

PILHA

- * First in, last out → O primeiro a entrar é o último a sair.
- * A célula contém um (ou mais) item(s) e um ponteiro que referencia a próxima célula.
- * No caso da pilha, a "próxima" célula é, na verdade, aquela que foi inserida na estrutura ANTES da atual. Por isso, na pilha, cada célula, logo após nascer, tem seu PROX apontando para a célula que está no topo. É isso que garante o empilhamento.
- * Na pilha, a adição e a remoção de células só ocorrem no topo. O ponteiro TOPO sempre marca o elemento mais recentemente inserido. Assim, cada vez que adicionamos uma nova célula, TOPO passa a apontar para ela. Porém, antes de mudar o endereço apontado por TOPO, fazemos essa célula que está nascendo apontar para onde TOPO aponta atualmente (a última célula colocada antes dela e, portanto, sua sucessora ao iterar sobre os PROX da pilha).

```
class Pilha {
    private Celula topo;
    // construtor
    public Pilha() {
        topo = null;
    }
    public void inserir (int x){...}
    public int remover () {...}
    public void mostrar () {...}
}
```

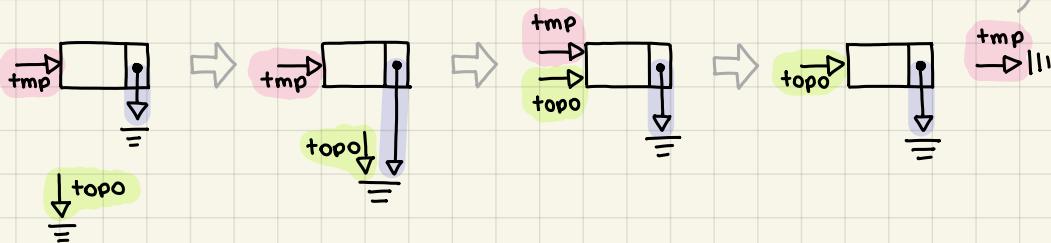
Quando fazemos Pilha pilha = new Pilha();
a pilha nasce assim:



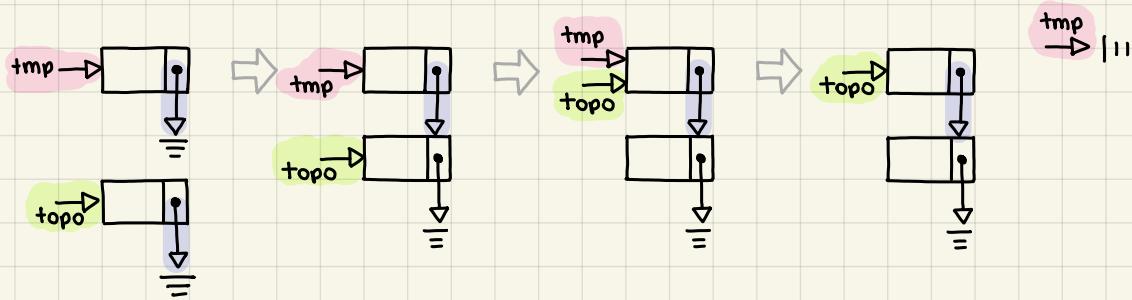
- * Chamamos de PROX o ponteiro dentro da célula que aponta para sua sucessora. Na lógica de inserção/remoção/visualizações da pilha, a sucessora apontada por PROX é na realidade a antecessora imediata de cada célula. Isto é, se o PROX da célula A aponta para a célula B, B foi inserida na pilha antes de A, porém A será mostrada e removida antes de B.

```
public void inserir (int x) {
    Celula tmp = new Celula (x);
    tmp.prox = topo;
    topo = tmp;
    tmp = null;
}
```

"soltarmos" o ponteiro tmp.



Inserindo a 2ª célula:

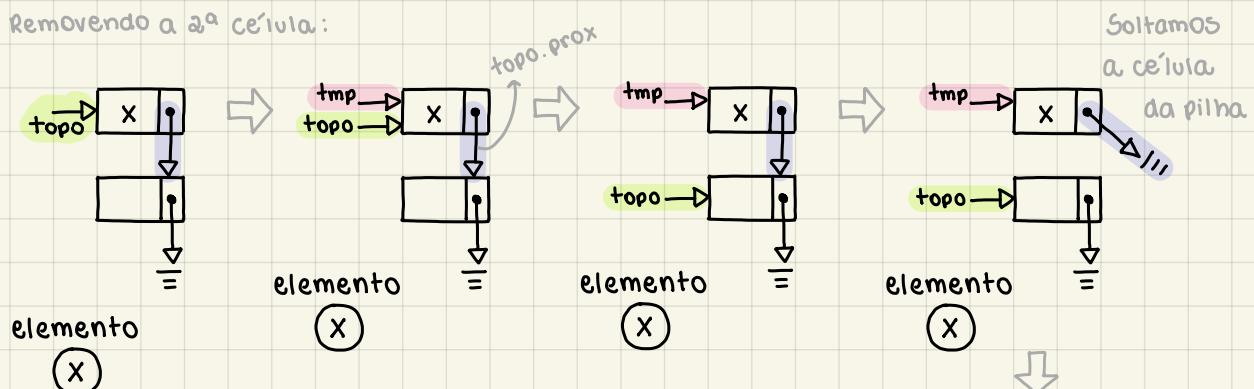


```

public int remover(){
    if (topo == null){
        System.out.println("ERRO");
        return -1;
    }
    int elemento = topo.elemento;
    Celula tmp = topo; // criamos um ponteiro que guarda um endereço para uma célula. Esse endereço será o mesmo endereço apontado por TOPO.
    topo = topo.prox;
    tmp.prox = null;
    tmp = null;
    return elemento;
}
  
```

→ Quando TOPO aponta para null, temos pilha vazia e, portanto, não há células para remover

Removendo a 2ª célula:

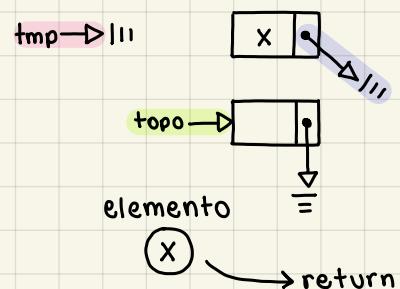


```

public void mostrar(){
    for (Celula i=topo; i!=null; i=i.prox){
        System.out.println(i.elemento);
    }
}
  
```

```

public int somar(){
    int sum=0;
    for (Celula i=topo; i!=null; i=i.prox){
        sum+=i.elemento;
    }
    return sum;
}
  
```



FILA

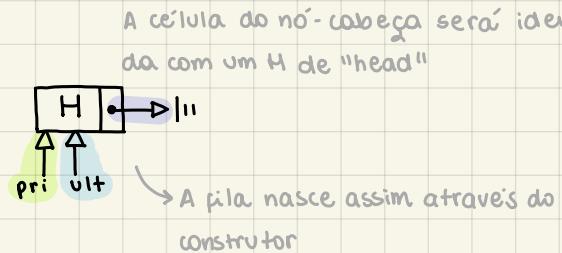
- * First in, first out → o primeiro a entrar é o primeiro a sair.
- * No construtor, criamos a fila já com uma célula, que chamamos de sentinelas ou nó-cabeça. Essa célula e seu conteúdo são absolutamente descartáveis, seu propósito é apenas ser sempre referenciada pelo ponteiro PRIMEIRO, o que facilita as operações no programa (especialmente quando temos que identificar e tratar a exceção de lista vazia).

```

class Fila() {
    // início da fila, frente
    private Celula primeiro;
    // o ponteiro ULTIMO se refere ao final da fila, ao elemento inserido mais recentemente
    private Celula ultimo;
    // para sabermos quantos elementos estão na fila. A sentinelas NÃO contará como célula
    // válida para contagem. A primeira célula, que consideraremos como ocupando a posição
    // de número zero da fila será a primeiro.prox
    private int tam;

    public Fila() {
        primeiro = new Celula(-1);
        ultimo = primeiro;
        tam = 0;
    }
}

```

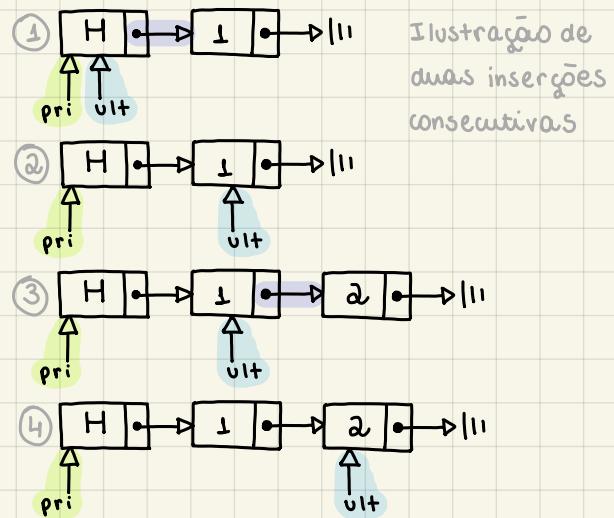


- * A inserção é sempre feita no final. Que nem em uma fila de pessoas, onde sempre se entra atrás/depois do último.
- * Ao contrário da pilha, em que cada célula nasce com seu PROX apontando para outra célula, na fila as células nascem sendo apontadas pelo PROX da anterior e com o seu próprio PROX permanecendo em null.
- * Assim, quando acessamos o conteúdo do .prox de uma célula na fila, somos direcionados para a célula que foi inserida DEPOIS dela e que, portanto, deve ser removida posteriormente.

```

public void inserir(int x){
    ultimo.prox = new Celula(x);
    ultimo = ultimo.prox;
    tam++;
}

```



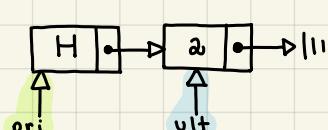
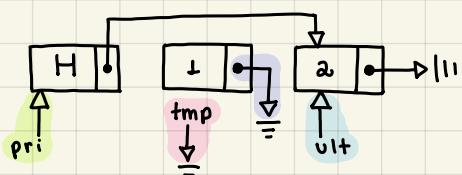
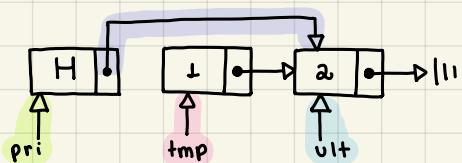
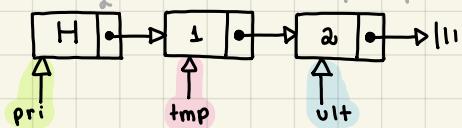
- * A remoção é sempre feita na ponta do PRIMEIRO. Primeiro a entrar, primeiro a sair. Removemos fisicamente a célula logo após a sentinelas.
- * Na inserção não temos esse problema, mas na remoção temos que tratar a exceção de fila vazia (apenas com a sentinelas).

```

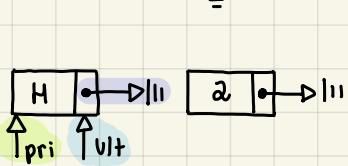
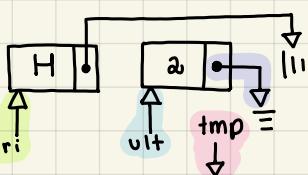
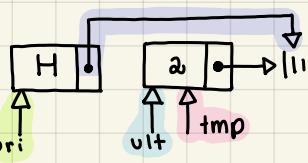
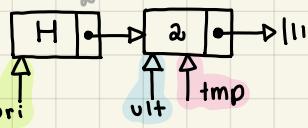
public int remover () {
    if (primeiro == ultimo) {
        System.out.println("ERRO");
        return -1;
    } else {
        Celula tmp = primeiro.prox;
        primeiro.prox = primeiro.prox.prox; //ou tmp.prox
        int elemento = tmp.elemento;
        tmp = tmp.prox = null;
        tam--;
        if (primeiro.prox == null) {
            ultimo = primeiro;
        }
        return elemento;
    }
}

```

Remoção de uma célula qualquer:



Remoção da última célula válida da fila:



Quando a lista fica vazia porque removemos todos os elementos que tínhamos colocado, temos `primeiro.prox = null`. ULTIMO não pode continuar apontando para a célula removida.

* Na hora de iterar sobre a fila com nó-cabeça, lembre-se de começar em `primeiro.prox`, a nossa primeira célula válida que consideraremos como ocupante da posição 0.

```

public void mostrar () {
    if (primeiro == ultimo) {
        System.out.println("ERRO. tentou mostrar lista vazia");
    } else {
        for (Celula i = primeiro.prox; i != null; i = i.prox) {
            System.out.println(i.elemento);
        }
    }
}

```

```

public int terceiroElemento () {
    int count = 0;
    Celula i = primeiro.prox;
    while (count < 3) {
        i = i.prox;
        count++;
    }
    return i.elemento;
}

```

LISTA SIMPLES

- * A lista simples nada mais é do que uma fila na qual podemos inserir e remover elementos em qualquer posição que tivermos. É exatamente a mesma estrutura da fila porém a manipulamos sem seguir a lógica de "first in, first out".
- * Também usamos célula sentinela na lista simples.

```
void inserirFim( int x ) { → FILA
|   Ultimo. prox = new Celula(x);
|   Ultimo = Ultimo. prox;
|   tam++;
}
```

```
int removerInicio( ) { → FILA
|   if (primeiro == ultimo){
|     System.out.println("ERRO. lista vazia");
|   } else {
|     Celula tmp = primeiro. prox;
|     primeiro. prox = tmp. prox;
|     int num = tmp. elemento;
|     tmp. prox = null;
|     if (primeiro. prox == null) {
|       ultimo = primeiro;
|     }
|     tam--;
|     return num;
|   }
}
```

```
int removerFim() { → PILHA
|   if (tam == 0 || tam == 1) {
|     return removerInicio();
|   } else {
|     int elemento = ultimo. elemento;
|     Celula i = primeiro. prox;
|     while (i. prox. prox != null) {
|       i = i. prox;
|     }
|     ultimo = i;
|     ultimo. prox = null;
|     i = null;
|     tam--;
|     return elemento;
|   }
}
```

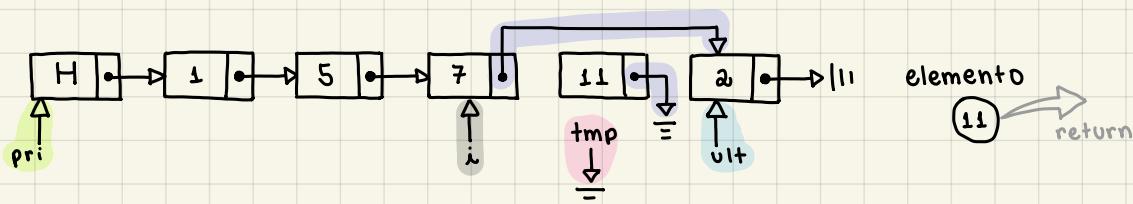
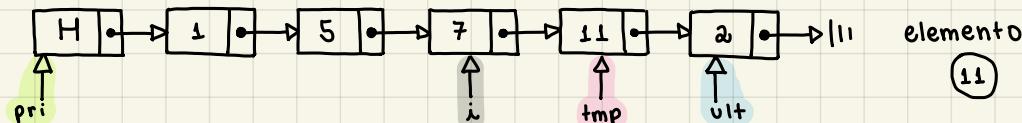
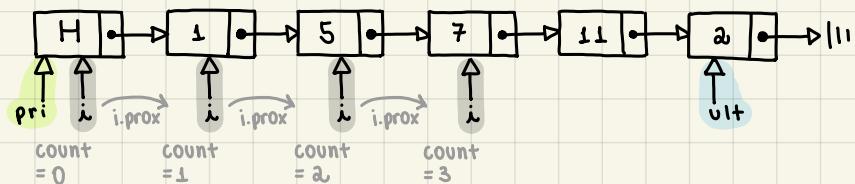
```
void inserirInicio( ) { → PILHA
```

```
|   Celula tmp = new Celula(x);
|   tmp. prox = primeiro. prox;
|   primeiro. prox = tmp;
|   if (primeiro == ultimo) {
|     ultimo = tmp;
|   }
|   tmp = null;
|   tam++;
| }
```

```
int remover( int pos ) {
```

```
|   if (pos < 0 || pos >= tam) {
|     System.out.println("ERRO. Posição inválida");
|     return -1;
|   } else if (pos == 0) {
|     return removerInicio();
|   } else if (pos == tam - 1) {
|     return removerFim();
|   } else {
|     int count = 0; → queremos parar uma célula antes
|     Celula i = primeiro; → daquela que será removida, por
|     while (count < pos) { → isso é como se iniciássemos a
|       i = i. prox; → contagem no -1 ao declarar i como
|       count++; → primeiro ao invés de primeiro. prox
|     }
|     Celula tmp = i. prox;
|     int elemento = tmp. elemento;
|     i. prox = tmp. prox;
|     tmp. prox = null;
|     tam--;
|     return elemento;
|   }
}
```

Remoção da célula na posição 3



```
void inserir (int x, int pos){
    if (pos < 0 || pos >= tam) {
        System.out.println ("ERRO. Posição inválida.");
    } else if (pos == 0) {
        return inserirInício (x);
    } else if (pos == tam) {
        return inserirFim (x);
    } else {
        int count = 0;
        Celula i = primeiro;
        while (count < pos) {
            i = i.prox;
            count++;
        }
        Celula tmp = new Celula (x);
        tmp.prox = i.prox;
        i.prox = tmp;
        tmp = null;
        tam++;
    }
}
```

checklist conteúdo para prova 2:

- pilha
- fila
- lista simples
- lista dupla
- matriz flexível → só aprender a percorrer
- árvore binária
 - inserção
 - percurso e caminhamento
 - remoção
 - círculo em C
- exercícios extra
- prova antiga
- tabela de composição dos algoritmos de ordenação
- quicksort
- revisar insertion sort

LISTA DUPLA

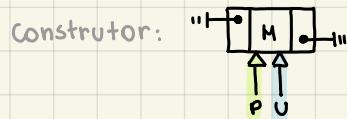
```
class Celula {
    public int elemento;
    public Celula prox;
    public Celula ant;
    public Celula (int x) {
        this. elemento = x;
        this. prox = null;
        this. ant = null;
    }
}
```

```
class ListaDupla {
    private Celula primeiro;
    private Celula ultimo;
    private int tam;

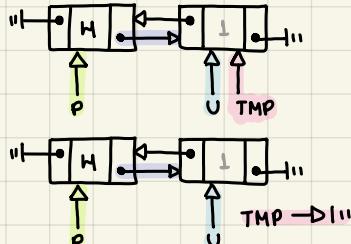
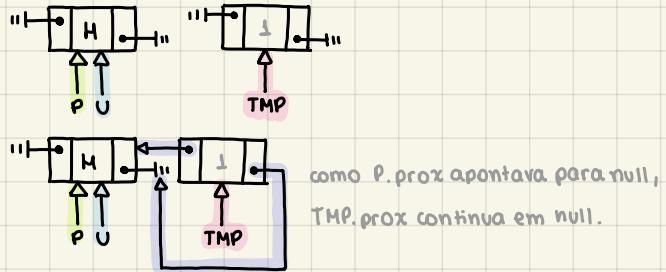
    public ListaDupla() {
        this.primeiro = new Celula(-1);
        this.ultimo = primeiro;
        this.tam = 0;
    }

    public void inserirInicio (int x) {
        Celula tmp = new Celula(x);
        tmp.ant = primeiro;
        tmp.prox = primeiro.prox;
        primeiro.prox = tmp;
        if(primeiro == ultimo) {
            ultimo = tmp;
        } else {
            tmp.prox.ant = tmp;
        }
        tmp = null;
        tam++;
    }
}
```

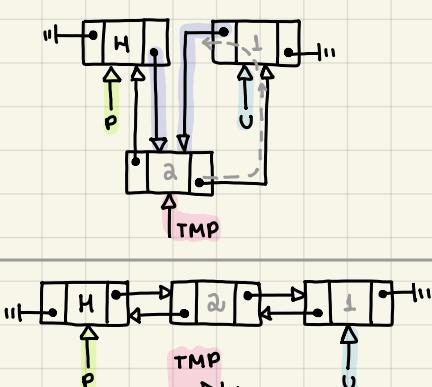
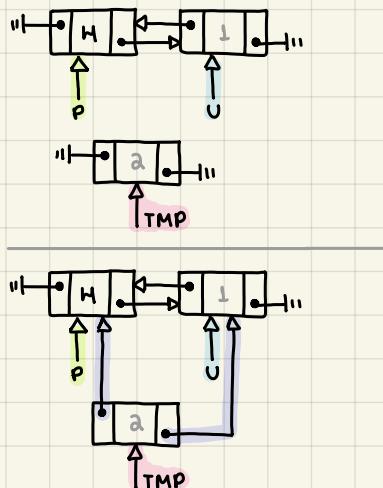
```
public void inserirFim (int x) {
    ultimo.prox = new Celula(x);
    ultimo.prox.ant = ultimo;
    Ultimo = ultimo.prox;
    tam++;
}
```



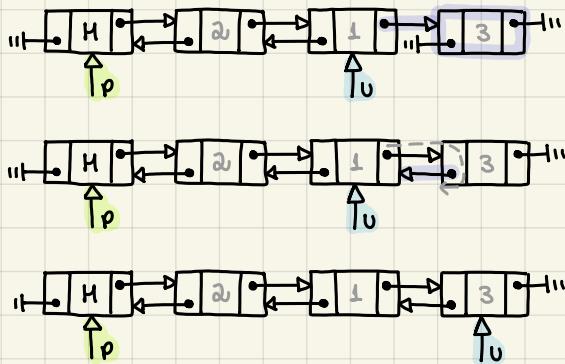
Inserir inicio em lista vazia:



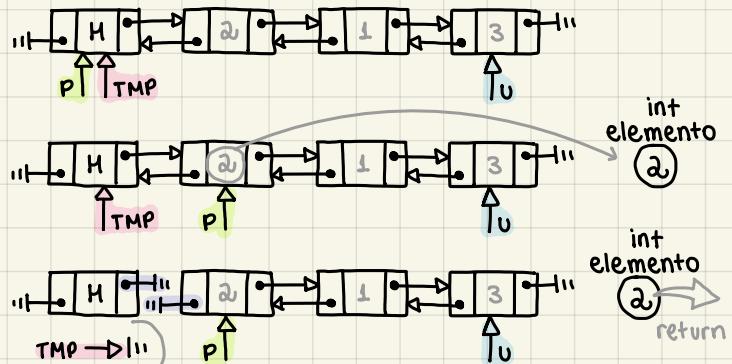
Inserir inicio em lista ja preenchida:



Inserir fim:



Remover inicio em lista com mais de 1 elemento:



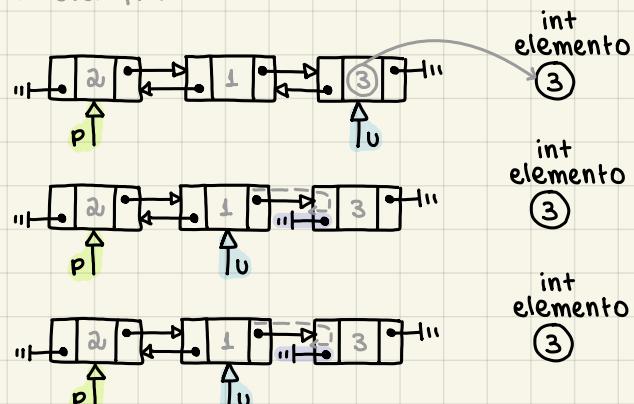
Obs.: em C, teríamos que fazer FREE antes de soltar o ponteiro TMP para efetivamente apagar a célula que retiramos da lista.

```
public int removerInicio() {
    if (primeiro == ultimo) {
        System.out.println("ERRO. lista vazia");
        return -1;
    } else {
        Celula tmp = primeiro;
        primeiro = primeiro.prox;
        int elemento = primeiro.elemento;
        tmp.prox = primeiro.ant = null;
        tmp = null; tam--;
        return elemento;
    }
}
```

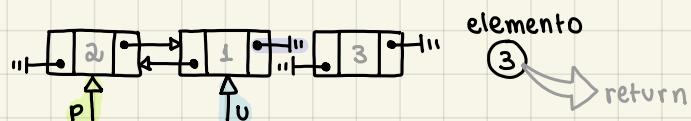
Esse método faz com que a célula sentinelा mude a cauda removida. No entanto, como seu valor já foi retornado e logicamente removido da estrutura, o nó cabeca continua tendo conteúdo descartável e irrelevante, mesmo que ele seja uma "antiga célula válida".

```
public int removerFim () {
    if (tam == 0 || tam == 1) {
        return removerInicio();
    } else {
        int elemento = ultimo.elemento;
        ultimo = ultimo.ant;
        ultimo.prox.ant = null;
        ultimo.prox = null;
        tam--;
        return elemento;
    }
}
```

Remover fim:



Em C, precisaríamos de dar FREE em ultimo.prox antes de defini-lo como NULL para apagar fisicamente a célula removida.

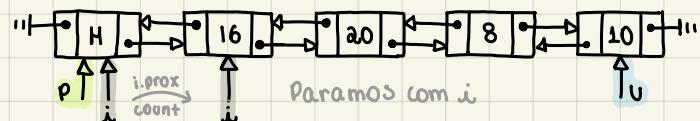
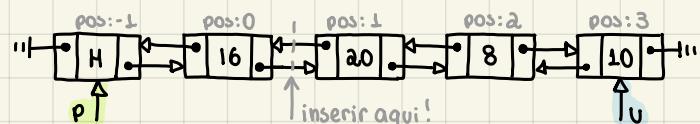


```

public void inserir(int x, int pos) {
    if (pos < 0 || pos > tam) {
        System.out.println("ERRO. Posição inválida");
    } else if (pos == 0) {
        inserirInício(x);
    } else if (pos == tam) {
        inserirFim(x);
    } else {
        int count = 0;
        Celula i = primeiro;
        while (count < pos) {
            i = i.prox;
            count++;
        }
        Celula tmp = new Celula(x);
        tmp.ant = i;
        tmp.prox = i.prox;
        i.prox = tmp;
        tmp.prox.ant = tmp;
        tmp = null;
        tam++;
    }
}

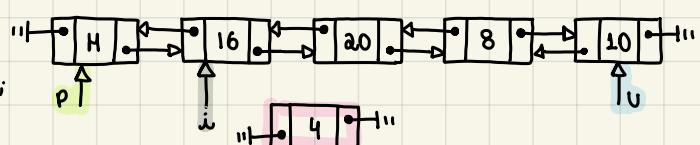
```

Inserir o número 4 na posição 1:

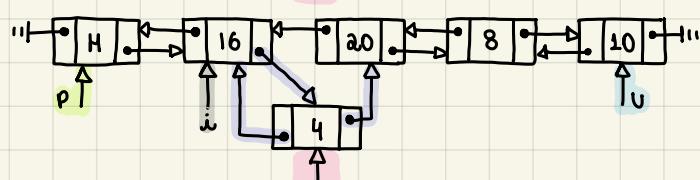


Paramos com i.

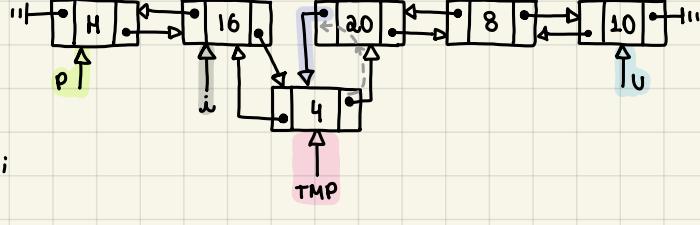
Apontando para a célula na posição 0 da fila.



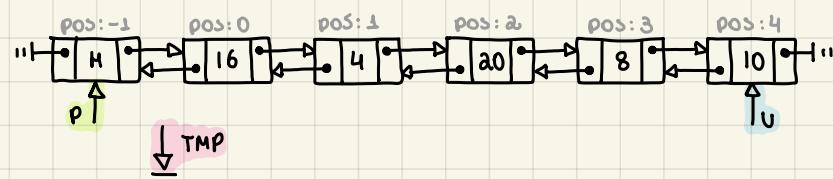
TMP



TMP



TMP



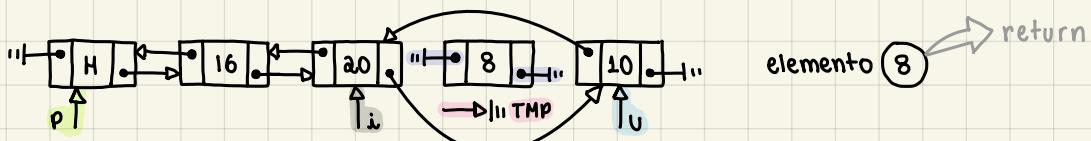
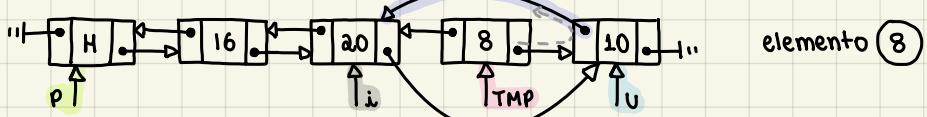
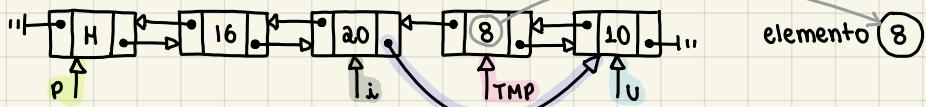
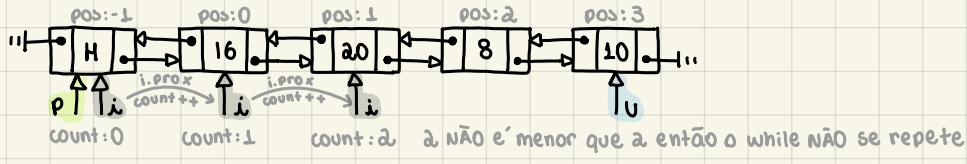
=
↓ TMP

```

public int remover(int pos) {
    if (pos < 0 || pos > tam) {
        System.out.println("ERRO");
    } else if (pos == 0) {
        return removerInício();
    } else if (pos == tam - 1) {
        return removerFim();
    } else {
        int count = 0;
        Celula i = primeiro;
        while (count < pos) {
            i = i.prox;
            count++;
        }
        Celula tmp = i.prox;
        int elemento = tmp.elemento;
        i.prox = tmp.prox;
        tmp.prox.ant = i;
        tmp = tmp.prox = tmp.ant = null;
        tam--;
        return elemento;
    }
}

```

Remover o elemento na posição 2:

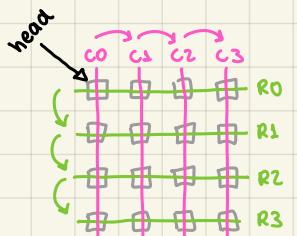


MATRIZ FLEXÍVEL

Editar elemento de uma célula já existente: int e, preencher a matriz.

```

void editar (Matrix* matrix, int row, int col, int x){ coordenadas da célula onde se deseja inserir
  if (matrix && matrix->head){                                     → assumindo que a matriz é quadrada
    if (row >= 0 && col >= 0 && row < matrix.size && col < matrix.size){
      Cell* ptr = matrix.head;
      // navegar de linha em linha (movimento vertical do ponteiro)
      for (int i = 0; i < row; i++){
        ptr = ptr->down;
      }
      // navegar de coluna em coluna (movimento horizontal do ponteiro)
      for (int j = 0; j < col; j++){
        ptr = ptr->right;
      }
      // inserir dados
      ptr->data = x;
      ptr = null;
    } else {
      cout << "Posição inválida";
    }
  } else {
    cout << "ERRO";
  }
}
  
```



- O ponteiro da matriz e a cabeça da matriz não podem ser NULL
- Os índices devem estar dentro dos limites da matriz
- A partir da cabeça, podemos movermos o ponteiro auxiliar ptr na direção da direita e para baixo.
- ptr é local à função e será destruído no final da função, não é realmente necessário o NULL

Percorrer e printar cada elemento: no caso, TODOS os elementos

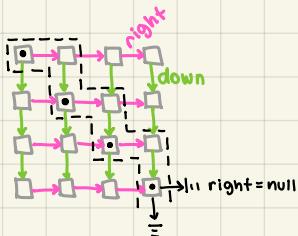
```
void print ( Matrix* m){  
    if (m && m->head){  
        // Definir apontadores auxiliares  
        Cell* row = m->head; // I indica o inicio da linha atual → vai de linha em linha  
        Cell* ptr = row; // Ponteiro que será usado para percorrer as células dentro de uma linha  
  
        // O loop externo percorre as linhas (movimento vertical)  
        for (int y=0; y<m.size; y++){  
            // O loop interno percorre as colunas dentro de cada linha  
            for (int x=0; x<m.size; x++){  
                printf ("[%d] ", ptr->data);  
                ptr = ptr->right; // Avançamos o ponteiro para a direita  
            }  
  
            // pular linha para formatar expressão  
            printf ("\n");  
            // ir para a linha de baixo  
            row = row->down;  
            // posicionar o ponteiro que vai de célula em célula horizontalmente no  
            // inicio dessa próxima  
            // linha.  
            ptr = row;  
        }  
        printf ("\n");  
    } else {  
        println ("ERRO");  
    }  
}
```

Mostrar diagonal principal:

Temos que passar o head da matriz como → primeiro parâmetro ($m \rightarrow head$)

```
void mainDiagonal (Cell* ptr){
```

```
// Printar elemento da célula atual  
printf ("[%d]\n", ptr->data);  
// Enquente todos os elementos, chamar recursivamente a função  
if (ptr->right && ptr->right->down){  
    mainDiagonal (ptr->right->down);  
}  
}
```



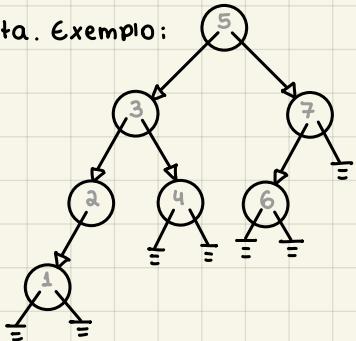
- Para percorrer a diagonal secundária, fazemos um encapsulamento que posiciona ptr na célula mais à direita da primeira linha e aí passamos ele como parâmetro para a função principal que de fato printa.

```
void secondaryDiagonal (Matrix* m){
```

```
    Cell* ptr = null;  
    for (ptr = m->head; ptr->right != NULL; ptr = ptr->right);  
    secondaryDiagonalPrint (ptr);  
}
```

AÁRVORE BINÁRIA

- * Custo de inserção, remoção e pesquisa pode ser $\Theta(\lg(n))$ comparações.
- * Formada por um conjunto finito de nós (vértices) conectados por arestas.
- * Um nó com filhos é chamado de nó interno.
- * Um nó sem filhos é chamado de folha.
- * O nó no nível 0 é a raiz.
- * Altura (h): maior distância entre um nó e a raiz.
- * Árvore binária é aquela em que cada nó possui no máximo dois filhos.
- * Na árvore binária de pesquisa, cada nó é maior que todos seus vizinhos à esquerda e menor que todos à direita. Exemplo:



* Árvore balanceada: árvore em que para TODOS os nós a diferença entre a altura de suas árvores da esquerda e da direita sempre será 0 ou ± 1 .

MENORES À ESQUERDA E MAIORES À DIREITA!

- * Características da árvore binária completa:

- Cada nó é uma folha OU possui exatamente dois filhos.
- Todos os nós folhas possuem uma altura h .
- O número de nós internos é $2^h - 1$
- O número de folhas é 2^h
- O número total de nós é $(2^h - 1) + (2^h) = 2^{h+1} - 1$

Isso significa que o número total de nós NÃO pode ser uma potência de 2, pois sobraria um nó a mais

- * Classe de nó de árvore binária de pesquisa em Java:

```

class No {
    int elemento;
    No esq;
    No dir;
    No (int num){
        this(num,null,null);
    }
    No (int num, No e ,No d){
        this.elemento = num;
        this.esq=e;
        this.dir=d;
    }
}
    
```

* Classe de árvore binária de pesquisa em Java:

```

class ArvoreBinaria {
    No raiz;
    ArvoreBinaria () {this.raiz=null;}
    void inserir (int x) {...}
    void inserirPai (int x) {...}
    boolean pesquisar (int x){...}
    void caminharCentral () {...}
    void caminharPre (){...}
    void caminharPos (){...}
    void remover (int x) {...}
}
    
```

R
↓
=

Funcionamento básico da INSERÇÃO:

- ① IF a raiz estiver vazia, insere-se o elemento nela.
- ② ELSE IF o novo elemento for menor que o da raiz, chama-se recursivamente a inserção para a subárvore da esquerda.
- ③ ELSE IF o novo elemento for maior que o da raiz, chama-se recursivamente a inserção para a subárvore da direita.
- ④ ELSE IF o novo elemento for igual ao da raiz, não inserir um elemento repetido.

Inserção em Java com retorno de referência:

```
public void inserirVoid (int x) {  
    raiz = inserirRetorna (x, raiz);  
}  
  
public No inserirRetorna (int x, No i){  
    if (i==null) {  
        //caso base: encontramos a posição para inserir o novo nó  
        i = new No (x);  
    } else if (x < i.elemento) {  
        //Se o valor a inserir é menor que o do nó atual, vamos para a subárvore da esquerda  
        i.esq = inserirRetorna (x, i.esq);  
    } else if (x > i.elemento) {  
        i.dir = inserirRetorna (x, i.dir);  
    } else  
        //O valor já existe na árvore  
        System.out.println ("ERRO em inserirRetorna");  
    }  
  
    //Retornamos o nó atual (possivelmente com seus ponteiros atualizados)  
    return i;  
}
```

Inserção em Java com passagem de pai:

```
public void inserirPai01 (int x) {  
    if (raiz == null){  
        raiz = new No (x);  
    } else if (x < raiz.elemento) {  
        // passa o filho esquerdo da raiz como filho e a raiz como pai  
        inserirPai02 (x, raiz.esq, raiz);  
    } else if (x > raiz.elemento) {  
        inserirPai02 (x, raiz.dir, raiz);  
    } else{  
        //se x = raiz.elemento, o número já existe na árvore  
        System.out.println ("ERRO em inserirPai01");  
    }  
}
```

```

public void inserirPai02(int x, No filho, No pai) {
    if (filho == null) {
        //Caso base: encontramos a posição correta para inserir o novo nó
        if (x < pai.elemento) {
            pai.esq = new No(x);
        } else {
            pai.dir = new No(x);
        }
    } else if (x < filho.elemento) { → Se x é menor que o elemento do nó atual (filho), chama recursivamente para o filho esquerdo. O nó atual se torna o novo pai.
        inserirPai02(x, filho.esq, filho);
    } else if (x > filho.elemento) {
        inserirPai02(x, filho.dir, filho);
    } else {
        System.out.println("ERRO em inserirPai02");
    }
}

```

Análise de complexidade da INSERÇÃO:

- Melhor caso: $\Theta(1)$ comparações
- Pior caso: $\Theta(n)$ comparações. Acontece quando inserimos os elementos na ordem crescente ou decrescente.
- Caso médio: $\Theta(\log(n))$ comparações. Acontece quando inserimos um elemento na folha de uma árvore balanceada. → altura da árvore balanceada

Funcionamento básico da PESQUISA:

- ① IF a raiz estiver vazia, retornar uma pesquisa negativa.
- ② ELSE IF o elemento procurado for igual ao da raiz, retornar uma pesquisa positiva.
- ③ ELSE IF o elemento procurado for menor que o da raiz, chamar o método de pesquisa para a subárvore esquerda.
- ④ ELSE (elemento procurado é maior que o da raiz) chamar o método de pesquisa para a subárvore da direita.

```

boolean pesquisarEncapsulado(int x){
    return pesquisar(x,raiz);
}

```

- Melhor caso: $\Theta(1)$ comparações
- Pior caso: $\Theta(n)$ comparações. Acontece quando inserimos os elementos em ordem e o elemento desejado está na folha.
- Caso médio: $\Theta(\log(n))$ comparações. Acontece quando a árvore está balanceada e procuramos um elemento localizado em uma das folhas.

```

boolean pesquisar(int x, No i){
    boolean resp;
    if (i == null) {
        resp = false;
    } else if (x == i.elemento) {
        resp = true;
    } else if (x < i.elemento) {
        resp = pesquisar(x,i.esq);
    } else {
        resp = pesquisar(x,i.dir);
    }
    return resp;
}

```

! 8 exercícios nos slides de pesquisa e cominhamento

CAMINHAMENTO:

→ Imprime os elementos em ordem crescente

CENTRAL OU EM ORDEM: Esquerda, nó, direita

```
void caminharCentral (No i){
    if (i != null) {
        caminharCentral (i.esq);
        System.out.print (i.elemento + " ");
        caminharCentral (i.dir);
    }
}
```

Imprime o elemento do nó atual. A impressão após a visita ao filho esquerdo e antes da visita ao filho direito.

O caminhar central:

- visita o filho esquerdo
- visita o nó atual
- visita o filho direito

Se i for null, a função não faz nada e retorna.

Chama a função recursivamente para o filho esquerdo do nó atual. Isso significa que iremos percorrer toda a subárvore esquerda antes de processar o nó atual.

Após processar o nó atual, iremos percorrer a subárvore da direita.

PÓS-FIXADO OU PÓS-ORDEM: Esquerda, direita, nó

```
void caminharPos (No i){
    if (i != null) {
        caminharPos (i.esq);
        caminharPos (i.dir);
        System.out.print (i.elemento + " ");
    }
}
```

- Das folhas para a raiz, sobe.
- Visita cada nó da árvore APÓS ter visitado seus filhos.
- Ideal para deletar uma árvore. Ao liberar (apagar) um nó, temos certeza de que seus filhos já foram liberados, evitando "referências pendentes" (Pontos na memória que apontam para dados que já foram excluídos).

PRÉ-FIXADO OU PRÉ-ORDEM: Nó, esquerda, direita

```
void caminharPre (No i){
    if (i != null) {
        System.out.print (i.elemento + " ");
        caminharPre (i.esq);
        caminharPre (i.dir);
    }
}
```

- Frequentemente usado para salvar a estrutura de uma árvore em um arquivo ou transmitir sua estrutura para recriá-la em outro lugar. Como processamos cada nó antes dos filhos, permite salvar a estrutura hierárquica de maneira natural.

ACHAR O MAIOR ELEMENTO:

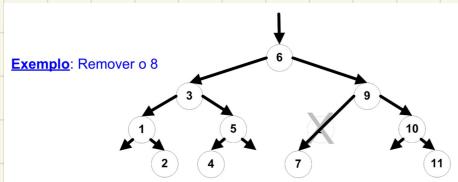
```
int getMaior(){
    int resp = -1;
    if (raiz != null) {
        No i;
        for (i = raiz; i.dir != null; i = i.dir) {
            resp = i.elemento;
        }
    }
    return resp;
}
```

ACHAR O MENOR ELEMENTO:

```
int getMenor(){
    int resp = -1;
    if (raiz != null) {
        No i;
        for (i = raiz; i.esq != null; i = i.esq) {
            resp = i.elemento;
        }
    }
    return resp;
}
```

Fucionamento básico da REMOÇÃO:

- ① IF o elemento estiver em uma folha, removê-la.
- ② ELSE IF o elemento estiver em um nó intermedio com um único filho, remover o nó e fazer com que seu pai aponte para seu filho.



- ③ ELSE IF o elemento estiver em um nó intermedio com 2 filhos, o elemento a ser removido deve ser substituído pelo maior nó da subárvore à esquerda OR pelo menor nó da subárvore à direita. Um dos descendentes do nó retirado (não precisa ser filho imediato dele) toma o seu lugar.

```

void removerVoid (int x){
    raiz = remover (x,raiz);
}
    
```

```
NO removerRetorna (int x, NO i){
```

```

if (i == null){
    System.out.println ("ERRO. O elemento não existe na árvore");
} else if (x < i.elemento){
    i.esq = removerRetorna (x,i.esq);
} else if (x > i.elemento){
    i.dir = removerRetorna (x,i.dir);
} else if (i.dir == null){
    i = i.esq;
} else if (i.esq == null){
    i = i.dir;
} else {
    i.esq = maioresg (i,i.esq);
}
return i;
}
```

```

No maioresg (NO i, NO j){
if (j.dir == null){
    i.elemento = j.elemento;
    j = j.esq;
} else {
    j.dir = maioresg (i,i.dir);
}
return j;
}
    
```

- MELHOR CASO: $\Theta(1)$ comparações . Quando o elemento a ser removido está na raiz.
- PIOR CASO: $\Theta(n)$ comparações . Quando inserimos os elementos em ordem e o elemento desejado está na folha.
- CASO MÉDIO: $\Theta(\lg(n))$ comparações . Quando a árvore está balanceada e desejamos remover um elemento localizado em uma das folhas.

QUICKSORT

- ① Escolha de um Pivô: Primeiramente, o Quicksort escolhe um elemento do array para ser o "pivô". Esse pivô é o ponto de referência que vamos usar para dividir os elementos em dois grupos: elementos menores que o pivô à esquerda e elementos maiores à direita.
- ② Partitionamento: O algoritmo reorganiza o array fazendo trocas e efetivamente colocando os elementos menores que o pivô de um lado e os maiores do outro. Ao final de uma iteração da função, que é recursiva, o pivô estará na sua posição correta na ordem final do array.
- Percorra o array a partir da esquerda enquanto $array[i] < \text{pivo}$
 - Percorra o array a partir da direita enquanto $array[j] > \text{pivo}$
 - Se $i \leq j$, então troque $array[i]$ com $array[j]$
 - Continue o processo enquanto $i < j$
- ③ Recursão: o Quicksort é aplicado recursivamente nas sub-listas resultantes (à esquerda e à direita do pivô). Esse processo continua até que os arrays sejam pequenos o suficiente para que não precisem mais ser divididos, ou seja, quando têm zero ou um elemento.

```
void quicksort (int esq, int dir) { → na 1ª chamada, teremos quicksort (0, n-1) sendo n o número de elementos do array
|   int i = esq;
|   int j = dir;
|   pivo = array [(esq+dir)/2];
|   while (i <= j) {
|     while (array [i] < pivo) {
|       i++;
|     }
|     while (array [j] > pivo) {
|       j--;
|     }
|     if (i <= j) {
|       swap (i, j);
|       i++; j--;
|     }
|   }
|   if (esq < j) {
|     quicksort (esq, j);
|   }
|   if (i < dir) {
|     quicksort (i, dir);
|   }
}
```

*escolher o meio
é num caso*



COMPARAÇÕES:

- Bom para a ordenação de grandes conjuntos de dados, rápido.
- "In-place": não precisa de muita memória extra. Usa apenas uma pequena pilha como memória auxiliar
- Instável: pode mudar a ordem relativa de elementos iguais

MOVIMENTAÇÕES:

- Melhor caso: sistematicamente, cada partição divide o array em duas partes iguais. O pivô perfeito é a mediana. $\Theta(n \times \lg(n)) \rightarrow n$ comparações realizadas em cada um dos $\lg(n)$ passos. Em listas grandes medianas de 3 elementos aleatórios é uma opção
- Pior caso: sistematicamente, o pivô é o menor ou o maior elemento do array, eliminando um elemento a cada chamada do algoritmo. $C(n) = \Theta(n^2)$. Quando escolhemos o primeiro ou o último elemento e o array está em ordem crescente ou decrescente