EDA1 - Somativa 2

Recursão

- É uma função que chama-se a si mesmo.
- Ex: **fatorial**.
- n! = n.(n-1).(n-2)...1 → **Expressão matematicamente**
    - Ex: **Código de fatorial.**

```
fat = n;
for (i = n-1; i>0; i--) {
    fat*=i;
}
```

- **Obs: chamamos essa solução de iterativa.**

Estrutura recursiva

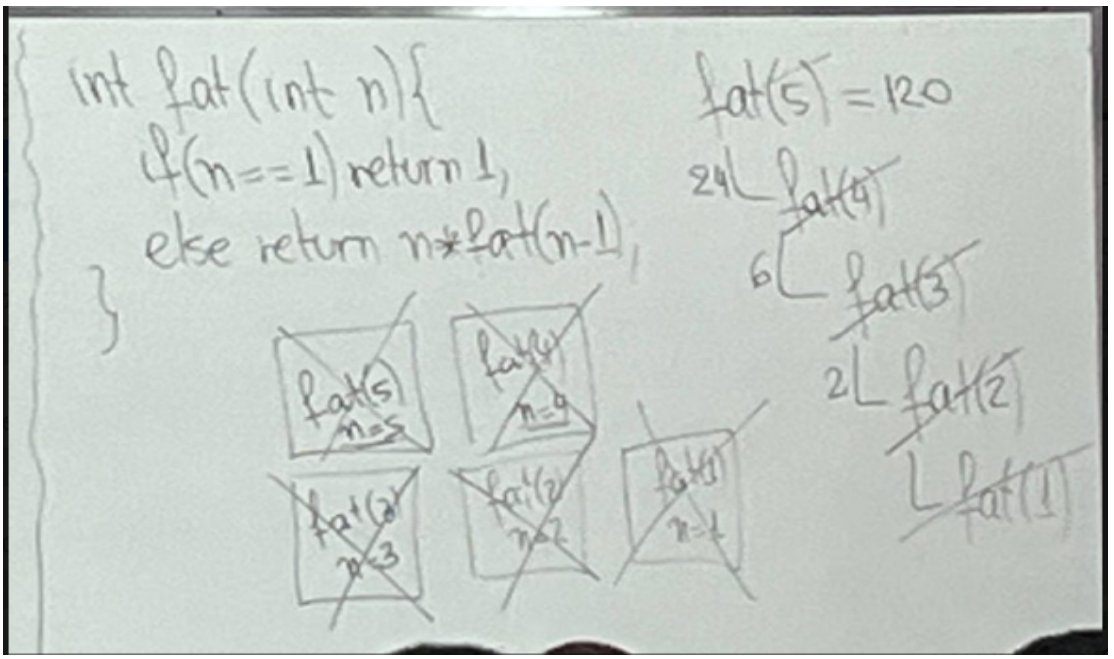
- Um problema que possui a seguinte característica: qualquer instância pode ser resolvida a partir da solução de uma instância de tamanho menor.
    - instância → Caso particular de um problema
    - Consegue resolver um problema de certo tamanho a partir de uma instância de um tamanho menor → Cadeia
        -

→ Laços são soluções iterativas => Fazem i vezes

➔ No fatorial:
n! = n.(n-1).(n-2).(n-3)...1 = n.(n-1)
➔ Conseguimos resolver multiplicando n pelo fatorial de n-1

```
int fat (int n) {
     if(n==1) return 1;
     else return n*fat(n-1); #Vou rodar até retornar 1 , sem
                              precisar de laços
}
```

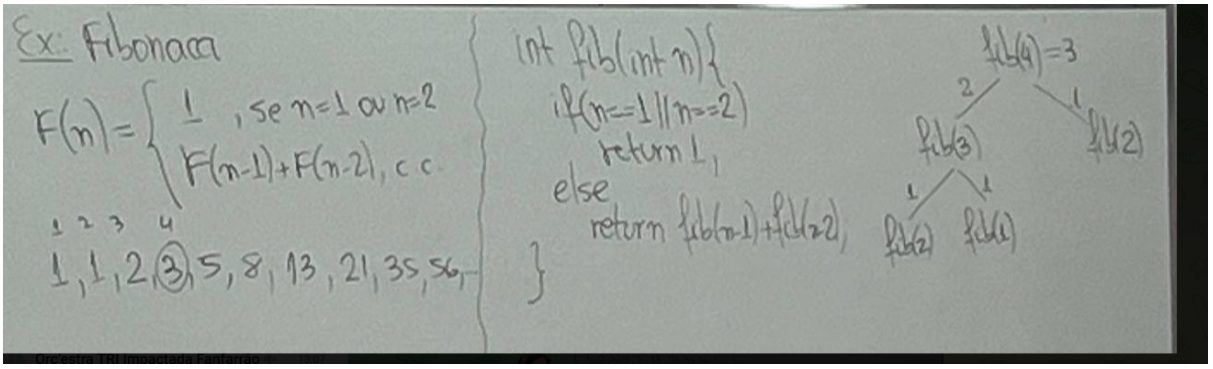
ÁRVORE DE RECURSÃO: fat(5) → fat(4) → fat(3) → fat(2) → fat(1) nessa última ele retorna 1 e termina de executar



"Você reduz até onde você pode simplificar e depois retorna até n fazendo combinação"

↳ Quando volta para fat2 retornando 1 e encerrando a caixa de memória de fat1, fat2 retorna 2 para fat3 que irá retornar 6 e encerrando a caixa de memória de fat2, fat3 retorna 6 para fat4 que irá retornar 24 e encerrando a caixa de memória de fat3, fat4 retorna 24 para fat5 que irá retornar 120 e encerrando a caixa de memória de fat4.

- Ex: **Fibonacci**



Aula 14/07

Exemplo 01: calcule a^b (com a e b pertencendo aos inteiros positivos e o zero)

$a^b = a.a^{(b-1)}$ → Estrutura recursiva

$a^0 = 1$

- Primeiro exercício da lista:

```
int pot (int a, int b) {
      if(b == 0) return 1;
      else return a*pot(a, b-1);
}
```

Explicação de como funciona essa recursão

2^5 = 32→2^4 = 16 → 2^3= 8→ 2^2 = 4 → 2^1= 2 → 2^0 = 1 {aqui teremos o 1 que era esperado e daqui voltaremos de trás para frente, ou seja até o 32 novamente}

Exemplo 02: Imprimir uma string.

→ Estrutura recursiva:

str[n] → str[0] e str[n-1]

Caso base → str[i] = '\0'.

```
void imp (char *str) {
      if( (*str)! = '\0') {
            printf("%c", *str);
            imp(str++);
      }
}
```

Exemplo 03: Imprimir uma String ao contrário.

→ Estrutura recursiva:

str[n] → str[0] e str[n-1]

Caso base → str[i] = '\0'.

```
void imp (char *str) {
      if( (*str)! = '\0') {
            imp(str++);
            printf("%c", *str);
      }
}
```

Quando trabalhamos com recursão, precisamos ficar atentos à posição das chamadas antes e após a recursão

- **Antes da Recursão**: A chamada ocorre antes de entramos na árvore de recursão, assim os comandos são executados à medida que descemos pela

árvore (**na ida**).

Tal fato ocorre no exemplo 02

- **Depois da Recursão:** A chamada ocorre depois da recursão, dessa forma os comandos são executados após a árvore de recursão chegar ao fim, ou seja, quando estivermos voltando para a função no topo da árvore (dizemos que os comandos são executados **na volta**).

Exemplo: **Régua inglesa.**

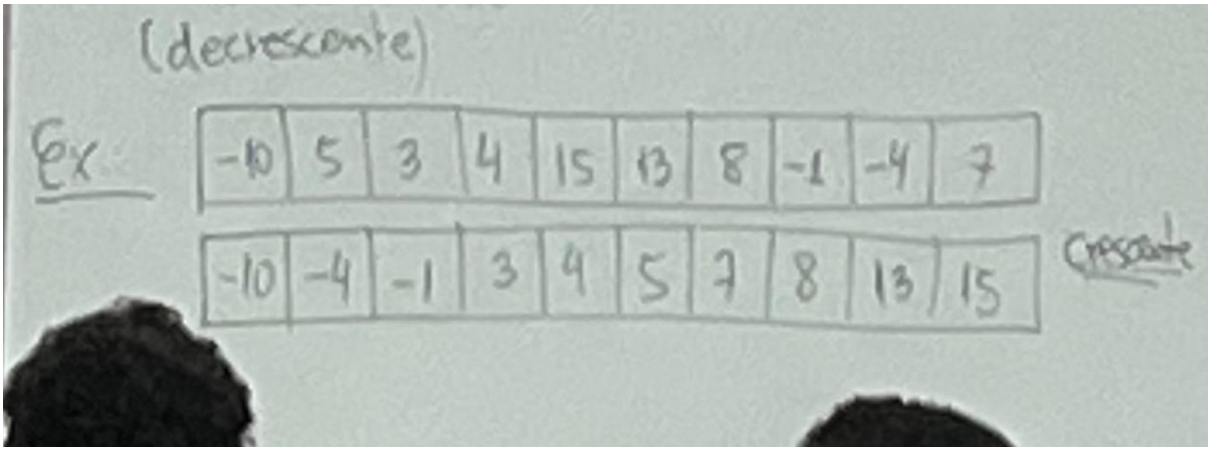
- Uma régua inglesa de grau n são 2^{n-1} linhas tais que:

---

aula 19/07

O Problema de ordenação:

- Dado um vetor numérico v[0…n-1], queremos ordenar seus elementos em ordem não-decrescente(**crescente**) ou não-crescente(**Decrescente**)



- Algoritmos de ordenação:
  - O(n^2) - seleção- inserção-bolha (bolha)
  - O($n.lgn$) quicksort // Mergesort (heapsort)
    - quicksort e mergesort são recursivos
  - O(n) - ordenam conjuntos particulares → São conjuntos que você consegue supor algo sobre eles.
    - Contagem
    - Distribuição

- Ordenação por seleção

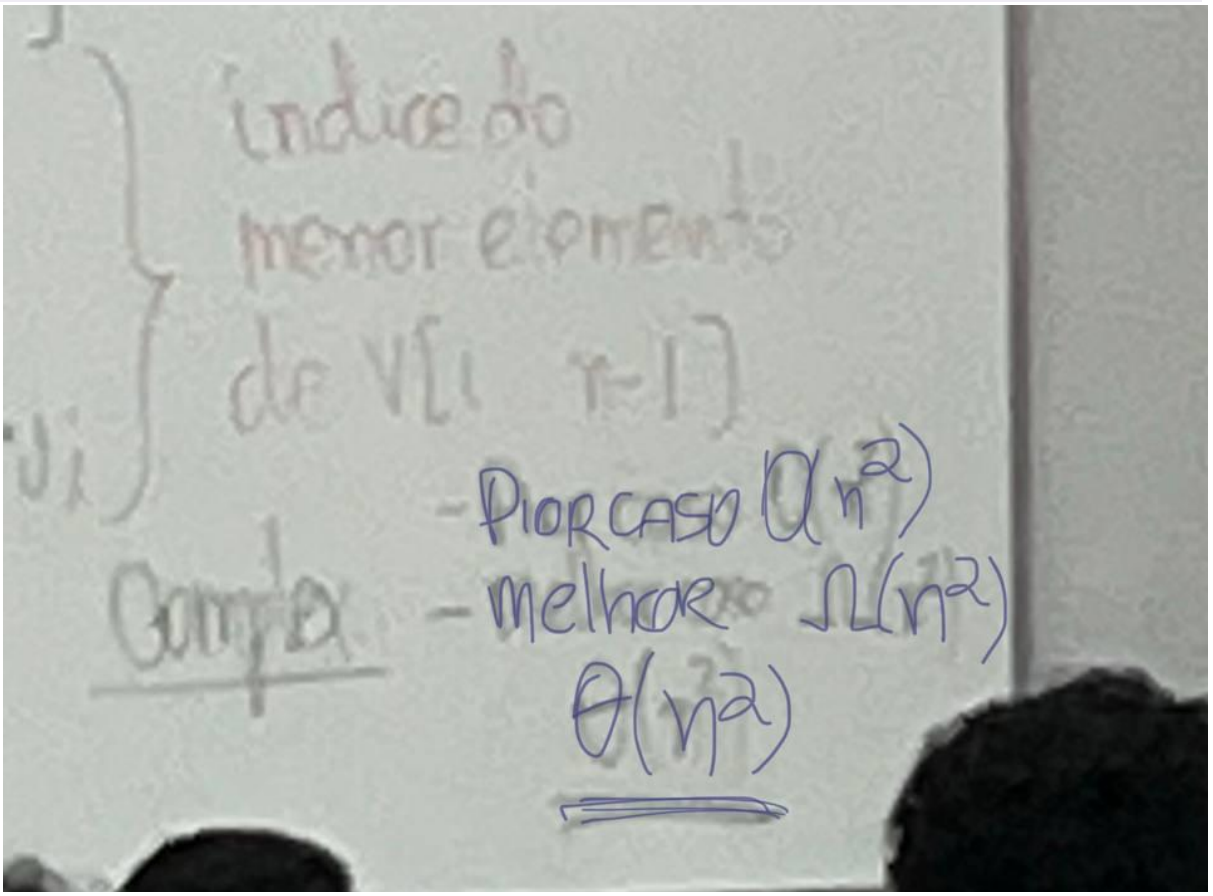
  Traz o menor valor para a primeira posição, troca de posição com os outros, se ele empurrar os outros elementos. → O(n)

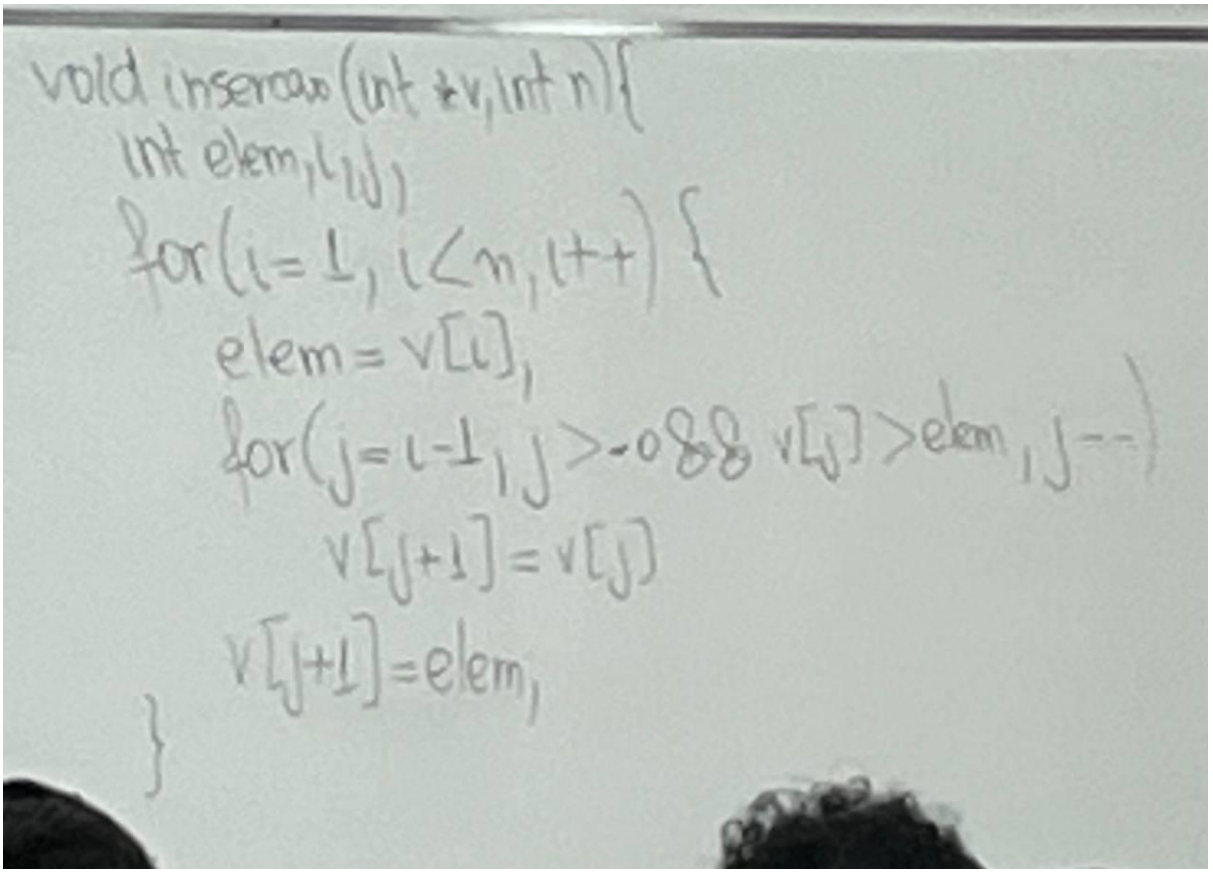
  1. Encontrar o menor valor do vetor v [i… n-1]
  2. Colocar o menor em v[i]

Obs: Trocar a posição do menor valor com o valor de v[i] ou deslocar o valor pelo

vetor até ele ser v[i] ?
Trocar a posição de v[i] é mais “barato” que deslocar a posição, pois o segundo é de ordem O(n)

```
void selecao (int *v, int n) {
      int i,j,menor;
      for ( i = 0; i < n; i++) {
            menor=i;
            for ( j = i+1; j < n; j++) {
                  if (v[j] < v[menor])
                        menor = j;
                        troca_vet(v,i,menor);
            }
      }
}
```





Estabilidade:

- Dizemos que um algoritmo de ordenação é **estável** se ele preserva a ordem relativa de elementos iguais.
- O Algoritmo de seleção é **instável**, pois não preserva a ordem. → **Boa questão de prova**

O Problema de busca:

- Dado um conjunto de elementos, dizer se x pertence a esse conjunto (e onde está,se for o caso).
- ex: V[10] = {-3,5,8,10,-15,0,7,1,4,6}
  X = 0 → Sim, a posição é 5

```
int busca(int *v, int n; int x) {
    for (int i = 0; i < n; i++) {
           if(v[i]==x) return i;
    }
}
```

- retorna a posição ou -1 se x não pertence ao vetor
- complexidade O(n)

O Problema de busca em vetor ordenado

- Dado um vetor v[0…n-1] ordenado e um elemento x qualquer,

queremos descobrir j tal que v[j-1]<x<=v[j].

- Ex: v[10] = {-15,-10,-2,-1,0,3,5,7,9,13}
  - x = 3 → retorna 5
  - x = -2 → retorna 2

```
int buscaBinaria (int x, int n, int v[]) {
   int e = -1, d = n;
   while (e < d-1) {
      int m = (e + d)/2;
      if (v[m] < x) e = m;
      else d = m;
   }
   return d;
}
```

complexidade O(lgn)

- Recursividade:
  - caso base: e == d - 1
  - Redução: buscar entre **e** e **m** ou **m** e **d**.

```
// Esta função recebe um vetor crescente
// v[0..n-1] e um inteiro x e devolve um
// índice j em 0..n tal que
// v[j-1] < x <= v[j].

int buscaBinaria2 (int x, int n, int v[]) {
   return bb (x, -1, n, v);
}
```

```
int bb (int *v, int x, int e, int d) {
      if (e == d-1) return d;
      else{
            int m = (e+d)/2;
            if (v[m] < x) return bb(v,x,m,d);
            else return bb(v,x,e,m);
      }
}
```

índices	0	1	2	3
elementos	10	-1	2	-3

Ao passar por ordenação:

índices	3	1	2	0
elementos	-3	-1	2	10

obs: chama-se vetor de permutação.

Listas encadeadas

- Vetores
  - ➔ Armazenam um conjunto de elementos.
  - ➔ Ocupam posições sequenciais na memória.
    - ➢ int v[100] → v é um ponteiro p/ primeira posição.
    - ➢ v[i] → v + i * sizeof
  - ➔ Tamanho pré fixado.
    - ➢ Tamanho fixo: int v[100] → É preferível que utilize fixado em tamanhos grandes pois aí o sistema já entenderia o tanto de memória que deveria utilizar.
    - ➢ Tamanho variável: alocação dinâmica de memória.

      VLA: variable length array
      int n;
      scanf("%d", &n);
      int v[n];

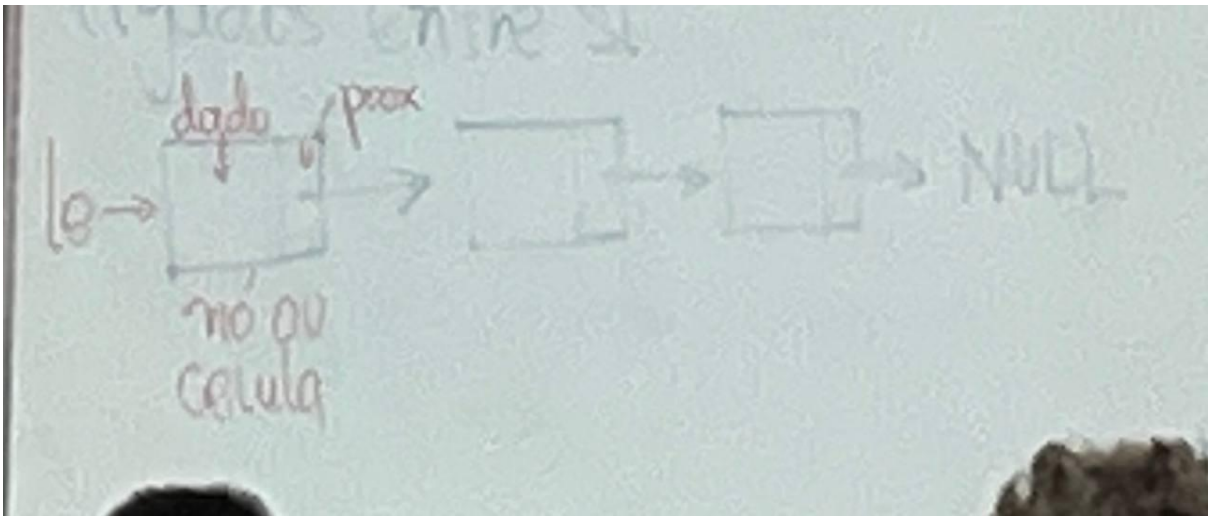
- E se quisermos inserir ou remover um elemento de um vetor, em qualquer posição? Qual o custo?
- **TRADE OFF**: Bom manipular, ruim para acessar os dados dentro do vetor.

```
int remove(int *v, int n, int i){
      int elem = v[i];
      for(int j = i, j < n - 1, j++){
            v[i] = v[i+1]
      }
      return elem;
}
```

- Concluindo:
    1. Vetores são estruturas de acesso rápido.
    2. Vetores não são boas estruturas para manipulação massiva de dados.

→ Trade-off: “Não existe lanche de graça” - Luciano Freitas

- Uma lista encadeada é um conjunto de elementos independentes interligados entre si.
- Liga esses elementos com apontadores/ponteiros, se chama “nó”/”célula”, dentro eu tenho um dado → pode ser um num, uma string um carácter.



- Representação dos nós:

```
typedef Struct no {
      int dado;
      Struct no *prox;
} no;
```

*Typedef
→ no lista;
→ no *lista (daremos preferência por usar ponteiros mesmo)

- Operações:
    1. Criação:

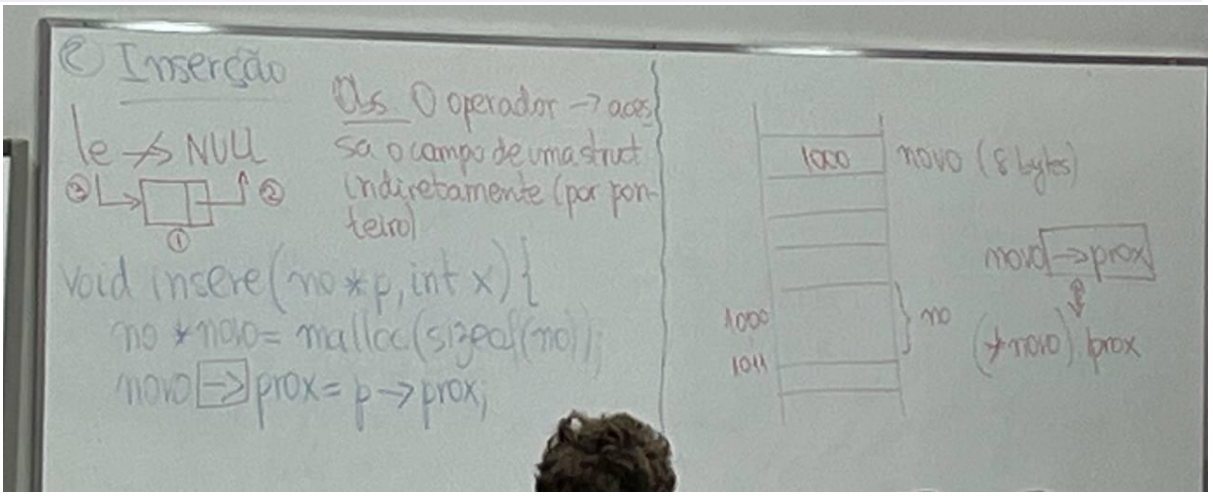
le → null (lista enc.vazia)

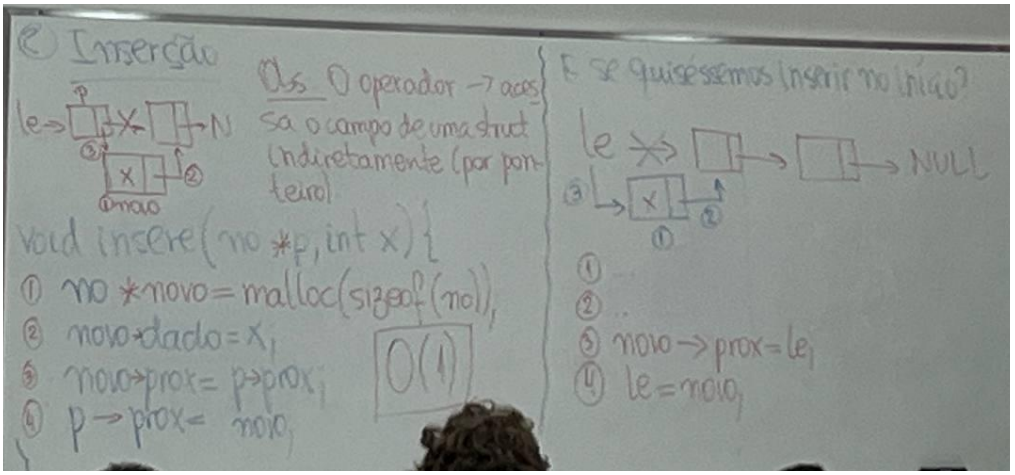
```
no *cria() {
      no *le = NULL;
      return le;
}
```

2. Inserção:
   le → NULL

```
void insere (no *p, int x) {
      r;
```

```
        novo → prox = p
}
```





Aula 04/08

Operações:

obs: lista encadeada com cabeça

1. **Criação:**

```
no *cria(){
      no *le = malloc (sizeof(no));
      le → prox = null;
      return le;
}
```

chamada: no*le = cria();

2. **Inserção:**

```
void insere (no *p, int x) { //insere depois de p
no *novo = malloc(sizeof(no));
novo -> dado = x;
novo -> prox = p -> prox;
p -> prox = novo;
}
```

Complexidade: O(n)

obs:nós usamos ponteiro pois não queremos que a variável deixe de existir após a função → Boa questão de provar

Obs:Só se cria novos nós quando usamos malloc, pois com alocação automática o nó é destruído no final da função

3. **Remoção:**

```
int remove (no*p) { //Remove o nó seguinte a p
      no *lixo = p -> prox;
      int dado = lixo -> dado;
      p -> prox = lixo -> prox;
      free(lixo);
      return dado;
}
```

Complexidade:O(n)

```
int remove (no *p, int *y) {
      no *lixo = p -> prox;
      if (lixo == null ) return e
*y = lixo -> dado;;
      p-> prox = lixo -> prox;
      free(lixo);
      return 1;
}
```

4.Destruição: //A gente chama para destruir toda a lista encadeada

```
 void destroi(no*le) {
      int dummy;
      while(remove(le,&dummy));
      free(le);
}
```

Biblioteca

- Implementação de um conjunto de função com finalidade comum.
- Em C:

- . h →header (protótipo pos/assinatura)
- .c →código, implementação