Curso: Ciência da Computação

Disciplina: Estruturas de Dados

Profª Luciana Ap. Oliveira Betetto

Aula 03 - Listas

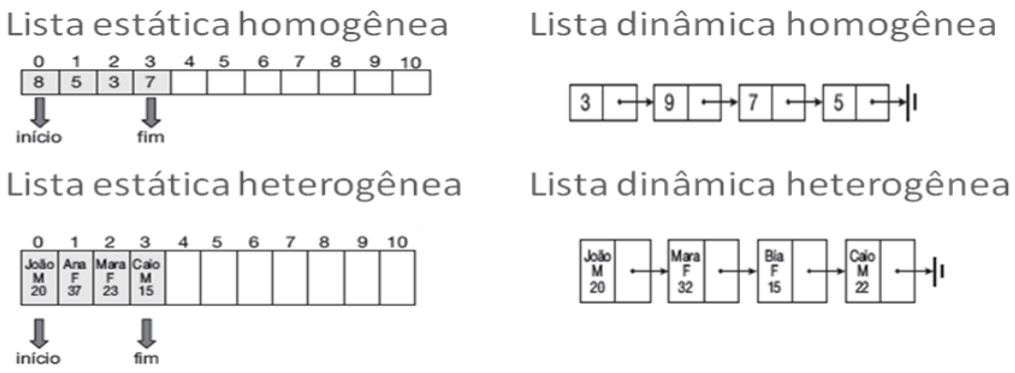
1. Introdução

Na representação de um conjunto de dados pode-se usar vetores. No entanto, o vetor não é uma estrutura de dados muito flexível, pois precisamos dimensioná-lo com um número máximo de elementos. Se o número de elementos que precisarmos armazenar exceder a dimensão do vetor, teremos um problema, pois não existe uma maneira simples e barato (computacionalmente) para alterar a dimensão do vetor em tempo de execução. Por outro lado, se o número de elementos que precisarmos armazenar no vetor for muito inferior à sua dimensão, estaremos subutilizando o espaço de memória reservado.

A solução nesses casos consiste em utilizar **estruturas** de dados que **cresçam** conforme precisarmos armazenar novos elementos, e **diminuam** conforme precisarmos retirar elementos armazenados anteriormente. Essas estruturas são chamadas **dinâmicas** e armazenam cada um dos seus elementos por **alocação dinâmica**.

Uma estrutura de dados do tipo ***lista*** representa um conjunto de dados organizados em ordem linear. Quando a estrutura lista é representada por um arranjo, ou seja, é feita a utilização de vetores na representação, tem-se o uso de endereços contíguos de memória do computador e a ordem linear é determinada pelos índices do vetor, o que em algumas situações exige um maior esforço computacional. Tal representação denomina-se **lista estática**. Quando a estrutura lista é representada por elementos que, além de conter o dado, possuem também um ponteiro para o próximo elemento, ou seja, elementos encadeados, tem se a representação denominada **lista dinâmica**. Toda lista dinâmica tem pelo menos um ponteiro para o início.

Quando um elemento de uma lista possui apenas um dado primitivo, como um número, chama se **lista homogênea**, e quando um elemento de uma lista contém um dado composto, como nome e salário de um funcionário, chama se **lista heterogênea**. As figuras abaixo ilustram listas estáticas e dinâmicas, homogêneas e heterogêneas.



De uma forma geral, essas estruturas são chamadas de listas encadeadas. A escolha por uma representação dependerá da aplicação. Algumas delas são:

- Lista Simplesmente Encadeada e não Ordenada
- Lista Simplesmente Encadeada e Ordenada
- Lista Duplamente Encadeada e não Ordenada
- Lista Duplamente Encadeada e Ordenada
- Lista Circular

Em quaisquer representações deve-se ter em mente que a **cada novo elemento do conjunto a ser armazenado deverá ser reservado um espaço na memória para a sua ocupação**. Como a cada momento pode-se fazer essa alocação de memória não se pode garantir que os elementos armazenados ocupem um espaço de memória contíguo, como aconteceria com vetores. Dessa forma, é necessário sempre referenciar a lista e seu próximo elemento; caso contrário, perde-se o acesso à lista.

2. Categorias de Alocação de Memória

Ao desenvolver uma implementação para listas lineares, o primeiro problema que surge é: **como podemos armazenar os itens da lista na memória do computador?** Sabemos que, antes de executar um programa, o computador precisa carregar seu código executável para a memória. Nesse momento, uma parte da memória total disponível fica livre, pois raramente um programa ocupa toda a memória instalada no computador.

Da área de memória que é reservada para uso do programa, uma parte é destinada ao armazenamento de instruções e o restante ao armazenamento de dados. Quem determina quanto de memória será usado para instruções é o compilador. Alocar áreas para armazenar dados, porém, é responsabilidade do programador. Sob o ponto de vista do programador, a alocação de memória pode ser classificada:

	Sequencial	**Encadeada**
Estática	Estática Sequencial	Estática Encadeada
Dinâmica	Dinâmica Sequencial	Dinâmica Encadeada

2.1. Alocação Estática X Alocação Dinâmica

Na alocação estática, a quantidade de memória destinada ao armazenamento de dados é determinada no momento da compilação e, durante a execução do programa, essa quantidade permanece constante. Por outro lado, na alocação dinâmica, a quantidade de memória destinada ao armazenamento de dados varia enquanto o programa é executado.

Considerando que uma variável é essencialmente uma área de memória, dizemos que sua alocação é *estática* se sua existência é prevista (declarada) no código fonte do programa; caso contrário, dizemos que sua alocação é *dinâmica*.

2.2. Alocação Sequencial X Alocação Encadeada

A forma mais natural de armazenar uma lista L no computador consiste em alocar seus itens em células de memória consecutivas, um após o outro. A maior vantagem no uso de *alocação sequencial* de memória para armazenar uma lista linear é que, dados o endereço inicial da área alocada e o índice de um item da lista, podemos acessar esse item rapidamente. O ponto fraco dessa forma de alocação surge quando precisamos inserir ou remover itens. Neste caso, um grande esforço será necessário para movimentar outros itens já armazenados na lista, de modo a abrir espaço para o item a ser inserido, ou a ocupar o espaço liberado pelo item que foi removido.

.....	x_1	x_2	x_3	....	x_n

Figura 1. Esquema de alocação sequencial.

Em vez de manter os itens da lista linear agrupados em células de memória consecutivas, na *alocação encadeada*, os itens podem ocupar células espalhadas por toda a memória. Neste caso, para preservar a relação de ordem linear, junto com cada item é armazenado o endereço do próximo item da lista.

Assim, no esquema de alocação encadeada, os itens são armazenados em blocos de memória denominados *nós*. Cada nó é composto por dois campos, sendo um para guardar o item x e outro para guardar o endereço do nó que contém o item sucessor de x.

Dois endereços especiais devem ser destacados:

- o endereço do **primeiro nó da lista**, representado por $\boldsymbol{L}$
- o endereço nulo, que indica o **final da lista**, representado por **Ø**

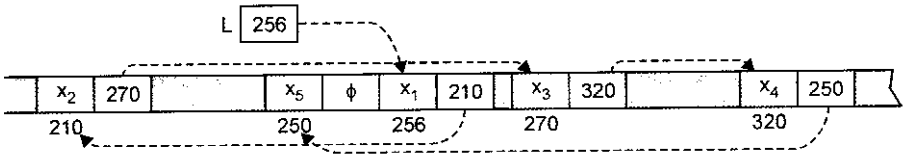


Figura 2. Esquema de alocação encadeada.

A maior vantagem da alocação encadeada é a facilidade que ela oferece para inserção e remoção de itens. Como os itens não precisam estar armazenados em nós consecutivos durante uma operação de remoção ou inserção, nenhum item da lista precisa ser movimentado; basta atualizar o campo de ligação do nó que armazena o item predecessor daquele a ser inserido ou removido.

Por outro lado, como apenas o primeiro item da lista pode ser acessado diretamente a partir de L, a grande desvantagem da alocação encadeada surge quando precisamos acessar uma posição arbitrária da lista. Neste caso, devemos partir do primeiro item e seguir os

campos de ligação, um a um, até alcançar a posição desejada. Em listas muito extensas, essa operação pode ter alto custo em relação a tempo.

Exercícios:

1. Defina sucintamente, cada categoria de alocação de memória: estática sequencial, estática encadeada, dinâmica sequencial e dinâmica encadeada.

2. Por que as listas devem ser implementadas na forma sequencial, quando acessar um item arbitrário for uma operação muito requisitada?

3. Por que os valores L e Ø são considerados especiais no esquema de alocação encadeado apresentado na Figura 2?

3. Lista Simplesmente Encadeada e não Ordenada

Numa lista encadeada, para cada novo elemento inserido na estrutura, alocamos um espaço de memória para armazená-lo. Desta forma, o espaço total de memória gasto pela estrutura é proporcional ao número de elementos nela armazenado. No entanto, não podemos garantir que os elementos armazenados na lista ocuparão um espaço de memória contíguo, portanto não temos acesso direto aos elementos da lista. Para que seja possível percorrer todos os elementos da lista, devemos explicitamente guardar o encadeamento dos elementos, o que é feito armazenando-se, junto com a informação de cada elemento, um ponteiro para o próximo elemento da lista. A Figura 3 ilustra o arranjo da memória de uma lista encadeada.

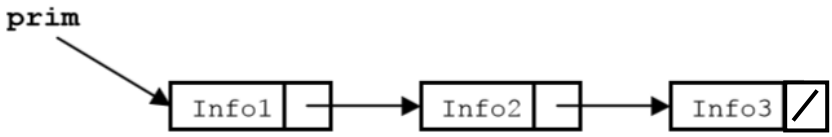
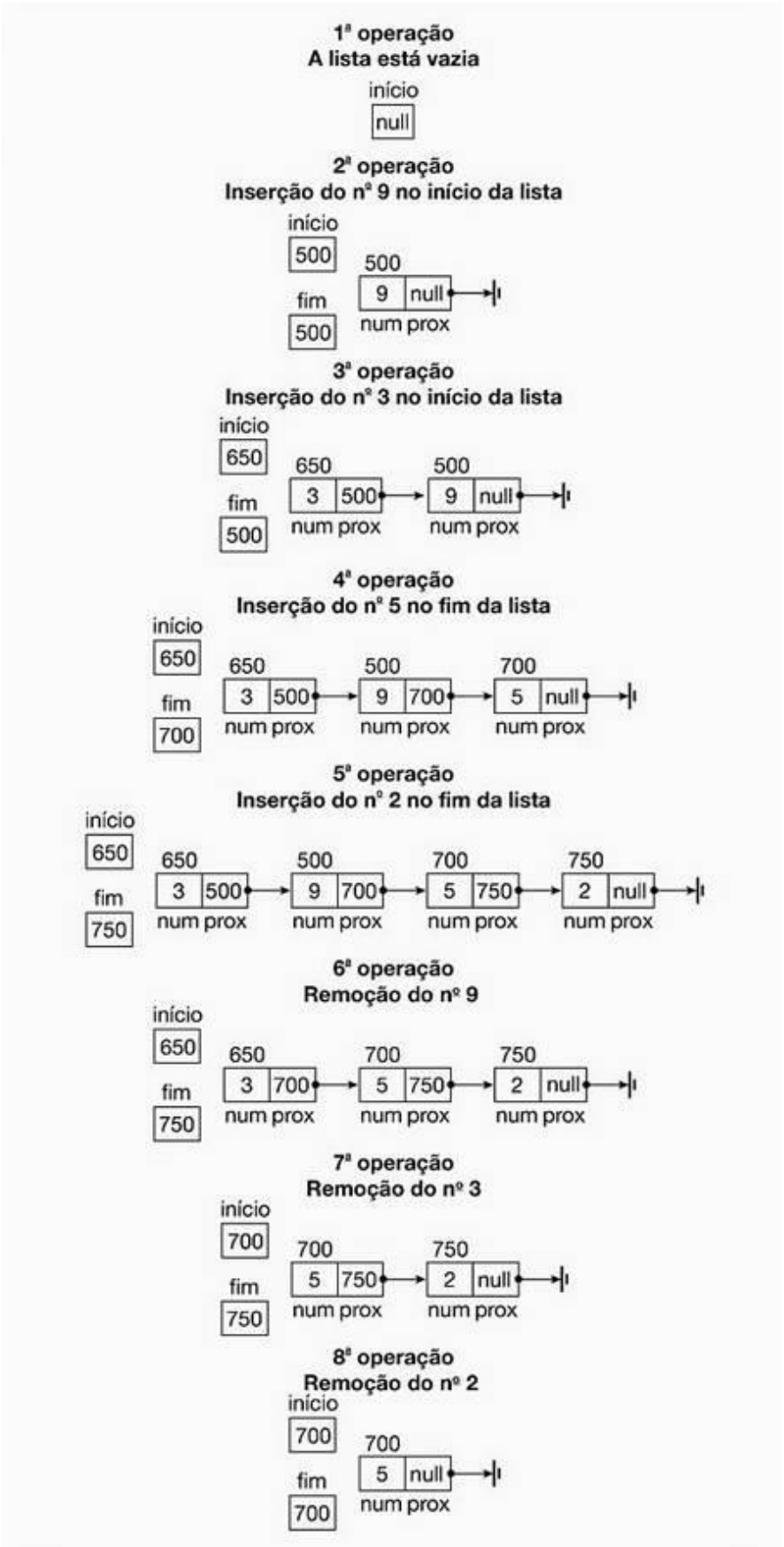


Figura 3. Arranjo da memória de uma lista encadeada.

A estrutura consiste numa sequência encadeada de elementos, em geral chamados de nós da lista. Observa-se na Figura 3 a existência de um ponteiro (*prim*) para o primeiro nó da lista que contém um valor (*Info1*) e um ponteiro para o próximo nó seguindo o encadeamento, e assim por diante. O último nó aponta para *NULL*, sinalizando que não existe um próximo elemento.

Devemos notar que trata se de uma estrutura auto-referenciada, pois, além do campo que armazena a informação (no caso, um número inteiro), há um campo que é um ponteiro para uma próxima estrutura do mesmo tipo. Considerando a definição de Lista, podemos definir as principais funções necessárias para implementarmos uma lista encadeada: inserir no início da lista, inserir no fim, consultar toda a lista, remover um elemento qualquer e esvaziá-la.

Abaixo, segue ilustração com endereços de memória meramente ilustrativos.



Operações Básicas em Listas:

3.1. Inicialização da Lista

A função que inicializa uma lista deve criar uma lista vazia, sem nenhum elemento. Como a lista é representada pelo ponteiro para o primeiro elemento, uma lista vazia é representada pelo ponteiro *NULL*, pois não existem elementos na lista. A função tem como valor de retorno a lista vazia inicializada, isto é, o valor de retorno é *NULL*.

3.2. Inserção de elementos na Lista

Uma vez criada a lista vazia, podemos inserir novos elementos nela. Para cada elemento inserido na lista, devemos alocar dinamicamente a memória necessária para armazenar o elemento e encadeá-lo na lista existente. A função de inserção mais simples insere o novo elemento no início da lista.

Cada elemento a ser colocado na lista deve ser armazenado primeiramente em uma estrutura que representa o nó da lista. Esse nó possui dois campos: uma para guardar o elemento e outro para guardar um ponteiro, que inicialmente aponta para *NULL*.

A função de inserção aloca dinamicamente o espaço para armazenar o novo nó da lista, guarda a informação no novo nó e faz este nó apontar para (isto é, ter como próximo elemento) o elemento que era o primeiro da lista. A função então retorna o novo valor que representa a lista, que é o ponteiro para o novo primeiro elemento.

Quando for necessário fazer várias inserções no início de uma lista deve criar o nó *Novo* com seu novo elemento (conteúdo), fazer o campo *prox* do nó recém criado apontar para onde *prim* está apontado, ou seja, para o primeiro elemento da lista, e depois fazer o ponteiro da lista (*prim*) apontar para o nó recém criado.

A Figura 4 ilustra a operação de inserção do nó *Novo* no início da lista.

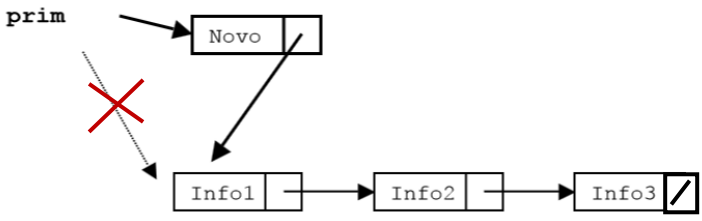


Figura 4. Inserção de um novo elemento no início da lista.

OBS: A ordem das operações é importante para não se perder a lista. Se apontarmos primeiro *prim* para *Novo*, perderemos toda a lista já armazenada, já que o campo *prox* de *Novo* não saberá para onde apontar.

3.3. Percorrendo os elementos da lista

Em situações de busca por um valor, ou para impressão de uma lista, deve-se percorrer a lista a partir do primeiro elemento até onde for desejado (até achar o elemento ou chegar no fim).

3.4. Verificando se lista está vazia

Pode ser útil implementarmos uma função que verifique se uma lista está vazia ou não. A função recebe a lista e retorna 1 se estiver vazia ou 0 se não estiver vazia. Como sabemos, uma lista está vazia se seu valor é $NULL$.

3.5. Buscando um elemento

Em situações de busca por um valor ou para impressão de uma lista, deve-se percorrer a lista a partir do primeiro elemento até onde for desejado (até achar o elemento ou chegar ao final). A função recebe a informação referente ao elemento que queremos buscar e fornece como valor de retorno o ponteiro do nó da lista que representa o elemento. Caso o elemento não seja encontrado na lista, o valor retornado é $NULL$.

3.6. Remoção de um elemento da lista

Para remover um elemento x da lista, primeiro é necessário localizá-lo e sinalizá-lo. Para localizar um elemento basta fazer uma busca na lista, como feito anteriormente, e deixar esse elemento x sinalizado, bem como o elemento que está antes dele. Isso deve ser feito para que o nó que está antes de x aponte para o elemento que está depois de x. Depois do elemento estar localizado e sinalizado, faz-se o ajuste dos ponteiros e libera a memória ocupada pelo nó que possui o elemento removido. A função tem como parâmetros de entrada a lista e o valor do elemento que desejamos retirar, e deve retornar o valor atualizado da lista, pois, se o elemento removido for o primeiro da lista, o valor da lista deve ser atualizado.

A função para retirar um elemento da lista é mais complexa. Se descobrirmos que o elemento a ser retirado é o primeiro da lista, devemos fazer com que o novo valor da lista passe a ser o ponteiro para o segundo elemento, e então podemos liberar o espaço alocado para o elemento que queremos retirar. Se o elemento a ser removido estiver no meio da lista, devemos fazer com que o elemento anterior a ele passe a apontar para o elemento seguinte, e então podemos liberar o elemento que queremos retirar. Devemos notar que, no segundo caso, precisamos do ponteiro para o elemento anterior para podermos acertar o encadeamento da lista. O caso de retirar o último elemento da lista recai no caso de retirar um elemento no meio da lista.

As Figuras 5 e 6 ilustram as operações de remoção.

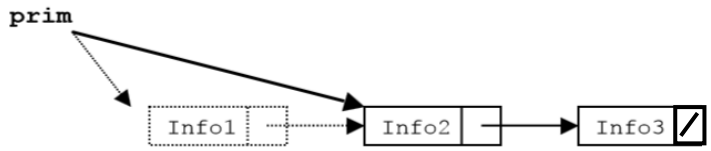


Figura 5. Remoção do primeiro elemento da lista.

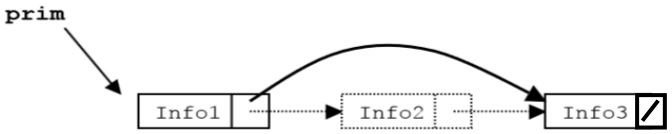


Figura 6. Remoção de um elemento no meio da lista.

3.7. Destruir uma lista

A cada inserção de elementos aloca-se memória. Ao terminar de usar a lista deve-se também liberar a memória utilizada.

A função para destruir uma lista percorre elemento a elemento, liberando-os. É importante observar que devemos guardar a referência para o próximo elemento antes de liberar o elemento corrente (se liberássemos o elemento e depois tentássemos acessar o encadeamento, estaríamos acessando um espaço de memória que não estaria mais reservado para nosso uso).

Linguagem Java:

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;
import java.util.TreeSet;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class EstruturaDados {
    private static class LISTA {

        public int num;
        public LISTA prox;
    }

    public static void main(String[] args) {

        Scanner leia = new Scanner(System.in);
        LISTA inicio = null;
        LISTA fim = null;
        LISTA aux;
        LISTA anterior;
        int op, numero, achou;
```

```
do {
   System.out.println("1 - Inserir no início da lista");
   System.out.println("2 - Inserir no fim da lista");
   System.out.println("3 - Consultar toda Lista");
   System.out.println("4 - Remover da Lista");
   System.out.println("5 - Esvaziar a Lista");
   System.out.println("6 - Sair");
   System.out.print("Digite sua opção: ");
   op = leia.nextInt();
   if (op < 1 || op > 6) {
       System.out.println("Operação Inválida!");
   }

   switch (op) {
       case 1:
           System.out.print("Digite o número a ser inserido no início da lista: ");
           LISTA novo = new LISTA();
           novo.num = leia.nextInt();
           if (inicio == null) {
               inicio = novo;
               fim = novo;
               novo.prox = null;
           } else {
               novo.prox = inicio;
               inicio = novo;
           }
           System.out.println("Numero inserido no início da Lista");
           break;
       case 2:
           System.out.println("Digite o numero a ser inserido ao fim da lista: ");
           novo = new LISTA();
           novo.num = leia.nextInt();
           if (inicio == null) {
               inicio = novo;
               fim = novo;
               novo.prox = null;
           } else {
               fim.prox = novo;
               fim = novo;
               fim.prox = null;
           }
           System.out.println("Numero Inserido no fim da lista!!");
           break;
       case 3:
           if (inicio == null) {
               System.out.println("Lista Vazia");
           } else {
               System.out.println("Consultando toda a lista");
               aux = inicio;
               while (aux != null) {
                   System.out.println(aux.num+ "  ");
                   aux = aux.prox;
               }
           }
           break;
       case 4:
           if (inicio == null) {
               System.out.println("Lista Vazia");
           } else {
               System.out.println("Digite o elemento a ser removido: ");
               numero = leia.nextInt();
               aux = inicio;
               anterior = null;
               achou = 0;
               while (aux != null) {
                   if (aux.num == numero) {
```

```
                                achou = achou + 1;
                                if (aux == inicio) {
                                    inicio = aux.prox;
                                    aux = inicio;
                                } else if (aux == fim) {
                                    anterior.prox = null;
                                    fim = anterior;
                                    aux = null;
                                } else {
                                    anterior.prox = aux.prox;
                                    aux = aux.prox;
                                }
                            } else {
                                anterior = aux;
                                aux = aux.prox;
                            }

                            if (achou == 0) {
                                System.out.println("Numero não encontrado!");
                            } else if (achou == 1) {
                                System.out.println("Numero removildo 1 vez");
                            } else {
                                System.out.println("Numero removildo" + achou + "vezes");
                            }
                        }

                    }
                    break;
                case 5:
                    if (inicio == null) {
                        System.out.println("Lista vazia!");
                    } else {
                        inicio = null;
                        System.out.println("Lista esvaziada");
                    }
                    break;
            }

        } while (op != 6);

    }
}
```

Linguagem C:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <ctype.h>
#include <locale.h>
#include <string.h>
#include <math.h>
#include <iostream>

using namespace std;
int main (void)
{
    setlocale(LC_ALL, "Portuguese");//habilita a acentuação para o português

    // Definindo o registro que representará cada elemento da lista
    struct LISTA
    {
        int num;
        LISTA *prox;
    };
```

```
// A lista está vazia, logo, o ponteiro início tem o valor NULL
// O ponteiro início conterá o endereço do primeiro elemento da lista
LISTA *inicio = NULL;
// O ponteiro fim conterá o endereço do último elemento da lista
LISTA *fim = NULL;
// O ponteiro aux é um ponteiro auxiliar
LISTA *aux;
//O ponteiro anterior é um ponteiro auxiliar
LISTA *anterior;
// Apresentando o menu de opções
int op, numero, achou;
do
{
    cout<<"MENU DE OPÇÕES\n";
    cout<<"\n1 - Inserir no início da lista";
    cout<<"\n2 - Inserir no fim da lista";
    cout<<"\n3 - Consultar toda a lista";
    cout<<"\n4 - Remover da lista";
    cout<<"\n5 - Esvaziar a lista";
    cout<<"\n6 - Sair";
    cout<<"\n\n\nDigite sua opção: ";
    cin>>op;
    if (op< 1 || op> 6)
        cout<<"Opção Inválida!!";
    if (op == 1)
    {
        cout<<"Digite o número a ser inserido no início da lista: ";
        LISTA *novo = new LISTA();
        cin>>novo->num;
        if (inicio == NULL)
        {
            //A lista estava vazia e o elemento inserido será o primeiro e o último
            inicio = novo;
            fim = novo;
            fim->prox = NULL;
        }
        else
        {
       //A lista já contém elementos e o novo elemento será inserido no início da lista
            novo->prox = inicio;
            inicio = novo;
        }
        cout<<"Número inserido no início da lista!!\n\n";
    }
    if (op == 2)
    {
        cout<<"Digite o número a ser inserido no final da lista: ";
        LISTA *novo = new LISTA();
        cin>>novo->num;
        if (inicio == NULL)
        {
            //A lista estava vazia e o elemento inserido será o primeiro e o último
            inicio = novo;
            fim = novo;
            fim->prox = NULL;
        }
        else
        {
       //A lista já contém elementos e o novo elemento será inserido no início da lista
            fim->prox = novo;
            fim = novo;
            fim->prox=NULL;
        }
        cout<<"Número inserido no fim da lista!!\n\n";
    }
```

```
if (op == 3)
 {
     if (inicio == NULL)
     {
         //A lista está vazia!
         cout<<"Lista vazia!!";
     }
     else
     {
         //A lista contém elementos e estes se~rao mostrados do início ao fim
         cout<<"\nConsultando toda a lista\n";
         aux = inicio;
         while (aux != NULL)
         {
             cout<<aux->num<<"  ";
             aux = aux->prox;
         }
     }
 }
 if (op == 4)
 {
     if (inicio == NULL)
     {   // A lista está vazia
         cout<<"Lista vazia!!\n\n";
     }
     else
     {
         //A lista contém elementos e o elemento a ser removido deve ser digitado
         cout<<"\nDigite o elemento a ser removido:";
         cin>>numero;
         //Todas as ocorrencias da lista, iguais ao número digitado, serão removidas
         aux = inicio;
         anterior = NULL;
         achou = 0;
         while (aux != NULL)
         {
             if (aux->num == numero)
             {
                 //O número digitado foi encontrado na lista e será removido
                 achou = achou +1;
                 if (aux == inicio)
                 {
                     //O número a ser removido é o primeiro da lista
                     inicio = aux->prox;
                     delete(aux);
                     aux = inicio;
                 }
                 else if (aux == fim)
                 {
                     //O número a ser removido é o último da lista
                     anterior->prox = NULL;
                     fim = anterior;
                     delete (aux);
                     aux = NULL;
                 }
                 else
                 {
                     //O número a ser removido está no meio da lista
                     anterior->prox = aux->prox;
                     delete(aux);
                     aux= anterior->prox;
                 }
             }
             else
             {
                 anterior = aux;
```

```
                    aux= aux->prox;
                }
            }
            if (achou == 0)
                cout<<"Número não encontrado";
            else if (achou == 1)
                cout<<"Número removido 1 vez";
            else
                cout<<"Número removido "<<achou<<" vezes";
        }
    }
    if (op == 5)
    {
        if (inicio = NULL)

        {
            //A lista está vazia
            cout<<"Lista vazia!!\n\n";
        }
        else
        {
            //A lista será esvaziada
            aux = inicio;
            while (aux != NULL)
            {
                inicio = inicio->prox;
                delete(aux);
                aux=inicio;
            }
            cout<<"Lista esvaziada\n\n";
        }

    }
    getch();
}
while (op != 6);
return 0;

}
```

4. Análise da Complexidade:

As principais operações realizadas em listas são: inserção no início da lista, inserção no fim, consulta e remoção. A seguir, serão mostradas as análises para essas operações:

Inserção no início da lista:
Para fazer a operação de inserção no início de uma lista encadeada, é necessário apenas realizar operações de atribuições atualizando o ponteiro do início. Com isso, o tempo gasto na operação é constante, *O(1)*.

Inserção no fim da lista:
Para fazer a operação de inserção no fim de uma lista encadeada, é necessário apenas realizar operações de atribuições atualizando o ponteiro do fim. Com isso, o tempo gasto na operação é constante, *O(1)*.

Consultar toda a lista:
Para realizar a operação de consultar toda a lista é necessário acessar cada elemento. Com isso, considerando que uma lista possui n elementos, o tempo necessário para tal operação é $O(n)$.

Remoção:
A remoção de um elemento em uma lista, consiste em procurá-lo e depois acertar os ponteiros para que a lista continue sendo acessada após a remoção. Na busca pelo elemento a ser removido, percorre-se, no pior caso, todos os elementos da lista, gastando com isso tempo proporcional ao tamanho dela, ou seja, $O(n)$.

Esvaziar a lista:
A operação de esvaziamento da lista consiste em remover todos os elementos dela. O tempo gasto nessa operação depende da linguagem de programação utilizada. No caso da linguagem JAVA, não é necessário realizar a remoção de cada um dos elementos, apenas atualiza-se o ponteiro para o início da lista em nulo e as memórias alocadas serão desalocadas por um procedimento JAVA. Já na linguagem C/C++, é necessário desalocar cada um dos elementos da lista, gastando tempo proporcional ao tamanho dela, ou seja, $O(n)$.

5. Listas em Python

Uma **lista** (*list*) em Python é uma sequência ou coleção ordenada de valores, contendo zero ou mais elementos. Cada valor na lista é identificado por um índice. Os valores que formam uma lista são chamados **elementos** ou **itens**. Listas são similares a strings, que são uma sequência de caracteres, no entanto, diferentemente de strings, os itens de uma lista podem ser de tipos diferentes.

A sintaxe de uma lista, em Python é:

```
lista = [1, 2, 3]
print(lista)
```
[1, 2, 3]

No exemplo acima temos uma lista de inteiros, mas uma lista pode conter floats, strings, caracteres ou outras listas.

```
lista1 = ['ana', 4, 1.13]
print(lista1)
```
['ana', 4, 1.13]

```
# inserir um elemento em determinada posição na lista:
lista[0]='julia'
print(lista)
```
['julia', 2, 3]

```
# verificar se um elemento encontra-se na lista:
lista = [1, 2, 3, 4]
num = 10
```

```
if num in lista:
        print('elemento encontrado')
else:
        print('elemento não encontrado')
elemento não encontrado
# testar com num = 3
elemento encontrado

#imprimir o tamanho da lista:
print(len(lista))
4

#imprimir todos os elementos da lista
for i in lista:
        print(i)
1
2
3
4


#acrescentar elementos numa lista
lista.append(10)
print(lista)
[1, 2, 3, 4, 10]

#acrescentar na posição 1, o elemento 20
lista.insert(1,20)
print(lista)
[1, 20, 2, 3, 4, 10]

#acrescentar na última posição, o elemento 50
lista.insert(len(lista),50)
print(lista)
[1, 20, 2, 3, 4, 10, 50]

# exibe determinado elemento através da sua posição
print(lista[0]) # exibe o elemento que se encontra no índice 0
1

# exibe qual elemento encontra-se numa posição
print(lista.index(1))
20

# exibe o último elemento da lista
print(lista[-1])
50
```

```
# exibe um intervalo de elementos da lista
print(lista[1:5])  # o último elemento do intervalo é um limite superior aberto, ou seja,
                     não será incluso na relação
[20, 2, 3, 4]

#caso não determine o último elemento do intervalo, irá imprimir a lista toda
print(lista[1: ])
[20, 2, 3, 4, 10, 50]

# determinar o número de vezes que um elemento encontra-se na lista
print(lista.count(4))
1

lista.append(4) # vamos acrescentar mais uma ocorrência do elemento 4 na lista
print(lista)
[1, 20, 2, 3, 4, 10, 50, 4]

print(lista.count(4))
2

# remover determinado elemento da lista
lista.remove(3)
print(lista)
[1, 20, 2, 4, 10, 50, 4]

lista.remove(4) # nesse caso, remove a primeira ocorrência do número 4
print(lista)
[1, 20, 2, 10, 50, 4]

# pop() remove pela posição (índice)
lista.pop(0) # nesse caso, remove o primeiro elemento da lista
print(lista)
[20, 2, 10, 50, 4]

lista.pop(len(lista) - 1) # nesse caso, remove o último elemento da lista
print(lista)
[20, 2, 10, 50]
```

Caso tente remover um elemento que não existe na lista, irá retornar ERRO!

```
# exibir a lista em ordem inversa
lista.reverse()
print(lista)
[50, 10, 2, 20]

OU
print(lista[:: -1])
[50, 10, 2, 20]
```

```
# ordenar os elementos de uma lista em ordem crescente
lista.sort()
print(lista)
```
[2, 10, 20, 50]

```
# trabalhando com duas listas
lista1 = [1, 2, 3, 4, 5]
print(lista1)
lista2 = [1, 3.14, 'ana']
print(lista2)
```
[1, 2, 3, 4, 5]
[1, 3.14, 'ana']

```
# concatenação de listas
lista3 = lista1 + lista2
print(lista3)
```
[1, 2, 3, 4, 5, 1, 3.14, 'ana']

```
# gerar uma lista que exiba os números pares até 100
pares = [num for num in range (101) if (num % 2 == 0)]
print(pares)
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100]

Tuplas são objetos como as listas, com a diferença de que as tuplas são imutáveis como strings. Uma vez criadas, não podem ser modificadas. A sintaxe de tuplas é:

```
tupla = (1,2,3)
print(tupla)
```
(1,2,3)

```
tupla[0] = 0
# a resposta será uma mensagem de ERRO
```

É possível transformar uma lista numa tupla.

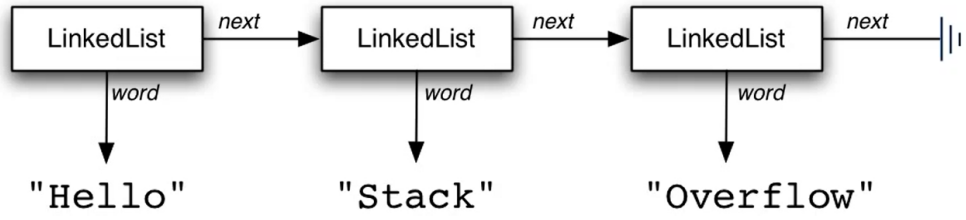
```
lista = [1, 2, 3, 4]
t_lista = tuple(lista)
print(type(t_lista))
```
<class 'tuple'>

```
print(t_lista) # para certificarmos que é tupla, irá exibir o resultado entre parênteses
```
(1, 2, 3, 4)

```
# dicionários associam pares de elementos (chave + valor)
dic = {‘marcos’ : 28 , ‘joao’ : 10}
print(dic)
print(dic[‘joao’])
```

6. Lista Simplesmente Encadeada em Python

Numa “Lista Ligada” ou “Lista Encadeada” podemos armazenar dados. Cada retângulo na imagem abaixo representa um nó ou vértice que armazena um dado (no primeiro retângulo está armazenado “Hello”, no segundo “Stack” e no terceiro “Overflow”). Todo nó contém uma referência para o próximo nó, na imagem abaixo representado pela seta (next). O último nó se liga a None (NULL - fim da lista).



Linguagem Python:

```
class Node: # criamos uma classe, um nó

        def __init__(self, label): # definimos o construtor (init), label é o nome do nó
                self.label = label
                self.next = None # um nó guarda a referência para o próximo, mas inicialmente é none(nulo)

        def getLabel(self): # método para receber o label
                return self.label

        def setLabel(self, label): # método para setar o label
                self.label = label

        def getNext(self): # função p receber o próximo nó
                return self.next

        def setNext(self, next): # função para setar os nós
                self.next = next


class LinkedList: # criamos uma classe, "Lista Linkada"

        def __init__(self):
                self.first = None # no início, o primeiro nó aponta para none
                self.last = None # no início, o último nó aponta para none
                self.len_list = 0 # variável que retorna o tamanho da lista

        def push(self, label, index): # função push insere na lista, passando label e index

                if index >= 0: # o índice não pode ser negativo

                        # cria o novo nó
                        node = Node(label) # o nó recebe um label

                        # verifica se a lista está vazia
```

```
                    if self.empty(): # se lista vazia, esse nó será o primeiro e último
                            self.first = node
                            self.last = node
                    else:

                            if index == 0:
                                    # inserção no início
                                    node.setNext(self.first) # o nó criado será o primeiro
                                    self.first = node
                            elif index >= self.len_list:
                                    # inserção no final
                                    self.last.setNext(node) # o nó criado estará no final
                                    self.last = node
                            else:
                                    #inserção no meio, então preciso percorrer a lista p inserir
                                    prev_node = self.first # irá armazenar o nó anterior
                                    # nó atual (current node) recebe o primeiro
                                    curr_node = self.first.getNext()
                                    curr_index = 1 # índice atual (current index) inicia em 1

                                    while curr_node != None:
                                            # se índice atual for igual ao índice a ser inserido
                                            if curr_index == index:
                                            #será necessário fazer os apontamentos do anterior e do próximo nós
                                                    # seta o curr_node como o próximo do nó
                                                    node.setNext(curr_node)
                                                    # seta o node como o próximo do prev_node
                                                    prev_node.setNext(node)
                                                    break
                                            # se não encontrei a posição, devo continuar percorrendo a lista
                                                    prev_node = curr_node
                                                    curr_node = curr_node.getNext() # pega o próximo da lista
                                                    curr_index += 1 # incremanta o índice

                    # atualiza o tamanho da lista
                    self.len_list += 1
def pop(self, index): # função pop remove da lista
            # se lista não estiver vazia, se índice não negativo e não for fim da lista
            if not self.empty() and index >= 0 and index < self.len_list:

                    flag_remove = False

                    if self.first.getNext() == None:
                            # possui apenas um elemento, então irei removê-lo
                            self.first = None
                            self.last = None
                            flag_remove = True # remove o nó
                    elif index ==0:
                            # remove do início, mas possui mais de um elemento
                            self.first = self.first.getNext() # o primeiro recebe o próximo
                            flag_remove = True # remove o nó
                    else:
                            '''
                            Se está nesse ponto é porque a lista possui mais
                            de um elemento e a remoção não é no início.
                            '''
                            prev_node = self.first # irá armazenar o nó anterior
                            curr_node = self.first.getNext() # nó atual (current node) recebe o primeiro
                            curr_index = 1

                            while curr_node != None:

                                    if index == curr_index:
                                            # o nó anterior aponta para o próximo do nó corrente
                                            prev_node.setNext(curr_node.getNext())
                                            curr_node.setNext(None)
                                            flag_remove = True # remove o nó
                                            break
                                    # continuo percorrendo a lista
                                    prev_node = curr_node
                                    curr_node = curr_node.getNext()
                                    curr_index += 1

                    if flag_remove:
```

```
                        # atualiza o tamanho da lista
                        self.len_list -= 1 # decrementa a lista de nós

    def empty(self): # função que verifica se a lista está vazia
            if self.first == None:
                    return True
            return False

    def leng(self): # função que retorna o tamanho da lista
            return self.len_list

    def show(self): # função para exibir os nós da lista

            curr_node = self.first

            while curr_node != None: # enquanto não for fim da lista
                    print(curr_node.getLabel(), end=' ') # imprime os nós na mesma linha, separando por espaço
                    curr_node = curr_node.getNext() # continua percorrendo a lista
            print(' ')
```

#Testando o Algoritmo:

```
lista = LinkedList()
lista.push('Ana', 0)
lista.show()
lista.push('Lucia', 1)
lista.show()
lista.push('Pedro', 0)
lista.show()
lista.push('Bia', 2)
lista.show()
lista.push('Rafaela', 4)
lista.show()
lista.push('Sonia', 2)
lista.show()
print('Tamanho da lista: %d\n' % lista.leng())
print(' ')
```

 # teste de remoção

```
lista.pop(0)
lista.show()
lista.pop(2)
lista.show()
lista.pop(3)
lista.show()
print('Tamanho da lista: %d\n' % lista.leng())
```

Referências:

Ascencio, A.F.G. & Araujo, G.S. Estruturas de Dados. Editora Pearson, 2011.
Laureano, Marcos. Estrutura de Dados com Algoritmos e C. Editora Brasport, 2008.
https://www.hashtagtreinamentos.com/funcoes-lambda-python