



Aula 03:

# Listas

## Parte 1

**PCO001 – Algoritmos e Estruturas de Dados**

Prof. João Paulo Reus Rodrigues Leite

joaopaulo@unifei.edu.br

Na computação, é **fundamental** o trabalho com **conjuntos de dados**. Nossos programas vão conter **coleções** de valores numéricos, de registros de leituras de sensores, dados de pessoas, etc..

Se os dados possuem algo em comum, ficam melhor organizados se estiverem dispostos em uma mesma **estrutura**.

Existem vários tipos de estruturas, cada um com seus **pontos fortes e fracos**. E qual delas eu devo utilizar em meus programas?

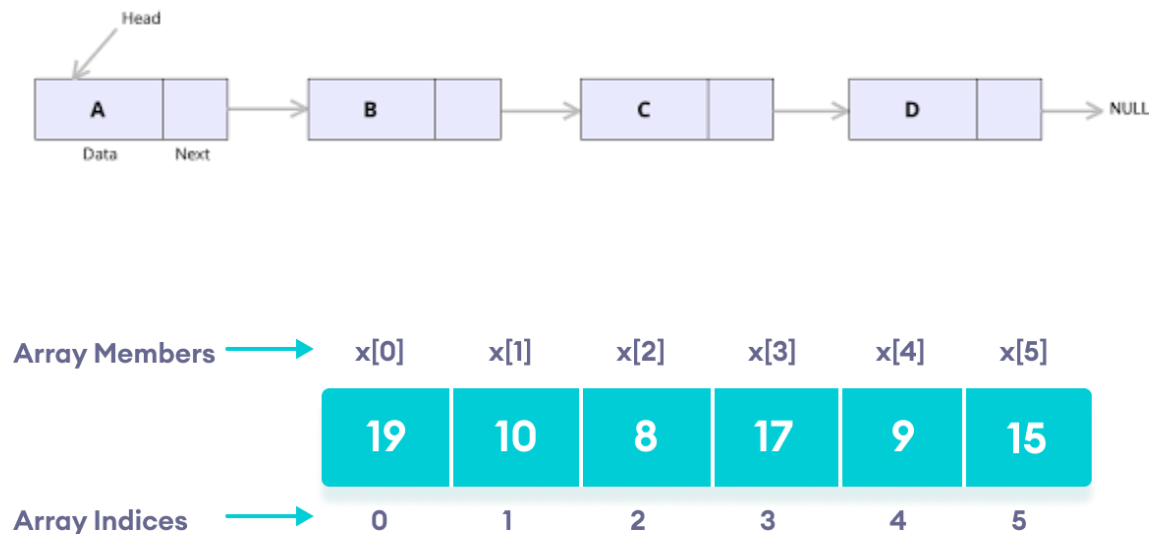
A escolha deve se basear no contexto do problema, no tipo de operação importante nele, e na **eficiência dessas operações mais utilizadas** (tempo, espaço, etc.).

As operações básicas que podem ser utilizadas para manipular uma estrutura são:

- **Inserção** de um dado;
- **Remoção** de um dado;
- **Buscar** um dado específico.

Podemos, ainda, ter operações para encontrar o maior elemento, o menor, contar os elementos, alterá-los, e assim por diante. No entanto, quase todas elas derivam das primitivas. Por isso, vamos nos concentrar nelas.

Dentre as estruturas de dados, as **listas** são as de manipulação mais simples, e representam um conjunto de dados organizados em ordem **linear**, ou unidimensional.



O tipo de armazenamento de uma lista linear pode ser classificado de acordo com a **posição relativa de dois nós consecutivos** da lista **na memória**.

Quando dados subsequentes na estrutura estão **alocados contiguamente** na memória, chamamos de **alocação sequencial** (**vetores**) e, de outra maneira, quando se encontram “**espalhados**” pela **memória**, chamamos de **alocação encadeada** (**listas encadeadas**).

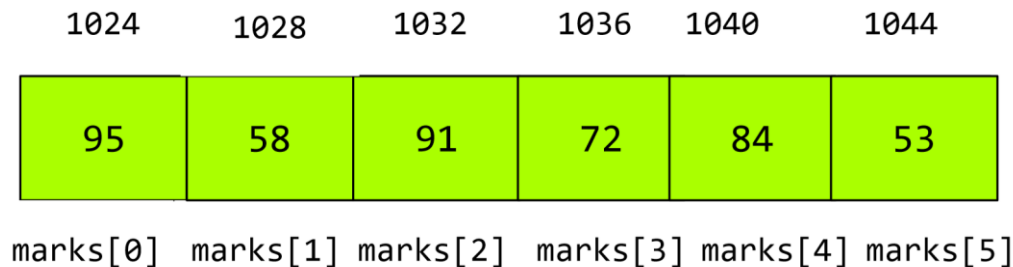
A escolha de um ou outro tipo de lista depende essencialmente das **operações que serão executadas sobre ela**.

# Resumo da Aula

Na aula de hoje veremos:

- Listas lineares **estáticas** (arrays ou vetores)
- Listas lineares **dinâmicas**
  - Lista **encadeada** simples
  - Lista **duplamente encadeada**
  - Lista encadeada **circular**

Uma **lista estática sequencial**, também conhecida como lista por contiguidade ou **vetor**, é uma coleção que armazena uma sequência de objetos, todos do mesmo tipo, em posições consecutivas da memória. Veja um vetor chamado “**marks**”, mostrado na figura abaixo:



Para criar a lista, basta utilizarmos a declaração:

```
int marks[N]; // no caso da figura, N == 6
```

Em C++, os dados poderão ser acessados através de seus **índices**, de 0 a N-1.

# Busca:

Dado um **valor inteiro**  $k$  e um **vetor de inteiros**  $\text{vet}[0 \dots N-1]$ , com  $N \geq 0$ , considere o problema de encontrar um índice  $i$  tal que  **$\text{vet}[i] == k$** .

```
int search(int vet[], int key) {  
    for(int i = 0; i < N; i++)  
        if(vet[i] == key)  
            return i;  
    return -1; // convenção: retornar -1 se não encontrar  
}
```

O algoritmo é **simples e elegante**. Funciona corretamente para qualquer arranjo, mesmo que  $N == 0$ . Para análise de sua complexidade, devemos considerar **o pior caso**, em que o valor  $\text{key}$  não é encontrado no vetor. Quando isso acontece, o laço interno faz  $N$  comparações, que resulta na **complexidade  $O(N)$** .



## Inserir um valor:

Dado um **valor inteiro**  $k$ , um **vetor de inteiros**  $vet[0 \dots N-1]$ , com  $N \geq 0$ , e uma **posição**  $j$  entre 0 e  $N-1$ , considere o problema de inserir o valor  $k$  na posição  $j$  do vetor  $vet$ .

Se for desejado apenas **atualizar o valor** do vetor  $vet$  na posição  $j$ , trata-se de uma operação de **tempo constante  $O(1)$**  dada pelo comando:

```
vet[j] = k; // insere valor k na posição  $0 \leq j \leq N-1$ 
```

No entanto, o que é mais interessante é tentarmos inserir o valor *entre os elementos das posições  $j-1$  e  $j$* , o que nos obriga a “**abrir espaço**” para o novo elemento. Isso só é possível quando o vetor não estiver cheio. Para isso, podemos manter uma variável que contém o “fim” atual do nosso vetor (índice do último elemento), sendo que  $fim < N$ .

Veja:

# Inserir um valor:

Dado um **valor inteiro**  $k$ , um **vetor de inteiros**  $vet[0 \dots N-1]$ , com  $N \geq 0$ , e uma **posição**  $j$  entre 0 e  $N-1$ , considere o problema de inserir o valor  $k$  na posição  $j$  do vetor  $vet$ .

```
bool insert(int vet[], int k, int j, int fim) {  
    if(fim == N-1)  
        return false; // Vetor cheio  
  
    int i = fim + 1;  
    while(i >= j) { // abre espaço  
        vet[i] = vet[i-1];  
        i--;  
    }  
    vet[i] = k; // insere item entre j-1 e j  
  
    return true; // inserido com sucesso  
}
```

O algoritmo é **simples**, e funciona corretamente para qualquer arranjo, desde que  $fim < N$ . Para análise de sua complexidade, devemos considerar **o pior caso**, em que o valor  $k$  é inserido na **primeira posição de um vetor cheio**. Quando isso acontece, o laço *while* faz  $N-1$  atribuições (mais a atribuição final, fora do laço), que resultam na **complexidade  $O(N)$** .

## Inserir um valor:

Dado um **valor inteiro**  $k$ , um **vetor de inteiros**  $vet[0 \dots N-1]$ , com  $N \geq 0$ , e uma **posição**  $j$  entre 0 e  $N-1$ , considere o problema de inserir o valor  $k$  na posição  $j$  do vetor  $vet$ .

```
void insert_simple(int vet[], int k, int j) {  
    int i = N-1;  
    while(i >= j) { // abre espaço  
        vet[i] = vet[i-1];  
        i--;  
    }  
    vet[i] = k; // insere item entre j-1 e j  
}
```


A versão acima é simplificada, e não considera a questão entre capacidade e quantidade de itens. Nela, o último item é “sacrificado” para que seja inserido outro. O vetor é sempre considerado “cheio”. Também tem **complexidade  $O(N)$** .

## Buscar e remover um valor:

Dado um **valor inteiro**  $k$ , um **vetor de inteiros**  $vet[0 \dots N-1]$ , com  $N \geq 0$ , considere o problema de buscar o valor  $k$  no vetor  $vet$  e removê-lo.

Ao encontrar o valor, podemos simplesmente substituí-lo por um valor convencionalizado, como -1 (em **tempo constante** - fora a busca).

No entanto, estamos interessados no caso em que os **elementos subsequentes ocupam a posição esvaziada**, de maneira análoga ao realizado na função de inserção. Como já analisamos, a busca possui complexidade linear  $O(n)$ , assim como a inserção. Como o processo é semelhante, teremos o mesmo tipo de complexidade.

5	7	8	2	3	4	6	9
<del>5</del>	7	8	2	3	4	6	9
							
7	8	2	3	4	6	9	-1

# Buscar e remover um valor:

Dado um **valor inteiro**  $k$ , um **vetor de inteiros**  $vet[0 \dots N-1]$ , com  $N \geq 0$ , considere o problema de buscar o valor  $k$  no vetor  $vet$  e removê-lo.

```
bool remove(int vet[], int k, int fim) {
    int pos = -1;
    for(int i = 0; i <= fim; i++) // encontrar elemento
        if(vet[i] == k) {
            pos = i;
            break;
        }

    if(pos == -1) return false;

    for(int i = pos; i < fim; i++) // remover elemento
        vet[i] = vet[i+1];
    vet[fim] = -1;

    return true;
}
```

Se o elemento for encontrado na primeira posição, serão necessárias **N atribuições** para removê-lo da lista e mover os demais elementos para preencher seu espaço. Caso ele seja encontrado na última posição, foram necessárias **N comparações** na busca. Portanto, de qualquer maneira, a **complexidade é  $O(n)$** .

Uma implementação interessante da **lista estática** seria com a utilização de uma *struct*. A *struct* seria responsável por manter o **vetor de dados** e uma variável com o valor atual do índice do último elemento do vetor. Veja:

```
#include <iostream>
#define N 15 // capacidade máxima da lista (15 itens)

using namespace std;

struct ListaEstatica {
    int vet[N];
    int ifinal = 0; // inicialmente não tem itens
};

// imprime vetor
void print(int vet[], int fim) {
    for(int i = 0; i < fim; i++)
        cout << vet[i] << " ";
}

int search(int vet[], int key, int fim) {
    for(int i = 0; i < fim; i++)
        if(vet[i] == key)
            return i;
    return -1; // convenção: retornar -1 se não encontrar
}
```

Dessa maneira, fica **mais organizado e fácil de acompanhar** o crescimento da estrutura estática

```

bool insert(int vet[], int k, int j, int fim) {
    if(fim == N-1)
        return false; // Vetor cheio

    int i = fim + 1;
    while(i >= j) { // abre espaço
        vet[i] = vet[i-1];
        i--;
    }
    vet[i] = k; // insere item entre j-1 e j

    return true; // inserido com sucesso
}

bool remove(int vet[], int k, int fim) {
    int pos = -1;
    for(int i = 0; i <= fim; i++) // encontrar elemento
        if(vet[i] == k) {
            pos = i;
            break;
        }

    if(pos == -1) return false;

    for(int i = pos; i < fim; i++) // remover elemento
        vet[i] = vet[i+1];
    vet[fim] = -1;

    return true;
}

```

Veja que as funções escritas anteriormente funcionam **sem modificação alguma**.

```

int main() {
    ListaEstatica lista; // estrutura lista estática
    int key = 0;

    while(key != -1) {
        cout << "Entre com o elemento a ser inserido (-1 para sair): ";
        cin >> key;
        if(key == -1) break;
        if(insert(lista.vet, key, 0, lista.ifinal))
            lista.ifinal++;
        cout << "Vetor: ";
        print(lista.vet, lista.ifinal);
        cout << "\n";
    }

    key = 0;
    while(key != -1) {
        cout << "Entre com o elemento a ser buscado (-1 para sair): ";
        cin >> key;
        if(key == -1) break;
        cout << "Encontrado na posicao: ";
        cout << << search(lista.vet, key, lista.ifinal) << endl;
    }

    key = 0;
    while(key != -1) {
        cout << "Entre com o elemento a ser removido (-1 para sair): ";
        cin >> key;
        if(key == -1) break;
        if(remove(lista.vet, key, lista.ifinal))
            lista.ifinal--;
        cout << "Vetor: ";
        print(lista.vet, lista.ifinal);
        cout << "\n";
    }

    return 0;
}

```

Apenas nas chamadas das funções, são passados os membros da struct definida (*vet* e/ou *ifinal*).

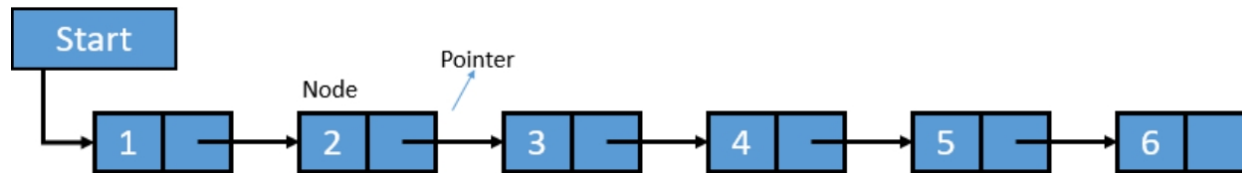
**O código fica mais limpo e a utilização da estrutura fica mais segura.**



A representação de listas lineares estáticas, alocadas contiguamente na memória (vetores), apresenta **dois problemas principais**:

1. **O número máximo de itens na lista é limitado** pelo tamanho do vetor (sua capacidade é constante).
2. Algumas operações primitivas implementadas para representação por contiguidade **comprometem o desempenho da lista** por representarem um grande esforço computacional. Por exemplo, para **a inserção de um elemento no início de uma lista** com 2000 elementos, seria necessário deslocar 2000 nós em uma posição para abrir espaço para o novo item (fora a atribuição do nó na posição zero).

Para resolver os dois problemas, passamos a utilizar uma **lista encadeada dinâmica**. Nessa estrutura, cada um dos elementos (chamaremos de nós, ou **node**, em inglês) armazena um dado e um ponteiro para o próximo elemento da lista. Veja:



Como **cada elemento contém o endereço de memória para o próximo**, eles não precisam estar alocados contiguamente na memória. Podemos navegar na lista através dos links entre eles.

O encadeamento de dados irá nos auxiliar a **melhorar o desempenho em operações de inserção e remoção**. Já a utilização de ponteiros será responsável por superar a dificuldade da **capacidade fixa da estrutura**.

A principal estrutura da lista encadeada dinâmica é o nó da lista (**Node**), representado pela struct abaixo:

```
struct Node {  
    int dado; // dado armazenado no nó  
    Node* next; // endereço para próximo nó da lista  
};
```

Este tipo de estrutura é chamada de **autorreferente**, pois contém um membro ponteiro que armazena o endereço de outra estrutura do mesmo tipo.

Na função *main*, podemos manter referências para o **início e final** da lista, visando facilitar as operações primitivas:

```
Node *inicio = nullptr; // lista sem elementos  
Node *fim = nullptr; // não há último elemento
```

Portanto, como ficariam as funções para inserção/remoção no início da lista e busca por um elemento?

```

void insert_ini(Node* &ini, int elem) {
    // se a lista estiver vazia
    if(ini == nullptr) {
        Node* novo = new Node(); // cria novo nó
        novo->dado = elem;
        novo->next = nullptr;
        ini = novo; // passa a ser o inicial
    } else { // se a lista não estiver vazia
        Node* novo = new Node(); // cria novo nó
        novo->dado = elem;
        novo->next = ini; // aponta para o que era o inicial
        ini = novo; // passa a ser o inicial
    }
}

```

A inserção de um elemento no início da lista passa a ser uma operação de complexidade constante, pois independe do tamanho da lista. **Complexidade  $O(1)$ .**

Repare que o ponteiro precisa ser passado por **referência (&)**, para que possamos modificar seu valor dentro da função.

```

int search(Node *ptr, int key) {
    int index = 0;

    while(ptr != nullptr) {
        if(ptr->dado == key) return index; // se é o buscado, retorne sua posição
        index++; ptr = ptr->next; // caminha para o próximo nó
    }

    return -1; // não encontrado
}

```

A busca por um elemento da lista **continua linear**, uma vez que o elemento pode não estar na lista, e isso precisar ser checado  $n$  vezes. **Complexidade  $O(n)$ .**

E a **remoção**? Se formos remover um elemento do início do vetor, a remoção é independente do tamanho da estrutura, e possui **complexidade  $O(1)$** .

```
void remove_ini(Node* &ini) {  
    if(ini != nullptr) { // se houver algum elemento na lista  
        Node* aux = ini; // guarda endereço de ini  
        ini = ini->next; // próximo elemento passa a ser o inicial  
        delete aux; // libera memória  
    }  
}
```

Caso seja necessário buscar o elemento, a operação passa a ter **complexidade  $O(n)$** .

```
void remove_elem(Node* &ini, int elem) {  
    Node* prev = nullptr;  
    Node* aux = ini;  
  
    // caminha na lista procurando elemento  
    while(aux != nullptr && aux->dado != elem) {  
        prev = aux;  
        aux = aux->next;  
    }  
  
    if(aux != nullptr) { // encontrou elemento  
        if(aux == ini)  
            ini = ini->next; // era o primeiro  
        else  
            prev->next = aux->next; // não era o primeiro  
        delete aux; // libera memória  
    }  
}
```

# Exercícios

- Modifique o código para que a lista encadeada **mantenha** os dados inseridos sempre **em ordem**. Nesse caso, qual a complexidade da inserção?
- Da maneira como está, a inserção na última posição ainda é  $O(n)$ , pois implica na navegação através da lista encadeada. Como podemos resolver isso e fazer **como que a inserção/remoção do último elemento sejam também em  $O(1)$** ?