utilizar o identificador como se fosse um novo tipo. Portanto, um apontador para nó em C++ seria declarado da seguinte forma:

```
1 node * newPt; // 'node' é um novo tipo em C++
```

Função `BuildOneTwoThree()`

Logo abaixo no Código 2 está o código fonte de uma função simples que utiliza operaçõesV com apontadores para construir a lista $\{1, 2, 3\}$. O desenho da memória apresentado na Figura 2 corresponde ao estado da memória ao término da execução da função `BuildOneTwoThree()`. Esta função demonstra como chamadas ao comando `new`[5] e atribuição de apontadores (`=`) trabalham para construir a estrutura de nós ligados no *heap*.

Exercício

Escreva uma versão alternativa para `BuildOneTwoThree()` com o menor número de atribuições (`=`) ainda capaz de gerar a mesma estrutura de memória.

1.1.1 A Função `Length()`

A função `Length()`, listada no Código 3, recebe uma lista encadeada e computa o número de elementos armazenados na lista. `Length()` é uma função de lista simples, mas ela demonstra

[5] Na maioria dos códigos exemplo deste documento omitimos, por questão de espaço, o uso de `try`/`catch` para tratar a exceção `bad_alloc` que pode ser lançada pelo `new`, mas esta é uma boa prática de programação que deve ser observada sempre.

Código 2 Código da função `BuildOneTwoThree()`.

```
/*
   Constrói a lista {1,2,3} no heap e armazena seu ponteiro head na pilha
   de variáveis locais. Retorna o apontador head para quem invocou.
*/
struct node* BuildOneTwoThree() {
    typedef struct node * nodePtr; // Definindo 'apelido' p/ o novo tipo.
    struct node* head   = nullptr; // Usando estilo C de declaração.
    node*        second = nullptr; // Usando estilo C++ de declaração.
    nodePtr      third  = nullptr; // Usando nodePtr definido com 'typedef'.

    head   = new node; // aloca 3 nós no heap
    second = new node;
    third  = new node;

    head->data = 1; // configura o primeiro nó
    head->next = second; // note: regra de atribuição de apontadores

    second->data = 2; // configura o segundo nó
    second->next = third;

    third->data = 3; // configura o terceiro nó
    third->next = nullptr;

    // Neste ponto, a lista encadeada é referenciada por "head"
    // conforme desenho esquemático anterior
    return head;
}
```

vários conceitos que serão utilizados posteriormente em funções mais complexas de manipulação de listas.

Código 3 Código da função `Length()` para calcular o comprimento da lista.

```
/*! Calcula o comprimento de uma lista passada como parâmetro.
 *  @param head O apontador para o primeiro nó da lista.
 *  @return Inteiro positivo correspondente ao tamanho da lista.
 */
int Length( node* head ) {       // Apontador de início da lista (AIL).
    node* current = head;        // Apontador que vai percorrer a lista.
    int count = 0;               // Inicializar contador de elementos.
    while ( current != nullptr ) // Enquanto não chegar ao final...
    {
        count++;                 // ... conte e...
        current = current->next; // ... mova para frente um nó.
     }
    return count;                // Retorne a contagem.
}
```

Existem duas características típicas de uma lista encadeada demonstrada em `Length()`:

1. **Passe a lista inteira por argumento enviando apenas o apontador para o seu início.** A

lista inteira é passada como argumento para `Length()` através de um único apontador — o apontador de início de lista ou AIL. O apontador original é copiado do código cliente (módulo que invocou `Length()`) para a variável `head`, local ao `Length()`. Copiar este apontador **não duplica a lista inteira**. Esta ação apenas copia o apontador de maneira que tanto o código cliente quanto o `Length()` possuem apontadores para a mesma estrutura de lista. Esta é a clássica situação de *compartilhamento* de memória comum em código com apontadores. Ambos o código cliente e o `Length()` tem cópias do AIL, mas eles compartilham a estrutura de nós alocadas à lista.

2. **Itere sobre a lista com um apontador local.** O código para *iterar sobre* (*percorrer* ou *varrer*) todos os elementos da lista é um *idioma* bem comum em código de lista encadeada:

```
1 node* current = head;
2 while (current != nullptr) {
3     // Faz algo com o nó '*current'
4     current = current->next; // Avança p/ próximo nó da lista
5 }
```

Os pontos de destaque do Código 3 são:

(a) **Linha 6.** O apontador local, `current` no exemplo, inicia apontando para o mesmo nó que o apontador `head` através do comandos de atribuição `current=head`. Quando a função termina, `current` é automaticamente desalocado, uma vez que ele é apenas uma variável local ordinária, mas os nós alocados no *heap* permanecem.

(b) **Linha 8.** O laço `while` testa se alcançou o fim da lista com `(current != nullptr)`. Este teste naturalmente captura o caso especial de lista vazia — `current` já vai ser `nullptr` logo de cara e o laço irá terminar antes mesmo de começar.

(c) **Linha 11.** O comando `current = current->next` presente no final do laço avança o apontador local para o próximo nó da lista. Quando não houver mais ligações a seguir, este comando faz `current` apontar para `nullptr`. Se por acaso algum código de lista encadeada que você estiver desenvolvendo entrar em laço infinito, possivelmente o problema será o esquecimento deste comando de avanço na lista.

Invocando o `Length()`

Aqui está um código cliente típico que invoca `Length()`. Primeiramente invocamos `BuildOneTwoThree()` para criar e armazenar a lista em um AIL que é uma variável local. Então a função invoca `Length()` sobre a lista e recupera o resultado em uma variável local.

Desenhos da Memória

A melhor forma de realizar o *design* e visualizar código para listas encadeadas é usar desenhos para ver como as operações com apontadores estão atuando sobre a memória. Os desenhos que vêm a seguir reportam o estado da memória antes e durante a invocação da função `Length()` — aproveite a oportunidade para praticar observando a representação da memória através dos desenhos de forma

Código 4 Código da função `LengthTest()` que calcula o comprimento da lista criada com a função `BuildOneTwoThree()`.

```
1 void LengthTest( ) {
2     node* myList = BuildOneTwoThree( );
3     int len = Length( myList ); // resultado é len == 3
4 }
```

a melhorar seu conhecimento sobre apontadores. Adotando o hábito simples de desenhar a memória para entender o resultado das operações com apontadores você será capaz de desenvolver códigos de manipulação de apontadores mais complexos e corretos.

Começe com os códigos do `Length()` e `LengthTest()` e uma folha de papel em branco. Desenhe ao longo da execução dos códigos, atualizando seu desenho para refletir o estado da memória em cada passo. O desenho da memória deve distinguir memória *heap* da pilha local de memória (*stack*). Lembrete: o comando `new` aloca memória no *heap* que apenas pode ser desalocada através de uma chamada explicita do comando *delete*. Em contraste, as variáveis da pilha local para cada função são automaticamente alocadas quando a função é iniciada e desalocadas quando a função termina. Nossos desenhos exibem a pilha de memória local do cliente acima da pilha de memória local da função invocada, mas qualquer convenção que você adotar estará correta contanto que você se dê por conta de que o cliente e o invocado possuem pilhas de memória local distintas[6].

Desenho 1: Antes da chamada à `Length()`

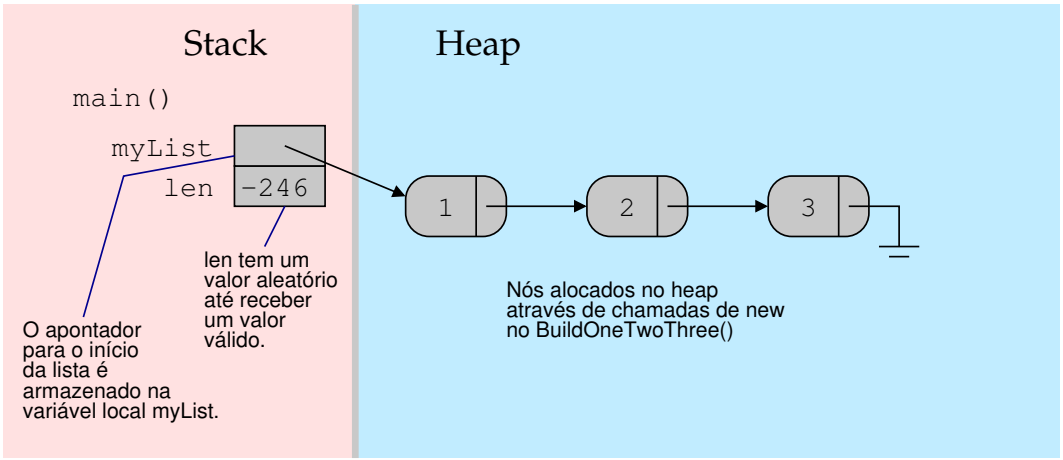
Na Figura 3-(a) temos o estado da memória antes de invocar a função `Length()` dentro de `LengthTest()` (Código 4). Neste ponto `BuildOneTwoThree()` já montou a lista $\{1, 2, 3\}$ no *heap* e retornou o AIL. Tal apontador foi recebido pela função `LengthTest()` e armazenado na sua variável local `myList`. A variável local `len` apresenta um valor aleatório — ela apenas receberá o valor 3 quando a chamada à função `Length()` retornar.

Desenho 2: No meio da execução de `Length()`

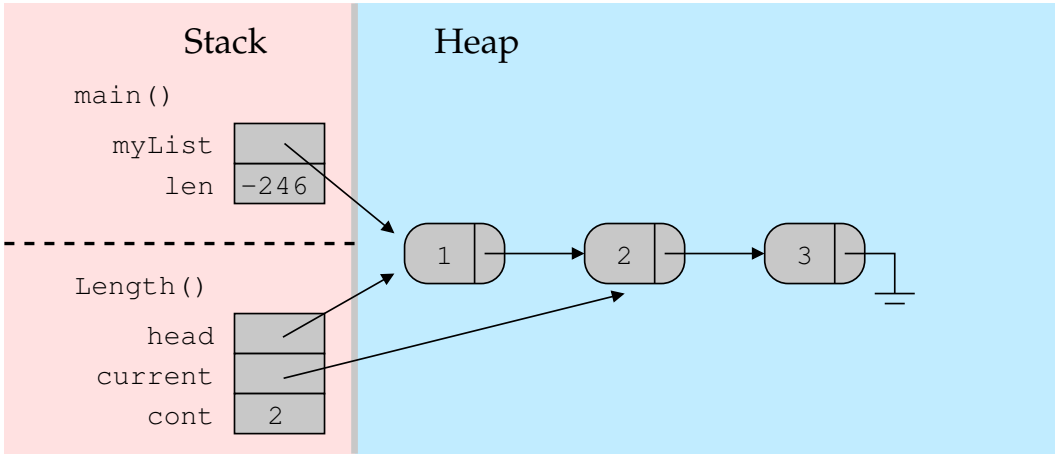
Na Figura 3-(b) temos o estado da memória na metade da execução da função `Length()`. As variáveis locais à `Length()`, `head` e `current`, foram automaticamente alocadas. O apontador `current` iniciou apontando para o primeiro nó da lista, e a primeira iteração do laço `while` o fez avançar para o segundo nó.

Note como as variáveis locais ao `Length()` (`head` e `current`) são separadas (linha pontilhada) das variáveis locais de `LengthTest()` (`myList` e `len`). As variáveis locais `head` e `current` irão ser desalocadas automaticamente quando `Length()` encerrar sua execução. Isto não representa problema — as ligações alocadas no *heap* permanecerão apesar dos apontadores que apontavam para lista serem desalocados.

[6] Consulte o documento "Pointers and Memory" [1] para maiores detalhes.



(a)



(b)

Figura 3: Representação da memória: (a) antes da chamada de `Length()`; e (b) durante a execução de `Length()`.

Exercício

Pergunta 1: Suponha que façamos `head = nullptr` no final de `Length()` — isto iria prejudicar a variável `myList` da função `LengthTest()`?
Resposta: *Não.* `head` *é uma variável local que foi inicializada com uma* cópia *do parâmetro real, portanto mudanças na cópia não afetam o parâmetro original. Em resumo, mudanças feitas nas variáveis locais de uma função não afetam as variáveis locais de outra função.*

Pergunta 2: Se a lista passada como argumento para `Length()` não contivesse elementos, será que a função `Length()` conseguiria lidar com este caso especial?
Resposta: *Sim. A representação de lista vazia seria um apontador de início de lista com valor* `nullptr`. *Simule a execução de* `Length()` *para este caso e descubra o que acontece.*

1.2 Construção Básica de Listas

A função `BuildOneTwoThree()` é um bom exemplo didático de manipulação de apontadores, mas não é um mecanismo geral para construir listas de tamanho indefinido. A melhor solução será uma função independente que adiciona um único nó a uma lista qualquer. Assim poderemos invocar esta função quantas vezes forem necessárias para construir uma lista da forma que desejarmos. Antes de entrar em detalhes de codificação, podemos identificar os típicos *3-Passos De Ligação Em Lista*, que adicionam um único nó à frente de uma lista encadeada. Os 3 passos são:

1. **Alocar Novo Nó:** Alocar o novo nó no *heap* e atribuir ao seu campo `data` o valor específico que o nó deve armazenar.

```
1 node* newNode; // Declara
2 newNode = new node; // Aloca
3 newNode->data = dado_que_o_cliente_deseja_armazenar; // Atribui
```

2. **Ligar ao Próximo:** Atribuir valor ao campo `next` do novo nó de forma a fazê-lo apontar para o nó atualmente na frente da lista. Isto na verdade é apenas uma atribuição de apontadores — lembre-se *"atribuir um apontador para outro faz com que ambos apontem para o mesmo lugar."*

```
1 newNode->next = head;
```

3. **Atualizar o Apontador de Início da Lista:** Faça o AIL apontar para o novo nó, de forma que o novo nó passa a ser o primeiro nó (frente) da lista.

```
1 head = newNode;
```

Os 3-Passos De Ligação Em Lista em Código

A função simples `LinkTest()` demonstra os *3-Passos De Ligação Em Lista*:

Código 5 Código da função `LinkTest()` que segue os *3-Passos De Ligação Em Lista*.

```
1 void LinkTest( ) {
2     node* head = BuildTwoThree(); // Constrói uma lista {2,3}
3     node* newNode;                // Apontador para novo nó
4     newNode = new node;           // 1) Alocar
5     newNode->data = 1;            // Campo de dados
6     newNode->next = head;         // 2) Ligar ao Próximo
7     head = newNode;               // 3) Atualizar Início da Lista
8     // Agora head aponta para a lista {1, 2, 3}
9 }
```

Os 3-Passos De Ligação Em Lista em Desenho

O desenho da Figura 4 representa a memória no final de `LinkTest()`; apontadores que tiveram seus valores sobrescritos são representados por setas pontilhadas na cor cinza.

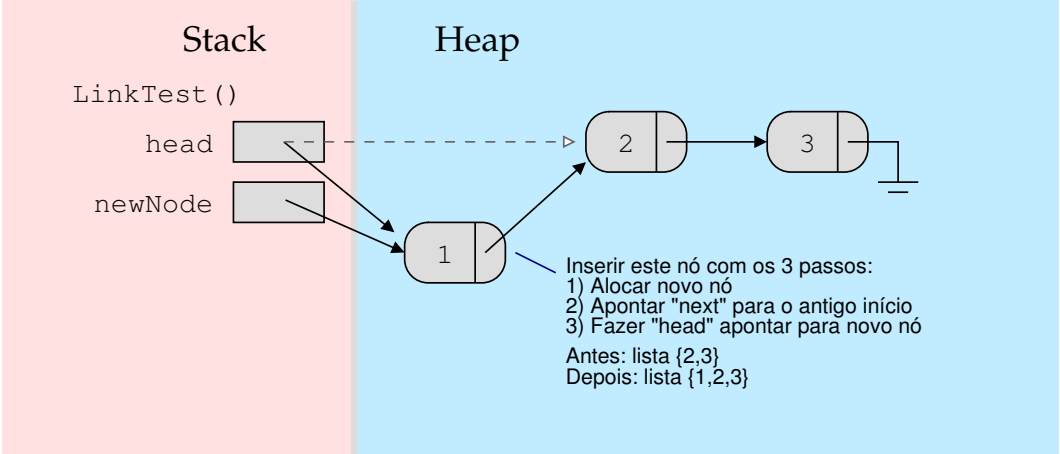


Figura 4: Representação da memória no final da execução de `LinkTest()`.

1.2.1 A Função `Push()`

Com os *3-Passos De Ligação Em Lista* na cabeça, o desafio é escrever uma função geral que adicione um único nó no início da lista, isto é, fazer o apontador de início de lista apontar para o nó recém inserido. Historicamente, esta função é chamada de `Push()`, uma vez que estamos adicionando a ligação no início da lista, o que parece conferir a lista um "estilo" de pilha. Alternativamente esta função poderia ser chamada de `InsertAtFront()`, mas iremos utilizar o nome `Push()`.

`WrongPush()`

Infelizmente `Push()` escrito em C++ apresenta um problema básico: quais deveriam ser os parâmetros para `Push()`? Esta é, lamentavelmente, uma área difícil para iniciantes em C++. Existe uma versão natural de codificação para o `Push()` que normalmente dá ao programador a falsa impressão de que funcionará corretamente mas tal versão comumente apresenta problemas fundamentais relacionados com passagem de apontadores por referência. Investigar porque esta versão inicial não funciona é uma boa desculpa para pratricar mais a elaboração de desenhos esquemáticos de memória, bem como motivar uma solução correta e tornar você um programador melhor.

`WrongPush()` está muito perto da versão correta. Ela executa os *3-Passos De Ligação Em Lista* e deixa a lista, enquanto dentro de `WrongPush()`, na forma correta. O problema ocorre na linha 5, ou seja, o último dos 3 passos que requer a atualização do AIL de forma a fazê-lo apontar para o novo nó. O que você acha que o comando `headCopy = newNode` realiza em `WrongPush()`? De fato ele atualiza um AIL, mas não o AIL correto! Perceba que `headCopy` é um AIL **local** à `WrongPush()`, ou seja, atualizar o AIL local **não** faz com que o AIL externo (o `head` declarado localmente em `WrongPushTest()`) também seja atualizado.

Código 6 Código da função `WrongPush()` que **não** consegue inserir um elemento na lista encadeada por não passar "head" por referência.

```
1  void WrongPush( node* headCopy, int data ) {
2      node* newNode = new node;
3      newNode->data = data;
4      newNode->next = headCopy;
5      headCopy = newNode; // ATENÇÃO! Esta linha não funciona
6  }
7  void WrongPushTest() {
8      node* head = BuildTwoThree( );
9      WrongPush( head, 1 ); // Tentando inserir 1, mas não funciona!
10 }
```

Exercício

Faça o desenho de memória correspondente a execução passo a passo do `WrongPushTest()` para verificar porque ele não funciona corretamente. Lembre-se que variáveis locais ao `WrongPush()` e `WrongPushTest()` são separadas (cada uma é definida em uma pilha local de memória).

Parâmetros por Referência em C/C++

O problema anterior representa uma "característica" básica da linguagem C/C++, que é o fato de que mudanças feitas nos parâmetros locais não são nunca refletidas na memória da função cliente. Esta é uma área tradicionalmente capciosa em programação C/C++. Vamos apresentar duas soluções para este problema, a primeira serve tanto para programas em C quanto em C++, a segunda faz uso de uma característica presente apenas em C++.

O solução para o problema anterior consiste em fazer com que a versão correta `Push()` seja capaz de alterar a memória local do cliente — mais especificamente a variável AIL local `head` de `PushTest()`. A forma tradicional de permitir a alteração de variáveis locais ao cliente é passar um apontador para a memória local ao invés de uma cópia — isto é denominado, respectivamente, passagem de parâmetros por *referência* e por *valor*. Assim, para modificar uma variável tipo '`int`' do cliente passamos seu endereço para um parâmetro '`int *`'. Para modificar um '`struct funcionário`' passamos um '`struct funcionário *`'. Para modificar um tipo '`X`' passamos '`X *`' e assim por diante (você entendeu a ideia...). No nosso caso em particular, o valor que queremos que seja modificado é '`struct node *`', logo devemos passar um '`struct node **`'. É apenas um detalhe o fato de que o valor que desejamos modificar já tem um asterisco (`*`), e é por isso que o parâmetro capaz de modificá-lo deve ter dois asteriscos (`**`). De outra maneira: o tipo do AIL é *"apontador para uma estrutura do tipo `node`"*; portanto, para modificar AIL precisamos passar um apontador para seu tipo, o que é descrito como *"um apontador para um apontador para uma estrutura do tipo `node`"*. Faça uso de sua capacidade de *abstração* para entender e estender o conceito de passagem de parâmetro por referência para tipos apontadores.

Portanto, ao invés de definir

```
1  WrongPush( node* head, int data )
```

devemos definir

```
1 Push( node** headRef, int data ).
```

A primeira versão passa uma cópia do AIL, enquanto que a segunda forma corretamente passa um apontador para o AIL. A regra é: modificar a memória do cliente, passando um apontador para sua memória. O parâmetro tem o sufixo `ref` para nos lembrar de que é um *referência* para um AIL (`struct node**`) ao invés de um AIL ordinário (`struct node *`) que apenas resultaria em uma cópia do AIL.

Versão Correta do `Push()`

Segue abaixo em Código 7 o código fonte das funções `Push()` e `PushTest()` que corretamente demonstram a ideia de inserção geral em uma lista. A lista é passada como argumento através de um apontador para o AIL. No código, isto é refletido de duas formas: (1) através do uso do operador '`&`' na frente do argumento na chamada de `Push()` (linhas 14 e 15), e; (2) através do uso de um '`*`' extra no parâmetro correspondente em `Push()` (linhas 6, 9 e 10). Dentro de `Push()`, o apontador para AIL é denominado de `headRef` ao invés de apenas `head` como uma forma de lembrar que não se trata de um simples apontador.

Código 7 Código da função `Push()` e `PushTest()` que corretamente demonstram a inserção de um nó em uma lista.

```
1  /*! Recebe uma lista e um valor de dado a ser armazenado na lista.
2      @param headRef Referência para AIL, necessário para alterar
3                     lista externamente.
4      @param data Dado a ser inserido.
5  */
6  void Push( node** headRef, int data ) {
7      node* newNode = new node; // Alocação normal.
8      newNode->data = data;
9      newNode->next = *headRef; // O '*' é usado p/ desreferenciar o ...
10     *headRef = newNode;       // verdadeiro AIL.
11 }
12 void PushTest( ) {
13     node* head = BuildTwoThree(); // Cria e retorna lista {2, 3}
14     Push( &head, 1 );                    // Perceba o uso de '&'
15     Push( &head, 13 );                   // Segunda chamada à Push().
16     // "head" agora aponta para lista {13, 1, 2, 3}.
17 }
```

Desenho do `Push()` Correto

Na Figura 5 demos o desenho representativo da memória antes do final da chamada à `Push()`. O valor original do apontador `head` é representado com uma seta pontilhada. Note que o parâmetro `headRef` dentro de `Push()` aponta de volta para o AIL original `head` definido em `PushTest()`. `Push()` utiliza `*headRef` para acessar e modificar o valor do original AIL `head`.

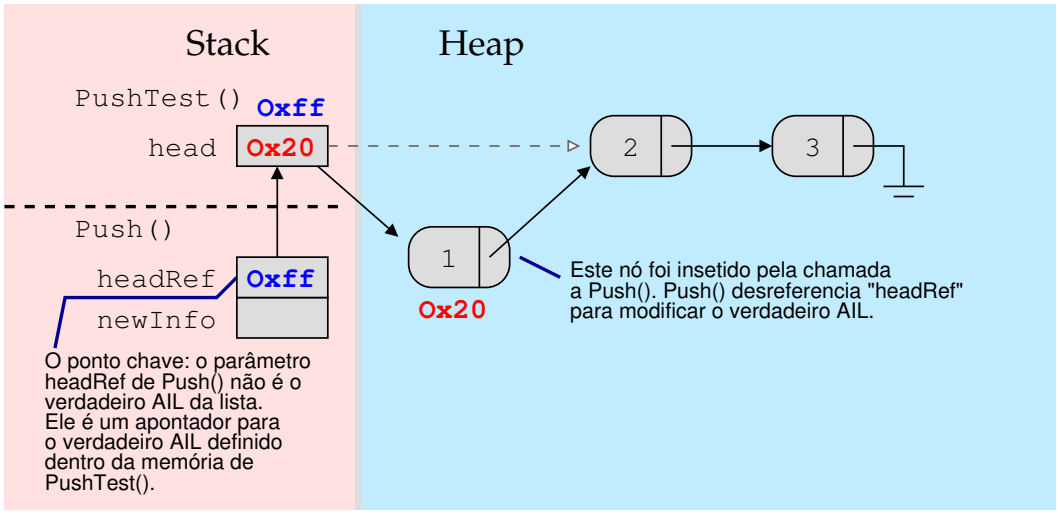


Figura 5: Representação da memória no final da execução de `Push()`.

Exercício

O desenho da Figura 5 apresenta o estado da memória no final da execução da primeira chamada à `Push()` em `PushTest()`. Estenda o desenho para acompanhar uma segunda chamada à `Push` (confira a linha 15 do Código 7, na página 15). O resultado deve produzir a seguinte lista $\{13, 1, 2, 3\}$.

Exercício

A função apresentada no Código 8 logo abaixo corretamente constrói uma lista com três elementos usando nada mais do que chamadas sucessivas à `Push()`. Faça desenhos da memória representado sua execução e mostre o estado final da lista. Esta situação também demonstra que `Push()` funciona corretamente para o caso especial de lista vazia.

Código 8 Código da função `PushTest2()` que cria uma lista com 3 elementos.

```
1 void PushTest2( ) {
2     node* head = nullptr; // Cria uma lista vazia.
3     Push( &head, 1 );
4     Push( &head, 2 );
5     Push( &head, 3 );
6     // head agora deve apontar para a lista {3, 2, 1}
7 }
```

E Como Ficaria no C++?

A linguagem C++ oferece, em sua definição, a possibilidade de criar argumentos por referência, de forma a facilitar a vida dos programadores. Assim, basta acrescentar o operador '`&`' na frente do tipo de parâmetro que se deseja passar por referência e *voilà*, temos um parâmetro por referência!

A principal vantagem é que no código cliente não é mais necessário acrescentar o '`&`' nem o '`*`' na frente do parâmetro dentro do código desenvolvedor (o invocado). Assim, as versões das funções `Push()` e `PushTest()` que utilizam esta facilidade são apresentadas no Código 9.

Código 9 Código da função `PushTest()` e `Push()` utilizando parâmetros por referência.

```
/*! A função Push() difere da versão anterior apenas pelo acréscimo
    de '&' à direita do tipo do parâmetro ''head''.
    Isto faz com que o compilador torne ''head'' uma referência
    para o valor original.
    Não é necessário usar '*' extras em Push().
*/
void Push( node * & head, int data ) {
    node* newNode = new node;
    newNode->data = data;
    newNode->next = head; // Nenhum '*' extra é necessário no ''head''...
    head = newNode;       // o compilador faz isso de forma transparente.
}
void PushTest() {
    node* head = BuildTwoThree(); // Cria lista {2, 3}.
    Push( head, 1 ); // Nenhum '&' extra é necessário, o compilador
    Push( head, 13 ); // também toma conta disto. ''head'' está sendo
    // modificado por estas duas chamadas.
    // Agora ''head'' aponta para a lista {13, 1, 2, 3}
}
```

O desenho de memória para esta versão em C++ é o mesmo para a versão em C. A diferença é que na versão C, os `*`s devem ser acrescentados ao código. No caso da versão C++ isso é feito transparentemente pelo compilador.

1.3 Técnicas de Codificação para Listas Encadeadas

Esta seção resume, na forma de uma enumeração, as principais técnicas de manipulação de listas introduzidas na Seção 1.1 e Seção 1.2.

1. **Iterar (Percorrer) ao Longo de Uma Lista**
   Uma técnica frequentemente utilizada em código de lista encadeada é iterar (i.e. percorrer) uma lista utilizando um apontador que aponta, sucessivamente, para cada um dos nós da lista. Tradicionalmente, isto é escrito dentro de uma estrutura de laço `while`. O AIL é copiado em uma variável local `current` a qual é utilizada para iterar ao longo da lista. O teste para saber se o final da fila foi alcançado é `current != nullptr`. O avanço é realizado com o comando `current = current->next`.

```
// Retorna o número de nós de uma lista (versão com while).
int Length( node* head ) {
    int count = 0;
    node* current = head;
    while ( current != nullptr ) { // Poderia ser: while ( current )
        count++;
        current = current->next
```

```
8       }
9       return( count );
10  }
```

Alternativamente, algumas pessoas preferem escrever o laço com `for` ao invés de `while`, concentrando em uma única linha a inicialização e o avanço de apontador:

```
1  for ( current = head; current != nullptr; current = current->next ) {
```

2. **Modificar um Apontador Através de um Apontador Por Referência**
   Muitas funções de lista precisam atualizar o AIL original. Para realizar isto na linguagem C/C++, precisamos passar um apontador para o AIL. Tal apontador para apontador é, às vezes, chamado de *apontador por referência*. Os principais passos desta técnicas são:

   - Faça a função receber um apontador para o AIL. Isto é a técnica padrão do C/C++ — passar um apontador para o *valor de interesse* que precisa ser modificado. Assim, para modificar um `struct node *`, devemos passar um `struct node **`.
   - Usar '`&`' na frente do argumento na chamada da função.
   - Use '`*`' na frente do parâmetro dentro da função invocada para acessar e modificar o valor de interesse.

   A simples função descrita a seguir faz o AIL apontar para `nullptr` utilizando a técnica de *apontador por referência*.

```
1  // Modifica o parâmetro ``head'', atribuindo-lhe nullptr.
2  // Usa um apontador por referência p/ acessar apontador original.
3  void ChangeToNull( node** headRef ) { // Apontador para o
4                                        // valor de interesse
5      *headRef = nullptr; // Usa '*' para acessar o valor de interesse.
6  }
7  void ChangeCaller( ) {
8      node* head1;
9      node* head2;
10     ChangeToNull( &head1 ); // Usa '&' p/ computar/passar um apontador
11     ChangeToNull( &head2 ); // para o valor de interesse.
12     // ``head1'' e ``head2'' apontam para nullptr.
13 }
```

   Abaixo na Figura 6 temos o desenho que representa como o apontador `headRef` em `ChangeToNull()` aponta de volta para a variável original na função cliente.

3. **Construir — No Início com o `Push()`**
   A maneira mais fácil de construir uma lista é inserir novos nós no início da lista com a função `Push()`. O código é curto e eficiente — listas naturalmente suportam operações realizadas em seu início. A desvantagem é que os elemento aparecerão na lista na ordem inversa em que foram inseridos. Se a ordem é irrelevante para aplicação em questão, então a inserção no início é a melhor escolha, de acordo com os motivos mencionados.

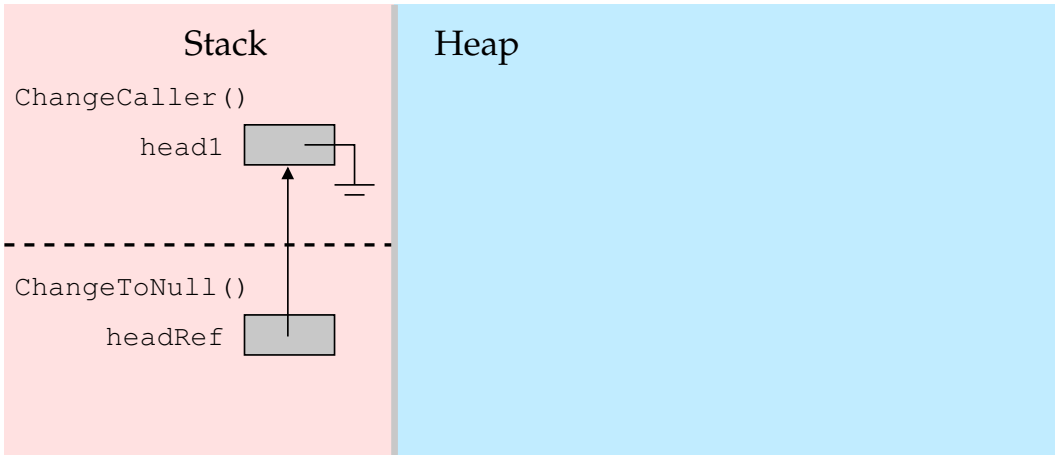


Figura 6: Representação da memória para a execução de `ChangeToNull()`.

```
1  node* AddAtHead( ) {
2      node* head = nullptr;
3      int i;
4      for ( i = 1; i < 6; i++ ) {
5          Push( &head, i );
6      }
7      // head == {5, 4, 3, 2, 1};
8      return( head );
9  }
```

4. **Construir — No Final com um Apontador Calda**

   E o que podemos dizer sobre adicionar nós no final da lista? Adicionar nós no final quase sempre requer a localização do último nó da lista, para então modificar seu campo `next` de `nullptr` para o novo nó a ser inserido. Este procedimento é demonstrado através da variável apontadora `tail` na Figura 7.

   Este é apenas um caso especial de uma regra geral: para inserir ou remover um certo nó X dentro de uma lista, é necessário um apontador para o nó imediatamente *anterior* à posição de X, de forma que seja possível modificar seu campo `next`. Muitos problemas de lista incluem o sub-problema de avançar um apontador até o nó imediatamente anterior ao ponto de inserção ou remoção. A única exceção acontece se o nó que sofrerá a operação é o primeiro da lista — neste caso o próprio AIL deve ser modificado. Os exemplos a seguir demonstram várias formas para manipulação sobre o primeiro nó da lista e sobre os demais nós do interior da lista.

5. **Construir — Caso Especial + Apontador Calda**

   Considere o problema de construir a lista $\{1, 2, 3, 4, 5\}$ através da inserção de nós na calda da lista. A dificuldade reside no fato que o primeiro nó deve ser inserido no AIL, mas todos os demais devem ser inseridos após o último nó utilizando o *apontador para calda da lista* (ACL) — no exemplo `tail`. A forma mais simples de lidar com ambos os casos é escrever o código

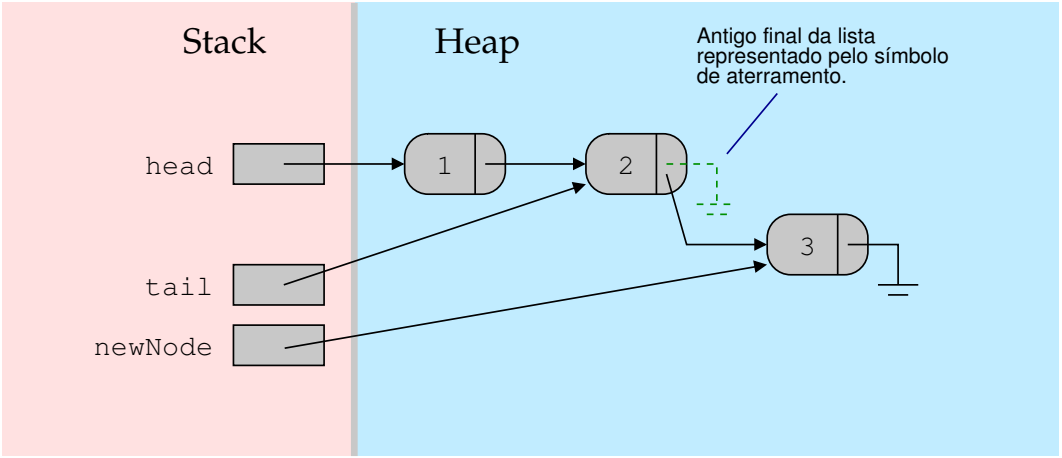


Figura 7: Representação da memória para adicionar um elemento no final de uma lista com a utilização de um apontador `tail`.

com tratamento separado para cada caso. O caso especial trata a inserção do primeiro nó no AIL, formando a lista $\{1\}$. Então existe um laço separado para lidar com o caso trivial, que utiliza um ACL para inserir os demais nós. O ACL é mantido atualizado, isto é, sempre apontando para o último nó, e cada nova inserção é realizada em `tail->next`. O único "problema" com esta solução é que escrever código para o caso especial do primeiro nó é um pouco insatisfatório. Apesar disto, esta abordagem é adequada para código funcional — é simples e eficiente.

```
1  node* BuildWithSpecialCase( ) {
2      node* head = nullptr;
3      node* tail;
4      int i;
5      // Lida com o caso especial e configura o ACL.
6      Push( &head, 1 );
7      tail = head;
8      // Resolve os demais casos fazendo uso do apontador ACL ("tail").
9      for ( i = 2; i < 6; i++ ) {
10         Push( &( tail->next ), i ); // adiciona nó em tail->next
11         tail = tail->next; // avança "tail", aponta p/ último nó
12     }
13     return( head ); // head == {1, 2, 3, 4, 5};
14 }
```

6. **Construir — Nó Dummy**

   Uma outra solução consiste em utilizar um nó temporário denominado de *dummy*[7], no início da lista. O truque é que com a existência deste nó na lista ela nunca estará totalmente vazia; desta forma elimina-se o caso especial de inserir pela primeira vez na lista. Portanto, toda e qualquer inserção é realizada como se já existisse um elemento na lista, ou seja, ficamos apenas com os caso trivial mencionado no item anterior. Assim o código para o primeiro nó

[7] Este nó não armazena valor válido para a lista.

é unificado com o código que lida com os demais nós, e assim todos as inserções são feitas depois de um campo `next` de um nó. O ACL, `tail`, tem o mesmo papel que no exemplo anterior. A diferença agora é que `tail` também é usado para a inserção do primeiro nó.

```
1  node* BuildWithDummyNode( ) {
2      node dummy; // Dummy é o nó temporário para o primeiro nó
3      node* tail = &dummy; // Inicializa tail para o dummy.
4      // Constrói a lista iniciando em dummy.next (igual a tail->next)
5      int i;
6      dummy.next = nullptr;
7      for ( i = 1; i < 6; i++ ) {
8          Push( &( tail->next ), i );
9          tail = tail->next;
10      }
11     // A lista criada, sem dummy, está em dummy.next
12     // dummy.next == {1, 2, 3, 4, 5};
13     return( dummy.next );
14 }
```

Algumas implementações de lista encadeada mantém o nó *dummy* permanentemente como parte da lista (é o que veremos, por exemplo na Seção 2 mais adiante). Para esta estratégia de "uso permanente de dummy"[8], a lista vazia não é representada por um apontador `nullptr`. Ao invés disto, toda lista tem um nó `dummy` no seu início, apontado por AIL. Portanto, uma lista vazia passa a ser representada por um AIL apontando para um nó *dummy* cujo campo `next` aponta para `nullptr`. Consequentemente, todos os algoritmos de manipulação devem saltar o nó *dummy*.

A solução de *dummy*-temporário demonstrada acima é um pouco incomum, mas evita que o *dummy* seja um integrante permanente da lista. Algumas soluções apresentadas neste documento fazem uso da estratégia de um nó *dummy* temporário, enquanto que outros exemplo, principalmente da Seção 2 em diante, fazem uso de um nó *dummy* permanente, doravante denominado de **nó-cabeça**.

7. **Construir — Com Referência Local para Apontador**

   O último mecanismo de criação de lista faz uso de referência local para um apontador e é ainda mais incomum que a versão anterior. É uma forma um pouco mais intrincada de unificar todos os casos de nós sem utilizar um nó *dummy*. O truque consistem em utilizar uma variável local *referência para apontador* (i.e. apontador para apontador) que sempre **aponta para o último apontador** na lista ao invés do último nó. Todas as inserções na lista são realizadas seguindo-se a tal referência para apontador. A referência para apontador local é inicializada com AIL. Posteriormente, ela aponta para o campo `next` definido no último nó da lista (perceba a diferença, ela aponta para o ponteiro `next` e não para o `node`!).

```
1  node* BuildWithLocalRef( ) {
2      node* head = nullptr;
3      node* &lastPtrRef = head; // Inicia apontado para o AIL.
4      int i;
```

[8] Neste caso, o nó *dummy* é chamado de *nó-cabeça*, por ser o 'cabeça' da lista.

```
5       for ( i = 1; i < 6; i++ ) {
6           Push( lastPtrRef, i ); // Insere nó no último apontador lista.
7           lastPtrRef = lastPtrRef->next; // Avance de forma a
8           // apontar para o "novo" apontador calda.
9       }
10      // head == {1, 2, 3, 4, 5};
11      return( head );
12  }
```

Esta técnica tem um código ainda mais curto, não desperdiça memória com o nó *dummy* extra *mas*... a parte interna do laço é assustadora! Mas não se preocupe pois ela é raramente utilizada e aparece aqui apenas para demonstrar quão interessante e versátil o uso de apontador pode ser. Veja a lógica da sua utilização.

- O apontador `lastPtrRef` sempre aponta o último **apontador** da lista. Inicialmente, na linha 3, ele aponta para o AIL. Posteriormente irá apontar para o campo `next` dentro do último nó da lista (linha 7).
- `Push( lastPtrRef, i )` na linha 6 insere um novo nó no último apontador. O novo nó torna-se o último nó da lista. (Repare que para a primeira vez este comando é equivalente a `Push( &head, 1 )`, que já foi utilizado anteriormente)
- O comando da linha 7 avança o `lastPtrRef` de maneira que ele passa a apontar para o campo `next` dentro do último nó — tal campo `next` passa a ser o último apontador da lista.

A Figura 8 apresenta o estado da memória para o código acima logo antes do terceiro nó ser inserido. Os valores anteriores de `lastPtrRef` são representados como setas pontilhadas na cor cinza.

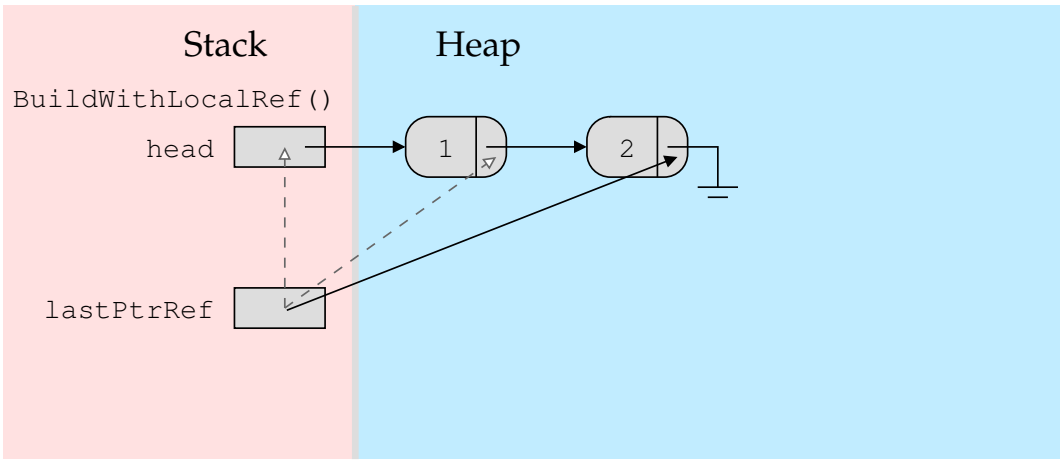


Figura 8: Representação da memória para o meio da execução de `BuildWithLocalRef()`. Note que o apontador `lastPtrRef` aponta para o apontador `head` (inicialmente) e depois passa a apontar para o campo `next` (que é um apontador) — ele nunca aponta para o nó inteiro, mas para o campo `next`.

1.4 Exemplos de Código

Esta seção apresenta listagem completa de código para demonstrar todas as técnicas descritas anteriormente.

1.4.1 Exemplo `AppendNode()`

Considere a função `AppendNode()` que é similar a `Push()`, exceto pelo fato que ela insere elementos no final da lista ao invés do começo. Se a lista estiver vazia, `AppendNode()` utiliza uma referência para apontador para modificar o AIL; caso contrário ela utiliza um laço para localizar o último nó da lista. Esta versão não utiliza `Push()`, ela constrói o novo nó diretamente.

```
1  node* AppendNode( node* & headRef, int num ) {
2      node* current = headRef; // current aponta para o primeiro nó
3      node* newNode;
4      newNode = new node;
5      newNode->data = num;
6      newNode->next = nullptr;
7      // caso especial para comprimento 0
8      if ( current == nullptr ) {
9          headRef = newNode;
10     }
11     else {
12         // Localiza o último nó.
13         while ( current->next != nullptr ) {
14             current = current->next;
15         }
16         current->next = newNode;
17     }
18 }
```

`AppendNode()` com `Push()`

Esta versão é muito similar, mas utiliza `Push()` para construir o novo nó. Para entender esta versão é necessário compreender bem o uso de referências para apontadores.

```
1  node* AppendNodePush( node* & headRef, int num ) {
2      node* current = headRef;
3      // caso especial de lista vazia
4      if ( current == nullptr ) {
5          Push( headRef, num );
6      } else {
7          // Localiza o último nó.
8          while ( current->next != nullptr ) {
9              current = current->next;
10         }
11         // Cria um nó depois do último nó
12         Push( &(current->next), num );
13     }
14 }
```

1.4.2 Exemplo `CopyList()`

Considere a função `CopyList()` que recebe uma lista e retorna uma cópia completa desta lista. Um apontador percorre a lista original da forma usual. Dois outros apontadores mantém a nova lista: um AIL e um ACL, este último sempre aponta para o último nó da nova lista. O primeiro nó é tratado como um caso especial, enquanto que os demais nós utilizam o ACL (variável `tail`) da forma convencional.

```
1  node* CopyList( node* head ) {
2      node* current = head; // usado para percorrer a lista original
3      node* newList = nullptr; // AIL da nova lista
4      node* tail = nullptr;    // ACL da nova lista
5      while ( current != nullptr ) {
6          if ( newList == nullptr ) { // caso especial para o primeiro nó
7              newList = new node;
8              newList->data = current->data;
9              newList->next = nullptr;
10             tail = newList;
11         }
12         else {
13             tail->next = new node;
14             tail = tail->next;
15             tail->data = current->data;
16             tail->next = nullptr;
17         }
18         current = current->next;
19     }
20     return( newList );
21 }
```

Desenho de Memória para `CopyList()`

Na Figura 9 temos o desenho de memória corresponde ao momento em que `CopyList()` acaba de copiar a lista $\{1, 2\}$.

`CopyList()` com `Push()` — Exercício

A implementação anterior é um pouco insatisfatória porque os *3-Passos De Ligação Em Lista* é repetido — uma vez para o primeiro nó e uma vez para todos os outros nós. Escreva a função `CopyList2()` que utiliza `Push()` para resolver a alocação e inserção de novos nós e, portanto, evitando a repetição de código mencionada.

`CopyList()` com `Push()` — Resposta

```
1  // Variante de CopyList() que faz uso da Push()
2  node* CopyList2( node* head )  {
3      node* current = head; // usado para percorrer a lista original
4      node* newList = nullptr; // AIL da nova lista
5      node* tail = nullptr;    // ACL da nova lista
6      while ( current != nullptr ) {
7          if ( newList == nullptr ) { // caso especial para o primeiro nó
```

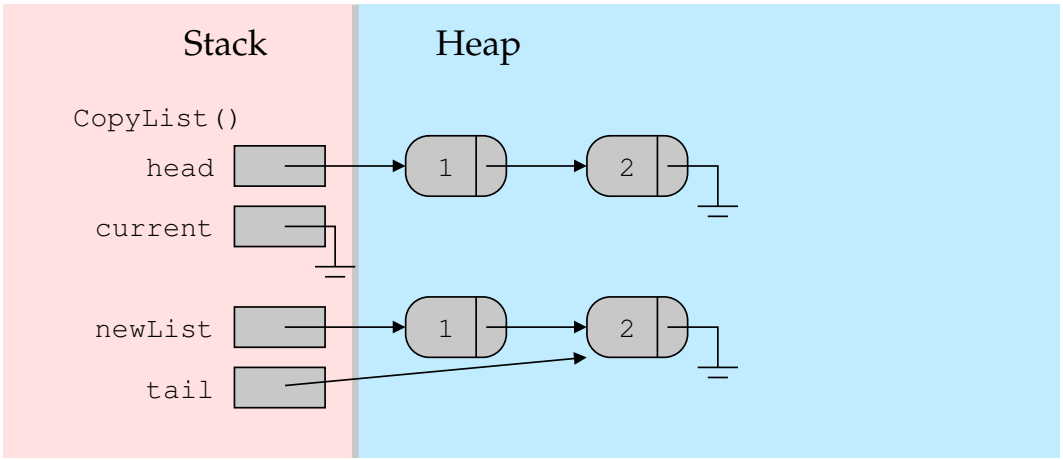


Figura 9: Representação da memória no final da execução de `CopyList()`.

```
8               Push( &newList, current->data );
9               tail = newList;
10          }
11          else {
12              Push( &(tail->next), current->data ); // adiciona cada nó no final
13              tail = tail->next; // avança tail para o novo último nó
14          }
15          current = current->next;
16      }
17      return( newList );
18  }
```

Exercício

Desenvolva uma nova versão de `CopyList` que utilize referência para apontadores de forma a acessar ACL e não tratar caso especial de primeiro nó. Veja o item 7, na página 21 para verificar como a inserção pode ser feita utilizando referência para ACL.

`CopyList()` Recursivo

Finalmente, por questões de completude, seque abaixo a versão recursiva de `CopyList()`. Ela tem a agradável característica de um código curto e elegante, marca registrada que códigos recursivos frequentemente apresentam. Contudo, não é uma boa opção para código de produção (comercial) uma vez que o uso de pilha de memória é proporcional ao tamanho da lista, o que pode não ser muito favorável para aplicações com listas longas.

```
1 // Variante recursiva
2  node* CopyList( node* head ) {
3      if ( head == nullptr ) return nullptr;
4      else {
5          node* newList = new node // Cria o novo nó
6          newList->data = current->data;
7          newList->next = CopyList( current->next ); // recursão p/ o resto
```

```
8           return( newList );
9       }
10  }
```

1.5 Variantes de Implementação de lista

Existem muitas variações da lista encadeada básica as quais apresentam vantagens individuais com relação à versão básica. Provavelmente é melhor ter um bom domínio sobre a lista encadeada básica e seu código antes de se preocupar em demasia com suas variantes.

- **Nó-Cabeça** Evita o caso em que o AIL é `nullptr`. Ao invés, escolhe-se uma representação de lista vazia como sendo um único nó-cabeça cujo campo `data` não é utilizado. A vantagem desta técnica é que a técnica de apontador de apontador (parâmetro por referência) não é necessária na implementação de operações como `Push()` Além disso, algumas das iterações são um pouco mais simples uma vez que é assumido que sempre existirá um nó-cabeça. A desvantagem é que a criação de uma lista agora requer a alocação (e consequente desperdício) de memória.

- **Circular** Ao invés de atribuir valor `nullptr` para o campo `next` do último nó, fazemos o mesmo apontar para o primeiro nó da lista. Neste caso não é mais necessário um AIL fixo, pois qualquer apontador que aponte para um nó da lista pode ser um AIL.

- **Apontador Calda** A lista não é mais representada apenas por um único AIL. Agora ela tem, além do AIL que aponta para o primeiro nó, um ACL que aponta para o último nó. A existência de um ACL permite que operações aplicadas ao final da lista — tais como adicionar um elemento no final ou concatenar duas lista — sejam implementadas eficientemente.

- **Estrutura de Cabeçalho** Esta é uma variante bem interessante em C que se parece bastante com a ideia de classe em C++. Neste caso define-se uma estrutura especial de cabeçalho que contém um campo AIL, um campo ACL e, possivelmente, um campo indicando o comprimento da lista para tornar certas consultas mais eficientes.

- **Duplamente Encadeada** Além do campo apontador `next` (próximo), cada nó possui um campo apontador `previous` (anterior). Com isso, inserções e remoções agora requerem algumas operações extras para acertar o novo campo `previous`. Porém, outras operações são simplificadas. Por exemplo, dado um apontador para um nó X, inserção e remoção de X podem ser realizadas diretamente, enquanto que na versão de encadeamento simples, era necessário iterar pela lista até localizar o ponto imediatamente anterior a X de forma a modificar seu campo `next`.

- **Lista de Blocos** Ao invés de armazenar apenas um único elemento do cliente em cada nó, armazenamos um pequeno vetor de tamanho constante com elementos do cliente em cada nó. Alterar o tamanho do número de elementos deste vetor por nó pode prover diferentes características de performance: muitos elementos por nó permite uma performance que se

aproxima dos vetores normais; poucos elementos por nó produz uma performance mais próxima de uma lista encadeada tradicional. Definir um tamanho adequado do bloco de armazenamento dos nós para a aplicação é uma boa forma de garantir excelentes níveis gerais de performance.

1.6 Exercícios de Listas Encadeadas

Nesta seção apresentamos 18 problemas de listas encadeadas organizados em ordem de dificuldade. Os primeiros são bem básicos e os últimos são relativamente avançados. Cada problema inicia com uma definição básica do que precisa ser realizado. Alguns dos problemas listados também incluem dicas ou desenhos para direcioná-los à solução. As soluções para todos os problemas estarão disponíveis brevemente.

É fácil simplesmente passar os olhos de forma passiva sobre as soluções — verificando sua existência sem que seus detalhes penetrem na sua mente. Para se beneficiar ao máximo destes problemas, você precisa fazer um esforço mental tentando, de fato, resolvê-los. Independentemente de você ser bem ou mau sucedido em resolver os problemas, você estará exercitando sua mente nas questões apropriadas relativas a lista, e a solução posteriormente fornecida fará mais sentido e será melhor apreciada.

Bons programadores são capazes de visualizar estruturas de dados e perceber como o código e a memória interagem. Listas encadeadas são bem adequadas para estimular este tipo de raciocínio visual. Utilize estes problemas para desenvolver suas habilidades de visualização. Faça desenhos esquemáticos da memória ao longo da execução manual do seu código. Utilize desenho de pré- e pós-condições de um problema para orientar seu raciocínio na direção da solução correta.

1. Escreva a função `Count()` que conta o número de vezes que um dado inteiro (no exemplo, `searchFor`) ocorre em uma lista. O código para a solução requer o idioma de programação clássico de iteração ao longo da lista, demonstrado na função `Length()` (veja Seção 1.1.1, Código 3, página 7).

   O protótipo da função é:

   ```
   int Count( node* head, int searchFor );
   ```

2. Escreva a função `GetNth()` que recebe uma lista encadeada L e um inteiro i como parâmetros e retorna o valor do dado armazenado na posição i da lista, ou seja o i-ésimo elemento. `GetNth()` utiliza a convenção de numeração do C/C++ na qual o primeiro elemento corresponde ao índice 0, o segundo ao 1, e assim sucessivamente. O índice passado i deve estar na faixa $[0; tamanho - 1]$. Se por acaso i estiver fora desta faixa a função deve lançar a exceção `std::out_of_range()` (inclua o cabeçalho `<stdexcept>`). Essencialmente, `GetNth()` é similar a operação de indexação em vetores `array[i]`.

   O protótipo da função é:

   ```
   int GetNth( node* head, int idx );
   ```

3. Escreva a função `DeleteList()` que recebe uma lista como parâmetro, desaloca toda a memória utilizada pela lista e faz o apontador AIL[9] apontar para `nullptr` (indicação de lista vazia).

   A implementação de `DeleteList()` necessitará de um referência para apontador, da mesma forma que em `Push()` (Seção 1.2.1, Código 7, página 15). Desta forma será possível modificar o AIL declarado no código cliente que invocou `DeleteList()`. A função deve ter o cuidado de não acessar o campo `next` dos nós *depois* de cada um ter sido removido, caso contrário a função cairá no clássico "*falha de segmentação*" ("*segmentation fault*") ou mesmo algo pior como derrubar o sistema!

   O protótipo da função é:

```
void DeleteList( node* & headRef );
```

4. Escreva a função `Pop()` que realiza a operação inversa da função `Push()`. `Pop()` recebe uma lista não-vazia, remove o nó localizado no início da lista e retorna os dados do nó removido. Se as únicas funções que você utilizar sobre uma lista forem `Push()` e `Pop()` então você estará simulando o comportamento da estrutura de dados *Pilha*. A função `Pop()` deverá lançar a exceção `std::runtime_error` caso a lista esteja vazia.

   O protótipo da função é:

```
int Pop( node* & headRef );
```

5. Escreva a função `InsertNth()` que recebe uma lista encadeada L, um inteiro i correspondente a uma posição na lista e um inteiro d que deverá ser inserido na lista na i-ésima posição. Assim como `GetNth()`, `InsertNth()` utiliza a convenção de numeração do C/C++ na qual o primeiro elemento corresponde ao índice 0, o segundo ao 1, e assim sucessivamente. O cliente pode especificar qualquer índice i na faixa $[0; tamanho]$ e o nó deve ser inserido na posição requisitada. Se por acaso i estiver fora desta faixa a função deve lançar a exceção `std::out_of_range`.

   Este é um problema mais difícil que, por exemplo, o `Push()` que só insere novos nós no início da lista. Portanto, será uma boa estratégia realizar alguns desenhos esquemáticos *antes* de iniciar a codificar uma solução.

   O protótipo da função é:

```
void InsertNth( node* & L, int i, int d );
```

6. Escreva a função `SortedInsert()` que recebe uma lista classificada em ordem *não decrescente* de seu campo de dados (`data`) e um nó como parâmetros, insere o nó em uma posição tal que a classificação da lista é mantida. Enquanto a função `Push()` aloca um novo nó para ser inserido na lista, `SortedInsert()` recebe um nó existente (i.e. previamente alocado pelo

[9] Apontador de Início de Lista.

código cliente) e apenas rearranja os ponteiros da lista para inserir o nó passado na posição correta. Existem várias soluções para este problema.

O protótipo da função é:

```
void SortedInsert( node* & headRef, node * newNode );
```

7. Escreva a função `InsertSort()` que recebe uma lista organizada em uma ordem qualquer e a classifica em ordem *não decrescente* de seu campo de dados (`data`). Ela deve fazer uso da função `SortedInsert()`, especificada no item 6.

   O protótipo da função é:

   ```
   void InsertSort( node* & headRef );
   ```

8. Escreva a função `Append()` que recebe duas listas encadeadas L_1 e L_2, concatena L_2 no final de L_1 e faz L_2 apontar para `nullptr` (indicação de lista vazia). Ambos os AILs passados para `Append()` precisam ser passados por referência, já que ambos podem precisar ser modificados. O segundo parâmetro L_2 sempre receberá o valor `nullptr`, mas em que situação precisaremos modificar o AIL correspondente à L_1? Esta situação ocorre quando L_1 é uma lista vazia. Neste caso o AIL de L_1 deve ser modificado de `nullptr` para o início da lista L_2. Por exemplo, seja $L_1 = \{\}$ e $L_2 = \{3, 4\}$, ao final de `Append(`L_1`,`L_2`)` teremos $L_1 = \{3, 4\}$ e $L_2 = \{\}$.

   O desenho da Figura 10 demonstra como fica a memória ao final da execução de `Append()` para duas listas não vazias. Os antigos valores dos apontadores são representados por setas pontilhadas na cor cinza. No contexto deste desenho, suponha a existência do código cliente `AppendTest()` que cria as duas listas e invoca `Append()`.

   O protótipo da função é:

   ```
   void Append( node* & L1Ref, struct node * & L2Ref );
   ```

9. Escreva a função `FrontBackSplit()` que recebe uma lista encadeada como parâmetro e a divide em duas sub-listas — uma correspondente à metade da frente e outra à metade de trás. Se a lista tem um número ímpar de elementos, o elemento extra deve pertencer a lista da frente. Por exemplo, aplicando a função sobre a lista $\{2, 3, 5, 7, 11\}$ deve gerar as duas sub-listas $\{2, 3, 5\}$ e $\{7, 11\}$.

   Dica: Provavelmente a estratégia mais simples consiste em computar o comprimento da lista, então usar um laço `for` para saltar o número certo de nós para achar o último nó da metade da frente, pra então cortar a lista neste ponto. Existe uma outra estratégia engenhosa que utiliza dois apontadores para percorrer a lista, um apontador "lento" que avança um nó em cada iteração, e um apontador "rápido" que avança dois nós em cada iteração. Quando o apontador rápido atingir o fim da lista, o apontador lento estará nas imediações da metade do caminho. Para ambas as estratégias sugeridas, cuidado especial deve ser tomado para certificar-se de que a lista será dividida no ponto certo.

   O protótipo da função é:

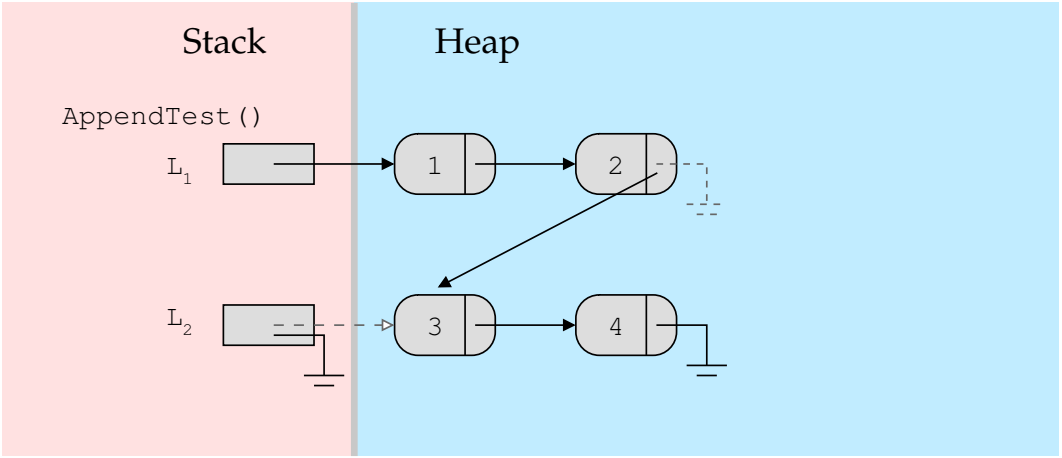


Figura 10: Representação da memória no final da execução de `Append()`. Antigos valores dos apontadores são representados por setas pontilhadas na cor cinza.

```
void FrontBackSplit( node* source,
                     node* & frontRef, node* & backRef );
\\ ou...
typdef struct node * nodePtr;
void FrontBackSplit( nodePtr source,
                     nodePtr & frontRef, nodePtr & backRef );
```

10. Escreva a função `RemoveDuplicates()` que recebe uma lista classificada em ordem *não decrescente* e remove todas as repetições de nós. Idealmente, a lista só deve ser percorrida uma vez para a realização desta tarefa.

    O protótipo da função é:

    ```
    void RemoveDuplicates( node* head );
    ```

11. Escreva a função `MoveNode()` que recebe duas listas, remove o elemento da frente da segunda lista e o insere no início da primeira lista. Pode-se afirmar que `MoveNode()` é uma variação de `Push()`, com a diferença que nenhum nó precisa ser alocado, mas simplesmente movido entre listas. Esta é uma conveniente função utilitária para auxiliar vários problemas posteriores. Por exemplo, aplicando `MoveNode({1,2,3},{1,2,3})` resultaria nas listas $\{1,1,2,3\}$ e $\{2,3\}$, respectivamente.

    O protótipo da função é:

    ```
    void MoveNode( node* & destRef, node* & sourceRef );
    ```

12. Escreva a função `AlternatingSplit()` que recebe uma lista como parâmetro e a divide em duas sub-listas menores. As sub-listas devem ser compostas de elementos retirados alternadamente da lista original. Por exemplo, se a lista original for $\{a,b,a,b,a\}$, então a primeira sub-lista deveria ser $\{a,a,a\}$ e a segunda sub-lista $\{b,b\}$. Você pode fazer uso da função

`MoveNode()` como função auxiliar. Os elementos das novas sub-listas podem estar em qualquer ordem e a lista original deve apontar para `nullptr` (lista vazia).

O protótipo da função é:

```
void AlternatingSplit( node* & source,
                       node* & L1Ref, node* & L2Ref );
```

13. Escreva a função `ShuffleMerge()` que recebe duas listas como parâmetros e as combina em uma única lista, retirando nós alternadamente de cada uma das listas originais. A lista resultante deve ser retornada por `ShuffleMerge()`. Por exemplo, `ShuffleMerge({1,2,3}, {7,13,1})` gera a lista $\{1, 7, 2, 13, 3, 1\}$. Se alguma das duas listas acabar antes da outra, todos os nós remanescentes da lista que ainda possui elementos devem ser remanejados para a lista final gerada. Você pode fazer uso da função `MoveNode()` como função auxiliar. Utilizando esta função juntamente com a `FrontBackSplit` é possível simular a ação de embaralhar cartas.

    O protótipo da função é:

    ```
    node* ShuffleMerge( node* a, node* b );
    ```

14. Escreva a função `SortedMerge()` que recebe como parâmetros duas listas classificadas em ordem *não decrescente* e as combina em uma única lista também classificada em ordem não decrescente. A lista resultante da combinação deve ser retornada por `SortedMerge()`. Idealmente, `SortedMerge()` deve fazer apenas uma passada em cada lista. Esta função é um pouco difícil de ser codificada corretamente — ela pode ser resolvida de forma recursiva ou iterativa. Existem vários casos especiais que devem ser identificados e testados no algoritmo proposto.

    O protótipo da função é:

    ```
    node* SortedMerge( node* a, node* b );
    ```

15. Escreva a função `MergeSort()` que implementa o algoritmo clássico de ordenação com o mesmo nome. Considerando-se que `FrontBackSplit()` e `SortedMerge()` estejam disponíveis e funcionando corretamente, é bem fácil implementar o `MergeSort()` recursivo: (1) dividir a lista em duas sub-listas menores; (2) recursivamente ordenar estas sub-listas invocando o `MergeSort()` sobre cada uma delas, e; (3) finalmente combinar as sub-listas ordenadas de volta em uma única lista ordenada. Ironicamente, implementar `MergeSort()` é mais fácil do que `FrontBackSplit()` ou `SortedMerge()`.

    O protótipo da função é:

    ```
    void MergeSort( node* & headRef );
    ```

16. Escreva a função `SortedIntersect()` que recebe como parâmetros duas listas classificadas em ordem *não decrescente*, cria e retorna uma nova lista representando as interseções entre as duas listas passadas. A nova lista deve ter sua própria memória — as listas originais *não*

devem ser modificadas. Em outras palavras, a construção da nova lista pode ser feita com `Push()`, e não `MoveNode()`. Idealmente, cada lista deve ser percorrida apenas uma vez.

O protótipo da função é:

```
node* SortedIntersect( node* a, node* b );
```

17. Escreva a função `Reverse()` que inverte a ordem de uma lista passada como parâmetro re-arranjando todos os campos apontadores `next` e o AIL. Idealmente, `Reverse()` deve alcançar seu objetivo com apenas uma única passada pela lista.

    Dica: Utilize três apontadores para percorrer a lista e realizar a inversão.

    O protótipo da função é:

    ```
    void Reverse( node* & headRef );
    ```

18. Escreva a função `RecursiveReverse()` que inverte a ordem de uma lista passada como parâmetro re-arranjando todos os campos apontadores `next` e o AIL. O problema a ser resolvido é o mesmo do item anterior, porém a solução deve ser recursiva. Idealmente, `RecursiveReverse()` deve alcançar seu objetivo com apenas uma única passada pela lista.

    O protótipo da função é:

    ```
    void RecursiveReverse( node* & headRef );
    ```

2 Lista Encadeada Simples Com Nó-Cabeça

Nesta seção vamos focar em mais detalhes as listas encadeadas com nó-cabeça, apresentado algoritmos para sua manipulação bem como estruturas de armazenamento na forma de classes do C++.

2.1 Fundamentos

Conforme mencionado anteriormente, uma *lista encadeada* é uma estrutura de dados do tipo *container*, ou seja, serve para armazenar elementos em uma certa ordem. A lista encadeada oferece operações de acesso geral, tais como *inserção*, *remoção* e *busca* arbitrária. Uma das características mais importantes de uma lista encadeada é seu caráter *dinâmico*, que permite armazenar um número de elementos limitado apenas pela memória disponível.

Uma lista encadeada consiste de uma sequência linear de *nós* dinamicamente alocados, que são *encadeados* (ou conectados) através de *apontadores* (ou ponteiros). Como citado anteriormente, versões mais elaboradas de listas encadeadas utilizam nós com apontadores para os nós sucessor e antecessor (*listas duplamente encadeadas*) e outras fazem com que o último nó aponte para o primeiro (*listas circulares*).

No nosso caso começaremos trabalhando com lista encadeada simples, cujo último nó aponta para `nullptr` (aterramento). Uma estrutura básica genérica representando este conceito é demonstrada abaixo:

```
1 template< typename T >
2 class ListNode {
3     public:
4         T elemento;
5         ListNode *prox;
6 };
```

Para facilitar a implementações dos algoritmos de manipulação de uma lista encadeada simples, é possível fazer uso de um nó especial chamado de *nó-cabeça*. Este nó especial tem as seguintes características:

- É o nó apontado pelo apontador de início de lista (AIL);
- Nunca é removido;
- Não contém valor válido. Porém, em algumas ocasiões pode conter informações relacionadas a lista, como, por exemplo, a quantidade de nós ou um apontador para o último elemento da lista.

Desta forma o primeiro nó de uma lista (nó-cabeça) é acessível via AIL (denominado de `ptLista` em nossos exemplos) e a partir dele é possível realizar o acesso sequencial à todos os elementos da lista. A Figura 11 apresenta alguns exemplos esquemáticos de listas, enquanto que o Código 10 apresenta uma possível definição da TAD lista encadeada simples (LES).

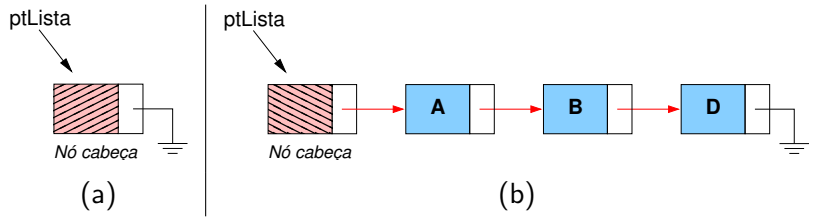


Figura 11: Exemplo de (a) uma lista encadeada com nó-cabeça *vazia*; e (b) uma lista encadeada simples com nó-cabeça e 3 elementos, aterrada em D.

Código 10 Pseudo-código do TAD lista encadeada simples.

```
tad LSE
    tipo Chave ≡ inteiro                          #chave é inteiro nesta TAD
    tipo Informação ≡ inteiro                     #informação também é inteiro nesta TAD
    tipo NoLSE ≡ estrutura                        #estrutura de nó de uma lista encadeada
        id: Chave                                 #chave identificadora do nó (ex. CPF)
        info: Informação                          #informação armazenada em um nó
        prox: ref NoLSE                           #ponteiro p/ próximo da lista
    fim

    var length: inteiro                           #comprimento atual da lista (opcional)
    var AIL: ref NoLSE                            #Apontador para Início de Lista

    #TAD LSE com nó cabeça
    construtor ()
        AIL ← aloque (NoLSE)                      #requisitando memória para o SO
        @{AIL}.prox ← ⊥
        length ← 0
    fim
    #declarando o método busca, privado.
    privado buscaOrd ≡ procedimento (Chave, ref NoLSE, ref NoLSE)
fim
```

2.2 Busca

Uma das operações mais importantes sobre lista é a *busca*, que consiste em localizar um determinado elemento na lista. A importância da busca deve-se ao fato de que ela é invocada pelas operações de *inserção* (descrita na Seção 2.3) e *remoção* (descrita na Seção 2.4).

A operação de busca consiste em percorrer a lista sequencialmente a partir do primeiro elemento válido até seu final, procurando pela primeira[10] ocorrência do elemento solicitado.

Uma das formas mais comum de implementar a busca consiste fazer com que ela retorne dois

[10] Note que é possível que uma mesma lista possua mais de uma instância do elemento solicitado.

apontadores: um para a primeira ocorrência na lista do elemento procurado (`pont`) e outro para o nó imediatamente anterior ao elemento procurado (`ante`). O apontador para o nó imediatamente anterior será utilizado nas operações de inserção e remoção. Se o elemento solicitado não estiver na lista então os apontadores `pont` e `ante` devem apontar, respectivamente, para um valor nulo (i.e. `nullptr`) e para o nó após o qual o elemento procurado deveria estar (possível ponto de inserção). O Código 11 apresenta o pseudo-código para o algoritmo de busca. Estamos assumindo que existe uma TAD `LES`, cujo um de seus campos é denominado de `AIL` e representa o apontador para início da lista.

Código 11 Pseudo-código do algoritmo de **busca** em lista encadeada simples (ordenada).

```
Entrada: Chave a ser procurada e apontadores para receber resultado da busca
Saída: pont aponta para o elemento procurado, ou ⊥, caso não esteja na lista, e; ante aponta para o
       elemento anterior ao procurado
procedimento LSE.buscaOrd(x: Chave; ante: ref NoLSE; pont: ref NoLSE)
    var ptr: ref NoLSE ← @{AIL}.prox        #ponteiro de trabalho: 1º nó válido
    ante ← AIL                                          #aponta para nó cabeça
    pont ← ⊥                                #assumimos que chave não está na lista
    enquanto ptr ≠ ⊥ faça                                #não chegar ao fim da lista
        se @{ptr}.chave < x então                          #devemos ainda avançar?
            ante ← ptr                                   #avança ponteiro anterior
            ptr ← @{ptr}.prox                          #atualiza ponteiro de percurso
        senão
            se @{ptr}.chave = x então                        #o atual é o procurado?
                pont ← ptr                              #apontar para nó procurado
            fim
            ptr ← ⊥                                                  #forçar saída
        fim
    fim
fim
```

2.3 Inserção

O processo de *inserção* em uma lista deve ser bem planejado para evitar que a lista se "quebre" ou a inserção seja feita em local inapropriado. A inserção pode ser implementado de várias maneiras, adicionando-se o elemento:

- Ao final da lista;
- No início da lista;
- De forma a preservar uma ordem pré-existente da lista;
- Logo após um apontador que aponta para um nó válido da lista.

A última opção é a mais versátil, pois basta fazer o apontador indicador de inserção apontar para posição desejada para que a inserção ocorra de forma correta. Neste caso, cabe ao programador certificar-se que o ponto de inserção manterá as características desejadas da lista (por exemplo, manter os elementos ordenados).

O Código 12 apresenta um pseudo-código para a inserção feita após um nó indicando ao algoritmo. Se o elemento a ser inserido já estiver na lista o algoritmo não faz nada.

Código 12 Pseudo-código do algoritmo de **inserção** em lista encadeada simples.

```
procedimento LSE.insere(x: Chave; novoValor: Informação)
    var pt: ref NoLSE ← ⊥                                  #ponteiro para novo nó
    var ante: ref NoLSE ← ⊥              #ponteiro anterior a posição de inserção
    var pont: ref NoLSE ← ⊥                                  #resultado da busca
    buscaOrd(x, ante, pont)
    se pont = ⊥ então                                  #elemento não está na lista
        pt ← aloque (NoLSE)                                      #solicitar novo nó
        @{pt}.info ← novoValor                                     #inicializar nó
        @{pt}.chave ← x                                  #inicializar chave do nó
        @{pt}.prox ← @{ante}.prox                                        #encadear
        @{ante}.prox ← pt                                      #acertar a lista
        ptr ← @{ptr}.prox                            #atualiza ponteiro de percurso
    senão
        escreva ("Elemento já está na lista!")
    fim
fim
```

A Figura 12 apresenta o processo de inserção de um novo nó segundo o algoritmo apresentado no Código 12.

2.4 Remoção

Outra operação importante em uma lista encadeada é a *remoção*. Para remover um elemento é preciso, primeiramente, encontrar o elemento que se deseja remover, mantendo um apontador para o mesmo. O procedimento de remoção é facilitado se tivermos um apontador para a posição imediatamente anterior ao nó que se deseja remover. Uma vez que os dois apontadores — para o nó a ser removido e para o nó antecessor imediato — estão configurados a remoção pode ser efetuada de forma simples. Portanto, mais uma vez o procedimento de busca deve ser utilizado.

O processo supracitado é ilustrado pela Figura 13 e o algoritmo correspondente pode ser encontrado no Código 13.

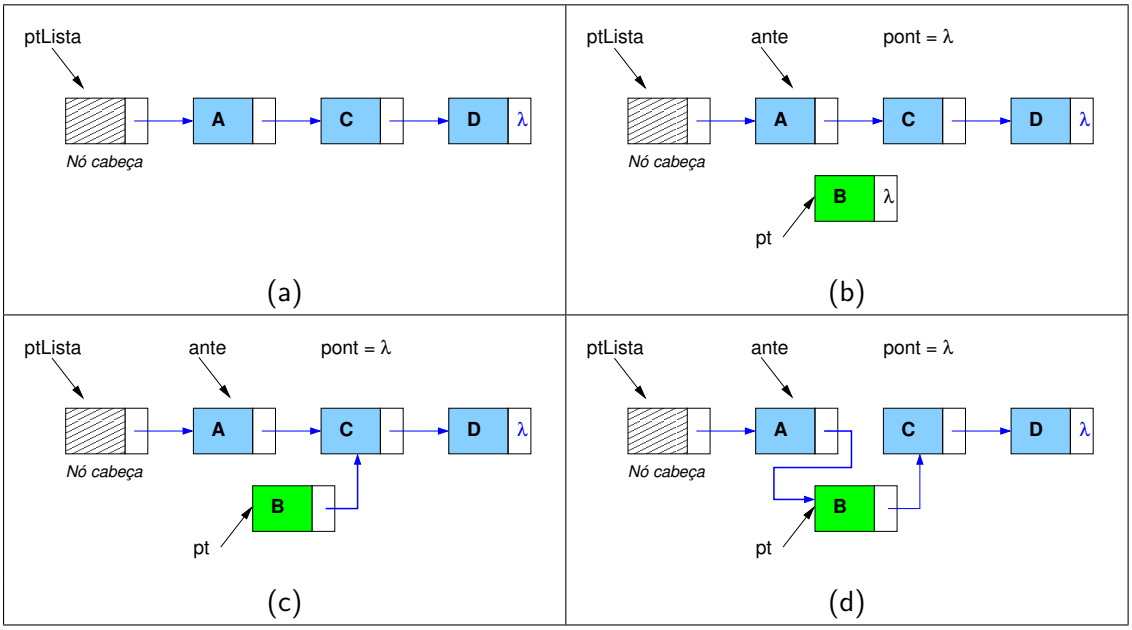


Figura 12: Exemplo de inserção de um novo nó em uma lista. Em (a) temos a lista encadeada original; (b) o novo nó (conteúdo **B**) é alocado e apontado por `pt`; (c) o novo nó é ligado a lista através de seu campo `prox`, e; (d) o encadeamento da lista é ajustado de forma a incluir o novo nó. O símbolo λ representa o valor `nullptr`.

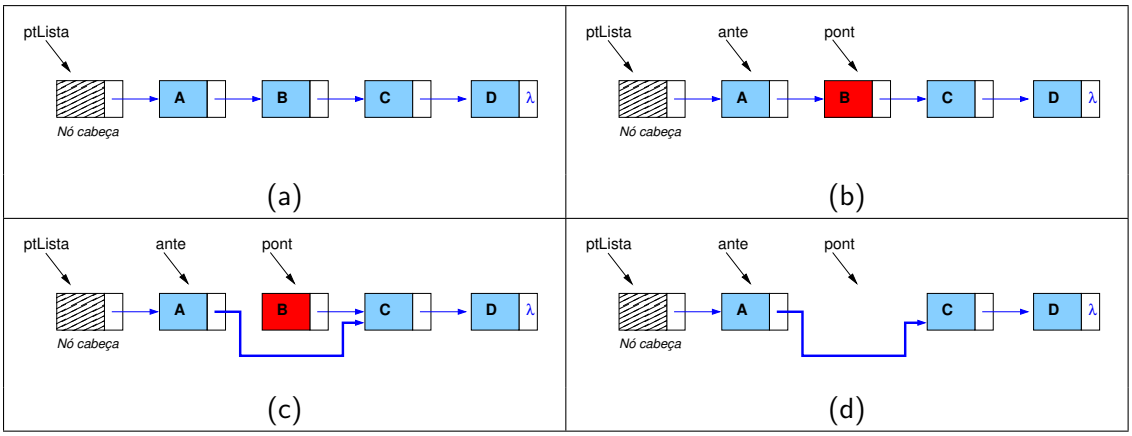


Figura 13: Exemplo de remoção de um nó de uma lista. Em (a) temos a lista encadeada original; (b) o novo a ser removido, apontado por `pont`, é localizado (conteúdo **B**) e seu antecessor é apontado por `ante`; (c) o novo a ser removido é contornado (*bypass*), e; (d) o nó marcado é desalocado. O símbolo λ representa o valor `nullptr`.

Código 13 Pseudo-código do algoritmo de **remoção** em lista encadeada simples.

```
função LSE.remove(x: Chave): Informação
    var valRecuperado: Informação ← ∅                    #valor que será removido
    var ante: ref NoLSE ← ⊥              #ponteiro anterior a posição de inserção
    var pont: ref NoLSE ← ⊥                                  #resultado da busca
    buscaOrd(x, ante, pont)
    se pont ≠ ⊥ então                                  #elemento não está na lista
        @{ante}.prox ← @{pont}.prox                     #passar ■por cima■ de pont
        valRecuperado ← @{pont}.info                             #guardar valor nó
        libere (pont)                                             #liberar memória
    senão
        escreva ("Elemento não está na lista!")
    fim
    retorna valRecuperado
fim
```

3 Classe Lista Duplamente Encadeada

Nesta seção descrevemos os detalhes para a implementação de uma classe `List` com template. Esta lista implementa a variação de encadeamento duplo.

Antes de entrar em detalhes específicos da lista, faz-se necessário introduzir o conceito de iteradores, um padrão de programação muito comum e importante para uma manipulação segura e eficiente de listas.

3.1 Iteradores

Iterador é um conceito importante relacionado a estruturas de dados que funcionam como *container* de dados. Trata-se de um mecanismo semelhante a um apontador utilizado para acessar os elementos de um container de forma segura. De fato, o iterador é um *padrão de implementação* (*design pattern*) adotado por bibliotecas profissionais como o STL (*Standard Template Library*) do C++ e a linguagem de progamação Java.

A função da entidade iterador (implementada como uma classe em C++) é manter um apontador para uma certa posição do container (no nosso caso uma lista). Através de um objeto iterador é possível manipularmos elementos da lista. Essa estratégia simples evita problemas básicos como "falha de segmentação", comumente provocada por apontadores *inválidos* ou *selvagens* (i.e. apontam para posições de memória desconhecidas, normalmente inválidas).

Apontadores inválidos são evitados ao se invocar métodos da classe iterador que verificam se o apontador é seguro (isto é aponta para algum elemento válido do container). Além disso a classe iterador pode oferecer sobrecarga de operadores (por exemplo, `operator==` para comparação de elementos) que facilitam, entre outras coisas, a operação de acessar sequencialmente todos os elementos de um container.

Desta forma, a função do iterador é manter o *status* sobre qual é a posição da lista atualmente sendo acessada pelo código cliente. Para que essa relação "íntima" entre lista e iteradores dê certo é necessário que a classe iterador seja declarada como *amiga* (`friend`) da classe `List`, ou seja, a classe iterador terá privilégios de acesso à membros e métodos privados e/ou protegidos da classe `Lista`.

Confira o exemplo no Código 14 que demonstra o uso de iteradores associados à classe `std::vector` do STL. Este programa cria um objeto `vec1` do tipo `std::vector` que inicialmente contém os caracteres de uma string `s` (linha 11). Na linha 12 um segundo `std::vector`, `vec2`, vazio é criado. A linha 13 cria um iterador que vai percorrer um container do tipo `std::vector<char>`. Linhas 15–16 utilizam o iterador criado para percorrer o container `vec1` e inserir cada um de seus elementos em `vec2`, utilizando o método de `std::vector` chamado `push_back()`. Note que o iterador é utilizado na linha 16 para acessar um elemento do container `vec1`, usando a sobrecarga do operador `*` (em `*i`). Ao final, `vec2` possui uma cópia de `vec1`, fato testado na linha 18.

Código 14 Exemplo de uso de **iterador** com o container `std::vector` do STL.

```
1  #include <cassert>
2  #include <iostream>
3  using std::cout;
4  using std::endl;
5
6  #include <vector>
7  using std::vector;
8
9  int main ( ) {
10    const char *s = "Adoro iteradores!";
11    vector< char > vec1 ( &s[0], &s[strlen(s)] );
12    vector< char > vec2;
13    vector< char >::iterator i;
14
15    for ( i = vec1.begin(); i != vec1.end(); ++i )
16       vec2.push_back( *i );
17
18    assert ( vec1 == vec2 );
19    cout << " --- Ok." << endl;
20
21    return 0;
22 }
```

3.2 Classe `List`: Uma Lista Duplamente Encadeada

Esta classe deverá ser implementada como uma *lista duplamente encadeada*, sendo necessário manter dois apontadores para ambas extremidades da lista. Esta decisão permite um tempo de resposta constante, quando a operação é realizada sobre um ponto conhecido da lista (por exemplo, sobre um iterador ou sobre uma das extremidades da lista).

Para tanto serão necessários quatro classes, a saber:

- `List`, que conterá apontadores para ambas as extremidades da lista (o AIL `head` e o ACL `tail`), o tamanho atual da lista e mais um conjunto de métodos para *inserir*, *remover*, *procurar* e *consultar* os elementos da lista.
- `Node`, que será uma *estrutura aninhada privada*[11]. Um nó contém o dado d a ser armazenado, apontadores para os nós antecessor p e sucessor n no encadeamento, e um único construtor que recebe d, p e n como parâmetros. Como esta `struct` é definida dentro da classe `List`, os seus campos podem ser acessados diretamente com o operador '`.`', sem a necessidade de codificar métodos *accessors*/*mutators*.
- `const_iterator`, que abstrai a noção de posição, e é uma *classe aninhada pública*. O `const_iterator` armazena um apontador para o nó "atual" da lista, e provê implementação das funcionalidades básicas de iteradores, todas na forma de *sobrecarga* de operadores tais como `=`, `==`, `!=` e `++`.

[11] Na verdade um `struct`, por *default*, é público, mas o `struct Node` deve ser declarado dentro da classe `List` sob a diretiva `private`.

- `iterator`, que abstrai a noção de posição, e é uma *classe aninhada pública*, derivada de `const_iterator` através de herança pública. O `iterator` tem a mesma funcionalidade do `const_iterator`, exceto pelo `operator*` que retorna uma referência para o item sendo visualizado, ao invés de uma referência *constante*. Note que `iterator` pode ser usado em qualquer rotina que requeira um `const_iterator`, mas o contrário não é verdade. Em outras palavras, um `iterator` *É-UM* `const_iterator`.

Para facilitar uma série de operações de manipulação da lista (por exemplo, evitando os casos especiais) e facilitar a implementação de iteradores, a classe `Lista` deverá possuir um nó-cabeça (denominado de `header`) e um nó-calda (denominado de `tail`). A Figura 14 apresenta uma ilustração esquemática da lista duplamente encadeada com os nós *sentinelas*.

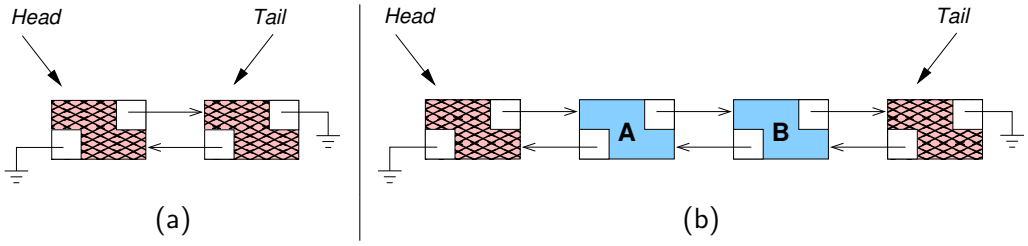


Figura 14: Exemplos de listas *duplamente encadeada* com *nó-cabeça* (`Head`) e *nó-calda* (`Tail`): (a) vazia e (b) com dois elementos.

Os Código 15 à 18 apresentam trechos com a declaração das classes, seus membros e todos os métodos que devem ser implementados. Perceba que todas as quatro classes são declaradas em um único arquivo denominado `List.h` e que o programa de teste (contendo o `main`) deve-se chamar de `TestList.cpp`. Para melhor organizar seu código, crie as classes dentro do *namespace* `MyList`.

Implementação

Vamos descrever, brevemente a função dos métodos mais importantes da classe `List` apresentada no Código 15. Perceba que a lista possui dois nós especiais, `head` e `tail` (linhas 42 e 43), que devem ser alocados no construtor padrão e no construtor cópia, e devem ser desalocados apenas no destrutor para evitar vazamento de memória. Além da alocação destes dois nós, é preciso inicializar os outros campos da classe, como a quantidade de elementos e os apontadores de cada um dos nós especiais, conforme Figura 14-(a). Como estas ações devem ser realizadas duas vezes (no construtor e no construtor cópia), podemos centralizá-las no método privado `init()` (linha 46), bastando invocá-lo para acionar estas ações.

Antes de prosseguirmos, cabe uma palavrinha sobre *mutators* e *accessors*. Vários dos métodos de `List` apresentam estas duas versões. A única diferença entre as versões *mutator* e *accessor* de um mesmo método é que a primeira versão retorna um iterador normal (`iterador`) e a segunda um iterador constante (`const_iterator`) que não permite alteração do valor do nó apontado por ele.

Código 15 Listagem parcial da classe `List`. Esta listagem contém referências para Códigos 16, 17 e 18, que contém a listagem das classes relacionadas.

```
1  template <typename Object>
2  class List {
3    private:
4      // O nó básico da lista.
5      struct Node { Veja Código 16 };
6
7    public:
8      // Classes iteradores.
9      class const_iterator { Veja Código 17 };
10     class iterator : public const_iterator { Veja Código 18 };
11
12     // Métodos básico que todas as classes deveriam oferecer
13     List( );
14     ~List( );
15     List( const List & rhs );
16     const List & operator= ( const List & rhs );
17
18     // Métodos específicos da classe
19     iterator begin( ); // Versão mutator
20     const_iterator begin( ) const; // Versão accessor
21     iterator end( ); // Versão mutator
22     const_iterator end( ) const; // Versão accessor
23     int size( ) const; // Retorna tamanho da lista
24     bool empty( ) const; // Retorna true se vazia, falso caso contrário
25     void clear( ); // Apaga todos os nós da fila, tornando-a vazia
26     Object & front( ) { return *begin( ); }
27     const Object & front( ) const;
28     Object & back( );
29     const Object & back( ) const;
30     void push_front( const Object & x ) { insert( begin( ), x ); }
31     void push_back( const Object & x );
32     void pop_front( );
33     void pop_back( );
34     iterator insert( iterator itr, const Object & x );
35     iterator erase( iterator itr ); // remove nó apontado por itr
36     iterator erase( iterator start, iterator end );
37     const_iterator find( const Object & x ) const;
38     iterator find( const Object & x );
39
40   private:
41     int    theSize;
42     Node *head;
43     Node *tail;
44
45     // Inicializa campos para representar lista vazia
46     void init( );
47 }; // Classe List
48
49 #endif
```

Um dos grupos de métodos mais importantes são os `begin()` e `end()`. O `begin()` retorna

Código 16 Listagem parcial da estrutura `Node`, parte da classe `List` (confira Código 15).

```
1  // O nó básico da lista duplamente encadeada. Aninhada dentro de
2  // List, a estrutura 'Node' é pública (dentro de List), mas é ainda
3  // assim é encapsulada para cliente pois é declarada na categoria
4  // 'private' em List.
5  struct Node {
6      Object  data; // Campo de dados
7      Node   *prev; // Apontador para o próximo nó
8      Node   *next; // Apontador para o nó anterior
9
10     // Construtor inline com vários parâmetro default
11     Node( const Object& d = Object( ), Node* p = nullptr, Node* n = nullptr )
12       : data( d ), prev( p ), next( n ) { /* Empty */ }
13 };
```

um iterador para o primeiro nó válido da lista, ou seja o nó logo após o `head`. Já o método `end()` retorna um iterador para o nó especial `tail`, ao invés do último nó válido da lista. Esta sutil diferença é para viabilizar alguns idiomas de programação bem naturais em programas C++ e iteradores do STL, como

```
iterator itr = begin();
for ( ; itr != end(); ++itr ) // Busca elemento x na lista
    if ( *itr == x ) break;
```

neste caso se `end()` retornasse o último nó válido o laço ao invés do nó `tail` o trecho em vermelho não faria sentido. Lembre-se que nos exemplos de lista apresentados na Seção 1 o final da lista (`end`) é representado pelo apontador `nullptr`. Note também que caso o método `begin()` seja invocado em uma lista vazia, será retornado o nó `tail` (primeiro nó apontado por `head`), que é o comportamento correto, uma vez que o nó `tail` simboliza fim de fila.

Uma dica importante no desenvolvimento dos demais métodos é reutilizar ao máximo os métodos `begin()` e `end()` para percorrer a lista e acessar elementos. O próximo grupo de métodos são os de operação básica como inserção e consulta na frente e na calda da fila (linhas 26 à 33), comportamento semelhante ao da estrutura de dados *Deque*. Note que os métodos `front()` e `push_front()` já estão implementados! Os métodos `size()`, `empty()` e `clear()` são auto-explicativos, com especial atenção para o método `clear()` que deve percorrer a lista para desalocar cada um de seus nós (veja problema 3, na página 28).

Logo a seguir, nas linhas 34 à 36, temos os métodos para inserção/remoção de elementos na/da lista. Este métodos trabalham sobre iteradores passados como argumentos, em especial o `insert()` insere um elemento *antes* o nó apontado pelo iterador passado como argumento, e o `erase()` com dois parâmetros — `start` e `end` — que remove todos nos na faixa $[start; end)$, ou seja, nó apontado pelo parâmetro `end` não é removido (intervalo fechado-aberto).

Por último temos o método `find()` que busca um elemento na lista, retornando o iterador `end()` (final de fila) caso o elemento não seja encontrado.

Código 17 Listagem da classe `const_iterator`, parte da classe `List` (confira Código 15).

```
1  class const_iterator {
2    public:
3      // Construtor público
4      const_iterator( );
5      // Retorna objeto armazenado na posição atual.
6      // Para const_iterator, este método é um accessor que retorna
7      // uma referência constante. Logo este operador só pode
8      // aparecer do lado direito de uma atribuição ou em comparações.
9      const Object & operator* ( ) const;
10     // Pré-incremento: ++it
11     const_iterator & operator++ ( );
12     // Pós-incremento: it++
13     const_iterator operator++ ( int );
14     // Pré-decremento: --it
15     const_iterator & operator-- ( )
16     // Pós-decremento: it--
17     const_iterator operator-- ( int )
18     bool operator== ( const const_iterator & rhs ) const;
19     bool operator!= ( const const_iterator & rhs ) const;
20
21   protected:
22     // Declarado como protected para ser acessível pela classe 'iterator'
23     Node *current;
24     // Construtor protegido que recebe um nó para ser apontado.
25     // Utilizado dentro da classe 'List' apenas e não pelo código cliente.
26     const_iterator( Node* p );
27     // Necessário p/ permitir acesso de 'List' aos campos desta classe.
28     friend class List<Object>;
29 };
```

Implementação dos Iteradores

Os métodos das classes de iteradores são bem auto-explicativos e dizem respeito a manipulação de um apontador (`current`, declarado na linha 23 do Código 17, página 44) que aponta para um nó da lista. As operações básica são de acesso do valor apontado por `current` com o operador '`*`' e as operações de avanço e retrocesso (`operator++` e `operator--`).

Para diferenciar a chamada `itr++` de `++itr` são declarados, respectivamente, os operadores `operator++()` e `operator++(int)`. O parâmetro `int` é usado apenas para diferenciar entre os dois métodos (assinatura diferentes) — o mesmo vale para o operador de decremento '`--`'. Vale ressaltar que o operador de pré-incremento (`++itr`) e pré-decremento (`--itr`) são mais eficientes do que suas contrapartidas em pós-incremento e pós-decremento (i.e. `itr++` e `itr--`). Você saberia justificar o porque disto?

Como `iterator` é declarado como uma extensão de `const_iterator`, alguns de seus métodos podem ser re-aproveitados, como o operador de comparação (`==`) e diferença (`!=`).

◀ FIM ▶

Código 18 Listagem da classe `iterator`, parte da classe `List` (confira Código 15).

```
1  // inheritance: IS-A relation
2  class iterator : public const_iterator {
3    public:
4      // Construtor público do iterator que invoca construtor da classe base.
5      iterator( ) { /* Empty */ }
6      // Retorna objeto armazenado na posição apontada por 'current'.
7      // For iterator, tem duas versões, uma accessor que permite sua
8      // utilização do apenas do lado direito de uma atribuição ou em
9      // comparações.
10     const Object & operator* ( ) const;
11     // Esta versão mutator é usada do lado esquerdo de atribuições.
12     Object & operator* ( );
13
14     // prefixo
15     iterator & operator++ ( );
16     // posfix
17     iterator operator++ ( int );
18     // prefixo
19     iterator & operator-- ( );
20     // posfix
21     iterator operator-- ( int );
22
23   protected:
24     // Construtor protegido que espera uma posição para apontar.
25     // É invocado principalmente dentro da classe 'List', mas
26     // não pelo cliente (que não tem acesso a este método).
27     iterator( Node *p );
28
29     // Necessário p/ permitir acesso de 'List' aos campos desta classe.
30     friend class List<Object>;
31 };
```

Referências

[1] Nick Parlante. Pointer and Memory, document #102, July 2000. Computer Science Education Library, Stanford University, USA, http://cslibrary.stanford.edu/102/.

[2] Nick Parlante. Linked List Basics, document #103, July 2001. Computer Science Education Library, Stanford University, USA, http://cslibrary.stanford.edu/103/.