Tópicos de Programação

Arthur Casals
(arthur.casals@usp.br)

IME - USP

Aula 4:

- Fundamentos de Estruturas de Dados

Na aula passada...

Estruturas de dados:

- Listas lineares (*arrays*)
- Listas ligadas
- Pilhas
- Filas
- Grafos
- Árvores

Na aula passada...

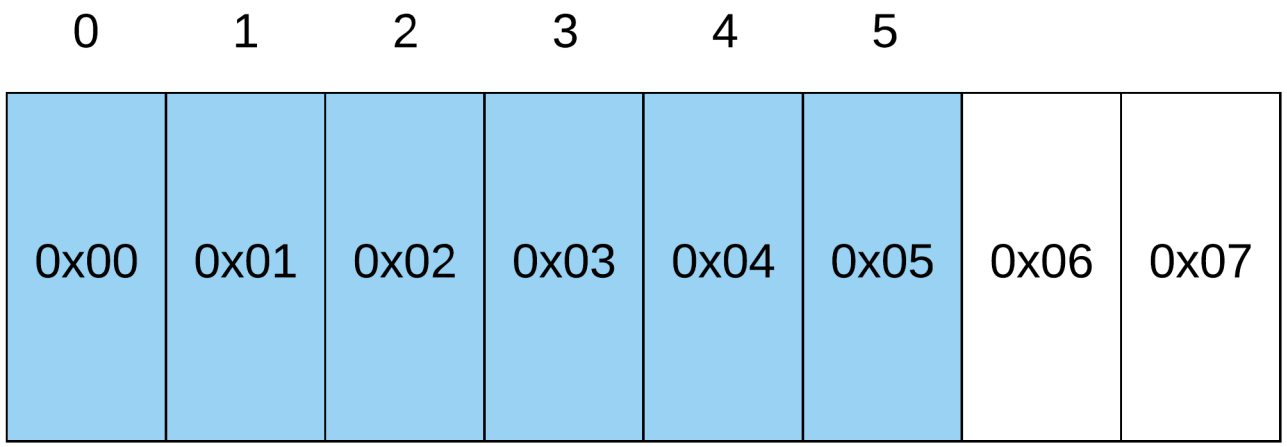
Listas lineares *(arrays):*

- Coleção de dados armazenados em espaços de memória *contínuos*
- Possui: *tamanho* e *início*
- Objetivo: agrupar dados do mesmo tipo
- Representação: espaços contínuos, numerados (*indexados*)

Na aula passada...

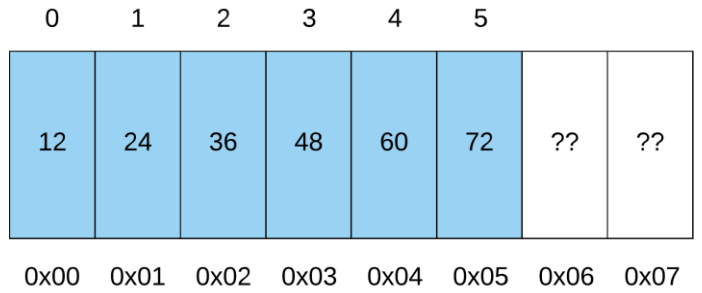
Listas lineares *(arrays)*:

› Exemplo: imagine uma lista com seis posições. Então:



Na aula passada...

Listas lineares *(arrays):*

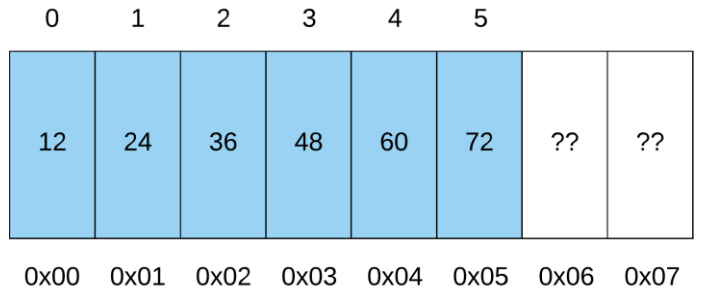


› Outras considerações:

- Ao ser alocada em memória, a lista ocupa um espaço aleatório
- A referência aos elementos da lista é feita através de posição na lista. ***Posição na lista não é posição em memória!***
- O tamanho de um *array* é sempre fixo
- Algumas operações podem custar caro!

Na aula passada...

Listas lineares *(arrays):*



› Vantagens:

- Acesso aos elementos via posicionamento é rápido
- Ao ocupar um espaço fixo na memória, maximiza eficiência de operações que envolvem *caching* (exemplo: multiplicação de matrizes)

Fundamentos de estruturas de dados

Listas lineares *(arrays):*

- Caso específico: *array* com duas dimensões
  - Uma lista linear bidimensional pode ser utilizada para representar uma matriz

Fundamentos de estruturas de dados

Listas lineares *(arrays):*

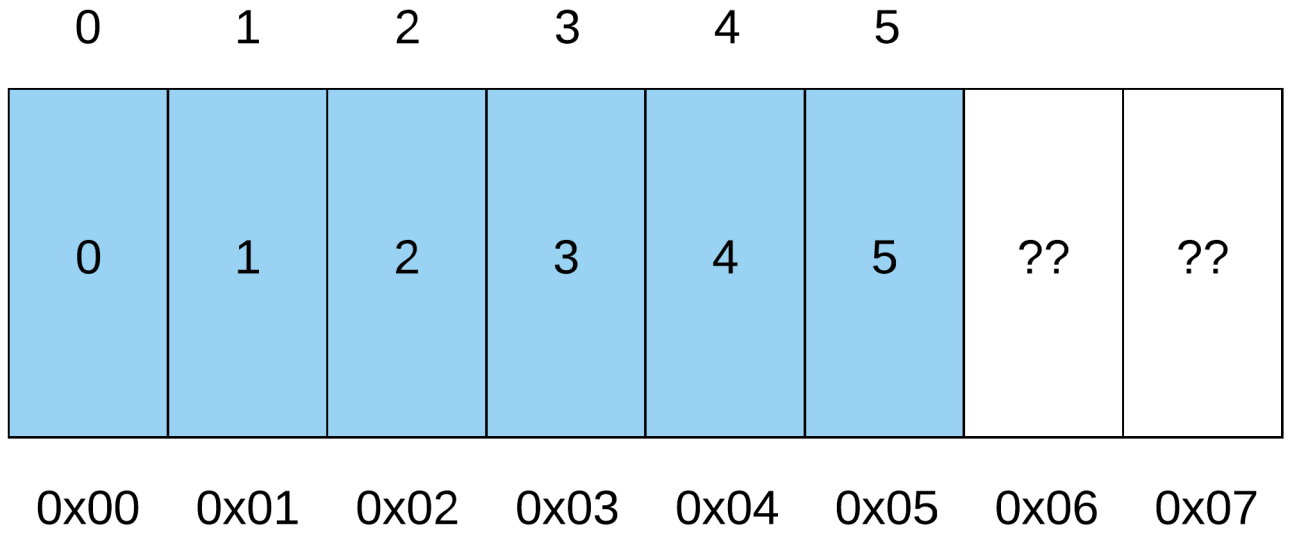
› Caso específico: *array* com duas dimensões

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} \Rightarrow \text{int matriz[3][2] = \{\{0, 1\}, \{2, 3\}, \{4, 5\}\}}$$

Fundamentos de estruturas de dados

Listas lineares *(arrays):*

› Caso específico: *array* com duas dimensões

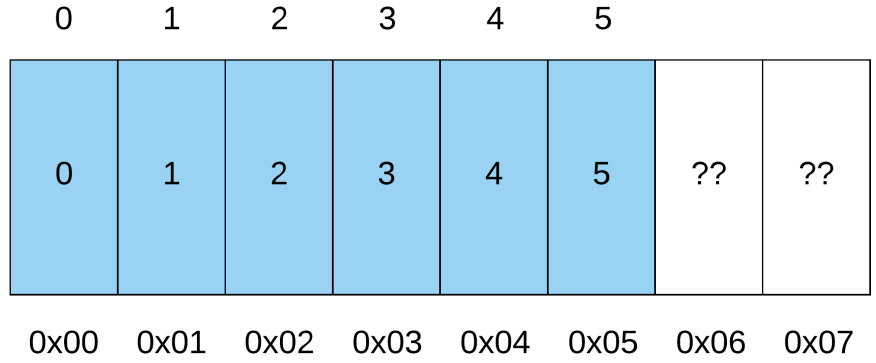


Fundamentos de estruturas de dados

Listas lineares *(arrays):*

- Exatamente igual a:

  int lista[6] = {0, 1, 2, 3, 4, 5}



- É responsabilidade do compilador saber a diferença!

Fundamentos de estruturas de dados

Listas lineares ligadas:

- Motivação:
  - *Arrays* possuem tamanho fixo (problemas de subdimensionamento ou superdimensionamento)
  - Algumas operações sobre *arrays* são computacionalmente caras

Fundamentos de estruturas de dados

Listas lineares ligadas:

- Vantagens sobre *arrays* :
  - Tamanho dinâmico (podem crescer ou diminuir de acordo com a necessidade)
  - Facilidade em operações de inclusão/exclusão de dados

Fundamentos de estruturas de dados

Listas lineares ligadas:

› Desvantagens:

- Acesso aleatório (via posição) não é permitido
- Cada elemento requer espaço extra para ponteiro
- *Caching*

Fundamentos de estruturas de dados

Listas lineares ligadas:

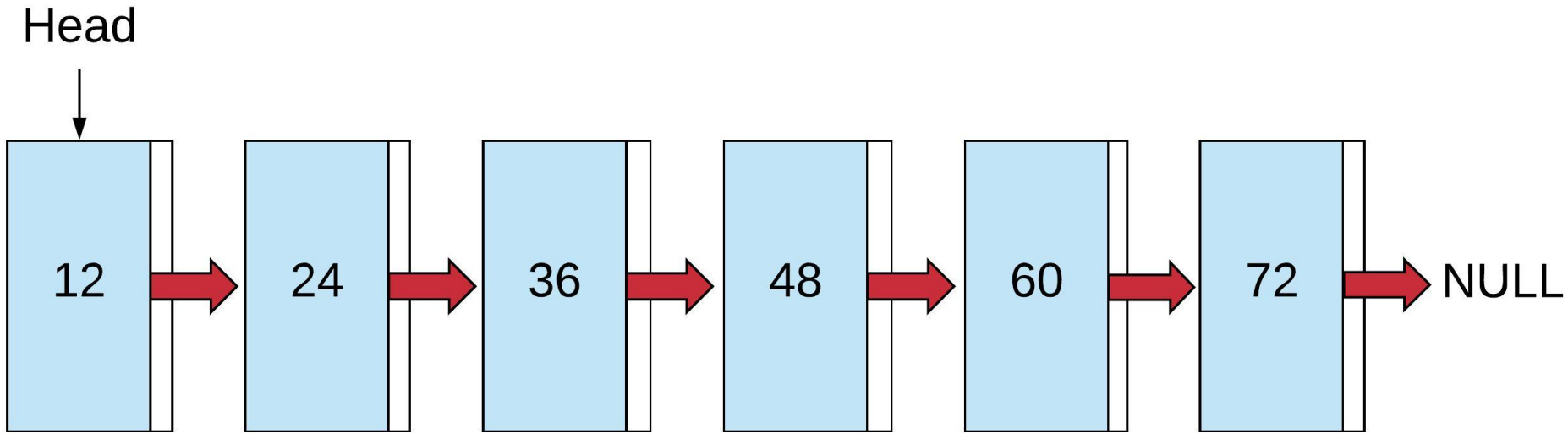
› Estrutura:

- Uma lista ligada é formada por nós
- Cada lista possui um nó inicial (*head*)
- Cada nó possui uma referência (apontador) para o próximo nó da lista

Fundamentos de estruturas de dados

Listas lineares ligadas:

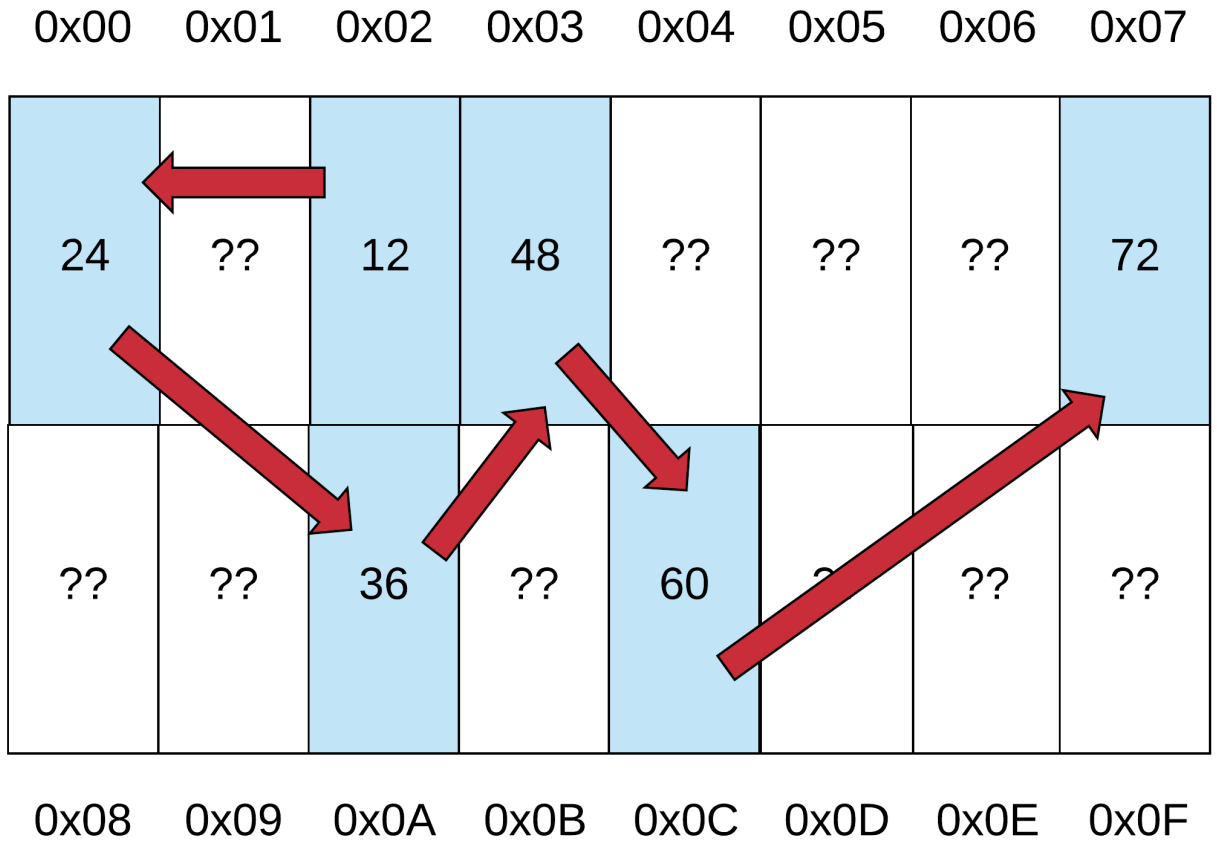
› Representação:



Fundamentos de estruturas de dados

Listas lineares ligadas:

› Em memória:



Fundamentos de estruturas de dados

Listas lineares ligadas:

› Representação em Java:

```
class Elemento
{
    int dado;
    Elemento proximo;
    Elemento(int i) {dado = d;}
}
```

Fundamentos de estruturas de dados

Listas lineares ligadas:

› Representação em C:

```
//Elemento (nó) de uma lista ligada
struct Elemento {
    int dado;
    struct Elemento *proximo;
};
```

Fundamentos de estruturas de dados

Construindo uma lista ligada em C:

```
int main() {
    struct Elemento* head = NULL;
    struct Elemento* segundo = NULL;
    struct Elemento* terceiro = NULL;
    head = (struct Elemento*)malloc(sizeof(struct Elemento)); //idem para segundo e terceiro
    head->dado = 1;
    head->proximo = segundo;
    segundo->dado = 2;
    segundo->proximo = terceiro;
    terceiro->dado = 3;
    terceiro->proximo = NULL;
    return 0;
}
```

Fundamentos de estruturas de dados

Construindo uma lista ligada em C:

```
struct Elemento* head = NULL;

struct Elemento* segundo = NULL;

struct Elemento* terceiro = NULL;
```

-> Três nós são inicializados na forma de ponteiros, com atribuição nula

Fundamentos de estruturas de dados

Construindo uma lista ligada em C:

```
head = (struct Elemento*)malloc(sizeof(struct Elemento)); //idem para segundo e terceiro
```

-> Para cada nó é alocado um espaço em memória correspondente ao tamanho da estrutura Elemento

Fundamentos de estruturas de dados

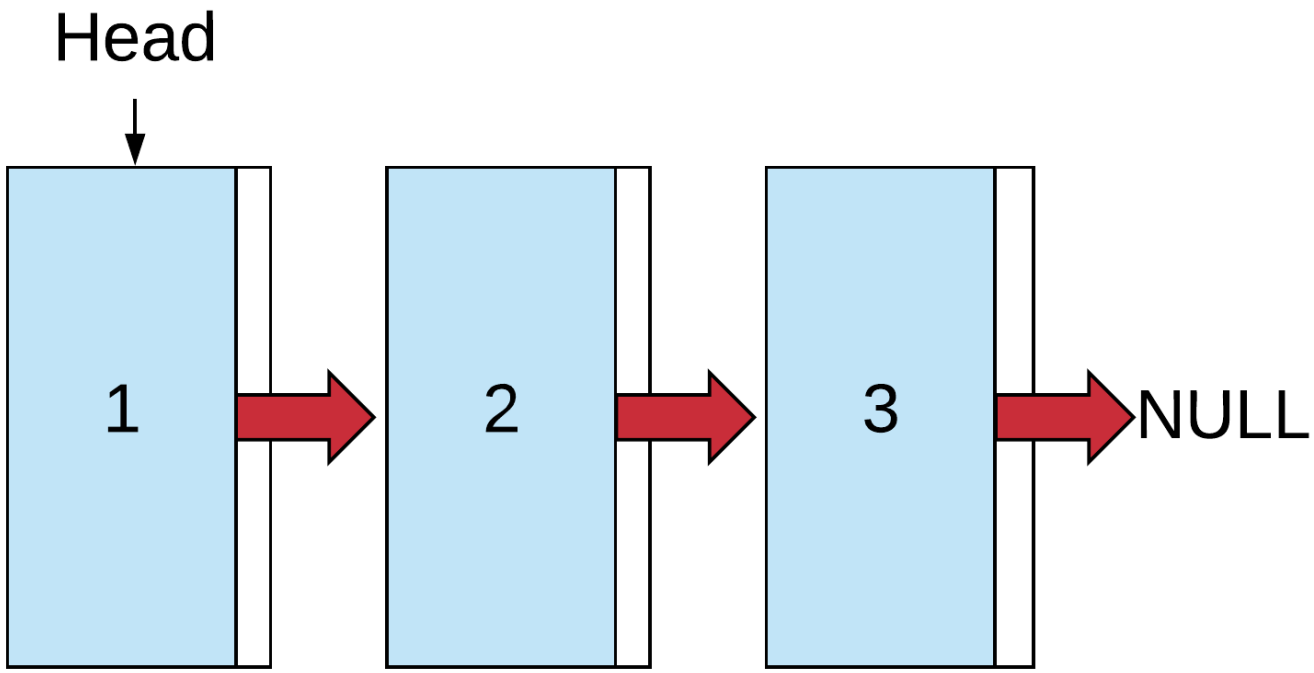
Construindo uma lista ligada em C:

```
head->dado = 1;
head->proximo = segundo;
segundo->dado = 2;
segundo->proximo = terceiro;
terceiro->dado = 3;
terceiro->proximo = NULL;
```

-> A cada nó é atribuído um valor ("dado") e uma referência para o próximo elemento ("proximo")

Fundamentos de estruturas de dados

Construindo uma lista ligada em C:



Fundamentos de estruturas de dados

Operações em listas ligadas:

› Inserção de elementos

› Remoção de elementos

› Tamanho

› Busca (verificar se um elemento pertence à lista)

› Troca de posições de nós (sem trocar os dados)

Fundamentos de estruturas de dados

Inserção de elementos:

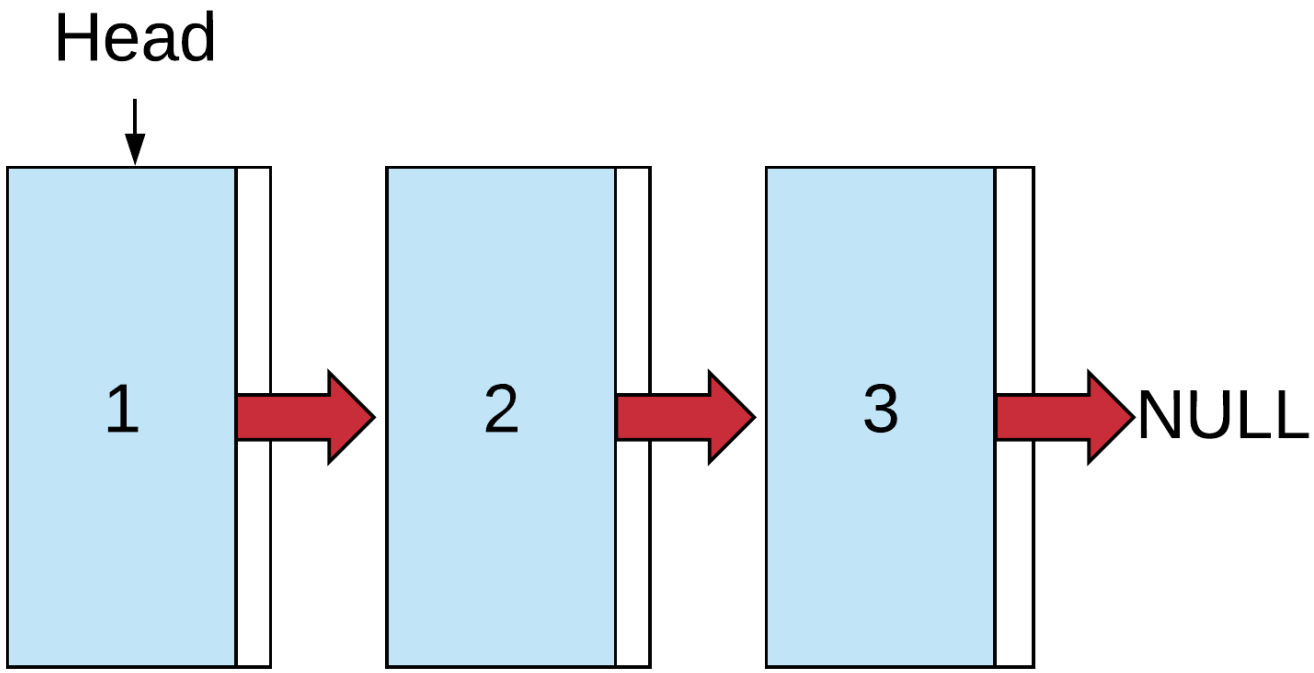
› No começo da lista

› Depois de um determinado nó

› Ao final da lista

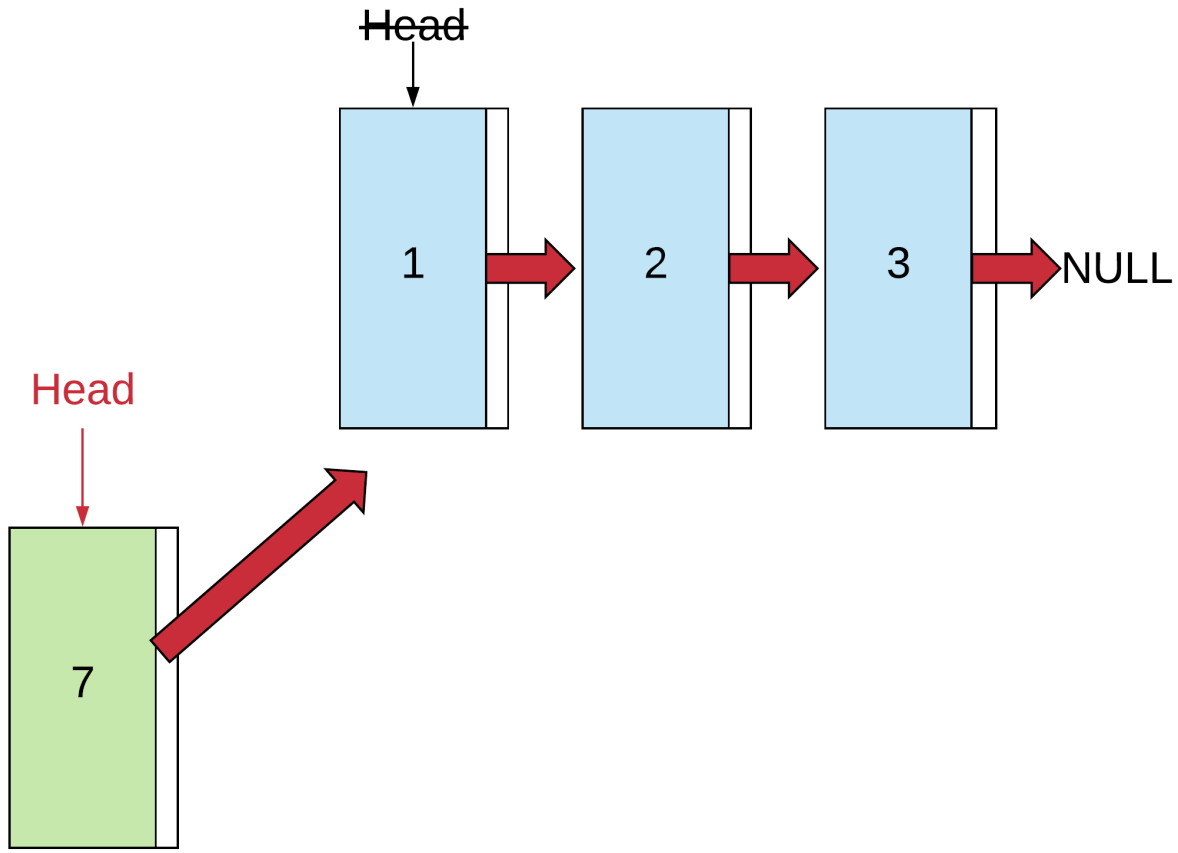
Fundamentos de estruturas de dados

Inserção de elementos no começo da lista:



Fundamentos de estruturas de dados

Inserção de elementos no começo da lista:



Fundamentos de estruturas de dados

```
void insere(struct Elemento** head_ref, int novo_dado)
{
    struct Elemento* no_novo = (struct Elemento*) malloc(sizeof(struct Elemento));
    no_novo->dado = novo_dado;
    no_novo->proximo = (*head_ref);
    (*head_ref) = no_novo;
}
```

Fundamentos de estruturas de dados

void insere(struct Elemento** head_ref, int novo_dado)

{

-> Parâmetros da função: *head* (referência para o começo da lista), dado para o novo elemento a ser inserido

Fundamentos de estruturas de dados

```
struct Elemento* no_novo = (struct Elemento*)
malloc(sizeof(struct Elemento));
```

-> Para cada novo nó, deve-se alocar o espaço em memória necessário previamente

Fundamentos de estruturas de dados

```
no_novo->dado = novo_dado;
no_novo->proximo = (*head_ref);
```

-> O novo nó recebe o dado necessário ("novo_dado") e aponta para o antigo *head* da lista

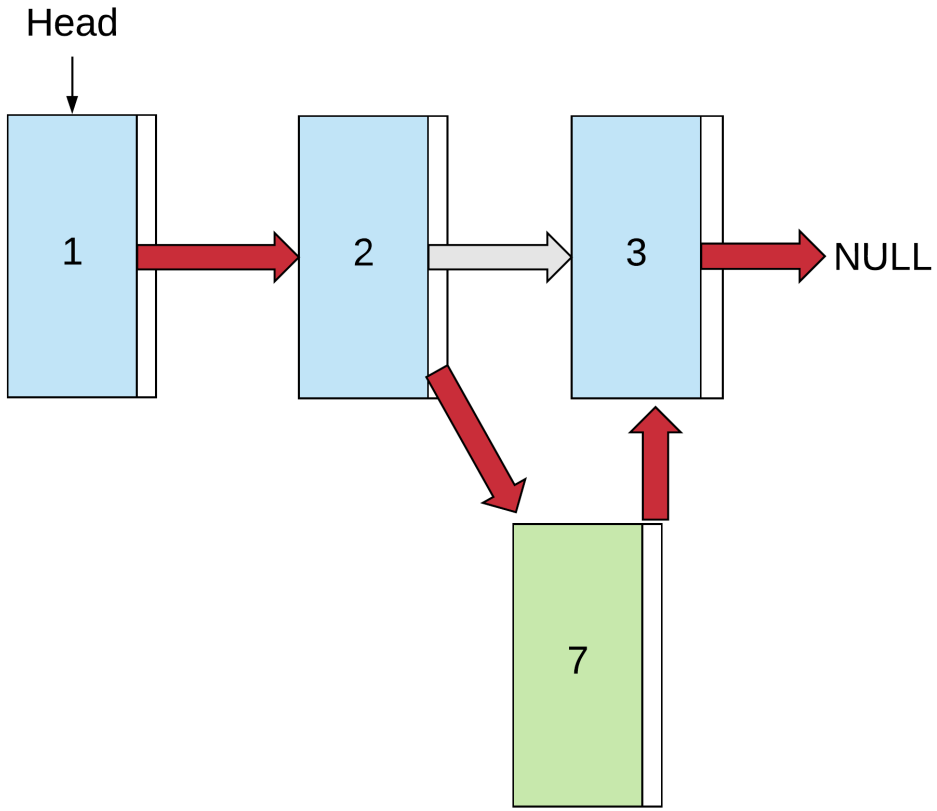
Fundamentos de estruturas de dados

```
(*head_ref) = no_novo;
```

-> A referência de início da lista é atualizada para apontar para o novo nó inserido

Fundamentos de estruturas de dados

Inserção de elementos depois de um determinado nó:



Fundamentos de estruturas de dados

```
void insereDepois(struct Elemento* no_anterior, int novo_dado)
{
    //checar se no_anterior é nulo
    struct Elemento* no_novo =(struct Elemento*) malloc(sizeof(struct Elemento));
    no_novo->dado   = novo_dado;
    no_novo->proximo = no_anterior->proximo
    no_anterior->proximo = no_novo;
}
```

Fundamentos de estruturas de dados

Inserção de elementos ao final da lista:

- Caso particular do exemplo anterior (inserção depois do último nó)
- Basta percorrer a lista até achar um nó que não aponta para ninguém (elemento->proximo == NULL)

Fundamentos de estruturas de dados

Remoção de elementos da lista:

- A partir de uma determinada posição
- A partir de um determinado nó (valor)

Fundamentos de estruturas de dados

Remoção de elementos da lista por valor/posição:

- Achar o nó anterior ao nó a ser removido;
- Modificar o ponteiro "próximo" do nó achado;
- Liberar a memória do nó removido

Fundamentos de estruturas de dados

```
void removerElementoPorValor(struct Elemento **head_ref, int valor) {
    struct Elemento* temp = *head_ref, *no_anterior;
    if (temp != NULL && temp->dado == valor) {
        *head_ref = temp->proximo;     //  Modifica head
        free(temp);                    //  Libera memória
        return;
    }
    while (temp != NULL && temp->dado != valor) {
        no_anterior = temp;
        temp = temp->proximo;
    }
    if (temp == NULL)
        return;
    no_anterior->proximo = temp->proximo; //  Remove o nó da lista
    free(temp);   //  Libera memória
}
```

Fundamentos de estruturas de dados

```
void removerElementoPorPosicao(struct Elemento **head_ref, int posicao) {
    if (*head_ref == NULL)
        return; //Se a referência inicial for nula, sai da função
    struct Elemento* temp = *head_ref;
    if (posicao == 0) {
        *head_ref = temp->proximo;    // Modifica head
        free(temp);                   // Libera memória
        return;
    }
    for (int i=0; temp!=NULL && i<position-1; i++) //Acha o nó anterior ao que deve ser removido
        temp = temp->proximo;
    if (temp == NULL || temp->proximo == NULL) //Se a posição desejada exceder o tamanho da lista, finaliza
        return;
    struct Elemento *prox_no = temp->proximo->proximo; //temp->proximo é o nó a ser removido
    free(temp->proximo);  // Libera memoria
    temp->proximo = prox_no;  // Remove o nó da lista
}
```

Fundamentos de estruturas de dados

Tamanho de uma lista:

- Por iteração
- Por recursão

Fundamentos de estruturas de dados

Tamanho de uma lista (método iterativo):

- Inicializar um contador em 0;
- Inicializar um ponteiro para nó, ("atual") inicialmente apontando para o elemento inicial (*head*);
- Enquanto "atual" for diferente de nulo:
  - Atualizar "atual" com o ponteiro para o próximo nó
  - Incrementar contador
- Retornar valor do contador

Fundamentos de estruturas de dados

```
int acharTamanhoIterativo(struct Elemento* head) {
    int contador = 0;   //  Inicializa Contador
    struct Elemento* atual = head;   //  Inicializa "atual"
    while (atual != NULL) {
        contador ++;
        atual = atual->proximo;
    }
    return contador;
}
```

Fundamentos de estruturas de dados

Tamanho de uma lista (método recursivo):

- Entrada: referência para *head*

- Se *head* for nulo, retornar 0;

- Senão, retornar 1 + tamanho de *head*->proximo

Fundamentos de estruturas de dados

```
int acharTamanhoRecursivo(struct Elemento* head)
{
        if (head == NULL) //caso inicial
                return 0;
        return 1 + acharTamanhoRecursivo(head->proximo);
}
```

Fundamentos de estruturas de dados

Busca em uma lista:

- Por iteração
- Por recursão

Fundamentos de estruturas de dados

Busca em uma lista (método iterativo):

- Inicializar um ponteiro para nó, ("atual") inicialmente apontando para o elemento inicial (*head*);
- Enquanto "atual" for diferente de nulo:
  - Se atual->dado for igual ao nó buscado, retornar Verdadeiro
  - Atualizar "atual" com o ponteiro para o próximo nó
- Retornar Falso

Fundamentos de estruturas de dados

```
bool buscalterativa(struct Elemento* head, int x) {
        struct Elemento* atual = head;   //  Initializa atual
        while (atual != NULL) {
                if (atual->dado == x)
                        return true;
                atual = atual->proximo;
        }
        return false;
}
```

Fundamentos de estruturas de dados

Busca em uma lista (método recursivo):

- Entrada:s referência para *head*, valor buscado (x)
- Se *head* for nulo, retornar Falso;
- Se o dado de *head* for igual a **x**, retornar Verdadeiro;
- Senão, retornar busca de *head*->proximo

Fundamentos de estruturas de dados

```
bool buscaRecursiva(struct Elemento* head, int x)
{
        if (head == NULL)
                return false;
        if (head->dado == x)
                return true;
        return buscaRecursiva(head->proximo, x);
}
```

Fundamentos de estruturas de dados

EXERCÍCIO: trocar nós em uma lista (sem trocar dados):

- Exemplo de lista original: 1->2->3->4->5->6
- Exemplo de entradas: *head, (apontando para o nó com valor igual a 1), x = 2, y = 5
- Exemplo de resultado: 1->5->3->4->2->6
- Sem trocar dados significa: **não pode** existir no algoritmo nada parecido com *elemento->dado = valor*
- Façam o algoritmo para uma lista qualquer