AED - Algoritmos e Estruturas de Dados

Aula 8 – Listas Sequenciais

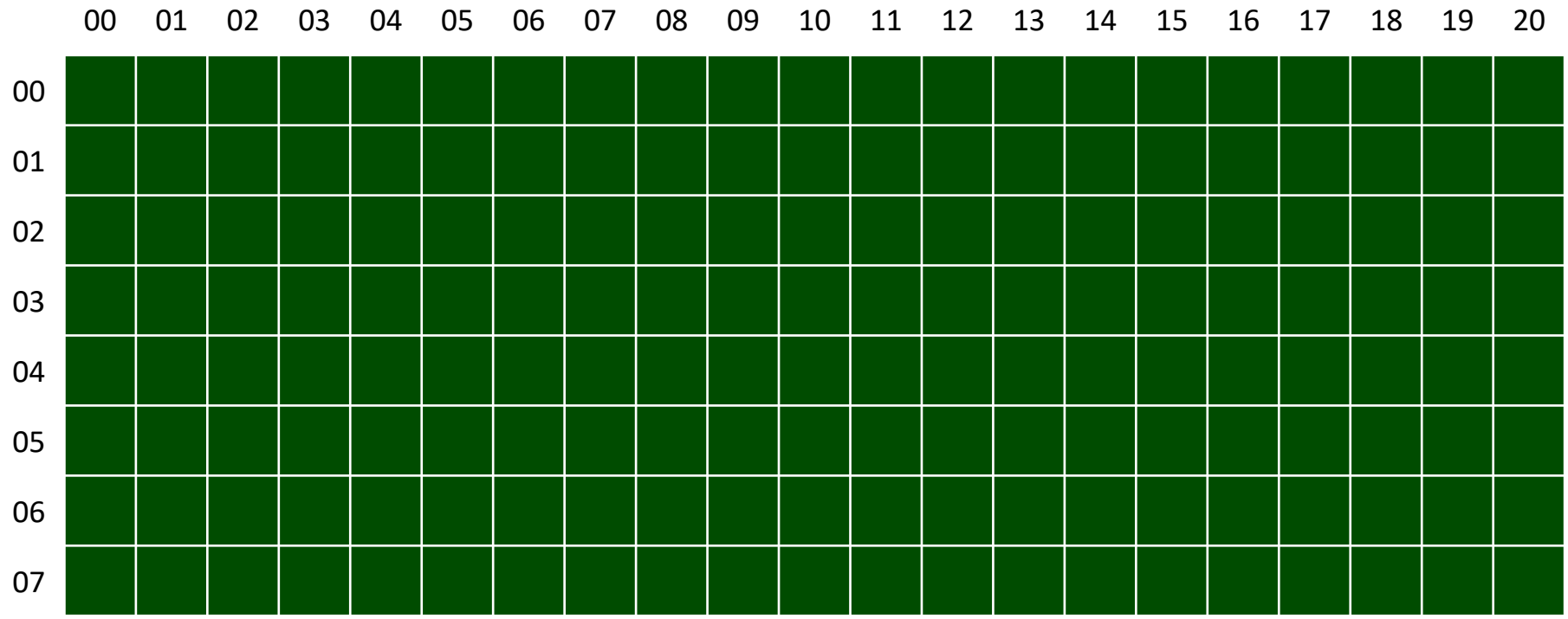
Prof. Rodrigo Mafort

Alocação Sequencial

- Ao definir uma lista (vetor) nas linguagens de programação temos duas opções:
  - Alocação sequencial:
    - Todo o espaço é alocado de uma única vez
    - Os elementos da lista são alocados de forma contígua (uma posição ao lado da outra).
    - Vantagem: Podemos acessar diretamente qualquer elemento da lista
    - Desvantagem: O espaço alocado não pode ser modificado (a lista tem 50 posições, não é possível estender para incluir 51 elementos)
  - Alocação dinâmica:
    - Alocação realizada de acordo com a necessidade
    - Cada elemento fica alocado em uma posição diferente da memória.
    - Não é possível acessar diretamente um elemento qualquer da lista. Apenas o primeiro.

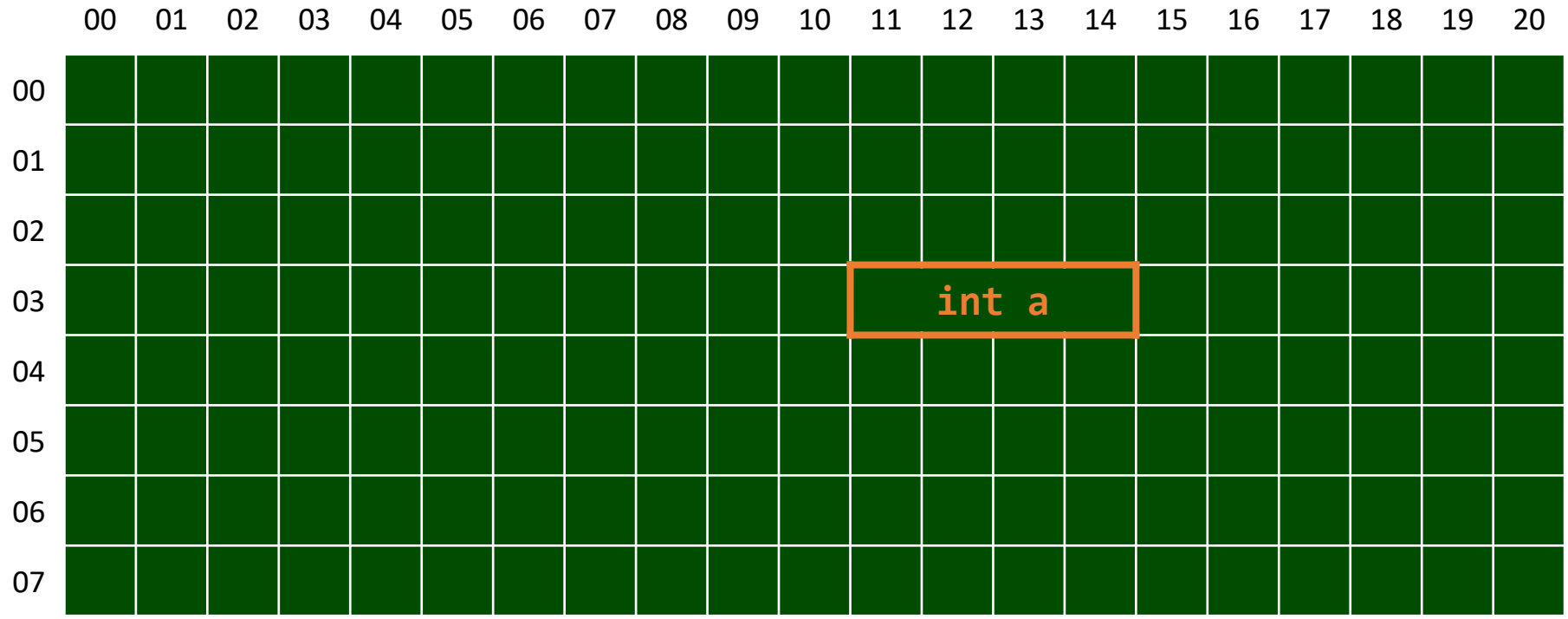
Alocação Sequencial

Suponha que podemos visualizar um pedaço da memória principal.
Esse corte apresenta os endereços de 0000 até 0720



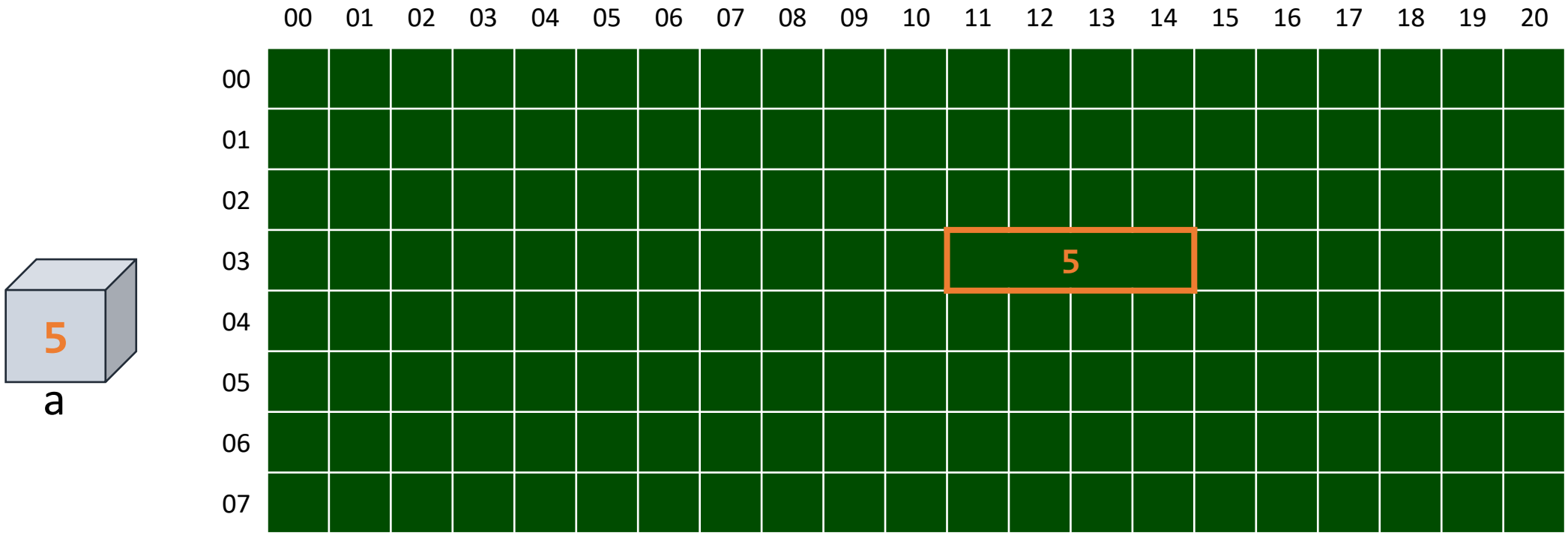
Alocação Sequencial

- Ao definir uma variável (por exemplo: `int a;`) alocamos um espaço de memória proporcional ao tipo e associamos o endereço ao nome.



Alocação Sequencial

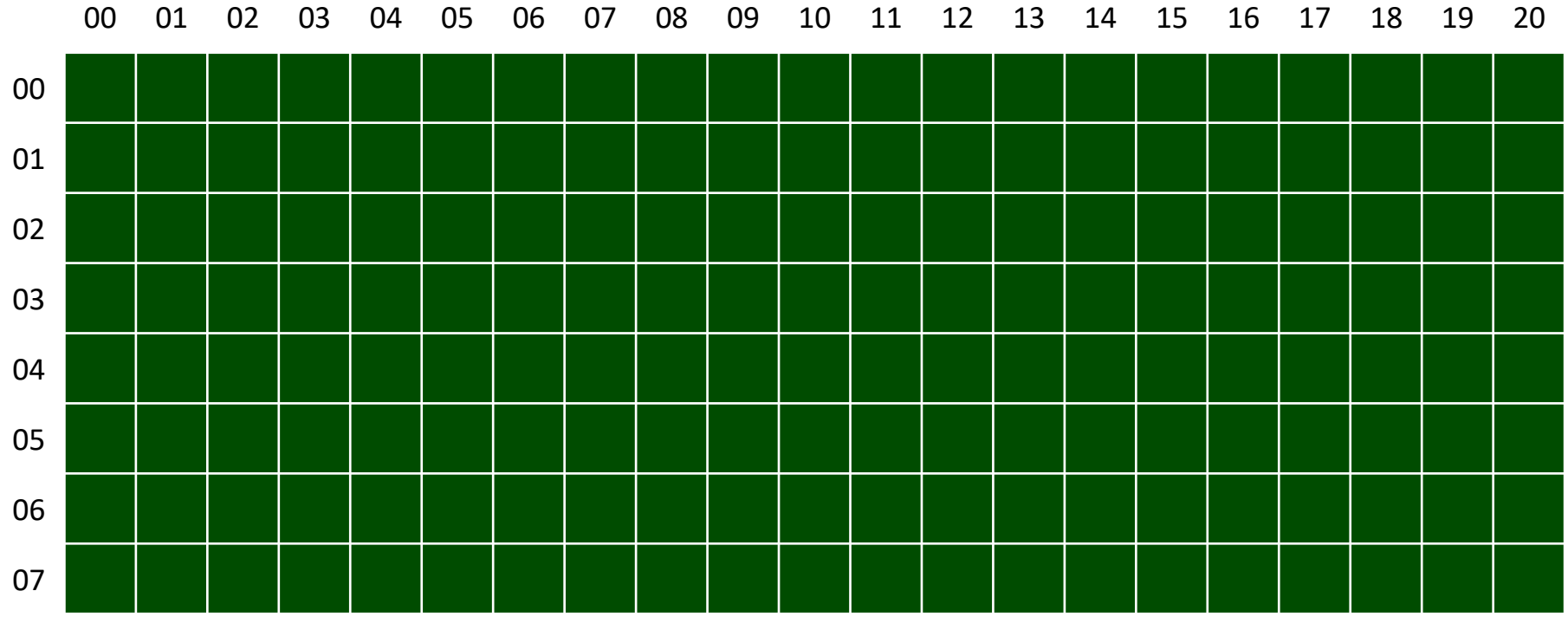
- Qualquer acesso a variável, implica em um acesso a esse endereço da memória: `a = 5;`



- `&a = 0311`

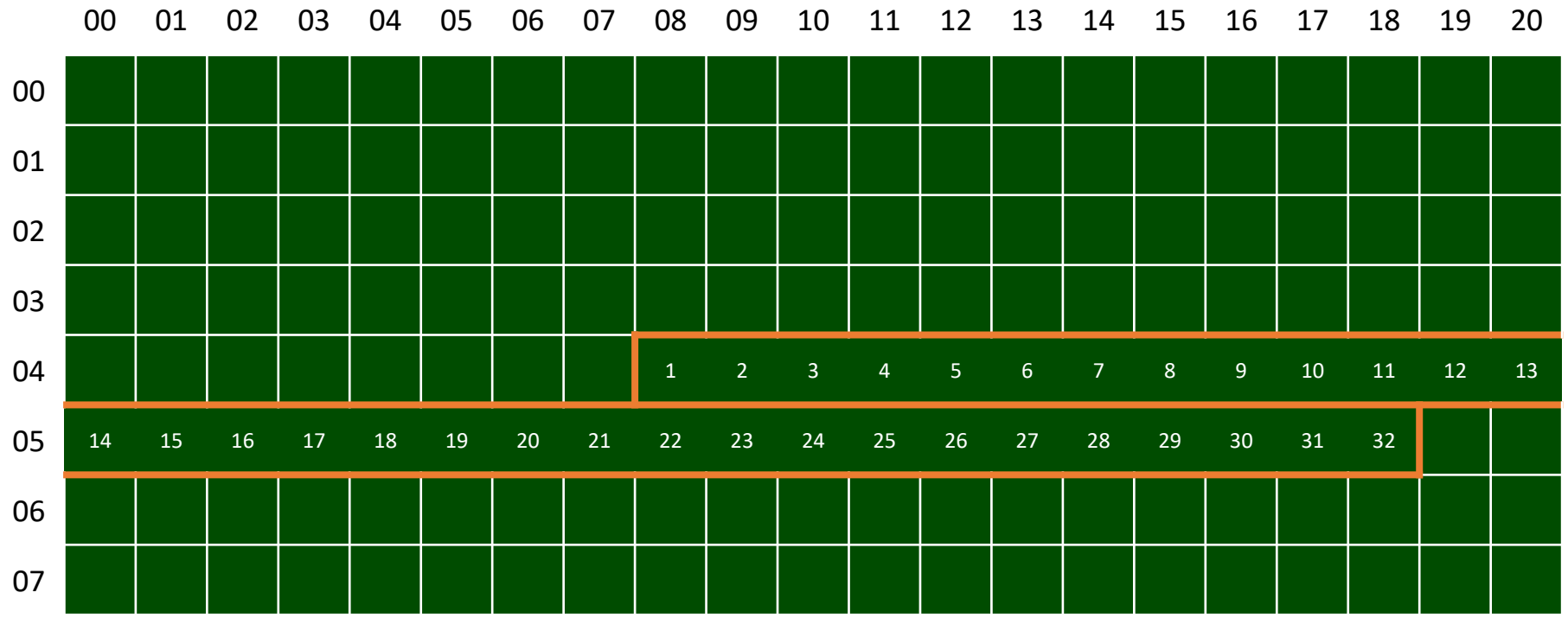
Alocação Sequencial

Quando definimos um vetor, precisamos alocar uma quantidade suficiente de memória para comportar o número de “células” que definimos. Lembrando que cada célula requer um número previamente determinado de bytes.
Por exemplo: `int v[8]` requer 8 * 4 = 32 bytes de memória.



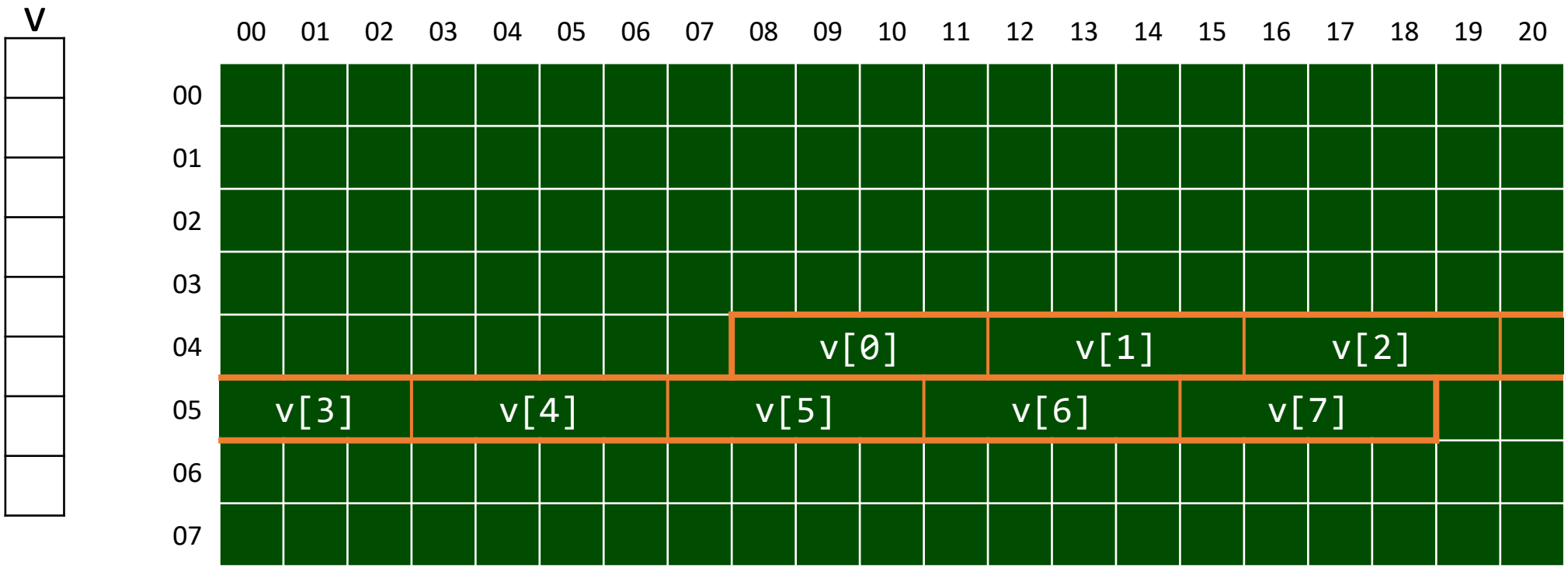
Alocação Sequencial

Entretanto, diferente de variáveis isoladas, esse espaço alocado deve ser contiguo. O vetor é alocado como um único bloco na memória.
A instrução `int v[8];` implica em alocar **um único bloco** com 32 bytes de memória.



Alocação Sequencial

Entretanto, diferente de variáveis isoladas, esse espaço alocado deve ser contiguo. O vetor é alocado como um único bloco na memória.
A instrução `int v[8];` implica em alocar **um único bloco** com 32 bytes de memória.

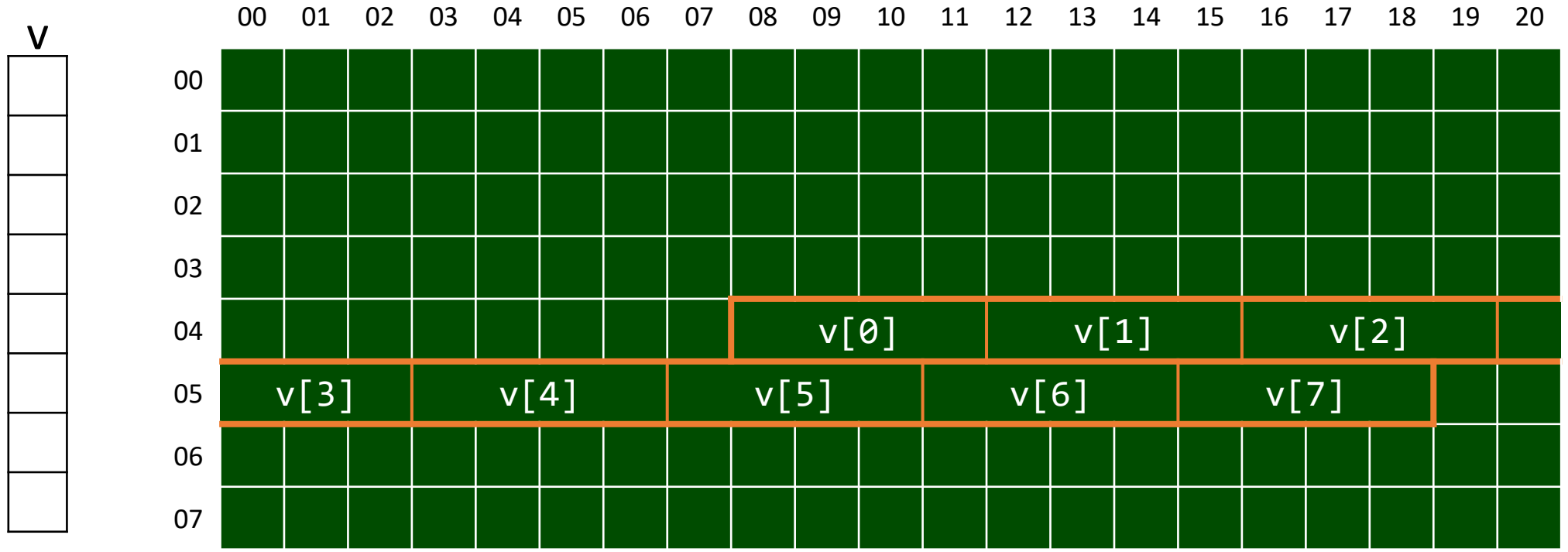


`&v = 0408`

Alocação Sequencial

Conhecendo o endereço inicial de v (0804), podemos acessar todas as posições.

```
v[i] = endereço inicial + i * 4 (bytes do tipo int)
v[0] = 0408 + 0; v[1] = 0408 + 4; ...; v[7] = 0408 + 28 = 0515
```

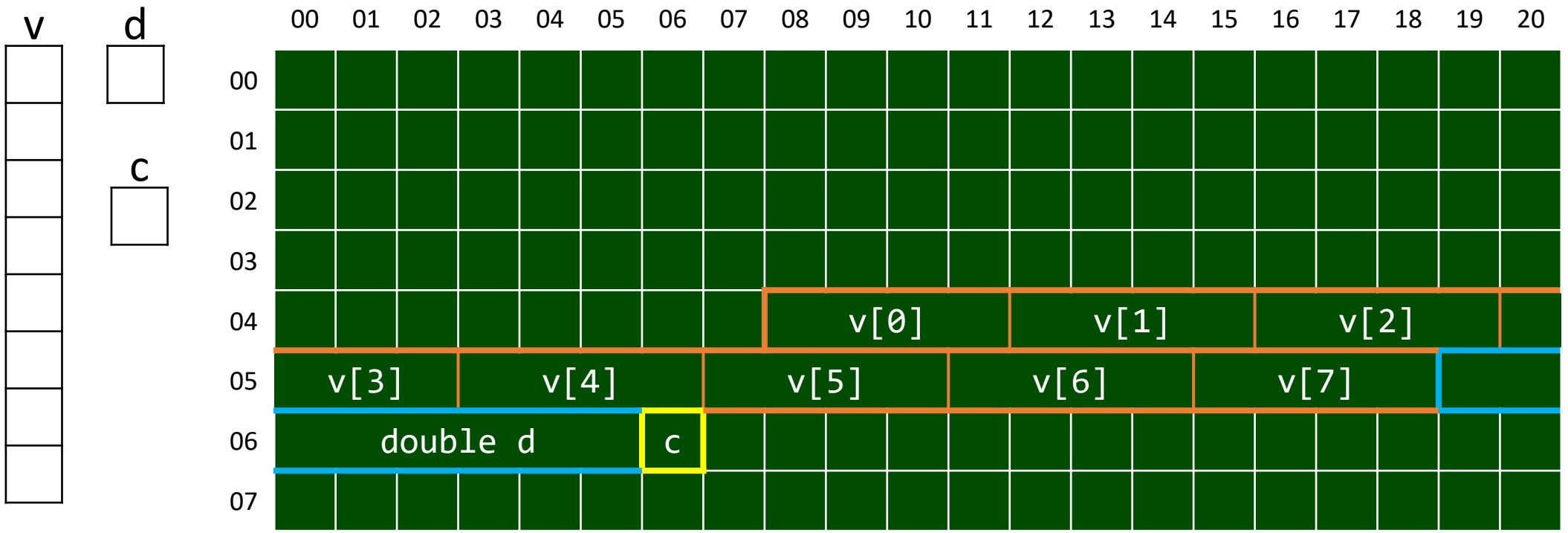


Alocação Sequencial

- Por que o tamanho de um vetor é fixo?
  - Porque ele o tamanho do bloco é definido no momento da alocação.
- Por que não podemos aumentar o tamanho?
  - Ao alocar mais espaço, não é possível assegurar que as novas posições serão contiguas com as primeiras posições do vetor.
- Como algumas linguagem permitem aumentar o tamanho das listas (list em Python ou vector em C++)?
  - Mentindo... Elas não aumentam.
  - Elas criam outra lista e copiam todo o conteúdo para a nova lista
  - E isso tem um custo computacional que deve ser considerado

Alocação Sequencial

Não é possível alocar mais endereços após o final do vetor.
Novas posições estarão dispersas na memória



Vale observar: v[8] deveria estar “armazenado” nos endereços 0518 até 0601
Ao tentar acessar v[8], estamos acessando parte do conteúdo da variável d (double).

Listas de Alocação Sequencial

- As listas constituem a estrutura de dados mais simples de que dispomos.

- Os dados são armazenados como elementos de um vetor.

- Precisamos verificar alguns aspectos:
  - Como definir a lista (o vetor)
  - Como verificar o número de elementos contidos na lista
  - Como inserir um novo elemento na lista
  - Como buscar por um determinado elemento na lista
  - Como remover um elemento na lista
  - Como alterar um elemento da lista

Exemplo de Lista

Lista L

ID:	ID:	ID:	ID:	ID:
Qtde:	Qtde:	Qtde:	Qtde:	Qtde:
Valor:	Valor:	Valor:	Valor:	Valor:

- Como definir a lista L?

Definir a lista L

```
typedef struct {
    int ID;
    float qtde;
    double valor;
} elemento;
```

Definição da estrutura utilizada para cada célula da lista.

```
elemento L[5];
```

Alocação estática da lista L

Observe que a lista terá no máximo 5 elementos.

Definir a lista L

```
#define TAM 5

typedef struct {
     int ID;
     float qtde;
     double valor;
} elemento;


elemento L[TAM];
```

Definição de uma **constante** que especifica o tamanho máximo da lista

Definição da estrutura utilizada para cada célula da lista.

Alocação estática da lista L

Observe que a lista terá no máximo TAM elementos.

Curiosidade: Alocação Estática

- Como permitir que o tamanho da lista seja determinado pelo usuário (o número máximo de elementos)?

- Vale lembrar que esse código não é permitido...

```
int T;
scanf("%d",&T);
elemento L[T];
```

Comando malloc

- A linguagem C possui um comando que permite alocar manualmente a memória associada a uma variável (uma lista também é uma variável).

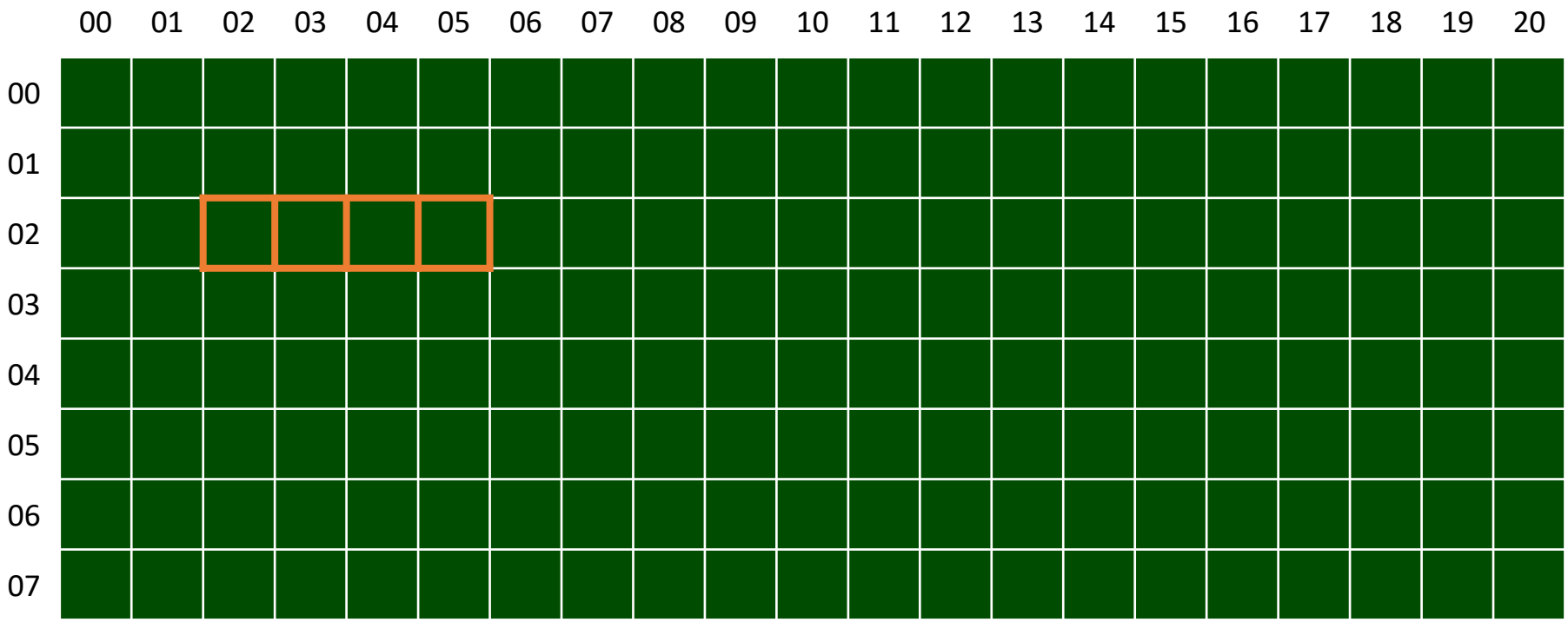
- O comando **malloc** recebe apenas um único parâmetro:
  - Quantos bytes de memória você quer alocar?

- E devolve uma única informação:
  - O **primeiro** endereço do espaço de memória que ele alocou.
  - Como lidar com o endereço? **Com um ponteiro**

Ao executar `(int*) malloc(4)`

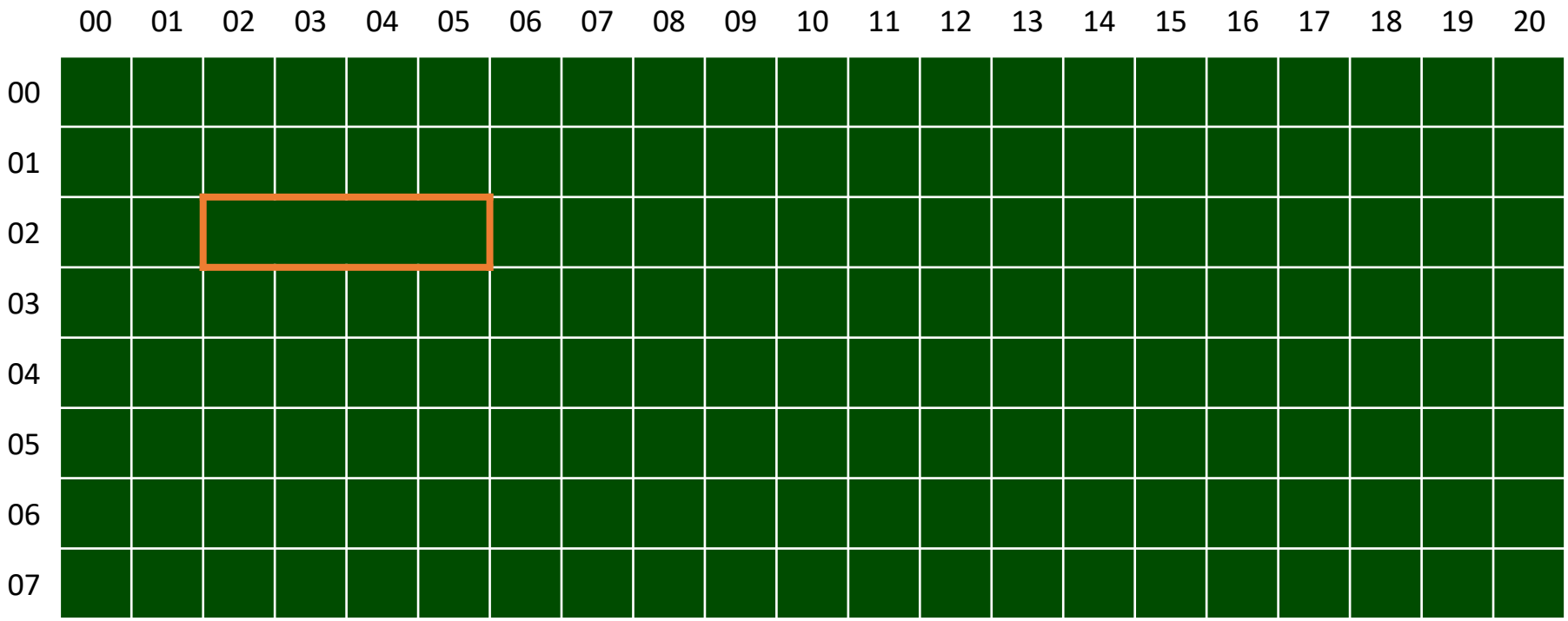
1. Reserve 4 bytes de memória (pedido ao sistema operacional)

Resultado: Primeiro endereço reservado - 0202



Ao executar `(int*) malloc(4)`

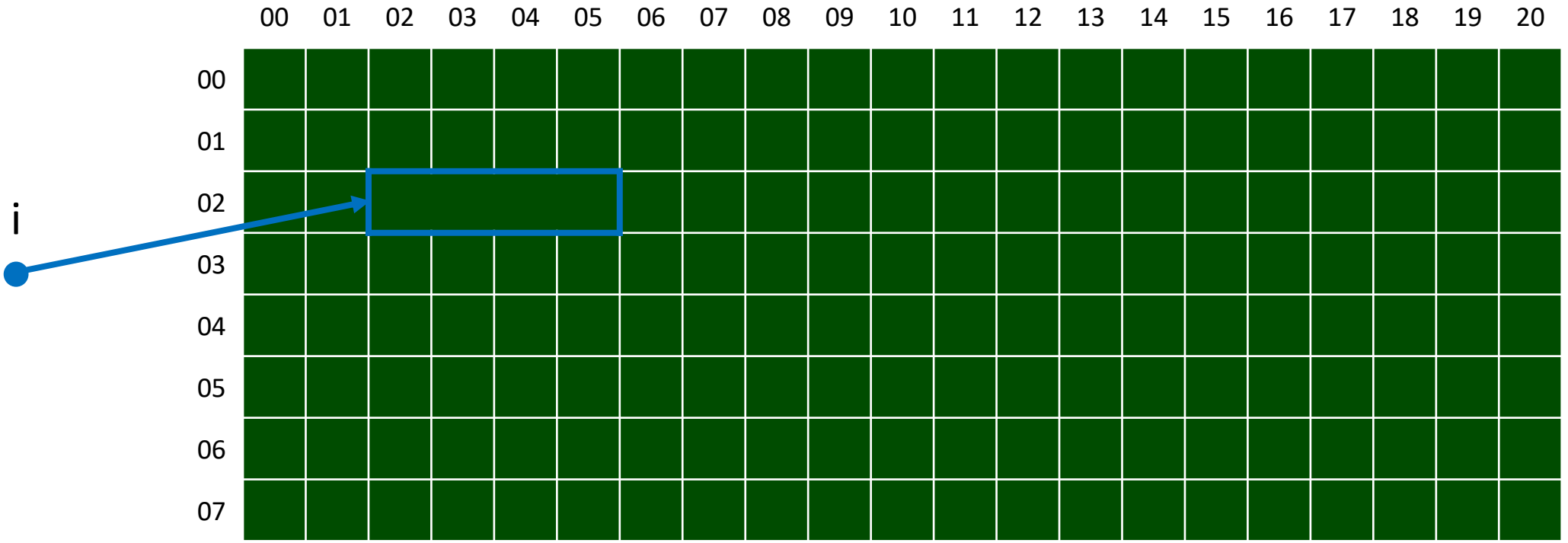
2. Converte esse endereço para um ponteiro para inteiros
   Lembrar: ponteiros para um determinado tipo “enxergam” o número de bytes necessários para esse tipo.



Ao executar `(int*) malloc(4)`

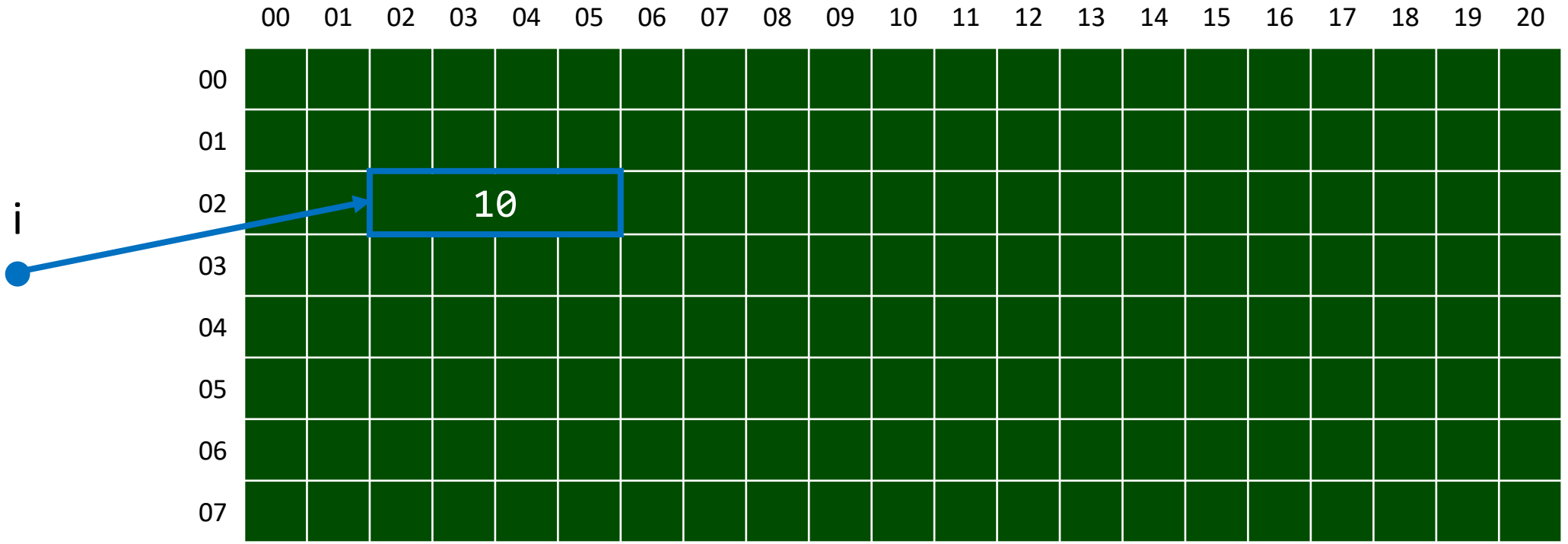
3. Armazene esse ponteiro para um endereço em uma variável do tipo ponteiro.

```
int* i = (int*) malloc (4);
```



Ao executar (int*) malloc(4)

```
int* i = (int*) malloc (4);
*i = 10;
```



Exemplo:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *a = (int*) malloc ( 4 );
    scanf("%d",a);
    *a = *a * *a;
    printf("%d",*a);
    return 0;
}
```

A função malloc está definida na biblioteca stdlib.

1. Aloque 4 bytes na memória (int = 4 bytes)
2. Converta o ponteiro retornado para um ponteiro para inteiros (int*)
3. Guarde esse endereço no ponteiro a

Função sizeof

- Para não precisarmos decorar ou calcular o tamanho requerido por cada tipo de dados (inclusive struct), podemos usar o comando sizeof.

- O comando recebe o nome de um determinado tipo e retorna quantos bytes na memória esse tipo requer.

- `sizeof(char) = 1`
- `sizeof(int) = 4`
- `sizeof(float) = 4`
- `sizeof(double) = 8`
- `sizeof(elemento) = 16 (int + float + double)`

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *a = (int*) malloc (sizeof(int));
    scanf("%d",a);
    *a = *a * *a;
    printf("%d",*a);
    return 0;
}
```

A função malloc está definida na biblioteca stdlib.

1. **Aloque o número de bytes que o tipo int requer.**
2. Converta o ponteiro retornado para um ponteiro para inteiros (int*)
3. Guarde esse endereço no ponteiro a

Comando malloc

- Como permitir que o tamanho da lista seja determinado pelo usuário (o número máximo de elementos)?

- Com o malloc podemos fazer isso:

```
int T;
scanf("%d",&T);
elemento* L;
L = (elemento*) malloc(sizeof(elemento) * T);
```

Exemplo malloc

```
int main()
{
    int T;
    scanf("%d",&T);

    elemento* L = (elemento*) malloc (sizeof(elemento) * T);
    for (int i  = 0; i < T; i++)
        scanf("%d %f %lf",&L[i].ID, &L[i].qtde, &L[i].valor);

    for(int i = 0; i < T; i++)
        printf("%d %.3lf\n",L[i].ID, L[i].qtde * L[i].valor);

    return 0;
}
```

Grandes poderes = Grandes responsabilidades

- Sempre que alocamos memória manualmente (malloc), precisamos liberar a memória.

- Do contrário, nosso programa nunca vai liberar a área de memória alocada pelo malloc
- Exceto quando o programa for fechado (toda a memória é liberada).

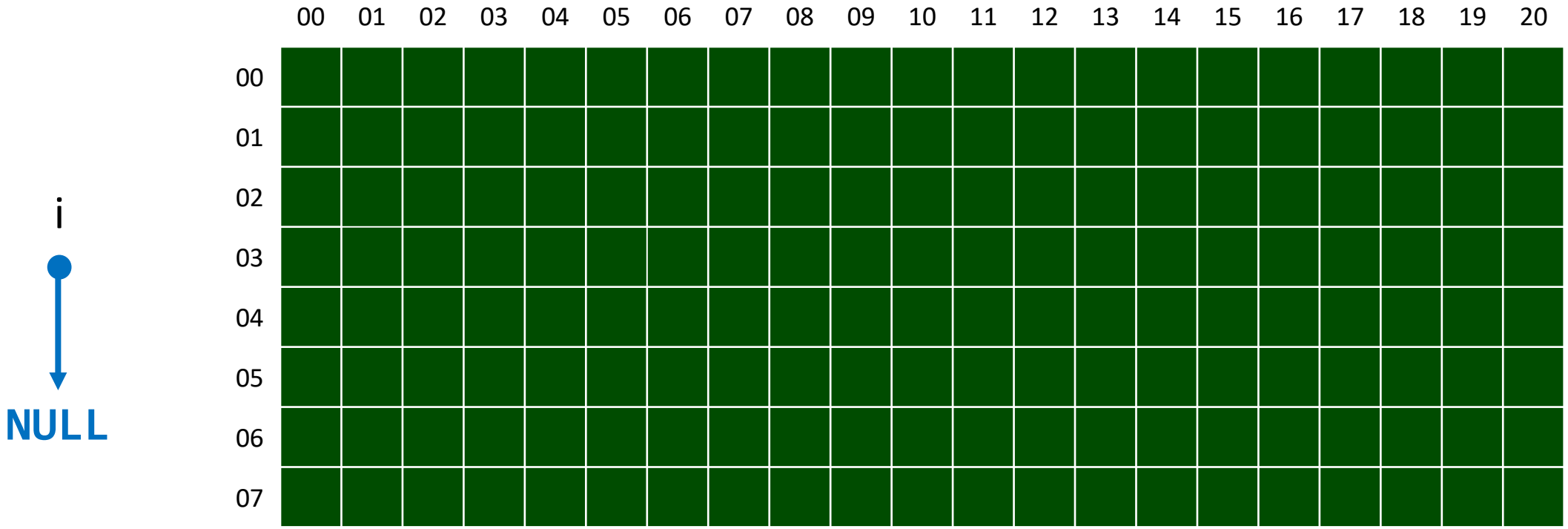
- O comando que libera a memória é o **free**.
- Ele requer o endereço de memória que alocamos.

Exemplo free

```
int main()
{
    int T;
    scanf("%d",&T);

    elemento* L = (elemento*) malloc (sizeof(elemento) * T);
    for (int i  = 0; i < T; i++)
        scanf("%d %f %lf",&L[i].ID, &L[i].qtde, &L[i].valor);

    for(int i = 0; i < T; i++)
        printf("%d %.3lf\n",L[i].ID, L[i].qtde * L[i].valor);

    free(L); //Liberar (devolver) a memória alocada anteriormente
    return 0;
}
```

Comando free

```
int* i = (int*) malloc (4);
*i = 10;
free(i);
```



Definir a lista L: usando o malloc

```
#define TAM 5

typedef struct {
      int ID;
      float qtde;
      double valor;
} elemento;



elemento* L;
L = (elemento*) malloc(TAM * sizeof(elemento));
...
free(L);
```

Listas de Alocação Sequencial

- As listas constituem a estrutura de dados mais simples de que dispomos.

- Os dados são armazenados como elementos de um vetor.

- Precisamos verificar alguns aspectos:
  - Como definir a lista (o vetor) ✓
  - Como verificar o número de elementos contidos na lista
  - Como inserir um novo elemento na lista
  - Como buscar por um determinado elemento na lista
  - Como remover um elemento na lista
  - Como alterar um elemento da lista

Verificar o número de elementos contidos na lista

- Após alocarmos uma lista, quantas posições dessa lista estão efetivamente ocupadas?

- Nenhuma! Não inserimos nenhum elemento na lista

- Como vamos controlar quantos elementos já inserimos na lista?
- Usando a mesma estratégia do exercício dos contêineres.

- Uma variável contadora: nElementos
- Na inicialização: nElementos = 0
- A cada inserção: nElementos = nElementos + 1
- A cada remoção: nElementos = nElementos - 1

Listas Sequenciais

- Ao usar listas precisamos de duas variáveis:
  - A própria lista (independente do tipo)
  - Uma variável para contabilizar quantas posições estão efetivamente ocupadas.

- Até agora para usar criar uma nova lista:
  - Inicializar/Declarar a lista
  - Inicializar a contadora de posições com 0

- Todas as operações (inserção, remoção, busca) vão necessitar dessas duas informações

Listas de Alocação Sequencial

- As listas constituem a estrutura de dados mais simples de que dispomos.

- Os dados são armazenados como elementos de um vetor.

- Precisamos verificar alguns aspectos:
  - Como definir a lista (o vetor) ✓
  - Como verificar o número de elementos contidos na lista ✓
  - Como inserir um novo elemento na lista
  - Como buscar por um determinado elemento na lista
  - Como remover um elemento na lista
  - Como alterar um elemento da lista

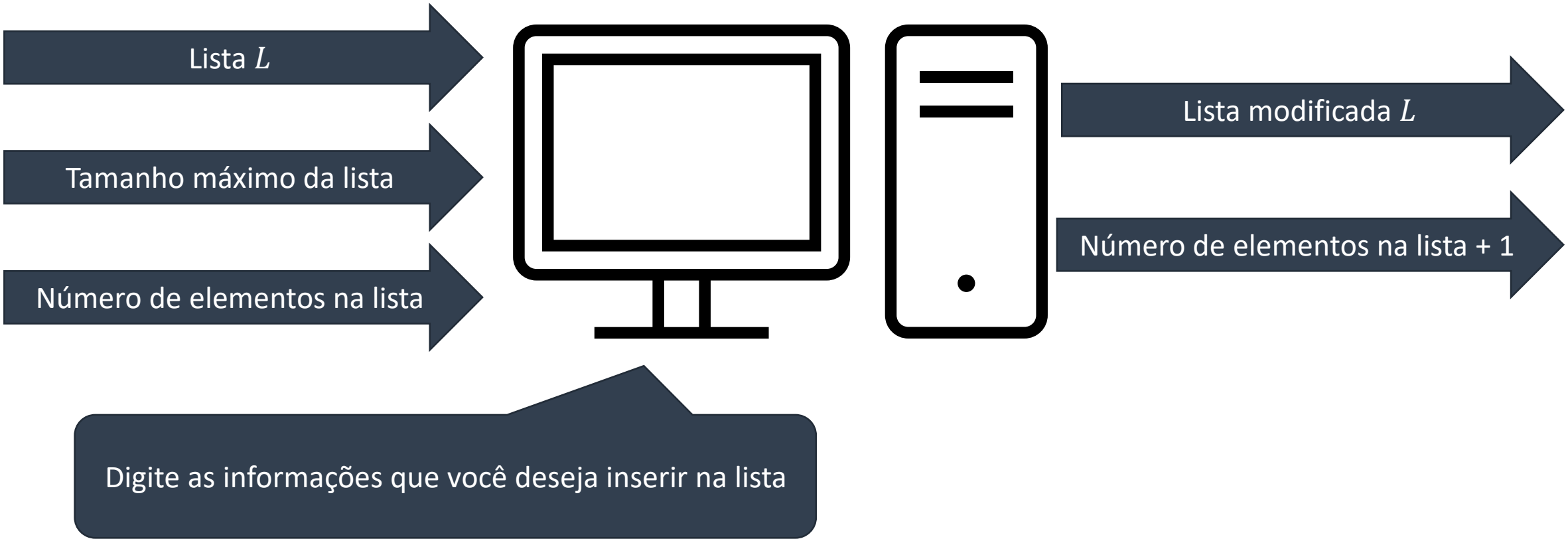
Inserção em Listas Sequenciais

- A inserção é a operação mais simples em uma lista

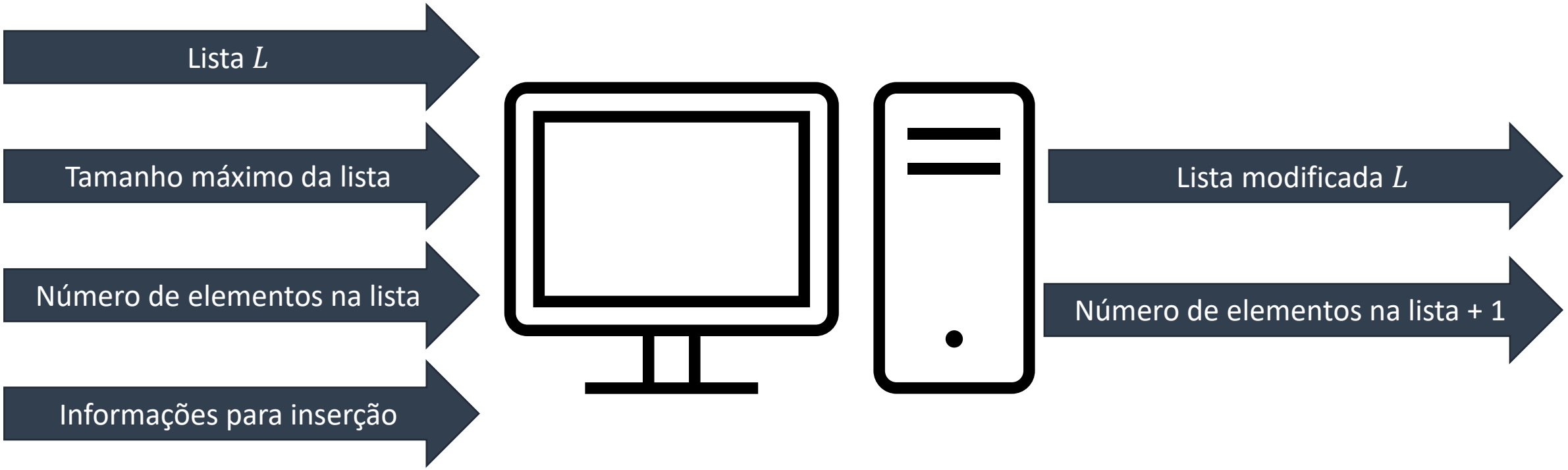
- Constituída de três passos:
  1. Verificar se a lista não está cheia. Se estiver, interromper a operação de inserção (não há espaço disponível).
  2. Ocupar a última posição da lista
  3. Incrementar a contadora de posições ocupadas.

- Precisa receber pelo menos três informações:
  1. Em qual lista devo inserir?
  2. Quantas posições essa lista comporta?
  3. Quantas posições estão ocupadas?

Inserção em Listas Sequenciais



Inserção em Listas Sequenciais



Saídas da inserção

- A lista atualizada (passagem por referência)
- O número de posições ocupadas (passagem por referência)?

- Um retorno para indicar se a inserção foi realizada?
  - Se 1, inserção OK
  - Se 0, erro na inserção (p.ex. lista cheia).

Inserção em Listas Sequenciais

```
int inserir(int lista[], int* nElementos, int tam)
{
    if (*nElementos < tam)
    {
        int n;
        scanf("%d",&n);
        lista[*nElementos] = n;
        *nElementos = *nElementos + 1;
        printf("Inserção realizada com sucesso.\n");
    }
    else
    {
        printf("Lista Cheia.\n");
        return 0;
    }
}
```

Listas de Alocação Sequencial

- As listas constituem a estrutura de dados mais simples de que dispomos.

- Os dados são armazenados como elementos de um vetor.

- Precisamos verificar alguns aspectos:
  - Como definir a lista (o vetor) ✓
  - Como verificar o número de elementos contidos na lista ✓
  - Como inserir um novo elemento na lista ✓
  - Como buscar por um determinado elemento na lista
  - Como remover um elemento na lista
  - Como alterar um elemento da lista

Busca em listas sequenciais

- Antes de entender a busca precisamos entender como a inserção foi realizada.

- Houve algum critério na ordenação? **Não**.
- Cada novo elemento foi inserido no final da lista (ocupando a última posição da lista).

- Se a inserção seguiu a ordem 1, 2, 3, 4, 5, a lista é [1,2,3,4,5]
- Se a inserção seguiu a ordem 5, 4, 3, 2, 1, a lista é [5,4,3,2,1]
- Se a inserção seguiu uma ordem aleatória, a lista é aleatória

Busca em listas sequenciais

- Como não houve nenhum critério (p.ex. uma ordenação), a única forma de encontrar um elemento em uma lista é procurando por toda a lista.

- Essa operação é chamada de **busca exaustiva**.
- Só podemos afirmar que um elemento não está na lista após exaurir toda a lista (olhar em cada posição ocupada).

- Em uma lista com 100 elementos, quantas comparações precisamos?
  - Estou com sorte: 1 comparação (o elemento é o primeiro da lista)
  - Estou com azar: 100 comparações (o elemento não está na lista)

Busca em listas sequenciais

- Ideia geral da busca exaustiva:
  - Comparar cada elemento com a chave buscada
  - Se encontrei, a busca é encerrada e a posição do elemento é retornada
  - Se não:
    - Se ainda não encerramos a lista, passar para a próxima posição da lista
    - Do contrário, terminar a busca: o elemento não está na lista

- Como em geral as listas são formadas por estruturas, precisamos buscar por uma determinada chave (um campo do struct).

Busca em listas sequenciais

- O que a busca precisa para trabalhar:
  - A lista L
  - O número de posições ocupadas na lista
  - A chave buscada

- O que a busca devolve:
  - A posição do elemento encontrado ou
  - Uma sinalização de que o elemento não foi encontrado

- Como sinalizar que o elemento não foi encontrado na lista
  - **Retornar -1 (um sinal previamente combinado)**
  - Retornar nElementos + 1

Busca em listas sequenciais

```
int buscar(int lista[], int nElementos, int chave)
{
    for (int i = 0; i < nElementos; i++)
    {
        if (lista[i] == chave)
            return i;
    }
    return -1; //Se não executamos nenhum return anterior
}
```

Busca em Listas

- Existem outras estratégias que podem acelerar a busca
  - Busca exaustiva com interrupção
  - Busca binária

- Mas essas estratégias exigem que a lista seja sempre mantida em ordem (em geral, crescente).

- Listas sequenciais ordenadas serão assunto de aulas futuras.

Listas de Alocação Sequencial

- As listas constituem a estrutura de dados mais simples de que dispomos.

- Os dados são armazenados como elementos de um vetor.

- Precisamos verificar alguns aspectos:
  - Como definir a lista (o vetor) ✓
  - Como verificar o número de elementos contidos na lista ✓
  - Como inserir um novo elemento na lista ✓
  - Como buscar por um determinado elemento na lista ✓
  - Como remover um elemento na lista
  - Como alterar um elemento da lista

Remoção em listas sequenciais

- A remoção de elementos em listas sequencias pode ser realizada:
  - Para remover sempre um elemento de uma posição específica:
    - Se remover sempre o último elemento: comportamento de pilha
    - Pilha: Último a entrar – Primeiro a sair

    - Se remover sempre o primeiro elemento: comportamento de fila
    - Fila: Primeiro a entrar – Primeiro a sair

  - Remover um determinado elemento (por exemplo: remover o elemento de chave 123).

  - Como pilhas e filas serão tema de aulas futuras, vamos considerar apenas o último caso.

Remoção em listas sequenciais

- A remoção de uma lista requer primeiro identificar o elemento que queremos remover.

- Para isso, precisamos usar a busca.
  - Se a busca retornar um índice: remover o elemento nessa posição
  - Se a busca retornar -1: o elemento não pode ser removido da lista (ele não está contido na lista).

- Desta forma:
  - Realizar busca
  - Remover o elemento que a busca apontou

Remoção em listas sequenciais

- Problemas na remoção:
  - Considere a seguinte lista:

  - Remover o elemento 3.
    - Mas qual 3? Existem dois!

  - Como lidar com elementos duplicados?
    1. Remover o primeiro elemento encontrado
    2. Não permitir inserções de elementos duplicados (mesma chave)

Remoção em listas sequenciais

- Problemas na remoção:
  - Considere a seguinte lista:

  - Remover o elemento 5.
    - OK! Só tem um.

Ficou um “buraco” na lista.
Como lidar com esses buracos?

Remoção em listas sequenciais

- Problemas na remoção:

- Como fechar o “buraco” que ficou na lista?

- Duas estratégias:
  1. Todos os demais elementos dão um passo a frente.
  2. Mover o último elemento para fechar o buraco. Vale observar que as listas não ficam com buracos na última posição. Basta decrementar o contador de elementos na lista.

Remoção em listas sequenciais

- Quando aplicar a primeira estratégia: Todos os demais elementos dão um passo a frente.
  - Quando a lista está ordenada (mover o último quebra a ordenação)
  - Quando temos uma fila (não queremos que o último passe a frente do segundo da fila).
  - Vale observar que esse problema não ocorre em pilhas.

- Quando usar a segunda estratégia: Mover o último elemento para fechar o buraco.
  - Casos em que a ordenação da lista não importa.
  - Nesse momento, esse é o nosso caso.

Remoção em listas sequenciais

- Perguntas interessantes:
  - As duas estratégias demandam o mesmo trabalho? Isto é: tem um custo computacional similar?
    - Mover os elementos uma posição para frente:
      - Necessário fazer n trocas (cada elemento da um passo a frente)
    - Trocar com o último elemento:
      - Uma única troca (o elemento do “buraco” com o último elemento da lista.
    - A primeira estratégia é muito mais custosa do que a segunda. Logo, seu uso deve ser mais criterioso.
  - Vale a pena usar a segunda estratégia e reordenar a lista?
    - Vamos estudar o custo dos algoritmos de ordenações em breve.

Remoção em listas sequenciais

- Problemas na remoção:

- Como fechar o “buraco” que ficou na lista?

- Duas estratégias:
  1. Todos os demais elementos dão um passo a frente.
  2. **Mover o último elemento para fechar o buraco**.

     Vale observar que as listas não ficam com buracos na última posição. Basta decrementar o contador de elementos na lista.

Remoção em listas sequenciais

- Podemos então resumir o processo de remoção em:
  1. Buscar o elemento na lista
  2. Sabendo a posição, guardar/apresentar as informações do elemento
  3. Trocar esse elemento com o último da lista
  4. Decrementar o contador de elementos na lista

- Vale observar que esse elemento não será apagado "fisicamente". Mas como ele está além do limite da lista "fingimos" que ele não existe mais.

Remoção em listas sequenciais

```
int remover(int lista[], int *nElementos, int chave)
{
    int pos = buscar(lista, *nElementos, chave);
    if (pos >= 0)
    {
        int aux = lista[pos];
        lista[pos] = lista[*nElementos - 1];
        lista[*nElementos] = aux;
        *nElementos = *nElementos - 1;
        printf("Remoção efetuada com sucesso");
        return 1;
    }
    else
    {
        printf("Elemento não está na lista.");
        return 0;
    }
}
```

Preciso mesmo trocar?

Por que salvar o elemento na última posição?

Remoção em listas sequenciais

```
int remover(int lista[], int *nElementos, int chave)
{
    int pos = buscar(lista, *nElementos, chave);
    if (pos >= 0)
    {
        lista[pos] = lista[*nElementos - 1];
        *nElementos = nElementos - 1;
        printf("Remoção efetuada com sucesso");
        return 1;
    }
    return 0;
}
```

Remoção em listas sequenciais

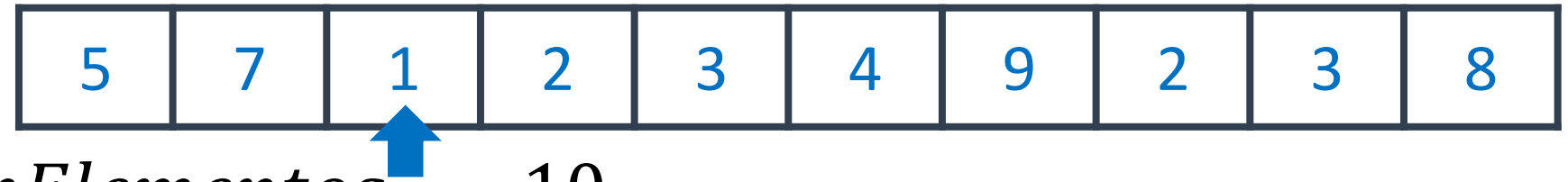
- Lista L

- $nElementos = 10$
- $chave = 1$

- 1) Buscar pela chave na lista
  - A busca retorna 2 como resposta (a chave 1 está na posição 2)

Remoção em listas sequenciais

- Lista L



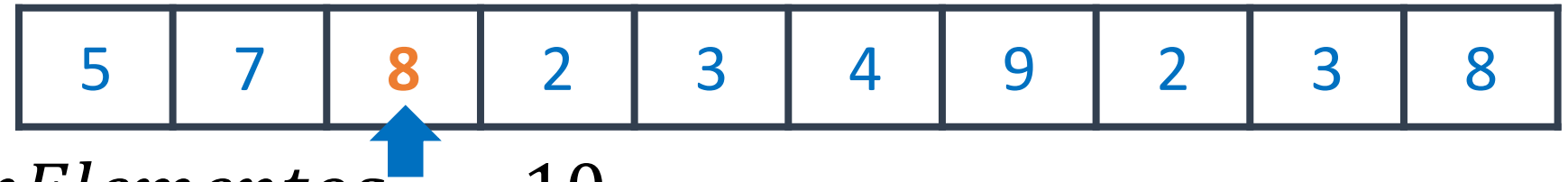
- $nElementos = 10$
- $chave = 1$
- $pos = 2$

- 2) Mover o último elemento para fechar o “buraco”

```
lista[pos] = lista[*nElementos - 1];
lista[2] = lista[9];
```

Remoção em listas sequenciais

- Lista L



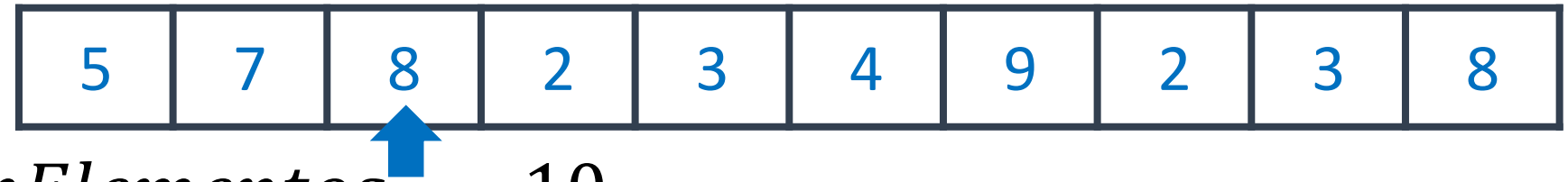
- $nElementos = 10$
- $chave = 1$
- $pos = 2$

- 2) Mover o último elemento para fechar o “buraco”

```
lista[pos] = lista[*nElementos - 1];
lista[2] = lista[9];
```

Remoção em listas sequenciais

- Lista L



- $nElementos = 10$
- $chave = 1$
- $pos = 2$

- 3) Decrementar o contador de elementos na lista

```
*nElementos = nElementos - 1;
```

Remoção em listas sequenciais

- Lista L

- $nElementos = 9$
- $chave = 1$
- $pos = 2$

- 3) Decrementar o contador de elementos na lista

```
*nElementos = nElementos - 1;
```

Listas de Alocação Sequencial

- As listas constituem a estrutura de dados mais simples de que dispomos.

- Os dados são armazenados como elementos de um vetor.

- Precisamos verificar alguns aspectos:
  - Como definir a lista (o vetor) ✓
  - Como verificar o número de elementos contidos na lista ✓
  - Como inserir um novo elemento na lista ✓
  - Como buscar por um determinado elemento na lista ✓
  - Como remover um elemento na lista ✓
  - Como alterar um elemento da lista

Alteração em listas sequenciais

- Assim como a remoção, a alteração requer que o elemento seja localizado na lista.
- Todos os problemas relacionados a chaves duplicadas se aplicam na alteração também.

- Após a busca:
  - Se o elemento não foi localizado: interromper a alteração
  - Alterar a informação desejada do elemento.

Alteração em listas sequenciais

- A alteração de elementos pode ocasionar alguns problemas:
  - Como tratar chaves repetidas?
    - Impedir a alteração?
    - Permitir chaves duplicadas?
  - Como lidar com a alteração da chave em listas ordenadas?
    - Se a chave for alterada, a ordenação pode ser comprometida.
    - Por exemplo: [1,2,3,4,5]. Suponha que a chave do 2 é alterada para 6. [1,6,3,4,5]
    - Nesses casos, um tratamento comum consiste em:
      1. Remover o elemento
      2. Inserir novamente, com a chave correta.

Alteração em listas sequenciais

```
int alterar(int lista[], int *nElementos, int chave)
{
    int pos = buscar(lista, *nElementos, chave);
    if (pos >= 0)
    {
        //Fazer a alteração necessária:
        scanf("%d",lista[pos]);
        printf("Alteração efetuada com sucesso");
        return 1;

    }
    else
    {
        printf("Elemento não está na lista.");
        return 0;
    }
}
```

Listas de Alocação Sequencial

- As listas constituem a estrutura de dados mais simples de que dispomos.

- Os dados são armazenados como elementos de um vetor.

- Precisamos verificar alguns aspectos:
  - Como definir a lista (o vetor) ✓
  - Como verificar o número de elementos contidos na lista ✓
  - Como inserir um novo elemento na lista ✓
  - Como buscar por um determinado elemento na lista ✓
  - Como remover um elemento na lista ✓
  - Como alterar um elemento da lista ✓

Conceito de TAD

- Quando implementamos uma determinada estrutura, podemos empacotar as ideias atreladas a essa estrutura como um TAD.

- Tipo Abstrato de Dados:
  - Definição de como o TAD será armazenado (vetores)
  - Definição de estruturas auxiliares de controle (quantas posições estão ocupadas)
  - Métodos para manipular esse tipo de dados: Inserção, Busca, Remoção, etc..
  - O conjunto desses fatores é chamado TAD.

- No caso das listas, aprendemos o TAD Lista.

Listas usando POO

- Primeiro vamos criar uma classe para cada item da lista.
- Na implementação, vamos precisar definir um atributo que indique a chave pela qual os itens vão ser identificados.
- Dois elementos serão iguais se apresentarem chaves iguais.

Item.h

```
#ifndef ITEM_H
#define ITEM_H
class Item
{
    public:
        Item(int _chave);
        Item();
        int getChave();
    protected:
        int chave;
};
#endif // ITEM_H
```

Item.cpp

```
#include "Item.h"
Item::Item()
{

}
Item::Item(int _chave)
{
    chave = _chave;
}
int Item::getChave()
{
    return chave;
}
```

Listas usando POO

- Em seguida, vamos criar uma classe para a lista.
- Essa classe deverá conter dois atributos:
  - Uma estrutura subjacente para armazenar os itens
  - Um contador para indicar quantos elementos estão armazenados
- Essa classe deverá conter métodos para:
  - Adicionar um elemento
  - Remover um elemento
  - Buscar por uma determinada chave
  - Acessar uma determinada posição da lista
- Além de um construtor, para inicializar a estrutura subjacente.
- E um destrutor, para liberar a memória usada pela estrutura.

Classe Lista Sequencial: ListaSequencial.h

```
#ifndef LISTASEQUENCIAL_H
#define LISTASEQUENCIAL_H
#include "Item.h"

#define MAX 10

class ListaSequencial
{
    public:
        ListaSequencial();
        virtual ~ListaSequencial();
        bool Adicionar(Item* novo);
        bool Remover(int chave);
        int BuscarPos(int chave);
        Item* Buscar(int chave);
        Item* Acessar(int pos);
        int getTamanho();
    private:
        Item **lista;
        int tamanho;
};

#endif // LISTASEQUENCIAL_H
```

ListaSequencial.cpp

```
ListaSequencial::ListaSequencial()
{
    //ctor
    lista = new Item*[MAX]; //new => criar nova
    tamanho = 0;
}

ListaSequencial::~ListaSequencial()
{
    delete lista; //delete => liberar a memória alocada
}
```

ListaSequencial.cpp

```
bool ListaSequencial::Adicionar(Item* novo)
{
    if (tamanho < MAX)
    {
        lista[tamanho] = novo;
        tamanho++;
        return true;
    }
    else
    {
        return false;
    }
}
```

ListaSequencial.cpp

```
int ListaSequencial::BuscarPos(int chave)
{
    for (int i = 0; i < tamanho; i++)
        if (lista[i]->getChave() == chave)
            return i;
    return -1;
}

Item* ListaSequencial::Buscar(int chave)
{
    int pos = BuscarPos(chave);
    if (pos != -1) return lista[pos];
    else return nullptr; //nullptr => ponteiro para null (nada)
}
```

ListaSequencial.cpp

```
bool ListaSequencial::Remover(int chave)
{
    int pos = BuscarPos(chave);
    if (pos == -1)
        return false;

    lista[pos] = lista[tamanho-1];
    tamanho--;
    return true;
}
```

ListaSequencial.cpp

```
Item* ListaSequencial::Acessar(int pos)
{
    if (pos < tamanho)
        return lista[pos];
    else
        return nullptr;
}

int ListaSequencial::getTamanho()
{
    return tamanho;
}
```

Para pensar: Operações em Listas Ordenadas

- Inserção:
  - O que é menos custoso computacionalmente:
    - Inserir na última posição e reordenar a lista?
    - Inserir no lugar correto? Mas como?
- Na remoção:
  - O que é menos custoso computacionalmente:
    - Mover o último e reordenar a lista?
    - Deslocar os elementos que estão à frente do elemento removido para ocupar o espaço (um passo à frente)
- Em resumo:
  - Como estimar o custo de uma operação?
  - Quanto custa um ordenar uma lista?

Exercícios

- Implemente um sistema de controle de funcionários de uma empresa.

- A empresa deseja armazenar os seguintes dados de cada funcionário:
  - ID
  - Nome
  - DataNascimento
  - Salario
  - CargaHoraria

- Implemente um sistema para cadastrar funcionários, remover funcionários do cadastro, buscar um funcionário no cadastro e oferecer um aumento para um funcionário em específico (pelo ID do mesmo).
- Implemente um Menu questionando qual operação deve ser realizada. Programe também uma opção SAIR.