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 Simplesmente Encadeadas
(EDL – TADS4M)

1. Introdução

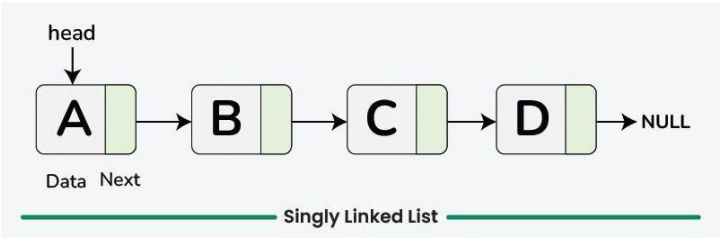
Nas aulas anteriores trabalhamos com estruturas de dados implementadas em vetores, as chamadas estruturas de dados sequenciais. Neste contexto, estudamos as listas, pilhas e filas. Porém, estas mesmas estruturas podem armazenar seus dados na memória de uma maneira diferente, sem a necessidade de que seus elementos estejam posicionados em endereços contíguos de memória. Assim, temos as chamadas estruturas encadeadas ou ligadas. Vamos iniciar nosso estudo das estruturas encadeadas pela **lista simplesmente encadeada ou lista encadeada simples**.

Uma **lista simplesmente encadeada** é uma estrutura de dados dinâmica formada por uma sequência de nós, onde cada nó armazena uma **chave** que o identifica, possivelmente outros dados referentes a esta chave, e um **ponteiro para o próximo nó** da lista. Diferente das listas sequenciais, os elementos da lista encadeada não precisam ocupar posições contíguas na memória, pois a ligação entre eles é feita por meio de ponteiros.

2. Estrutura da Lista Duplamente Encadeada

A estrutura da lista duplamente encadeada é formada, basicamente, pelos seus nós e a cabeça da lista ou *head*, do inglês. A seguir, são apresentados maiores detalhes sobre estes elementos:

- **Nós**: unidades básicas que armazenam a chave, demais dados e um ponteiro (`next`) que aponta para o próximo nó da lista;
- **Cabeça da lista (**`head`**)**: ponteiro que aponta para o primeiro nó da lista. Caso a lista esteja vazia, o ponteiro `head` é nulo;
- **Final da lista**: não há um ponteiro específico/adicional para determinar o final da lista, entretanto, o último nó da lista terá o seu ponteiro `next` nulo, indicando que a lista terminou.



A partir dessas definições, podemos resumir os ponteiros existentes na lista encadeada simples: i. o ponteiro `head` sempre aponta para o primeiro nó da lista; ii. cada nó da lista possui um ponteiro interno denominado `next` que aponta para o próximo nó da lista. Se o ponteiro `next` de um determinado nó for nulo, isso indica que este nó é o último da lista. Assim, com o auxílio destes ponteiros, é possível acessar a lista e percorrê-la, elemento a elemento.

3. Principais Algoritmos

Sabe-se que as estruturas de dados, de uma forma geral, possuem operações básicas em comum, são elas: busca, inserção e remoção. Nesta parte do documento, iremos apresentar trechos de código para essas operações nas listas simplesmente encadeadas. Para isso, vamos considerar que os elementos da lista devem ser ordenados de forma crescente. Manter a lista encadeada ordenada pelo valor das chaves de seus nós oferece alguns benefícios, por exemplo:

- **Busca mais eficiente:** embora a busca ainda tenha complexidade assintótica $O(n)$, ela pode ser interrompida mais cedo ao detectar que a chave não poderá ser mais encontrada. Isso ocorre quando o valor da chave do nó atual torna-se maior que o valor da chave buscada. Desta forma, o tempo médio de busca é reduzido;
- **Inserção organizada:** novos elementos possuem posições corretas para serem inseridos, mantendo a estrutura adequada para as operações de consultas ordenadas;
- **Facilidade para operações avançadas:** listas ordenadas podem ser úteis em algoritmos de mesclagem (*merge*) de listas, contagem acumulada, entre outros.

Ao analisarmos os trechos de código que ainda serão apresentados, veremos que nem tudo é vantagem. Inserir em ordem geralmente é **mais custoso** do que inserir em uma lista desordenada, porque exige percorrer a lista até encontrar a posição correta de inserção — $O(n)$ em lista encadeada. O ato em si de remover um elemento da lista é bastante simples, entretanto a busca pela posição do elemento é linear. Se houver muitas inserções, com muita frequência, muitas vezes é melhor usar uma estrutura como **árvore de busca binária balanceada** ou **tabela *hash***.

3.1 Operação de Busca (*Search*)

O código da operação de busca apresentado a seguir recebe como argumento o valor da chave a ser procurada na lista por meio do parâmetro x. O ponteiro `current` é utilizado para percorrer a lista encadeada simples, 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) {
        current = current->next;
    }
    if (current != nullptr && current->key == x) {
        return current;
    }
    return nullptr;
}
```

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 e o valor da chave do nó atual for menor do que x (`current->key < x`). Em seguida, é testado se a execução da estrutura de repetição foi finalizada porque a chave procurada foi encontrada (`current->key == x`). Se isso for verdadeiro, retorna-se o ponteiro `current`, que aponta para o nó que possui como chave o valor x. Caso contrário, isso significa que a busca foi realizada até o final da lista e a chave não foi encontrada, retornando nulo (`nullptr`).

3.2 Operação de Inserção (*insertOrdered*)

Para inserir em uma lista ordenada devemos considerar algumas possíveis situações. Primeiramente, como a lista é ordenada, o novo elemento deve ser inserido em uma posição adequada, de modo a manter a lista ordenada. Logo, essa posição deve ser encontrada dentro da lista. Sabendo disso, é possível que a inserção ocorra no início da lista, no final da lista ou em alguma posição intermediária. O código apresentado a seguir permite que existam chaves de igual valor na lista, assim, sempre ocorrerá a inserção. Muitas vezes, dependendo da aplicação, a repetição de chaves não é permitida.

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

    // Insert at the beginning if list is empty or smaller
    if (head == nullptr || head->key >= x) {
        newNode->next = head;
        head = newNode;
        return;
    }

    Node* prev = head;
    Node* current = head->next;

    while (current != nullptr && current->key < x) {
        prev = current;
        current = current->next;
    }

    newNode->next = current;
    prev->next = newNode;
}
```

O objetivo do código é inserir a chave x na lista encadeada ordenada. Inicialmente, o novo nó é criado com o valor de sua chave igual a x. Em seguida, a partir da estrutura de decisão **if**, verifica-se se a lista está vazia (`head == nullptr`) ou se o primeiro elemento da lista é maior ou igual que o valor da chave a ser inserida (`head->key >= x`). Caso essa condição seja satisfeita, isso indica que o novo nó deve ser inserido no início da fila e a operação de inserção é efetuada com o ajuste adequado dos ponteiros `next` do novo nó e `head`.

Se a posição de inserção do novo nó não for o início da fila, será necessário percorrer a lista em busca da posição correta de inserção. Neste sentido, os ponteiros `current` e `prev` são utilizados para percorrer a lista, um na frente do outro. O ponteiro `current` segue caminhando na frente enquanto o

final da lista não é encontrado (`current != nullptr`) e o valor da chave do nó atual/corrente seja menor que o valor de x. Ao final da execução da estrutura de repetição, o ponteiro `prev` aponta para o nó anterior à posição onde o novo nó deve ser inserido. Já o ponteiro `current` aponta para o nó seguinte à posição onde o novo nó deve ser inserido ou será igual a nulo, indicando que o novo nó será inserido no final da lista. Dessa forma, a inserção do novo nó na lista ocorre fazendo com que `next` do ponteiro `prev` aponte para o novo nó e o ponteiro `next` do novo nó aponte para `current`.

3.3 Operação de Remoção (*removeOrdered*)

Na operação de remoção, devemos realizar uma busca 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 esse ajuste, a memória que foi alocada dinamicamente para este nó deve ser liberada. A seguir, é apresentado trecho de código que realiza a remoção em uma lista ordenada simplesmente encadeada.

```
void removeOrdered(int x) {
    if (head == nullptr) return; // empty list

    // If the first node is the target
    if (head->key == x) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

    Node* prev = head;
    Node* current = head->next;

    while (current != nullptr && current->key < x) {
        prev = current;
        current = current->next;
    }

    if (current != nullptr && current->key == x) {
        prev->next = current->next;
        delete current;
    }
}
```

Inicialmente, verifica-se se a lista está vazia. Se a lista estiver sem elementos, obviamente, a chave não estará na lista e a remoção não ocorrerá. Caso exista elementos na lista, a próxima etapa é verificar se a chave se encontra em seu primeiro elemento. Se a chave x estiver no nó apontado por `head`, deve-se atualizar este ponteiro, fazendo-o apontar para `head->next`, causando a remoção. Observe que é utilizado um ponteiro temporário, `temp`, para que o valor original do endereço apontado por `head` não seja perdido. Este valor é importante para permitir a liberação da memória ocupada pelo nó que foi removido da lista encadeada (`delete temp`).

Caso a chave x não se localize no primeiro elemento da lista, será necessário procurá-la nos demais nós da lista. Essa busca assemelha-se àquela que foi aplicada no código de inserção. São utilizados dois ponteiros, `current` e `prev`, que percorrem a lista. O ponteiro `current` se desloca na frente e o ponteiro `prev` segue-o um nó atrás. O deslocamento desses ponteiros ocorre enquanto o ponteiro

`current` não chega ao final da lista e o valor da chave do nó corrente (`current->key`) for menor do que a chave procurada.

Ao término do deslocamento dos ponteiros `current` e `prev`, é necessário verificar se a chave foi encontrada. Se isso ocorrer o valor da chave do nó apontado por `current` será igual a x. Neste caso, o próximo nó de `prev` é atualizado (`prev->next = current->next`), removendo a chave da lista simplesmente encadeada, e a memória alocada para o nó atual/corrente que possui x como chave é liberada (`delete current`).

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

4. Resumo

- As listas simplesmente encadeadas é uma estrutura dinâmica formada por nós não contíguos na memória e um ponteiro `head` que aponta para o primeiro nó da lista;
- Cada nó é constituído por uma chave que o identifica, dados e um ponteiro `next`, que aponta para o próximo nó da lista;
- Operações de inserção e remoção realizadas no início são rápidas, $O(1)$, enquanto a busca, inserção e remoção em posições arbitrárias apresentam custo linear, $O(n)$;
- A ordenação da lista apresenta algumas vantagens práticas como reduzir o tempo médio de busca e consequentemente das operações de inserção e remoção. Entretanto, a complexidade dessas operações continua sendo linear.