

Estrutura de Dados

Claudio Cesar de Sá, Alessandro Ferreira Leite, Lucas Hermman
Negri, Gilmário Barbosa

Departamento de Ciência da Computação
Centro de Ciências e Tecnologias
Universidade do Estado de Santa Catarina

5 de março de 2018

Sumário I

Agradecimentos

Vários autores e colaboradores ...

- Ao Google Images ...

Onde estamos ...

Disciplina

Estrutura de Dados – EDA001

- **Turma:**
- **Professor:** Claudio Cesar de Sá
 - claudio.sa@udesc.br
 - Sala 13 Bloco F
- **Carga horária:** 72 horas-aula • Teóricas: 36 • Práticas: 36
- **Curso:** BCC
- **Requisitos:** LPG, Linux, sólidos conhecimentos da linguagem C – há um documento específico sobre isto
- **Período:** 1º semestre de 2018
- **Horários:**
 - 3^a 15h20 (2 aulas) - F-205 – aula expositiva
 - 5^a 15h20 (2 aulas) - F-205 – lab

Ementa

Ementa

Representação e manipulação de tipos abstratos de dados. Estruturas lineares. Introdução a estruturas hierárquicas. Métodos de classificação. Análise de eficiência. Aplicações.

Objetivos I

- ***Geral:***

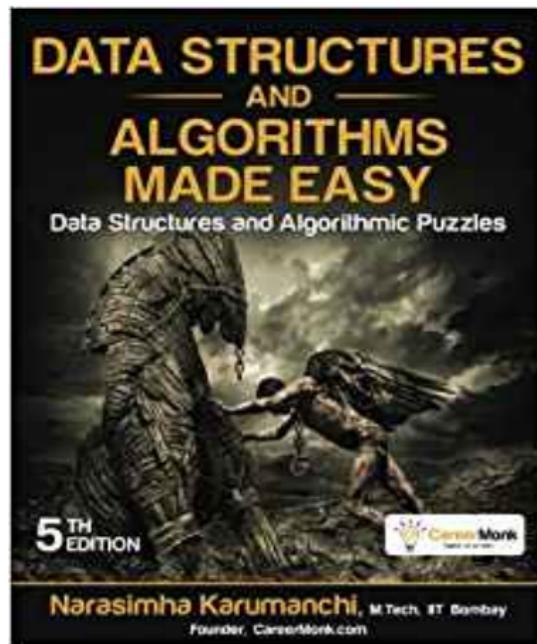
Há um documento específico sobre isto = Plano de Ensino

Objetivos II

- *Específicos:*

Há um documento específico sobre isto = Plano de Ensino

Livros que estarei usando ...



Conteúdo programático

Há um documento específico sobre isto = Plano de Ensino

Bibliografia UDESC

Há um documento específico sobre isto = Plano de Ensino

Conteúdo programático

Há um documento específico sobre isto = Plano de Ensino

Ferramentas ... nesta ordem

- Linux (veja o diretório de linux no GitHub deste curso)
- Algumas aulas sobre uso comandos do Linux (nivelamento)
- Linguagem C (ora o compilador g++)
- Codeblock, Geany, Sublime, Atom, etc

Metodologia e avaliação I

Metodologia:

As aulas serão expositivas e práticas. A cada novo assunto tratado, exemplos são demonstrados utilizando ferramentas computacionais adequadas para consolidar os conceitos tratados.

Metodologia e avaliação II

Avaliação

- n-provas – $\approx 90\%$
 - P_1 : xx/mar ou ago
 - P_2 : xx/abril ou set
 - P_3 : xx/maio ou set
 - P_4 : xx/junho ou out
 - P_5 : xx/junho ou nov
 - TF : Trabalho Final sobre árvores $\approx 10\%$
 - P_F : 2x/junho ou nov
- (provão: todo conteúdo)
- Exercícios de laboratório – \approx pouco ... %
- Presença e participação: 75% é o mínimo obrigatório para a UDESC. Quem quiser faltar por razões diversas, ou assuntos específicos, trate pessoalmente com o professor.

Metodologia e avaliação III

- Tarefas extras que geram pontos por excelência
- Média para aprovação: 6,0 (seis)
Nota maior ou igual a 6,0, repito a mesma no Exame Final. Caso contrário, regras da UDESC se aplicam.
- Sítio das avaliações: <https://run.codes/Users/login> código da disciplina: **SJ95**
- Turma: 2018-1

Dinâmica de Aula I

- Há um monitor na disciplina – Lucas – ver no site de monitoria da UDESC os horários
- Há uma lista de discussão (para avisos e dúvidas gerais):
eda-lista@googlegroups.com
- ≈ Teoria na 3a. feira
- ≈ Prática na 5a. feira
- E/ou 50% do tempo em teoria, 50% implementações
- Onde tudo vai estar atualizado?

Dinâmica de Aula II

- https://github.com/claudiosa/CCS/tree/master/estrutura_dados_EDA
- Ou seja, tudo vai estar *rolando* no GitHub do professor
- No Google: github + claudiosa
- Finalmente ...

Dinâmica de Aula III

- Questões específicas (leia-se: notas, dor-de-dente, etc) venha falar pessoalmente com o professor!

Bibliografia I

Básica:

- Há um documento específico sobre isto = Plano de Ensino ... veja em detalhes tudo que foi escrito aqui
- Mais uma vez: https://github.com/claudiosa/CCS/tree/master/estrutura_dados_EDA

Antes de Começarmos I

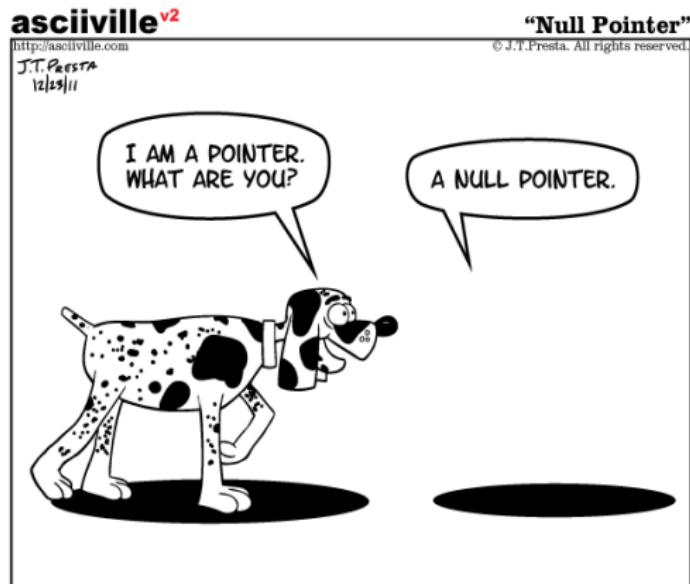
- Todos os cursos de Estrutura de Dados começam com uma motivação em torno da área para Ciência
- Vou omitir ... mas reflita se ela é ou não onipresente no nosso cotidiano?
- Exemplos: bancos eletronicos, web, smartphones, etc

Onde estamos ...

Capítulo 02 – Ponteiros I

Pontos fundamentais a serem cobertos:

- ① Pré-requisito: prática na linguagem C
- ② Exemplos – lúdicos
- ③ Ponteiros aos diversos tipos de dados
- ④ Uso de Memória
- ⑤ Alocação de memória Estática x Dinâmica
- ⑥ Funções para alocação de memória
- ⑦ Utilizando as funções para alocação de memória
- ⑧ Alocação de memória e



Motivação aos Ponteiros I

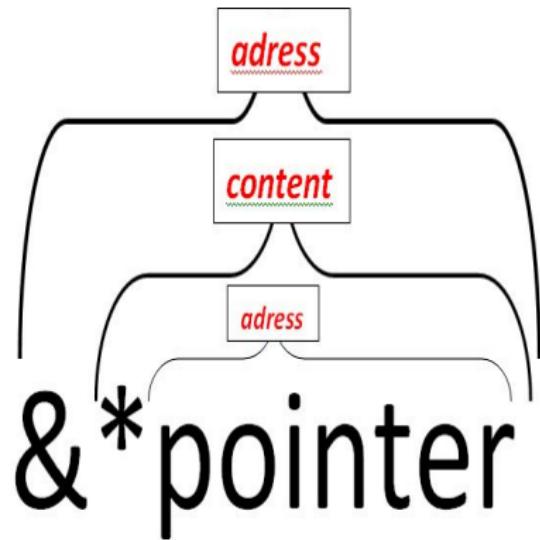


Figura 1: A história está por vir ...

Exemplos Lúdicos

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     int *ptr; // declara um ponteiro -- ptr-- para um inteiro
6             // um ponteiro para uma variavel do tipo inteiro
7     a = 90;
8     ptr = &a;
9     printf("Valor de A: %d\n", a);
10    printf("Valor de ptr: %d \t Conteudo via ptr: %d\n", ptr, *ptr);
11    return 1;
12 }
```

Exemplos Lúdicos

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     int *ptr; // declara um ponteiro -- ptr-- para um inteiro
6             // um ponteiro para uma variavel do tipo inteiro
7     a = 90;
8     ptr = &a;
9     printf("Valor de A: %d\n", a);
10    printf("Valor de ptr: %d \t Conteudo via ptr: %d\n", ptr, *ptr);
11    return 1;
12 }
```

- Ao executarmos esse código, qual será a sua saída?

Exemplos e agora?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     int a;
6     int *ptr; // declara um ponteiro -- ptr-- para um inteiro
7             // um ponteiro para uma variavel do tipo INTEIRO
8     a = 2017;
9     ptr = &a;
10    system("clear");
11    printf("Valor de A: %d\t Endereco de A: %x\n", a, &a);
12    printf("Valor de ptr: %d \t Conteudo via ptr: %d\n", ptr, *ptr);
13    printf("Endereco de PTR: %x\n", &ptr);
14    return 1;
15 }
```

Exemplos e agora?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     int a;
6     int *ptr; // declara um ponteiro -- ptr-- para um inteiro
7             // um ponteiro para uma variavel do tipo INTEIRO
8     a = 2017;
9     ptr = &a;
10    system("clear");
11    printf("Valor de A: %d\t Endereco de A: %x\n", a, &a);
12    printf("Valor de ptr: %d \t Conteudo via ptr: %d\n", ptr, *ptr);
13    printf("Endereco de PTR: %x\n", &ptr);
14    return 1;
15 }
```

Exemplos e agora?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     int a;
6     int *ptr; // declara um ponteiro -- ptr-- para um inteiro
7             // um ponteiro para uma variavel do tipo INTEIRO
8     a = 2017;
9     ptr = &a;
10    system("clear");
11    printf("Valor de A: %d\t Endereco de A: %x\n", a, &a);
12    printf("Valor de ptr: %d \t Conteudo via ptr: %d\n", ptr, *ptr);
13    printf("Endereco de PTR: %x\n", &ptr);
14    return 1;
15 }
```

- Qual é a saída do código acima?
- Quando não souber como funciona um comando em C?
- Várias respostas ... pense nelas e veja a melhor para você!

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...
- Quanto o * é a essência das linguagens C e C++

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...
- Quanto o * é a essência das linguagens C e C++
- Entendendo isto vais entender o que há nas bibliotecas, como STL, etc

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...
- Quanto o * é a essência das linguagens C e C++
- Entendendo isto vais entender o que há nas bibliotecas, como STL, etc
- Resumindo:

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...
- Quanto o * é a essência das linguagens C e C++
- Entendendo isto vais entender o que há nas bibliotecas, como STL, etc
- Resumindo:

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...
- Quanto o * é a essência das linguagens C e C++
- Entendendo isto vais entender o que há nas bibliotecas, como STL, etc
- Resumindo:
 - O nome do ponteiro retorna o endereço para o qual ele aponta

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...
- Quanto o * é a essência das linguagens C e C++
- Entendendo isto vais entender o que há nas bibliotecas, como STL, etc
- Resumindo:
 - O nome do ponteiro retorna o endereço para o qual ele aponta
 - O operador (&) junto ao nome do ponteiro retorna o endereço do ponteiro

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...
- Quanto o * é a essência das linguagens C e C++
- Entendendo isto vais entender o que há nas bibliotecas, como STL, etc
- Resumindo:
 - O nome do ponteiro retorna o endereço para o qual ele aponta
 - O operador (&) junto ao nome do ponteiro retorna o endereço do ponteiro
 - O operador (*) junto ao nome do ponteiro retorna o conteúdo da variável apontada

Ponteiro

- O correto entendimento e uso de ponteiros é crítico para uma programação bem-sucedida em C.

Ponteiro

- O correto entendimento e uso de ponteiros é crítico para uma programação bem-sucedida em C.
- Ponteiros são muitos utilizados em C, em parte porque eles são, às vezes, a única forma de expressar uma computação.

- O correto entendimento e uso de ponteiros é crítico para uma programação bem-sucedida em C.
- Ponteiros são muitos utilizados em C, em parte porque eles são, às vezes, a única forma de expressar uma computação.
- Em alguns casos, o uso de ponteiro resulta em um código mais compacto e eficiente que obtido de outras formas.

- O correto entendimento e uso de ponteiros é crítico para uma programação bem-sucedida em C.
- Ponteiros são muitos utilizados em C, em parte porque eles são, às vezes, a única forma de expressar uma computação.
- Em alguns casos, o uso de ponteiro resulta em um código mais compacto e eficiente que obtido de outras formas.
- Ponteiros e vetores são intimamente, relacionados.

- O correto entendimento e uso de ponteiros é crítico para uma programação bem-sucedida em C.
- Ponteiros são muitos utilizados em C, em parte porque eles são, às vezes, a única forma de expressar uma computação.
- Em alguns casos, o uso de ponteiro resulta em um código mais compacto e eficiente que obtido de outras formas.
- Ponteiros e vetores são intimamente, relacionados.
- Aquela parte de vetores terem dimensões especificadas e fixas, é conhecida como **alocação estática**.

- O correto entendimento e uso de ponteiros é crítico para uma programação bem-sucedida em C.
- Ponteiros são muitos utilizados em C, em parte porque eles são, às vezes, a única forma de expressar uma computação.
- Em alguns casos, o uso de ponteiro resulta em um código mais compacto e eficiente que obtido de outras formas.
- Ponteiros e vetores são intimamente, relacionados.
- Aquela parte de vetores terem dimensões especificadas e fixas, é conhecida como **alocação estática**.
- Os ponteiros irão servir para contornar esta limitação

Ponteiro

- Basicamente há três razões para utilizar ponteiros:

Ponteiro

- Basicamente há três razões para utilizar ponteiros:
 - ➊ Ponteiros fornecem os meios pelos quais as funções podem modificar seus argumentos;

- Basicamente há três razões para utilizar ponteiros:
 - ① Ponteiros fornecem os meios pelos quais as funções podem modificar seus argumentos;
 - ② Ponteiros são usados para suportar as rotinas de **alocação dinâmica** em C;

- Basicamente há três razões para utilizar ponteiros:
 - ① Ponteiros fornecem os meios pelos quais as funções podem modificar seus argumentos;
 - ② Ponteiros são usados para suportar as rotinas de **alocação dinâmica** em C;
 - ③ O uso de ponteiros pode aumentar a eficiência de certas rotinas.

- Basicamente há três razões para utilizar ponteiros:
 - ① Ponteiros fornecem os meios pelos quais as funções podem modificar seus argumentos;
 - ② Ponteiros são usados para suportar as rotinas de **alocação dinâmica** em C;
 - ③ O uso de ponteiros pode aumentar a eficiência de certas rotinas.
- Por outro lado, ponteiros podem ser comparados ao uso do comando **goto**, como uma forma diferente de escrever códigos impossíveis de entender.

Ponteiros e endereços

- Em um máquina típica, a memória é organizada como um vetor de células consecutivas numeradas ou endereçadas, que podem ser manipuladas individualmente ou em grupos contínuos.

Ponteiros e endereços

- Em um máquina típica, a memória é organizada como um vetor de células consecutivas numeradas ou endereçadas, que podem ser manipuladas individualmente ou em grupos contínuos.
- Uma situação comum é que qualquer *byte* pode ser um *char*, um par de células de um *byte* pode ser tratado como um inteiro *short*, etc.

Ponteiros e endereços

- Em um máquina típica, a memória é organizada como um vetor de células consecutivas numeradas ou endereçadas, que podem ser manipuladas individualmente ou em grupos contínuos.
- Uma situação comum é que qualquer *byte* pode ser um *char*, um par de células de um *byte* pode ser tratado como um inteiro *short*, etc.
- Um ponteiro é um grupo de células que podem conter um endereço.

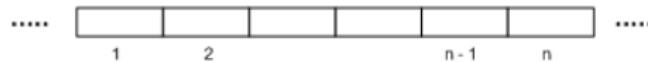


Figura 2: Representação da memória de uma máquina típica

Ponteiro

Definição

- É uma **variável que contém um endereço de memória**. Esse endereço é normalmente a posição de memória de uma outra variável.

Ponteiro

Definição

- É uma **variável que contém um endereço de memória**. Esse endereço é normalmente a posição de memória de uma outra variável.
- Se uma variável contém o endereço de uma outra, então a primeira é dita um ponteiro para a segunda.

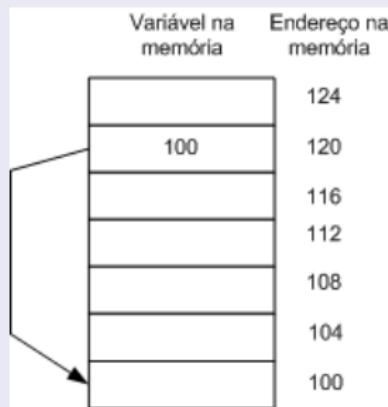


Figura 3: Representação de ponteiro

Variáveis Ponteiros

- ① A linguagem C permite o armazenamento e a manipulação de valores de endereço de memória.

Variáveis Ponteiros

- ① A linguagem C permite o armazenamento e a manipulação de valores de endereço de memória.
- ② Para cada tipo existente (`int long float double char`), há um tipo ponteiro capaz de armazenar endereços de memória em que existem valores do tipo correspondente armazenados.

Variáveis Ponteiros

- ① A linguagem C permite o armazenamento e a manipulação de valores de endereço de memória.
- ② Para cada tipo existente (`int long float double char`), há um tipo ponteiro capaz de armazenar endereços de memória em que existem valores do tipo correspondente armazenados.
- ③ Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente.

Algumas das razões para utilizar ponteiros são:

- ① Ponteiros fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem.

Algumas das razões para utilizar ponteiros são:

- ① Ponteiros fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem.
- ② Para criar estruturas de dados complexas, como listas encadeadas e árvores binárias, onde uma estrutura de dados deve conter referências sobre outra.

Algumas das razões para utilizar ponteiros são:

- ① Ponteiros fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem.
- ② Para criar estruturas de dados complexas, como listas encadeadas e árvores binárias, onde uma estrutura de dados deve conter referências sobre outra.
- ③ Para comunicar informações sobre a memória, como na função **malloc** que retorna a localização de memória livre através do uso de ponteiro.

Algumas das razões para utilizar ponteiros são:

- ① Ponteiros fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem.
- ② Para criar estruturas de dados complexas, como listas encadeadas e árvores binárias, onde uma estrutura de dados deve conter referências sobre outra.
- ③ Para comunicar informações sobre a memória, como na função **malloc** que retorna a localização de memória livre através do uso de ponteiro.
- ④ Notações de ponteiros compilam mais rapidamente tornando o código mais eficiente.

Algumas das razões para utilizar ponteiros são:

- ① Ponteiros fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem.
- ② Para criar estruturas de dados complexas, como listas encadeadas e árvores binárias, onde uma estrutura de dados deve conter referências sobre outra.
- ③ Para comunicar informações sobre a memória, como na função **malloc** que retorna a localização de memória livre através do uso de ponteiro.
- ④ Notações de ponteiros compilam mais rapidamente tornando o código mais eficiente.
- ⑤ Para manipular matrizes mais facilmente através de movimentação de ponteiros para elas (ou parte delas), em vez de a própria matriz.

Variáveis do Tipos de Ponteiros

- A linguagem C não reserva uma palavra especial para a declaração de ponteiros.

Variáveis do Tipos de Ponteiros

- A linguagem C não reserva uma palavra especial para a declaração de ponteiros.
- As variáveis do tipo ponteiro são declaradas da seguinte forma: *tipo* com os nomes das variáveis precedidos pelo caractere *.

```
1     int *a, *b;  
2
```

Variáveis do Tipos de Ponteiros

- A linguagem C não reserva uma palavra especial para a declaração de ponteiros.
- As variáveis do tipo ponteiro são declaradas da seguinte forma: *tipo* com os nomes das variáveis precedidos pelo caractere *.

```
1   int *a, *b;  
2
```

- A instrução acima declara que ***a** e ***b** são do tipo **int** e que ***a** e ***b** são ponteiros, isto é **a** e **b** contém endereços de variáveis do tipo **int**.

Operadores de Ponteiros

- A linguagem C oferece dois operadores unários para trabalharem com ponteiros.

Operador	significado
&	("endereço de")
*	("conteúdo de")

Operadores de Ponteiros

- O operador & (“endereço de”), aplicado a variáveis, resulta no **endereço da posição da memória** reservada para a variável. Por exemplo,

```
1     a = &b;  
2
```

Operadores de Ponteiros

- O operador & (“endereço de”), aplicado a variáveis, resulta no **endereço da posição da memória** reservada para a variável. Por exemplo,

```
1     a = &b;  
2
```

- coloca em **a** o endereço da memória que contém a variável **b**.

Operadores de Ponteiros

- O operador & (“endereço de”), aplicado a variáveis, resulta no **endereço da posição da memória** reservada para a variável. Por exemplo,

```
1     a = &b;  
2
```

- coloca em **a** o endereço da memória que contém a variável **b**.
- O endereço não tem relação algum com o valor da variável **b**.

Operadores de Ponteiros

- O operador & (“endereço de”), aplicado a variáveis, resulta no **endereço da posição da memória** reservada para a variável. Por exemplo,

```
1     a = &b;  
2
```

- coloca em **a** o endereço da memória que contém a variável **b**.
- O endereço não tem relação algum com o valor da variável **b**.
- Após as declarações as duas variáveis armazenam “lixos”, pois não foram inicializadas.

Operadores de Ponteiros

- O operador * (“conteúdo de”), aplicado a variáveis do tipo ponteiro, acessa o conteúdo do endereço da memória pela variável ponteiro, isto é, devolve o conteúdo da variável apontada pelo operando. Por exemplo:

```
1     a = *b;  
2
```

Operadores de Ponteiros

- O operador * (“conteúdo de”), aplicado a variáveis do tipo ponteiro, acessa o conteúdo do endereço da memória pela variável ponteiro, isto é, devolve o conteúdo da variável apontada pelo operando. Por exemplo:

1 **a = *b;**
2

- Coloca o valor de **b** em **a**, ou seja, **a** recebe o valor que está no endereço **b**.

Atribuição e acessos de endereço

```
1  /* a recebe o valor 5*/
2  a = 5;
3
4  /* p recebe o endereço de a (p aponta para a). */
5  p = &a;
6
7  /* conteúdo de p recebe o valor 10 */
8  *p = 10;
9
```

Atribuição e acessos de endereço

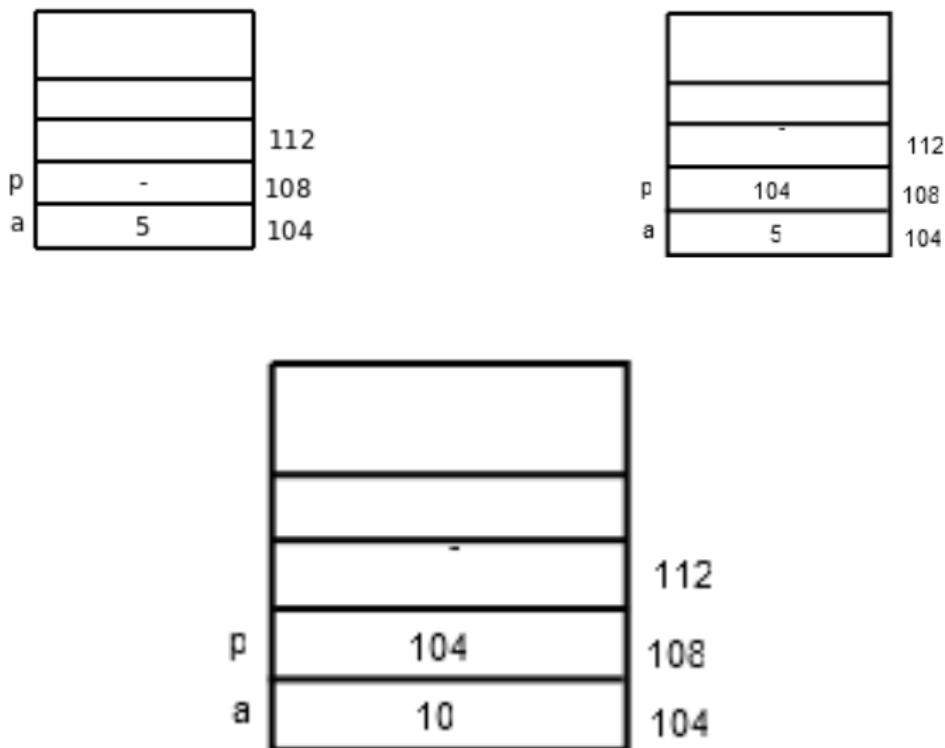


Figura 4: Efeito da atribuição de variáveis na pilha de execução

Atribuição de Ponteiros

- Assim como ocorre com qualquer variável, também podemos atribuir um valor a um ponteiro. Exemplo:

```
1 void main(void) {
2     int a = 10;
3     int *b, *c;
4     b = &a;
5     c = b;
6     printf("%p", c);
7 }
8 }
```

Variável na memória	Endereço na memória
	124
	120
	116
	112
c	104
b	100
a	100

Operações com Ponteiros

- **Incremento/Decremento:**

Operações com Ponteiros

- **Incremento/Decremento:**

- Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento (++).

Operações com Ponteiros

- **Incremento/Decremento:**

- Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento (++).
- Incrementar um ponteiro implica na movimentação do mesmo para o próximo tipo apontado.

Operações com Ponteiros

- **Incremento/Decremento:**

- Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento (`++`).
- Incrementar um ponteiro implica na movimentação do mesmo para o próximo tipo apontado.
- Se `pa` é um ponteiro para inteiro tem no seu conteúdo o valor 200 (um endereço), depois de executada a instrução, `pa++`, o valor de `pa` será o endereço 204 (compiladores onde um inteiro é 4 bytes) e não 201.

Operações com Ponteiros

- **Incremento/Decremento:**

- Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento (`++`).
- Incrementar um ponteiro implica na movimentação do mesmo para o próximo tipo apontado.
- Se `pa` é um ponteiro para inteiro tem no seu conteúdo o valor 200 (um endereço), depois de executada a instrução, `pa++`, o valor de `pa` será o endereço 204 (compiladores onde um inteiro é 4 bytes) e não 201.
- Logo, cuidar com o compilador e arquitetura em questão: 16, 32, 64 ou 128 bits

Operações com Ponteiros

- **Incremento/Decremento:**

- Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento (`++`).
- Incrementar um ponteiro implica na movimentação do mesmo para o próximo tipo apontado.
- Se `pa` é um ponteiro para inteiro tem no seu conteúdo o valor 200 (um endereço), depois de executada a instrução, `pa++`, o valor de `pa` será o endereço 204 (compiladores onde um inteiro é 4 bytes) e não 201.
- Logo, cuidar com o compilador e arquitetura em questão: 16, 32, 64 ou 128 bits
- Com isso, cada vez que incrementamos `pa` ele apontará para o próximo tipo apontado.

Operações com Ponteiros

- **Incremento/Decremento:**

- Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento `(++)`.
- Incrementar um ponteiro implica na movimentação do mesmo para o próximo tipo apontado.
- Se **pa** é um ponteiro para inteiro tem no seu conteúdo o valor 200 (um endereço), depois de executada a instrução, `pa++`, o valor de **pa** será o endereço 204 (compiladores onde um inteiro é 4 bytes) e não 201.
- Logo, cuidar com o compilador e arquitetura em questão: 16, 32, 64 ou 128 bits
- Com isso, cada vez que incrementamos **pa** ele apontará para o próximo tipo apontado.
- O mesmo é verdadeiro para o operador de decremento `(--)`

Operações com Ponteiros

- **Incremento/Decremento:**

- Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento (`++`).
- Incrementar um ponteiro implica na movimentação do mesmo para o próximo tipo apontado.
- Se `pa` é um ponteiro para inteiro tem no seu conteúdo o valor 200 (um endereço), depois de executada a instrução, `pa++`, o valor de `pa` será o endereço 204 (compiladores onde um inteiro é 4 bytes) e não 201.
- Logo, cuidar com o compilador e arquitetura em questão: 16, 32, 64 ou 128 bits
- Com isso, cada vez que incrementamos `pa` ele apontará para o próximo tipo apontado.
- O mesmo é verdadeiro para o operador de decremento (`- -`)
- Cuidar ainda: associatividade (`esq \Leftrightarrow dir`) e precedência (ver manual da linguagem) \Rightarrow fazer os exercícios e ir anotando as respostas

Operações com Ponteiros

- **Comparações entre Ponteiros:**

Operações com Ponteiros

- **Comparações entre Ponteiros:**
 - Ponteiros podem ser comparados:

```
1 if (pa <> pb)
```

Operações com Ponteiros

- **Comparações entre Ponteiros:**

- Ponteiros podem ser comparados:

```
1 if (pa <> pb)
```

- Testes relacionais com \geq , \leq , $>$, $<$ são aceitos entre ponteiros, desde que os operandos sejam ponteiros.

Operações com Ponteiros

- **Comparações entre Ponteiros:**

- Ponteiros podem ser comparados:

```
1 if (pa <> pb)
```

- Testes relacionais com \geq , \leq , $>$, $<$ são aceitos entre ponteiros, desde que os operandos sejam ponteiros.
- O tipo dos operandos devem ser o mesmo, para não obter resultados sem sentido.

Operações com Ponteiros

- **Comparações entre Ponteiros:**

- Ponteiros podem ser comparados:

```
1 if (pa <> pb)
```

- Testes relacionais com \geq , \leq , $>$, $<$ são aceitos entre ponteiros, desde que os operandos sejam ponteiros.
- O tipo dos operandos devem ser o mesmo, para não obter resultados sem sentido.
- Variáveis ponteiros podem ser testadas quanto à igualdade ($==$) ou desigualdade ($!=$) onde os operandos são ponteiros, ou um dos operandos NULL.

Operações com Ponteiros

- Atribuição:

Operações com Ponteiros

- **Atribuição:**

- Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1 pa = &a;
```

Operações com Ponteiros

- **Atribuição:**

- Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1 pa = &a;
```

- **Leitura de valores:**

Operações com Ponteiros

- **Atribuição:**

- Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1 pa = &a;
```

- **Leitura de valores:**

- O operador (*) devolve o valor guardado no endereço apontado.

Operações com Ponteiros

- **Atribuição:**

- Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1 pa = &a;
```

- **Leitura de valores:**

- O operador (*) devolve o valor guardado no endereço apontado.

- **Leitura do endereço do ponteiro:**

Operações com Ponteiros

- **Atribuição:**

- Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1 pa = &a;
```

- **Leitura de valores:**

- O operador (*) devolve o valor guardado no endereço apontado.

- **Leitura do endereço do ponteiro:**

- Os ponteiros variáveis também têm um endereço e um valor.

Operações com Ponteiros

- **Atribuição:**

- Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1 pa = &a;
```

- **Leitura de valores:**

- O operador (*) devolve o valor guardado no endereço apontado.

- **Leitura do endereço do ponteiro:**

- Os ponteiros variáveis também têm um endereço e um valor.
- O operador (&) retorna a posição da memória onde o ponteiro está localizado. Em resumo:

Operações com Ponteiros

- **Atribuição:**

- Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1 pa = &a;
```

- **Leitura de valores:**

- O operador (*) devolve o valor guardado no endereço apontado.

- **Leitura do endereço do ponteiro:**

- Os ponteiros variáveis também têm um endereço e um valor.
- O operador (&) retorna a posição da memória onde o ponteiro está localizado. Em resumo:

- ➊ O nome do ponteiro retorna o endereço para o qual ele aponta.

Operações com Ponteiros

- **Atribuição:**

- Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1 pa = &a;
```

- **Leitura de valores:**

- O operador (*) devolve o valor guardado no endereço apontado.

- **Leitura do endereço do ponteiro:**

- Os ponteiros variáveis também têm um endereço e um valor.
- O operador (&) retorna a posição da memória onde o ponteiro está localizado. Em resumo:

- ➊ O nome do ponteiro retorna o endereço para o qual ele aponta.
- ➋ O operador (&) junto ao nome do ponteiro retorna o endereço do ponteiro.

Operações com Ponteiros

- **Atribuição:**

- Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1 pa = &a;
```

- **Leitura de valores:**

- O operador (*) devolve o valor guardado no endereço apontado.

- **Leitura do endereço do ponteiro:**

- Os ponteiros variáveis também têm um endereço e um valor.
- O operador (&) retorna a posição da memória onde o ponteiro está localizado. Em resumo:

- ➊ O nome do ponteiro retorna o endereço para o qual ele aponta.
- ➋ O operador (&) junto ao nome do ponteiro retorna o endereço do ponteiro.
- ➌ O operador (*) junto ao nome do ponteiro retorna o conteúdo da variável apontada.

Chamadas por valor × referência

- A passagem de argumentos para funções em C são feitas por valor (“chamada por valor”).
- Na passagem de parâmetro por valor a função chamada **não pode alterar** uma variável da função que fez a chamada.
- Sim, a chamada por valor cópia protege o conteúdo
- Mas, muitas vezes a duplicação do valor da variável deve ser evitado, ai precisamos da chamada por referência.
- Uso de ponteiros

Atribuição e acessos de endereço

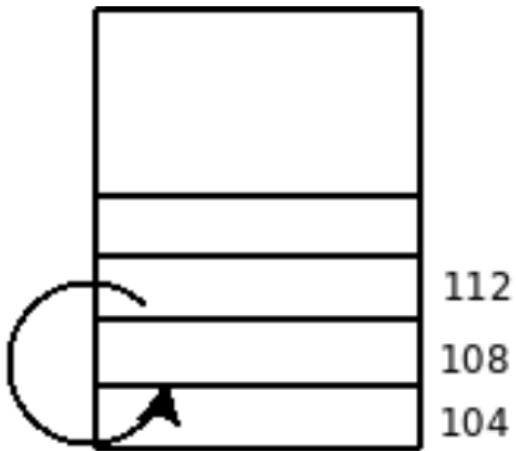


Figura 5: Representação gráfica do valor de um ponteiro

Chamada por referência

```
1 /*Ordena o vetor v de tamanho n, v[0 .. n-1]
2 * em ordem crescente.
3 */
4 void ordenacaoSelecao(int n, int v[]){
5     int i, j;
6     for(i = 0; i < n -1; i++)
7         for(j = i + 1; j < n; j++)
8             if (v[j] < v [i])
9                 troca(v[i],v[j]);
10 }
11
12 void troca(int x, int y) {
13     int tmp =x;
14     x = y;
15     y = tmp;
16 }
```

Chamada por referência

- O código anterior cumpre o seu objetivo?

Chamada por referência

- O código anterior cumpre o seu objetivo?
- Por causa da chamada por valor, a função **troca** não afeta os argumentos **x** e **y** da rotina que chama, ou seja, o código está **ERRADO**.

Chamada por referência

- O código anterior cumpre o seu objetivo?
- Por causa da chamada por valor, a função **troca** não afeta os argumentos **x** e **y** da rotina que chama, ou seja, o código está **ERRADO**.
- Para obter o efeito desejado é necessário passar os endereços dos valores a serem permutados:

Chamada por referência

- O código anterior cumpre o seu objetivo?
- Por causa da chamada por valor, a função **troca** não afeta os argumentos **x** e **y** da rotina que chama, ou seja, o código está **ERRADO**.
- Para obter o efeito desejado é necessário passar os endereços dos valores a serem permutados:
 - **troca(&v[i], &v[j]);**

Chamada por referência

- O código anterior cumpre o seu objetivo?
- Por causa da chamada por valor, a função **troca** não afeta os argumentos **x** e **y** da rotina que chama, ou seja, o código está **ERRADO**.
- Para obter o efeito desejado é necessário passar os endereços dos valores a serem permutados:
 - **troca(&v[i], &v[j]);**
- O que muda na função troca?

Chamada por referência

```
1 /* Ordena o vetor v de tamanho n, v[0 .. n-1]
2  * em ordem crescente.
3 */
4 void ordenacaoSelecao(int n, int v[]){
5     int i, j;
6     for(i = 0; i < n -1; i++)
7         for(j = i + 1; j < n; j++)
8             if (v[j] < v [i])
9                 troca(&v[i],&v[j]);
10 }
11 /*Permuta x e y*/
12 void troca(int *x, int *y) {
13     int tmp = *x;
14     *x = *y;
15     *y = tmp;
16 }
```

Passagem por referência

- No código anterior os argumentos da função **troca** foram declarados como ponteiros.
- Os parâmetros ponteiros da função **troca** são ditos como de **entrada e saída**.
- Dessa forma, qualquer modificação realizada em **troca** fica visível à função que chamou.
- Para que uma função gere o efeito de chamada por referência, os ponteiros devem ser utilizados na declaração dos argumentos e a função chamadora deve mandar endereços como argumentos.

Onde estamos ...

Ponteiros e matrizes

- Em C, há um estreito relacionamento entre ponteiros e matrizes.

Ponteiros e matrizes

- Em C, há um estreito relacionamento entre ponteiros e matrizes.
- O compilador transforma matrizes em ponteiros após a compilação do código.

Ponteiros e matrizes

- Em C, há um estreito relacionamento entre ponteiros e matrizes.
- O compilador transforma matrizes em ponteiros após a compilação do código.
- Qualquer operação que possa ser feita com índices de uma matriz pode ser feita com ponteiros.

Ponteiros e matrizes

- Em C, há um estreito relacionamento entre ponteiros e matrizes.
- O compilador transforma matrizes em ponteiros após a compilação do código.
- Qualquer operação que possa ser feita com índices de uma matriz pode ser feita com ponteiros.
- O nome de uma matriz é um endereço, ou seja, um ponteiro.

Ponteiros e matrizes

- Em C, há um estreito relacionamento entre ponteiros e matrizes.
- O compilador transforma matrizes em ponteiros após a compilação do código.
- Qualquer operação que possa ser feita com índices de uma matriz pode ser feita com ponteiros.
- O nome de uma matriz é um endereço, ou seja, um ponteiro.
- Ponteiros e matrizes são idênticos na maneira de acessar a memória.

Ponteiros e matrizes

- Em C, há um estreito relacionamento entre ponteiros e matrizes.
- O compilador transforma matrizes em ponteiros após a compilação do código.
- Qualquer operação que possa ser feita com índices de uma matriz pode ser feita com ponteiros.
- O nome de uma matriz é um endereço, ou seja, um ponteiro.
- Ponteiros e matrizes são idênticos na maneira de acessar a memória.
- Um **ponteiro variável** é um endereço onde é armazenado um outro endereço.

Exemplos de programas com matrizes

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int strtam(char    *  );
5
6 int main(void)
7 {   char vetor[] = "ABCDEFG" ;
8     //  char *lista = "Ola como vai?"; ... HUM ...
9     char *pt;
10    pt = vetor; //mais saudavel => pt = &vetor[0];
11    // valido para char apenas
12    system("clear");
13    printf("O tamanho de \"%s\" e %d caracteres.\n", vetor, strtam( pt
14    printf("\n ... Acabou ....");
15    return 1;
16 }
17
18 int strtam(char *s){
19     int tam=0;
20     //while(*(s + tam++) != '\0');
21     while(*s != '\0')
22     {
23         tam++;
24             s++;
25     }
26     return tam; //tam-1; --> \0
```

Exemplos de programas com matrizes

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int strtam(char    *  );
5
6 int main(void)
7 {   char vetor[] = "ABCDEFG" ;
8     //  char *lista = "Ola como vai?"; ... HUM ...
9     char *pt;
10    pt = vetor; //mais saudavel => pt = &vetor[0];
11    // valido para char apenas
12    system("clear");
13    printf("O tamanho de \"%s\" e %d caracteres.\n", vetor, strtam( pt
14    printf("\n ... Acabou ....");
15    return 1;
16 }
17
18 int strtam(char *s){
19     int tam=0;
20     //while(*(s + tam++) != '\0');
21     while(*s != '\0')
22     {
23         tam++;
24             s++;
25     }
26     return tam; //tam-1; --> \0
```

Exemplos de programas com matrizes

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main(void)
{
5     float matriz [50][50];
6     float *pt_float;
7     int count;
8     pt_float = matriz[0]; // OU &matriz[0];
9     // mas nao pt_float = matriz;
10    for (count=0;count < 2500 ; count++)
11    {
12        *pt_float = 0.0;
13        pt_float++;
14    }
15    printf("\n ... Acabou ....\n");
16    return 1;
17}
18}
```

Exemplos de programas com matrizes

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main(void)
{
    float matriz [50][50];
    float *pt_float;
    int count;
    pt_float = matriz[0]; // OU &matriz[0];
// mas nao pt_float = matriz;
    for (count=0;count < 2500 ; count++)
    {
        *pt_float = 0.0;
        pt_float++;
    }
    printf("\n ... Acabou ....\n");
    return 1;
}
```

Pergunta de aluno do laboratório

Exemplos de programas com matrizes

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main(void)
5 {    int matriz [] [3] = {{9, 8, 7},
6                           {99, 88, 77},
7                           {1, 2, 3}};
8     int *pt_int;
9     int count;
10    pt_int = &matriz [0] [0]; // tem que indicar
11                      // a celula
12    for (count=0; count<9; count++)
13    {
14        printf("%d : ", *pt_int );
15        pt_int++;
16    }
17    printf("\n ... Acabou ....\n");
18    return 1;
19 }
```

Dúvida de aula ... qual a saída do código acima?

Exemplos de programas com matrizes

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main(void)
5 {    int matriz [] [3] = {{9, 8, 7},
6                           {99, 88, 77},
7                           {1, 2, 3}};
8     int *pt_int;
9     int count;
10    pt_int = &matriz [0] [0]; // tem que indicar
11                      // a celula
12    for (count=0; count<9; count++)
13    {
14        printf("%d : ", *pt_int );
15        pt_int++;
16    }
17    printf("\n ... Acabou ....\n");
18    return 1;
19 }
```

Dúvida de aula ... qual a saída do código acima?

```
1 [ccs@gerzat ponteiros]$ ./a.out
2 9 8 7 99 88 77 1 2 3
3 ... Acabou ....
```

Exemplo de programas com matrizes

```
1 /*imprime os valores da matriz*/
2 main(){
3     int nums [5] = {100,200,90,20,10};
4     int d;
5     for(d = 0; d < 5; d++)
6         printf("%d\n", nums[d]);
7 }
8
9 /*usa ponteiros para imprimir os valores da matriz*/
10 main(){
11     int nums [5] = {100,200,90,20,10};
12     int d;
13     for(d = 0; d < 5; d++)
14         printf("%d\n", *(nums + d));
15 }
```

Ponteiros e matrizes

- O segundo programa é idêntico ao primeiro, exceto pela expressão `*(nums + d)`.

Ponteiros e matrizes

- O segundo programa é idêntico ao primeiro, exceto pela expressão `*(nums + d)`.
- O efeito de `*(nums + d)` é o mesmo que `nums[d]`.

Ponteiros e matrizes

- O segundo programa é idêntico ao primeiro, exceto pela expressão `*(nums + d)`.
- O efeito de `*(nums + d)` é o mesmo que `nums[d]`.
- A expressão `*(nums + d)` é o endereço do elemento de índice `d` da matriz.

Ponteiros e matrizes

- O segundo programa é idêntico ao primeiro, exceto pela expressão `*(nums + d)`.
- O efeito de `*(nums + d)` é o mesmo que `nums[d]`.
- A expressão `*(nums + d)` é o endereço do elemento de índice **d** da matriz.
- Se cada elemento da matriz é um inteiro e $d = 3$, então serão pulados 6 bytes para atingir o elemento de índice 3.

Ponteiros e matrizes

- O segundo programa é idêntico ao primeiro, exceto pela expressão **$*(\text{nums} + d)$** .
- O efeito de **$*(\text{nums} + d)$** é o mesmo que **$\text{nums}[d]$** .
- A expressão **$*(\text{nums} + d)$** é o endereço do elemento de índice **d** da matriz.
- Se cada elemento da matriz é um inteiro e $d = 3$, então serão pulados 6 bytes para atingir o elemento de índice 3.
- Assim, a expressão **$*(\text{nums} + d)$** não significa avançar 3 *bytes*, além *nums* e sim 3 elementos da matriz.

Cuidados

```
int vetor[10];
int *ponteiro, i;
ponteiro = &i;

/* as operacoes a seguir sao invalidas */
vetor = vetor + 2; /* ERRADO: vetor nao eh variavel */
vetor++;           /* ERRADO: vetor nao eh variavel */
vetor = ponteiro;  /* ERRADO: vetor nao eh variavel */

/* as operacoes abaixo sao validas */

ponteiro = vetor;   /* CERTO: ponteiro eh variavel */
ponteiro = vetor+2; /* CERTO: ponteiro eh variavel */
```

Indireção múltipla

O que é *indireção* múltipla?

- *Indireção múltipla* ou **ponteiro de ponteiros** é quando temos um ponteiro apontando para outro ponteiro que aponta para o valor final.

Indireção múltipla

O que é *indireção* múltipla?

- *Indireção múltipla* ou **ponteiro de ponteiros** é quando temos um ponteiro apontando para outro ponteiro que aponta para o valor final.
- O valor normal de um ponteiro é o endereço de uma variável que contém o valor desejado.

Indireção múltipla

O que é *indireção* múltipla?

- *Indireção múltipla* ou **ponteiro de ponteiros** é quando temos um ponteiro apontando para outro ponteiro que aponta para o valor final.
- O valor normal de um ponteiro é o endereço de uma variável que contém o valor desejado.
- No caso de um ponteiro para um ponteiro, o primeiro contém o endereço do segundo que aponta para a variável que contém o valor desejado.

Indireção múltipla

O que é *indireção múltipla*?

- *Indireção múltipla* ou **ponteiro de ponteiros** é quando temos um ponteiro apontando para outro ponteiro que aponta para o valor final.
- O valor normal de um ponteiro é o endereço de uma variável que contém o valor desejado.
- No caso de um ponteiro para um ponteiro, o primeiro contém o endereço do segundo que aponta para a variável que contém o valor desejado.
- A *indireção múltipla* pode ser levada a qualquer dimensão desejada.

Indireção múltipla



Figura 6: Indireção simples

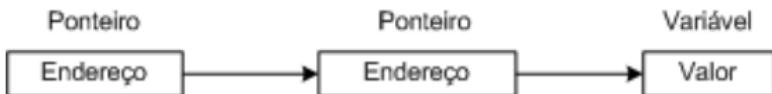


Figura 7: Indireção múltipla

Indireção múltipla

Declaração

- Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal.

Indireção múltipla

Declaração

- Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal.
- A declaração de ponteiro de ponteiro é realizada colocando-se um * adicional na frente do nome da variável. Exemplo:

```
1 int **a;  
2 float **b;
```

Indireção múltipla

Declaração

- Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal.
- A declaração de ponteiro de ponteiro é realizada colocando-se um * adicional na frente do nome da variável. Exemplo:

```
1 int **a;  
2 float **b;
```

- No primeiro exemplo, temos a declaração de um ponteiro para um ponteiro de inteiro (**int**).

Indireção múltipla

Declaração

- Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal.
- A declaração de ponteiro de ponteiro é realizada colocando-se um * adicional na frente do nome da variável. Exemplo:

```
1 int **a;  
2 float **b;
```

- No primeiro exemplo, temos a declaração de um ponteiro para um ponteiro de inteiro (**int**).
- É importante salientar que **a** não é um ponteiro para um número inteiro, mas um ponteiro para um ponteiro inteiro.

Exemplo de indireção múltipla

```
1 int main(void) {
2     int x, *a, **b;
3     x = 4;
4     a = &x;
5     b = &a;
6     printf("%d ", **b);
7 }
```

Onde estamos ...

Ponteiros no Retorno de Funções

- Este caso está presente em quase todas funções da linguagem C

Ponteiros no Retorno de Funções

- Este caso está presente em quase todas funções da linguagem C
- Como ponteiros e vetores são praticamente a mesma coisa, a existência de funções que retornam um ponteiro, é presente na linguagem C

Ponteiros no Retorno de Funções

- Este caso está presente em quase todas funções da linguagem C
- Como ponteiros e vetores são praticamente a mesma coisa, a existência de funções que retornam um ponteiro, é presente na linguagem C
- Sim, pois ai não a cópia de valor no retorno, e sim o ponteiro que indica tal endereço de início

Ponteiros no Retorno de Funções

- Este caso está presente em quase todas funções da linguagem C
- Como ponteiros e vetores são praticamente a mesma coisa, a existência de funções que retornam um ponteiro, é presente na linguagem C
- Sim, pois ai não a cópia de valor no retorno, e sim o ponteiro que indica tal endereço de início
- Funções de vetores de caracteres: `strcmp`, etc

Exemplo I

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 // funcao que retorna via ponteiro
4 float * f_media(int , int);
5 int main(void)
6 {
7     float resultado;
8     float *pt_float;
9
10    pt_float = f_media(3,4);
11    resultado = * pt_float; // apenas lendo
12    // onde o ponteiro esta apontando : a funcao
13    // system("clear"); ... cuidar ... causou danos ...
14    printf("VALORES: %.2f : %.2f", *pt_float, resultado );
15    printf("\nENDERECOS: %x : %x : %x", pt_float, &pt_float,
16                                     &resultado );
17    printf("\nENDERECOS: %x ", & f_media );
18    printf("\n ... Acabou ....\n");
19    return 1;
20 }
```

Exemplo II

```
21
22 //uma atribuicao de conteudo entre ponteiros: float
23 float * f_media(int a, int b)
24 {
25     float res = (a+b)*0.5; // media 2 valores
26     float *pt_aux = & res;
27     printf("\nNA FUNCAO: %x : %x\n", pt_aux, &pt_aux );
28     return pt_aux;
29 }
```

Onde estamos ...

Ponteiro para Funções

- Um recurso muito poderoso da linguagem C, é o ponteiro para função.

Ponteiro para Funções

- Um recurso muito poderoso da linguagem C, é o ponteiro para função.
- Embora uma função não seja uma variável, ela tem uma posição física na memória que pode ser atribuída a um ponteiro. Exemplo:

```
1 int main(){  
2     int (*func)(const char*, ...);  
3     func = printf;  
4     (*func)("%d\n", 1);  
5 }
```

Ponteiro para Funções

- Um recurso muito poderoso da linguagem C, é o ponteiro para função.
- Embora uma função não seja uma variável, ela tem uma posição física na memória que pode ser atribuída a um ponteiro. Exemplo:

```
1 int main(){  
2     int (*func)(const char*, ...);  
3     func = printf;  
4     (*func)("%d\n", 1);  
5 }
```

- No código acima, a instrução

```
1 int (*func)(const char*, ...);
```

declara uma função do tipo **int**. Nesse exemplo, estamos declarando que **func** é uma função do tipo ponteiro para inteiro que aponta para o endereço da função *printf*, através da instrução:

```
1 func = printf;
```

Ponteiro para Funções

- A declaração anterior é possível e válida, porque quando compilamos uma função, o código-fonte é transformado em código-objeto e um ponto de entrada é estabelecido.

Ponteiro para Funções

- A declaração anterior é possível e válida, porque quando compilamos uma função, o código-fonte é transformado em código-objeto e um ponto de entrada é estabelecido.
- Quando é feita uma chamada à função, enquanto o programa está sendo executado, é efetuado uma chamada em linguagem de máquina para esse ponto de entrada.

Ponteiro para Funções

- A declaração anterior é possível e válida, porque quando compilamos uma função, o código-fonte é transformado em código-objeto e um ponto de entrada é estabelecido.
- Quando é feita uma chamada à função, enquanto o programa está sendo executado, é efetuado uma chamada em linguagem de máquina para esse ponto de entrada.
- Portanto, se um ponteiro contém o endereço do ponto de entrada de uma função, então ele pode ser usado para chamar essa função.

Ponteiro para Funções

- A declaração anterior é possível e válida, porque quando compilamos uma função, o código-fonte é transformado em código-objeto e um ponto de entrada é estabelecido.
- Quando é feita uma chamada à função, enquanto o programa está sendo executado, é efetuado uma chamada em linguagem de máquina para esse ponto de entrada.
- Portanto, se um ponteiro contém o endereço do ponto de entrada de uma função, então ele pode ser usado para chamar essa função.
- Resumindo: `func` pode funcionar como um sinônimo para `printf`

Ponteiro para Funções

- A declaração anterior é possível e válida, porque quando compilamos uma função, o código-fonte é transformado em código-objeto e um ponto de entrada é estabelecido.
- Quando é feita uma chamada à função, enquanto o programa está sendo executado, é efetuado uma chamada em linguagem de máquina para esse ponto de entrada.
- Portanto, se um ponteiro contém o endereço do ponto de entrada de uma função, então ele pode ser usado para chamar essa função.
- Resumindo: `func` pode funcionar como um sinônimo para `printf`
- Dessa forma, ao executarmos a instrução:

```
1 (*func) ("\\%d\\n", 1);
```

estamos executando a função `printf`, logo, será apresentado na saída o valor 1.

Ponteiro para Funções

- O endereço de uma função é obtido usando o nome da função sem parênteses ou argumentos.

Ponteiro para Funções

- O endereço de uma função é obtido usando o nome da função sem parênteses ou argumentos.
- Observe que não colocamos parênteses junto ao nome da função. Se eles estiverem presentes como em:

```
1 func = printf();
```

, estariámos atribuindo a *func* o valor retornado pela função e não o endereço dela.

Ponteiro para Funções

- O endereço de uma função é obtido usando o nome da função sem parênteses ou argumentos.
- Observe que não colocamos parênteses junto ao nome da função. Se eles estiverem presentes como em:

```
1 func = printf();
```

, estaríamos atribuindo a *func* o valor retornado pela função e não o endereço dela.

- O nome de uma função desacompanhado de parênteses é o endereço dela.

Ponteiro para Funções

- Nesse exemplo, nada é obtido e bastante confusão é introduzida. Porém, há momentos em que é vantajoso passar funções arbitrárias para procedimentos, ou manter uma matriz de funções.

Ponteiro para Funções

- Nesse exemplo, nada é obtido e bastante confusão é introduzida. Porém, há momentos em que é vantajoso passar funções arbitrárias para procedimentos, ou manter uma matriz de funções.
- Por exemplo, escolher o melhor algoritmo para resolver um problema.

Ponteiro para Funções

- Nesse exemplo, nada é obtido e bastante confusão é introduzida. Porém, há momentos em que é vantajoso passar funções arbitrárias para procedimentos, ou manter uma matriz de funções.
- Por exemplo, escolher o melhor algoritmo para resolver um problema.
- Um primeiro passo para o conceito de *funções genéricas* (ou *anônimas*)

Ponteiro para Funções

```
1 /**
2  * Ordena o vetor v de tamanho n, utilizando
3  * o algoritmo de ordenacao implementado pela
4  * funcao: algOrdenacao.
5 */
6 void ordenar(int v[], int n,
7     void (*algOrdenacao)(int v[], int n)){
8     (*algOrdenacao)(v,n);
9 }
```

- O exemplo acima é um caso clássico discutido na literatura, leia sobre ele!
- Vamos há um exemplo original e completo para fixarmos o aprendizado

Exemplo completo: ponteiro para funções I

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 // funções a serem apontada via ponteiros
5 float f_div(int a, int b);
6 int f_dist(char a, char b);
7
8 int main(void)
9 {
10     char a = 'a' , z = 'z';
11     int x = 3 , y = 4 ;
12     // cria ponteiros para funções ... veja tipagem
13     float (* pt_divisao) (int, int);
14     int   (* pt_distancia) (char, char) ;
15     /* inicializa ponteiros as funções */
16     pt_divisao = f_div ;
17     pt_distancia = f_dist; /* inicializa ponteiro */
18
19     // chamando as funções função ... retornando algo
20     float R_1 = (*pt_divisao) (x, y);    /* invoca função */
```

Exemplo completo: ponteiro para funções II

```
21     int R_2 = (*pt_distancia) (a, z); /* invoca função */
22
23     system("clear"); // OK ... mas cuidar
24
25     printf("SAIDAS: %6.2f : %d", R_1 , R_2 );
26     printf("\nENDERECOS: %x = %x ", pt_divisao, f_div );
27     printf("\nENDERECOS: %x : %x ", &pt_divisao, &f_div );
28     printf("\nENDERECOS: %x = %x ", pt_distancia, f_dist );
29     printf("\nENDERECOS: %x : %x ", &pt_distancia, &f_dist );
30     printf("\n ... Acabou ....\n");
31
32 }
33 // AS FUNCOES
34 float f_div(int a, int b)
35 {
36     //printf("\n %d .. %d \n", a , b);
37     float resp = ((float)a) / b;
38     return ( resp );
39
40 int f_dist(char a, char b)
41 {
```

Exemplo completo: ponteiro para funções III

```
42     return (b - a);  
43 }
```

Quanto há uma saída:

Uma saída:

SAIDAS: 0.75 : 25

ENDERECOS: 632178e3 = 632178e3

ENDERECOS: 9f06eac8 : 632178e3

ENDERECOS: 6321792e = 6321792e

ENDERECOS: 9f06ead0 : 6321792e

... Acabou

Ponteiro para Funções

- Em resumo podemos:
 - ① Passar endereços de variáveis como parâmetros
 - ② Passar ponteiros como parâmetros
 - ③ Retornar um ponteiro de uma função
 - ④ Declarar um ponteiro para uma função
 - ⑤ Atribuir o endereço de uma função a um ponteiro
 - ⑥ Chamar a função através do ponteiro para ela
- Mas não podemos:
 - ① Incrementar ou decrementar ponteiros para funções.
 - ② Incrementar ou decrementar nomes de funções.

Onde estamos ...

Alocação Dinâmica

- Na declaração de um vetor visto até agora é necessário conhecer e informar o número de elementos do vetor.

Alocação Dinâmica

- Na declaração de um vetor visto até agora é necessário conhecer e informar o número de elementos do vetor.
- Esse pré-dimensionamento é um fator limitante, pois, nos obriga a conhecer de antemão a quantidade de elementos do vetor.

Alocação Dinâmica

- Na declaração de um vetor visto até agora é necessário conhecer e informar o número de elementos do vetor.
- Esse pré-dimensionamento é um fator limitante, pois, nos obriga a conhecer de antemão a quantidade de elementos do vetor.
- Uma solução é dimensionar o vetor com um número muito alto, para não termos limitações no momento de utilização do programa.

Alocação Dinâmica

- Na declaração de um vetor visto até agora é necessário conhecer e informar o número de elementos do vetor.
- Esse pré-dimensionamento é um fator limitante, pois, nos obriga a conhecer de antemão a quantidade de elementos do vetor.
- Uma solução é dimensionar o vetor com um número muito alto, para não termos limitações no momento de utilização do programa.
- Essa solução leva a um desperdício de memória.

Alocação Dinâmica

- Na declaração de um vetor visto até agora é necessário conhecer e informar o número de elementos do vetor.
- Esse pré-dimensionamento é um fator limitante, pois, nos obriga a conhecer de antemão a quantidade de elementos do vetor.
- Uma solução é dimensionar o vetor com um número muito alto, para não termos limitações no momento de utilização do programa.
- Essa solução leva a um desperdício de memória.
- Qual a solução?

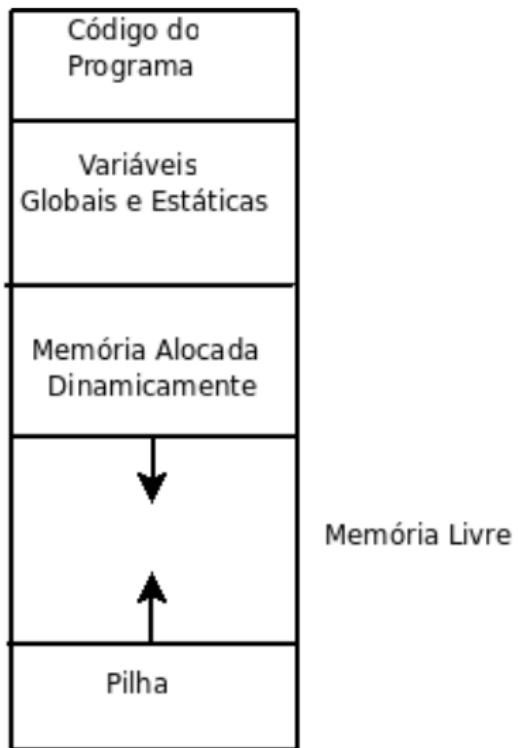
Alocação Dinâmica

- Na declaração de um vetor visto até agora é necessário conhecer e informar o número de elementos do vetor.
- Esse pré-dimensionamento é um fator limitante, pois, nos obriga a conhecer de antemão a quantidade de elementos do vetor.
- Uma solução é dimensionar o vetor com um número muito alto, para não termos limitações no momento de utilização do programa.
- Essa solução leva a um desperdício de memória.
- Qual a solução?
 - Utilizar alocação dinâmica.

Alocação Dinâmica

- Na declaração de um vetor visto até agora é necessário conhecer e informar o número de elementos do vetor.
- Esse pré-dimensionamento é um fator limitante, pois, nos obriga a conhecer de antemão a quantidade de elementos do vetor.
- Uma solução é dimensionar o vetor com um número muito alto, para não termos limitações no momento de utilização do programa.
- Essa solução leva a um desperdício de memória.
- Qual a solução?
 - Utilizar alocação dinâmica.
 - Isto é, requisitar espaços de memória em tempo de execução.

Uso da memória



Funções para alocação dinâmica de memória

- A função básica para alocar memória é **malloc**.

Funções para alocação dinâmica de memória

- A função básica para alocar memória é **malloc**.
- A função **malloc** recebe como parâmetro o número de *bytes* que se deseja alocar e retorna o endereço inicial da área de memória alocada.

Funções para alocação dinâmica de memória

- A função básica para alocar memória é **malloc**.
- A função **malloc** recebe como parâmetro o número de *bytes* que se deseja alocar e retorna o endereço inicial da área de memória alocada.
- O código seguinte realiza a alocação dinâmica de um vetor de inteiros com 10 elementos.

```
1 int *v;  
2 v = malloc(10*4);
```

Funções para alocação dinâmica de memória

- A função básica para alocar memória é **malloc**.
- A função **malloc** recebe como parâmetro o número de *bytes* que se deseja alocar e retorna o endereço inicial da área de memória alocada.
- O código seguinte realiza a alocação dinâmica de um vetor de inteiros com 10 elementos.

```
1  int *v;  
2  v = malloc(10*4);
```

- Após a execução, se a alocação for bem-sucedida, **v** armazenará o endereço inicial de uma área contínua de memória suficiente para armazenar 10 valores inteiros.

Funções para alocação dinâmica de memória

- A função básica para alocar memória é **malloc**.
- A função **malloc** recebe como parâmetro o número de *bytes* que se deseja alocar e retorna o endereço inicial da área de memória alocada.
- O código seguinte realiza a alocação dinâmica de um vetor de inteiros com 10 elementos.

```
1  int *v;  
2  v = malloc(10*4);
```

- Após a execução, se a alocação for bem-sucedida, **v** armazenará o endereço inicial de uma área contínua de memória suficiente para armazenar 10 valores inteiros.
- Podemos tratar **v** como tratamos um vetor declarado estaticamente.

Funções para alocação dinâmica de memória

- A função básica para alocar memória é **malloc**.
- A função **malloc** recebe como parâmetro o número de *bytes* que se deseja alocar e retorna o endereço inicial da área de memória alocada.
- O código seguinte realiza a alocação dinâmica de um vetor de inteiros com 10 elementos.

```
1  int *v;  
2  v = malloc(10*4);
```

- Após a execução, se a alocação for bem-sucedida, **v** armazenará o endereço inicial de uma área contínua de memória suficiente para armazenar 10 valores inteiros.
- Podemos tratar **v** como tratamos um vetor declarado estaticamente.
- Para ficarmos independente de compilador e máquinas, usamos o operador **sizeof()**.

```
1  int *v;  
2  v = malloc(10*sizeof(int));
```

Funções para alocação dinâmica de memória

- A função **malloc** pode ser utilizada para alocar espaço para armazenar valores de qualquer tipo.

Funções para alocação dinâmica de memória

- A função **malloc** pode ser utilizada para alocar espaço para armazenar valores de qualquer tipo.
- O retorno da função **malloc** é um ponteiro genérico, para um tipo qualquer, representado por **void***, que pode ser convertido para o tipo apropriado da atribuição.

Funções para alocação dinâmica de memória

- A função **malloc** pode ser utilizada para alocar espaço para armazenar valores de qualquer tipo.
- O retorno da função **malloc** é um ponteiro genérico, para um tipo qualquer, representado por **void***, que pode ser convertido para o tipo apropriado da atribuição.
- É comum fazer a conversão explicitamente, realizando um **cast** para o tipo correto. Exemplo:

```
1 int *v;  
2 v = (int*) malloc(10 * sizeof(int));
```

Funções para alocação dinâmica de memória

- A função **malloc** pode ser utilizada para alocar espaço para armazenar valores de qualquer tipo.
- O retorno da função **malloc** é um ponteiro genérico, para um tipo qualquer, representado por **void***, que pode ser convertido para o tipo apropriado da atribuição.
- É comum fazer a conversão explicitamente, realizando um **cast** para o tipo correto. Exemplo:

```
1 int *v;
2 v = (int*) malloc(10 * sizeof(int));
```

- Se porventura, não houver espaço livre suficiente para realizar a alocação, a função retorna um endereço nulo, representado por **NULL**, definido em *stdlib.h*.

Funções para alocação dinâmica de memória

- A função **malloc** pode ser utilizada para alocar espaço para armazenar valores de qualquer tipo.
- O retorno da função **malloc** é um ponteiro genérico, para um tipo qualquer, representado por **void***, que pode ser convertido para o tipo apropriado da atribuição.
- É comum fazer a conversão explicitamente, realizando um **cast** para o tipo correto. Exemplo:

```
1 int *v;
2 v = (int*) malloc(10 * sizeof(int));
```

- Se porventura, não houver espaço livre suficiente para realizar a alocação, a função retorna um endereço nulo, representado por **NULL**, definido em *stdlib.h*.
- Podemos verificar se a alocação foi realizada adequadamente, testando o retorno da função **malloc**. Exemplo:

```
1 v = (int*) malloc(10 * sizeof(int));
2 if (v == NULL)
3     printf("Memoria insuficiente.\n");
```

Funções para alocação dinâmica de memória

- Para liberar um espaço de memória alocado dinamicamente, usamos a função **free()**.

Funções para alocação dinâmica de memória

- Para liberar um espaço de memória alocado dinamicamente, usamos a função **free()**.
- A função **free()** recebe como parâmetro o ponteiro da memória a ser liberado.

```
1 free(v);
```

Funções para alocação dinâmica de memória

- Para liberar um espaço de memória alocado dinamicamente, usamos a função **free()**.
- A função **free()** recebe como parâmetro o ponteiro da memória a ser liberado.

```
1 free(v);
```

- Só podemos passar para a função **free()** um endereço de memória que tenha sido alocado dinamicamente.

Funções para alocação dinâmica de memória

- Para liberar um espaço de memória alocado dinamicamente, usamos a função **free()**.
- A função **free()** recebe como parâmetro o ponteiro da memória a ser liberado.

```
1 free(v);
```

- Só podemos passar para a função **free()** um endereço de memória que tenha sido alocado dinamicamente.
- Não podemos acessar o espaço de memória depois de liberado.

Funções para alocação dinâmica de memória

Função	Descrição
malloc(<qtd. bytes>)	Aloca uma área da memória e retorna a referência para o endereço inicial, se existir memória disponível. Caso contrário, retorna NULL .
sizeof(<tipo>)	Retorna a quantidade de memória necessária para para alocar um determinado tipo.
free(<variável>)	Libera o espaço de memória ocupado por uma variável alocada dinamicamente.

Onde estamos ...

Exemplo 01 completo: AD – *malloc* I

```
1  /*
2  malloc() : Allocates requested size of bytes and returns a pointer fir
3  calloc() : Allocates space for an array elements, initializes to zero
4          (the block initializes the allocates memory to all bits zer
5  free(): deallocate the previously allocated space
6  realloc(): Change the size of previously allocated space
7  */
8 #include <stdio.h>
9 #include <stdlib.h>
10
11 int main()
12 {
13     int num, i, *ptr, sum = 0;
14
15     printf("Entre numero de elementos: ");
16     scanf("%d", &num);
17
18     ptr = (int*) malloc(num * sizeof(int));
19     //memoria alocada usando malloc
20     if(ptr == NULL)
```

Exemplo 01 completo: AD – *malloc* II

```
21 {
22     printf("\n Erro! Memoria NAO alocada.\n");
23     exit(0);
24 }
25
26 printf("Lendo o vetor de elementos: ");
27 for(i = 0; i < num; ++i)
28 {
29     scanf("%d", ptr + i);
30     sum += *(ptr + i);
31 }
32
33 printf("\nSOMA FINAL = %d\n", sum);
34 free(ptr);
35 return 0;
36 }
```

Uma saída:

- Laboratório ⇒ você
- Para o exemplo acima, gere um arquivo texto com 1000 inteiros, por exemplo
- Modifique o exemplo do professor, digamos, calcule a média no lugar de uma simples soma
- Crie um ponteiro para função média
- Confira onde foram alocadas estas variáveis
- Verifique os resultados
- Valide o seu aprendizado com o texto acima

Exemplo 01 completo: AD – *calloc* I

```
1  /*
2  calloc() : Allocates space for an array elements, initializes to zero
3 */
4
5 #include <stdio.h>
6 #include <stdlib.h>
7
8 int main()
9 {
10     int num, i, *ptr, sum = 0;
11
12     printf("Entre numero de elementos: ");
13     scanf("%d", &num);
14     // AQUI EH UM VETOR DE N POSICOES ... pequena diferenca
15     ptr = (int*) calloc(num, sizeof(int));
16
17     if(ptr == NULL)
18     {
19         printf("\n Erro! Memoria NAO alocada.\n");
20         exit(0);
```

Exemplo 01 completo: AD – *calloc* II

```
21 }  
22  
23     printf("Lendo o vetor de elementos: ");  
24     for(i = 0; i < num; ++i)  
25     {  
26         scanf("%d", ptr + i);  
27         sum += *(ptr + i);  
28     }  
29  
30     printf("\nSOMA FINAL = %d\n", sum);  
31     free(ptr);  
32     return 0;  
33 }
```

Uma saída:

- Laboratório ⇒ você
- Para o exemplo acima, gere um arquivo texto com 1000 inteiros, por exemplo
- Modifique o exemplo do professor, digamos, calcule a média no lugar de uma simples soma
- Crie um ponteiro para função média
- Confira onde foram alocadas estas variáveis
- Verifique os resultados
- Valide o seu aprendizado com o texto acima

Exemplo 01 completo: AD – *realloc* I

```
1  /*
2  realloc() : Change the size of previously allocated space
3  */
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main()
8 {
9     int *ptr, i , n1, n2;
10    printf("\n Entre TAM do array -> N1: ");
11    scanf("%d", &n1);
12    // Aloca sequencialmente ....
13    ptr = (int*) malloc(n1 * sizeof(int));
14
15    printf("Enderecos alocados na memoria: ");
16    for(i = 0; i < n1; ++i)
17        printf("\n %u\t %d ",ptr + i, *(ptr + i));
18
19    printf("\nNUM NOVO (N2) TAMANHO DE ARRAY (>,=,<): ");
20    scanf("%d", &n2);
```

Exemplo 01 completo: AD – *realloc* II

```
21 // Aloca, reaproveitando o inicio do anterior n1
22 ptr = (int*) realloc(ptr, n2); // cuidar com cast (int*)
23 for(i = 0; i < n2; ++i)
24     printf("\n %u\t %d ",ptr + i, *(ptr + i));
25
26 printf("\n N1: %d\t N2: %d\n", n1, n2);
27
28 return 0;
29 }
```

Uma saída:

- Laboratório ⇒ você
- Para o exemplo acima, gere um arquivo texto com 1000 inteiros, por exemplo
- Modifique o exemplo do professor, digamos, calcule a média no lugar de uma simples soma
- Crie um ponteiro para função média
- Confira onde foram alocadas estas variáveis
- Verifique os resultados
- Valide o seu aprendizado com o texto acima

Onde estamos ...

Capítulo 03 – Pilhas

Pontos fundamentais a serem cobertos:

- ① Contexto e motivação
- ② Definição
- ③ Implementações
- ④ Exercícios



Introdução

- Uma das estruturas de dados mais simples
- Embora seja uma das estrutura de dados mais utilizadas em programação
- A pilha se *fortalece* quando combinada dentro de outras estruturas
 - Uso de pilhas na sequência de visita a nós de uma árvore
 - Dentro de outras estruturas como filas (depois)
- Há uma metáfora emprestada do mundo real, que a computação utiliza pilhas para resolver muitos problemas de forma simplificada.

Aplicações

Alguns exercícios são clássicos (e devemos implementá-los):

- Balanceamento de símbolos. Exemplo: ([aaa])
- Conversão da notação infixa para pós-fixa
- Conversão da notação infixa para in-fixa
- Avaliação de uma expressão pós-fixa. Exemplo: 2 3 +
- Implementações de chamadas de funções (inclusive as chamadas recursivas de funções)
- Armazenamento de páginas visitadas no navegador em uma dada janela (botão **back**)
- Sequência de comandos de um editor de texto, e depois aplique o *undo*, ou **crtl-z**
- Casamento de *tags* in HTML e XML
- As teclas ↑ e ↓ na console ou terminal do Linux, duas pilhas neste caso!

Definição

Definição

Um conjunto ordenado de itens no qual novos itens podem ser inseridos e a partir do qual podem ser eliminados em uma extremidade denominada **topo** da pilha.

Definição

Definição

Um conjunto ordenado de itens no qual novos itens podem ser inseridos e a partir do qual podem ser eliminados em uma extremidade denominada **topo** da pilha.

Definição

Uma seqüência de objetos, todos do mesmo tipo, sujeita às seguintes regras de comportamento:

- ① Sempre que solicitado a remoção de um elemento, o elemento removido é o último da seqüência.
- ② Sempre que solicitado a inserção de um novo elemento, o objeto é inserido no fim da seqüência (**topo**).

Pilha

- Uma pilha é um objeto dinâmico, constantemente mutável, onde elementos são inseridos e removidos.
- Em uma pilha, cada novo elemento é inserido no topo.
- Os elementos da pilha só podem ser retirado na ordem inversa à ordem em que foram inseridos
 - O primeiro que sai é o último que entrou (*clássico*)
 - Por essa razão, uma pilha é dita uma estrutura do tipo:
LIFO(*last-in, first* ou UEPS último a entrar é o primeiro a sair.)

Operações básicas

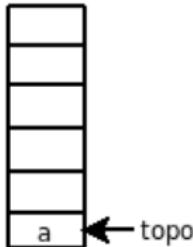
- As operações básicas que devem ser implementadas em uma estrutura do tipo pilha são:

Operação	Descrição
$\text{push}(p, e)$	empilha o elemento e , inserindo-o no topo da pilha p .
$\text{pop}(p)$	desempilha o elemento do topo da pilha p .

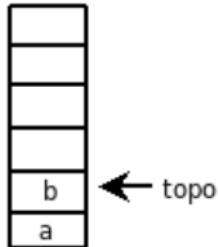
Tabela 1: Operações básicas da estrutura de dados pilha.

Exemplo

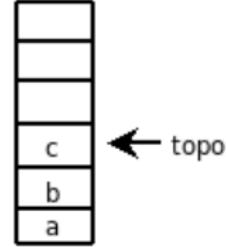
push(a)



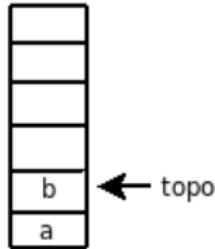
push(b)



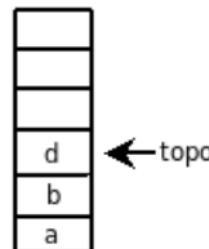
push(c)



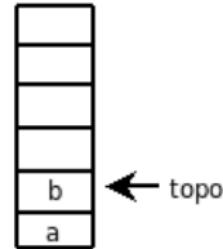
pop
retorna c



push(d)



pop()
retorna d



Operações auxiliares

- Além das operações básicas, temos as operações “auxiliares”. São elas:

Operação	Descrição
create	cria uma pilha vazia.
empty(p)	determina se uma pilha p está ou não vazia.
free(p)	libera o espaço ocupado na memória pela pilha p .

Tabela 2: Operações auxiliares da estrutura de dados pilha.

Interface do Tipo Pilha – Típica

```
1 /* Definicao da estrutura */
2 typedef struct { DEFINA O SEU MODELO AQUI } Pilha;
3
4 /*Aloca dinamicamente ou estaticamente a estrutura pilha,
5   inicializando seus campos e retorna seu ponteiro.*/
6 Pilha * create(void);
7
8 /*Insere o elemento e na pilha p.*/
9 void push(Pilha *p, int e);
10
11 /*Retira e retorna o elemento do topo da pilha p*/
12 int pop(Pilha *p);
13
14 /*Informa se a pilha p esta ou nao vazia.*/
15 int empty(Pilha *p);
```

Implementações

- ➊ Baseada em um simples vetor
- ➋ Baseada em um vetor dinâmico
- ➌ Baseada em lista encadeada

Implementações

- ➊ Baseada em um simples vetor
- ➋ Baseada em um vetor dinâmico
- ➌ Baseada em lista encadeada
- ➍ mas todas usam ponteiros!

Implementação de Pilha com Vetor

- Normalmente as aplicações que precisam de uma estrutura pilha, é comum saber de antemão o número máximo de elementos que precisam estar armazenados simultaneamente na pilha.
- Essa estrutura de pilha tem um limite conhecido.
- Os elementos são armazenados em um vetor.
- Essa implementação é mais simples.
- Os elementos inseridos ocupam as primeiras posições do vetor.

Implementação de Pilha com Vetor

- Seja p uma pilha armazenada em um vetor VET de N elementos:
 - ➊ O elemento $vet[\text{topo}]$ representa o elemento do topo.
 - ➋ A parte ocupada pela pilha é $vet[0 .. \text{topo} - 1]$.
 - ➌ A pilha está vazia se $\text{topo} = -1$.
 - ➍ Cheia se $\text{topo} = N - 1$.
 - ➎ Para desempilhar um elemento da pilha, não vazia, basta

$$x = vet[\text{topo} --]$$

(recupera valor do topo e depois decrementa)

- ➏ Para empilhar um elemento na pilha, em uma pilha não cheia, basta

$$vet[+ + t] = e$$

(soma antes e depois insere)

Implementação de Pilha com Vetor

```
1 #define N 20 /* numero maximo de elementos*/
2 #include <stdio.h>
3 #include "pilha.h"
4
5 /*Define a estrutura da pilha*/
6 struct pilha{
7     int topo;          /* indica o topo da pilha */
8     int elementos[N]; /* elementos da pilha*/
9 };
10
11 Pilha* create(void){
12     Pilha* p = (Pilha * ) malloc(sizeof(Pilha));
13     p->topo = -1;      /* inicializa a pilha com 0 elementos */
14     return p;
15 }
16
```

Implementação de Pilha com Vetor

- Empilha um elemento na pilha

```
1 void push(Pilha *p, int e){  
2     if (p->topo == N - 1){ /* capacidade esgotada */  
3         printf("A pilha esta cheia");  
4         exit(1);  
5     }  
6     /* insere o elemento na proxima posicao livre */  
7     p->elementos[++p->topo] = e;  
8 }
```

Implementação de Pilha com Vetor

- Desempilha um elemento da pilha

```
1 int pop(Pilha *p)
2 {
3     int e;
4     if (empty(p)){
5         printf("Pilha vazia.\n");
6         exit(1);
7     }
8
9     /* retira o elemento do topo */
10    e = p->elementos[p->topo--];
11    return e;
12 }
```

Implementação de Pilha com Vetor

```
1 /**
2  * Verifica se a pilha p esta vazia
3 */
4 int empty(Pilha *p)
5 {
6     return (p->t == -1);
7 }
```

Exemplos de Uso

- Na área computacional existem diversas aplicações de pilhas.
- Alguns exemplos são: caminhamento em árvores, chamadas de sub-rotinas por um compilador ou pelo sistema operacional, inversão de uma lista, avaliar expressões, entre outras.
- Uma das aplicações clássicas é a conversão e a avaliação de expressões algébricas. Um exemplo, é o funcionamento das calculadoras da HP, que trabalham com expressões pós-fixadas.

Exercícios

- ① Os exercícios propostos no início deste capítulo (slide inicial)
- ② Escreva uma função que inverta a ordem das letras de cada palavra de uma sentença, preservando a ordem das palavras. Suponha que as palavras da sentença são separadas por espaços. A aplicação da operação à sentença **AMU MEGASNEM ATERCES**, por exemplo, deve produzir **UMA MENSAGEM SECRETA**.
- ③ Implemente uma função que receba uma pilha como parâmetro e retorne o valor armazenado em seu topo, restaurando o conteúdo da pilha. Essa função deve obedecer ao protótipo:

```
char topo(Pilha* p);
```

- ④ Implemente uma função que receba duas pilhas, p_1, p_2 , e passe todos os elementos da pilha p_2 para o topo da pilha p_1 . Essa função deve obedecer ao protótipo:

```
void concatena(Pilha* p1, Pilha* p2);
```

Onde estamos ...

Capítulo 04 – Filas

Pontos fundamentais a serem cobertos:

- ① Contexto e motivação
- ② Definição
- ③ Implementações
- ④ Exercícios



© Nick White/Image Source/Corbis

Introdução

- Assim como a estrutura de dados Pilha, Fila é outra estrutura de dados bastante utilizada em computação.
- Um exemplo é a implementação de uma fila de impressão.
- Se uma impressora é compartilhada por várias máquinas, normalmente adota-se uma estratégia para determinar a ordem de impressão dos documentos.
- A maneira mais simples é tratar todas as requisições com a mesma prioridade e imprimir os documentos na ordem em que foram submetidos – o primeiro submetido é o primeiro a ser impresso.

Aplicações

- ① Escalonamento de processos na CPU (processos com a mesma prioridade) os quais são executados em ordem de chegada
- ② Simulação de filas no mundo real tais como: filas em banco, compra de tickets, etc
- ③ Multi-programação (*time-sharing*)
- ④ Transferência assíncrona de dados (IO de arquivos, pipe, sockets)
- ⑤ Fila de espera em um *call-center*
- ⑥ Encontrar número de atendentes em supermercados e caixas bancários dado uma demanda de pessoas na sala de espera

Aplicações Indiretas

- ① Estrutura de dados auxiliares em algoritmos
- ② Componente de outras estruturas

Definição

Um conjunto ordenado de itens a partir do qual podem-se eliminar itens numa extremidade (chamada de **início** da fila) e no qual podem-se inserir itens na outra extremidade (chamada **final** da fila).

Representação

- Os nós de uma fila são armazenados em endereços contínuos.
- A Figura ?? ilustra uma fila com três elementos.

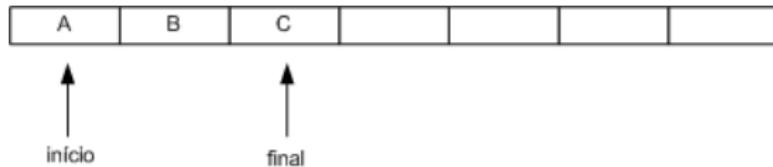


Figura 8: Exemplo de representação de fila.

- Após a retirada de um elemento (*primeiro*) temos:

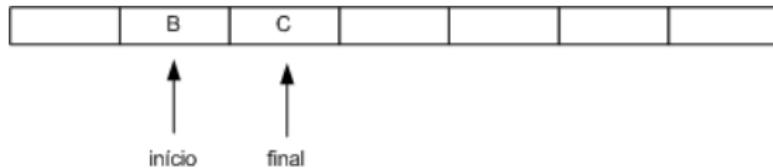


Figura 9: Representação de uma fila após a remoção do elemento “A”.

Implementações

Quais as estruturas usadas por filas?

- Baseada num vetor simples – limitada
- Baseada num vetor circular simples – limitada – igualmente poderosa
- Baseada num vetor circular dinâmico – **i**limitada
- Baseada numa lista encadeada (depois de listas)

Representação

- Após a inclusão de dois elementos temos:

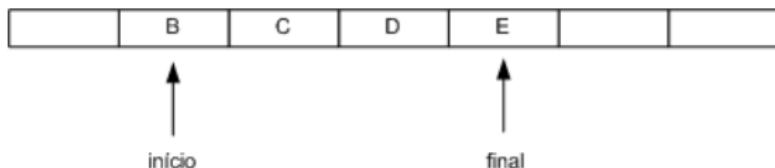


Figura 10: Representação de uma fila após a inclusão de dois elementos “D” e “E”.

- Como podemos observar, a operação de inclusão e retirada de um item da fila incorre na mudança do endereço do ponteiro que informa onde é o início e o término da fila.

Representação

- Em uma fila, o **primeiro** elemento inserido é o primeiro a ser removido.
- Por essa razão, uma fila é chamada **fifo (*first-in first-out*)** – primeiro que entra é o primeiro a sair – ao contrário de uma pilha que é **lifo (*last-in, first-out*)**
- Para exemplificar a implementação em C, vamos considerar que o conteúdo armazenado na fila é do tipo inteiro.
- Nos exemplos do prof é um outro tipo de dado
- A estrutura de fila possui a seguinte representação:

```
1 struct fila{  
2     int elemento[N];  
3     int ini;  
4     int n; // quantos tem na fila  
5 };  
6  
7 typedef struct fila Fila;
```

Representação

- Trata-se de uma estrutura heterogênea constituída de membros distintos entre si.
- Os membros são as variáveis *ini* e *fim*, que serve para armazenar respectivamente, o início e o fim da fila e o vetor *elemento* de inteiros que armazena os itens da fila.

Operações Primitivas

- As operações básicas que devem ser implementadas em uma estrutura do tipo Fila são:

Operação	Descrição
criar()	aloca dinamicamente a estrutura da fila.
insere(f,e)	adiciona um novo elemento (<i>e</i>), no final da fila <i>f</i> .
retira(f)	remove o elemento do início da fila <i>f</i> .

Tabela 3: Operações básicas da estrutura de dados fila.

Operações Auxiliares

- Além das operações básicas, temos as operações “auxiliares”. São elas:

Operação	Descrição
vazia(f)	informa se a fila está ou não vazia.
libera(f)	destrói a estrutura, e assim libera toda a memória alocada

Tabela 4: Operações auxiliares da estrutura de dados fila.

Interface do Tipo Fila

```
1 typedef struct fila Fila;
2 /* Aloca dinamicamente a estrutura Fila, inicializando seus
3  * campos e retorna seu ponteiro. A fila depois de criada
4  * estah vazia.*/
5 Fila* criar(void);
6
7 /* Insere o elemento e no final da fila f, desde que,
8  * a fila nao esteja cheia.*/
9 void insere(Fila* f, int e);
10
11 /* Retira o elemento do inicio da fila, e fornece o
12  * valor do elemento retirado como retorno, desde que a fila
13  * nao esteja vazia*/
14 int retira(Fila* f);
15
16 /*Verifica se a fila f estah vazia*/
17 int vazia(Fila* f);
18
19 /*Libera a memoria alocada pela fila f*/
20 void libera(Fila* f);
```

Reflexão – Pausa

- Como em pilhas, e se estas estruturas puderem ser reutilizadas em diversos tipos de problemas?
- Ou seja, o reuso das mesmas estruturas de dados em problemas diferentes!
- O que é isto?
- Bem-vindos ao Tipos Abstractos de Dados (TAD ou TDA)
- Coloque as declarações de funções em um arquivo .h (Veja stdio.h)
- Coloque as funções em arquivos .c
- Deixe os arquivos .o para serem *linkeditados* no código relocável!
- Basicamente fizemos isto com o *Makefile*

Implementação de Fila com Vetor

- Assim como nos casos da pilha e lista, a implementação de fila será feita usando um vetor para armazenar os elementos.
- Isso implica, que devemos fixar o número máximo de elementos na fila.
- O processo de inserção e remoção em extremidades opostas fará a fila “*andar*” no vetor.
- Por exemplo, se inserirmos os elementos 8, 7, 4, 3 e depois retirarmos dois elementos, a fila não estará mais nas posições iniciais do vetor.

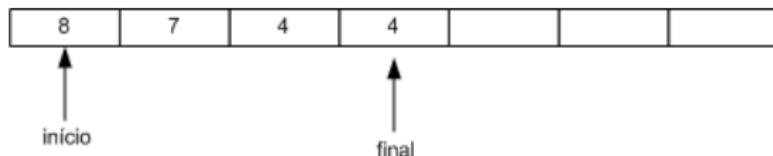


Figura 11: Fila após inserção de quatro elementos (um erro na figura 4/3)

Implementação de Fila com Vetor

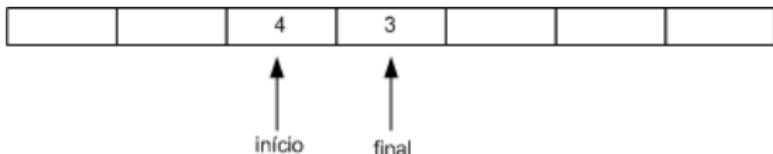


Figura 12: Fila após retirar dois elementos.

- Com essa estratégia, é fácil observar que, em um dado instante, a parte ocupada pelo vetor pode chegar a última posição.
- Uma solução seria ao remover um elemento da fila, deslocar a fila inteira no sentido do início do vetor.
- Entretanto, essa método é bastante ineficiente, pois cada retirada implica em deslocar cada elemento restante da fila. Se uma fila tiver 500 ou 1000 elementos, evidentemente esse seria um preço muito alto a pagar.

Implementação de Fila com Vetor

- Para reaproveitar as primeira posições do vetor sem implementar uma “*re-arrumação*” dos elementos, podemos incrementar as posições do vetor de forma “*circular*”.
- Para essa implementação, os índices do vetor são incrementados de maneira que seus valores progridam “*circularmente*”.
- Dessa forma, se temos 100 posições no vetor, os índices assumem os seguintes valores:

0, 1, 2, 3, ⋯ , 98, 99, 0, 1, 2, 3, ⋯ , 98, 99, ⋯

Porquê um *vetor circular*?

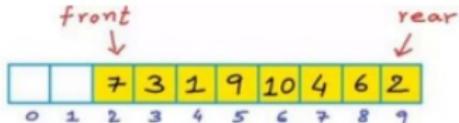
Reflexões

- Tamanho fixo – muito interessante para maioria dos problemas
- Velocidade
- Fácil de manipular
- *Circular* é a manipulação!
- Calcule $f(i + 1)$ dado que a cabeça ou fim de estivessem na posição i em um vetor de N posições (adote a convenção no sentido-horário)
- Cálculo com papel e caneta mesmo!

Vetor Circular: Explicações

Queue: implementation Cyclic Array

no end on array
wrapping around
ring buffer / circular buffer
push/pop - O(1)

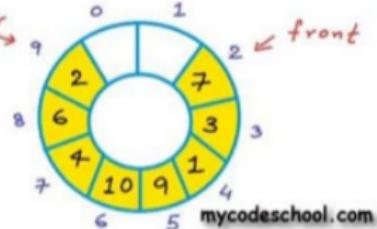


when item inserted to rear, tail's pointer moves upwards
when item deleted, head's pointer moves downwards

current_position = i

next_position = (i + 1) % N

prev_position = (N + i - 1) % N



Vetor Circular: Explicações

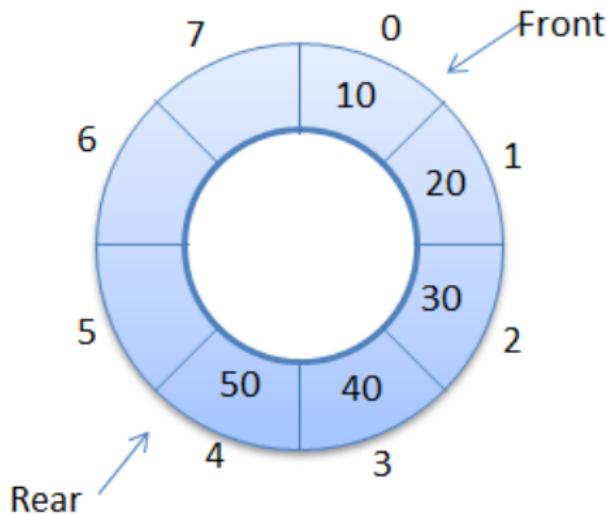
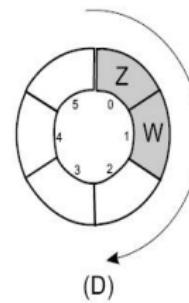
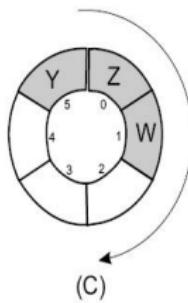
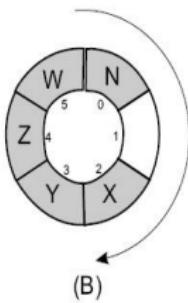
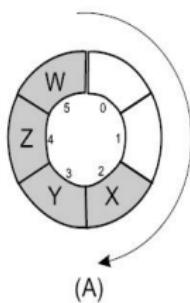


Figura 13: Notação aqui empregada

Vetor Circular: Explicações

	Frente	Cauda	Vetor						Operação
			0	1	2	3	4	5	
A)	2	5		X	Y	Z	W		Insere N
B)	2	0	N	X	Y	Z	W		

	Frente	Cauda	Vetor						Operação
			0	1	2	3	4	5	
C)	5	1	Z	W					Y Retira da Fila
D)	0	1	Z	W					



Função de Criação

- A função que cria uma fila, deve criar e retornar o ponteiro de uma fila vazia
- A função deve informar onde é o início da fila, ou seja, fazer `f->ini = 0`, como podemos ver no código abaixo
- A complexidade de tempo para criar a fila é constante, ou seja, $O(1)$

```
1 /* Aloca dinamicamente a estrutura Fila, inicializando seus
2  * campos e retorna seu ponteiro. A fila depois de criada
3  * estara vazia.
4 */
5 Fila* criar(void)
6 {
7     Fila* f = malloc(sizeof(Fila));
8     f->n = 0; // quantidade CORRENTE elementos
9     f->ini = 0; // inicio
10    return f;
11 }
```

Função de Inserção

- Para inserir um elemento na fila, usamos a próxima posição livre do vetor, indicada por **n**.
- Devemos assegurar que há espaço para inserção do novo elemento no vetor, haja vista se tratar de um vetor com capacidade limitada.
- A complexidade de tempo para inserir um elemento na fila é constante, ou seja, $O(1)$.

```
1 /* Insere o elemento e no final da fila f.*/
2 void insere(Fila* f, int e)
3 {
4     int fim;
5     if (f->n == N){
6         printf("Fila cheia!\n");
7     }else{
8         fim = (f->ini + f->n) % N;
9         f->elementos[fim] = e;
10        f->n++;
11    }
12 }
```

Função de Remoção

- A função para retirar o elemento do início da fila fornece o valor do elemento retirado como retorno.
- Para remover um elemento, devemos verificar se a fila está ou não vazia.
- A complexidade de tempo para remover um elemento da fila é constante, ou seja, $O(1)$.

```
1 int retira(Fila* f)
2 {
3     int e;
4     if ( vazia(f) )
5         printf("Fila vazia!\n");
6     else{
7         e = f->elementos[f->ini];
8         f->ini = (f->ini + 1) % N;
9         f->n--;
10    }
11    return e;
12 }
```

Exemplo de Uso da Fila

```
1 #define N 10
2 #include <stdio.h>
3 #include "fila.h" // TDA.h
4
5 int main(void)
6 {
7     Fila * f = criar();
8
9     int i;
10    for (i = 0; i < N; i++)
11        insere(f, i * 2);
12
13    printf("\nElementos removidos: ");
14
15    for (i = 0; i < N/2; i++)
16        printf("%d ", retira(f));
17
18 }
```

Exercícios

- ① Implemente um esquema de uma fila usando duas pilhas
- ② Encontrar o maior sub-conjunto de uma janela w para um vetor circular N . Basicamente, preencher o vetor inteiro, posicionar inicio em fim de fila, tal que a diferença seja w . Rotacione com acionamentos de chegadas e partidas, até encontrar a maior soma neste vetor. Ou seja, $|\text{rear} - \text{front} + 1| = w$. Função:
`abs(rear - front + 1) = w`
- ③ Construa uma função que copie uma fila, para uma nova fila invertida.
- ④ Faça uma simulação aleatória de chegadas e partidas em uma fila, tal que a cada passo exiba as mensagens de fila cheia, fila vazia, número de elementos corrente na fila. Faça isto para um número considerável de simulações. Este exercício é a base na simulação de sistemas de filas.
- ⑤ Implemente um sistema de fila com prioridades (será um dos projetos)

Referências

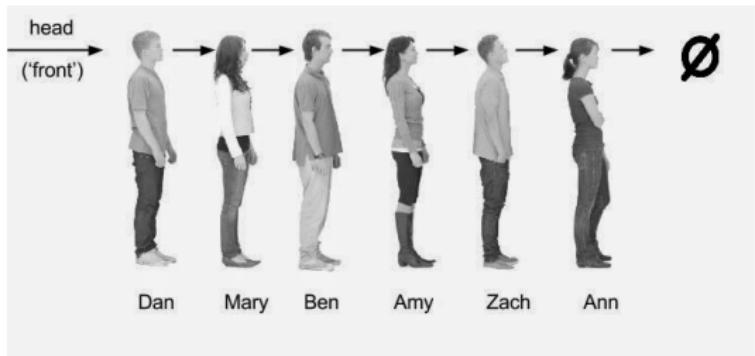
- ① Karumanchi, Narashimha (2017). *Data Structures and Algorithms Made Easy – Data Structures and Algorithms and Puzzles*. CareerMonk.com
- ② Tenenbaum, A. M., Langsam, Y., and Augenstein, M. J. (1995). *Estruturas de Dados Usando C*. MAKRON Books, pp. 207-250.
- ③ Wirth, N. (1989). *Algoritmos e Estrutura de Dados*. LTC, pp. 151-165.

Onde estamos ...

Capítulo 05 – Listas

Pontos fundamentais
a serem cobertos:

- ① Contexto e motivação
- ② Definição
- ③ Implementações
- ④ Exercícios



Atenção:

- Duas partes neste capítulo
- Listas com um vetor de tamanho fixo internamente
- Listas estruturadas em nós alocados dinamicamente
- Conceitualmente, equivalentes!
- Implementações e exemplos aqui discutidas: alocação dinâmica!
- Aqui justifica-se o estudo extensivo de ponteiros no início do curso
- A terminologia aqui adotada é proveniente de vários livros e do professor!

Introdução

- Uma seqüência de nós ou elementos dispostos em uma ordem estritamente linear.
- Cada elemento da lista é acessível um após o outro, em ordem.
- Pode ser implementada de várias maneiras
 - ① Em um vetor
 - ② Em uma estrutura que tem um vetor de tamanho fixo e uma variável para armazenar o tamanho da lista
 - ③ Conjunto de nós criados e ligados dinamicamente (abordagem aqui adotada nos códigos apresentados)

Introdução

- Uma seqüência de nós ou elementos dispostos em uma ordem estritamente linear.
- Cada elemento da lista é acessível um após o outro, em ordem.
- Pode ser implementada de várias maneiras
 - ① Em um vetor
 - ② Em uma estrutura que tem um vetor de tamanho fixo e uma variável para armazenar o tamanho da lista
 - ③ Conjunto de nós criados e ligados dinamicamente (abordagem aqui adotada nos códigos apresentados)
 - ④ As duas implementações iniciais são exercícios de disciplinas anteriores.

Motivação

- ① Talvez a estrutura de dados mais importante
- ② Generaliza Pilhas e Filas
- ③ Utilizada em várias outras estruturas como grafos e árvores

Motivação

- ➊ Talvez a estrutura de dados mais importante
- ➋ Generaliza Pilhas e Filas
- ➌ Utilizada em várias outras estruturas como grafos e árvores
- ➍ Os exemplos em código apresentados, utilizam extensivamente endereçamentos de memória (ponteiros e ponteiros para ponteiros) e alocações dinâmicas de memória
- ➎ Contudo, para fins conceitual usaremos uma lista *rígida* nestes slides – 1a. parte

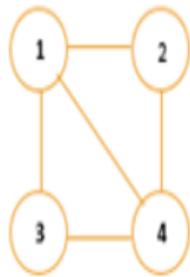
Aplicações

- ① Uso em outros algoritmos e estrutura de dados como árvores e grafos

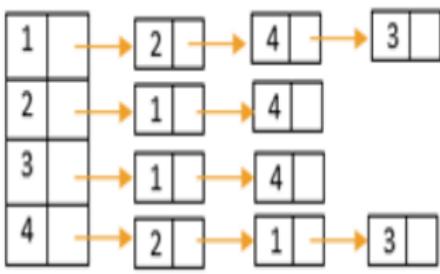
Aplicações

- ① Uso em outros algoritmos e estrutura de dados como árvores e grafos
- ② Conceitualmente, todo estudo aqui feito é aplicado aos outros paradigmas de linguagens de programação

Exemplo de Uso – Matriz Adjacência – Completa



(a) Undirected Graph



(b) List Representation

	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0

(c) Matrix Representation

Figura 14: Uso de lista para representar matriz de adjacência – grafo não-orientado

Exemplo de Uso

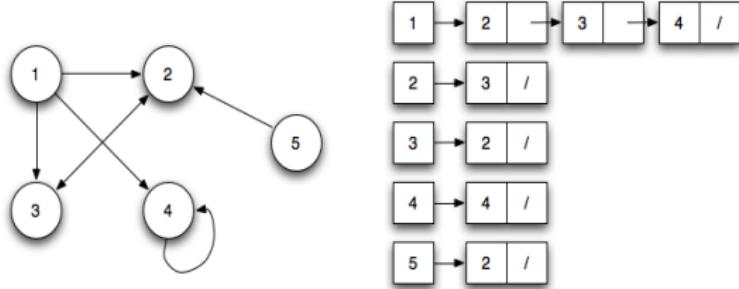


Figura 15: Matriz de adjacência – Unidirecional

Definição

Definição

- Um conjunto de nós, $x_1, x_2, x_3, \dots, x_n$, organizados estruturalmente de forma a refletir as posições relativas dos mesmos.
- Se $n > 0$, então x_1 é o primeiro nó.
- Seja L uma lista de n nós, e x_k um nó $\in L$ e k a posição do nó em L .
- Então, x_k é precedido pelo nó x_{k-1} e seguido pelo nó x_{k+1} .
- O último nó de L é x_{n-1} . Quando $n = 0$, dizemos que a lista está vazia.

Representação

- Os nós de uma lista são armazenados em *endereços contínuos* (apenas os endereços)
- A relação de ordem é representada pelo fato de que se o endereço do nó x_i é conhecido, então o endereço do nó x_{i+1} também pode ser determinado.
- A Figura ?? apresenta a representação de uma lista linear de n nós, com endereços representados por k

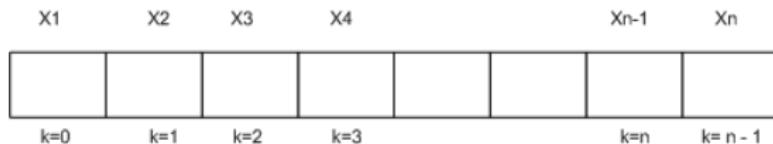


Figura 16: Exemplo de representação de lista – usando um vetor

Representação como um Vetor de Inteiros

- Para exemplificar a implementações em C, vamos considerar que o conteúdo armazenado na lista é do tipo inteiro (nos códigos são *strings*).
- A estrutura da lista possui a seguinte representação:

```
1 struct lista{  
2     int cursor;  
3     int elemento[N];  
4 }  
5 typedef struct lista Lista;
```

- Trata-se de uma estrutura heterogênea constituída de membros distintos entre si.
- Os membros são as variáveis *cursor*, a qual armazena a quantidade de elementos da lista, e o vetor *elemento* de inteiros que armazena os nós da lista.
- Até o momento uma alocação estática – lembra filas e pilhas

Representação Estática – Tamanho Fixo

- Para atribuirmos um valor a algum membro da lista devemos utilizar a seguinte notação:

```
1 Lista->elemento[0] = 1 //atribui o valor 1 ao primeiro elemento da lista
2 Lista->elemento[n-1] = 4 //atribui o valor 4 ao ultimo elemento da lista
```

Operações Primitivas

- As operações básicas que devem ser implementadas em uma estrutura do tipo Lista são:

Operação	Descrição
criar()	cria uma lista vazia.
inserir(l, e)	insere o elemento e no final da lista l .
remover(l, e)	remove o elemento e da lista l .
imprimir(l)	imprime os elementos da lista l .
pesquisar(l, e)	pesquisa o elemento e na lista l .

Tabela 5: Operações básicas da estrutura de dados lista.

Operações auxiliares

- Além das operações básicas, temos as operações “auxiliares”. São elas:

Operação	Descrição
<code>empty(l)</code>	determina se a lista l está ou não vazia.
<code>destroy(l)</code>	libera o espaço ocupado na memória pela lista l .

Tabela 6: Operações auxiliares da estrutura de dados lista.

Interface do Tipo Lista

```
1 /* Aloca dinamicamente a estrutura lista, inicializando
2  * seus campos e retorna seu ponteiro. A lista depois
3  * de criada terah tamanho igual a zero. Sem malloc ... ainda */
4 Lista* criar(void);
5
6 /* Insere o elemento e no final da lista l, desde que,
7  * a lista nao esteja cheia ... dada a limitacao inicial */
8 void inserir(Lista* l, int e);
9
10 /* Remove o elemento e da lista l,
11  * desde que a lista nao esteja vazia e o elemento
12  * e esteja na lista. A funcao retorna 0 se o elemento
13  * nao for encontrado na lista ou 1 caso contrario. */
14 void remover(Lista* l, int e);
15
16 /* Pesquisa na lista l o elemento e. A funcao retorna
17  * o endereco(indice) do elemento se ele pertencer a lista
18  * ou -1 caso contrario.*/
19 int pesquisar(Lista* l, int e);
20
21 /* Lista os elementos da lista l. */
22 void imprimir(Lista* l);
```

Implementação das Listas de Tamanho Fixo

A utilização de vetores para implementar a lista traz algumas vantagens como:

- ① Os elementos são armazenados em posições contíguas da memória
- ② Basta ver a estrutura, internamente é um vetor
- ③ Economia de memória, pois os ponteiros para o próximo elemento da lista são explícitos
- ④ Há um índice de acesso direto e o *cursor* para indicar o último elemento

Implementação das Listas de Tamanho Fixo

No entanto, as desvantagens são:

- ① Custo de inserir/remover elementos da lista

Implementação das Listas de Tamanho Fixo

No entanto, as desvantagens são:

- ① Custo de inserir/remover elementos da lista
- ② Neste caso se refere ao deslocamento células a frente no caso de inserção

Implementação das Listas de Tamanho Fixo

No entanto, as desvantagens são:

- ① Custo de inserir/remover elementos da lista
- ② Neste caso se refere ao deslocamento células a frente no caso de inserção
- ③ ou deslocamento células para trás no caso de remoção

Implementação das Listas de Tamanho Fixo

No entanto, as desvantagens são:

- ① Custo de inserir/remover elementos da lista
- ② Neste caso se refere ao deslocamento células a frente no caso de inserção
- ③ ou deslocamento células para trás no caso de remoção
- ④ Finalmente: limitação da quantidade de elementos da lista
- ⑤ Este é ponto ... tamanho fixo!
- ⑥ Aqui, usamos alocação dinâmica, mas todos conceitos aqui são complementares

Função de Criação

- A função que cria uma lista, deve criar e retornar uma lista vazia;
- A função deve atribuir o valor zero ao tamanho da lista, ou seja, fazer $l->cursor = 0$, como podemos ver no código abaixo.
- A complexidade de tempo para criar a lista é constante, ou seja, $O(1)$.

```
1  /*
2  * Aloca dinamicamente a estrutura lista, inicializando seus
3  * campos e retorna seu ponteiro. A lista depois de criada
4  * terá tamanho igual a zero.
5  */
6 Lista* criar(void){
7     Lista* l = (Lista*) malloc(sizeof(Lista));
8     l->cursor = 0;
9     return l;
10 }
```

Função de Inserção

- A inserção de qualquer elemento ocorre no final da lista, desde que a lista não esteja cheia.
- Com isso, para inserir um elemento basta atribuirmos o valor ao elemento cujo índice é o valor referenciado pelo campo *cursor*, e incrementar o valor do cursor, ou seja fazer
`l->elemento[l->cursor++] = valor`, como podemos verificar no código abaixo, a uma complexidade de tempo constante, $O(1)$.

```
1 /*
2  * Insere o elemento e no final da lista l, desde que,
3  * a lista nao esteja cheia.
4  */
5 void inserir(Lista* l, int e){
6     if (l == NULL || l->cursor == N){
7         printf("Error. A lista esta cheia\n");
8     }else{
9         l->elemento[l->cursor++] = e;
10    }
11 }
```

Função de Remoção

- Para remover um elemento da lista, primeiro precisamos verificar se ele está na lista, para assim removê-lo, e deslocar os seus sucessores, quando o elemento removido não for o último.
- A complexidade de tempo da função de remoção é $O(n)$, pois é necessário movimentar os n elementos para remover um elemento e ajustar a lista.

```
1 /* remove um elemento da lista */
2 void remover(Lista* l, int e){
3     int i, d = pesquisar(l,e);
4     if (d != -1){
5         for(i = d; i < l->cursor; i++)
6         {
7             l->elemento[i] = l->elemento[i + 1];
8         }
9         l->cursor--;
10    }
11 }
```

Função de Pesquisa

- Para pesquisar um elemento qualquer na lista é necessário compará-lo com os elementos existentes, utilizando alguns dos algoritmos de busca conhecidos;
- A complexidade de tempo dessa função depende do algoritmo de busca implementado. Se utilizarmos a busca seqüencial, a complexidade da função será $O(n)$. No entanto, é possível baixá-lo para $O(n \log n)$.

```
1 int pesquisar(Lista* l, int e){  
2     if (l == NULL)  
3         return;  
4  
5     int i = 0;  
6     while (i <= l->cursor && l->elemento[i] != e)  
7         i++;  
8  
9     return i > l->cursor ? -1 : i;  
10 }
```

Função de Impressão

- A impressão da lista ocorre através da apresentação de todos os elementos compreendidos entre o intervalo: $[0..l - > cursor]$.
- A complexidade de tempo da função de impressão é $O(n)$, pois no pior caso, quando lista estiver cheia, é necessário percorrer os n elementos da lista.

```
1 /* Apresenta os elementos da lista l. */
2 void imprimir(Lista* l){
3     int i;
4     for(i = 0; i < l->cursor; i++)
5         printf("%d ", l->elemento[i]);
6     printf("\n");
7 }
```

Exemplo de Uso da Lista

```
1 \#include <stdio.h>
2 \#include "list.h"
3 int main(void)
4 {
5     Lista* l = criar();
6     int i, j = 4;
7
8     /* Inserir 5 elementos na lista */
9     for (i = 0; i < 5; i++)
10        inserir(l, j * i);
11
12    /* Apresenta os elementos inseridos na lista*/
13    imprimir(l);
14    /* Remove o segundo elemento da lista*/
15    remover(l, j);
16    /* Apresenta os elementos da lista */
17    imprimir(l);
18 }
```

Listas Dinamicamente Encadeadas – LE

Atenção:

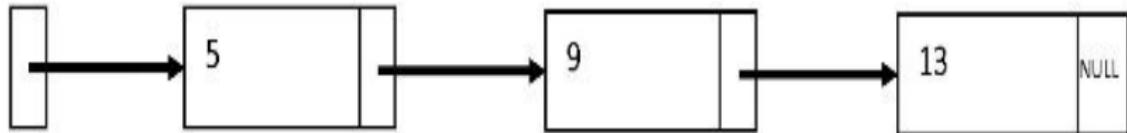
- Esta é a 2a. parte do capítulo
- Os conceitos vistos com listas de *arrays*, agora com nós alocados dinamicamente
- Acompanhe os exemplos do repositório da disciplina
- Basicamente os métodos são os mesmos visto para lista com *arrays*

Listas Encadeadas (ora Listas *Ilimitadas*)

Complemento

- Nesta parte vamos discutir as listas *ilimitadas*
- Crescem ou não dinâmicamente de acordo com a disponibilidade de memória
- O usuário controla a memória etc
- Todos conceitos vistos anteriormente, são aqui preservados
-

Estrutura clássica da implementação dessas listas



```
struct Node {  
    int data;  
    Node *nextPtr;  
};
```

```
Node *head;
```

Figura 17: Lista Simplesmente Encadeada – LSE ou LE

Conceitualmente, nada foi modificada em relação ao que se tinha anteriormente!

Espalhadas pela memória principal ... não contíguas!

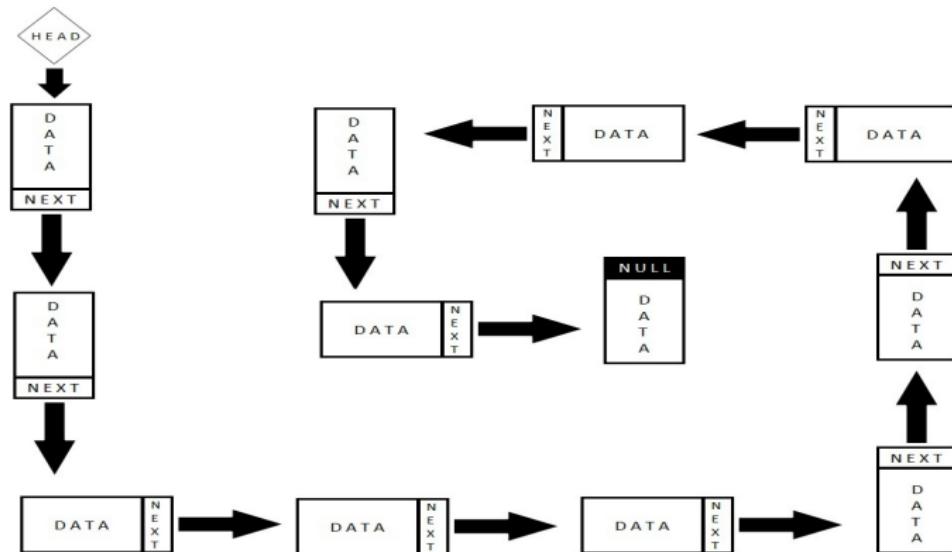


Figura 18: Lista Simplesmente Encadeada – LSE

Truques destas Implementações de Listas

Observações:

- Ao contrário do que foi visto, estas estruturas são alocadas dinamicamente na MP;
- Idem quanto as remoções e a sua liberação de área;
- O ponto é que facilmente pode se perder o ponteiro de início de lista se o mesmo entrar no cálculo ou numa operação de lista;
- Logo, vamos passar apenas o seu endereço afim de preservá-lo na função principal

Truques destas Implementações de Listas

Observações:

- Uma proteção (*blindagem*) deste ponteiro cabeça de lista
- Ou seja, passando apenas o seu endereço para as funções, estas terão que receber como um ponteiro de ponteiro!
- Sim, pois virá o endereço de um ponteiro que tem o início de lista, logo, uma estrutura de nó com um ponteiro de ponteiro é agora interessante.
- Veja as implementações e as estude com cuidado!
- Faça um desenho de tudo que está escrito ... vais precisar!

Complexidade de Implementações de Listas

Parameter	Linked List	Array	Dynamic Array
Indexing	$O(n)$	$O(1)$	$O(1)$
Insertion/deletion at beginning	$O(1)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Insertion at ending	$O(n)$	$O(1)$, if array is not full	$O(1)$, if array is not full $O(n)$, if array is full
Deletion at ending	$O(n)$	$O(1)$	$O(n)$
Insertion in middle	$O(n)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Deletion in middle	$O(n)$	$O(n)$, if array is not full (for shifting the elements)	$O(n)$
Wasted space	$O(n)$ (for pointers)	0	$O(n)$

Figura 19: Comparativo da complexidade quanto as implementações

Comparativo de Implementações de Listas – *Vetores* (dinâmicos e estáticos) × Nós Encadeados

Vantagens dos Vetores (ou *Arrays*):

- Simples e fácil de usar
- Tempo de acesso constante, pois há um índice para acesso de seus elementos
- Se o *arrays* for dinâmico, *malloc* e *realloc*, temos crescimento e encolhimentos adaptativos. Se e somente se forem dinâmicos!

Desvantagens dos Vetores (ou *Arrays*):

- Pré-aloca uma quantidade e permanece fixa até o final
- Desperdícios quando há células não utilizadas
- Complexo quanto a movimentações de células ou blocos (muitas células sendo inseridas ou excluídas simultaneamente ≈ operação em *bloco*)

Comparativo de Implementações de Listas – *Vetores* (dinâmicos e estáticos) × Nós Encadeados

Vantagens das listas:

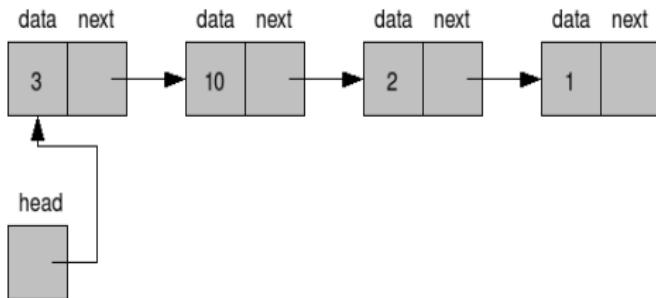
- Crescem em tempo constante
- Operações como criar, destruir, tomam tempo constante, outras operações $O(n)$, ver tabela
- Não há desperdícios de memória: *alocou = usou*

Desvantagens das listas:

- Principal problema é o tempo de acesso as células: num vetor $O(1)$ e nas listas $O(n)$
- Outro problema são os espaços contíguos: num vetor, quando alocado pode ir para uma memória *cache*, e se beneficiar de mecanismos de *caching* de CPUs e SOs modernos
- Finalmente, algum atraso em operações mais complexas, tal como alocar um NULL no último ponteiro da lista

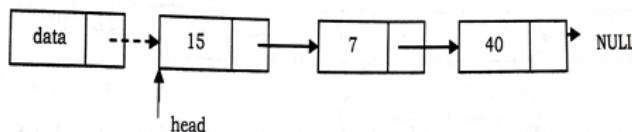
Implementações de Alguns Métodos

Incluindo um nó no início da lista



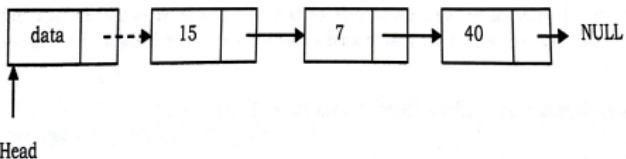
- Update the next pointer of new node, to point to the current head.

New node



- Update head pointer to point to the new node.

New node



Incluindo um nó numa posição da lista

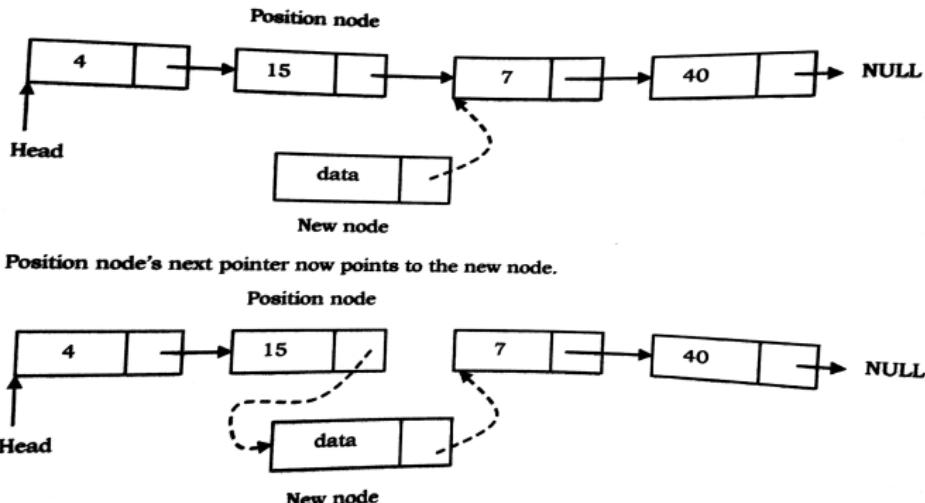
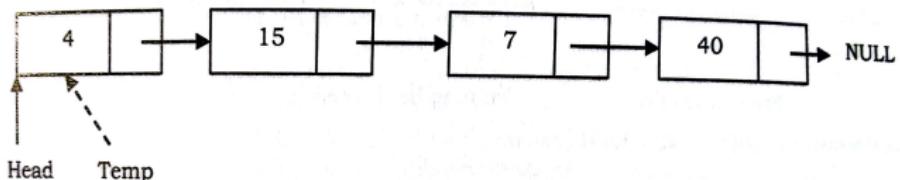


Figura 21: Incluir um nó numa posição da lista

Excluindo o nó no início da lista – *cabeça* da lista

Create a temporary node which will point to the same node as that of head.



Now, move the head nodes pointer to the next node and dispose of the temporary node.

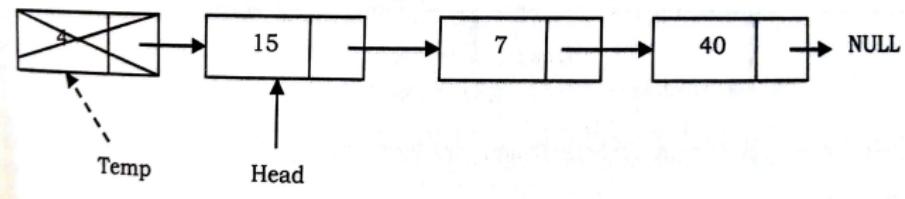


Figura 22: Exclui o nó no início da lista

Excluindo um nó no meio da lista

Similar to the previous case, maintain the previous node while traversing the list. Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.

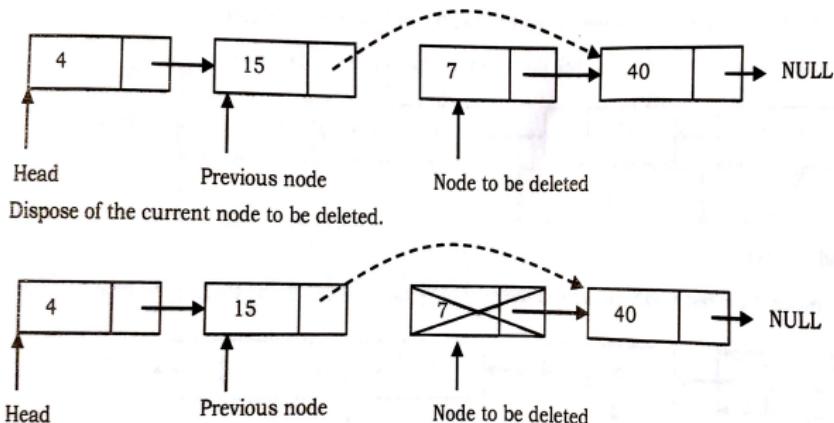
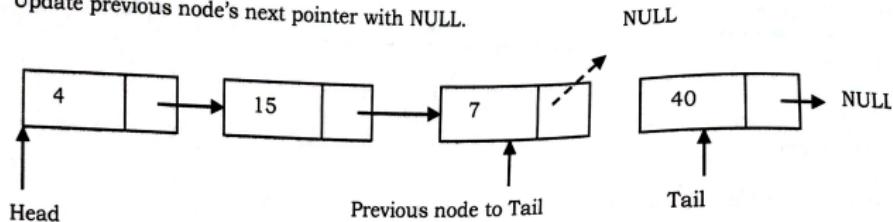


Figura 23: Exclui nó no meio da lista – k-ésima posição ou por conteúdo

Excluindo o último nó da lista

- Update previous node's next pointer with NULL.



- Dispose of the tail node.

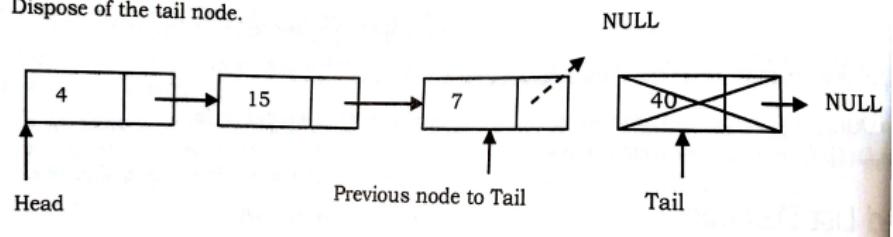


Figura 24: Exclui o último nó da lista

Listas Duplamente Encadeadas – LDE

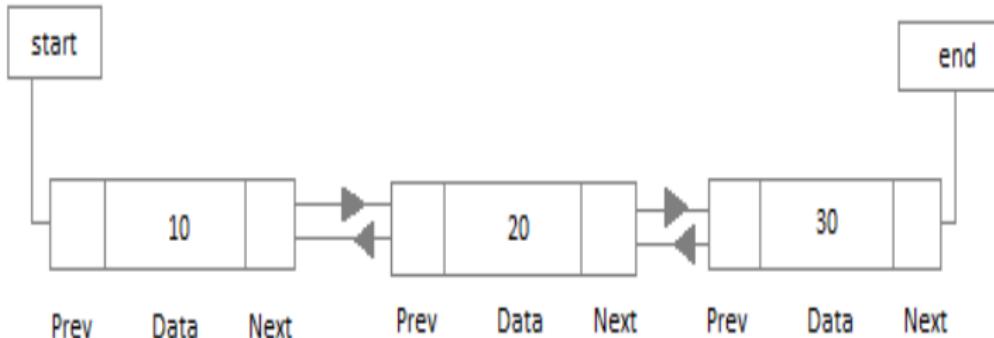


Figura 25: Listas Duplamente Encadeadas – LDE

- Para onde estão apontando o primeiro e o último ponteiro ?
- Podem ser circulares?

Comparativo com as LSE ou LE

Vantagens:

- Dado um determinado nó na lista, se pode navegar em ambas direções
- A remoção é mais simples que a LSE, pois se tem no nó corrente o endereço do nó anterior e seguinte
- Se entendeste os métodos das LSE, vais entender os métodos das LDE

Desvantagens:

- Principal desvantagem: cada nó precisa de um ponteiro extra (para o nó prévio)
- Assim, na exclusão e inserção, algumas linhas de código a mais, embora mais claras de entender;

Aplicações

Exemplos:

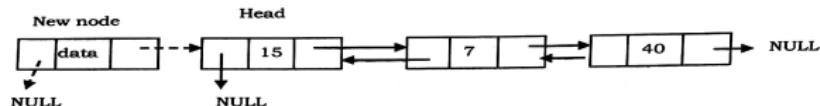
- Paginadores em geral
- Buscas posicionais com deslocamentos em relação a origem, último nó, e ainda do nó corrente

Inserindo um nó em listas duplamente encadeadas – LDE

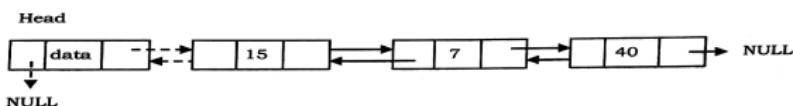
Inserting a Node in Doubly Linked List at the Beginning

In this case, new node is inserted before the head node. Previous and next pointers need to be modified and it can be done in two steps:

- Update the right pointer of the new node to point to the current head node (dotted link in below figure) and also make left pointer of new node as NULL.



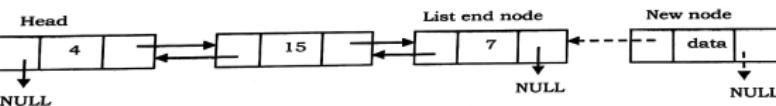
- Update head node's left pointer to point to the new node and make new node as head.



Inserting a Node in Doubly Linked List at the Ending

In this case, traverse the list till the end and insert the new node.

- New node right pointer points to NULL and left pointer points to the end of the list.



- Update right pointer of last node to point to new node.

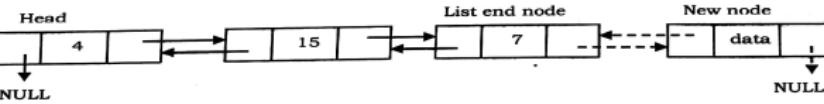


Figura 26: Incluindo um nó numa LDE

Listas Circulares – LC

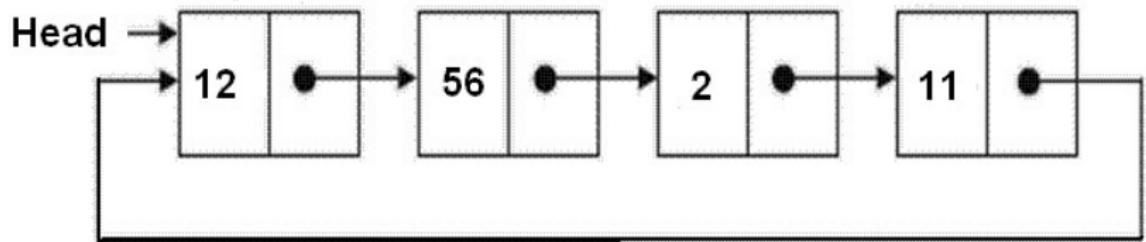


Figura 27: Listas Circulares – LC

Comparativo com as LSE e LDE

Vantagens:

- Uma construção elegante
- O nó inicial da lista, cabeça, pode variar ... desloque o *header* da lista para posição seguinte, e este mantém toda lista original (um bom problema aqui)
- Operações sobre o *header* na lista tornam esta uma estrutura versátil!

Desvantagens:

- Enquanto que LDE e LSE, o último nó é ligado a NULL, aqui não!
- Logo, algum trabalho em encontrar o último da lista (o *tail*), e com isto fazer o uso do método
- Cuidados extras aos métodos das LC, pois um *laço infinito* é provável e fácil
- Um *marcador extra* de fim de lista pode ser necessário em algumas

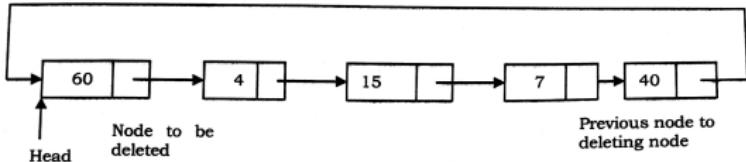
Aplicações

Exemplos:

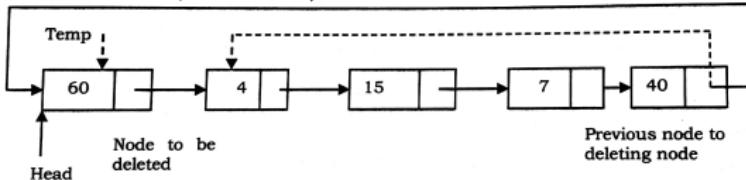
- Sistemas operacionais ... escalonadores de recursos
- Gerenciamento de processos em CPU. Ex: *time sharing* e/ou *round-robin*
- Sistemas de prevenção de *dead-lock*

Excluindo o nó *cabeça* de lista

- Find the tail node of the linked list by traversing the list. Tail node is the previous node to the head node which we want to delete.



- Create a temporary node which will point to the head. Also, update the tail nodes next pointer to point to next node of head (as shown below).



- Now, move the head pointer to next node. Create a temporary node which will point to head. Also, update the tail nodes next pointer to point to next node of head (as shown below).

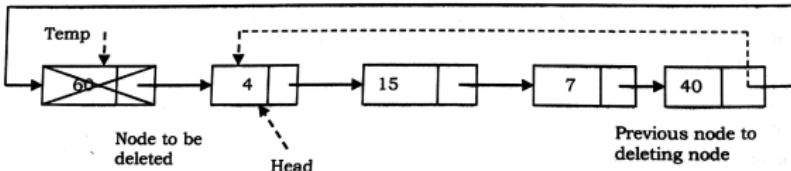


Figura 28: Exclui primeiro nó da lista circular – fácil

Nas implementações está a exlcusão genérica, pois ...

Listas Mistas

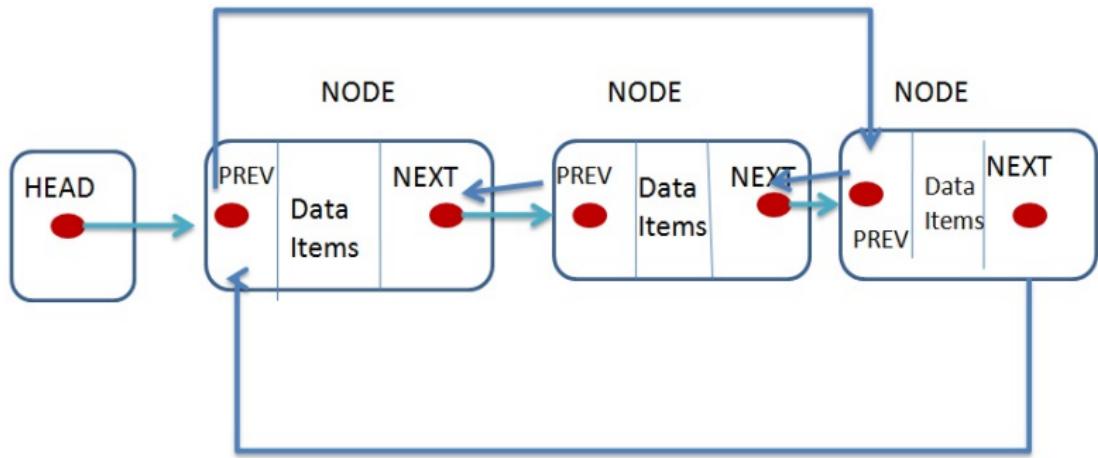
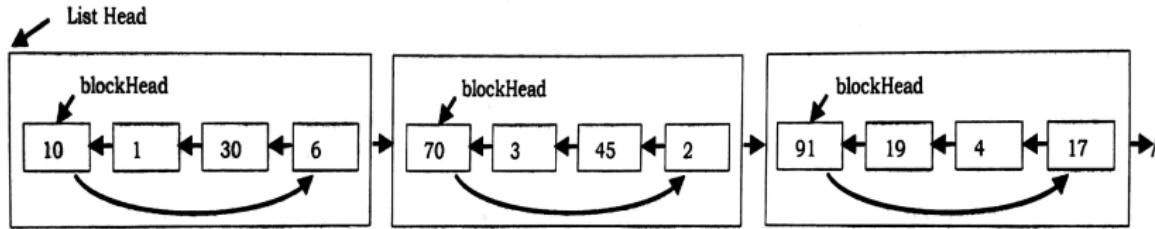


Figura 29: Listas Mistas – DE e Circular

Listas Blocadas ou Encapsuladas – *enrolled*

One of the biggest advantages of linked lists over arrays is that inserting an element at any location takes only $O(1)$ time. However, it takes $O(n)$ to search for an element in a linked list. There is a simple variation of the singly linked list called *unrolled linked lists*.

An unrolled linked list stores multiple elements in each node (let us call it a block for our convenience). In each block, a circular linked list is used to connect all nodes.

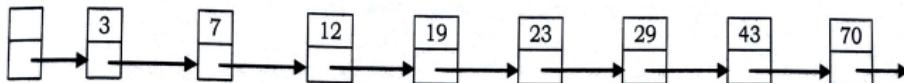


Assume that there will be no more than n elements in the unrolled linked list at any time. To simplify this problem, all blocks, except the last one, should contain exactly $\lceil \sqrt{n} \rceil$ elements. Thus, there will be no more than $\lfloor \sqrt{n} \rfloor$ blocks at any time.

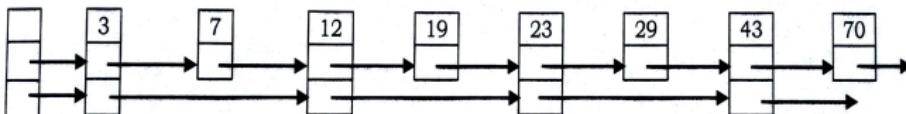
Figura 30: Listas Encapsuladas

Listas Niveladas – skiped

Skip Lists with One Level



Skip Lists with Two Levels



Skip Lists with Three Levels

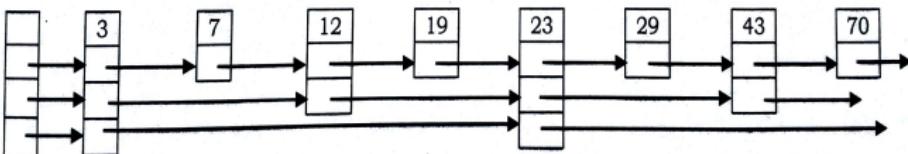


Figura 31: Listas Niveladas

Exercícios

Aqui esta *lista* é grande ...

- Inverter a lista
- *Merge* de duas listas
- Concatenar duas listas simples
- Ver exercícios do run.codes

Conclusões

- Comprova-se a importância do (profundo) entendimento de ponteiros
- Há também algoritmos recursivos que tratam listas
- Listas generalizam filas e pilhas
- Há muitas variações sobre listas, haja visto a sua aplicabilidade (enorme)
- Crie sua própria biblioteca
- Siga um padrão STL do C++ (futuros maratonistas)
- Sua concepção e uso é estendida a árvores e grafos – próximos capítulos

Referências

- ① Karumanchi, Narashimha (2017). *Data Structures and Algorithms Made Easy – Data Structures and Algorithms and Puzzles*. CareerMonk.com
- ② Tenenbaum, A. M., Langsam, Y., and Augenstein, M. J. (1995). Estruturas de Dados Usando C. MAKRON Books, pp. 207-250.
- ③ Wirth, N. (1989). Algoritmos e Estrutura de dados. LTC, pp. 151-165.

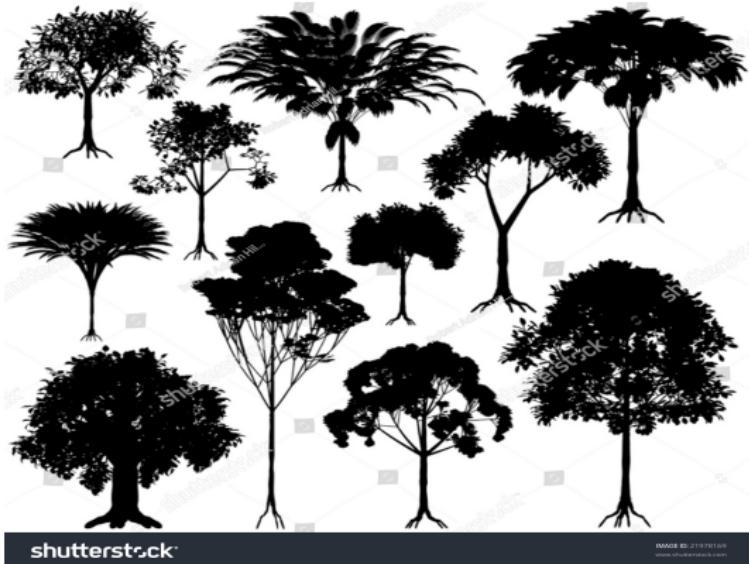
Onde estamos ...

Capítulo 06 – Árvores

Pontos fundamentais a serem cobertos:

mento

- ① Contexto e motivação
- ② Definição
- ③ Implementações
- ④ Exercícios



Definição

- Uma árvore é uma estrutura hierárquica composta por nós e ligações entre eles
- Pode ser vista como um grafo acíclico
- Cada nó possui somente um pai e zero ou mais filhos
- Muitas definições ...

Estrutura

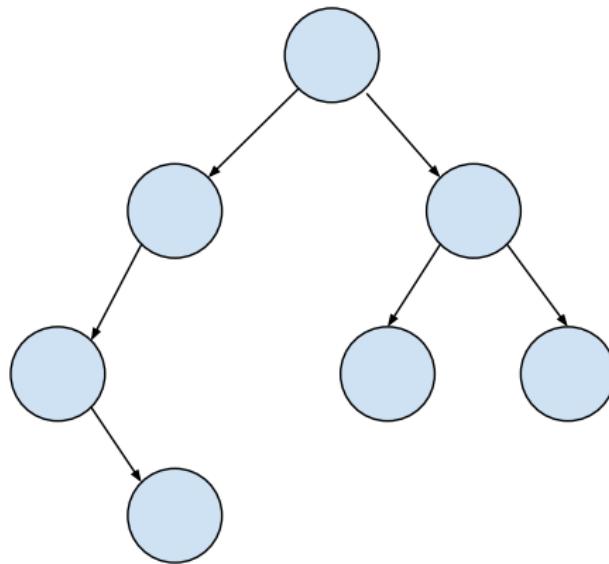


Figura 32: Exemplo de uma árvore

Roadmap para estudo

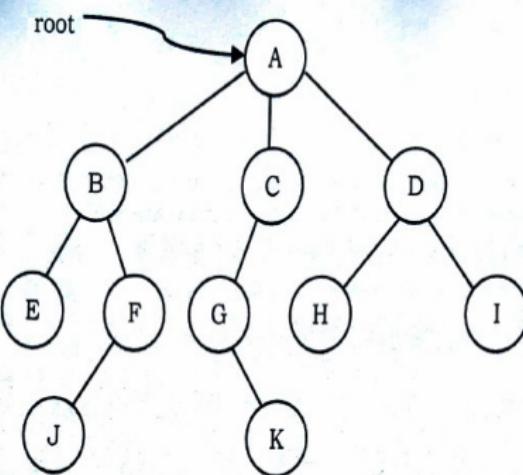
mantendo um *foco*:

- ① Conceitos de árvores genéricas etc ...
- ② Árvores Binárias
- ③ Árvore Binária de Busca
- ④ Árvore AVL (iniciais dos nomes: *Georgii Adelson-Velsky et Evgenii Landis (en), qui l'ont publié en 1962 sous le titre An Algorithm for the Organization of Information*)
- ⑤ Projeto 10% :
 - Implemente uma AVL;
 - Leia um conjunto de dados numéricos contidos no arquivo fornecido (um valor por linha: string, int, float, char), inserindo-os sequencialmente na AVL implementada;
 - Imprima o percurso (valores dos nós) em pré-ordem, em-ordem e pós-ordem, além da altura da árvore
 - Com a altura dará para ver se a AVL está OK!
- ⑥ Vídeos bem legais no Youtube da UNIVEST

Características – Requisitos

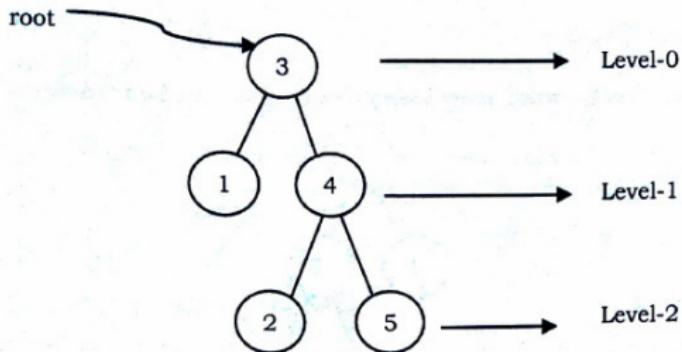
- Como é um nó?
- Qual o grau de um nó?
- Como devem estar estruturados os valores dos nós?
- O que é uma chave do nó?
- O que é a altura?
- Nível?
- Caminhos

Glossário – 01



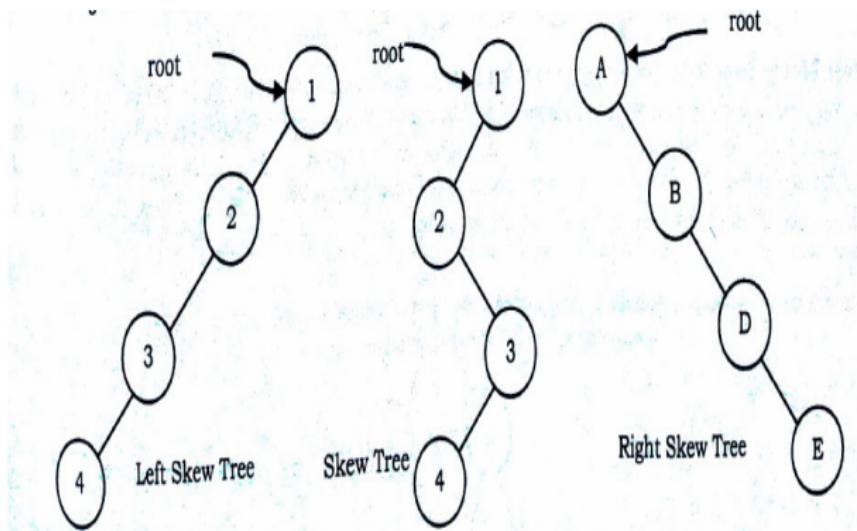
- The *root* of a tree is the node with no parents. There can be at most one root node in a tree (node *A* above example).
- An *edge* refers to the link from parent to child (all links in the figure).
- A node with no children is called *leaf node* (*E, J, K, H* and *I*).
- Children of same parent are called *siblings* (*B, C, D* are siblings of *A*, and *E, F* are the siblings of *B*).
- A node *p* is an *ancestor* of node *q* if there exists a path from *root* to *q* and *p* appears on the path. The is called a *descendant* of *p*. For example, *A, C* and *G* are the ancestors of *K*.
- The set of all nodes at a given depth is called the *level* of the tree (*B, C* and *D* are the same level). The node is at level zero.

Glossário – 02



- The *depth* of a node is the length of the path from the root to the node (depth of G is 2, $A - C - G$).
- The *height* of a node is the length of the path from that node to the deepest node. The height of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a height of zero. In the previous example, the height of B is 2 ($B - F - J$).
- *Height of the tree* is the maximum height among all the nodes in the tree and *depth of the tree* is the maximum depth among all the nodes in the tree. For a given tree, depth and height returns the same value. But for individual nodes we may get different results.
- The size of a node is the number of descendants it has including itself (the size of the subtree C is 3).
- If every node in a tree has only one child (except leaf nodes) then we call such trees *skew trees*. If every node has only left child then we call them *left skew trees*. Similarly, if every node has only right child then we call them *right skew trees*.

Glossário – 03



Árvores Genéricas

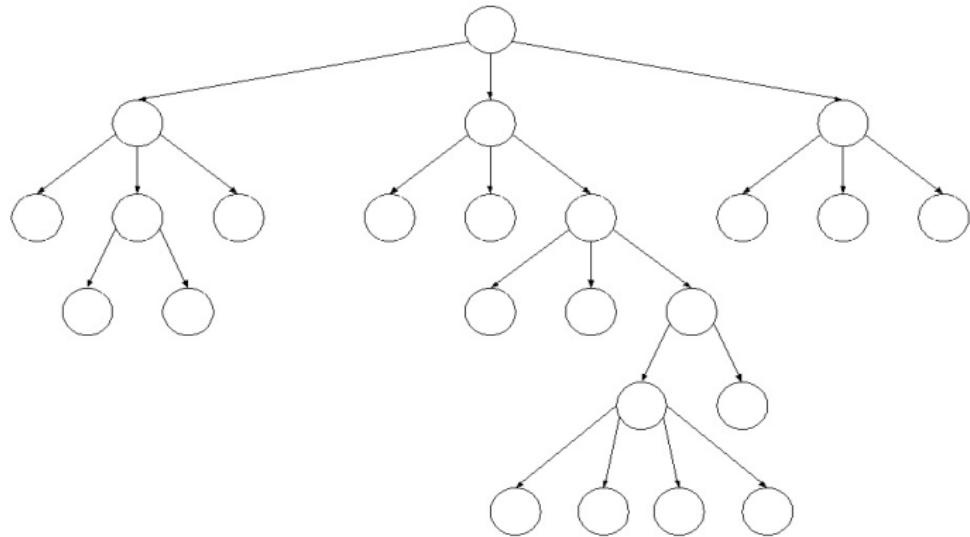


Figura 33: Usando os problemas de árvores genéricas para apresentar Árvores Binárias (AB)

Transformando uma Árvore Genérica em Binária

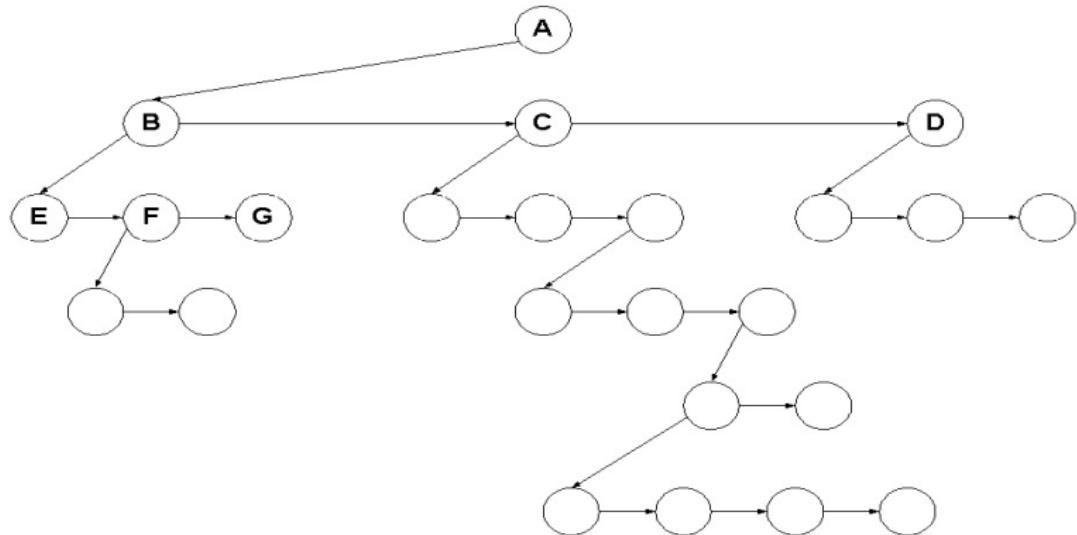
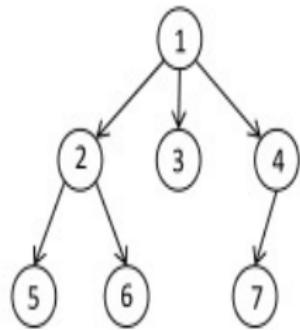
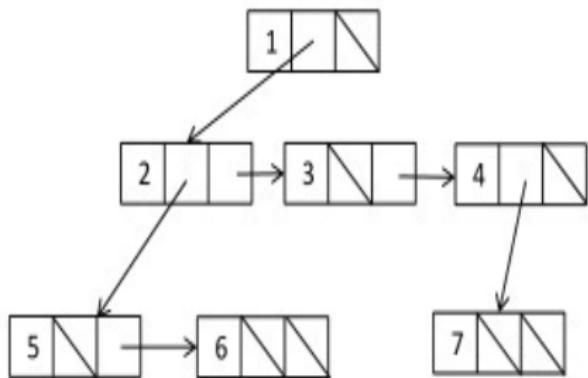


Figura 34: Usando os problemas de árvores genéricas para apresentar Árvores Binárias(AB)

Representação Computacional de uma Árvore Genérica



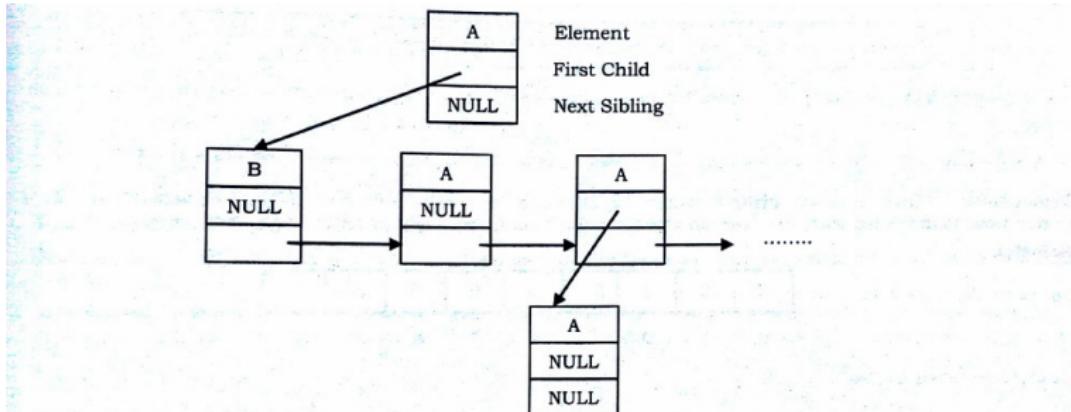
Tree Structure



Linked List representation of A Tree

Figura 35: Felizmente há um algoritmo que transforme Árvores Genéricas em Binárias (AB)

Representação Computacional de uma Árvore Genérica



Based on this discussion, the tree node declaration for general tree can be given as:

```
struct TreeNode {  
    int data;  
    struct TreeNode *firstChild;  
    struct TreeNode *nextSibling;  
};
```

Note: Since we are able to convert any generic tree to binary representation; in practice we use binary trees. We can treat all generic trees with a first child/next sibling representation as binary trees.

Figura 36: Veja a *struct ...* lembra o quê?

Aplicações

- Área de compiladores: análise sintática
- Buscas com complexidade na ordem de: $O(\log n)$
- Na área de IA para construção de árvores de decisão: mineração de dados (*big data*)
- Organização de taxonomias de conhecimento
- Estruturas hierárquicas em geral

Aplicação de Árvores

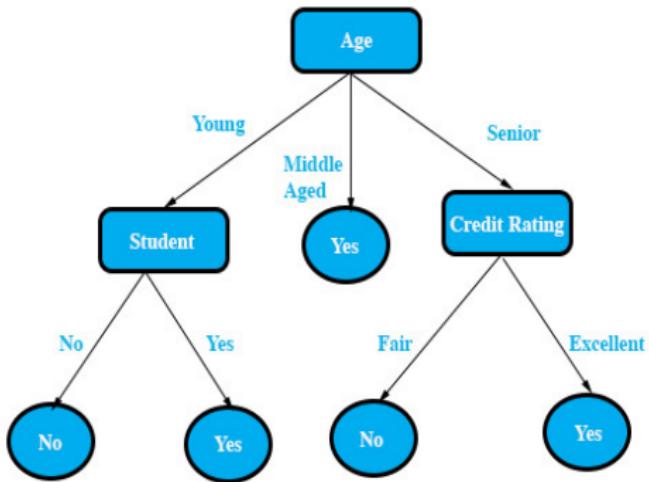
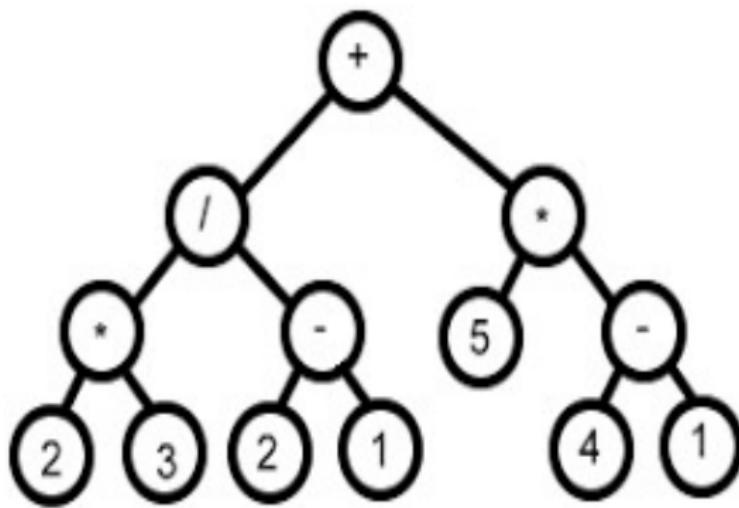


Figura 37: Árvore de Decisão

Aplicação de Árvores



Expression tree for $2*3/(2-1)+5*(4-1)$

Figura 38: Árvores de expressões – binárias – novamente

Árvores Binárias de Buscas - ABBs

- Árvores Genéricas (AGs): foram utilizadas para motivação as ABBs
- Computacionalmente, as ABBs tem um interesse maior que as AGs
- Assim, se inicia com ABBs, e seus algoritmos serão adaptados as AGs

Árvores Binárias de Buscas - ABBs

- Árvores Genéricas (AGs): foram utilizadas para motivação as ABBs
- Computacionalmente, as ABBs tem um interesse maior que as AGs
- Assim, se inicia com ABBs, e seus algoritmos serão adaptados as AGs

Definição:

Árvore onde cada nó possui até 2 filhos. O filho da esquerda só pode conter chaves menores do que a do pai, enquanto que o filho da direita só comporta chaves maiores do que a do pai.

Árvore Binária de Busca

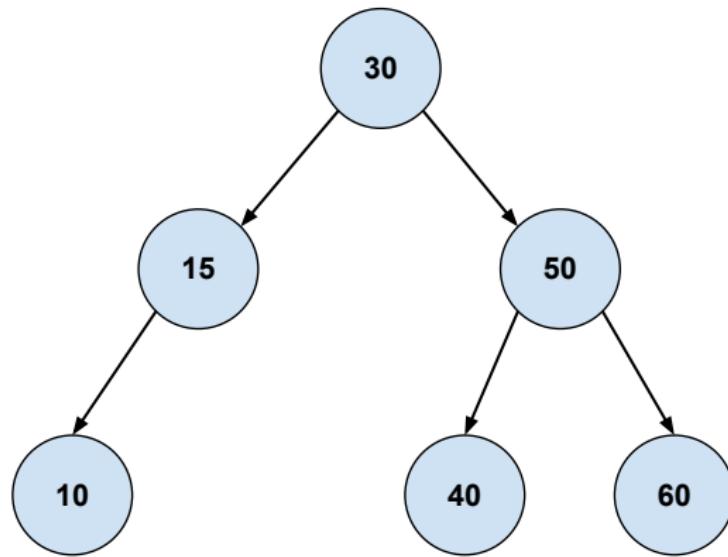
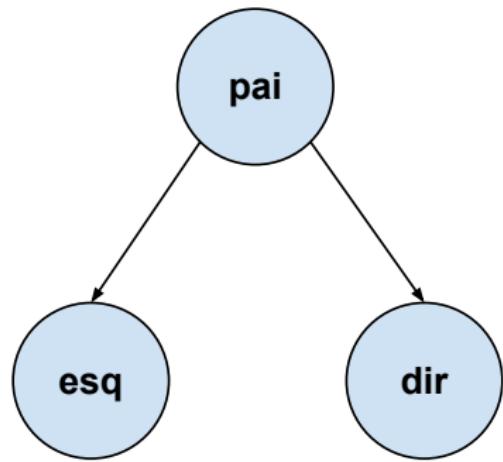


Figura 39: Exemplo de árvore binária de busca

Árvore Binária de Busca



```
struct ArvoreBinaria {  
    struct ArvoreBinaria* esq;  
    struct ArvoreBinaria* dir;  
  
    // contém a chave e os dados satélite  
    Tipoltem valor;  
};
```

Figura 40: Estrutura básica / nó

Operações Básicas

Operações Básicas

- Inserção
- Visitas na árvore (percorrer os nós)
- Busca
- Remoção – faltando

Usos Comuns

- Dicionários / vetores associativos
- Filas de prioridades

Reflexões da última aula

Alguns pontos atenção:

- Muitos conceitos importantes
- Algoritmos *pequenos* para ABs usando a recursividade
- A recursividade tem um custo de espaço computacional (conteúdo ilustrado ao longo do curso)
- Alternativa e uma estrutura fundamental em ABs: **pilhas**
- Empilhar/Desempilhar uma sequência de nós numa busca ou meta
- ABs \neq ABB (são diferentes)
- Exemplo: o conceito de **chave** é onipresente nas ABBs
- Exemplifique esta diferença e avance após esta figura no seu caderno

Complexidade Computacional

- Quando a árvore está balanceada todas as três operações podem ser implementadas com complexidade computacional igual a $O(\log n)$
- Faça o desenho e verifique o crescimento exponencial de nós na árvore, e sua função inversa é um logaritmo na base 2 (estude isto!)
- No pior caso (desbalanceamento) estas operações possuem complexidade $O(n)$ [?].
- *Quizz:* uma árvore binária com n nós, quantas topologias de árvores distintas podemos formar?

Complexidade Computacional

- Quando a árvore está balanceada todas as três operações podem ser implementadas com complexidade computacional igual a $O(\log n)$
- Faça o desenho e verifique o crescimento exponencial de nós na árvore, e sua função inversa é um logaritmo na base 2 (estude isto!)
- No pior caso (desbalanceamento) estas operações possuem complexidade $O(n)$ [?].
- *Quizz:* uma árvore binária com n nós, quantas topologias de árvores distintas podemos formar?
- Comece com $n = 1, n = 2, n = 3, \dots$

Complexidade Computacional

- Quando a árvore está balanceada todas as três operações podem ser implementadas com complexidade computacional igual a $O(\log n)$
- Faça o desenho e verifique o crescimento exponencial de nós na árvore, e sua função inversa é um logaritmo na base 2 (estude isto!)
- No pior caso (desbalanceamento) estas operações possuem complexidade $O(n)$ [?].
- *Quizz:* uma árvore binária com n nós, quantas topologias de árvores distintas podemos formar?
- Comece com $n = 1, n = 2, n = 3, \dots$
- R: $2^n - n$

Árvore Binária de Busca - Inserção

```
INSERÇÃO(ARVORE, ITEM) {
    SE ARVORE == NULO
        ARVORE->ITEM = ITEM
        return

    SE ITEM->CHAVE < ARVORE->CHAVE
        SE ARVORE->ESQ = NULO ENTÃO
            ARVORE->ESQ = ARVORE(ITEM)
        SENÃO
            INSERÇÃO(ARVORE->ESQ, ITEM)
        SENÃO
            SE ARVORE->DIR = NULO ENTÃO
                ARVORE->DIR = ARVORE(ITEM)
            SENÃO
                INSERÇÃO(ARVORE->DIR, ITEM)
    }
}
```

- Se a árvore estiver vazia em t_0 ($\ominus \oplus$), esta vira uma ABB ao final
- Vá ao fonte implementado – insere nó

Árvore Binária de Busca - Inserção

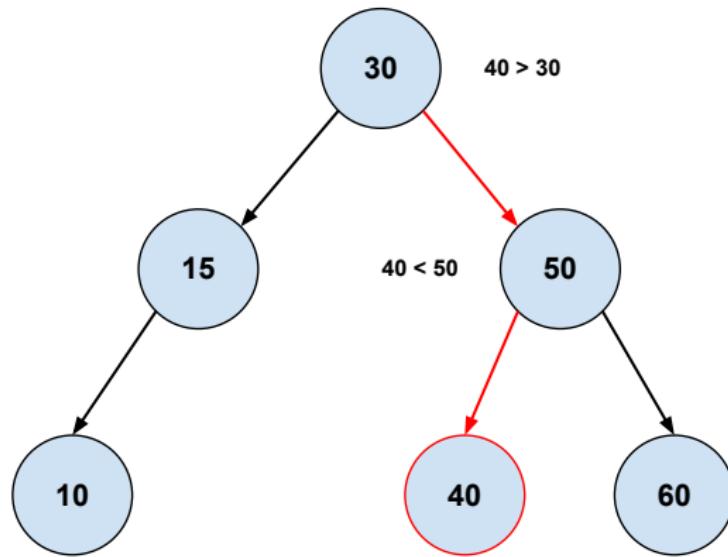


Figura 41: Exemplo de inserção da chave 40

Árvore Binária de Busca - Inserção

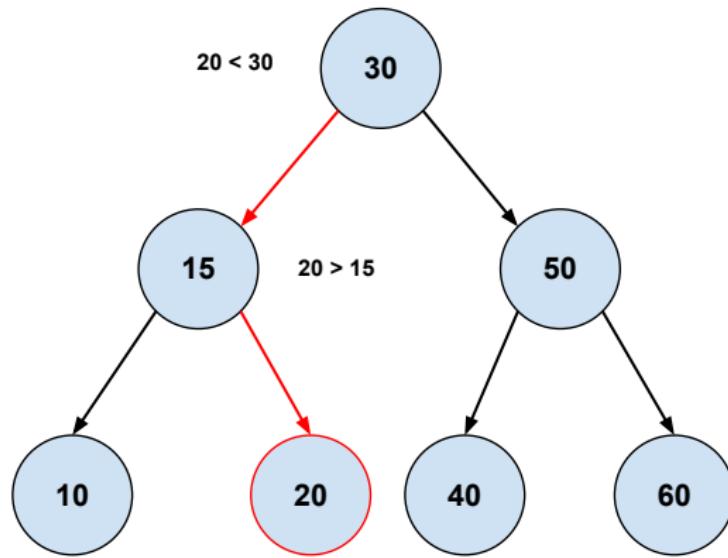


Figura 42: Exemplo de inserção da chave 20

Exercícios – Inserção

Seguindo o algoritmo acima de inserção, construa as árvores binárias para seguintes entradas:

- $30 \rightarrow 20 \rightarrow 40 \rightarrow 10 \rightarrow 25 \rightarrow 35 \rightarrow 50$
- $30 \rightarrow 40 \rightarrow 20 \rightarrow 35 \rightarrow 25 \rightarrow 10 \rightarrow 50$
- $50 \rightarrow 40 \rightarrow 20 \rightarrow 35 \rightarrow 25 \rightarrow 10 \rightarrow 30$
- $50 \rightarrow 25 \rightarrow 20 \rightarrow 35 \rightarrow 40 \rightarrow 10 \rightarrow 30$
- $10 \rightarrow 20 \rightarrow 25 \rightarrow 30 \rightarrow 35 \rightarrow 40 \rightarrow 50$
- Que reflexões?

Percorrendo Árvores Binárias

- Uma operação muito comum é percorrer uma árvore binária, o que significa passar por todos os nós, pelo menos uma vez.
- O conceito de visitar significa executar uma operação com a informação armazenada no nó, por exemplo, imprimir seu conteúdo.
- Na operação de percorrer a árvore pode-se passar por alguns nós mais de uma vez, sem porém visitá-los.
- Uma árvore é uma estrutura não seqüencial, diferentemente de uma lista, por exemplo. Não existe ordem natural para percorrer árvores e portanto podemos escolher diferentes maneiras de percorrê-las.
- Iremos estudar três métodos para percorrer árvores.
- Todos estes três métodos podem ser definidos recursivamente e se baseiam em **três operações básicas**: visitar a raiz, percorrer a subárvore da esquerda e percorrer a subárvore da direita.
- A única diferença entre estes métodos é a ordem em que estas operações são executadas.

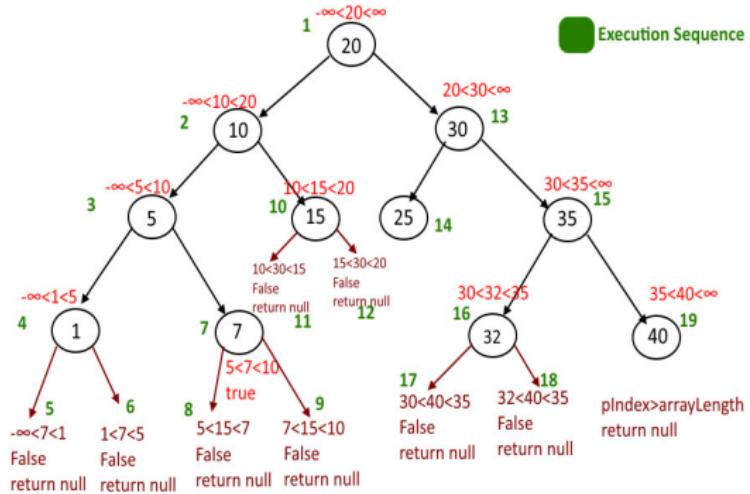
Percorso em pré-ordem

O primeiro método, conhecido como percurso em pré-ordem, implica em executar recursivamente os três passos na seguinte ordem:

- ① Visitar a raiz (imprimir esta);
- ② Percorrer a subárvore da esquerda em pré-ordem;
- ③ Percorre a subárvore da direita em pré-ordem.

Exemplo em pré-ordem

```
int[] preOrder = { 20, 10, 5, 1, 7, 15, 30, 25, 35, 32, 40 };
```



Outros métodos:

In-ordem:

De modo recursivo, execute os 3 passos que se seguem:

- ① Percorrer a subárvore da esquerda em pré-ordem;
- ② Visitar a raiz (imprimir esta);
- ③ Percorre a subárvore da direita em pré-ordem.

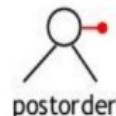
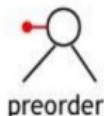
Pós-ordem:

De modo recursivo, execute os 3 passos que se seguem:

- ① Percorrer a subárvore da esquerda em pré-ordem;
- ② Percorre a subárvore da direita em pré-ordem;
- ③ Visitar a raiz (imprimir esta);

Resumo das Estratégias

The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a “flag” attached to each node, as follows:



To traverse the tree, collect the flags:

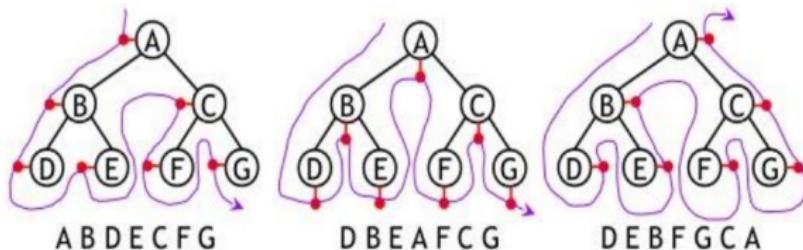


Figura 43: Resumo de *transversal tree* – percorrer árvores – reflexões

Exercícios

- Para as árvores construídas no exercício, mostre a sequência dos nós em: *pré-ordem*, *in-ordem* e *pós-ordem*

Exercícios

- ① Para as árvores construídas no exercício, mostre a sequência dos nós em: *pré-ordem*, *in-ordem* e *pós-ordem*
- ② Usando *pós-ordem* e uma pilha, verifique como seria para implementar um algoritmo que fizesse um *pretty-tree* de uma ABB

Exercícios

- ① Para as árvores construídas no exercício, mostre a sequência dos nós em: *pré-ordem*, *in-ordem* e *pós-ordem*
- ② Usando *pós-ordem* e uma pilha, verifique como seria para implementar um algoritmo que fizesse um *pretty-tree* de uma ABB
- ③ Ainda sobre o algoritmo do *pós-ordem*, qual outro método importante que se utiliza a mesma idéia?

Árvore Binária de Busca – Busca

```
BUSCA(ARVORE, CHAVE) {  
    SE ARVORE = NULO  
        return NULO  
  
    SE ARVORE->CHAVE = CHAVE  
        return ARVORE  
  
    SE CHAVE < ARVORE->CHAVE  
        return BUSCA(ARVORE->ESQ, CHAVE)  
    SENÃO  
        return BUSCA(ARVORE->DIR, CHAVE)  
}  
}
```

- Aqui há o método iterativo também implementado
- Importante pois é a base do conceito de percurso em ABBs

Árvore Binária de Busca – Remoção

A remoção de um nó se enquadra em um dos seguintes casos:

- ① Remoção de um nó folha (nenhum filho)
- ② Remoção de um nó com somente um filho
- ③ Remoção de um nó com dois filhos

Remoção em Figuras – Caso 1

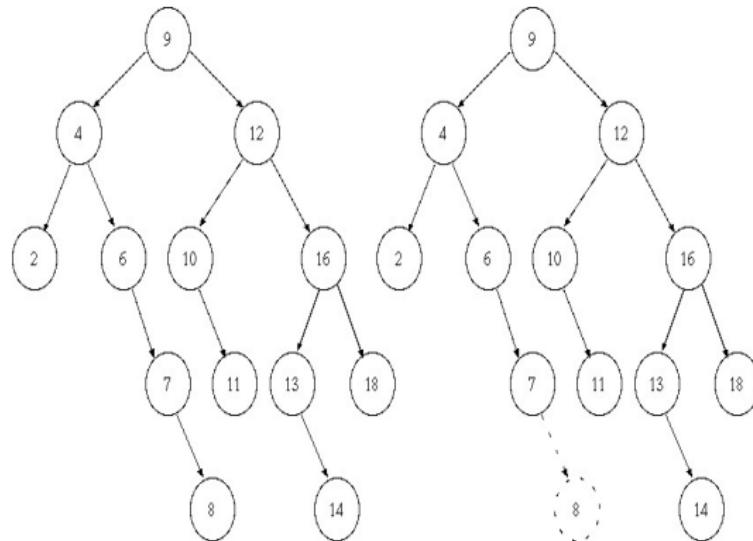


Figura 44: Remoção de uma folha – temp figure

Requisito: ser uma ABB!

Remoção em Figuras – Caso 2

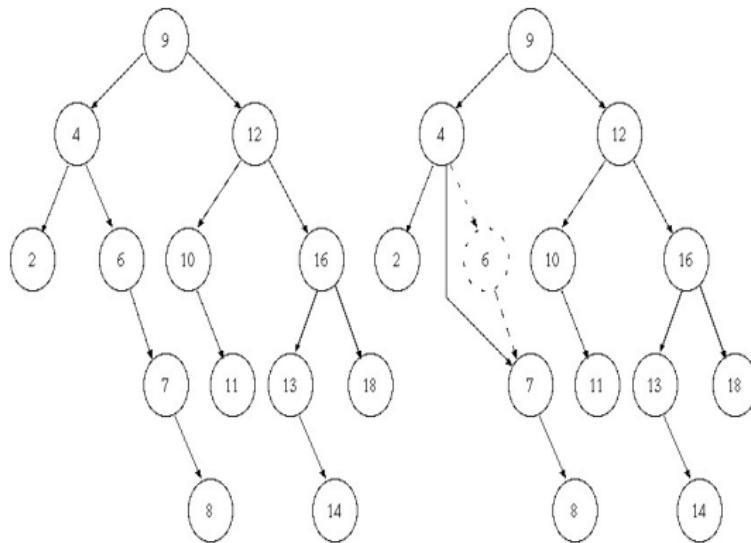


Figura 45: Remoção de um nó sem um dos filhos – esquerda ou direita igual a NULL

Requisito: ser uma ABB!

Remoção em Figuras – Caso 3

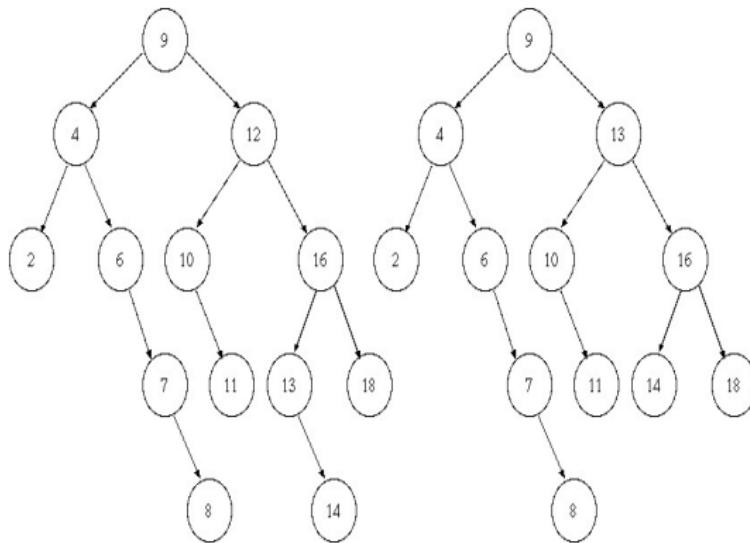


Figura 46: Remoção de um nó com sub-árvores em ambos os lados: remova o nó 12

Requisito: ser uma ABB!

Exemplificando – uma raiz sem um dos filhos

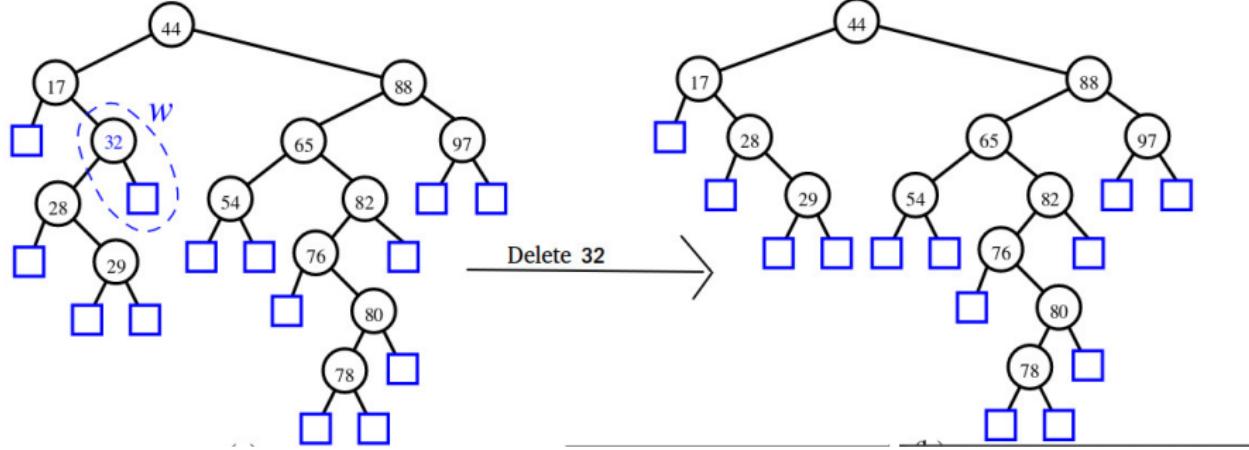


Figura 47: Quadrado AZUL é NULL

Requisito: ser uma ABB!

Exemplificando a Remoção – uma raiz com dois filhos

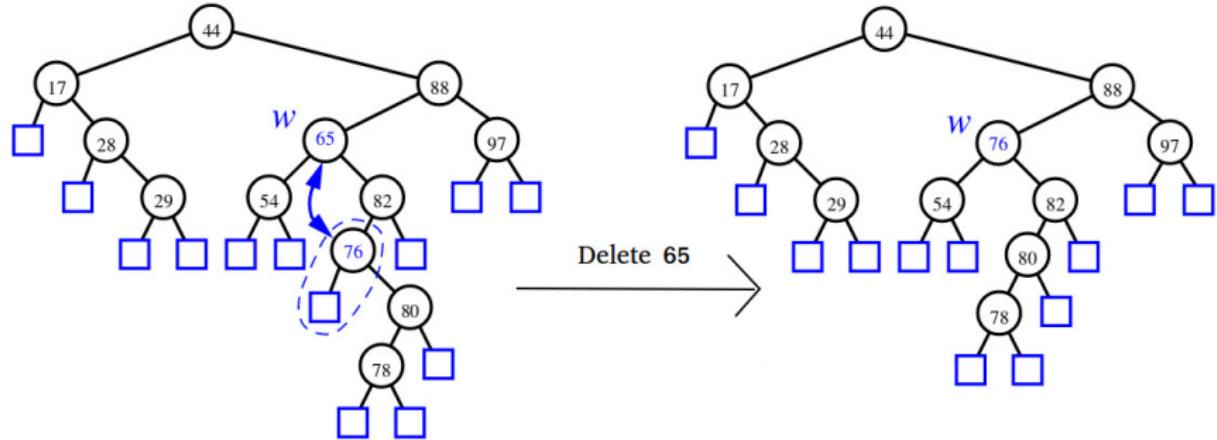


Figura 48: Quadrado AZUL é NULL

- A nova raiz será a menor chave à esquerda, da sub-árvore a direita do nó que se deseja excluir!
- Requisito: ser uma ABB!

Finalizando a remoção – último exemplo

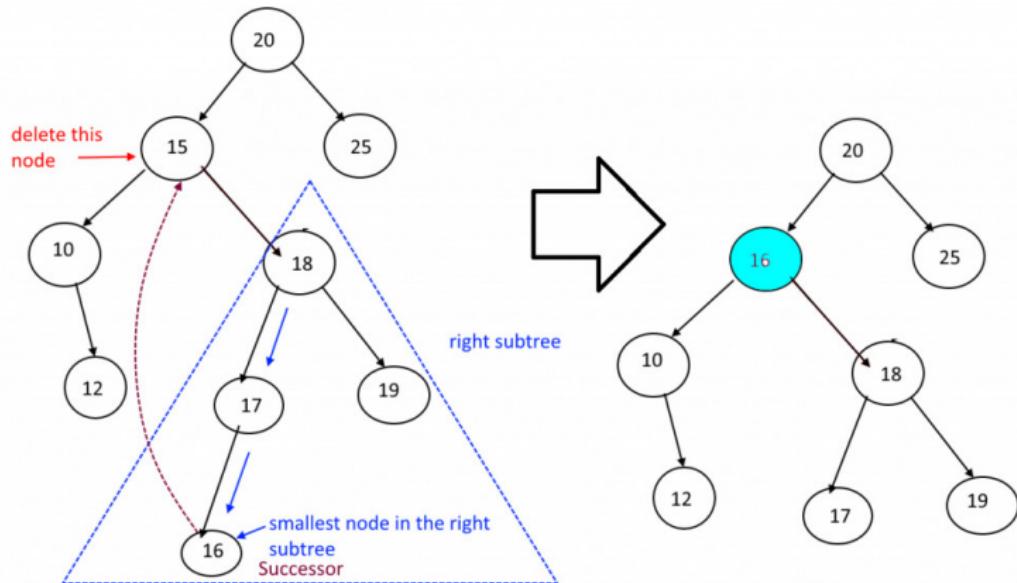


Figura 49: A idéia vale para os dois lados: é a menor chave à esquerda da sub-árvore a direita!

Reflexões

- A remoção é a base das partes mais difíceis de ABs
- Pois o próximo assunto é *balanceamento*
- Implemente este método, pois tem um bom grau de dificuldade!

Balanceamento

- Uma árvore binária de busca balanceada garante operações de busca, inserção e remoção com complexidade $O(\log n)$, onde n é o número de nós, o que a torna atrativa para diversas aplicações.
- Determinadas sequências de inserções ou remoções podem fazer com que uma ABB fique desbalanceada, tornando suas operações $O(n)$.
- Para verificar se uma árvore (ABB) está desbalanceada é necessário algumas **métricas!**

Balanceamento

- Uma árvore binária de busca balanceada garante operações de busca, inserção e remoção com complexidade $O(\log n)$, onde n é o número de nós, o que a torna atrativa para diversas aplicações.
- Determinadas sequências de inserções ou remoções podem fazer com que uma ABB fique desbalanceada, tornando suas operações $O(n)$.
- Para verificar se uma árvore (ABB) está desbalanceada é necessário algumas **métricas!**
- Por exemplo: número de nós por sub-árvores, alturas, etc

Cálculo da Altura

A altura de um nó ou da árvore é a maior profundidade de uma de suas sub-árvore a esquerda ou direita. As folhas tem altura 0.
Um algoritmo é dado por:

```
ALTURA(ARVORE) {  
    SE ARVORE = NULO  
        return -1  
  
    A1 = ALTURA(ARVORE->DIR)  
    A2 = ALTURA(ARVORE->ESQ)  
  
    return maior(A1, A2) + 1  
}
```

Lembre: a raiz da árvore tem a maior altura!

Cálculo da Altura

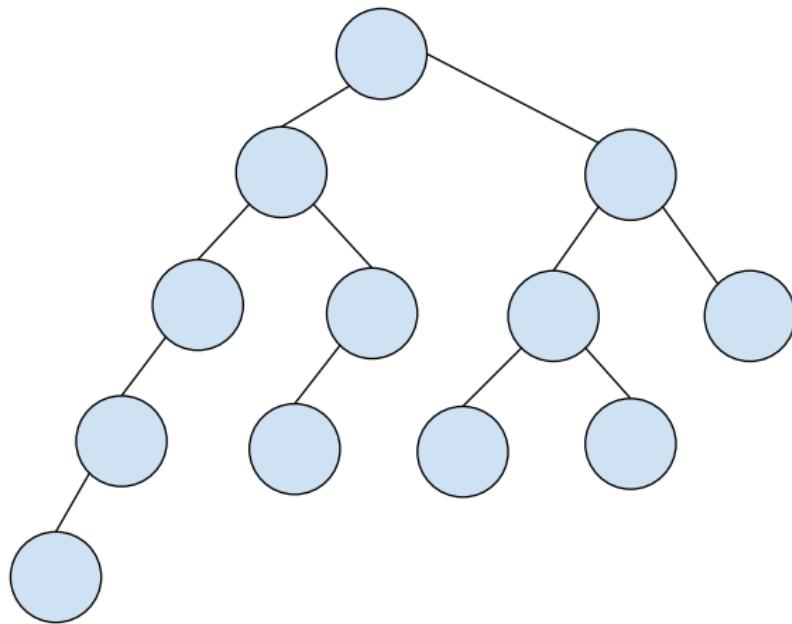


Figura 50: Exercício: determine a altura de cada subárvore.

Cálculo da Altura

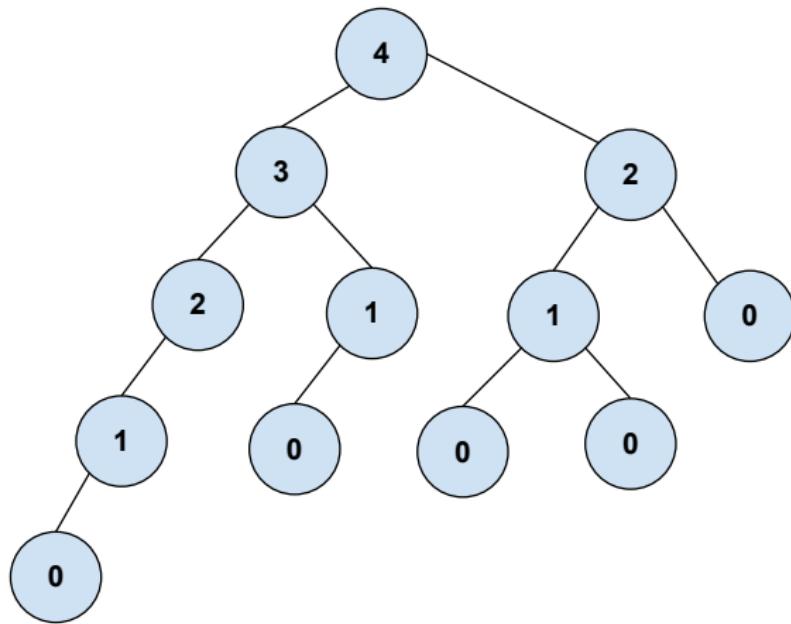


Figura 51: Resposta do exercício.

Cálculo do Fator de Balanceamento

O fator de balanceamento de uma árvore (ou nó, por definição é uma árvore) é a diferença entre as alturas de suas sub-árvore a esquerda ou direita.

Um algoritmo é dado por:

```
FB(ARVORE) {  
    A1 = ALTURA(ARVORE->ESQ)  
    A2 = ALTURA(ARVORE->DIR)  
    return A1 - A2  
}
```

Atenção: o cálculo é por nó, onde este define uma árvore recursiva!

Balanceamento

- Uma ABB está balanceada quando cada nó possui um FB igual a -1, 0 ou 1
- A diferença das alturas entre dois nós no mesmo nível é um valor absoluto
- Uma inserção ou remoção pode tornar uma árvore desbalanceada, necessitando de **rotações** para o seu balanceamento

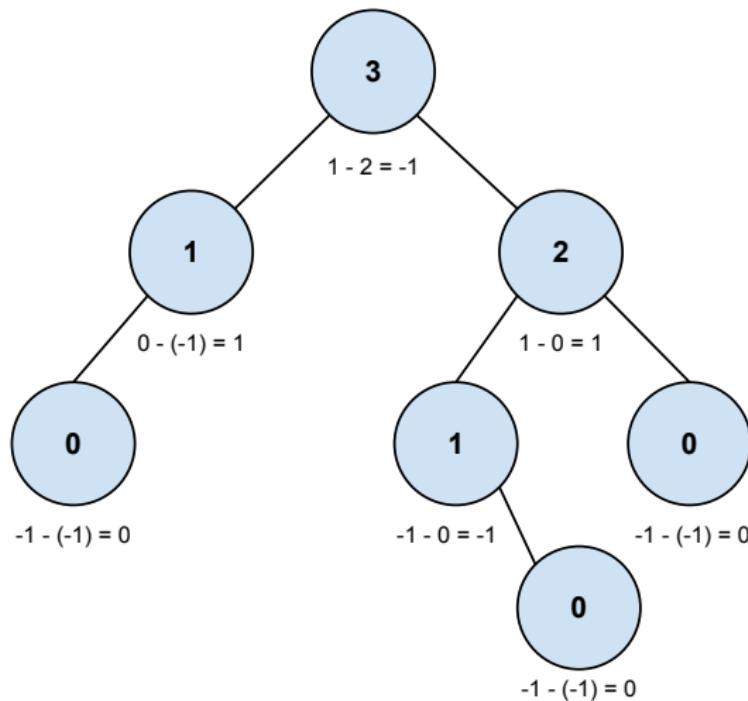
Balanceamento

- Uma ABB está balanceada quando cada nó possui um FB igual a -1, 0 ou 1
- A diferença das alturas entre dois nós no mesmo nível é um valor absoluto
- Uma inserção ou remoção pode tornar uma árvore desbalanceada, necessitando de **rotações** para o seu balanceamento
- Por convenção: $A_{esquerda} - A_{direita}$
- Isto é: se $FB < 0$ então $A_{esquerda} < A_{direita}$

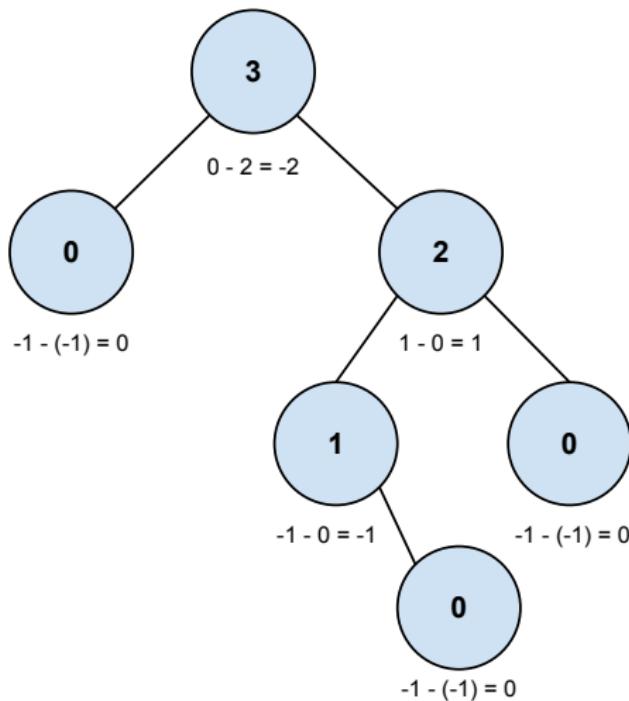
Balanceamento

- Uma ABB está balanceada quando cada nó possui um FB igual a -1, 0 ou 1
- A diferença das alturas entre dois nós no mesmo nível é um valor absoluto
- Uma inserção ou remoção pode tornar uma árvore desbalanceada, necessitando de **rotações** para o seu balanceamento
- Por convenção: $A_{esquerda} - A_{direita}$
- Isto é: se $FB < 0$ então $A_{esquerda} < A_{direita}$
- Define-se *nó de articulação* ou *pivô* o nó em que vão ocorrer rotações, esquerda ou a direita

Exemplo de ABB Balanceada



Exemplo de ABB Desbalanceada

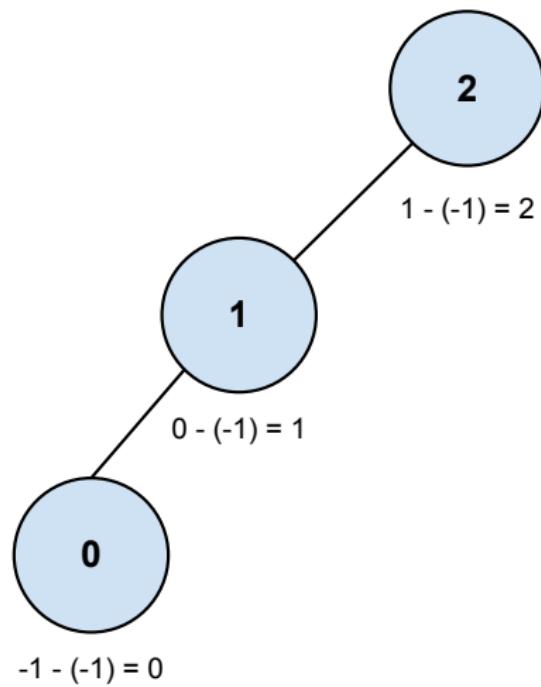


Operação de rotação

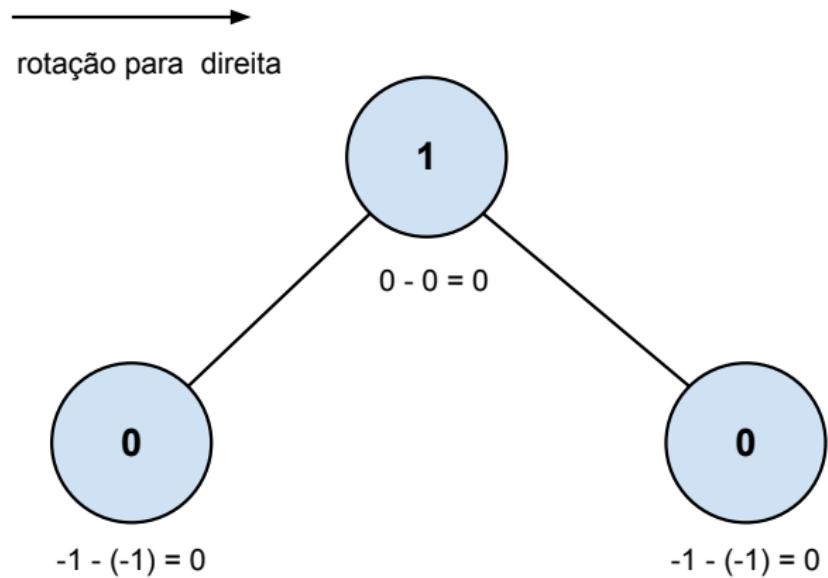
```
ROTACAO_DIREITA(RAIZ) {  
    PIVO      = RAIZ->ESQ  
    RAIZ->ESQ = PIVO->DIR  
    PIVO->DIR = RAIZ  
    RAIZ      = PIVO  
}
```

```
ROTACAO_ESQUERDA(RAIZ) {  
    PIVO      = RAIZ->DIR  
    RAIZ->DIR = PIVO->ESQ  
    PIVO->ESQ = RAIZ  
    RAIZ      = PIVO  
}
```

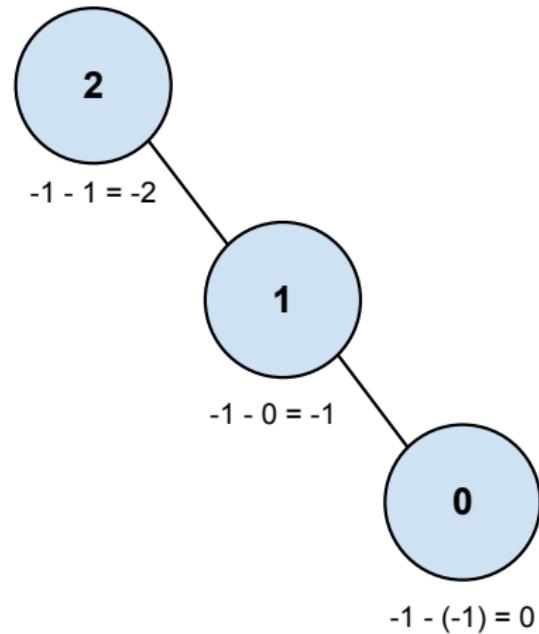
Rotação para Direita



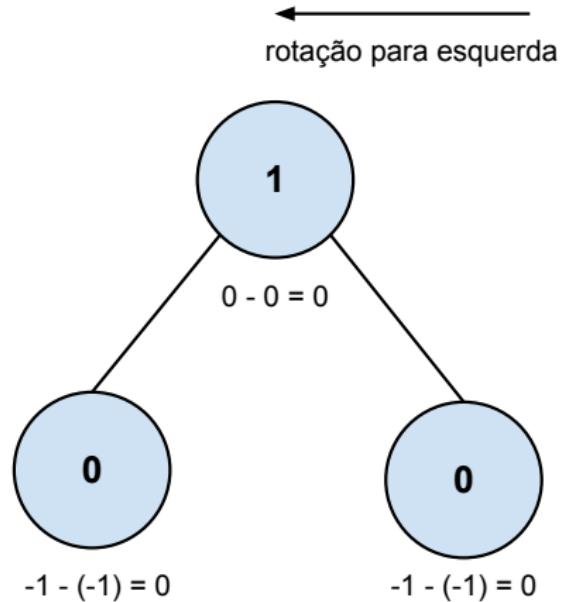
Rotação para Direita



Rotação para Esquerda



Rotação para Esquerda



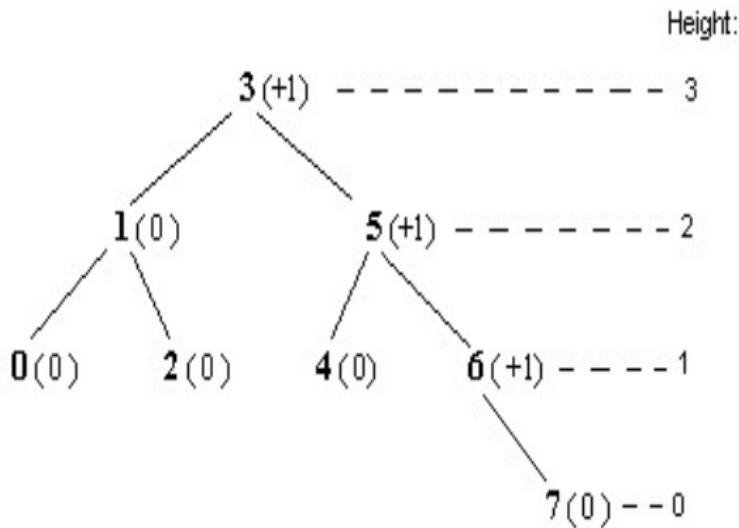
Árvores AVL

- **AVL** desenvolvida por G. M. Adelson-Velskii and E. M. Landis
- Critério do fator de balanceamento de -1 a $+1$ deve ser seguido em cada nó da árvore
- Garante o balanceamento da árvore ao realizar rotações após cada inserção ou remoção na ABB
- As operações são análogas a busca de um balanceamento
- Relembrando: se $FB > 0$ ramo esquerdo mais longo, caso $FB < 0$ ramo direito é maior!

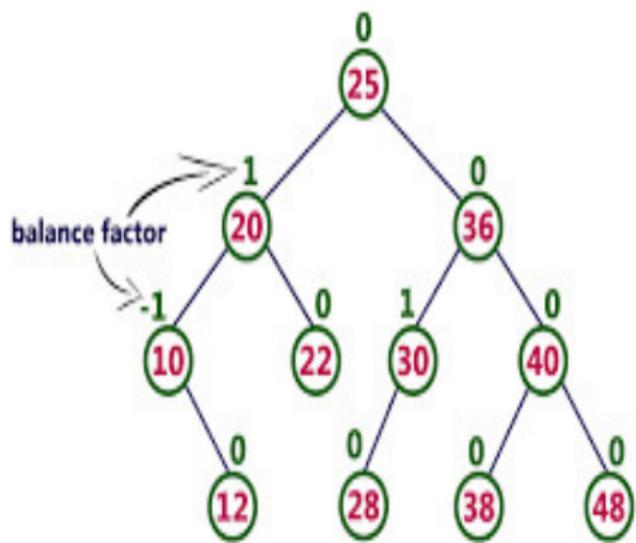
Árvores AVL

- **AVL** desenvolvida por G. M. Adelson-Velskii and E. M. Landis
- Critério do fator de balanceamento de -1 a $+1$ deve ser seguido em cada nó da árvore
- Garante o balanceamento da árvore ao realizar rotações após cada inserção ou remoção na ABB
- As operações são análogas a busca de um balanceamento
- Relembrando: se $FB > 0$ ramo esquerdo mais longo, caso $FB < 0$ ramo direito é maior!
- **Acompanhe as explicações de sala de aula**

Exemplo de AVL e Altura

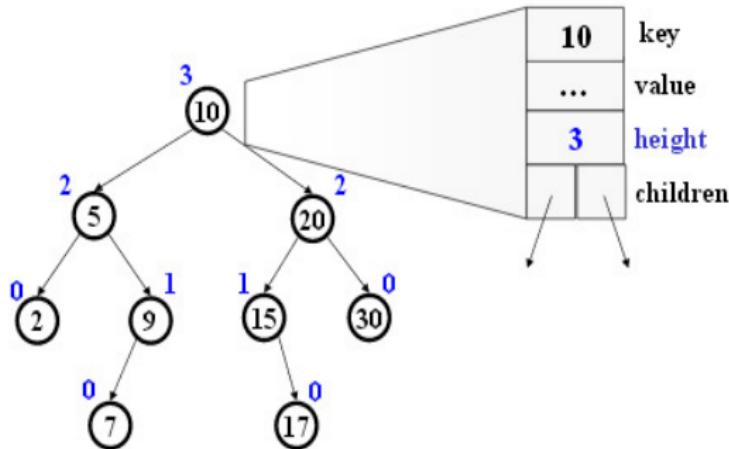


Exemplo de AVL e FB



Estrutura do Nó – facilidades na implementação da AVL

An AVL Tree



Um campo a mais: a altura corrente do nó!

Balanceamento – Inserção

```
BALANCEAMENTO(RAIZ) {  
    SE FB(RAIZ) = -2 ENTÃO  
        SE FB(RAIZ->DIR) = -1 ENTÃO  
            ROTACAO_ESQUERDA(RAIZ)  
        SENÃO  
            ROTACAO_DIREITA(RAIZ->DIR)  
            ROTACAO_ESQUERDA(RAIZ)  
    SENÃO SE FB(RAIZ) = 2 ENTÃO  
        SE FB(RAIZ->ESQ) = 1 ENTÃO  
            ROTACAO_DIREITA(RAIZ)  
        SENÃO  
            ROTACAO_ESQUERDA(RAIZ->DIR)  
            ROTACAO_DIREITA(RAIZ)  
}  
}
```

Balanceamento – Conclusões Parciais

- Para que as árvores AVL tenham um bom desempenho, é essencial que o balanceamento seja calculado eficientemente, isto é, sem a necessidade de percorrer toda a árvore após cada modificação
- Manter a árvore estritamente balanceada após cada modificação tem seu preço (desempenho = custo computacional).
- Árvores AVL são utilizadas normalmente onde o número de consultas é muito maior do que o número de inserções e remoções e quando a localidade de informação não é importante
- Estas conclusões serão verificadas no projeto final.

Resumindo as rotações necessárias nas AVLs:

A cada inserção há um rebalanceamento. Após uma inserção (vai valer para exclusão) os seguintes casos violam uma AVL. Logo, seguem os balanceamentos necessários:

Caso 1: uma articulação na sub-árvore à esquerda do filho à esquerda de X (simples)

Caso 2: uma articulação na sub-árvore à direita do filho à esquerda de X (simples)

Caso 3: uma articulação na sub-árvore à esquerda do filho à direita de X (dupla)

Caso 4: uma articulação na sub-árvore à direita do filho à direita de X (dupla)

Lembrar a convenção do FB!

Rotação para Esquerda – LL

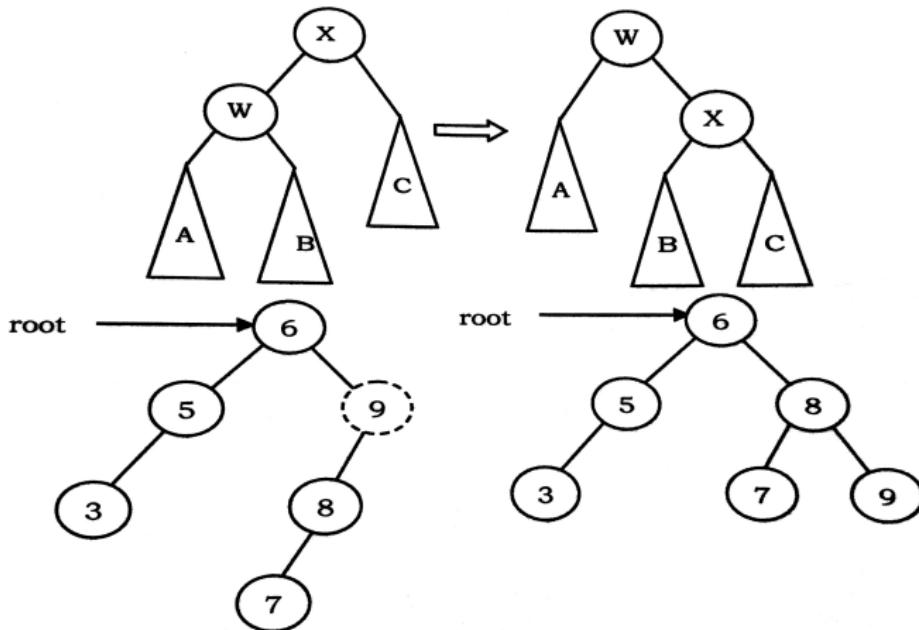


Figura 52: Algum nó tem o lado esquerdo maior ou igual a 2, tornando-a desbalanceada. No exemplo: nó 9

Rotação para Direita – RR

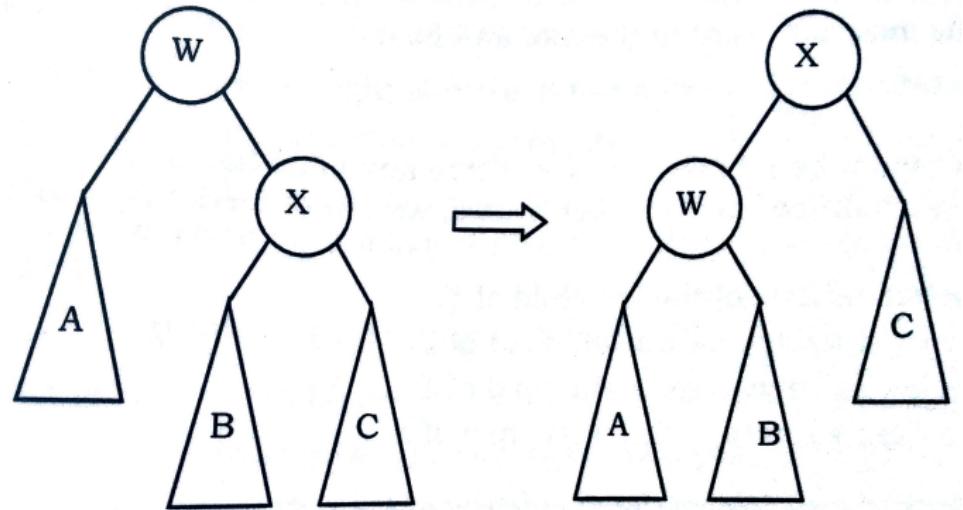


Figura 53: Alguma inserção na sub-árvore a direita, a partir do nó X

Rotação para Direita – RR – Exemplo

Time Complexity: O(1). Space Complexity: O(1).

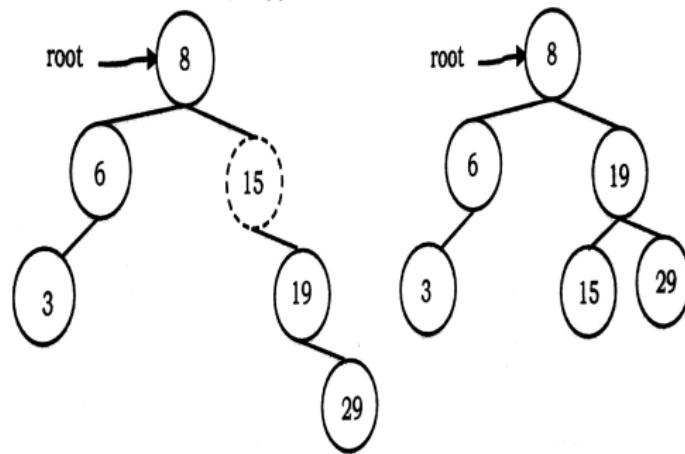
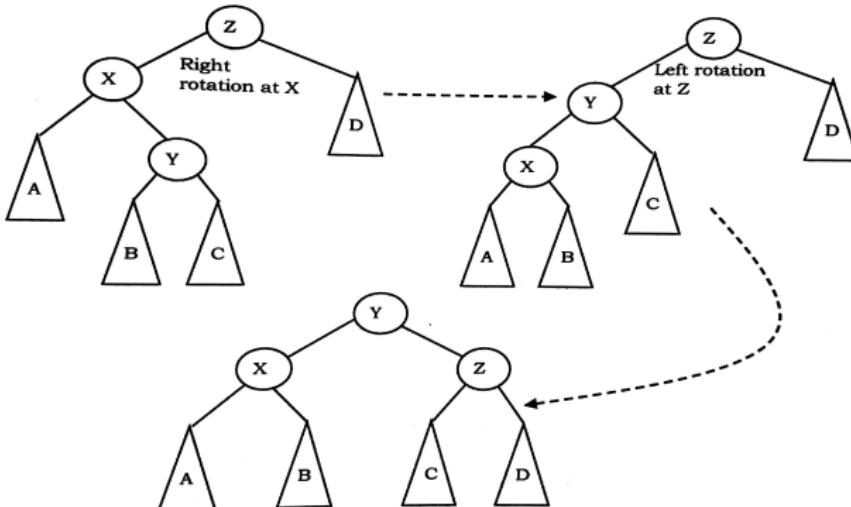


Figura 54: Algum nó tem o lado direito com altura maior ou igual a 2, tornando-a desbalanceada. Neste exemplo, o nó **15** tem um $FB > 2$. Pode ter ocorrido a inserção do 15, 19 ou 29!

Rotação Dupla Esquerda–Direita – LR

Double Rotations

Left Right Rotation (LR Rotation) [Case-2]: For case-2 and case-3 single rotation does not fix the problem. We need to perform two rotations.



As an example, let us consider the following tree: The insertion of 7 is creating the case-2 scenario and the right side tree is the one after the double rotation.

Figura 55: Uma inserção na sub-árvore interna **Y** vai provocar um desbalanceamento. Neste caso, duas rotações são necessárias.

Rotação Dupla Esquerda–Direita – LR – Exemplo

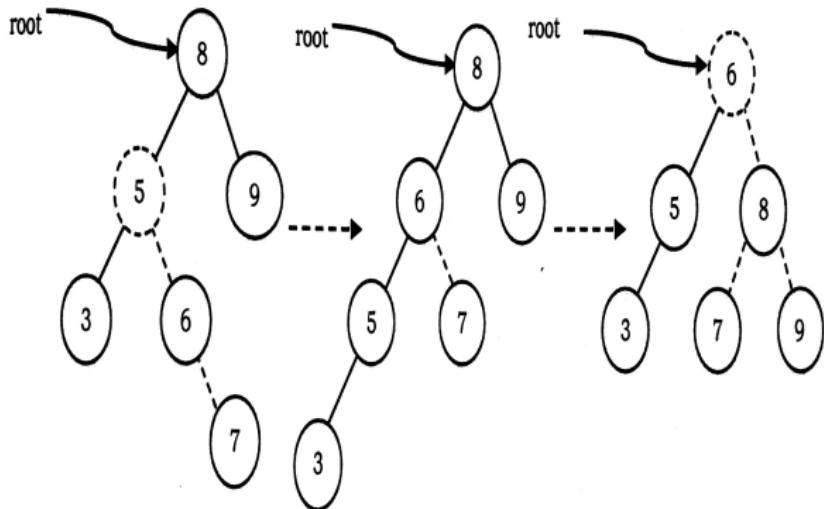


Figura 56: A inserção do nó 7 (ou 6) gerou desbalanceamento que necessita de duas rotações.

Rotação Dupla Direita–Esquerda – RL

Right Left Rotation (RL Rotation) [Case-3]: Similar to case-2, we need to perform two rotations to scenario.

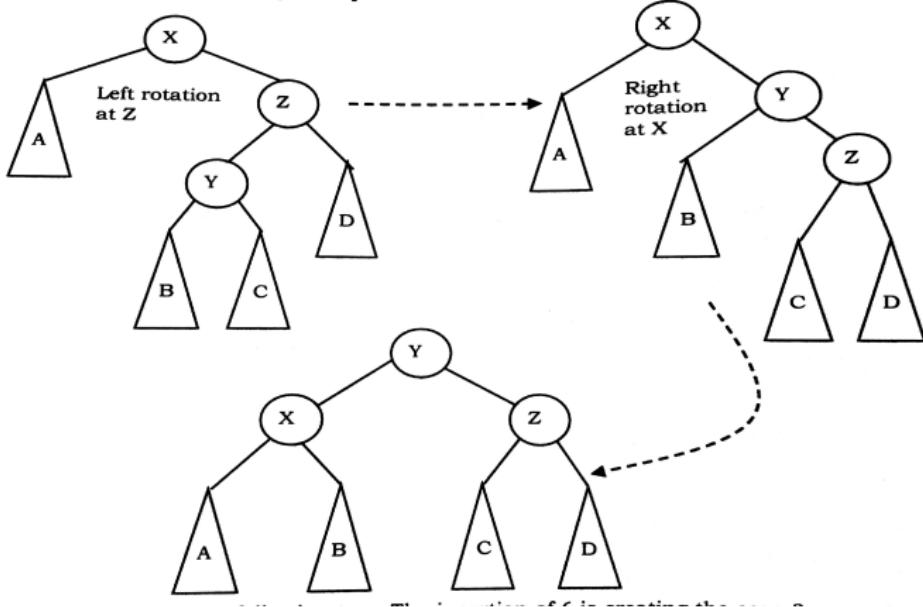


Figura 57: Uma inserção na sub-árvore **interna–esquerda** Y vai provocar um desbalanceamento. Neste caso, duas rotações são necessárias.

Rotação Dupla Direita–Esquerda – RL – Exemplo

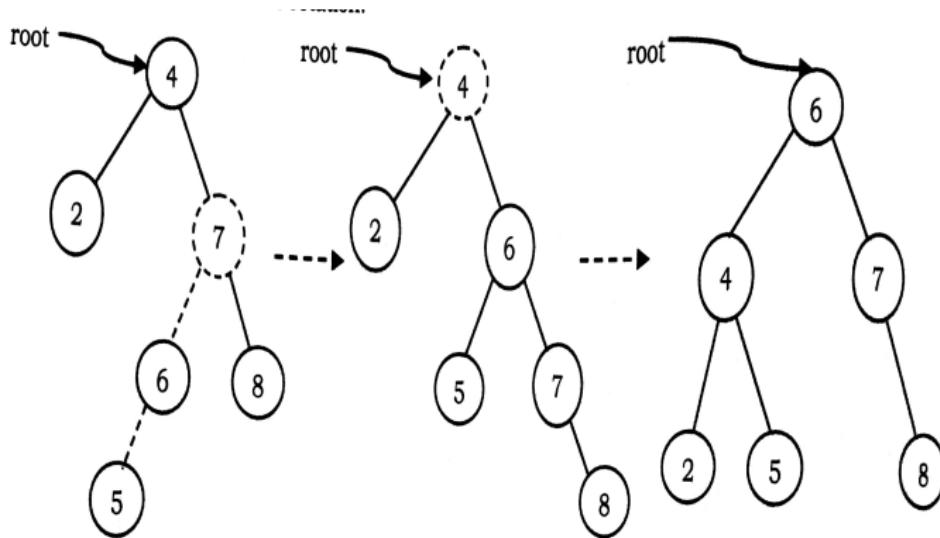


Figura 58: A inserção dos nós 5 (ou 6) gerou um desbalanceamento que necessita de duas rotações.

Resumo das AVLs

Finalizando AVLs

- Não são as mais eficazes, mas são bem didáticas (leia-se: elegância)

Resumo das AVLs

Finalizando AVLs

- Não são as mais eficazes, mas são bem didáticas (leia-se: elegância)
- Um balanceamento após cada operação de inserção e remoção, torna a ALV lenta!

Resumo das AVLs

Finalizando AVLs

- Não são as mais eficazes, mas são bem didáticas (leia-se: elegância)
- Um balanceamento após cada operação de inserção e remoção, torna a ALV lenta!
- Alternativas: árvores de espalhamento, Arvore B (férias)

Resumo das AVLs

Finalizando AVLs

- Não são as mais eficazes, mas são bem didáticas (leia-se: elegância)
- Um balanceamento após cada operação de inserção e remoção, torna a ALV lenta!
- Alternativas: árvores de espalhamento, Arvore B (férias)
- Construa uma ABB com a seguinte entrada: 1..7 (crescente) e transforme-a numa AVL
- Construa uma ABB com a seguinte entrada: 7..1 e transforme-a numa AVL
- Faça passo-a-passo para verificar se entendeste tudo!
- Acompanhe as explicações em aula (ou no vídeo da FUVEST, profs. Norton e Luciano)

Árvore de Espalhamento

- Uma alternativa de melhorar os procedimentos de AVL
- Reestrutura a árvore em cada operação de inserção, busca ou remoção por meio de operações de rotação
- Nome original: *splay tree* [?]. Não confundir com a Árvore N-Ária de Espalhamento (ANE) criada por professores da UDESC

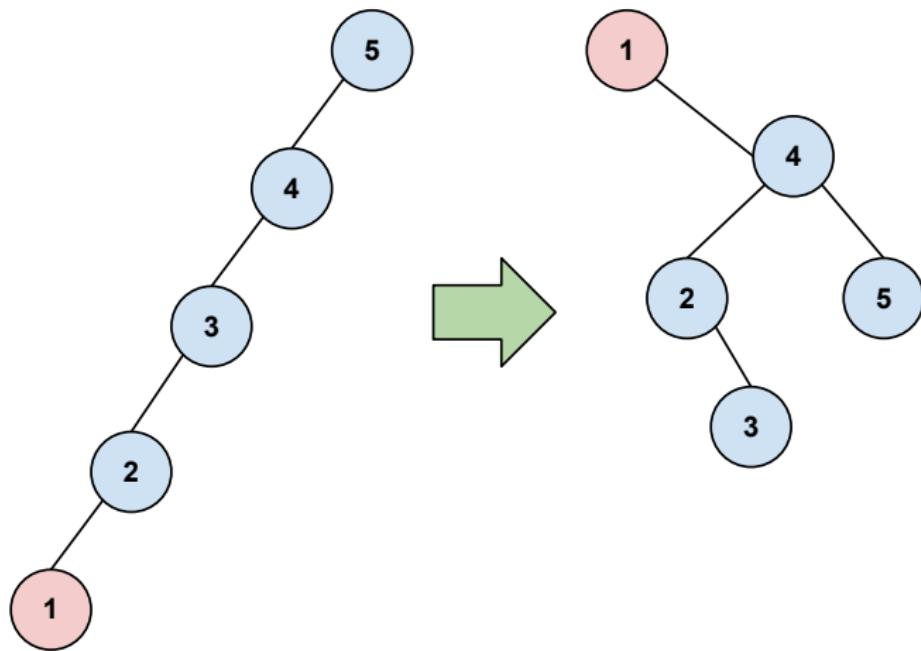
Árvore de Espalhamento

- Evita a repetição de casos ruins [$O(n)$] devido ao seu rebalanceamento natural
- Não realiza o cálculo de fatores de balanceamento, simplificando sua implementação
- Pior caso para uma operação se mantém $O(n)$, mas, ao considerar uma cadeia de operações, *garante* uma complexidade amortizada de $O(\log n)$ para suas operações básicas

Árvore de Espalhamento

- Se baseia na operação de espalhamento, que utiliza rotações para mover uma determinada chave até a raiz
- A sua complexidade $O(\log n)$ em uma análise amortizada é garantida pelas rotações efetuadas, o que a difere do uso simples de heurísticas como o *mover para a raiz*

Exemplo - Espalhamento pela chave 1



Operações Básicas

Espalhamento Move a chave desejada para a raiz por uma sequência bem definida de operações de rotação

Busca Busca uma chave na árvore

Inserção Insere uma nova chave na árvore

Remoção Remove uma chave da árvore

Operações Básicas

- Uma árvore de espalhamento é uma árvore binária de busca válida, logo operações como os percursos (pré, em e pós ordem) são idênticas as operações em uma ABB
- As operações de inserção, busca e remoção podem ser definidas com base na operação de espalhamento

Árvore de Espalhamento - Busca

```
BUSCA(RAIZ, CHAVE) {  
    return ESPALHAMENTO(RAIZ, CHAVE)  
}
```

Árvore de Espalhamento - Inserção

```
INSERE(RAIZ, CHAVE) {  
    INSERE_ABB(RAIZ, CHAVE)  
    return ESPALHAMENTO(RAIZ, CHAVE)  
}
```

Árvore de Espalhamento - Remoção

```
REMOVE(RAIZ, CHAVE) {  
    RAIZ = ESPALHAMENTO(RAIZ, CHAVE)  
  
    SE RAIZ->DIR ENTÃO  
        AUX = ESPALHAMENTO(RAIZ->DIR, CHAVE)  
        AUX->ESQ = RAIZ->ESQ  
    SENÃO  
        AUX = RAIZ->ESQ  
  
    return AUX  
}
```

Estratégias de Espalhamento

Duas estratégias:

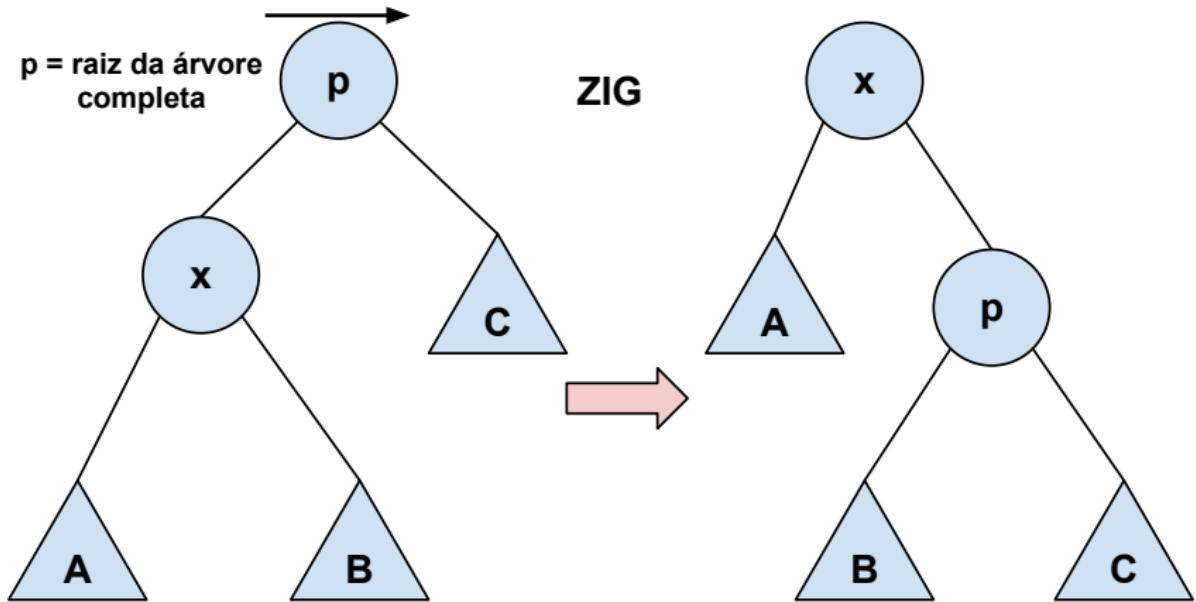
Bottom-Up Parte do nó acessado e o movimenta para a raiz da árvore por meio de rotações

Top-Down Parte do nó raiz, rotacionando e *removendo do caminho* os nós entre a raiz e o nó desejado, armazenando-os em duas árvores auxiliares, remontando a árvore completa na sua etapa final.

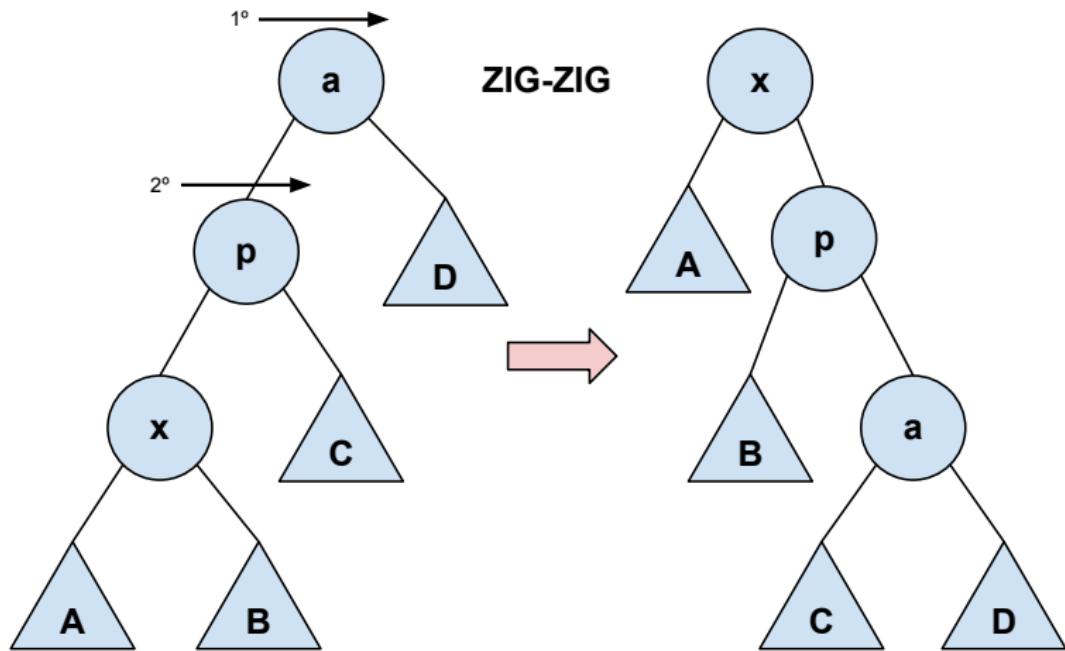
Espalhamento Bottom-Up

- Na estratégia Bottom-Up, a operação de espalhamento realiza rotações subindo gradativamente de níveis, a partir da chave desejada
- Enquanto a chave não estiver na raiz, deve-se verificar qual o caso aplicável (ZIG, ZIG-ZIG ou ZIG-ZAG) e realizar as rotações necessárias

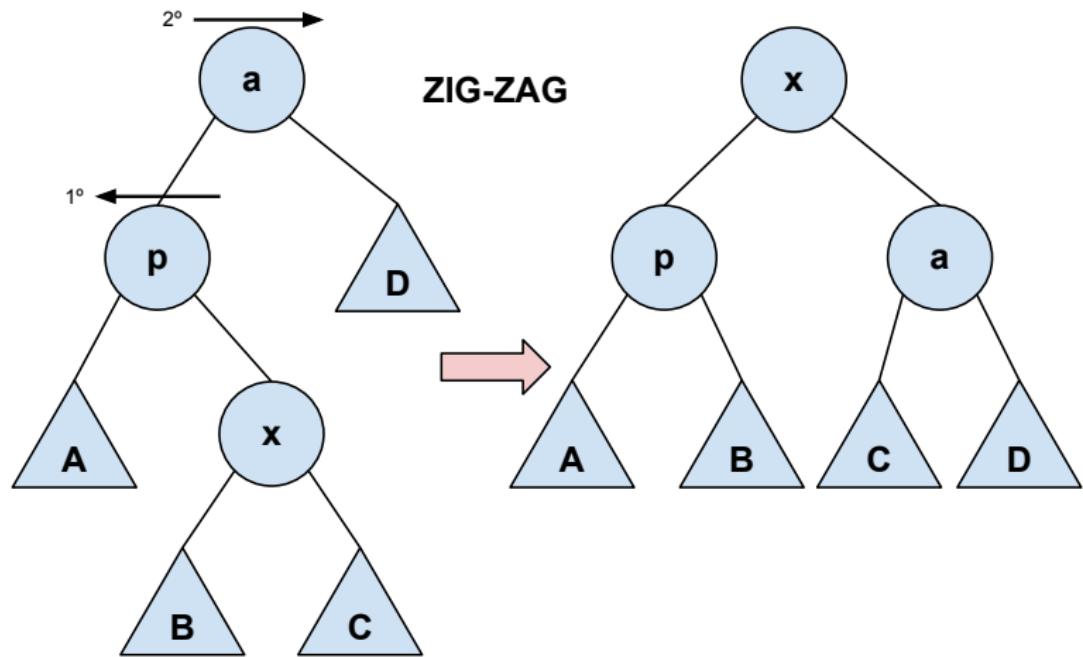
Caso 1: ZIG



Caso 2: ZIG-ZIG



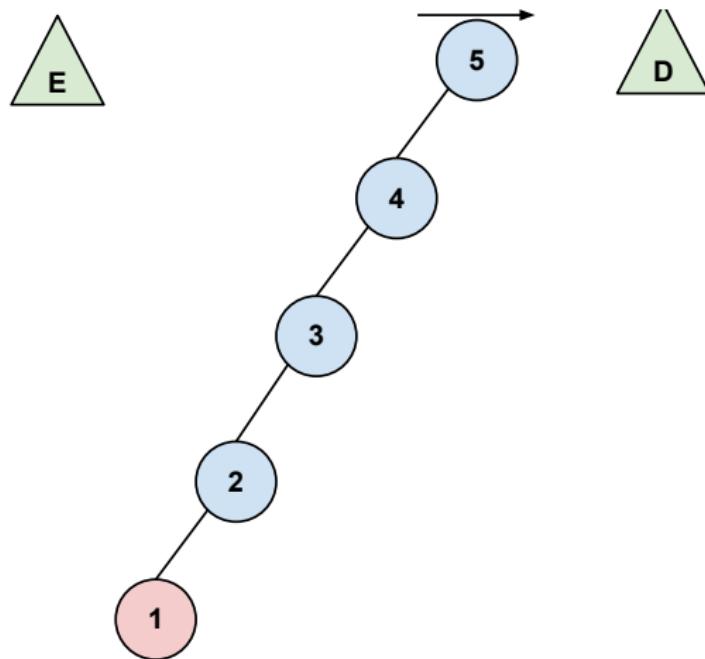
Caso 3: ZIG-ZAG



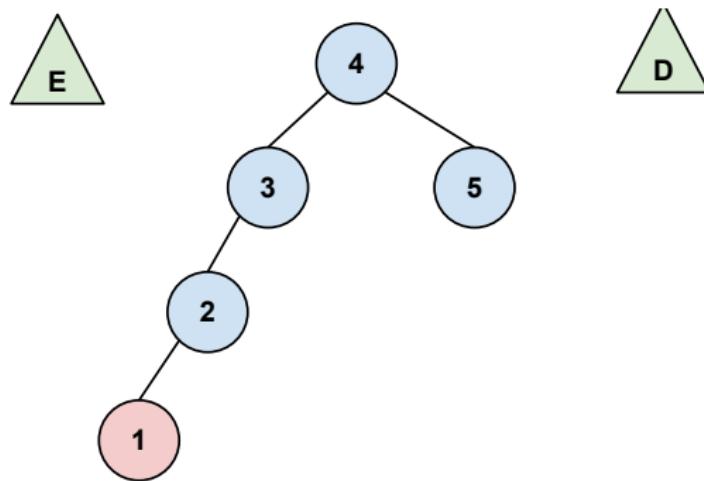
Espalhamento Top-Down

- Na estratégia Top-Down as chaves que estão no caminho da chave desejada para a raiz são rotacionadas e removidas para árvores auxiliares seguindo uma sequência de operações bem definidas
- Quando a chave desejada chega até a raiz, a árvore é remontada pelo retorno das chaves removidas

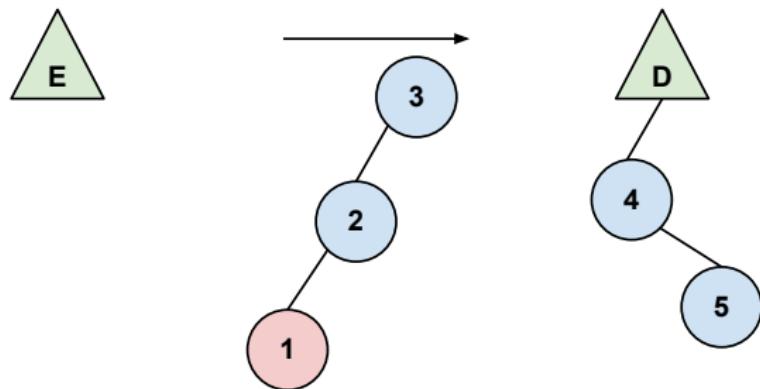
Exemplo: Top-Down 1/6



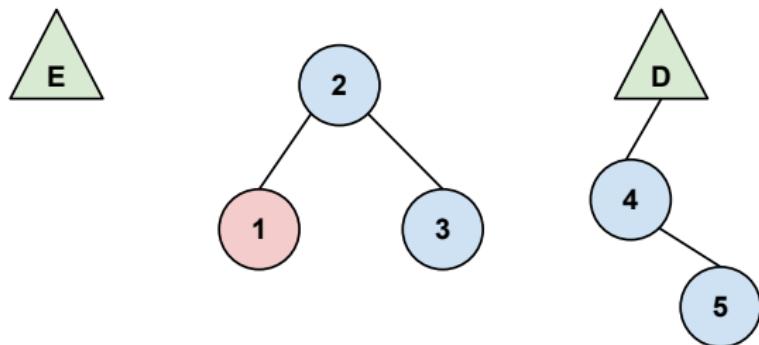
Exemplo: Top-Down 2/6



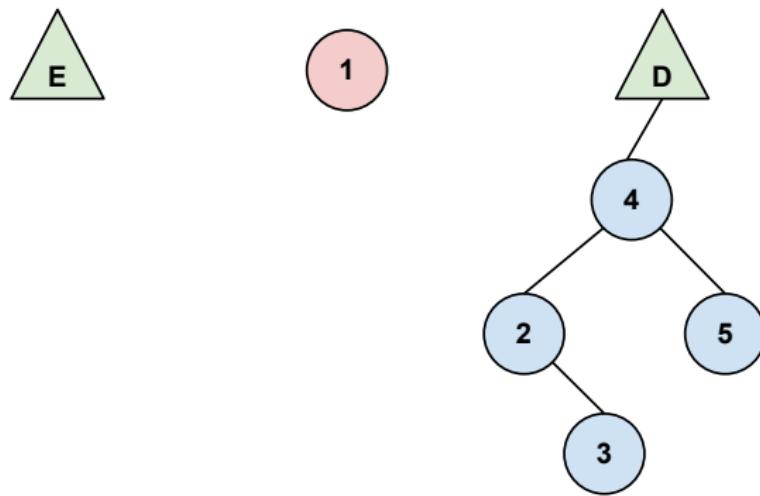
Exemplo: Top-Down 3/6



Exemplo: Top-Down 4/6



Exemplo: Top-Down 5/6



Exemplo: Top-Down 6/6

