Alocação Dinâmica

ECM404

TAD

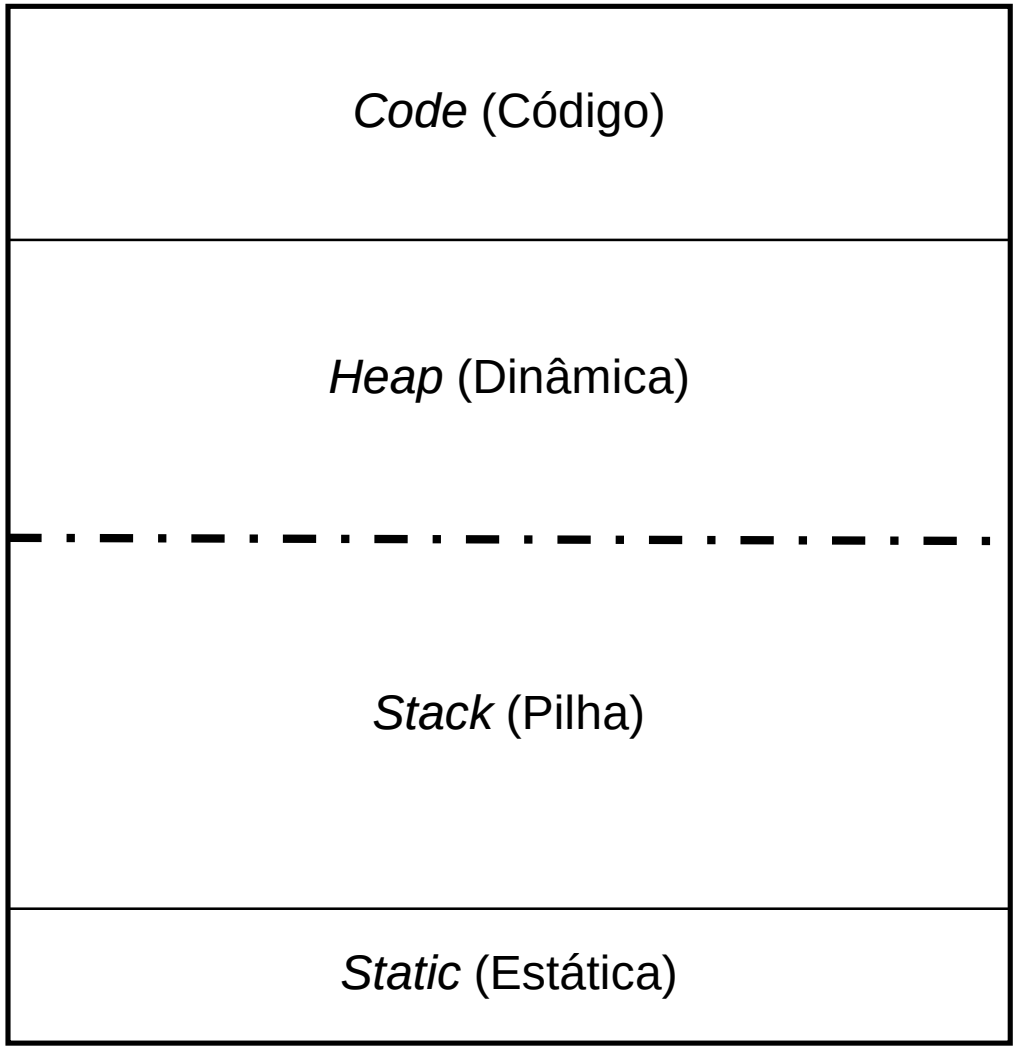
- Tipos Abstratos de Dados (TAD) – Implementações genéricas de comportamentos que podem ser aplicadas para qualquer tipo de dados, variando em função do problema modelado.
- TAD's que estudaremos no curso:
  - Lista:
    - Pilha e Fila;
  - Grafo:
    - Árvores;

Memória RAM

- A memória R.A.M. (Random Access Memory) é a memória controlada diretamente pelo microprocessador;
- Nela, os programas, com suas instruções em linguagem de máquina e dados, são carregados ao serem executados;
- Para gerenciar de forma organizada a memória R.A.M., o microprocessador a divide em 4 áreas:

Divisão da Memória RAM

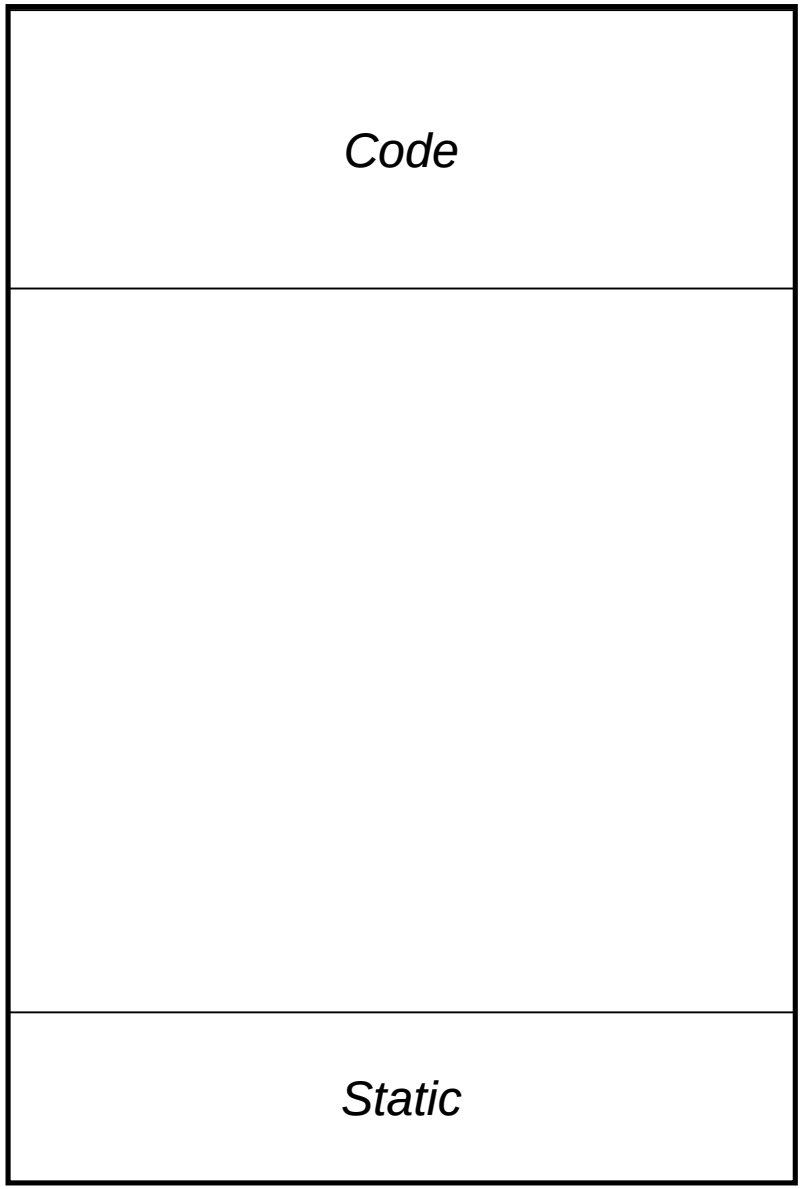
Executando um programa



Divisão da Memória RAM

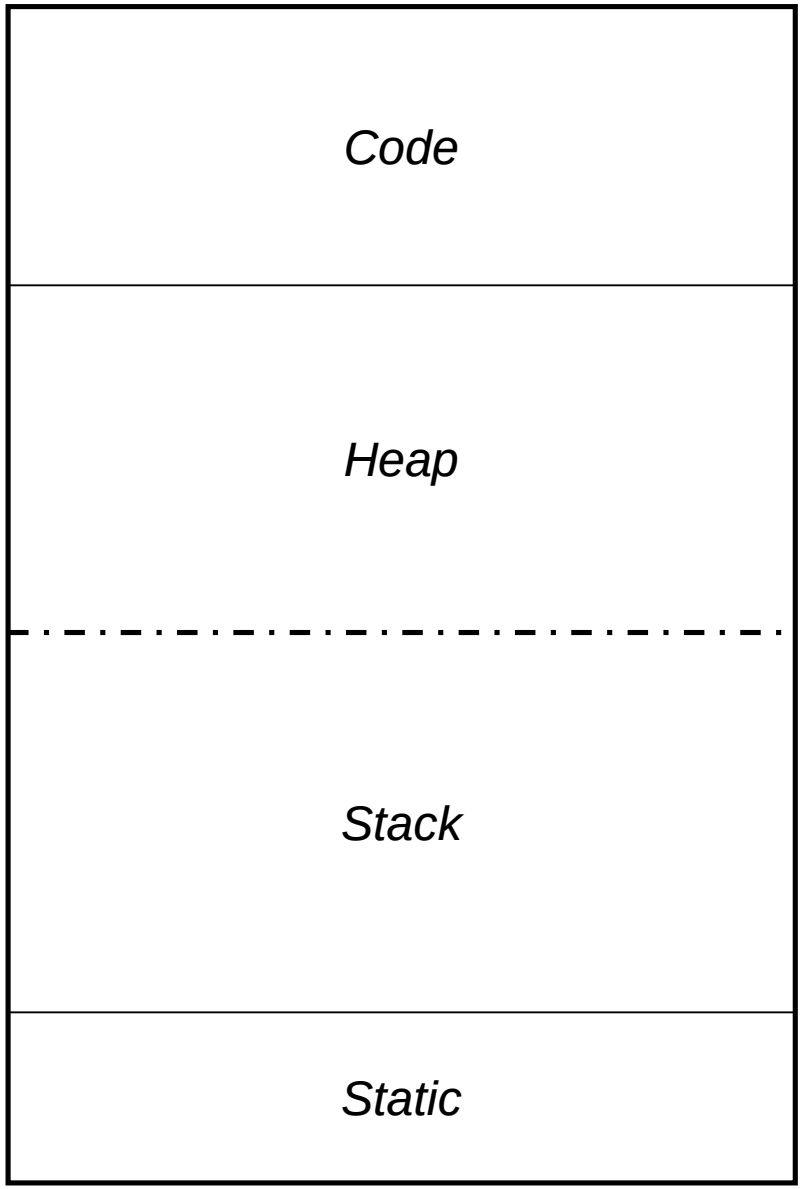
- **Code:** região da memória destinada ao armazenamento das instruções do programa, assim que o programa é executado;

- **Static:** após o programa ser carregado, as variáveis globais e as constantes são alocadas na memória estática;



Divisão da Memória RAM

- **Stack:** utilizada para armazenar informações sobre as sub-rotinas, seus pontos de retorno e variáveis locais;
- **Heap:** aqui é feita a alocação dinâmica (em tempo de execução);
- Não existe um limite físico entre a ***Stack*** e a ***Heap;***



Problema

- Qual a solução quando a quantidade de posições de memória para implementar uma solução é desconhecida?

- A declaração de vetores com tamanhos grande pode representar um desperdício de memória ou ainda sua falta em algumas situações;

Alocação Dinâmica de Memória

- A ***alocação dinâmica*** possibilita alocar memória em tempo de execução do código, criando um ponteiro para um vetor com o tamanho desejado na memória ***heap.***
- Esta alocação é realizada utilizando a função ***malloc( ).*** O retorno será um ponteiro para a primeira posição de memória do bloco alocado.

void *****_malloc_ (**unsigned int** _num_);

Alocação Dinâmica de Memória

- Observação: como a função retorna um ponteiro do tipo void, seu valor deve ser convertido para um ponteiro do tipo do dado que será armazenado.

```
int main (int argc, char *argv[]){
  int n, i, *dados;
  printf("Informe o tamanho do vetor: ");
  scanf("%i", &n);
  dados = (int*) malloc(n * sizeof(int));
  for (i = 0; i < n; i++)
      scanf("%i", (dados+i) );
  for (i = 0; i < n; i++)
      printf("Elemento%i: %i\n",i, *(dados+i) );
  free(dados);
  return 0;
}
```

Alocação Dinâmica de Memória

- A função ***sizeof( )*** retorna o tamanho em bytes de um tipo fornecido como parâmetro.

  É utilizada para determinar a quantidade de memória necessária para alocação.

```
int main (int argc, char *argv[]){
  int n, i, *dados;
  printf("Informe o tamanho do vetor: ");
  scanf("%i", &n);
  dados = (int*) malloc(n * sizeof(int));
  for (i = 0; i < n; i++)
      scanf("%i", (dados+i) );
  for (i = 0; i < n; i++)
      printf("Elemento%i: %i\n",i, *(dados+i) );
  free(dados);
  return 0;
}
```

Alocação Dinâmica de Memória

- a função ***free( )*** devolve para o sistema a memória tomada durante a alocação dinâmica. Quando um elemento alocado de forma dinâmica não for mais utilizado, a memória deve ser liberada utilizando a função ***free( )***.

```
int main (int argc, char *argv[]){
  int n, i, *dados;
  printf("Informe o tamanho do vetor: ");
  scanf("%i", &n);
  dados = (int*) malloc(n * sizeof(int));
  for (i = 0; i < n; i++)
      scanf("%i", (dados+i) );
  for (i = 0; i < n; i++)
      printf("Elemento%i: %i\n",i, *(dados+i) );
  free(dados);
  return 0;
}
```

Listas – Definição

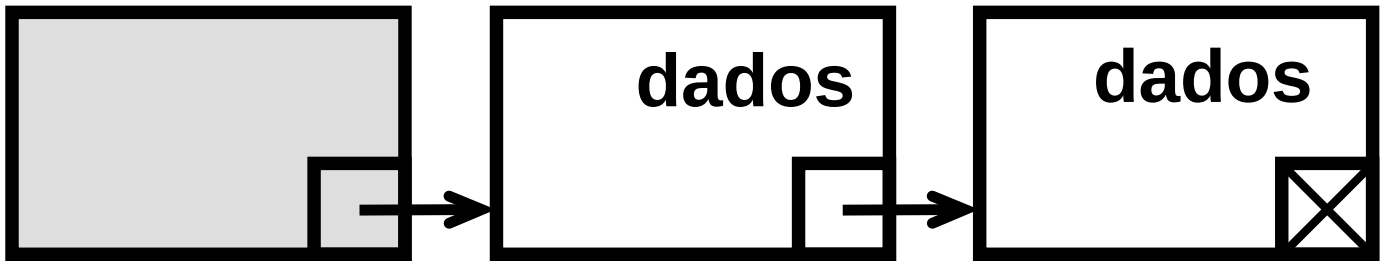
- Listas são estruturas dinâmicas de dados (utilizam alocação dinâmica de memória) e permitem manipular seus elementos (inserir, remover, buscar, ordenar etc.).
- A título de comparação, as variáveis indexadas (vetores e matrizes) são estruturas de dados estáticas porque o número de elementos reservados efetivamente não pode ser alterado durante a execução do programa.

Resumindo

- ***Variáveis indexadas estáticas:***
  - Sempre ocupam a mesma quantidade de memória.
- ***Variáveis indexadas dinâmicas:***
  - Ocupam a quantidade necessária de memória;
  - É possível inserir ou remover elementos que estão na última posição.
- ***Listas ligadas:***
  - Ocupam a quantidade necessária de memória;
  - É possível inserir ou remover elementos em qualquer posição.

Lista Ligada

- Estabelece ligações entre as informações armazenadas na memória Heap. Para isso, é necessário que uma informação saiba o endereço da outra;
- Primeiro elemento (cabeça):
  - Não possui nenhum dado referente ao conteúdo da lista;
  - Facilita as operações, embora não seja obrigatória sua criação;



Lista Ligada

- Para facilitar a implementação da estrutura da lista ligada, criaremos um novo tipo para os “nós” da lista.

```
#define MAX 100

typedef struct {
    float valor;
    char texto[MAX];
} Dados;

typedef struct SLista {
    Dados dados;
    struct SLista *prox;
} TLista;
```

Lista Ligada

```
int main (int argc, char *argv[]){
  char op;
    Dados dados;
    TLista *lista;
    lista = CriarLista();
    do {
        printf("valor: ");
        scanf("%f",&dados.valor);
        printf("texto: ");
        LerString(dados.texto);
        InserirNoFim(lista, dados);
        printf("Mais itens (S/N)? ");
        scanf("%c", &op);
    } while (toupper(op) != 'N');
    ExibirLista(lista);
    lista = DestruirLista(lista);
    return 0;
}
```

Lista Ligada – Elemento Cabeça

- O cabeça não serve para armazenar informações, mas sim, para indicar a existência da lista a partir de um certo endereço de memória;
- No programa principal o ponteiro ***lista*** sempre apontará para o cabeça.

Lista Ligada – Criação da Lista

- Chamada da função ***CriarLista:***

lista = CriarLista( );

- Ações:
  - Alocar memória para o cabeça;
  - Alocação OK? Indicar lista vazia (ponteiros);
  - Retornar o endereço do elemento cabeça para o programa principal.

Protótipo: TLista* CriarLista(void);

Lista Ligada – CriarLista

```
TLista* CriarLista(void) {
  TLista *p;   /* Ponteiro para a lista */
  p = (TLista*) malloc(sizeof(TLista));
  if (p == NULL){
    printf("Não pode criar a lista");
    exit(EXIT_FAILURE);
  }

  p->prox = NULL;   /* Atribui ponteiro nulo */
  return p;   /* Retorna endereço da lista */
}
```

Lista Ligada – Inserção na Lista

- A inserção será feita no final da lista, simulando uma fila.
- A função de inserção deve receber o endereço do elemento cabeça e as informações a serem inseridas.

Lista Ligada – Inserção na Lista

- Chamada da função ***InserirNoFim:***

InserirNoFim(lista, dados);

- Ações:
  - Cria novo nó;
  - Alocação OK? Armazena os dados neste nó;
  - Encontra o final da lista. Ao encontrar, apontar último elemento para o novo nó criado e marcar o novo nó como último;

Protótipo: void InserirNoFim(TLista*, Dados);

Lista Ligada – InserirNoFim

```
void InserirNoFim(TLista *p, Dados dados) {
    TLista* novo;
    novo = (TLista*) malloc(sizeof(TLista));
    if (novo == NULL) {
        printf("Nao foi possivel alocar memoria!");
        exit(EXIT_FAILURE);
    }

    novo->dados = dados;

    TLista *aux;
    aux = p;
    while(aux->prox != NULL)
        aux = aux->prox;

    novo->prox = NULL;
    aux->prox = novo;
}
```

Lista Ligada – Exibição da Lista

- Chamada da função ***ExibirLista:***

```
ExibirLista(lista);
```

- Ações (Se a lista não estiver vazia):
  - Apontar para o primeiro nó com informação:
    - Exibir informações;
    - Avançar para o próximo nó.

Protótipo: void ExibirLista(TLista*);

Lista Ligada – ExibirLista

```
void ExibirLista(TLista *p) {
    TLista *aux;
    aux = p;
    aux = aux->prox;
    while(aux != NULL) {
        printf("%4.2f\t%s\n",
                  aux->dados.valor,
                  aux->dados.texto);
        aux = aux->prox;
    }
}
```

Lista Ligada – Destruição da Lista

- Chamada da função ***DestruirLista:***

DestruirLista(lista);

- Ações:
  - Destrói a lista a partir do cabeça até o último elemento;
  - Retorna um endereço vazio.

Protótipo: TLista* DestruirLista(TLista*);

Lista Ligada – DestruirLista

```
TLista* DestruirLista(TLista *p) {
    TLista *aux;
    aux = p;
    while(aux->prox != NULL) {
        aux = aux->prox;
        free(p);
        p = aux;
    }
    free(p);
    return NULL;
}
```

Lista Ligada

```c
int main (int argc, char *argv[]){
  char op;
    Dados dados;
    TLista *lista;
    lista = CriarLista();
    do {
        printf("valor: ");
        scanf("%f",&dados.valor);
        printf("texto: ");
        LerString(dados.texto);
        InserirNoFim(lista, dados);
        printf("Mais itens (S/N)? ");
        scanf("%c", &op);
    } while (toupper(op) != 'N');
    ExibirLista(lista);
    lista = DestruirLista(lista);
    return 0;
}
```

```
valor: 1
texto: a
Mais itens (S/N)? s
valor: 2
texto: b
Mais itens (S/N)? s
valor: 3
texto: c
Mais itens (S/N)? s
valor: 4
texto: d
Mais itens (S/N)? s
valor: 5
texto: e
Mais itens (S/N)? n
1.00    a
2.00    b
3.00    c
4.00    d
5.00    e
```

Atividade

- Abra a pasta Downloads no VS Code
  - Utilize o comando git clone https://github.com/ECM404/lista8.git --recursive
  - Entre na pasta utilizando o comando cd lista8