

EDB-I

Estrutura de Dados Básicas I

Aula 19

TAD - Sequência

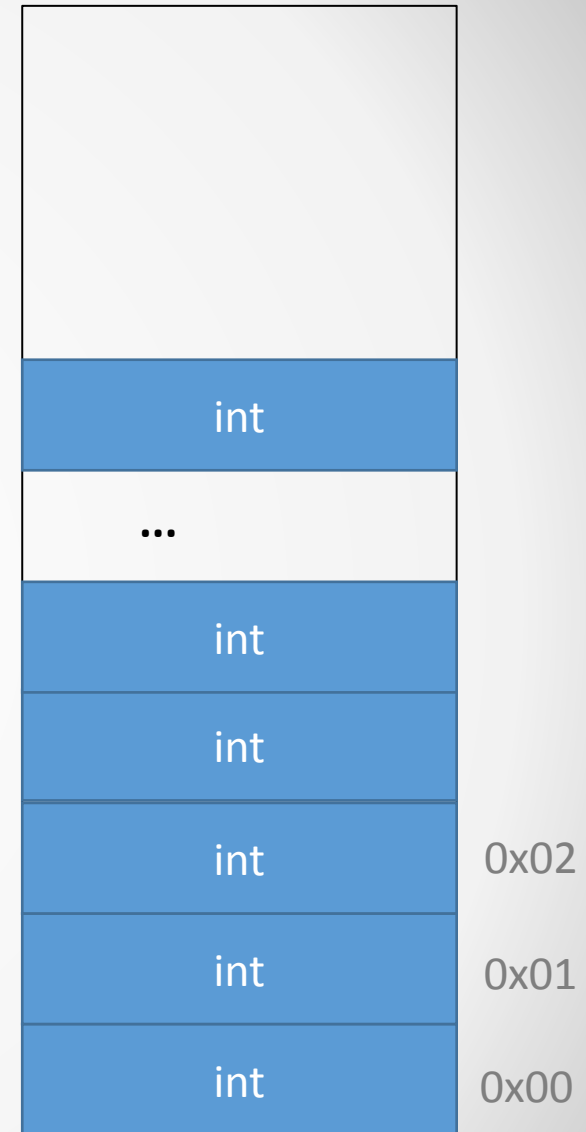
(material baseado nas notas de aula do Prof. César Rennó Costa e Prof. Eiji Adachi)

TAD Sequência

- Definição
 - Conjunto de elementos organizados em uma sequência, i.e., com relação de precedência.
- Operações
 - Criar
 - Acessar (início, posição aleatória, fim)
 - Inserir (início, posição aleatória, fim)
 - Remover (início, posição aleatória, fim)
 - Destruir

Lista (linear) sequencial

- Estrutura de dados que implementa um TAD Sequência
- Elementos são armazenados em um bloco de dados de maneira **contínua**
- Todos os dados devem ter o **mesmo tipo**
- Implementação com base em **array**



Criando uma lista sequencial

Alocação estática

Tamanho definido em tempo de compilação

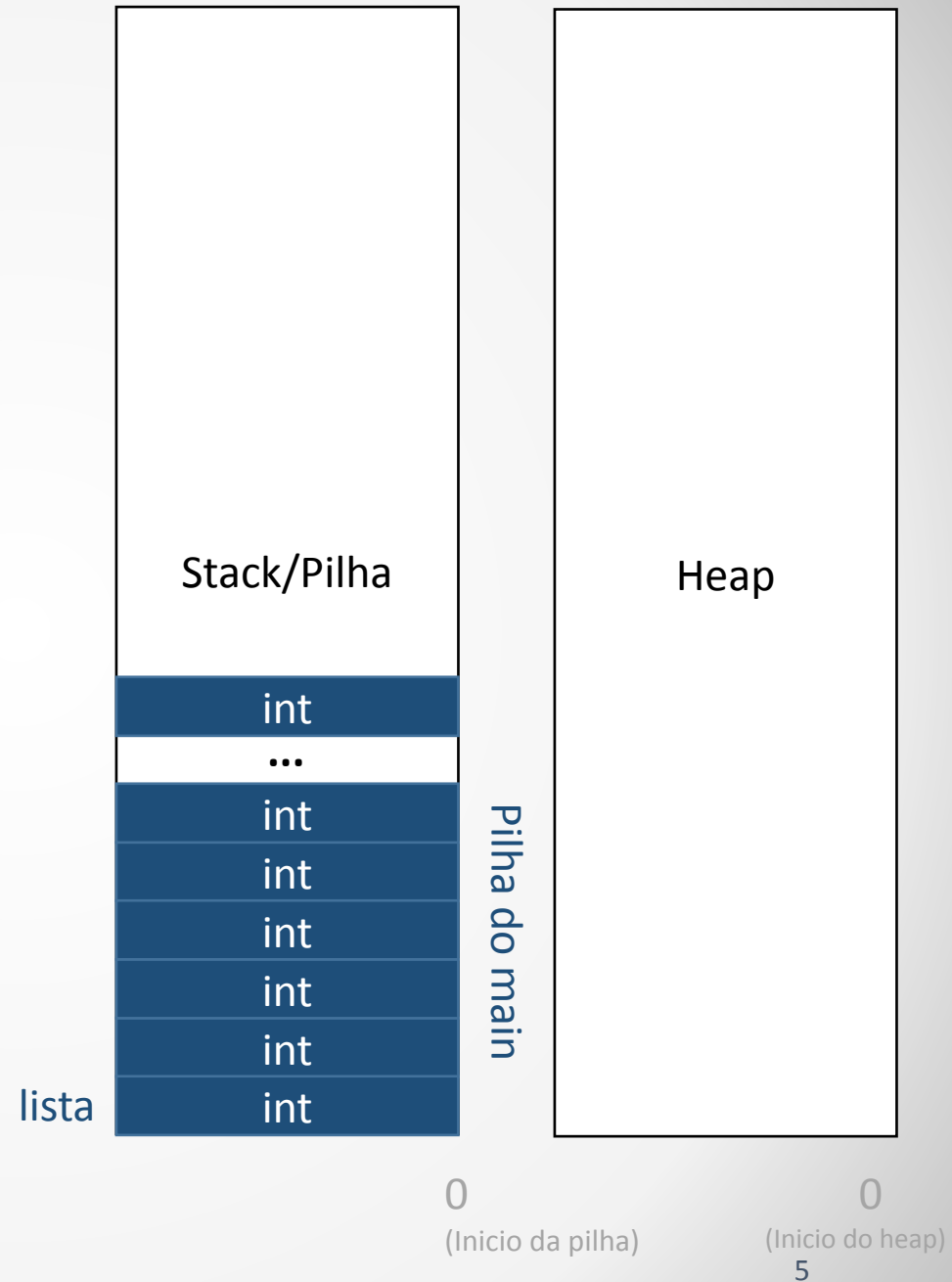
```
int lista[20];
```

Alocação dinâmica

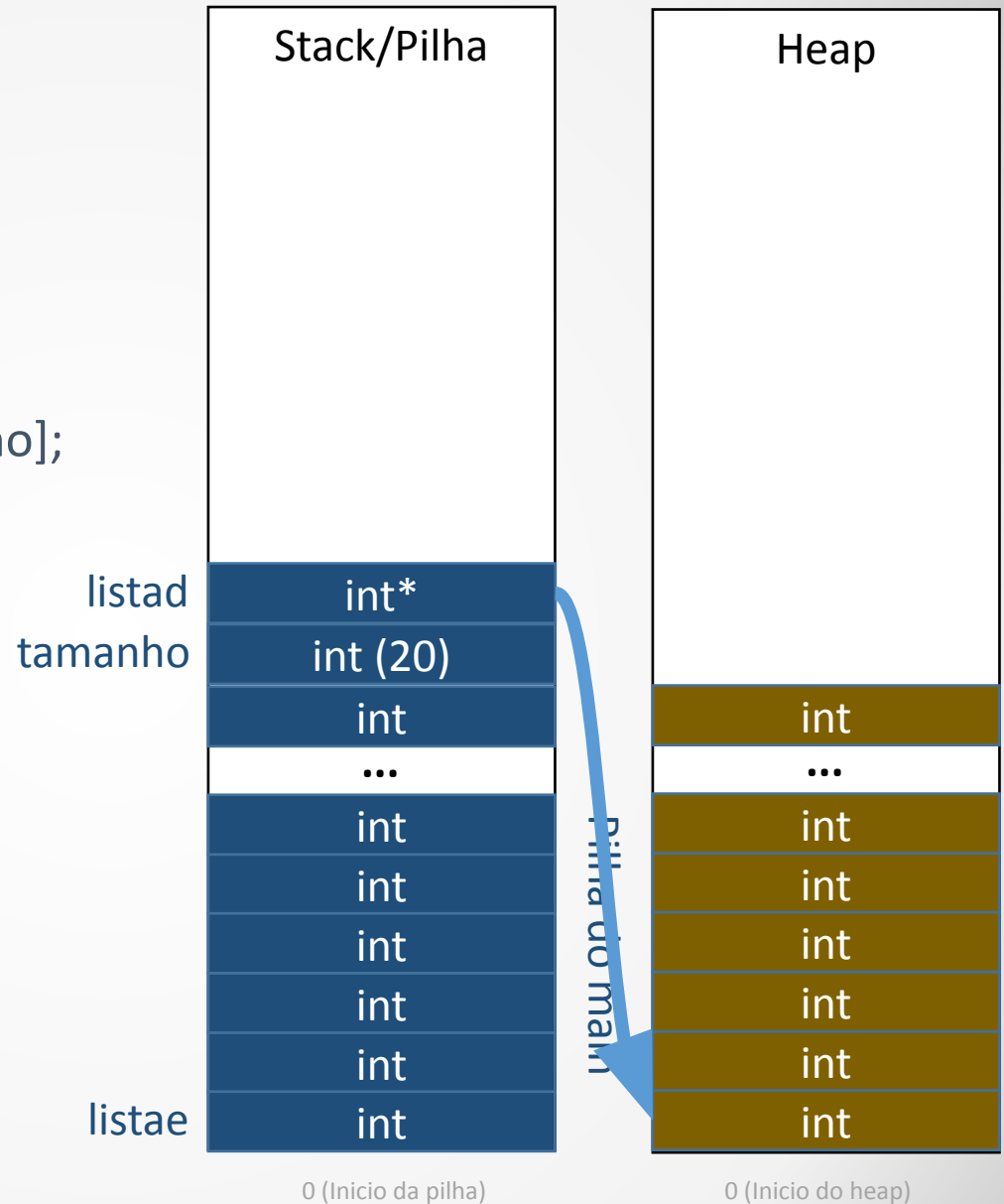
Tamanho definido em tempo de execução

```
int tamanho = 20;  
int* lista = new lista[tamanho];
```

- Alocação estática
 - `int lista[20]`




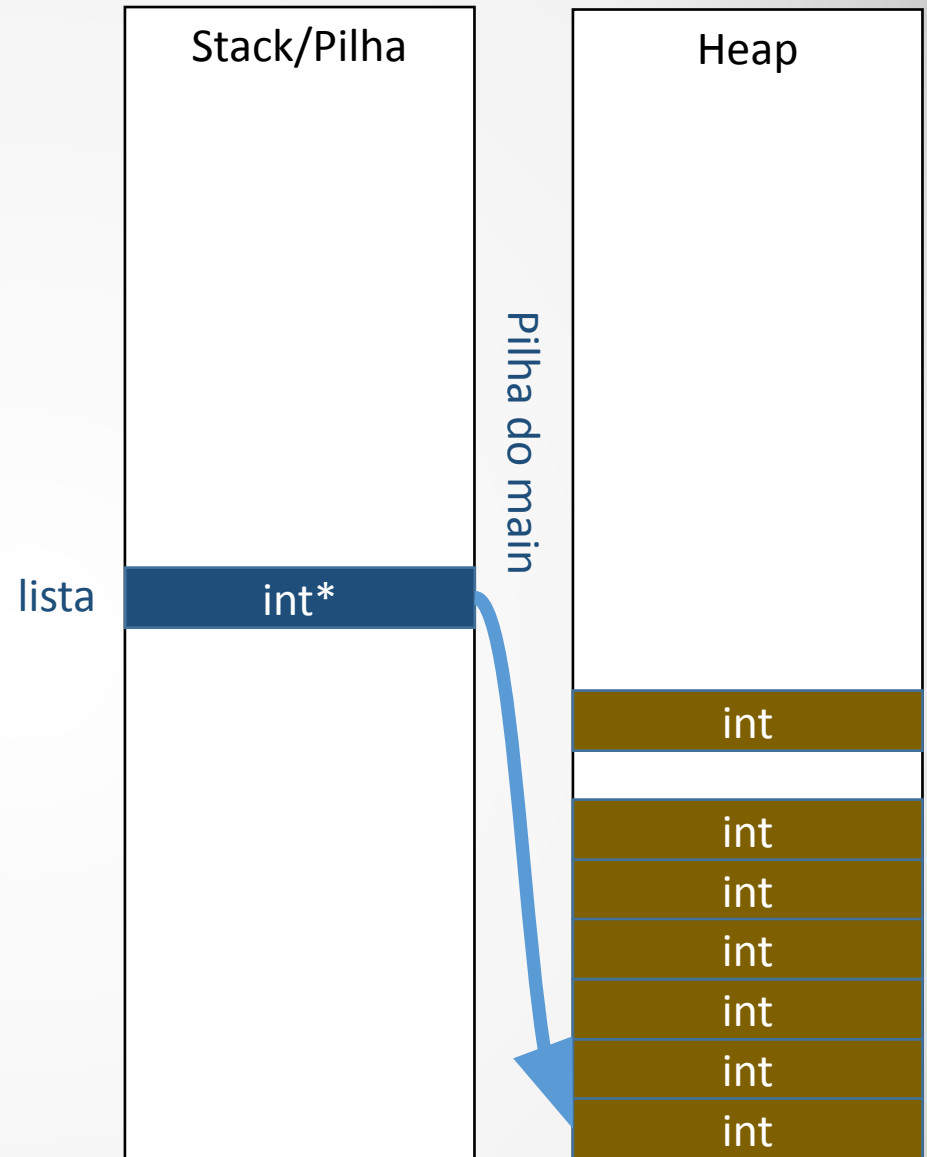
- Alocação estática
 - `int listae[20];`
- Alocação dinâmica
 - `int tamanho = 20;`
`int* listad = new int[tamanho];`



Criando uma lista sequencial dinâmica

```
int* criarListaSequencial(int tamanho){  
    return new int[tamanho];  
}
```

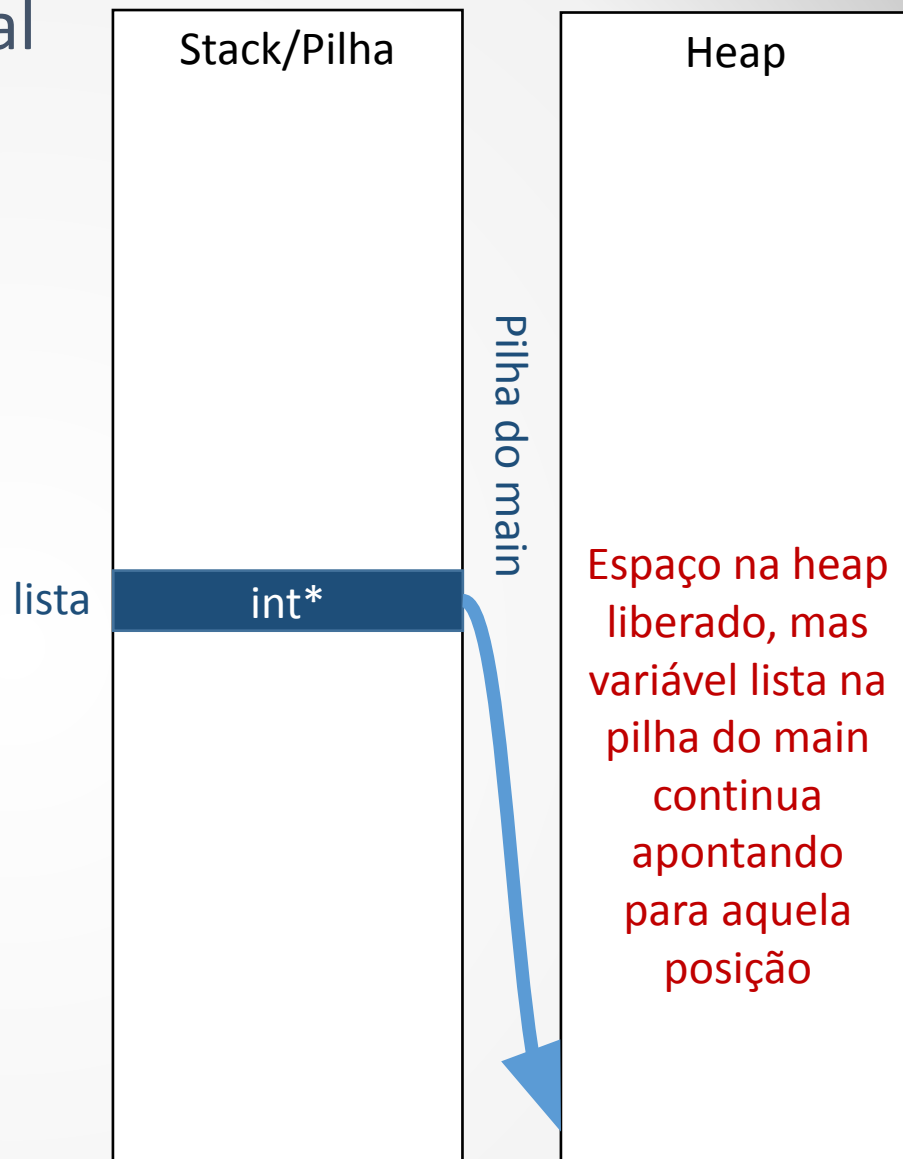

```
int main(){  
    ...  
    int* lista = criarListaSequencial(10);  
      
}
```



Destruindo uma lista sequencial dinâmica

```
void destruirListaSequencial(int* lista){  
    if(lista != NULL) delete[] lista;  
}
```

```
int main(){  
    ...  
    int* lista = criarListaSequencial(10);  
    destruirListaSequencial(lista);  
    ...  
}
```



Destruindo uma lista sequencial dinâmica

```
void destruirListaSequencial(int*& lista){  
    if(lista != nullptr) delete[] lista;  
    lista = nullptr;  
}
```

```
int main(){  
    ...  
    int *lista = criarListaSequencial(10);  
    destruirListaSequencial(lista);  
    ...  
}
```



lista

Stack/Pilha

int*

Pilha do main

Heap

Espaço na heap
liberado e
variável lista
agora aponta
para nullptr,
opção mais
segura

Destruindo uma lista sequencial dinâmica

```
void destruirListaSequencial(int** lista){  
    if(*lista != nullptr) delete[] *lista;  
    *lista = nullptr;  
}
```

```
int main(){  
    ...  
    int *lista = criarListaSequencial(10);  
    destruirListaSequencial(&lista);  
    ...  
}
```

lista

Stack/Pilha

int*

Pilha do main

Heap

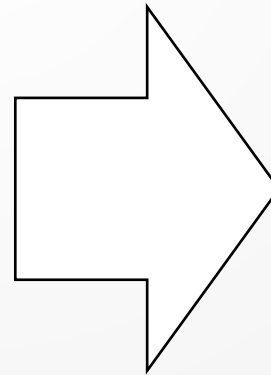
Espaço na heap
liberado e
variável lista
agora aponta
para nullptr,
opção mais
segura

TAD Sequência

- Dados
 - Elementos em sequência
- Operações
 - ~~Criar~~
 - Acessar (início, meio, fim)
 - Inserir (início, meio, fim)
 - Remover (início, meio, fim)
 - ~~Destruir~~

Acessar

- Elementos são posicionados em sequência
- Cada elemento tem um tamanho pré-definido



$N-1 * x04$

x20

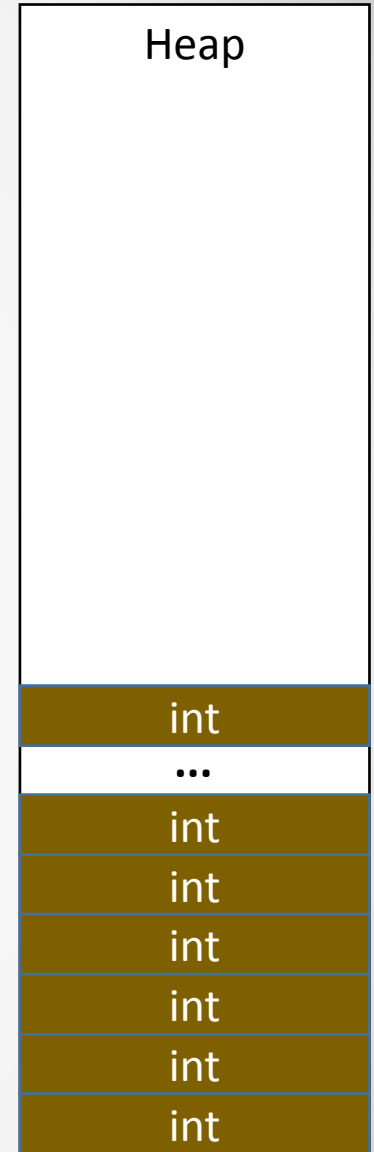
x16

x12

x08

x04

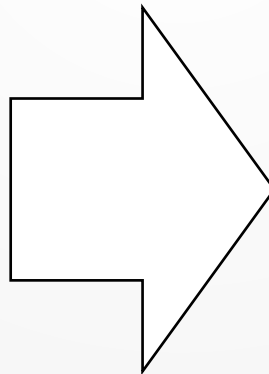
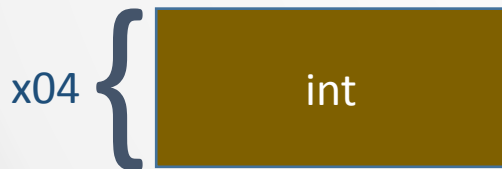
x00



0 (Início do heap)

Acessar

- Elementos são posicionados em sequência
- Cada elemento tem um tamanho pré-definido
- É possível calcular a posição exata de cada elemento na memória por simples aritmética de ponteiros



$\text{lista} + (N-1) * \text{x04}$

$\text{lista} + 5 * \text{x04}$

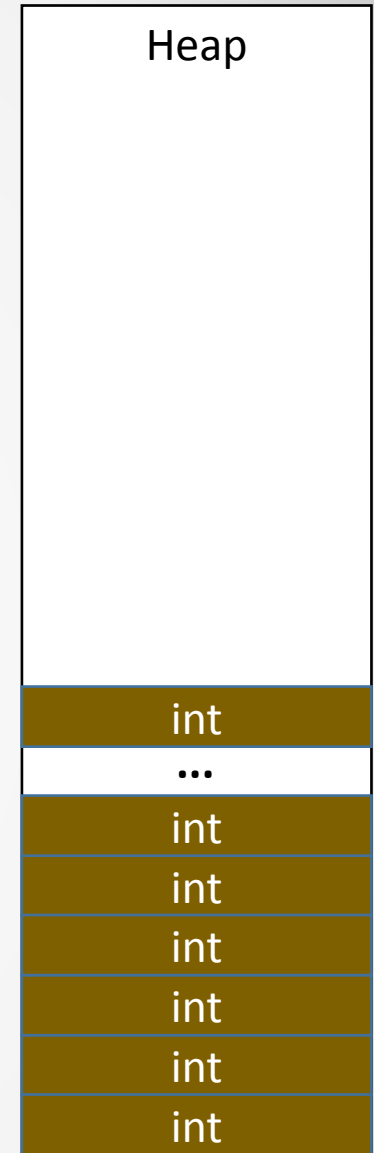
$\text{lista} + 4 * \text{x04}$

$\text{lista} + 3 * \text{x04}$

$\text{lista} + 2 * \text{x04}$

$\text{lista} + 1 * \text{x04}$

$\text{lista} + 0 * \text{x04}$



0 (Início do heap)

Acessar

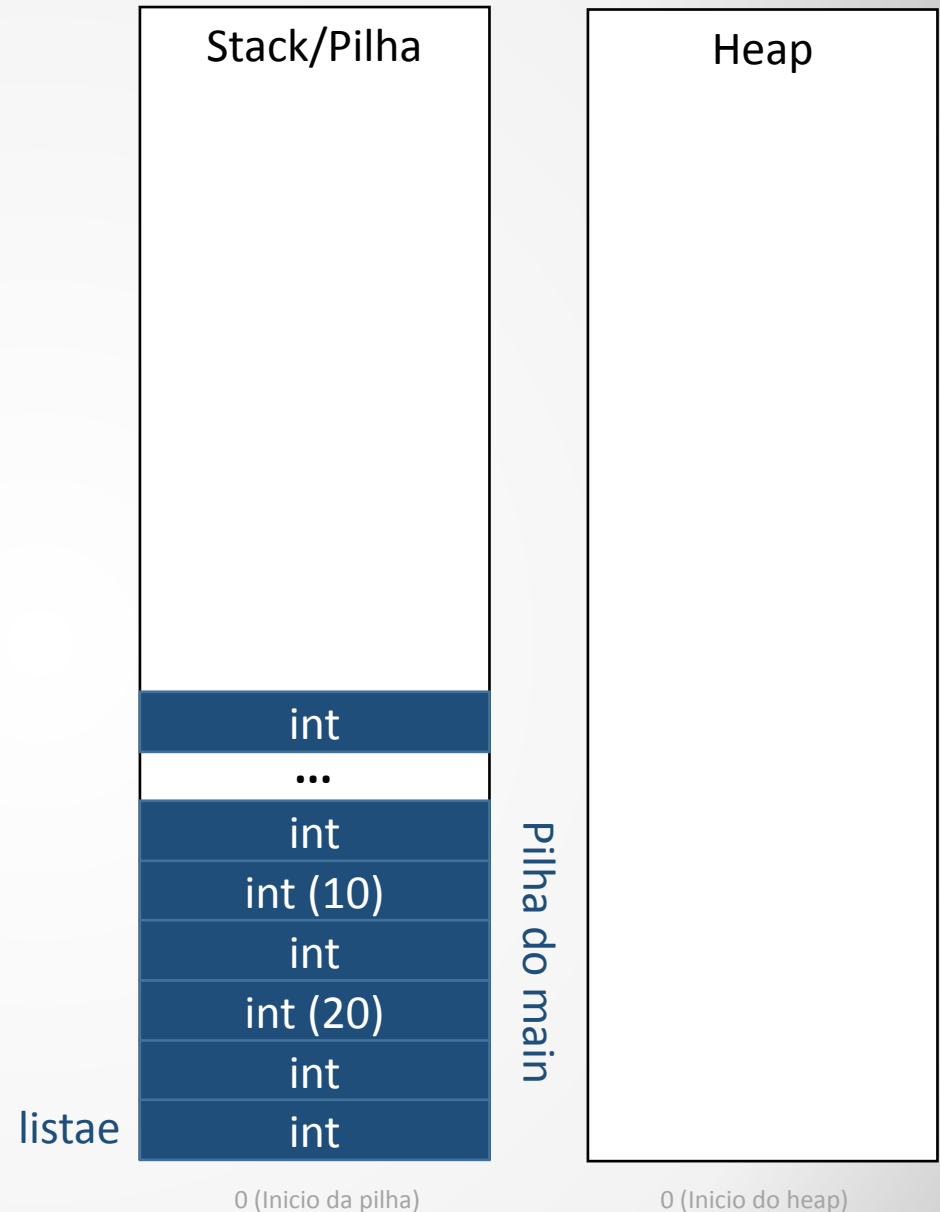
- Acesso a um elemento

- Alocação estática

```
int listae[20]
```

```
//Duas formas iguais de acessar  
listae[2] = 20;
```

```
*(listae + 4*sizeof(int)) = 10;
```



Acessar

- Acesso a um elemento

- Alocação dinâmica

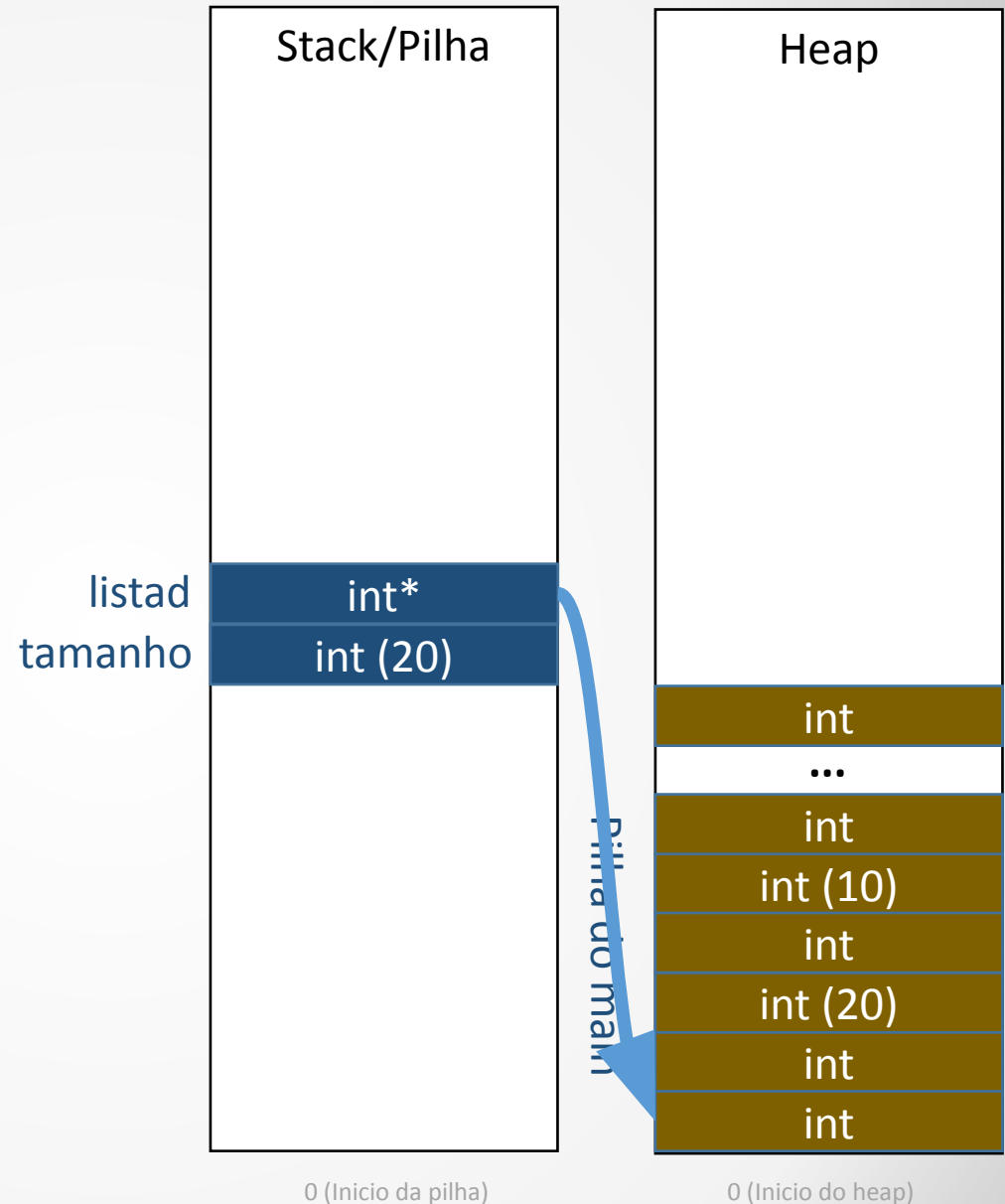
```
int tamanho = 20;
```

```
int *listad = new int[tamanho]
```

```
//Duas formas iguais de acessar
```

```
listad[2] = 20;
```

```
*(listad + 4*sizeof(int)) = 10;
```



Acesso

- Acesso ao primeiro elemento
lista[0]
- Acesso ao último elemento
 - necessário saber a **quantidade** (Q) de elementos
lista[Q-1]

Acessar

Acesso na posição **pos**

```
int acessarListaS(int* lista, int qtd, int pos){  
    if(pos >= qtd || pos < 0) return -1;  
    return lista[pos];  
}
```

Constante

Acesso no início

```
int acessarInicioListaS(int* lista, int qtd){  
    if(qtd == 0) return -1;  
    return lista[0];  
}
```

Constante

Acesso no fim

```
int acessarFimListaS(int* lista, int qtd){  
    if(qtd == 0) return -1;  
    return lista[qtd-1];  
}
```

Constante

TAD Sequência

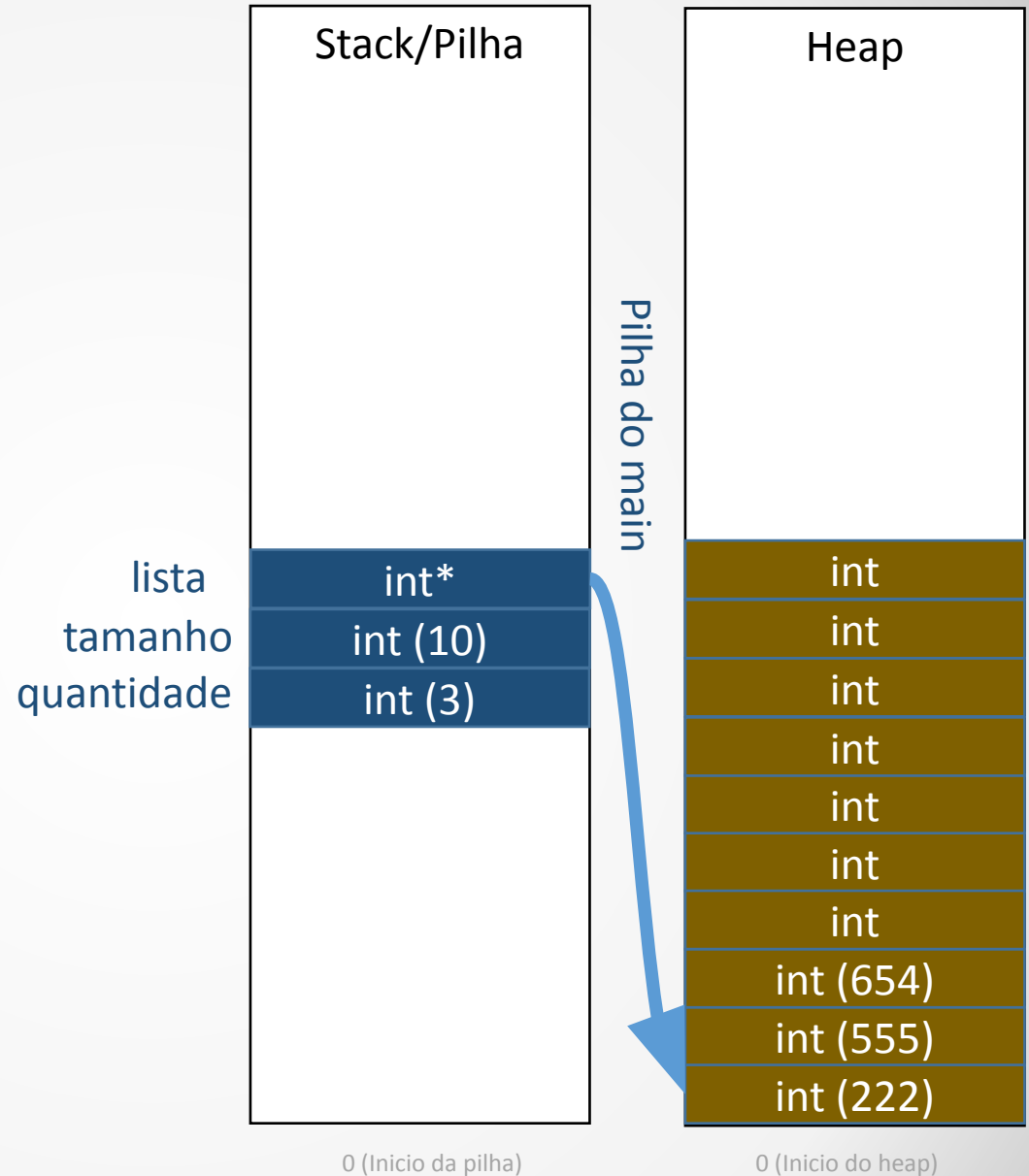
- Dados
 - Elementos em sequência
- Operações
 - ~~Criar~~
 - ~~Acessar (início, meio, fim)~~
 - Inserir (início, meio, fim)
 - Remover (início, meio, fim)
 - ~~Destruir~~

Inserção

- A lista deve manter todos os elementos organizados de maneira contínua
- Inserção no início:
 - 3 5 4 3 4 5 6 -> 9 3 5 4 3 4 5 6
- Inserção no fim:
 - 3 5 4 3 4 5 6 -> 3 5 4 3 4 5 6 9
- Inserção na posição P:
 - 3 5 4 3 4 5 6 -> 3 5 4 9 3 4 5 6

Inserir no fim

```
int main(){  
    int tamanho = 10;  
    int quantidade = 0;  
    int* lista = criarListaS (tamanho);  
    // inserir 222 no fim  
    // inserir 555 no fim  
    // inserir 654 no fim  
    ...  
}
```



Inserir no fim

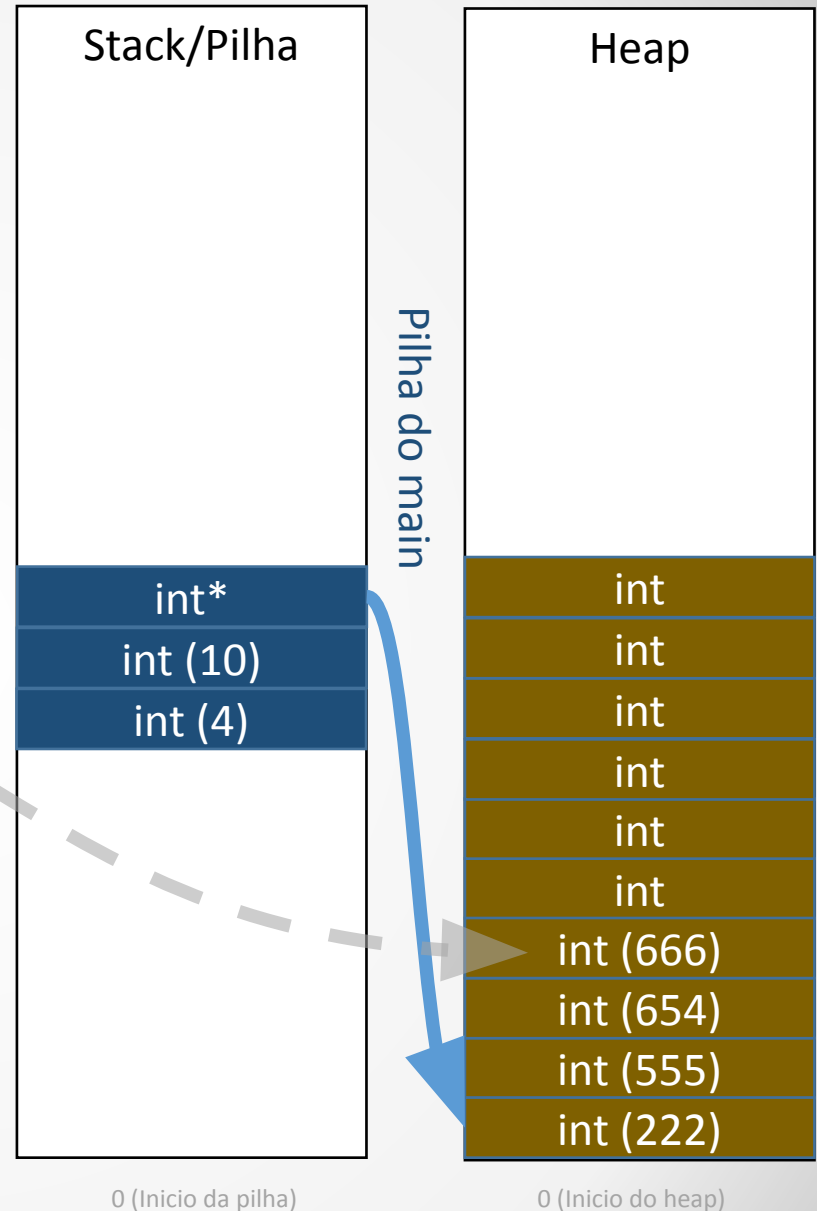
```
int main(){  
    int tamanho = 10;  
    int quantidade = 0;  
    int* lista = criarListaS (tamanho);  
    // inserir 222 no fim  
    // inserir 555 no fim  
    // inserir 654 no fim  
    // inserir 666 no fim  
    ...  
}
```

Dados necessários para uma função de inserção:

- (1) Ponteiro para a lista
- (2) Tamanho da lista
- (3) Quantidade de elementos



lista
tamanho
quantidade

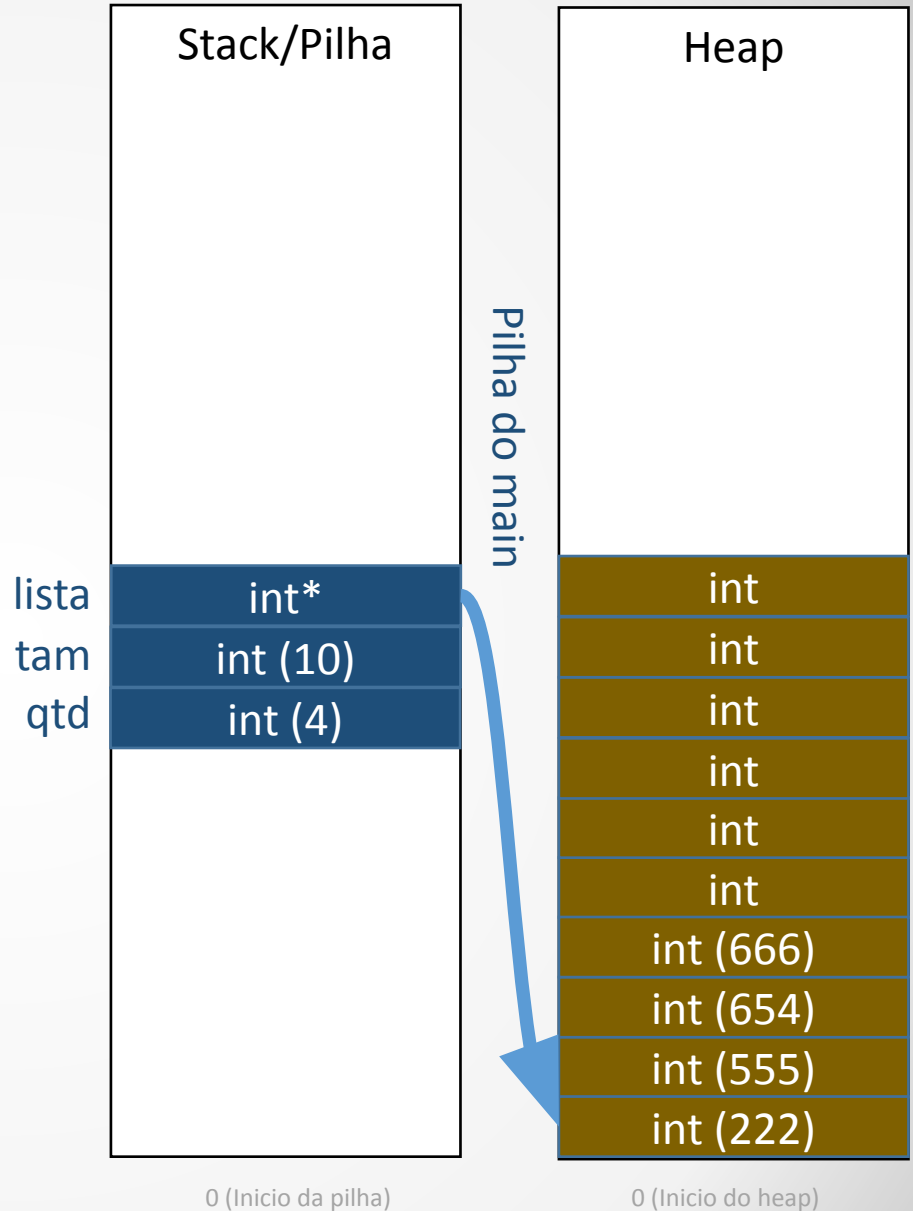


Inserir no fim

```
int main(){
    int tam = 10;
    int qtd = 0;
    int* lista = criarListaS (tam);
    inserirFimS(lista, tam, qtd, 222);
    inserirFimS(lista, tam, qtd, 555);
    inserirFimS(lista, tam, qtd, 654);
    inserirFimS(lista, tam, qtd, 666);
    ...
}
```

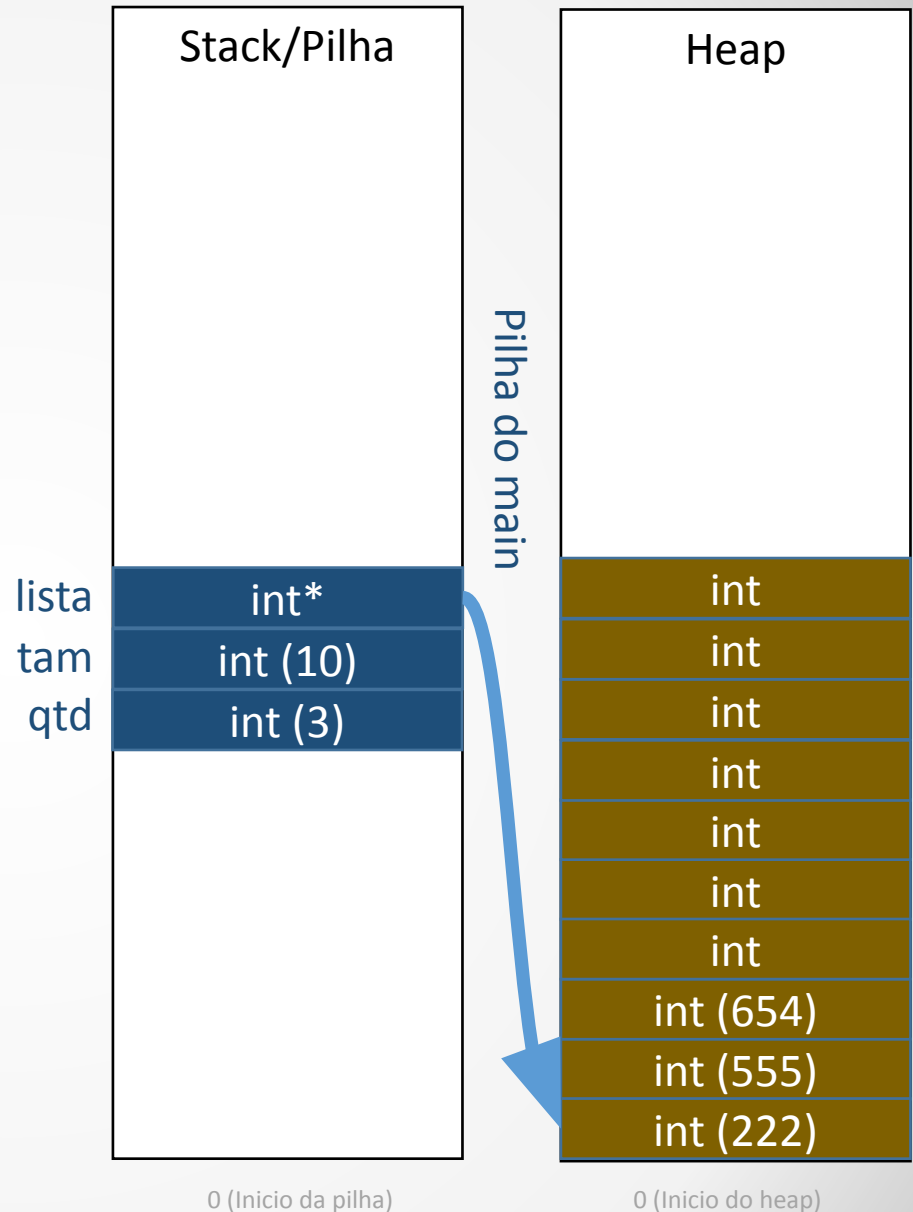
Passagem por referência
pois Q será modificado

```
void inserirFimS(int* ls, int N, int& Q, int elem){
    if(ls==nullptr || Q==N) return;
    ls[Q] = elem; // insere na última posição
    Q++;         // incrementa a qtd de elementos
}
```



Inserir no início

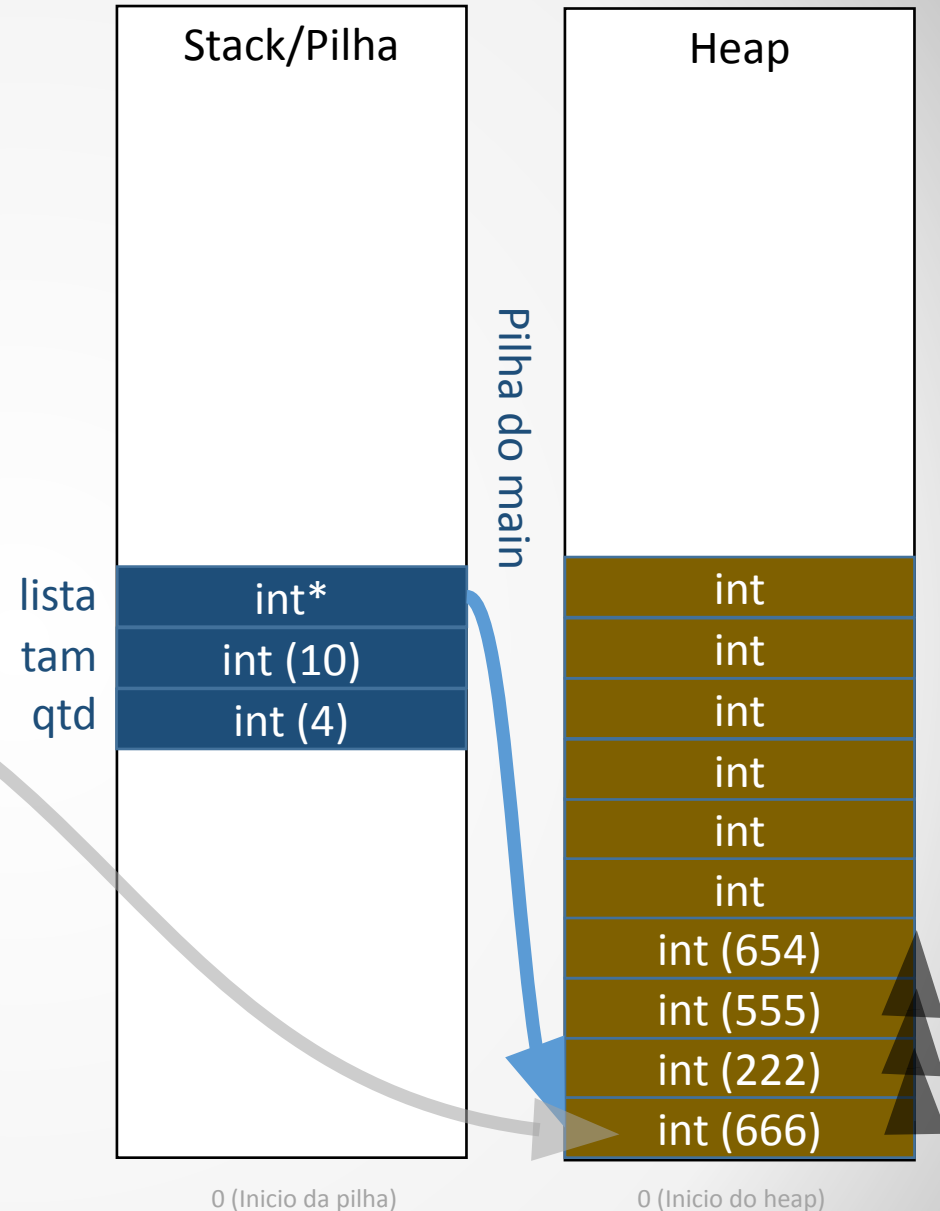
```
int main(){  
    int tamanho = 10;  
    int quantidade = 0;  
    int* lista = criarListaS (tamanho);  
    inserirFimS(lista, tam, qtd, 222);  
    inserirFimS(lista, tam, qtd, 555);  
    → inserirFimS(lista, tam, qtd, 654);  
    // inserir 666 no início  
    ...  
}
```



Inserir no início

```
int main(){  
    int tam = 10;  
    int qtd = 0;  
    int* lista = criarListaS (tam);  
    inserirFimS(lista, tam, qtd, 222);  
    inserirFimS(lista, tam, qtd, 555);  
    inserirFimS(lista, tam, qtd, 654);  
    ➡ // inserir 666 no início  
    ...  
}
```

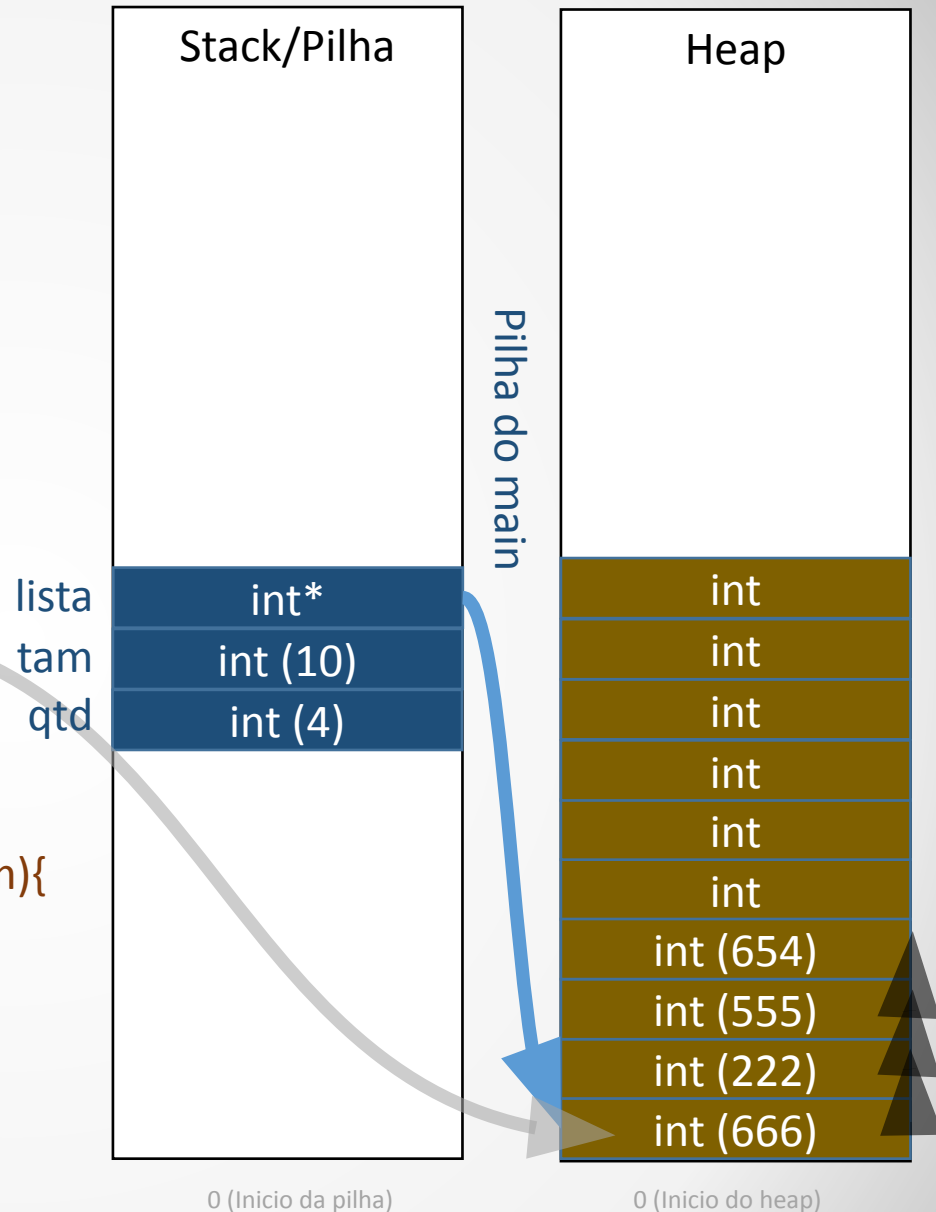
Inserção no início requer o deslocamento dos outros elementos da lista.



Inserir no início

```
int main(){  
    int tam = 10;  
    int qtd = 0;  
    int *lista = criarListaS (tam);  
    inserirFimS(lista, tam, qtd, 222);  
    inserirFimS(lista, tam, qtd, 555);  
    inserirFimS(lista, tam, qtd, 654);  
    ➡ inserirInicioS(lista, tam, qtd, 666);  
    ...  
}
```

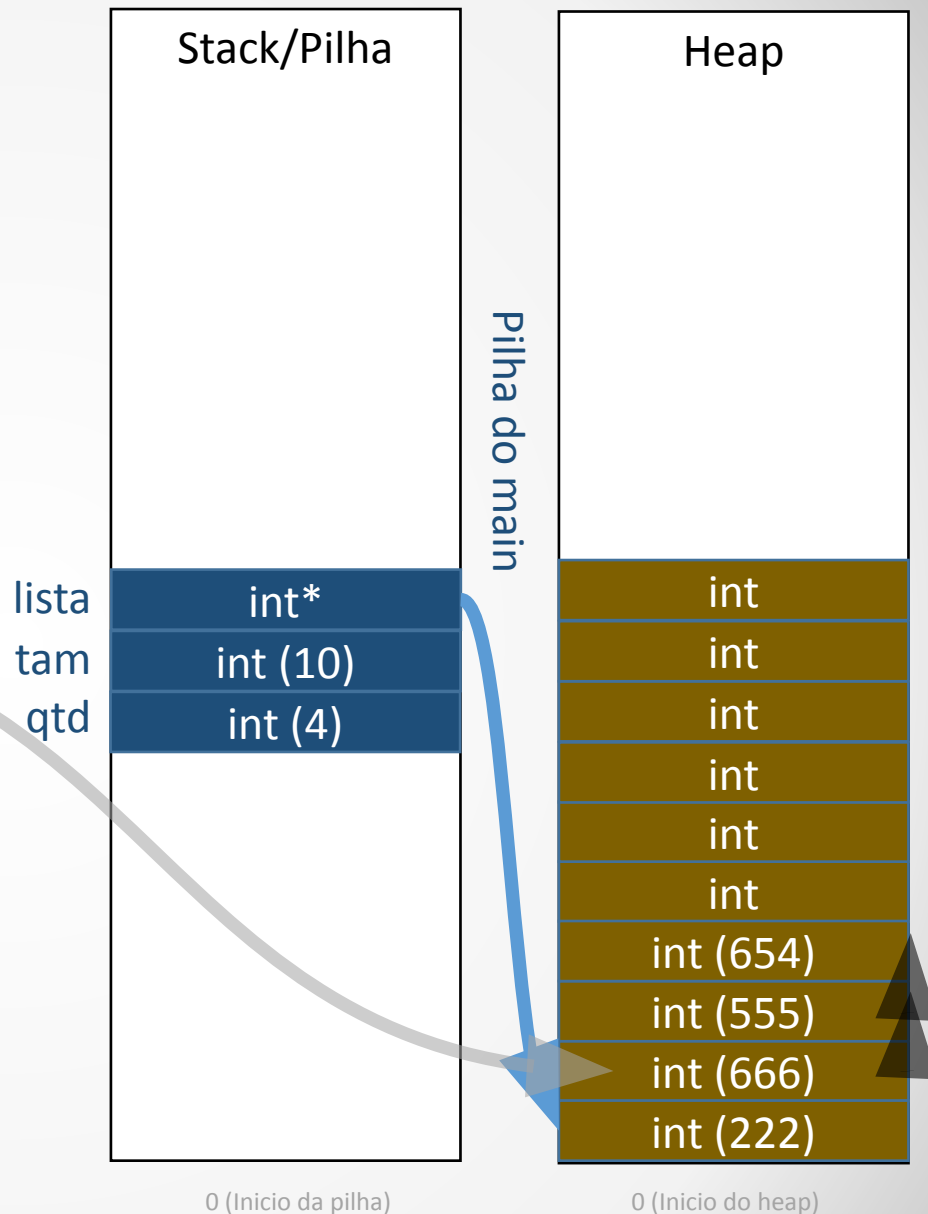
```
void inserirInicioS(int* ls, int N, int& Q, int elem){  
    if(ls==nullptr || Q==N) return;  
    for(int i=Q; i>0; i--) ls[i] = ls[i-1]; //empurra  
    ls[0] = elem;    // insere na 1a posição  
    Q++;    // incrementa a qtd de elementos  
}
```



Inserir no meio

```
void main(){  
    int tam = 10;  
    int qtd = 0;  
    int *lista = criarListaS (tam);  
    inserirFimS(lista, tam, qtd, 222);  
    inserirFimS(lista, tam, qtd, 555);  
    inserirFimS(lista, tam, qtd, 654);  
    ➡ // inserir 666 na 2a posição  
    ...  
}
```

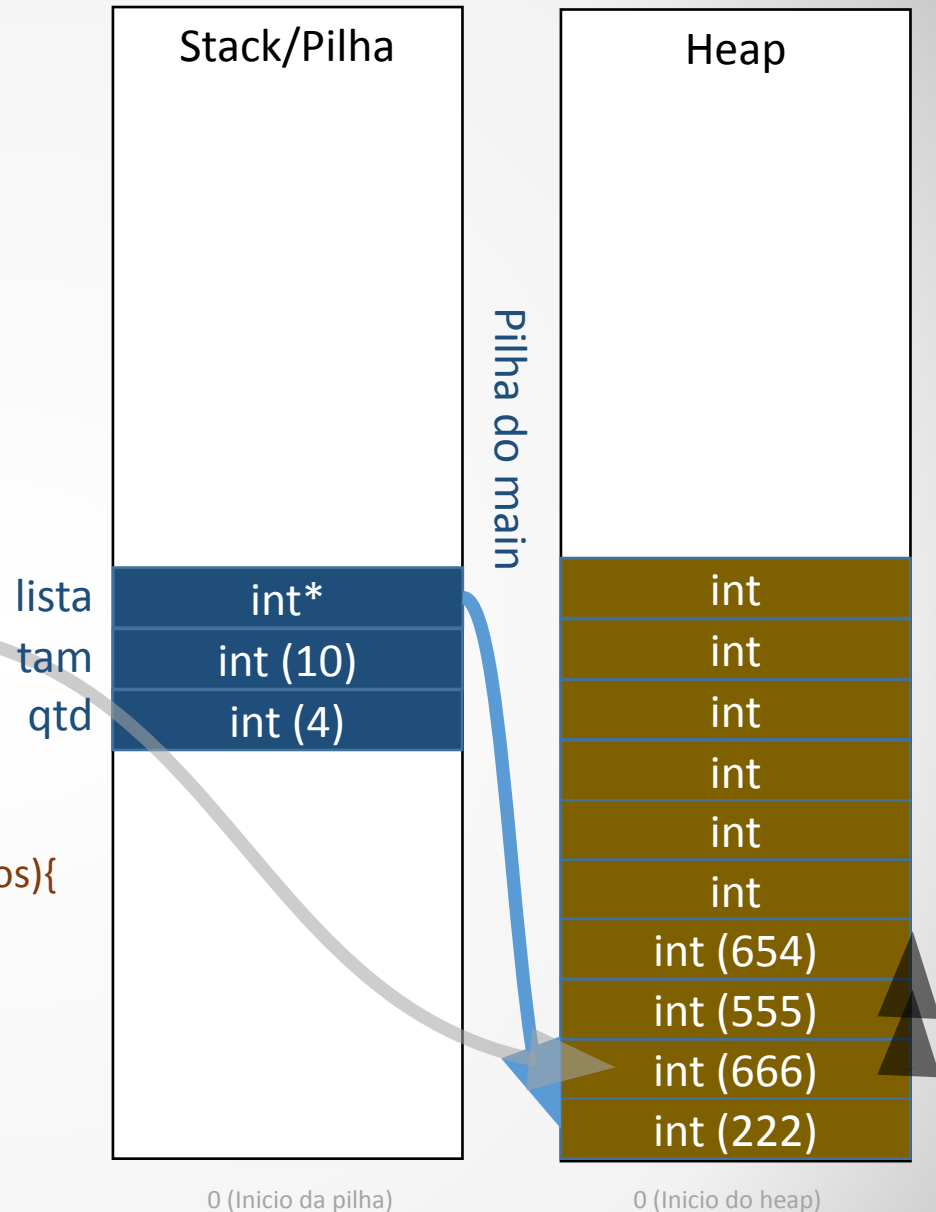
Inserção no meio requer o deslocamento dos elementos da lista subsequentes a esta posição.



Inserir no meio

```
void main(){  
    int tam = 10;  
    int qtd = 0;  
    int *lista = criarListaS (tam);  
    inserirFimS(lista, tam, qtd, 222);  
    inserirFimS(lista, tam, qtd, 555);  
    inserirFimS(lista, tam, qtd, 654);  
    → inserirMeioS(lista, tam, qtd, 666, 1);  
    ...  
}
```

```
void inserirMeioS(int* ls, int N, int& Q, int elem, int pos){  
    if(ls==nullptr || Q==N || pos >= Q) return;  
    for(int i=Q; i>pos; i--) ls[i] = ls[i-1]; //empurra  
    ls[pos] = elem; // insere na 1a posição  
    Q++; // incrementa a qtd de elementos  
}
```



Inserir

Inserir na posição **pos**

```
void inserirMeioS(int* ls, int N, int& Q, int elem, int pos){  
    if(ls==nullptr || Q==N || pos >= Q) return;  
    for(int i=Q; i>pos; i--) ls[i] = ls[i-1];  
    ls[pos] = elem;  
    Q++;  
}
```

Linear

Inserir no inicio

```
void inserirInicioS(int* ls, int N, int& Q, int elem, int pos){  
    if(ls==nullptr || Q==N) return;  
    for(int i=Q; i>0; i--) ls[i] = ls[i-1];  
    ls[0] = elem;  
    Q++;  
}
```

Linear

Inserir no fim

```
void inserirFimS(int *ls, int N, int& Q, int elem, int pos){  
    if(ls==nullptr || Q==N) return;  
    ls[Q] = elem;  
    Q++;  
}
```

Constante

TAD Sequência

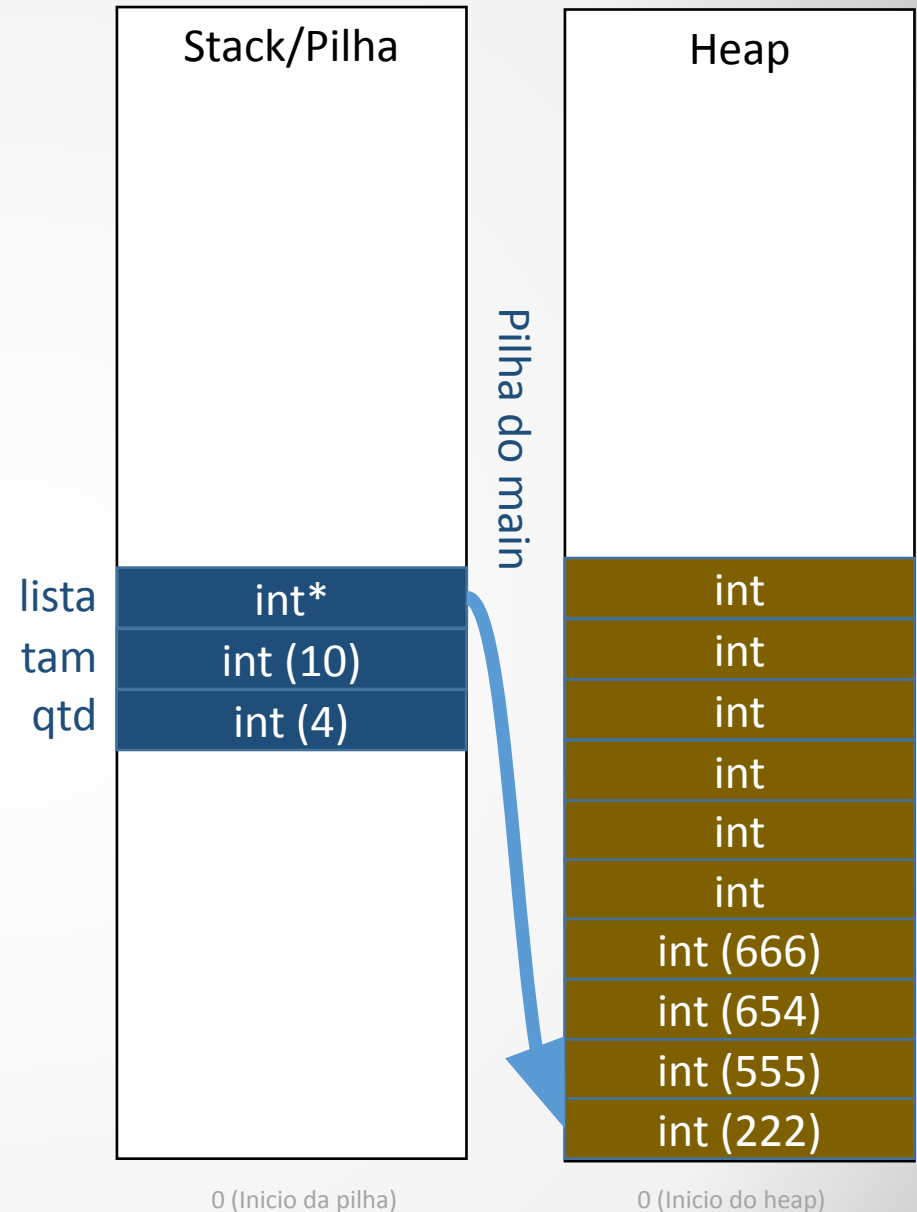
- Dados
 - Elementos em sequência
- Operações
 - ~~Criar~~
 - ~~Acessar (início, meio, fim)~~
 - ~~Inserir (início, meio, fim)~~
 - Remover (início, meio, fim)
 - ~~Destruir~~

Remoção

- A lista deve manter todos os elementos organizados de maneira contínua
- Remoção do início:
 - 9 3 5 4 3 4 5 6 -> 3 5 4 3 4 5 6
- Remoção do fim:
 - 3 5 4 3 4 5 6 9 -> 3 5 4 3 4 5 6
- Remoção da posição P:
 - 3 5 4 3 9 4 5 6 -> 3 5 4 3 4 5 6

Remover

```
void main(){  
    int tam = 10;  
    int qtd = 0;  
    int* lista = criarListaS (tam);  
    ➔ // inserir 222, 555, 654, 666 no fim  
    ...  
}
```

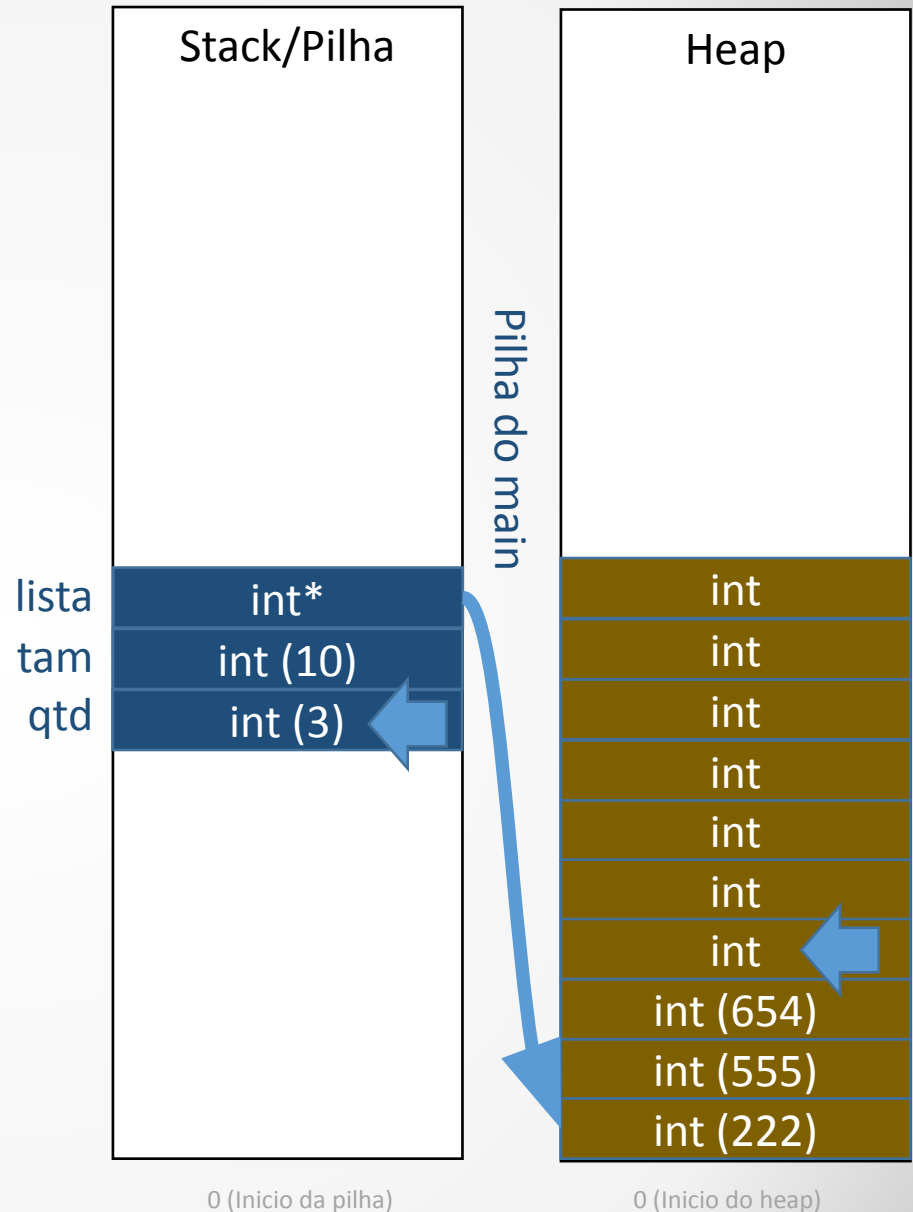


Remover do fim

```
void main(){  
    int tam = 10;  
    int qtd = 0;  
    int *lista = criarListaS (tam);  
    // inserir 222, 555, 654, 666 no fim  
    ➡ // remover do fim  
    ...  
}
```

Dados necessários para uma função de remoção:

- 1) Ponteiro para a lista
- 2) Quantidade de elementos



Remover do fim

```
void main(){  
    int tam = 10;  
    int qtd = 0;  
    int *lista = criarListaS (tam);  
    // inserir 222, 555, 654, 666 no fim  
    removerFimS(lista, qtd);
```

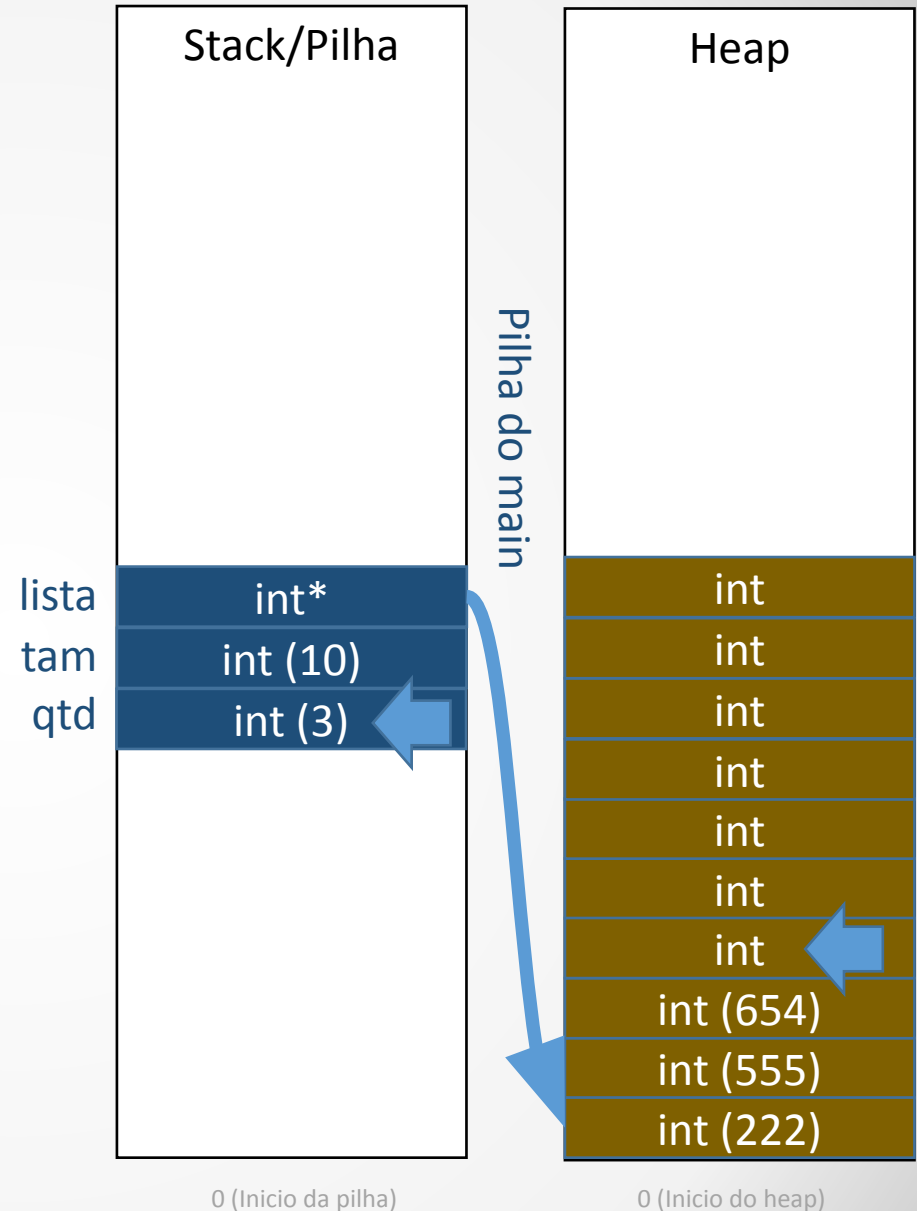


...

}

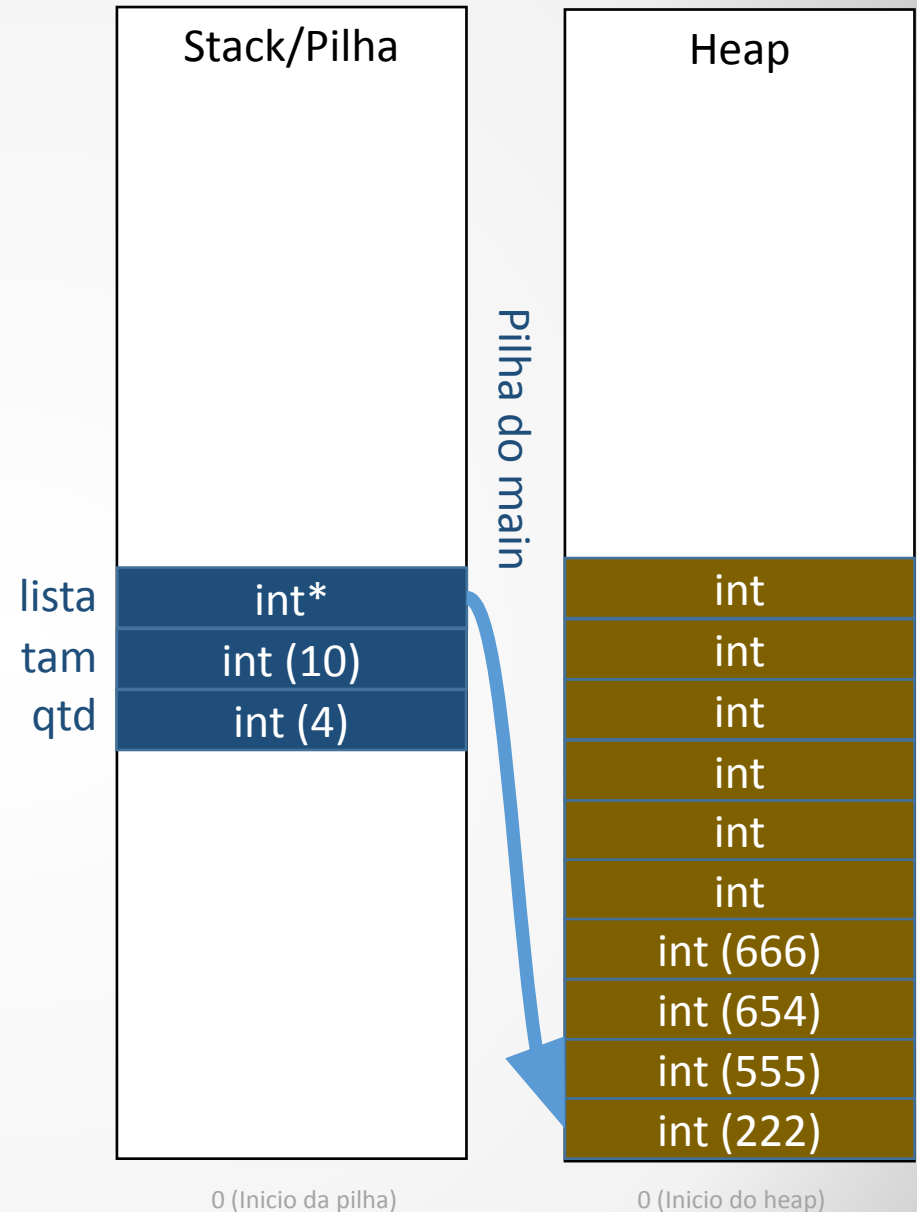
```
void removerFimS(int* ls, int& Q){  
    if(ls==nullptr || Q==0) return;  
    Q--;  
}
```

não é necessário apagar o elemento ls[Q], mas pode ser interessante fazê-lo dependendo do programa



Remover

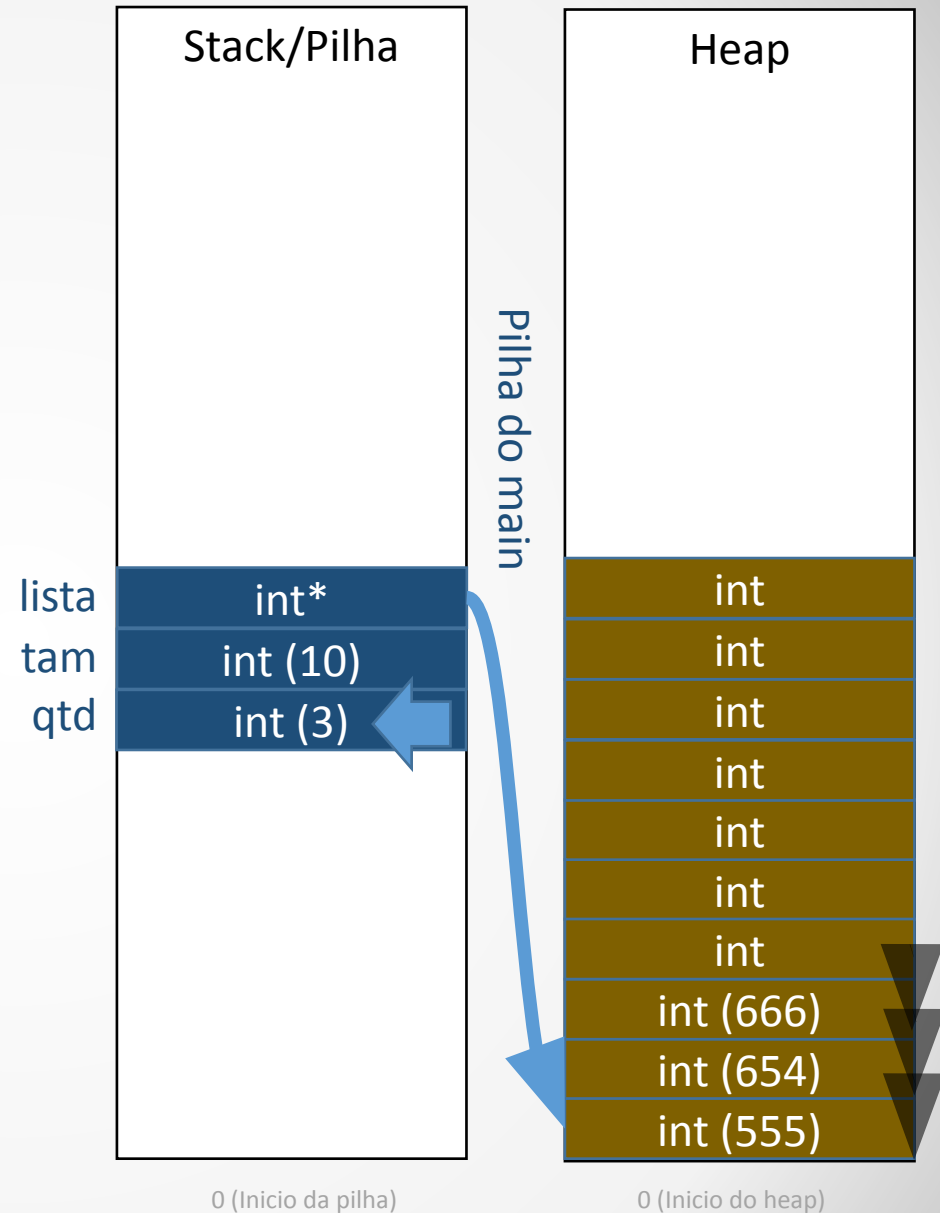
```
void main(){  
    int tam = 10;  
    int qtd = 0;  
    int *lista = criarListaS (tam);  
    ➔ // inserir 222, 555, 654, 666 no fim  
    ...  
}
```



Remover do início



Remoção no início requer o deslocamento dos outros elementos da lista.

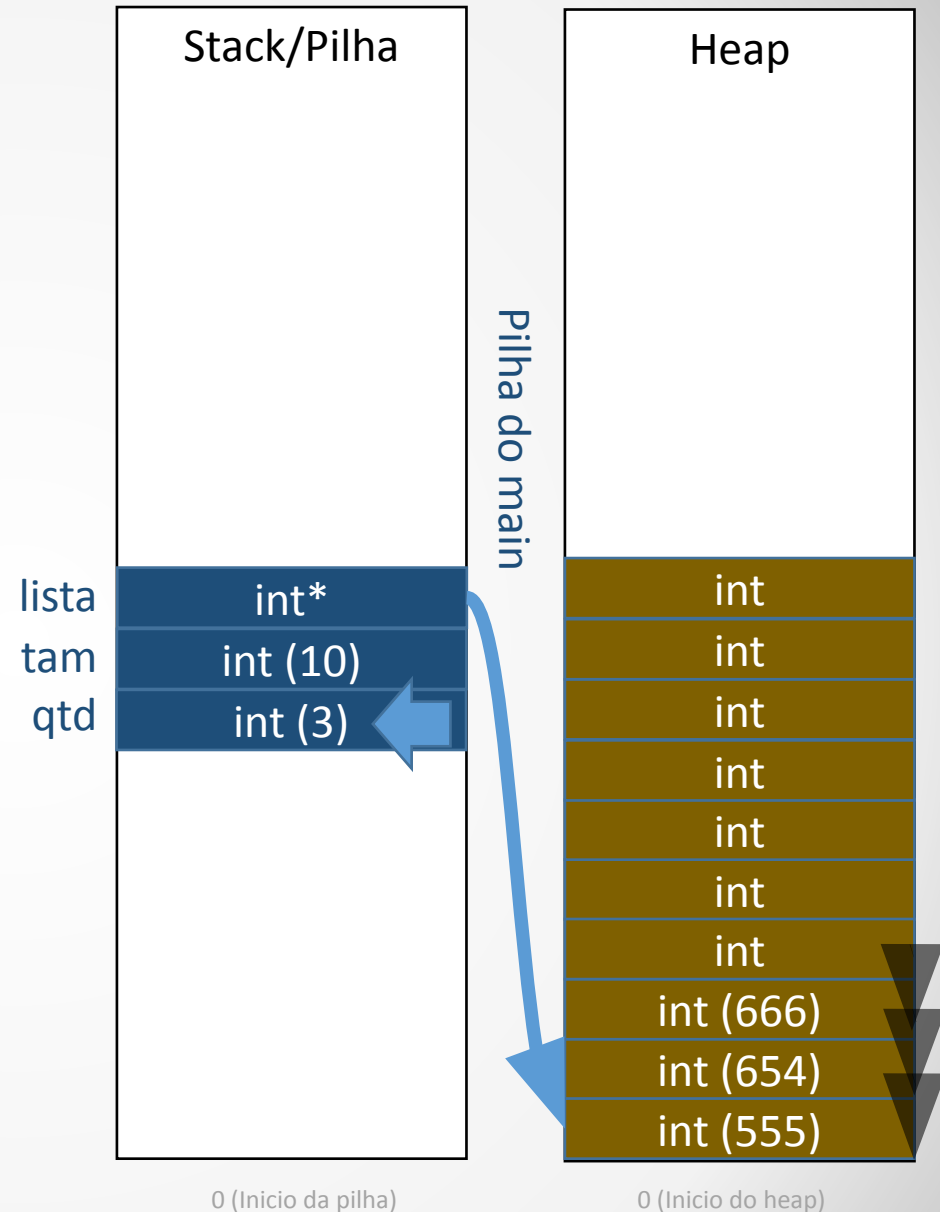


Remover do início

```
void main(){  
    int tam = 10;  
    int qtd = 0;  
    int *lista = criarListaS (tam);  
    // inserir 222, 555, 654, 666 no fim  
    removerInicioS(lista, qtd);  
    ...  
}
```

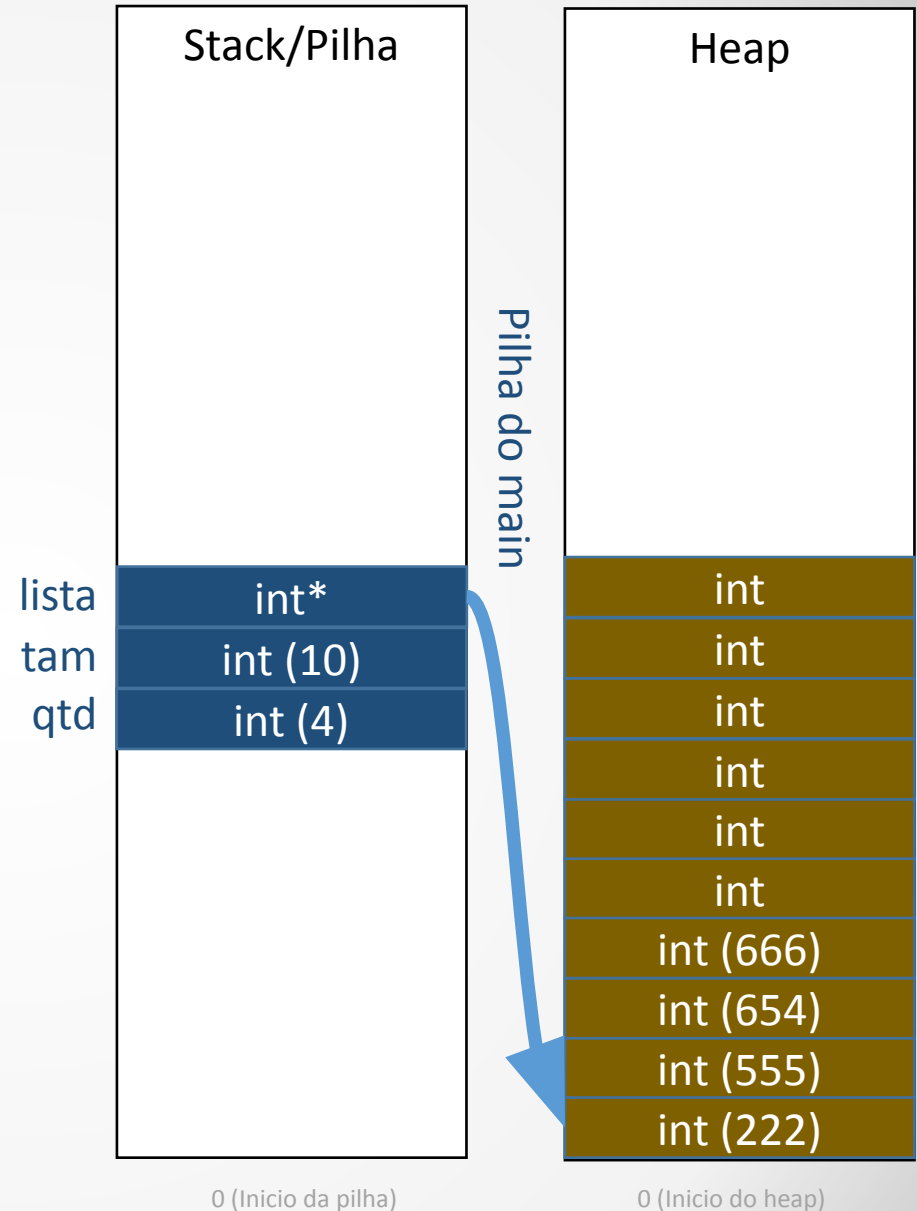


```
void removerInicioS(int* ls, int& Q){  
    if(ls==nullptr || Q==0) return;  
    for(int i=0; i<(Q-1); i++) ls[i] = ls[i+1];  
    Q--;  
    // não é necessário apagar o elemento ls[0]  
}
```



Remover

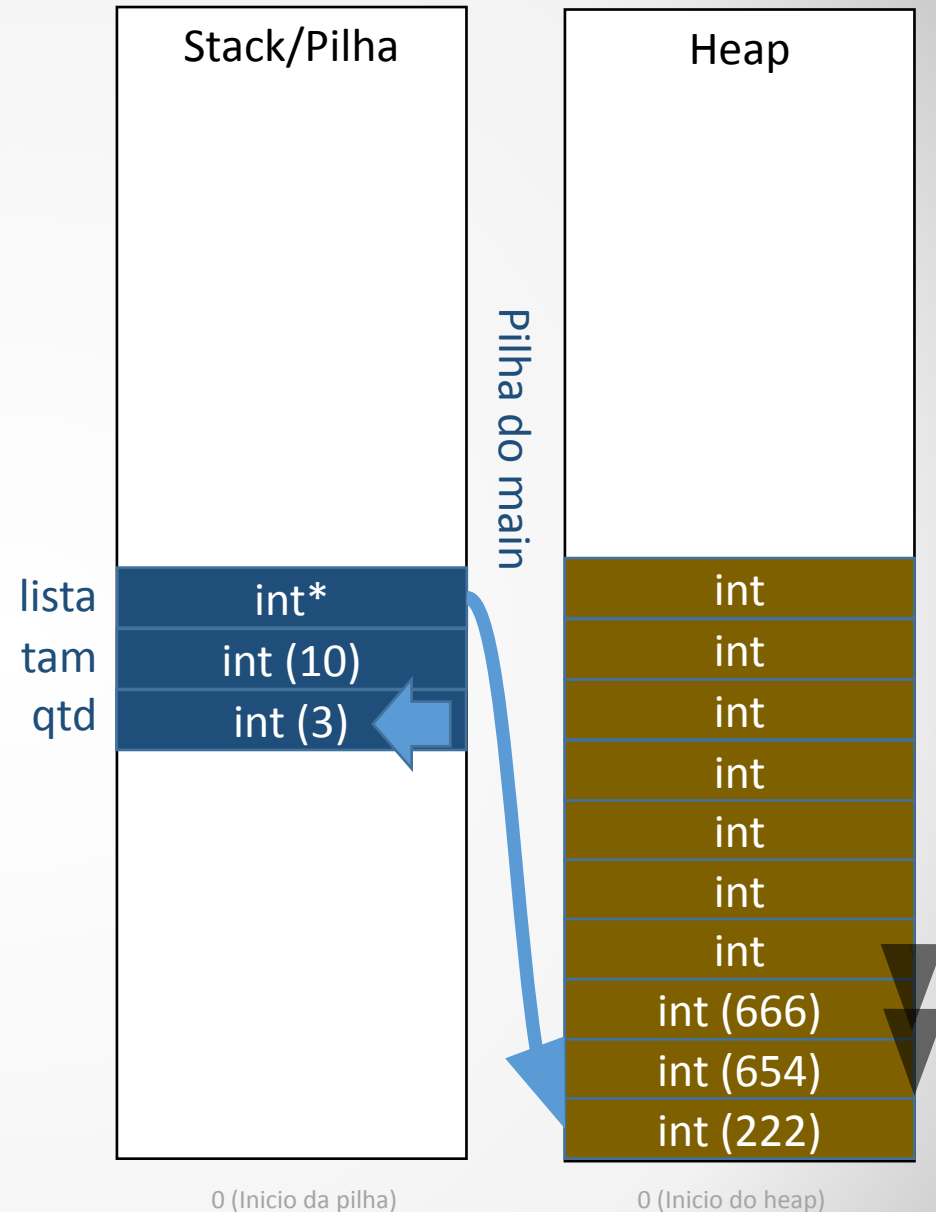
```
void main(){  
    int tam = 10;  
    int qtd = 0;  
    int *lista = criarListaS (tam);  
    ➔ // inserir 222, 555, 654, 666 no fim  
    ...  
}
```



Remover do meio

```
void main(){  
    int tam = 10;  
    int qtd = 0;  
    int *lista = criarListaS (tam);  
    // inserir 222, 555, 654, 666 no fim  
    ➔ // remover da 2a posição  
    ...  
}
```

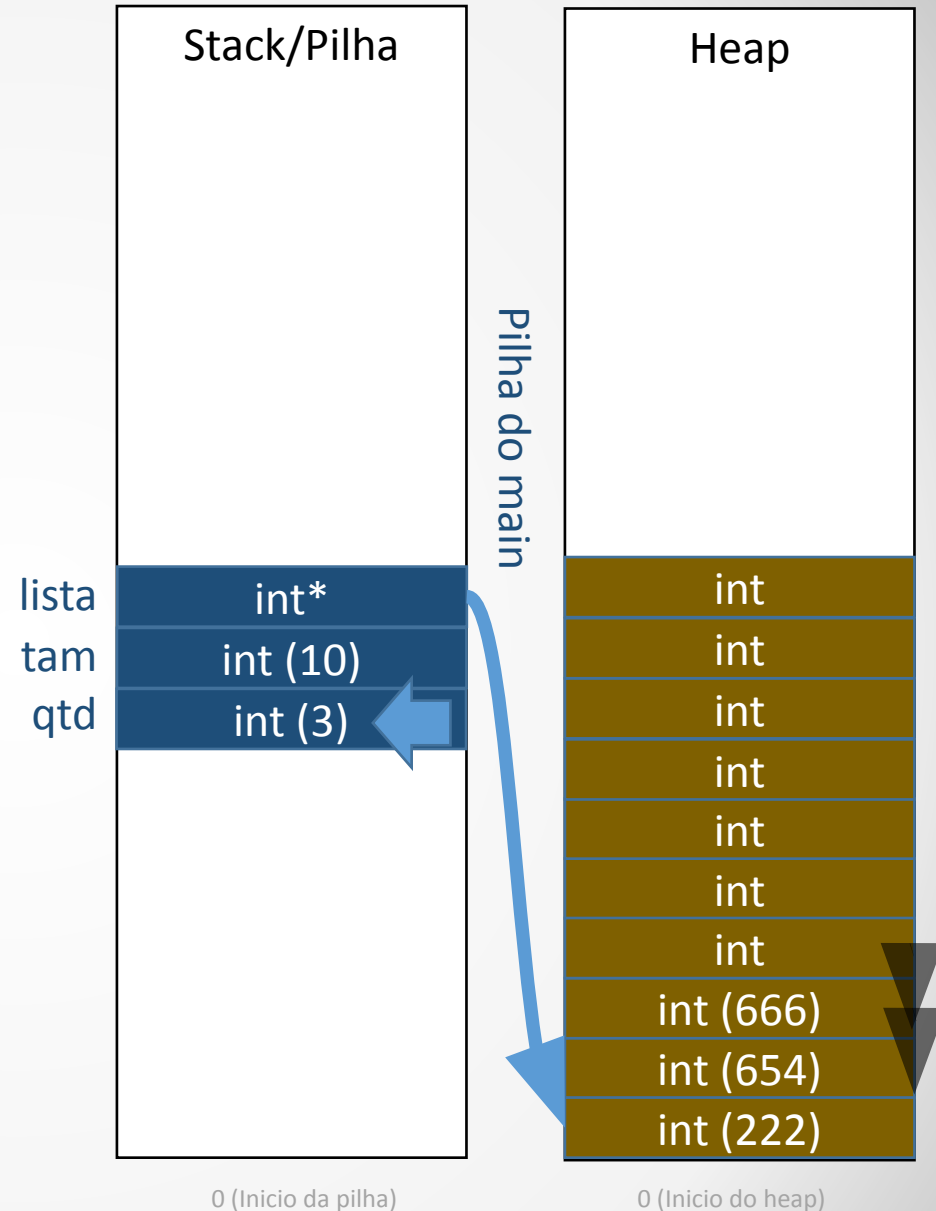
Remoção no meio requer o deslocamento dos elementos da lista subsequentes a esta posição.



Remover do meio

```
void main(){
    int tam = 10;
    int qtd = 0;
    int *lista = criarListaS (tam);
    // inserir 222, 555, 654, 666 no fim
    removerInicioS(lista, &qtd, 1);
    ...
}

void removerMeioS(int* ls, int& Q, int pos){
    if(ls==nullptr || Q==0) return;
    for(int i=pos; i<(Q-1); i++) ls[i] = ls[i+1];
    Q--;
    // não é necessário apagar o elemento ls[pos]
}
```



Remover

Remover na posição **pos**

```
void removerMeioS(int* ls, int& Q, int pos){  
    if(ls==nullptr || Q==0 || pos < Q) return;  
    for(int i=pos; i<(Q-1); i++) ls[i] = ls[i+1];  
    Q--;  
}
```

Linear

Remover do inicio

```
void removerInicioS(int* ls, int& Q){  
    if(ls==nullptr || *Q==0) return;  
    for(int i=0; i<(Q-1); i++) ls[i] = ls[i+1];  
    Q--;  
}
```

Linear

Remover do fim

```
void removerFimS(int* ls, int& Q){  
    if(ls==nullptr || Q==0) return;  
    Q--;  
}
```

Constante

Cabeça

- Estrutura de dados que agrega os dados descritivos de uma lista.

```
struct lista{  
    int tam;  
    int qtd;  
    int *dado;  
}
```

tam: int
qtd: int
dado: int*

Cabeça com alocação estática

```
struct lista{  
    int tam;  
    int qtd;  
    int* dado;  
};
```

```
int main(){  
    struct lista listaS;  
    criarListaSC(listaS, 7);  
    inserirInicioSC(listaS, 232);  
    inserirFimSC(listaS, 954);  
    inserirMeioSC(listaS, 311, 1);  
    destruirListaSC(listaS);  
    return 0;  
}
```

```
void criarListaSC(lista& ls, int tam){  
    ls.tam = tam;  
    ls.qtd = 0;  
    ls.dado = new int[tam];  
}
```

```
void destruirListaSC(lista& ls){  
    delete[] ls.dado;  
    ls.dado = nullptr;  
}
```

listaS

Stack/Pilha

tam: int (7)
qtd: int (3)
dado: int*

Heap

int

int

int

int

int(954)

int(311)

int(232)

0 (Início da pilha)

0 (Início do heap)

Fim do main

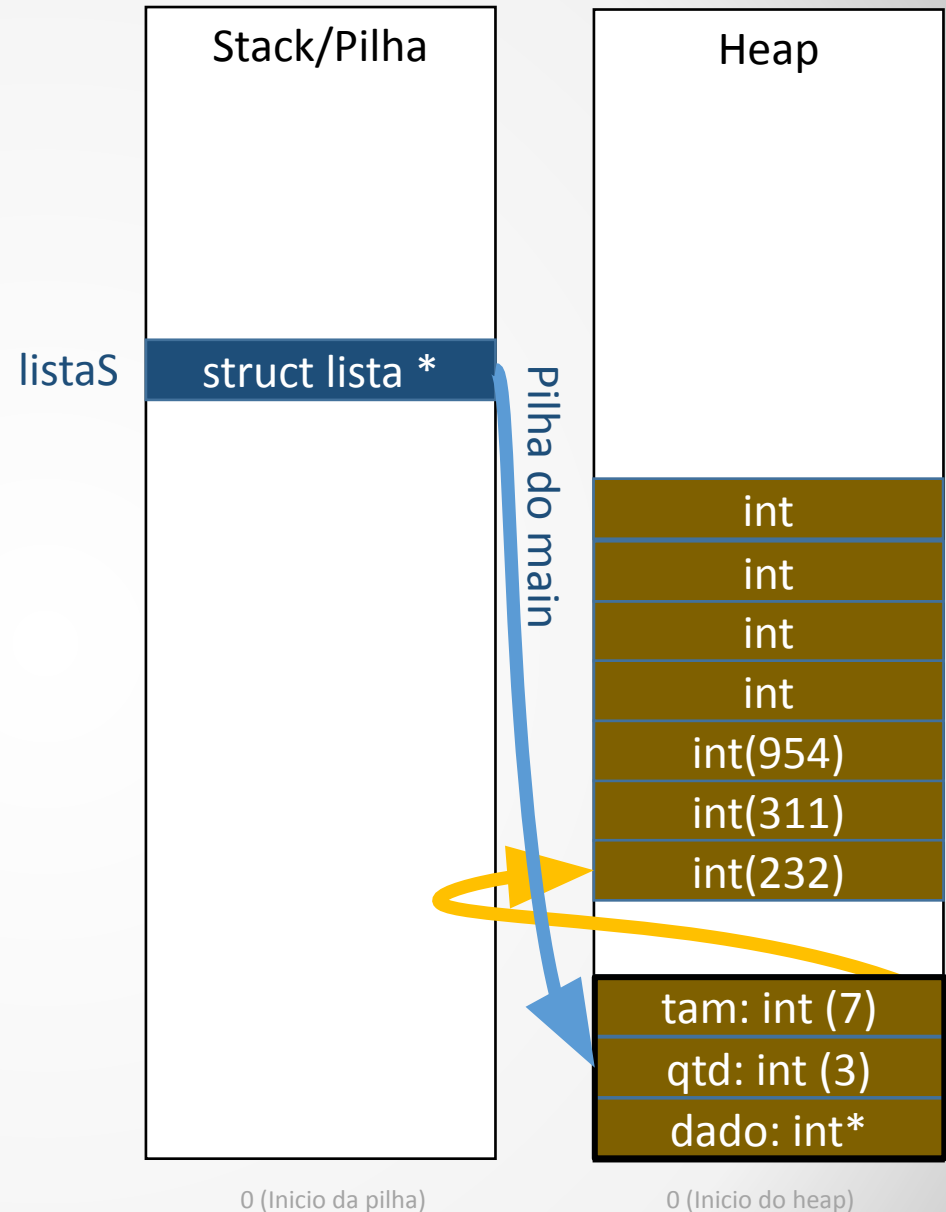
Cabeça com alocação dinâmica

```
struct lista{  
    int tam;  
    int qtd;  
    int* dado;  
};
```

```
int main(){  
    struct lista* listaS;  
    listaS = criarListaSC(7);  
    inserirInicioSC(listaS,232);  
    inserirFimSC(listaS,954);  
    inserirMeioSC(listaS,311,1);  
    destruirListaSC(listaS);  
    return 0;  
}
```

```
struct lista* criarListaSC(int tam){  
    struct lista* ls = new struct lista;  
    ls->tam = tam;  
    ls->qtd = 0;  
    ls->dado = new int[tam];  
    return ls;  
}
```

```
void destruirListaSC(lista* ls){  
    delete[] ls->dado;  
    delete ls; ls = nullptr;  
}
```



TAD Sequência

- Dados
 - Elementos em sequência
- Operações
 - Criar
 - Acessar (início, meio, fim)
 - Inserir (início, meio, fim)
 - Remover (início, meio, fim)
 - Destruir

Complexidade da
implementação
de lista sequencial

Linear
Constante

