

Estruturas de Dados I

Prof. Dr. Jesmmer Alves*

Instituto Federal Goiano Campus Morrinhos
Curso Bacharel em Ciência da Computação

* jesmmer.alves@ifgoiano.edu.br

Agenda

1 Sobre o Curso

2 Aspectos Preliminares

- Estruturas de Dados
- Análise de Algoritmos
- Notação Big “OH”
- Gerenciamento de Memória
- Pilhas

Informações Básicas

- Disciplina: Estrutura de Dados I
- Carga horária: 36.66h teóricas e 36.66h práticas

Ementa

Alocação dinâmica de memória. Estruturas dinâmicas: pilhas, filas, deck, listas encadeadas. Tipos abstratos de dados. Árvores. (Grafos - Introdução)

Bibliografia Básica

- CORMEN, Thomas H. et. al. Algoritmos - Teoria e Prática.
- GOODRICH, M. T., TAMASSIA, R. Estruturas de Dados e Algoritmos em Java.
- TENENBAUM, A.M. et al. Estruturas de Dados usando C.
- VINU V. Princípios de Estruturas de Dados Usando C e C++.

Agenda

1 Sobre o Curso

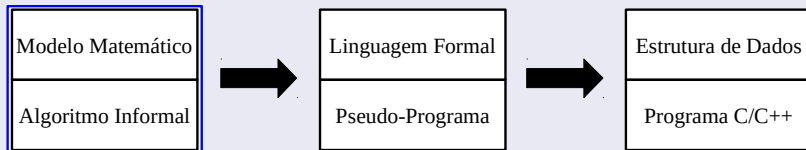
2 Aspectos Preliminares

- Estruturas de Dados
- Análise de Algoritmos
- Notação Big “OH”
- Gerenciamento de Memória
- Pilhas

Introdução sobre Estruturas de Dados

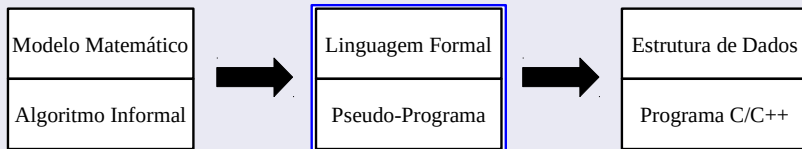
- Dados
 - entidade de um cálculo ou processo de manipulação
 - numéricos e alfanuméricos
 - simples ou conjunto de valores
- Estrutura de Dados
 - representação estrutural e relacionamentos + lógica funcional
 - dividir, conquistar e combinar
 - projeto e impacto na eficiência do programa

Passos para Elaboração de um Programa



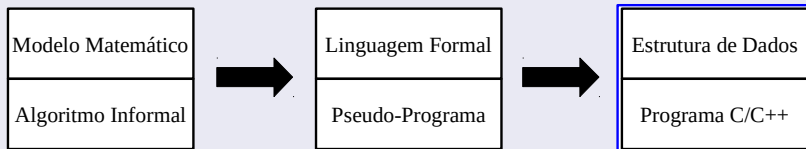
- visão de parte do sistema
- mapas mentais, fórmulas, ER, etc.
- algoritmo informal

Passos para Elaboração de um Programa



- padronização, regras
- algoritmo formal
- operações, expressões, funções

Passos para Elaboração de um Programa



- representação, relacionamento
- tipos de dados abstratos
- programa em uma linguagem e programação

Análise de Algoritmos

- analisar a corretude
- execução passo a passo
- prova matemática
- análise de simplicidade
- análise de performance
 - complexidade de espaço
 - complexidade de tempo



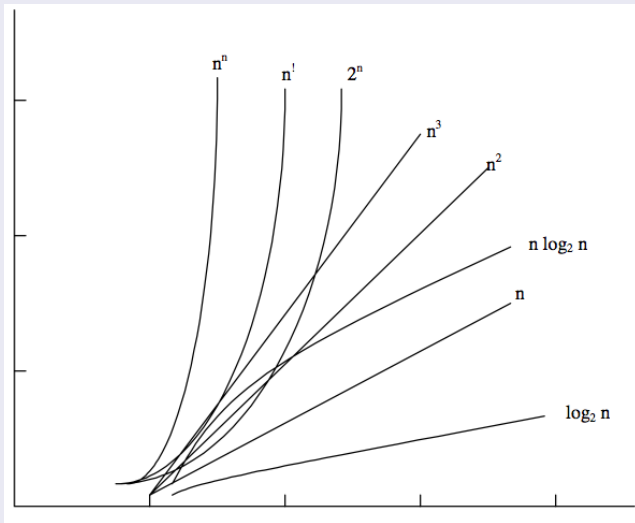
Complexidade de Espaço

- montante de memória necessário
- estimar tamanho do “problema”
- espaço de instrução (versão executável)
- espaço de dados
 - constantes e variáveis (fixo ou variável?)
 - variáveis estruturais (arrays e structures)
 - alocações dinâmicas
- espaço ambiente de pilha (chamada de funções)

Complexidade de Tempo

- montante de tempo necessário
- pode variar de acordo com o hardware
- analisar operações chave
- taxa de crescimento
 - $f(n)$ é uma função do tempo com relação à entrada n
 - funções exponenciais ($2^n, n^n, \dots$)
 - muito lento exceto para n pequeno
 - funções polinomiais (n, n^2, n^3, \dots)
- forma de análise
 - melhor caso, caso médio, pior caso

Complexidade de Tempo



Notação Big “OH”

- utilizado para analisar a performance do algoritmo
- determinar um limite superior para o tempo necessário
- busca sequencial (www.cs.usfca.edu/~galles/visualization/Search.html)
 - $f(n) = n \rightarrow O(n)$ (pior caso)
 - $f(n) = 1$ (melhor caso)
 - $f(n) = n/2$ (caso médio)
- busca binária (qual complexidade de tempo?)

Notação Big “OH” (exemplo 1)

```
A(){  
    int i;  
    for (i = 1 to n)  
        printf("Texto");  
}
```

- melhor caso?
- caso médio?
- pior caso?

Notação Big “OH” (exemplo 1)

```
A(){  
    int i;  
    for (i = 1 to n)  
        printf("Texto");  
}
```

- melhor caso? $\Rightarrow O(n)$
- caso médio?
- pior caso?

Notação Big “OH” (exemplo 1)

```
A(){  
    int i;  
    for (i = 1 to n)  
        printf("Texto");  
}
```

- melhor caso? $\Rightarrow O(n)$
- caso médio? $\Rightarrow O(n)$
- pior caso? $\Rightarrow O(n)$

Notação Big “OH” (exemplo 2)

```
B(){  
    int i, j;  
    for (i = 1 to n)  
        for (j = 1 to n)  
            printf("Texto");  
}
```

- melhor caso?
- caso médio?
- pior caso?

Notação Big “OH” (exemplo 2)

```
B(){  
    int i, j;  
    for (i = 1 to n)  
        for (j = 1 to n)  
            printf("Texto");  
}
```

- melhor caso? $\Rightarrow O(n^2)$
- caso médio? $\Rightarrow O(n^2)$
- pior caso? $\Rightarrow O(n^2)$

Notação Big “OH” (exemplo 3)

```
c(){  
    int i, j, k, n;  
    for (i = 1; i <= n; i++)  
        for (j = 1; j <= i; j++)  
            for (k = 1; k <= 100; k++)  
                printf("Texto");  
}
```

i =	1
j =	quantas vezes?
k =	?

Notação Big “OH” (exemplo 3)

```
c(){  
    int i, j, k, n;  
    for (i = 1; i <= n; i++)  
        for (j = 1; j <= i; j++)  
            for (k = 1; k <= 100; k++)  
                printf("Texto");  
}
```

i =	1
j =	quantas vezes?
k =	?

Notação Big “OH” (exemplo 3)

```
c(){  
    int i, j, k, n;  
    for (i = 1; i <= n; i++)  
        for (j = 1; j <= i; j++)  
            for (k = 1; k <= 100; k++)  
                printf("Texto");  
}
```

i =	1
j =	1 vez
k =	?

Notação Big “OH” (exemplo 3)

```
c(){  
    int i, j, k, n;  
    for (i = 1; i <= n; i++)  
        for (j = 1; j <= i; j++)  
            for (k = 1; k <= 100; k++)  
                printf("Texto");  
}
```

$$\left| \begin{array}{rcl} i & = & 1 \\ j & = & 1 \\ k & = & 100 \end{array} \right|$$

Notação Big “OH” (exemplo 3)

```
c(){  
    int i, j, k, n;  
    for (i = 1; i <= n; i++)  
        for (j = 1; j <= i; j++)  
            for (k = 1; k <= 100; k++)  
                printf("Texto");  
}
```

i =	1	2
j =	1	?
k =	100	?

Notação Big “OH” (exemplo 3)

```
c(){  
    int i, j, k, n;  
    for (i = 1; i <= n; i++)  
        for (j = 1; j <= i; j++)  
            for (k = 1; k <= 100; k++)  
                printf("Texto");  
}
```

i =	1	2
j =	1	2
k =	100	2 * 100

Notação Big “OH” (exemplo 3)

```
c(){  
    int i, j, k, n;  
    for (i = 1; i <= n; i++)  
        for (j = 1; j <= i; j++)  
            for (k = 1; k <= 100; k++)  
                printf("Texto");  
}
```

i =	1	2	3
j =	1	2	3
k =	100	$2 * 100$	$3 * 100$

Notação Big “OH” (exemplo 3)

```
c(){  
    int i, j, k, n;  
    for (i = 1; i <= n; i++)  
        for (j = 1; j <= i; j++)  
            for (k = 1; k <= 100; k++)  
                printf("Texto");  
}
```

i =	1	2	3	...	n
j =	1	2	3	...	n
k =	100	2 * 100	3 * 100	...	n * 100

Notação Big “OH” (exemplo 3)

```
c(){  
    int i, j, k, n;  
    for (i = 1; i <= n; i++)  
        for (j = 1; j <= i; j++)  
            for (k = 1; k <= 100; k++)  
                printf("Texto");  
}
```

i =	1	2	3	...	n
j =	1	2	3	...	n
k =	100	2 * 100	3 * 100	...	n * 100

$$\begin{aligned} &100 + 2 * 100 + 3 * 100 + \dots + n * 100 \\ &100(1 + 2 + 3 + \dots + n) \\ &100 \left(\frac{n(n+1)}{2} \right) = O(n^2) \end{aligned}$$

Exercícios de Revisão

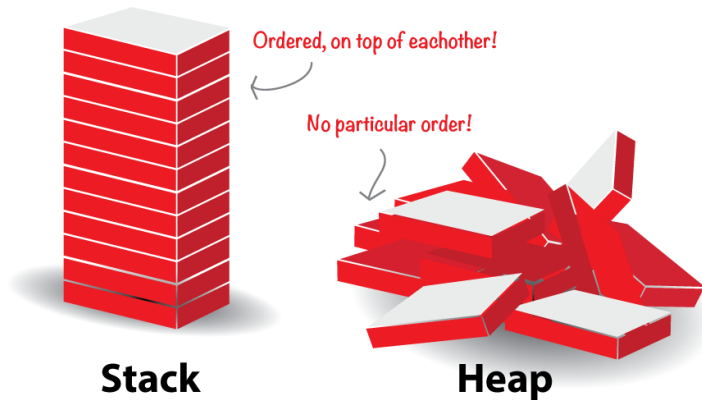
- 1 Diferencie complexidade de tempo e complexidade de espaço.
- 2 Fale sobre gerenciamento de alocação de memória dinâmica.
- 3 O que são tipos primitivos?
- 4 O que é mais importante analisar em um algoritmo: melhor caso, pior caso ou caso médio?
- 5 Considere a função “desfazer” de um editor de texto. Qual a melhor estrutura para implementar, pilhas ou filas?
- 6 Identifique uma situação que é conveniente utilizar a implementação filas.

Sobre Gerenciamento de Memória

- 1 Endereçamento de células de memória.
- 2 Gerenciar manualmente solicitações de alocação e liberação de memória .
- 3 Estruturas de dados dinâmica (adicionar, liberar e reorganizar).
- 4 Gerenciamento de memória dinâmica (em tempo de execução).
- 5 Aplicações: linked list, tree, etc.



Stack vs Heap



Stack vs Heap - Exemplos

```
int main () {  
    int valor = 5; //alocacao Stack  
  
    int *hvalor = new int; //alocacao Heap  
    *hvalor = 5;  
  
    return 0;  
}
```

Stack vs Heap - Exemplos

```
int main () {  
    int array[5]; //alocacao Stack  
  
    int *harray = new int[5]; //alocacao Heap  
    ...  
  
    return 0;  
}
```


Stack vs Heap

- ① Stack: alocação estática (quando compilar)
 - `int i, j;` *//dois bytes para inteiro*
 - `float a[5], f;` *//quatro bytes para real*
 - limitação no armazenamento
 - desperdício de espaço
- ② Heap: alocação dinâmica (durante execução)
 - reservar exatamente o espaço necessário
 - `malloc(); calloc(); realloc(); new(); free()`.

Stack vs Heap

- ① Stack: alocação estática (quando compilar)
 - `int i, j;` *//dois bytes para inteiro*
 - `float a[5], f;` *//quatro bytes para real*
 - limitação no armazenamento
 - desperdício de espaço
- ② Heap: alocação dinâmica (durante execução)
 - reservar exatamente o espaço necessário
 - `malloc(); calloc(); realloc(); new(); free()`.

Alocação de memória em C/C++ - malloc()

- reserva um bloco de bytes de memória
- retorna um ponteiro *void**
- o conteúdo não é inicializado
- sintaxe:

```
void* malloc (size_t size);
```

- exemplo:

```
buffer = (char*) malloc (sizeof(int));
```

Alocação de memória em C/C++ - malloc()

```
#include <stdlib.h> /* malloc, free, rand */
#include <iostream>
using namespace std;
int main () {
    int i,n;
    char * buffer;
    srand((unsigned)time(0));
    cout << "Qual o tamanho da string? "; cin >> i;

    buffer = (char*) malloc (i+1);
    if (buffer==NULL) exit (1);

    for (n = 0; n < i; n++)
        buffer[n] = rand()%26 + 97;
    buffer[i] = '\0';

    cout << "String aleatoria:" << buffer;
    free (buffer);

    return 0;
}
```

Alocação de memória em C/C++ - `calloc()`

- bloco de bytes de memória para um *array*
- retorna um ponteiro *void**
- conteúdo é inicializado com zero
- sintaxe:

```
void* calloc (size_t num, size_t size);
```

- exemplo:

```
pData = (int*) calloc (i, sizeof(int));
```

Alocação de memória em C/C++ - calloc()

```
#include <stdlib.h> /* malloc, free, rand */
#include <iostream>
using namespace std;
int main () {
    int i, n;
    int *pData;
    cout << "Qtd de numeros: ";
    cin >> i;
    pData = (int*) calloc (i,sizeof(int));
    if (pData == NULL) exit (1);

    for (n = 0; n < i; n++){
        cout << "Digite o " << n + 1 << "o. numero: ";
        cin >> pData[n];
    }

    cout << "Voce digitou: ";
    for (n = 0; n < i; n++) cout << pData[n] << " ";
    free(pData);
    return 0;
}
```

Alocação de memória em C/C++ - free()

- Libera o bloco de memória alocado por *malloc*, *calloc* ou *realloc*
- Evita *memory leak*
- Não retorna valores
- Sintaxe:
`void free (void* ptr);`
- Exemplo:
`B1= (int*) malloc (100*sizeof(int));`
`free (B1);`



Alocação de memória em C/C++ - free()

```
#include <stdlib.h> /* malloc, calloc, realloc, free */

int main () {
    int *buffer1, *buffer2, *buffer3;

    buffer1 = (int*) malloc (100*sizeof(int));
    buffer2 = (int*) calloc (100,sizeof(int));
    buffer3 = (int*) realloc (buffer2,500*sizeof(int));

    free (buffer1);
    free (buffer3);

    return 0;
}
```


Alocação de memória em C/C++ - `realloc()`

- Muda o tamanho do bloco de memória
- Muda o bloco de memória para outra localização
- Conteúdo é preservado
- Retorna um ponteiro para o novo bloco de memória
- Sintaxe:
`void* realloc (void* ptr, size_t size);`
- Exemplo:
`array2 = (int*) realloc (array1, qtd * sizeof(int));`

Alocação de memória em C/C++ - realloc()

```
#include <stdlib.h> /* malloc, free, rand */
#include <iostream>
using namespace std;

int main () {
    int entrada, n;
    int qtd = 0;
    int *numeros = NULL;
    int *mais_numeros = NULL;
    do {
        cout << "Digite um valor inteiro (0 para terminar): ";
        cin >> entrada;
        qtd++;
        mais_numeros = (int*) realloc (numeros, qtd * sizeof(int));
        if (mais_numeros != NULL) {
            numeros = mais_numeros;
            numeros[qtd-1] = entrada;
        }
        ...
    }
```

Alocação de memória em C/C++ - realloc() - continuação

```
...
else {
    free (numeros);
    cout << "Erro na re-alocacao de memoria";
    exit(1);
}
} while (entrada != 0);

cout << "Numeros digitados: ";
for (n = 0; n < qtd; n++)
    cout << numeros[n] << " ";

free (numeros);

return 0;
}
```

Alocação de memória em C/C++ - new e delete

- new

```
int *A = new int[n];
```

- delete

```
delete[] A;
```

Alocação de memória em C/C++ - new x malloc

new	malloc
Construtor	não é Construtor
retorna um tipo de dado	retorna void *
se falhar, throws	se falhar, NULL
usa memória livre	usa memória heap
permite sobreposição	não permite sobreposição
tamanho calculado pelo compilador	calculado manualmente

Atividades

- 1 Refaça todos os exemplos utilizando `new` e `delete`.
- 2 Diferencie `malloc` e `new`.
- 3 Diferencie `delete` e `free`.
- 4 Escreva um programa para encontrar o maior elemento de um conjunto de valores digitados, usando alocação dinâmica de memória.
- 5 Compare a alocação de memória Stack com alocação Heap.

Características

- Coleção de itens ordenada
- Estrutura de dados linear
- Estrutura de dados recursiva
- Somente uma opção para inserir ou remover (Topo)
- *Last-in-First-out*
- Implementada com *arranjos* ou *listas*



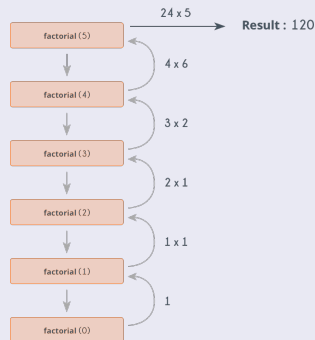
Aplicações

- Avaliar expressões *prefix*, *postfix* e *infix*
 - $5 + 6 * 7$?
- *Backtracking*
- Inverter *Strings*
- Chamadas de Funções



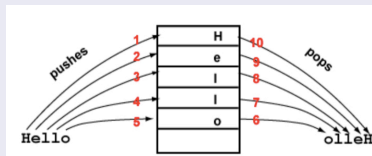
Aplicações

- Avaliar expressões *prefix*, *postfix* e *infix*
 - $5 + 6 * 7?$
- ***Backtracking***
- Inverter *Strings*
- Chamadas de Funções



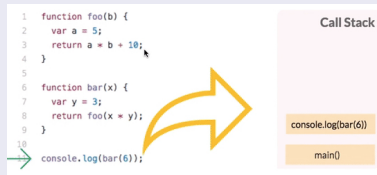
Aplicações

- Avaliar expressões *prefix*, *postfix* e *infix*
 - $5 + 6 * 7?$
- *Backtracking*
- **Inverter *Strings***
- Chamadas de Funções



Aplicações

- Avaliar expressões *prefix*, *postfix* e *infix*
 - $5 + 6 * 7$?
- *Backtracking*
- Inverter *Strings*
- Chamadas de Funções



Exemplos de Implementações

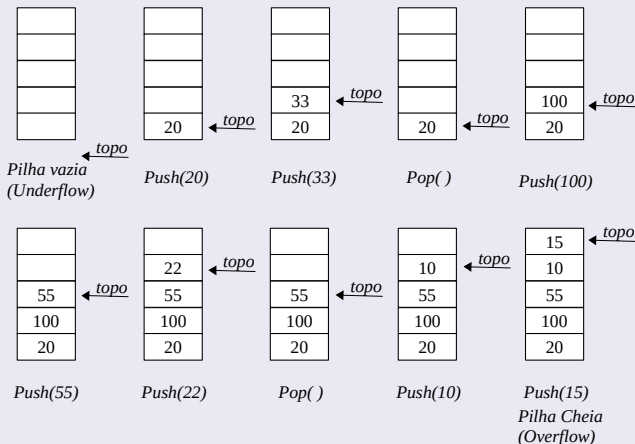
- Pilhas com Arrays (www.cs.usfca.edu/~galles/visualization/StackArray.html)
- Pilhas com Ponteiros e Listas (www.cs.usfca.edu/~galles/visualization/StackI)

Operações na Pilha

- *Push(x)*
 - Inserir um novo elemento na Pilha
 - *topo* é incrementado em 1
 - Pode ocorrer o *Overflow* (pilha cheia)
- *Pop()*
 - Remover um elemento da Pilha
 - *topo* é decrementado em 1
 - Pode ocorrer o *Underflow* (pilha vazia)

Pilhas (*Stacks*)

Operações na Pilha



Implementação com *Arranjos* ou *Ponteiros*

- Implementação com *Arranjo*
 - Fácil implementação
 - Tamanho não flexível
 - Otimização de memória ruim
- Implementação com *Ponteiros*
 - Implementação mais difícil
 - Tamanho flexível
 - Otimização de memória boa

Implementação com *Arranjos (Push)*

```
void Push(int x){
    if (top == MAX_SIZE - 1){
        cout << "\n Erro: A Pilha esta cheia!";
        return;
    }
    top = top + 1;
    A[top] = x;
    // A[++top] = x
}
```


Implementação com Arrays (Pop)

```
void Pop(){
    if (top == -1){
        cout << "\n Erro: A Pilha esta vazia!";
        return;
    }
    top--;
}
```

Atividades - Implementação de *Pilhas* com *Arranjos*

- 1 Utilize os exemplos dos *Slides* para escrever um programa em *C++* para fazer a implementação de uma *pilha* utilizando *arranjos*.
- 2 Insira a função *Estado* para: mostrar o elemento que está no topo da *pilha*; mostrar a quantidade de elementos; e informar se está vazia, cheia ou funcionando.
- 3 Insira a função *Imprimir* para escrever todos os elementos na *pilha*.
- 4 A função *main* deve disponibilizar um *Menu* para interagir com a *pilha*.
- 5 Identifique vantagens e desvantagens na implementação de *pilhas* com *Arranjos*.
- 6 Utilize a biblioteca *C++ STL* para implementar *pilhas*.



VINU V. DAS: Princípios de Estruturas de Dados Usando C e C++, MIT Press, (2006)



THOMAS H. et. al., Algoritmos - Teoria e Prática, (1989)



TENENBAUM, A.M. et al, Estruturas de Dados usando C., Pearson, (2010)



GOODRICH, M. T. e TAMASSIA, R., Estruturas de Dados e Algoritmos em Java., Bookman, (2013)