INSTITUTO FEDERAL
Rio Grande do Norte
Campus Nova Cruz

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO RIO GRANDE DO NORTE
Campus Nova Cruz – Código INEP: 24083003
Av. José Rodrigues de Aquino Filho, Nº 640, RN 120, Alto de Santa Luzia, CEP 59215-000, Nova Cruz (RN)
CNPJ: 10.877.412/0015-633 – Telefone: (84) 4005-4107

Texto Complementar:

Listas Duplamente Encadeadas
(EDL – TADS4M)

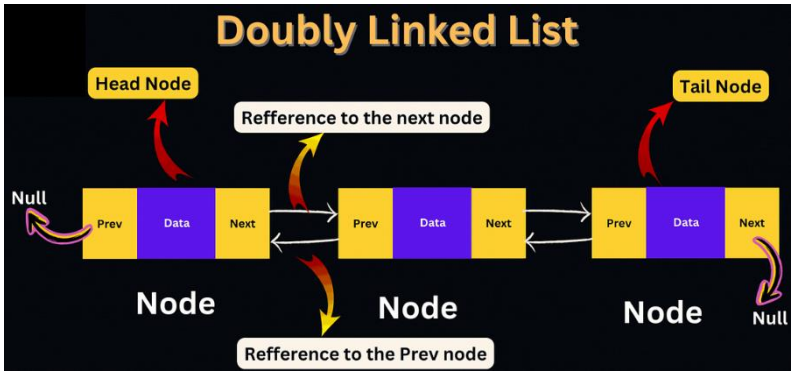
1. Introdução

Anteriormente, tivemos contato com as listas simplesmente encadeadas. Neste momento, começaremos a trabalhar com uma estrutura de dados bastante semelhante, conhecida como **lista duplamente encadeada**. Esta estrutura de dados dinâmica permite armazenar seus elementos de maneira organizada. Nela, **cada elemento ou nó** da estrutura **possui dois ponteiros**: um que faz referência para o elemento anterior da estrutura e outro que aponta para o elemento seguinte. Essa característica torna possível **percorrer a lista nos dois sentidos**, o que a torna mais flexível do que a lista simplesmente encadeada.

2. Estrutura da Lista Duplamente Encadeada

A estrutura da lista duplamente encadeada é formada, basicamente, pelos seus nós e dois outros ponteiros que possuem funções importantes. A seguir, são apresentados maiores detalhes sobre estes elementos:

- **Nós**: unidades básicas que armazenam a chave, demais dados, um ponteiro que aponta para o próximo nó da lista (`next`) e outro ponteiro que aponta para o nó anterior da lista (`previous`);
- **Cabeça da lista (`head`)**: ponteiro que aponta para o primeiro nó da lista. Caso a lista esteja vazia, o ponteiro `head` é nulo;
- **Cauda da lista (`tail`)**: ponteiro que aponta para o último nó da lista. Caso a lista esteja vazia, o ponteiro `tail` é nulo;
- **Final da lista**: o último nó da lista é indicado pelo ponteiro `tail`. Além disso, esse nó terá o seu ponteiro `next` igual a nulo, indicando que a lista não possui mais elementos adiante.



A lista duplamente encadeada permite o deslocamento bidirecional. O ponteiro `head` e o ponteiro `next` existente nos nós deste tipo de lista permitem que seja realizado o deslocamento no sentido direto, do início ao fim da lista. Por outro lado, o ponteiro `tail` e o ponteiro `previous` permitem a realização do deslocamento no sentido reverso, isto é, do final para o início da lista.

Os ponteiros `head` e `tail` possibilitam acesso rápido ao início e final da lista. Portanto, é possível inserir ou remover rapidamente elementos nestas posições. Isto é bastante útil quando desejamos implementar estruturas como pilhas, filas e deques tendo como base as listas duplamente encadeadas. Da mesma maneira que nas listas simplesmente encadeada, quando uma inserção ou remoção é realizada, não é necessário efetuar o deslocamento de elementos como ocorre nas listas sequenciais que se baseiam em vetores. Nas listas simplesmente ou duplamente encadeadas apenas é necessário ajustar os ponteiros contidos nos nós adjacentes e os ponteiros `head` e `tail` quando necessário. Assim, as operações de inserção e remoção tornam-se mais eficientes.

3. Principais Algoritmos

Novamente, iremos considerar que a estrutura de dados em estudo mantém os seus elementos ordenados. Como foi visto durante o nosso estudo da lista simplesmente encadeada, o ordenamento traz algumas vantagens, tais como: busca mais eficiente, inserção organizada e a facilidade para a realização de operações avançadas.

Na lista duplamente encadeada **ordenada**, tanto a inserção quanto a remoção devem ocorrer em pontos adequados da lista de acordo com a posição da chave na ordenação dos elementos ou nós. Dessa maneira, é preciso percorrer a lista até encontrar a posição correta de inserção do novo elemento ou a posição onde o elemento a ser removido se encontra. É essa característica de busca que faz com que as tradicionais operações de inserção e remoção nesta estrutura ocorram com complexidade assintótica linear — $O(n)$. O ato em si de inserir ou remover um elemento da lista é simples, apresentando complexidade constante $O(1)$, uma vez que apenas é necessário efetuar o ajuste de alguns dos ponteiros que fazem parte da estrutura. Entretanto a busca pela posição do elemento é assintoticamente linear.

3.1 Operação de Busca (*search*)

O algoritmo da operação de busca apresentado a seguir recebe como argumento o valor da chave a ser encontrada na lista por meio do parâmetro x. O ponteiro `current` é utilizado para percorrer a lista duplamente encadeada, sendo inicializado com o valor de `head`. Lembrem-se que o ponteiro `head` é responsável por apontar para o primeiro nó da lista.

```
Node* search(int x) {
    Node* current = head;
    while (current != nullptr && current->key <= x) {
        if (current->key == x)
            return current;  // Retorna o nó encontrado
        current = current->next;
    }
    return nullptr;  // Não encontrado
}
```

O código irá retornar o nó que possui como chave o valor x ou nulo, caso a chave não exista na lista. O laço de repetição **while** é executado enquanto o final da lista não for encontrado (`current != nullptr`) e o valor da chave do nó atual for menor do que x

(`current->key <= x`). O princípio de funcionamento da busca é idêntico ao da lista simplesmente encadeada, porém, neste exemplo de código, há uma pequena modificação para a lista duplamente encadeada. Agora, a verificação se a chave foi procurada foi encontrada localiza-se no interior do próprio laço de repetição **while** (`current->key == x`). Caso a chave seja encontrada em algum dos nós da lista, este nó será retornado pela operação. Em caso contrário, retorna-se nulo (`nullptr`).

Para o procedimento de busca por um elemento de chave x na lista simplesmente encadeada, foi apresentado o exemplo de código a seguir. Este código também é funcional para a lista duplamente encadeada. Nele, a verificação se a chave foi encontrada ocorre após o término da execução da estrutura **while**. Existem três possíveis situações que fazem com que a execução da estrutura de repetição seja finalizada: **i**. o ponteiro corrente chegou ao final da lista; **ii**. o valor da chave do nó atual é maior do que o valor da chave procurada x (isso é possível devido à ordenação) e **iii**. o valor da chave do nó atual é igual ao valor da chave procurada x. Nos casos **i** e **ii**, a chave procurada não existe na lista e por essa razão ela não foi encontrada.

```
Node* search(int x) {
    Node* current = head;
    while (current != nullptr && current->key < x) {
        current = current->next;
    }
    if (current != nullptr && current->key == x) {
        return current;
    }
    return nullptr;
}
```

Em termos de desempenho, esta última versão da busca é levemente superior, pois não testa se chave de valor x foi encontrada em cada iteração do **while**. Para deixar isso mais claro, na primeira versão de `search`, descrita neste documento, se durante a busca forem executadas 100 iterações do **while**, serão executadas 100 verificações para determinar se a chave foi encontrada. Na segunda versão, idêntica à apresentada para a lista simplesmente encadeada, esta verificação ocorre uma única vez.

Uma observação importante é que até o momento foram apresentadas propostas de código em que a busca é efetuada no sentido direto, ou seja, a busca pela chave é realizada do início para o final da lista. A lista duplamente encadeada nos fornece uma flexibilidade maior de movimentação, devido à presença do ponteiro adicional `previous` em cada um dos nós da lista e a existência do ponteiro `tail`, que aponta para o último nó da lista. Assim, também é possível efetuar o procedimento de busca por uma chave x na lista duplamente encadeada no sentido reverso, em que o percurso de busca é efetuado do final para o início da lista, conforme apresentado no código a seguir deixado para análise.

```
Node* searchReverse(int x) {
    Node* current = tail;

    while (current != nullptr && current->key > x) {
        current = current->previous;
    }

    // Após sair do laço, verifica se encontrou o nó
    if (current != nullptr && current->key == x)
        return current;

    return nullptr; // Não encontrado
}
```

3.2 Operação de Inserção (*insertOrdered*)

Para inserir em uma lista duplamente encadeada ordenada, é necessário encontrar a posição adequada para efetuar a inserção do novo elemento. Dessa maneira, devemos considerar as seguintes possibilidades: **i**. a lista estar vazia; **ii**. o novo elemento deve ser inserido no início da lista não vazia; **iii**. o novo elemento deve ser inserido no final da lista ou **iv**. deve ser inserido em alguma posição intermediária da lista. Para identificar se a inserção ocorrerá em algum dos dois últimos casos (**iii** e **iv**), é necessário realizar uma busca pelo valor da chave que se deseja inserir na lista. O código de inserção apresentado a seguir permite que a lista duplamente encadeada armazene elementos ou nós com chaves repetidas. Em muitas aplicações isso não será permitido.

```
void insertOrdered(int x) {
    Node* newNode = new Node(x);

    // Caso 1: Lista vazia
    if (head == nullptr) {
        head = tail = newNode;
        return;
    }

    // Caso 2: Inserção antes da cabeça
    if (x < head->key) {
        newNode->next = head;
        head->previous = newNode;
        head = newNode;
        return;
    }

    // Caso 3: Inserção no meio ou no final
    Node* current = head;
    while (current != nullptr && current->key < x) {
        current = current->next;
    }

    // Inserção no final
    if (current == nullptr) {
        newNode->previous = tail;
        tail->next = newNode;
        tail = newNode;
    }
    // Inserção no meio
    else {
        newNode->next = current;
        newNode->previous = current->previous;
        current->previous->next = newNode;
        current->previous = newNode;
    }
}
```

Analisando o código de inserção apresentado, inicialmente o novo nó com valor de chave igual a x é criado. Em seguida, verifica-se se a lista está vazia (`head == nullptr`). Se a lista estiver vazia, os ponteiros `head` e `tail` deverão apontar para o novo nó, que será, após a sua inserção, o único elemento presente na lista duplamente encadeada. Os ponteiros `next` e `previous` do novo elemento devem apontar para nulo.

Caso a lista não se encontre vazia, o algoritmo prossegue e verifica na sequência se o elemento deve ser inserido na primeira posição da lista. Para que isso ocorra, o valor da chave x do novo elemento deve ser menor do que a chave do primeiro elemento atualmente na lista (`x < head->key`). Nesse caso, o ponteiro `next` do novo nó deve apontar para o elemento apontado por `head`, o ponteiro `previous` do nó apontado por `head` deve apontar para o novo nó e o ponteiro `head` apontará para o novo nó primeiro elemento da lista.

Se a lista não estiver vazia e o novo elemento não será inserido no início da lista duplamente encadeada, então será necessário efetuar uma busca com o suporte do ponteiro `current` para determinar a posição correta da inserção do novo elemento. Ao sair da execução da estrutura de repetição **while**, se o valor do nó corrente for nulo (`current == nullptr`), isso significa que todos os elementos da lista foram percorridos e o novo elemento deve se tornar o último nó da lista duplamente encadeada. Assim, o ponteiro `previous` do novo nó deve apontar para o ponteiro `tail`, o ponteiro `next` do nó apontado por `tail` deve apontar para o nó do novo elemento e o ponteiro `tail` deverá apontar para o novo nó inserido no final da lista.

Em último caso, o novo nó será inserido entre dois elementos da lista, em uma posição intermediária. Assim, o ponteiro `current` sai da estrutura **while** apontando para o elemento posterior do novo nó na lista. Logo, o ponteiro `next` do novo nó apontará para o nó apontado por `current`; o ponteiro `previous` do novo nó deve apontar para o ponteiro `previous` do nó apontado por `current`; o ponteiro `next` do nó anterior ao nó apontado por `current` deve apontar para o novo nó e o ponteiro `previous` do nó apontado por `current` deve apontar também para o novo nó. Desse modo, a inserção é concluída adequadamente.

3.3 Operação de Remoção (*removeOrdered*)

Na operação de remoção, deve-se buscar pelo nó que possui a chave x que se deseja remover. Ao encontrar esse nó, ele deve ser retirado da lista a partir do ajuste adequado de alguns dos ponteiros presentes na lista. Após o ajuste, a memória alocada dinamicamente para o nó removido deve ser liberada. O código a seguir realiza a remoção em uma lista ordenada duplamente encadeada.

```
void remove(int x) {
    Node* nodeToRemove = search(x);
    if (nodeToRemove == nullptr) {
        cout << "Elemento nao encontrado.\n";
        return;
    }

    // Caso 1: Remover o primeiro elemento
    if (nodeToRemove == head) {
        head = head->next;
        if (head != nullptr)
            head->previous = nullptr;
        else
            tail = nullptr; // lista ficou vazia
    }
    // Caso 2: Remover o último elemento
    else if (nodeToRemove == tail) {
        tail = tail->previous;
        tail->next = nullptr;
    }
    // Caso 3: Remover elemento do meio
    else {
        nodeToRemove->previous->next = nodeToRemove->next;
        nodeToRemove->next->previous = nodeToRemove->previous;
    }

    delete nodeToRemove;
}
```

Inicialmente, realiza-se a busca pelo nó que possui a chave x que se deseja remover da lista. Como visto anteriormente, a busca retornará nulo se a chave não estiver na lista. Dessa forma, logo após o encerramento da busca, o algoritmo verifica a existência da chave na lista duplamente encadeada, testando se `nodeToRemove == nullptr`. Caso essa igualdade seja verdadeira, então a remoção não ocorre e o procedimento se encerra, pois a chave x não foi encontrada.

Se a chave x tiver sido encontrada, existem então três possibilidades: a chave está no primeiro nó da lista, a chave está no último nó da lista ou a chave está em um nó intermediário da lista. O código fornecido verifica se a chave está no primeiro nó da lista (`nodeToRemove == head`). Neste caso, o nó é removido fazendo com que o ponteiro `head` aponte para o mesmo endereço de memória apontado pelo ponteiro `next` do nó apontado por `head`. Se o ponteiro `head` for diferente de nulo, isso indica que a lista não se tornou vazia com a remoção, então o ponteiro `previous` do nó apontado por `head` passa a ser igual a nulo. Lembre-se que o ponteiro `previous` do primeiro elemento da lista duplamente encadeada deve ser nulo. Se o ponteiro `head` tiver se tornado nulo, pode-se afirmar que a lista se tornou vazia após a remoção. Neste cenário, o ponteiro `tail` também deve se tornar nulo.

Após verificar que a chave x não se encontra no primeiro nó da lista, o algoritmo apresentado verifica se a chave se encontra no último nó da lista. Para isso, testa-se se o ponteiro `nodeToRemove` é igual ao ponteiro `tail`. Lembre-se que o ponteiro `tail` tem a função de apontar para o último elemento da lista duplamente encadeada. Neste caso, o nó de chave x é removido fazendo com que o ponteiro `tail` aponte para o nó anterior (`tail = tail->previous`) e o ponteiro `next` do novo último elemento da lista aponte para nulo (`tail->next = nullptr`).

Caso o nó a ser removido não seja o primeiro e nem o último da lista, ele estará em alguma posição intermediária da lista, entre dois outros nós. Nesta situação, o ponteiro `next` do nó anterior ao nó a ser removido irá passar a apontar para o nó posterior ao nó que será retirado da lista (`nodeToRemove->previous->next = nodeToRemove->next`) e o ponteiro `previous` do nó posterior ao nó a ser removido irá apontar para o nó anterior ao nó que será retirado da lista (`nodeToRemove->next->previous = nodeToRemove->previous`). No final da remoção, independente do posicionamento do nó retirado da lista, ele será desalocado da memória (`delete nodeToRemove`).

3.4 Observações

Os três procedimentos apresentados são de complexidade assintótica linear, ou seja, $O(n)$. Isso ocorre porque em todos os seus algoritmos é necessário efetuar uma busca na lista simplesmente encadeada, seja para encontrar uma chave de valor x, ou para encontrar o local adequado para efetuar a inserção desta chave. Nessa busca, existe a possibilidade de que seja necessário percorrer todos os n elementos presentes na lista. Deve-se destacar que se a listar for ordenada de forma crescente ou decrescente, o tempo médio de busca é reduzido.

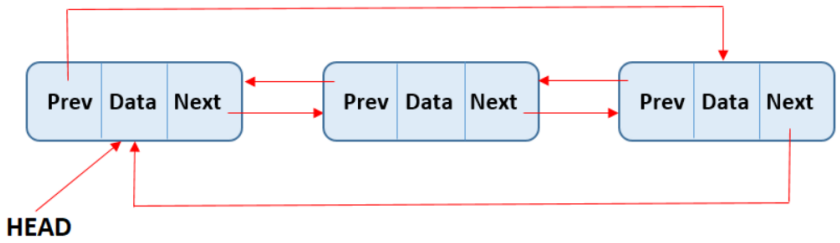
A estrutura de uma lista duplamente encadeada pode servir como base para a implementação de pilhas, filas e deques de maneira encadeada. Para isso, é necessário considerar as características dessas estruturas. Nas pilhas os seus elementos são empilhados e desempilhados apenas de seu topo (início ou final da lista, a depender da implementação). Dessa maneira, as operações de inserção (*push*) e remoção (*pop*) tornam-se mais simples, apresentando complexidade constante $O(1)$.

Nas filas, os elementos são enfileirados no final da lista e são desenfileirados de seu início. Isso simplifica as operações de inserção (*enqueue*) e de remoção (*dequeue*), as quais passam a apresentar complexidade constante $O(1)$. Nos deques, estrutura também conhecida do inglês como *double-ended* ou fila de duas extremidades, as inserções de elementos ocorrem tanto no início quanto no final da lista.

Da mesma maneira, as remoções devem ocorrer no início ou no final da lista. Assim, não há a necessidade de realizar o procedimento de busca e o acesso às posições de inserção ou remoção ocorrem com complexidade $O(1)$, pois os ponteiros `head` e `tail` garantem o acesso direto a essas posições.

4. Lista Duplamente Encadeada Circular

A lista duplamente encadeada circular é uma estrutura de dados em que cada um de seus nós contém ponteiros para o nó anterior (`previous`) e para o próximo nó (`next`), da mesma maneira que uma lista duplamente encadeada comum. A diferença é que em sua versão circular o ponteiro `next` do último nó da lista aponta para o primeiro elemento da lista e o ponteiro `previous` do primeiro nó aponta para o último elemento da lista. A figura a seguir facilita o entendimento desta estrutura. Note, que não é mais necessário ter o ponteiro `tail`, uma vez que o último elemento pode ser acessado por meio de `head->previous`.



Alguns pontos positivos da lista duplamente encadeada circular são:

- **Deslocamento contínuo**: não há início ou fim definidos, sendo possível percorrer a lista indefinidamente em qualquer direção. A navegação na lista pode ser cíclica, o que é útil para aplicações que precisam voltar ao início automaticamente.
- **Facilidade de inserção e remoção**: não é necessário tratar casos especiais para as inserções ou remoções no início ou final da lista. Isso ocorre porque todos os nós possuem um nó anterior e um nó posterior válidos. Observe que inserir após o último nó é a mesma coisa que inserir antes do primeiro. Para remover o primeiro ou o último nó não é necessário testar nulo (`nullptr`).
- **Navegação bidirecional**: é possível avançar ou retroceder na lista a partir dos ponteiros `next` e `previous`, respectivamente, que existem nos nós. Assim, é possível percorrer a lista no sentido direto e no sentido reverso;
- **Inexistência de ponteiros nulos**: nenhum dos ponteiros existentes nos nós é nulo e da estrutura é nulo, exceto quando a lista é vazia. Quando a lista se encontra vazia, o ponteiro `head` é nulo. Essa característica evita possíveis erros de tentativa de acesso a ponteiros nulos, eliminando a necessidade de verificações extras ao percorrer e efetuar operações na lista.