

Estrutura de Dados

Hilton Cardoso Marins Junior
hiltonmarins@gmail.com
2024

1.	INTRODUÇÃO	4
2.	ALOCÇÃO DE MEMÓRIA	5
2.2.	ALOCÇÃO DINÂMICA DE MEMÓRIA.....	5
2.3.	AS PARTES DE UMA VARIÁVEL PONTEIRO.....	9
2.4.	CUIDADO AO MANIPULAR PONTEIROS.....	10
2.5.	ARITMÉTICA DE PONTEIROS.....	10
2.6.	ARITMÉTICA DE PONTEIROS X INDEXAÇÃO.....	12
2.7.	PASSAGEM DE PARÂMETROS: POR VALOR X POR REFERÊNCIA.....	14
2.8.	PASSANDO VETOR COMO PARÂMETRO.....	16
3.	LISTA LINEAR	18
3.2.	LISTA LINEAR POR CONTIGUIDADE.....	18
3.3.	LISTA LINEAR POR ENCADEAMENTO	20
3.4.	VARIAÇÕES DE LISTAS ENCADEADAS.....	27
3.5.	ATIVIDADES.....	28
4.	LISTAS LINEARES COM DISCIPLINA DE ACESSO	29
4.2.	LIFO - LAST IN FIRST OUT.....	29
	<i>Pilha de execução de um programa</i>	<i>29</i>
	<i>Operações com pilha</i>	<i>31</i>
	<i>Representações de uma pilha.....</i>	<i>31</i>
4.3.	FIFO – FIRST IN FIRST OUT.....	32
	<i>Operações com uma fila</i>	<i>32</i>
	<i>Representações de uma fila</i>	<i>32</i>
4.4.	ATIVIDADES.....	34
5.	RECURSIVIDADE	37
5.2.	SOMATÓRIO DE INTEIROS.....	37
5.3.	CÁLCULO DO FATORIAL	41
5.4.	ATIVIDADES.....	41
6.	ÁRVORES	43
7.	ÁRVORE BINÁRIA	45
7.2.	TIPOS DE ÁRVORES BINÁRIAS.....	45
7.3.	A IMPORTÂNCIA DA ALTURA MÍNIMA.....	46
7.4.	OPERAÇÕES.....	47
7.5.	PERCURSO EM PROFUNDIDADE.....	49
7.6.	PERCURSO EM LARGURA	51
7.7.	ATIVIDADES.....	52
8.	ÁRVORE BINÁRIA DE BUSCA	55
8.2.	OPERAÇÕES.....	55
	<i>Busca</i>	<i>56</i>
	<i>Inserção</i>	<i>56</i>
8.3.	ATIVIDADES.....	57
9.	ÁRVORE AVL.....	62
9.2.	MANTENDO-SE AVL.....	63
	<i>Fator de balanceamento</i>	<i>63</i>
	<i>Rotação simples</i>	<i>64</i>
	<i>Rotação dupla</i>	<i>65</i>
9.3.	OPERAÇÕES.....	66
9.4.	ATIVIDADES.....	67

10. ÁRVORE B	72
10.2. ORDEM.....	73
10.3. ESTRUTURA DE UMA PÁGINA.....	74
10.4. BUSCA	74
10.5. INSERÇÃO.....	75

1. Introdução

Em nossos estudos anteriores sobre programação enfatizamos o **controle do fluxo de execução** de um programa, além dos conceitos básicos de programação.

Agora vamos além! Pretendemos nos dedicar à **organização dos dados** manipulados por um programa de modo a incorporar **eficiência** ao processamento. Se esta organização é projetada adequadamente, certas operações críticas poderão ser executadas com o menor custo possível de recursos, de **tempo e espaço** (memória).

Compreenderemos que a **eficiência** dos programas está diretamente ligada à **escolha da estrutura de dados mais adequada** para um determinado problema. Nossa preocupação será o custo envolvido em operações que armazenam e recuperam os dados manipulados pelos programas. Portanto estruturas de dados e algoritmos são temas fundamentais da ciência da computação, sendo utilizados nas mais diversas áreas do conhecimento e com os mais diferentes propósitos de aplicação.

Utilizamos **o termo estruturas de dados para designar as diferentes formas de organizar os dados a serem processados**. A organização e os métodos para manipular uma determinada estrutura de dados é o que lhe confere singularidade.

Bons estudos!

2. Alocação de memória

Os programas desenvolvidos até o momento realizam alocação de memória através da instrução que declara variável. A correspondente liberação da memória alocada ocorrerá automaticamente quando da finalização do bloco de instruções onde a variável tenha sido declarada, de acordo com o seu escopo. Esse tipo de alocação é conhecido como **alocação estática de memória**.

Essa forma de alocar memória apresenta uma limitação. Conheceremos um recurso capaz de incorporar mais flexibilidade aos programas no que se refere a alocação de memória. Seremos capazes de alocar e liberar espaço na memória quando necessário, durante a execução do programa. Trata-se da **alocação dinâmica de memória**.

2.2. Alocação dinâmica de memória

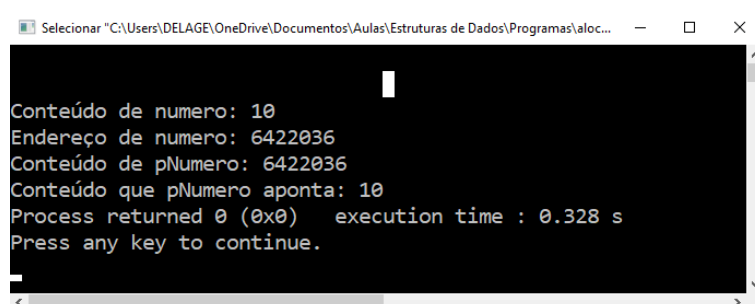
O tipo de dado que suporta esse sistema de alocação dinâmica de memória é o **ponteiro**, também conhecido como variável de referência ou apontador ou indicador. A compreensão exata de ponteiros está ligada diretamente ao fato de que **o conteúdo de uma variável do tipo ponteiro é o endereço do dado e não o dado propriamente dito**.

Analisaremos um programa para iniciarmos o entendimento sobre alocação dinâmica de memória e sua relação com os ponteiros.

alocacaoDinamica1.c

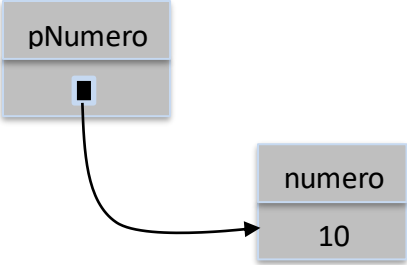
```
1  #include <stdio.h>
2  #include <locale.h>
3  main() {
4      setlocale(LC_ALL, "Portuguese");
5
6      int numero = 10;
7      int *pNumero = &numero;
8
9      printf("\n\nConteúdo de numero: %d", numero);
10     printf("\nEndereço de numero: %d", &numero);
11     printf("\nConteúdo de pNumero: %d", pNumero);
12     printf("\nConteúdo que pNumero aponta: %d", *pNumero);
13 }
```

Sugiro que execute o código acima, assim como todos os demais que veremos adiante em nossos estudos, e faça análises e simulações para melhor compreender. Você deverá obter uma saída como esta:



```
Selecionar "C:\Users\DELAG\OneDrive\Documentos\Aulas\Estruturas de Dados\Programas\aloc...
Conteúdo de numero: 10
Endereço de numero: 6422036
Conteúdo de pNumero: 6422036
Conteúdo que pNumero aponta: 10
Process returned 0 (0x0) execution time : 0.328 s
Press any key to continue.
```

Vamos aos comentários:

Linha	Comentário
7	<p>Lembre-se do que afirmamos anteriormente: o conteúdo de uma variável do tipo ponteiro é o endereço do dado e não o dado propriamente dito. Nesta linha, atribuímos o endereço da variável numero à variável pNumero, portanto pNumero passou a apontar para o conteúdo da variável numero, que no caso é 10.</p> <p>Veja abaixo o efeito dessa instrução.</p>  <p>Na linguagem C, o operador '*' antes do nome da variável, no momento da declaração, sinaliza que esta variável será um ponteiro, ou seja, não armazenará um valor do tipo declarado e sim um endereço para um local da memória que, este sim, armazenará um valor do tipo declarado.</p>
9	Simplesmente apresentamos o conteúdo da variável numero, que é 10.
10	Apresentamos o endereço de memória da variável numero. Já sabemos que o operador '&' precedendo uma variável, indica o seu endereço e não o seu conteúdo.
11	Apresentamos o conteúdo da variável pNumero, que é um endereço de memória, no caso o endereço da variável numero. Veja que a saída produzida por essa linha é idêntica à anterior, isso é uma prova de que pNumero aponta para a variável numero.
12	<p>O uso do operador '*' na declaração de uma variável indica que ela é um ponteiro. Já após ser declarada, o '*' indica que queremos fazer referência ao valor para o qual a variável aponta.</p> <p>Veja que na saída apresentada por essa linha visualiza-se o valor 10, que é o valor da variável numero e que é o valor para o qual pNumero está apontando, desta forma fizemos referência indireta à variável numero, através de pNumero.</p>

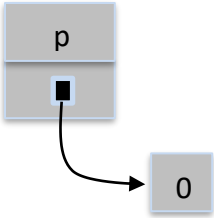
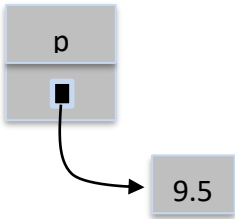
Você deve ter percebido que o ponteiro utilizado no código acima, utilizava um endereço já existente, já alocado de forma estática anteriormente. Agora veremos um exemplo de como alocar memória de forma dinâmica.

alocacaoDinamica2.c

```
1  #include <stdio.h>
2  #include <locale.h>
3
4  main() {
5      setlocale(LC_ALL, "Portuguese");
6
7      double *p;
8      p = (double*) malloc(8);
9
10     if (p == NULL) {
11         printf("Memória Insuficiente!");
12     }
13
14     *p = 9.5;
15
16     printf("\nEndereço do 1o. Byte: %d", p);
17     printf("\nValor double armazenado: %f", *p);
18     free(p);
19 }
```

Segue abaixo alguns comentários:

Linha	Comentário
7	<p>Na linguagem C, o operador ‘*’ antes do nome da variável, no momento da declaração, sinaliza que está variável será um ponteiro, ou seja, não armazenará um valor do tipo declarado e sim um endereço para um local da memória que, este sim, armazenará um valor do tipo declarado. Entretanto essa variável ainda não aponta para um endereço, veremos mais detalhes a seguir.</p> <p>Você já sabe que o compilador da linguagem C inicializa a variável, no momento da sua declaração, com um valor padrão, de acordo com o seu tipo. Para ponteiros esse valor padrão é a constante NULL, que indica endereço de memória 0, ou seja, memória inexistente, portanto, ponteiro com o conteúdo NULL indica que não aponta para nenhum endereço de memória.</p> <p>Efeito dessa instrução:</p> <div><div>p</div><div>NULL</div></div>
8	<p>Através da alocação dinâmica um programa pode obter memória enquanto está em execução. A função malloc() possui esse propósito.</p>

	<p>Cada vez que é feita uma solicitação de memória por <code>malloc()</code>, uma porção da memória livre restante é alocada.</p> <p>A função <code>malloc()</code> devolve um endereço de memória, onde o tipo de dado que será armazenado ainda está indefinido (<code>void</code>), o que significa que você pode atribuí-lo a qualquer tipo de ponteiro (desde que faça o casting corretamente). No nosso exemplo foi feito o casting para o tipo <code>double</code>.</p> <p>Outra particularidade de <code>malloc()</code> é a necessidade de informarmos a quantidade de bytes que queremos alocar. No nosso exemplo alocamos espaço para apenas um número real do tipo <code>double</code>, indicamos 8 bytes pois é o tamanho de um número do tipo <code>double</code> no computador em que esse programa foi testado.</p> <p>Efeito dessa instrução:</p>  <p>Diagrama: Um retângulo cinza com o rótulo 'p' no topo. Abaixo dele, um pequeno quadrado preto representa o ponteiro. Uma seta curva aponta desse ponteiro para um retângulo cinza contendo o valor '0'.</p> <p>Não é necessário saber o tamanho, em bytes, que cada tipo de dado ocupa na memória. Use função <code>sizeof()</code> para ajudá-lo. Uma das linhas abaixo poderiam substituir a linha 8.</p> <pre>p = (double*) malloc(sizeof(double)); //aloca 1 numero p = (double*) malloc(100 * sizeof(double)); //aloca 100 numeros</pre>
10	<p>Após uma chamada bem sucedida, <code>malloc()</code> devolve um ponteiro para o primeiro byte da região de memória alocada. Se não há memória disponível para satisfazer a requisição de <code>malloc()</code>, ocorre uma falha de alocação e <code>malloc()</code> devolve <code>NULL</code>. Por isso a necessidade dessa linha. Ela verifica se a alocação foi bem-sucedida.</p>
14	<p>Nesse ponto do programa, <code>p</code> está apontando para uma região já alocada, preparada para receber um número <code>double</code>, mas ainda não recebeu. Somente após a execução dessa linha é que essa região alocada receberá um valor <code>double</code>, que no caso é 9.5.</p> <p>Efeito dessa instrução:</p>  <p>Diagrama: Um retângulo cinza com o rótulo 'p' no topo. Abaixo dele, um pequeno quadrado preto representa o ponteiro. Uma seta curva aponta desse ponteiro para um retângulo cinza contendo o valor '9.5'.</p>

16	A saída será o conteúdo da variável p, ou seja, o endereço para onde p está apontando. Saiba que o endereço apresentado corresponde ao 1º byte dos 8 que foram alocados, ou seja, p não armazena os endereços de toda a região alocada, apenas o 1º endereço.
17	A presença do operador “*”, indica que estamos referenciando a parte de valor de p, ou seja, o valor para o qual p aponta. Que no caso é 9.5.
18	A função free() libera memória previamente alocada.

2.3. As partes de uma variável ponteiro

Concluiremos o entendimento básico sobre ponteiros. Para tanto vamos lembrar da sua característica fundamental: o conteúdo de uma variável do tipo ponteiro é o endereço do dado e não o dado propriamente dito.

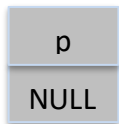
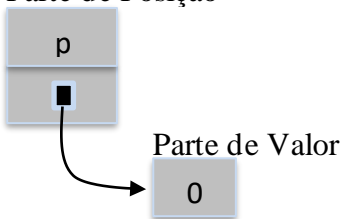
Isso implica no fato de que todo ponteiro é composto de duas partes: **posição e valor**. Vamos entender melhor a relação entre essas duas partes através da análise do trecho de código abaixo.

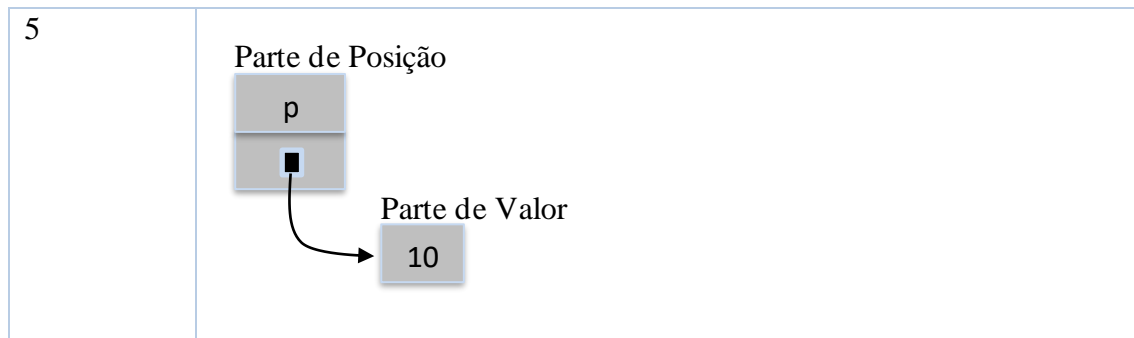
[alocacaoDinamica3.c](#)

```

1 main()
2 {
3     int *p;
4     p = (int*) malloc(sizeof(int));
5     *p = 10;
6 }
```

Veja os comentários a seguir:

Linha	Efeito produzido
3	<p>Parte de Posição</p> 
4	<p>Parte de Posição</p> 



2.4. Cuidado ao manipular ponteiros

Analise o código abaixo e tente identificar onde está o erro, execute-o para perceber melhor.

alocacaoDinamica4.c

```

1 main()
2 {
3     setlocale(LC_ALL, "Portuguese");
4
5     int *p;
6     *p = 100;
7
8     printf("\nParte de posição de p: %d", p);
9     printf("\nParte de valor de p: %d", *p);
10 }
```

Repare que as linhas 8 e 9 não foram executadas, ou seja, ocorreu um erro e a execução do programa foi abortada. O problema do código acima é que estamos atribuindo um valor (100) para uma posição da memória indefinida. Perceba que `p` não tem parte de valor, apenas de posição (que é `NULL`), ou seja, `p` foi declarada mas não alocada. Se `p` não tem parte de valor, como poderíamos referencia-la na linha 6?

Altere a linha 5 por esta: `int *p = (int) malloc(sizeof(int))`. Execute novamente o código e veja o resultado. Entenda que agora a variável `p` foi alocada corretamente na linha proposta acima, portanto a linha 6 não causará mais erro.

2.5. Aritmética de ponteiros

Existem apenas duas operações aritméticas que podemos realizar com ponteiros: adição e subtração. A operação de adição com ponteiros causa avanço na memória e subtração o contrário. Cada vez que um ponteiro é incrementado de uma unidade, ele passa a apontar para a posição de memória do próximo elemento de seu tipo base, e ao contrário quando decrementado. Analisemos o Código abaixo para melhor entendimento.

aritmeticaPonteiros1.c

```

1 main()
2 {
3     setlocale(LC_ALL, "Portuguese");
4
5     int *p = (int) malloc(sizeof(int) * 10);
```

```

6
7     int i;
8     for (i=0; i<10; i++){
9         printf("\nParte de posição do %do. inteiro: %d", i+1, p+i);
10    }
11 }

```

A saída produzida:

```

Parte de posição do 1o. inteiro: 1733808
Parte de posição do 2o. inteiro: 1733812
Parte de posição do 3o. inteiro: 1733816
Parte de posição do 4o. inteiro: 1733820
Parte de posição do 5o. inteiro: 1733824
Parte de posição do 6o. inteiro: 1733828
Parte de posição do 7o. inteiro: 1733832
Parte de posição do 8o. inteiro: 1733836
Parte de posição do 9o. inteiro: 1733840
Parte de posição do 10o. inteiro: 1733844
Process returned 0 (0x0)   execution time : 0.139 s
Press any key to continue.

```

Analisando os dados de saída, do programa acima, chegamos a importantes conclusões. Veja abaixo.

Linha	Efeito produzido
5	<p>Foi alocado espaço na memória para 10 números inteiros. A variável p passa a apontar para o endereço do primeiro número inteiro.</p> <p>Parte de Posição</p> <p>Parte de Valor</p>
9	<p>Nessa linha é apresentado o endereço da 1a região alocada acrescida da variável i. Aqui estamos usando aritmética de ponteiros para avançarmos pelas regiões de memória. Na 1a. repetição mostramos o endereço do 1o. inteiro, na 2a. repetição o endereço do 2o. inteiro, e assim sucessivamente até o último.</p> <p>Repare que no computador utilizado para testar esse programa, cada inteiro gasta 4 bytes, daí a razão para os endereços serem mostrados de 4 em 4. Concluímos que incrementar um ponteiro de 1 unidade, causa um avanço na memória de acordo com a quantidade de bytes que o tipo de dado ocupa.</p>

Faça simulações com esse programa, alterando o tipo de dado para double, float e char. Veja a saída produzida e tire suas conclusões.

Analise o código abaixo para concluirmos essa parte sobre aritmética de ponteiros.

aritméticaPonteiros2.c

```
1  #define MAX 5
2  main()
3  {
4      setlocale(LC_ALL, "Portuguese");
5
6      int *p = (int) malloc(sizeof(int) * 5);
7
8      int i;
9      for (i=0; i<MAX; i++){
10         printf("Informe um número: ");
11         scanf("%d", p + i);
12     }
13
14     printf("\n\n*** Mostra os valores - Do primeiro ao último ***");
15     for (i=0; i<MAX; i++){
16         printf("\nParte de valor do %do. inteiro: %d", i+1, *(p+i));
17     }
18
19     printf("\n\n*** Mostra os valores - Do último ao primeiro ***");
20     for (i=MAX-1; i>=0; i--){
21         printf("\nParte de valor do %do. inteiro: %d", i+1, *(p+i));
22     }
23 }
```

Veja que esse programa realiza a seguinte sequência de ações:

Linha	Efeito produzido
6	Foi alocado espaço na memória para 5 números inteiros.
9 a 12	A parte de valor de p recebe 5 números inteiros. Veja que não foi preciso utilizar o operador '&' no scanf pois a parte de posição de um ponteiro armazena endereço.
14 a 17	Percorre a região de memória alocada, do menor para o maior endereço, usando aritmética de ponteiros.
19 a 22	Percorre a região de memória alocada, do maior para o menor endereço, usando aritmética de ponteiros.

2.6. Aritmética de ponteiros x indexação

Certamente você recorda a forma como referenciamos um vetor na linguagem C. Usamos o processo de indexação por um número inteiro. Esse número inteiro, chamado de índice, indica a posição do vetor, conseqüentemente o elemento que desejamos referenciar.

Agora venho colocar uma questão. Você conhece a razão para o fato de usarmos o índice 0 para referenciar o 1º elemento do vetor? Não seria mais significativo se usássemos o índice 1 para indicar a 1ª posição?

A explicação para isso reside no fato de que o compilador usa aritmética de ponteiros para referenciar os elementos de um vetor. Temos que lembrar de que um vetor, na realidade, é um ponteiro que aponta para a 1ª posição da região alocada, conforme demonstramos anteriormente. E que usamos essa 1ª posição para acessar todas as demais.

O compilador, internamente, substitui a indexação pela operação aritmética equivalente.

```
int vetor[n] ⇔ int *vetor = (int) malloc(sizeof(int) * n);
vetor[0] ⇔ *(vetor + 0)
vetor[1] ⇔ *(vetor + 1)
vetor[2] ⇔ *(vetor + 2)
...
vetor[n] ⇔ *(vetor + n-1)
```

Analise o programa abaixo e veja que referenciamos o vetor usando indexação.

aritméticaPonteiros3.c

```
1  #define MAX 5
2  main()
3  {
4      setlocale(LC_ALL, "Portuguese");
5
6      int p[MAX];
7
8      int i;
9      for (i=0; i<MAX; i++){
10         printf("Informe um número: ");
11         scanf("%d", &p[i]);
12     }
13
14     printf("\n\n*** Mostra os valores - Do primeiro ao último ***");
15     for (i=0; i<MAX; i++){
16         printf("\nParte de valor do %do. inteiro: %d", i+1, p[i]);
17     }
18
19     printf("\n\n*** Mostra os valores - Do último ao primeiro ***");
20     for (i=MAX-1; i>=0; i--){
21         printf("\nParte de valor do %do. inteiro: %d", i+1, p[i]);
22     }
23 }
```

Veja abaixo uma nova versão do programa acima onde trocamos a indexação pela aritmética de ponteiros para referenciar o vetor. Ambos os programas apresentam exatamente a mesma funcionalidade, a diferença está apenas na forma de manipular o vetor: indexação ou aritmética de ponteiros.

Atente-se para as partes destacadas.

aritméticaPonteiros4.c

```
1  #define MAX 5
2  main()
```

```

3  {
4      setlocale(LC_ALL, "Portuguese");
5
6      int *p = (int) malloc(sizeof(int) * MAX);
7
8      int i;
9      for (i=0; i<MAX; i++){
10         printf("Informe um número: ");
11         scanf("%d", p+i);
12     }
13
14     printf("\n\n*** Mostra os valores - Do primeiro ao último ***");
15     for (i=0; i<MAX; i++){
16         printf("\nParte de valor do %do. inteiro: %d", i+1, *(p + i));
17     }
18
19     printf("\n\n*** Mostra os valores - Do último ao primeiro ***");
20     for (i=MAX-1; i>=0; i--){
21         printf("\nParte de valor do %do. inteiro: %d", i+1, *(p + i));
22     }
23 }

```

2.7. Passagem de parâmetros: por valor x por referência

Sabemos que existe a possibilidade de enviarmos dados para uma determinada função através de uma lista de parâmetros declarados em seu cabeçalho. Entenderemos agora que a forma que passávamos parâmetros para uma função, até o momento, era **por valor** e que após o estudo de ponteiros veremos que existe a possibilidade de passarmos parâmetros **por referência**.

Apresentamos abaixo a listagem de um programa que realiza a passagem de parâmetros conforme sempre fazíamos, ou seja, passando parâmetros por valor. Sugiro que execute esse código e perceba um fato que não demos importância quando aprendemos modularização.

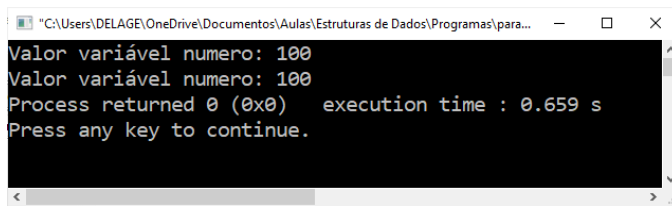
parametros1.c

```

1
1 void modificaNumero(int numero);
2
3 main()
4 {
5     setlocale(LC_ALL, "Portuguese");
6
7     int numero = 100;
8
9     printf("Valor variável numero: %d", numero);
10    modificaNumero(numero);
11    printf("\nValor variável numero: %d", numero);
12 }
13
14 void modificaNumero(int numero)
15 {
16     numero = numero * 2;
17 }

```

Ao executar percebe-se a seguinte saída:



```
"C:\Users\DELAG\OneDrive\Documentos\Aulas\Estruturas de Dados\Programas\para...
Valor variável numero: 100
Valor variável numero: 100
Process returned 0 (0x0) execution time : 0.659 s
Press any key to continue.
```

Veja que o valor da variável `numero` foi apresentado antes (linha 9) e depois (linha 11) da invocação da função `modificaNumero` (linha 10). Perceba que a variável `numero` foi alterada pela função `modificaNumero`, mas o seu valor original se manteve sem alteração.

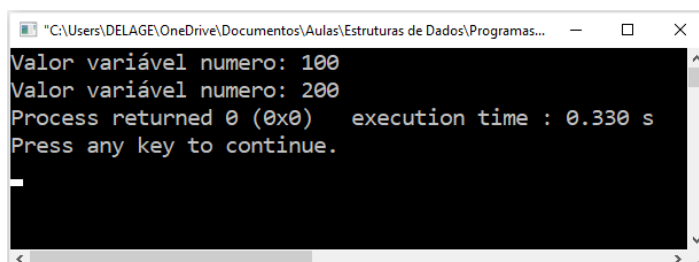
Isso é prova de que o parâmetro `numero` recebido pela função `modificaNumero` (linha 14) é uma cópia da variável original, também de nome `numero`, declarada na função `main` (linha 7). Com se trata de uma cópia, a alteração nesta cópia (linha 16) não reflete no valor original. Isso é o que chamamos de **passagem de parâmetros por valor**.

Agora veremos uma nova forma de passar parâmetros. Sugiro realizar as seguintes alterações no código anterior para que possamos compreender melhor as explicações.

`parametros2.c`

```
1 void modificaNumero(int *numero);
2
3 main()
4 {
5     setlocale(LC_ALL, "Portuguese");
6
7     int *numero = malloc(sizeof(int));
8     *numero = 100;
9
10    printf("Valor variável numero: %d", *numero);
11    modificaNumero(numero);
12    printf("\nValor variável numero: %d", *numero);
13 }
14
15 void modificaNumero(int *numero)
16 {
17     *numero = *numero * 2;
18 }
```

A única alteração no programa foi a troca do tipo da variável `numero` para ponteiro. Isso obrigou alterações em várias partes do código. Mas o que importa é a consequência desta alteração. Execute o código e analise a saída produzida.



```
"C:\Users\DELAG\OneDrive\Documentos\Aulas\Estruturas de Dados\Programas...
Valor variável numero: 100
Valor variável numero: 200
Process returned 0 (0x0) execution time : 0.330 s
Press any key to continue.
```

Perceba que agora, a variável `numero` não manteve o valor original, após a execução da função `modificaNumero`. Como a função recebe uma referência ao valor original, ou seja, um ponteiro, as alterações realizadas nesta referência refletirão no valor original. Isso é o que chamamos de **passagem de parâmetros por referência**, e que só é possível devido a característica fundamental dos ponteiros.

2.8. Passando vetor como parâmetro

Já vimos que um vetor nada mais é que um ponteiro para o primeiro endereço de uma região alocada na memória. Portanto, passar um vetor como parâmetro para uma função, corresponde exatamente ao que vimos anteriormente sobre passagem de parâmetros por referência.

Analisar o código abaixo e a respectiva saída produzida através de sua execução. Atente-se para o fato de que as alterações realizadas no parâmetro `salarios`, recebido pelas funções, refletiu em seu valor original.

`parametros3.c`

```
1  #define MAX 5
2
3  void leSalarios(float *salarios);
4  void mostraSalarios(float *salarios);
5  void reajustaSalarios(float *salarios, float percentual);
6
7  main()
8  {
9      setlocale(LC_ALL, "Portuguese");
10
11      float *salarios = malloc(sizeof(float) * MAX);
12
13      leSalarios(salarios);
14      mostraSalarios(salarios);
15
16      reajustaSalarios(salarios, 0.10);
17      mostraSalarios(salarios);
18  }
19
20 void leSalarios(float *salarios)
21 {
22     int i;
23     for(i=0; i<MAX; i++)
24     {
25
26         printf("\nInforme o %do. salário: ", i+1);
27         scanf("%f", (salarios+i));
28     }
29 }
30
31 void mostraSalarios(float *salarios)
32 {
33     int i;
34     for(i=0; i<MAX; i++)
35     {
36         printf("\n%do. salário: %f", i+1, *(salarios+i));
37     }
```



```

38 }
39
40 void reajustaSalarios(float *salarios, float percentual)
41 {
42     int i;
43     for(i=0; i<MAX; i++)
44     {
45         *(salarios+i) += *(salarios+i) * percentual;
46     }
47 }

```

```

"C:\Users\DELAGE\OneDrive\Documentos\Aulas\Estruturas de Dados\Programas\para...
Informe o 1o. salário: 100
Informe o 2o. salário: 200
Informe o 3o. salário: 300
Informe o 4o. salário: 400
Informe o 5o. salário: 500

1o. salário: 100,000000
2o. salário: 200,000000
3o. salário: 300,000000
4o. salário: 400,000000
5o. salário: 500,000000
1o. salário: 110,000000
2o. salário: 220,000000
3o. salário: 330,000000
4o. salário: 440,000000
5o. salário: 550,000000
Process returned 0 (0x0)   execution time : 13.221 s
Press any key to continue.

```

3. Lista linear

Lista linear é uma estrutura de dados que representa um conjunto de dados dispostos em **seqüência**. Os elementos da lista, também conhecidos como nós, podem conter, cada um deles, um dado primitivo ou um dado composto.

Sua característica fundamental está no fato de que seus elementos estão ligados um ao outro em seqüência. Inicialmente estudaremos as listas lineares por **contiguidade**, onde seus elementos são dispostos na memória, em posições contíguas. Em seguida, analisaremos uma outra forma de representar listas, que são as listas lineares por **encadeamento**, onde seus elementos apresentam uma ligação, entre si, independente da posição que ocupam na memória.

Essas duas formas de representar listas lineares incorporam soluções específicas para determinados tipos de problemas. É isso que estudaremos a seguir. Em breve seremos capazes de não só implementá-las como discernir a situação mais adequada para utilizá-las.

3.2. Lista linear por contiguidade

Esta representação explora a sequencialidade da memória do computador, de forma que os nós são dispostos em posições contíguas.

índice	0	1	2	3	4
valor	1o. nó	2o. nó	3o. nó	4o. nó	5o. nó
endereço	2293496	2293497	2293498	2293499	2293450

Percebe-se que a representação de uma lista por contiguidade já está disponível nas linguagens de programação, através dos vetores. Portanto, é necessário fazer, a priori, uma estimativa do comprimento máximo que a lista terá, para que o vetor que a contiver seja declarado com este tamanho. Esta limitação deve-se ao fato de que o vetor não pode ter o seu tamanho alterado dinamicamente para permitir a representação de uma lista de comprimento maior, ou menor, do que o inicialmente previsto.

Quanto à sua aplicabilidade, é fácil deduzir que a representação de listas por contiguidade é mais adequada para armazenar elementos, sobre os quais **não** são necessárias inclusões ou remoções em posições intermediárias. Operações de inserção e remoção de nós exigem um grande esforço computacional se realizadas em posições intermediárias. Este fato pode determinar o baixo desempenho do programa, que utiliza esse tipo de representação, caso estas operações sejam frequentes.

Entretanto, em operações de busca, esse tipo de representação alcança grande eficiência, já que o acesso a determinado elemento da lista ocorre de forma direta, sem a necessidade de passar por todos os elementos anteriores ao desejado. O acesso à informação é **direto**.

Veja a seguir dois exemplos de listas lineares por contiguidade.

listaLinear1.c

```
1  #define MAX 5
2  main()
3  {
4      setlocale(LC_ALL, "Portuguese");
5
6      float *listaNotas = (float*) malloc(sizeof(float) * MAX);
7
8      float media=0;
9      int i;
10     for (i = 0; i < MAX; i++)
11     {
12         printf("\nInforme a %da. nota: ", i+1);
13         scanf("%f", listaNotas + i);
14         media += *(listaNotas + i);
15     }
16
17     printf("\nMedia das notas: %f", (media/MAX));
18 }
```

Repare que o programa acima não apresenta novidade, trata-se de uma simples manipulação de vetor. Entretanto, trata-se de uma lista linear por contiguidade, onde seus elementos, as notas dos alunos, são armazenadas em posições contíguas da memória, caracterizando desta forma, o tipo de representação que estamos estudando no momento.

O segundo exemplo, a seguir, manipula uma lista linear por contiguidade de elementos compostos (struct). Trata-se de uma lista onde cada nó armazena a matrícula, o nome e a nota dos alunos de uma turma. O objetivo do programa é descobrir quais alunos alcançaram nota superior ou igual à média da turma.

listaLinear2.c

```
1
1  #define MAX 3
2
3  struct EstruturaAluno
4  {
5      int matricula;
6      char *nome;
7      float nota;
8  };
9
10 typedef struct EstruturaAluno Aluno;
11
12 main()
13 {
14     setlocale(LC_ALL, "Portuguese");
15
16     Aluno *listaAlunos = (Aluno*) malloc(sizeof(Aluno) * MAX);
17
18     int i;
19     float media = 0;
20
21     for (i = 0; i < MAX; i++)
```

```

22     {
23
24         printf("\nInforme a matricula do aluno: ");
25         scanf("%d", &((listaAlunos+i)->matricula));
26
27         (listaAlunos+i)->nome = (char *) malloc(sizeof(char) * 30);
28         printf("\nInforme o nome do aluno: ");
29         scanf("%s", (listaAlunos+i)->nome);
30
31         printf("\nInforme a nota do aluno: ");
32         scanf("%f", &((listaAlunos+i)->nota));
33
34         media += (listaAlunos+i)->nota;
35     }
36
37     media = media / MAX;
38
39     printf("\n\nAlunos com nota superior ou igual a média da turma:");
40     for (i = 0; i < MAX; i++)
41     {
42         if ((listaAlunos+i)->nota >= media){
43             printf("\n%d - %s", (listaAlunos+i)->matricula, (listaAlunos+i)->nome);
44         }
45     }
46 }

```

Vamos aos comentários:

Linha	Efeito produzido
3 a 8	É definido uma struct com 3 campos. Note que um dos campos é um vetor de char (linha 6) que foi declarado mas ainda não foi alocado.
10	Definimos um tipo de dado de nome Aluno com base na struct definida anteriormente. Mais adiante, no momento da criação da lista linear, definiremos que os elementos da lista serão desse tipo (Aluno).
25	Já sabemos que para referenciar um determinado campo de uma struct utilizamos o operador '.' para separar o nome da struct do nome do campo a ser referido. Continue utilizando essa regra de sintaxe da linguagem C, porém troque pelo operador '->' quando se tratar de ponteiros para struct.

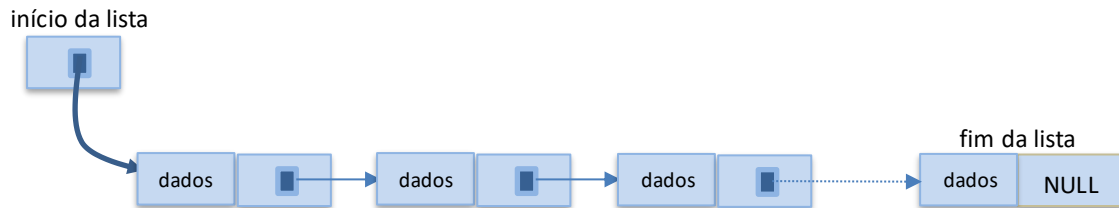
3.3. Lista linear por encadeamento

Em uma lista linear **por encadeamento**, a ligação entre os nós se dá independentemente da posição que ocupam na memória. Queremos dizer que os nós NÃO estão em posições contíguas da memória, na prática esses nós ocuparão posições aleatórias na memória de acordo com a disponibilidade de alocação. O encadeamento entre os nós é implementado através de ponteiros.

Percebemos algumas vantagens no uso desse tipo de encadeamento entre os nós. Uma delas é permitir o crescimento dinâmico do comprimento de uma lista, já que poderemos alocar memória dinamicamente.

Outra vantagem é diminuir o esforço computacional das operações de inserção ou remoção de nós em posições intermediárias, haja visto que não haverá necessidade de envolver o restante da lista. Evitando-se o deslocamento de dados, quando se trata de conjuntos com muitos elementos, consegue-se maior rapidez no processamento.

Para implementar a lista linear por encadeamento precisamos acrescentar, às informações que cada nó armazenará, um ponteiro que aponte para o próximo nó da lista. Cada elemento é associado ao outro através deste ponteiro (elo de ligação), permitindo desta forma utilizar posições **não contíguas** da memória. Veja a imagem abaixo.



Entenda que a ligação entre os dados representados acima NÃO é física, como pode parecer à primeira vista. Lembre-se de que não importa o endereço de cada nó na memória, o essencial é saber que cada nó apontará para o próximo.

Vejamos agora como implementar uma lista linear por encadeamento. Inicialmente vamos codificar o módulo principal, responsável pela definição e criação da lista, inicialmente vazia. Cada nó será composto por uma estrutura que possui duas partes, uma **referente aos dados** e outra para indicar o **próximo nó da lista**.

listaLinear3.c

```
1 #include <stdlib.h>
2
3 struct EstruturaAluno
4 {
5     int matricula;
6     char *nome;
7     float nota;
8
9     struct EstruturaAluno *prox;
10 };
11
12 typedef struct EstruturaAluno Aluno;
13
14 main()
15 {
16     Aluno *listaAluno = NULL;
17     ....
18 }
```

Linha	Efeito produzido
3 a 10	Foi definido uma estrutura de nome EstruturaAluno. Na linha 5,6 e 7 foi definido a parte de dados desta estrutura e na linha 9 definimos o ponteiro que apontará para o próximo nó da lista, que por sua vez é do mesmo tipo.
12	Foi definido um tipo de dado de nome Aluno. Cada nó da lista será desse tipo, ou seja, cada nó armazenará a matrícula do aluno, seu nome e sua nota, além de um ponteiro para o próximo nó.
16	<p>Aqui criamos a lista encadeada de nome listaAluno. Trata-se de um ponteiro, que no futuro, apontará para o primeiro nó da lista, que por sua vez apontará para o segundo e assim sucessivamente até o ultimo nó da lista, que apontará para NULL, indicando o fim da lista.</p> <p>Apesar de já termos criado a lista, ela ainda está vazia, sem elementos. Para sinalizar essa situação, ela foi inicializada com NULL para indicar que nossa lista encadeada está vazia, ou seja, o primeiro nó é NULL (não existe).</p>

Daremos continuidade implementando os algoritmos relativos às operações básicas com a lista criada anteriormente: inserção, remoção e percurso. Segue abaixo os respectivos protótipos das funções.

listaLinear3.c

```

1  #include <stdlib.h>
2
3  struct EstruturaAluno
4  {
5      int matricula;
6      char *nome;
7      float nota;
8
9      struct EstruturaAluno *prox;
10 };
11
12 typedef struct EstruturaAluno Aluno;
13
14 Aluno *insereInicio(Aluno *atual, Aluno no);
15 Aluno *insereFinal(Aluno *atual, Aluno no);
16 void imprime(Aluno *atual);
17 Aluno *libera(Aluno *atual);
18 Aluno *busca(Aluno *atual, int matricula);
19 Aluno *remove(Aluno *atual, int matricula);
20
21 main()
22 {
23     Aluno *listaAluno = NULL;
24     ....
25 }
```

Para realizar operações com uma lista encadeada, precisamos saber onde ela começa, termina e qual nó estamos manipulando no momento. A perda desse controle pode tornar

nossa lista inacessível. Sabemos que todas as operações com lista linear encadeada começam pelo primeiro nó, é através dele que acessaremos os demais, portanto imagine perder o endereço desse nó?

Outra situação crítica ocorre durante a inserção de um novo nó. Como implementaremos a inserção no início da lista, você pode imaginar os problemas decorridos se essa situação não for atualizada corretamente?

Diversas outras operações exigem que tenhamos o controle correto da lista que está sendo manipulada. Para tanto utilizaremos variáveis que nos auxiliarão nesse processo.

- **início:** ponteiro que aponta para o primeiro nó da lista. Podemos considerar o controle mais importante. Sua perda ocasiona a perda da lista. Qualquer operação com a lista se inicia pelo primeiro nó.
- **atual:** ponteiro que aponta para o nó atual, ou seja, o nó que está sendo manipulado (consultado, alterado, excluído, etc).
- **ant:** ponteiro que aponta o nó anterior ao atual. Controle utilizado em operações de remoção. Esse ponteiro é importante pois ele servirá para restabelecermos o encadeamento correto após a remoção de um nó.
- **novo:** ponteiro que aponta para um novo nó que será inserido na lista. Controle utilizado em operações de inserção.

Antes de analisar a função de inserção veja um exemplo de como ela seria invocada pelo módulo principal.

listaLinear3.c

```
1 //código omitido
2 main()
3 {
4     Aluno *listaAluno = NULL;    //Cria lista vazia
5
6     Aluno no;
7     no.matricula = 111;
8     no.nome = "Flavio";
9     no.nota = 10.0;
10    listaAluno = insereInicio(listaAluno, no);
11 }
```

Agora analisaremos o algoritmo de inserção em uma lista encadeada. Será implementado de forma que todo novo nó seja inserido no início da lista. Após o entendimento dessa operação, sugiro que proponha uma implementação que insira elementos ao final da lista. Pode-se pensar em diversas propostas, como inserir no meio da lista, antes ou depois de um elemento, seguindo algum critério, dentre outras. As possibilidades são diversas.

Inserção de nó no início da lista

```
1 Aluno *insereInicio(Aluno *atual, Aluno no)
2 {
3     Aluno *novo = (Aluno*) malloc(sizeof(Aluno));
4
5     novo->matricula = no.matricula;
6     novo->nome = no.nome;
7     novo->nota = no.nota;
8 }
```

```

9      if (atual == NULL) //Lista está vazia?
10     {
11         novo->prox = NULL;
12     }
13     else
14     {
15         novo->prox = atual;
16     }
17     return novo;
18 }

```

Linha	Efeito produzido
1	<p>Sobre o primeiro parâmetro: todas as funções que implementaremos receberá um ponteiro como este, pois sabemos que todas as operações, em uma lista linear por encadeamento, começam pelo primeiro nó. Portanto ao invocarmos essa função deveremos passar como primeiro parâmetro um ponteiro para o primeiro nó.</p> <p>Sobre o segundo parâmetro: representa os dados, será o nó a ser inserido na lista.</p> <p>Sobre o retorno: mais adiante veremos uma explicação para o fato desta função retornar um ponteiro.</p>
3	Alocamos o novo nó a ser inserido.
9	Verificamos se o ponteiro atual está NULL. Se for verdadeiro indica que nossa lista está vazia e este novo nó será o primeiro.
11	Registra o fato de que sendo novo o primeiro, o seu próximo não existe.
15	Essa linha só será executada se a lista não estiver vazia. Neste caso, esta linha registra o fato de que o próximo do novo será o atual, ficando desta forma o novo na primeira posição e o atual na segunda.
17	Esta linha é muito importante. Ela atualiza a lista, criada no módulo principal (main), retornando o novo nó, que será o ponto de partida para qualquer manipulação com a lista. Aqui encontramos a razão para o fato desta função ter um retorno do tipo ponteiro.

Em seguida veremos o algoritmo para percorrer os nós da lista. Fica sob sua responsabilidade implementar a invocação desta função pelo módulo principal do programa.

Trata-se de uma operação muito comum, que deve servir de base para muitas outras operações. A proposta desse algoritmo é imprimir todos elementos da lista.

Impressão dos nós da lista

```

1 void imprime(Aluno *atual)
2 {
3     if (atual == NULL)
4     {

```



```

5     printf("\nLista Vazia!");
6 }
7
8 while(atual != NULL)
9 {
10     printf("\nMatrícula: %d", atual->matricula);
11     printf("\nNome.....: %s", atual->nome);
12     printf("\nNota.....: %f", atual->nota);
13     printf("\n");
14     atual = atual->prox;
15 }
16 }

```

Linha	Efeito produzido
1	Veja que o parâmetro atual, da mesma forma que na função de inserção e outras que veremos em seguida, recebe um ponteiro para o 1o elemento da lista. O ato de percorrer uma lista não a modifica, portanto não há necessidade de retorno para atualizar a lista definida no modulo principal.
8	Quando atual for NULL indica que chegamos ao final da lista, tornando a condição falsa e conseqüentemente interrompendo o percurso.
14	Causa um avanço na lista, deixa de apontar para o nó atual e passa a apontar para o próximo.

Vimos que uma lista linear por encadeamento pode receber novos elementos durante a execução do programa, basta que novos nós sejam alocados dinamicamente. Agora veremos o papel inverso, analisaremos o algoritmo de remoção de elementos de uma lista, também de forma dinâmica, sendo que neste caso promoveremos o rearranjo dos elementos e a liberação da memória não mais desejada.

Remoção de um nó da lista

```

1 Aluno* remove(Aluno *atual, int matricula)
2 {
3     Aluno *ant = NULL; /* ponteiro para elemento anterior */
4
5     Aluno *inicio= atual; /*guarda o inicio da lista.*/
6     while((atual != NULL) && (atual->matricula != matricula))
7     {
8         ant = atual;
9         atual = atual->prox;
10    }
11
12    if(atual == NULL) return inicio; /* não achou: retorna lista original */
13
14    Aluno *exclui = atual;
15    if(ant == NULL)
16    {
17        /* remove elemento do inicio */
18        atual = atual->prox;
19        inicio = atual;
20    }
21    else
22    {
23        /* remove elemento intermediario ou último */

```

```
24         ant->prox = atual->prox;
25     }
26     free(exclui);
27     return inicio;
28 }
```

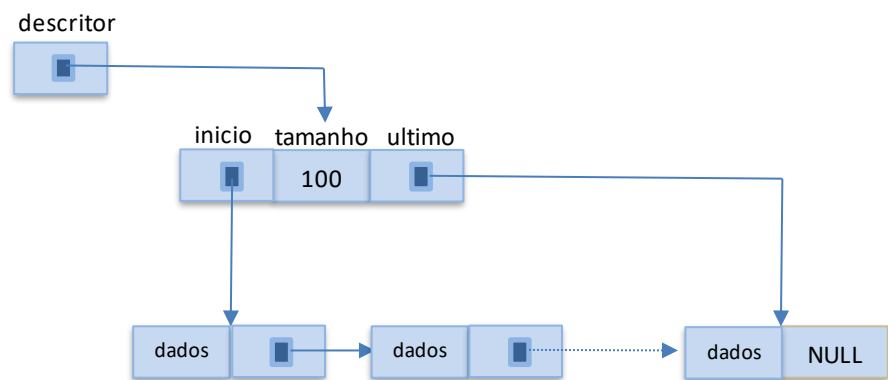
Linha	Efeito produzido
1	Primeiro parâmetro: é um ponteiro para o primeiro nó da lista. Já vimos que todas as operações com uma lista se iniciam pelo primeiro nó. Segundo parâmetro: corresponde à chave a ser removida, que no caso será a matrícula do aluno. Finalmente temos como retorno um ponteiro indicando um novo início da lista para o caso de se remover o primeiro nó. Para o caso de remoção de nós em posições intermediárias, esse retorno não fará diferença pois o início da lista permanecerá inalterado.
3	A variável <code>ant</code> é um ponteiro que indicará o nó anterior ao <code>atual</code> (o que será removido). Como <code>atual</code> começa apontando para o primeiro, portanto seu anterior não existe, por isso foi inicializada com <code>NULL</code> .
5	Guarda o início da lista para o caso de haver um novo início se o nó removido for o primeiro.
6	Enquanto (não é fim da lista) e (não achou matrícula a remover) faça
8 e 9	A medida que percorremos a lista, <code>atual</code> e <code>ant</code> irão apontando em sincronia em direção ao nó a ser removido. Quando <code>atual</code> chegar a apontar para o nó desejado, <code>ant</code> estará apontando para o anterior do <code>atual</code> . Assim será possível fazermos o próximo nó do anterior apontar para o próximo do nó removido , desta forma <code>atual</code> estará livre para remoção.
14	Prepara para excluir. Iremos liberar esse nó que <code>exclui</code> aponta, mas antes temos que fazer alguns ajustes.
15	Se <code>ant</code> for <code>NULL</code> , indica que removeremos o primeiro nó.
18 e 19	Configuramos um novo início da lista, que será o próximo do <code>atual</code> .
24	Para o caso de remoção em posição intermediária, configuramos o próximo nó do anterior para que aponte para o próximo do nó removido .
26	A exclusão é efetivada.
27	Essa linha é responsável por atualizar o ponto inicial da lista, criada no módulo principal, retornando o novo início (se removemos no início) ou antigo início (para remoção nas demais posições).

3.4. Variações de listas encadeadas

No tópico anterior vimos a lista linear por encadeamento simples, onde a ligação entre os nós se dá apenas em uma direção. Já em uma lista duplamente encadeada cada nó possui duas referências ao invés de apenas uma, onde uma referência aponta para o nó predecessor e outra para o sucessor. Com esta nova abordagem, o percurso pelos nós da lista pode ser feito em dois sentidos.

Outra variação seria a lista encadeada circular simples ou dupla, onde uma extremidade livre da lista aponta para outra.

Finalmente apresentaremos a lista encadeada com descritor. Nó descritor é aquele que guarda informações além do endereço do primeiro e/ou ultimo nó da lista encadeada. Com a presença de um nó descritor o acesso aos elementos da lista continuará sendo como vimos até agora, ou seja, sempre efetuado através do seu descritor que indicará o início e/ou o fim da lista. Entretanto, o nó descritor possibilita o armazenamento de outras informações da lista, como quantidade de nós, data da criação, autor, dentre outras a critério do programador.



(Lista linear por encadeamento simples com descritor)

3.5. Atividades

Agora é sua vez de praticar! Todos os algoritmos analisados até o momento são considerados básicos para manipulação de listas encadeadas. Servirão de base para fazer os exercícios propostos, basta realizar poucas adaptações. Se ainda se sente inseguro, volte e estude novamente!

- 1) Analise os programas listaLinear1, listaLinear2 e listaLinear3. Procure associar as explicações e representações feitas em sala de aula com o código. Se necessário execute e faça simulações.
- 2) Proponha uma nova funcionalidade para o programa listalinear3: uma função que seja capaz de inserir nós ao final da lista.
- 3) Proponha uma nova funcionalidade para o programa listalinear3: uma função que faça busca, ou seja, que verifique a existência de um determinado nó na lista. A função deve ser capaz de retornar todos os dados referentes a uma determinada matricula de aluno.
- 4) Proponha uma nova funcionalidade para o programa listalinear3: uma função que seja capaz de calcular a media aritmética das notas dos alunos.

4. Listas lineares com disciplina de acesso

Algumas situações exigem um **critério** que restringe a forma como ocorrerá a **inserção** e a **recuperação** dos elementos que compõem um conjunto de dados. Veremos a seguir dois critérios.

4.2. LIFO - Last in first out

Listas lineares com essa disciplina de acesso são denominadas **PILHAS** pois pode-se fazer uma analogia com uma pilha de livros, onde um novo livro sempre será colocado no topo da pilha e só podemos retirar o livro do topo da pilha, caso contrário ela desabaria.

Ao implementarmos esse tipo de lista temos que garantir que o último elemento inserido será o primeiro a ser recuperado.

A utilização do critério LIFO torna a Pilha uma ferramenta ideal para processamento de estruturas aninhadas de profundidade imprevisível, situação em que é necessário garantir que subestruturas mais internas sejam processadas antes das estruturas que as contenham. Vejamos a seguir alguns exemplos de estruturas aninhadas:

- a) Quando é necessário caminhar em um conjunto de dados e guardar uma lista de coisas a fazer posteriormente (Editor de Textos – desfazer digitação).
- b) Um bom exemplo de aplicação de pilha é o funcionamento das calculadoras da HP (Hewlett-Packard). Elas trabalham com expressões pós-fixadas, então para avaliarmos uma expressão como $(1-2)*(4+5)$ podemos digitar $1\ 2\ -\ 4\ 5\ +\ *$.

O funcionamento dessas calculadoras é muito simples. Cada operando é empilhado numa pilha de valores. Quando se encontra um operador, desempilha-se o número apropriado de operandos (dois para operadores binários e um para operadores unários), realiza-se a operação devida e empilha-se o resultado. Deste modo, na expressão acima, são empilhados os valores 1 e 2. Quando aparece o operador -, 1 e 2 são desempilhados e o resultado da operação, no caso -1 ($1 - 2$), é colocado no topo da pilha. A seguir, 4 e 5 são empilhados. O operador seguinte, +, desempilha o 4 e o 5 e empilha o resultado da soma, 9. Nesta hora, estão na pilha os dois resultados parciais, -1 na base e 9 no topo. O operador *, então, desempilha os dois e coloca -9 ($-1 * 9$) no topo da pilha.

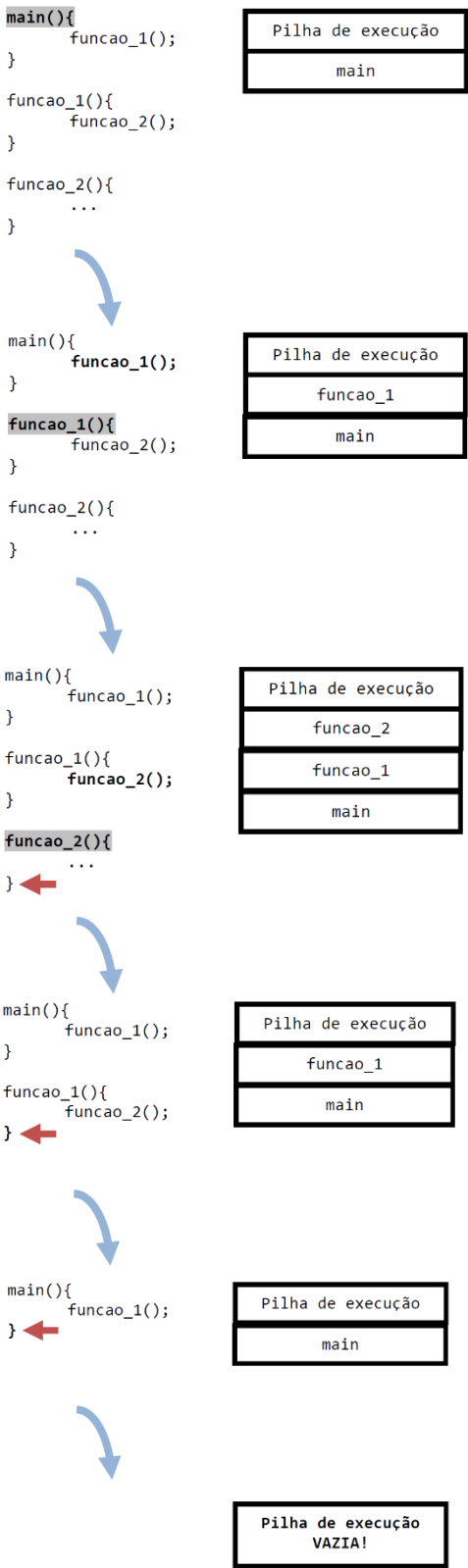
- c) Controle de sequências de chamadas de subprogramas e algoritmos recursivos. Veremos detalhadamente a seguir.

PILHA DE EXECUÇÃO DE UM PROGRAMA

A cada invocação de uma função é criado um novo "espaço de trabalho" exclusivo para ela, contendo todos os seus parâmetros e todas suas variáveis locais. Chamamos essa estrutura de pilha de execução devido ao fato de que cada espaço de trabalho é alocado sobre o espaço de trabalho da função que originou a invocação, formando desta forma uma pilha.

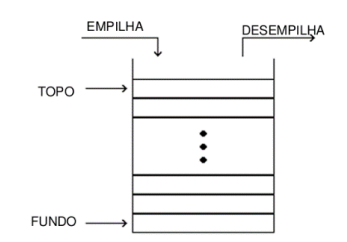
Quando a execução da função termina, o seu espaço de trabalho é retirado da pilha e descartado. O espaço de trabalho que estiver agora no topo da pilha é reativado e a execução é retomada do ponto em que havia sido suspensa.

A seguir veremos como a pilha de execução é formada e destruída durante a execução de diversas funções de um programa imaginário. A parte em **negrito** indica a função que está sendo executada no momento da análise.



OPERAÇÕES COM PILHA

Ao implementarmos as operações com uma pilha temos que garantir que a inserção (empilhamento) e recuperação (desempilhamento) de informações ocorra apenas em uma das extremidades da pilha.



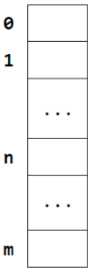
A figura abaixo mostra como funciona o empilhamento e o desempilhamento em uma pilha.

Ação	Conteúdo da Pilha
insere(a)	a
insere(b)	ba
insere(c)	cba
recupera	ba
insere(d)	dba
recupera	ba
recupera	a
recupera	Pilha vazia

REPRESENTAÇÕES DE UMA PILHA

A pilha, sendo uma lista linear, pode ser representada por contiguidade ou por encadeamento. A escolha de uma destas formas segue a mesma lógica estudada anteriormente.

Ao implementar uma pilha por contiguidade deve-se ter o controle do topo da pilha. O esquema abaixo facilita o entendimento, onde **n** é o topo e **m** o tamanho máximo permitido para a pilha.



A implementação de uma pilha por encadeamento é idêntica a de uma lista linear por encadeamento. Entretanto deve-se tomar cuidado em obedecer o critério de restrição que uma

pilha exige em operações de inserção e recuperação, independente de escolhermos a representação por contiguidade ou encadeamento.

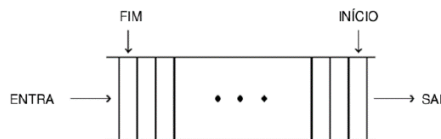
4.3. FIFO – First in first out

As listas lineares que obedecem a esse critério são denominadas de **FILAS** pois pode-se fazer uma analogia com o funcionamento de uma fila de pessoas em um guichê, onde as pessoas são atendidas na ordem de chegada.

Sistemas Operacionais, por exemplo, utilizam filas para regular a ordem na qual tarefas devem receber processamento e recursos devem ser alocados a processos.

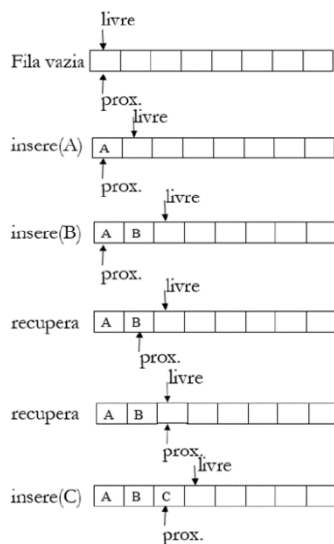
OPERAÇÕES COM UMA FILA

Ao implementarmos uma fila temos que garantir que o primeiro elemento inserido seja o primeiro a ser recuperado. Veja na imagem abaixo que a inserção e a recuperação são realizadas em extremidades opostas.



REPRESENTAÇÕES DE UMA FILA

A fila também pode ser representada por contiguidade ou por encadeamento. O importante é obedecer o critério de restrição que uma fila exige em operações de inserção e recuperação.



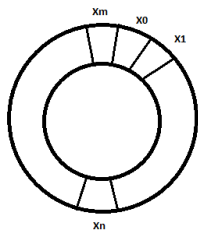
Veja ao lado uma sugestão de como controlar as operações em uma fila por contiguidade. A variável **livre** indica a posição que a inserção deve ocorrer e **prox** a posição a ser recuperada.

Quando a posição a inserir (**livre**) for a mesma que a recuperar (**prox**), significa que fila está vazia.

Quando **livre** for igual ao tamanho do vetor, indica fila cheia.

Perceba que as operações de inserção ocupam um espaço na memória que não é reaproveitado. Com algumas inserções e recuperações de itens, a fila vai de encontro ao limite do espaço alocado para ela. Como solucionar problema ?

A solução ideal seria reutilizar as posições que já foram recuperadas, ou seja, as anteriores à posição indicada por **prox**. Para isso usa-se a **representação circular de fila**. Veja a seguir.



Para possibilitar que posições recuperadas possam ser reutilizadas, devemos fazer com que as variáveis **livre** e **prox**, ao alcançarem seus valores de limite (**m**), voltem a ter valores iniciais (**0**). Isto pode ser implementado através da aritmética modular conforme veremos a seguir.

Precisaríamos também de uma variável (**n**) para controlar o tamanho da fila. A cada inserção é incrementada e a cada recuperação decrementada.

Usaremos a aritmética modular para fazer com que as variáveis **livre** e **prox**, ao alcançarem seus valores limite (**m**), voltem a ter valores iniciais (**0**). Para exemplificar consideraremos **m = 5**, **livre = 0** e **proximo = 0**:

Cálculo da posição a ser inserida será: **livre = livre % m**

1a. inserção:	livre = (0 % 5)	a posição livre no vetor é 0
2a. inserção: livre++	livre = (1 % 5)	a posição livre no vetor é 1
3a. inserção: livre++	livre = (2 % 5)	a posição livre no vetor é 2
4a. inserção: livre++	livre = (3 % 5)	a posição livre no vetor é 3
5a. inserção: livre++	livre = (4 % 5)	a posição livre no vetor é 4
6a. inserção: livre++	livre = (5 % 5)	a posição livre no vetor é 0
7a. inserção: livre++	livre = (1 % 5)	a posição livre no vetor é 1
...		

Cálculo da posição a ser recuperada será: **prox = prox % m**

1a. recuperação:	prox = (0 % 5)	a posição proximo no vetor é 0
2a. recuperação: prox++	proximo = (1 % 5)	a posição proximo no vetor é 1
3a. recuperação: prox++	proximo = (2 % 5)	a posição proximo no vetor é 2
4a. recuperação: prox++	proximo = (3 % 5)	a posição proximo no vetor é 3
5a. recuperação: prox++	proximo = (4 % 5)	a posição proximo no vetor é 4
6a. recuperação: prox++	proximo = (5 % 5)	a posição proximo no vetor é 0
7a. recuperação: prox++	proximo = (1 % 5)	a posição proximo no vetor é 1
...		

Para concluir repetiremos o que afirmamos anteriormente para as pilhas: “A implementação de uma fila por encadeamento é idêntica a de uma lista linear por encadeamento. Entretanto deve-se tomar cuidado em obedecer o critério de restrição que uma fila exige em operações de inserção e recuperação, independente de escolhermos a representação por contiguidade ou encadeamento”.

4.4. Atividades

1. O programa abaixo implementa uma pilha de números inteiros por contiguidade. Implemente as funções que estão sendo invocadas na função main() de forma que a aplicação funcione corretamente.

```
#define MAX 5          /* tamanho da pilha. */

void push(int i);      /* insere um elemento na pilha.*/
int pop(void);         /* recupera um elemento da pilha.*/
int getTamanho();      /* retorna o tamanho da pilha.*/

int pilha[MAX];        /* cria a pilha. */
int topo=0;            /* índice da posição livre e a ser recuperada. */
int tamanho=0;         /* armazena o tamanho da pilha. */

main(){
    push(1);
    push(2);
    push(3);
    push(4);
    printf("\nTamanho: %d",getTamanho());
    printf("\n%d",pop());
    printf("\n%d",pop());
    printf("\n%d",pop());
    printf("\nTamanho: %d",getTamanho());
    printf("\n%d",pop());
    printf("\n%d",pop());
    printf("\nTamanho: %d",getTamanho());
}
```

2. Responda as questões abaixo com base no programa desenvolvido anteriormente:

- a) Qual o número máximo de elementos que a pilha pode armazenar?
 - b) Qual o nome da variável que controla todas as operações sobre a pilha?
 - c) Quais as duas principais operações que se realiza com uma pilha e as respectivas funções implementadas no programa acima?
 - d) A forma como o seu programa representa a pilha está sujeita a limitações. Quais são essas limitações? O que fazer para eliminar essas limitações?
3. Implemente uma aplicação capaz de gerenciar um conjunto de números inteiros, seguindo o critério LIFO para armazenar e recuperar informações. Utilize a definição da estrutura de dados abaixo.

```
struct TipoPilha{
    int info;
    struct TipoPilha *prox;
};

typedef struct TipoPilha Pilha;
```

O programa deve disponibilizar as seguintes operações:

```
/* inserção de um elemento no topo da pilha. */
Pilha *insere(Pilha *atual, int v);

/* recupera um elemento do topo da pilha.*/
Pilha *recupera(Pilha *atual);

/* retorna o tamanho da pilha. */
int getTamanho(Pilha *atual);
```

4. Implemente um programa capaz de gerenciar um conjunto de senhas (string), seguindo o critério FIFO para armazenar e recuperar informações. Disponibilize APENAS as seguintes operações:
- a. Inserção de uma senha na fila.
 - b. Recuperação (atendimento) de uma senha da fila.

As senhas devem ser geradas da seguinte forma: IFET1, IFET2, IFET3, ...

Sugestão de opções para o usuário:

```
*****
SISTEMA DE GERENCIAMENTO DE SENHAS
*****
```

1. Gerar senha.
2. Atender ao usuário.
3. Sair do sistema.

* A opção 1 deve ser responsável por gerar uma nova senha e inseri-la na fila.

** A opção 2 deve ser responsável pela retirada da senha da fila.

5. Recursividade

Algo é recursivo se pode ser definido em termos de si próprio. Usaremos a definição de números naturais como exemplo. A definição formal dos números naturais diz que 0 (zero) é um número natural, e todo número natural tem um sucessor, que é também um número natural. Concluimos que em uma definição recursiva utilizamos objetos “previamente definidos” da mesma classe que está sendo definida.

Em termos de programação, podemos aplicar o conceito de recursividade ao implementarmos funções. Uma função é dita recursiva se invoca a si mesma, direta ou indiretamente. Naturalmente, toda função, recursiva ou não, deve possuir pelo menos uma chamada proveniente de um local exterior a ela.

Geralmente utilizamos a recursividade em problemas complexos através da resolução dos vários subproblemas mais simples, contribuindo para a solução do problema original.

Em muitos problemas estudados anteriormente poderíamos utilizar uma solução recursiva, mas preferimos utilizar as estruturas de repetição. Agora chegou o momento de aprendermos essa nova estratégia de programação. Sugiro começar a análise do problema identificando as duas partes de uma função recursiva:

- a) O **caso base**. Ao implementar uma função recursiva devemos nos preocupar com a sua finalização da mesma forma que fazemos ao elaborar uma estrutura de repetição. Não podemos permitir que as chamadas recursivas ocorram indefinidamente. O caso base é justamente a situação em que devemos interromper as chamadas recursivas e retornar os resultados.
- b) A **chamada recursiva**. A chamada recursiva incorpora a definição do problema, simplificando-o a cada chamada, procurando convergir para o caso base.

5.2. Somatório de inteiros

Vamos analisar um problema simples cuja solução pode ser implementada através da recursividade. Ao mesmo tempo, apresentaremos uma estratégia para identificar e incorporar o caso base e as chamadas recursivas na solução do problema.

Trata-se do somatório dos números inteiros positivos entre 1 e n . Veja a seguir.

$S_1 = 1$ → Caso base
 $S_2 = S_1 + 2$
 $S_3 = S_2 + 3$
 \dots
 $S_n = S_{n-1} + n$ } Chamadas Recursivas

```

int somaR(int n){
    if (n == 1){ //caso base
        return 1;
    }
    return somaR(n-1) + n; //ch recursivas
}
  
```

Vimos anteriormente que as funções recursivas apresentam muita semelhança com as estruturas de repetição. Então o ponto de partida para qualquer solução recursiva é identificar o padrão, relacionado ao problema, que se repete. No problema em questão o padrão repetitivo é o acúmulo de valores.

A outra preocupação que devemos ter é verificar se o problema é recursivo, ou seja, se a solução do problema depende da solução de subproblemas de menor nível. Nesse caso percebe-se que para cada cálculo há uma dependência de cálculos anteriores, exceto para o somatório do número 1. Pronto, aí está a recursividade!

Agora o próximo passo é identificar as duas partes de uma função recursiva: o caso base e as chamadas recursivas. A exceção citada no parágrafo anterior será o caso base, pois sinaliza o fim das chamadas recursivas, que neste caso é quando n for igual a 1.

Tente associar essa percepção com o código listado na coluna da direita. Sugerimos que utilize essa estratégia ao implementar suas funções recursivas. Agora vamos completar o entendimento analisando a pilha de execução criada pelas diversas chamadas à função `somaR()`.

Por *default*, quando um parâmetro é passado para uma função, uma cópia do seu valor é colocada em uma posição de memória temporária chamada **pilha de execução**. E esta cópia do valor permanece na pilha até que a função seja finalizada. Quando implementamos uma função recursiva, as chamadas da função são realizadas sem que as chamadas anteriores sejam finalizadas, com exceção do caso base.

1ª chamada

```
printf("\nSoma: %d", somaR(5));
```

1ª execução seguida da 2ª chamada

```

int somaR(int n){
    if (n == 1){
        return 1;
    }
    return somaR(n-1) + n;
}
  
```

Pilha
n
5

2ª execução seguida da 3ª chamada

```
int somaR(int n){  
    if (n == 1){  
        return 1;  
    }  
    return somaR(n-1) + n;  
}
```

Pilha
n
5
4

3ª execução seguida da 4ª chamada

```
int somaR(int n){  
    if (n == 1){  
        return 1;  
    }  
    return somaR(n-1) + n;  
}
```

Pilha
n
5
4
3

4ª execução seguida da 5ª chamada

```
int somaR(int n){  
    if (n == 1){  
        return 1;  
    }  
    return somaR(n-1) + n;  
}
```

Pilha
n
5
4
3
2

5ª execução e sua finalização

```
int somaR(int n){  
    if (n == 1){  
        return 1;  
    }  
    return somaR(n-1) + n;  
}
```

Pilha
n
5
4
3
2
1

- Devido a ocorrência do caso base, as sucessivas chamadas recursivas serão interrompidas e a execução atual será finalizada, com o valor de retorno igual a 1.
- O fluxo de execução retornará para a 4ª execução, que é o ponto de origem dessa 5ª execução.
- Os dados da pilha, relativos a essa execução, serão liberados (o valor 1), ocorrerá o desempilhamento.

Retorno para a 4ª execução e sua finalização

```
int somaR(int n){
    if (n == 1){
        return 1;
    }
    return somaR(n-1) + n;
}
```

Diagram showing the return value 2 for the recursive call somaR(n-1) and the final result 2.

Pilha
n
5
4
3
2

- A linha destacada será executada da seguinte forma: return 3
- O fluxo de execução retornará para a 3ª execução, que é o ponto de origem dessa 4ª execução.
- Os dados da pilha, relativos a essa execução, serão liberados (o valor 2), ocorrerá o desempilhamento.

Retorno para a 3ª execução e sua finalização

```
int somaR(int n){
    if (n == 1){
        return 1;
    }
    return somaR(n-1) + n;
}
```

Diagram showing the return value 3 for the recursive call somaR(n-1) and the final result 3.

Pilha
n
5
4
3

- A linha destacada será executada da seguinte forma: return 6
- O fluxo de execução retornará para a 2ª execução, que é o ponto de origem dessa 3ª execução.
- Os dados da pilha, relativos a essa execução, serão liberados (o valor 3), ocorrerá o desempilhamento.

Retorno para a 2ª execução e sua finalização

```
int somaR(int n){
    if (n == 1){
        return 1;
    }
    return somaR(n-1) + n;
}
```

Diagram showing the return value 6 for the recursive call somaR(n-1) and the final result 6.

Pilha
n
5
4

- A linha destacada será executada da seguinte forma: return 10
- O fluxo de execução retornará para a 1ª execução, que é o ponto de origem dessa 2ª execução.
- Os dados da pilha, relativos a essa execução, serão liberados (o valor 4), ocorrerá o desempilhamento.

Retorno para a 1ª execução e sua finalização

```
int somaR(int n){
    if (n == 1){
        return 1;
    }
    return somaR(n-1) + n;
}
```

Diagram showing the return value 10 for the recursive call somaR(n-1) and the final result 10.

Pilha
n
5

- A linha destacada será executada da seguinte forma: return 15.
- O fluxo de execução retornará para a chamada externa que é o ponto de origem dessa 1ª execução.
- Os dados da pilha, relativos a essa execução, serão liberados (o valor 5), ocorrerá o desempilhamento. A pilha estará vazia!

5.3. Cálculo do fatorial

Veremos mais um problema clássico que também pode ter uma solução recursiva. Trata-se do cálculo do fatorial. Podemos definir o fatorial de um número n da seguinte forma: $n! = (n) \times (n-1) \times (n-2) \dots \times 1$ ou desta forma: $n! = n \times (n-1)!$

Utilizando a mesma estratégia do problema anterior, percebemos um padrão se repetindo que é o acúmulo de valores. Verifica-se também que se trata de um problema recursivo pois o cálculo do fatorial de cada número depende do cálculo do fatorial do número anterior. A exceção fica por conta do valor igual a 1, pois o fatorial desse número é, por definição igual a 1, portanto não dependendo de nenhum outro cálculo, olha o caso base aí!

Vejamos uma definição mais formal para o cálculo do fatorial de um número n , que inclua a recursividade e suas duas partes. Agora definimos o fatorial de n a partir dos fatoriais dos números naturais menores que ele, ou seja, para calcular o fatorial de um determinado número há necessidade de se recorrer aos fatoriais dos números anteriores, com exceção para n igual a 1.

$$n! = \begin{cases} 1, & \text{se } n = 1 \longrightarrow \text{Caso base} \\ n * (n-1)!, & \text{se } n > 1 \longrightarrow \text{Chamadas Recursivas} \end{cases}$$

Agora fica fácil a implementação da função recursiva:

```
int fatR(int n){
    if (n==1){
        return 1;    //caso base
    }
    return n * fatR(n-1);    //chamadas recursivas
}
```

5.4. Atividades

1. Acredito que o entendimento sobre recursividade depende muito da compreensão que temos sobre a pilha de execução de uma função recursiva. Portanto sugiro que construa a pilha de execução que será formada/liberada devido as diversas chamadas à função `fatR()`, implementada acima.
2. Escreva um código recursivo para imprimir o n -ésimo termo da sequência de Fibonacci. A sequência de Fibonacci é definida da seguinte forma: O primeiro termo é 1. O segundo também é 1. Cada termo da sequência é definido como sendo a soma dos dois termos anteriores. Segue abaixo a definição recursiva dos termos da série de Fibonacci:

```

fib(1) = 1
fib(2) = 1
fib(n) = fib(n-1) + fib(n-2), para n > 2

```

Exemplo: 1,1,2,3,5,8,13,21,34,....

3. Escreva um código recursivo para imprimir os números inteiros de 1 a n.
4. Escreva um código recursivo para imprimir os números ímpares de 1 a n.
5. Escreva um código recursivo para imprimir os números pares de 1 a n.
6. Faça um esquema mostrando o estado da pilha da memória durante as chamadas da função X abaixo.

```

int X (int n, int m){
    if ((n==m) || (m==0) || (n==0)){
        return 1;
    }
    return X(n-1,m)+X(n-1,m+1);
}

```

- a) Qual o valor de X(5,3)?
- b) Quantas chamadas serão feitas à função X?

7. Mostre, através de teste de mesa, os resultados das seguintes funções:

A)

```

int f1(int n) {
    if (n==1) || n==0){
        return 1;
    }
    if (n<0){
        return 0;
    }
    return n * f1(n-1);
}

```

Considere as chamadas:

```

f1(0);
f1(1);
f1(5);

```

B)

```

int f2(int n) {
    if (n == 0) {
        return 1;
    }

    if (n == 1){
        return 1;
    } else {
        return f2(n-1)+ 2 * f2(n-2);
    }
}

```

Considere as chamadas:

```

f2(0);
f2(1);
f2(5);

```

6. Árvores

Árvores são estruturas de dados que apresentam características que permitem superar as limitações impostas pelas estruturas lineares como listas, pilhas e filas. São ideais para estabelecer hierarquia entre seus elementos, conforme podemos perceber nas diversas imagens abaixo.

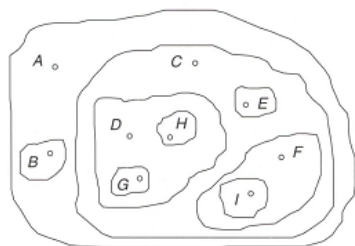
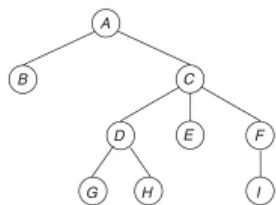


Diagrama de Inclusão



Representação Hierárquica

(A (B) (C (D (G) (H)) (E) (F (I))))

Parênteses Aninhados

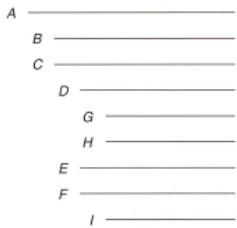


Diagrama de Barras

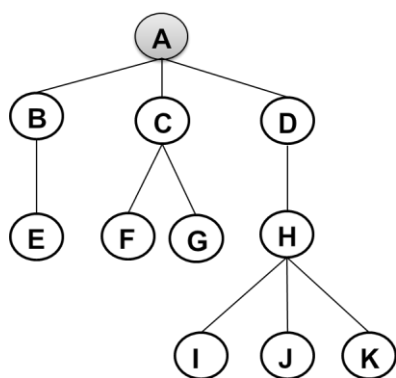
Antes de estudarmos os algoritmos de manipulação de árvores é importante conhecer os principais elementos de uma árvore. Raiz é o nó localizado no topo da árvore, é o ponto de partida para qualquer algoritmo de manipulação de árvore.

Folha é o nó localizado em alguma extremidade da árvore. Tem como característica não possuir nós filhos.

O grau de um nó indica o número de subárvores que possui. O grau de uma árvore é igual ao do nó de grau máximo, considerando todos os nós.

O nível de um nó é o comprimento do caminho que vai da raiz até o nó considerado.

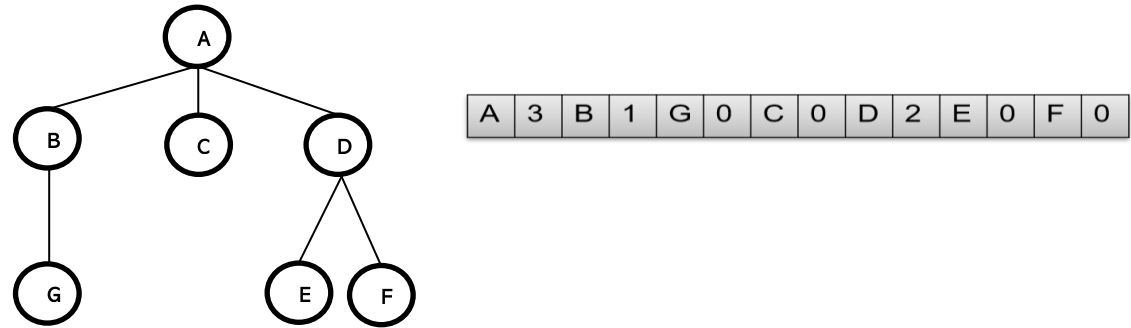
A altura, ou profundidade, da árvore é o nível do nó folha que tem o mais longo caminho até a raiz. A árvore vazia é uma árvore de altura -1, por definição. Uma árvore com um único nó, tem altura 0.



Nó	Grau	Nível
A	3	0
B	1	1
C	2	1
D	1	1
E	0	2
F	0	2
G	0	2
H	3	2
I	0	3
J	0	3
K	0	3

Assim como todas as estruturas estudadas até o momento, a árvore pode ser representada por contiguidade ou encadeamento. A alocação de árvores por contiguidade não é muito utilizada devido às dificuldades para realizar operações com um nó específico (inserções, remoções e busca). Entretanto esta forma de alocação é útil quando os nós da árvore devem ser processados na mesma ordem em que aparecem alocados fisicamente.

Veja abaixo um exemplo de representação por contiguidade. Perceba que os números que aparecem à direita da informação indicam o grau do nó correspondente. Cada nó armazena sua respectiva informação e referências das suas subárvores.



A	3	B	1	G	0	C	0	D	2	E	0	F	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---

Dedicaremos apenas ao estudo de árvores com representação por encadeamento pois permite maior facilidade de manipulação, com diversas ordens de acesso aos nós.

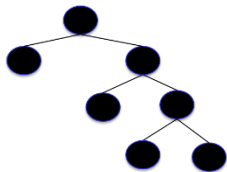
7. Árvore Binária

Árvores binárias são aplicadas em diversos problemas, destacamos as operações de busca e ordenação. Como outro exemplo, podemos citar a utilização na representação de expressões aritméticas sob forma hierárquica que estabeleça prioridade dos operadores envolvidos.

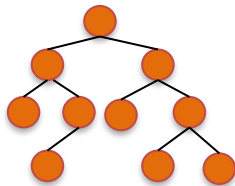
São estruturas do tipo árvore onde o grau de cada nó é menor ou igual a 2. Quando o grau do nó for igual a 2 temos a subárvore a direita (sad) e a subárvore a esquerda (sae), quando o grau do nó for igual a 1 temos uma subárvore direita ou uma subárvore esquerda e se o grau do nó for igual a 0 não temos subárvores.

A regra relacionada ao grau, que caracteriza a árvore binária, é válida todos os seus nós. Conclui-se que muitas operações sobre árvores binárias utilizam recursão. É o que veremos mais adiante ao estudarmos seus algoritmos.

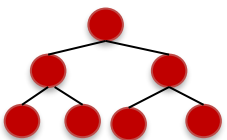
7.2. Tipos de árvores binárias



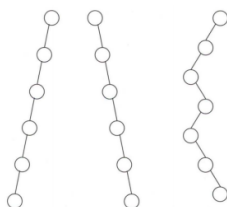
Árvore **estritamente binária**: cada nó possui 0 ou 2 filhos.



Árvore binária **completa**: se v é um nó tal que alguma subárvore de v é vazia, então v se localiza ou no último (maior) ou no penúltimo nível da árvore.



Árvore binária **cheia**: se v é um nó tal que alguma subárvore de v é vazia, então v se localiza no último (maior) nível da árvore. v é um nó folha. Toda árvore binária cheia é completa.



Na árvore **zigzag**, seus nós interiores possuem exatamente uma subárvore vazia.

7.3. A importância da altura mínima

Quando analisamos a complexidade de algoritmos relacionados às árvores, devemos dar atenção especial à altura. Em uma árvore zigzague, sua altura é igual ao número de nós, ou seja, sua **altura é máxima** para uma árvore binária. Trata-se de uma árvore binária degenerada pois transformou-se em uma lista, apresentando uma **complexidade de busca da ordem de $O(n)$** .

Em outro extremos, temos as árvores binárias completa e a cheia. Essas, sempre apresentam **altura mínima**. A eficiência, no pior caso, do algoritmo de busca, por exemplo, será tão maior quanto menor for a altura da árvore. É uma conclusão óbvia relacionarmos a altura mínima com algo positivo em termos de eficiência de um algoritmo.

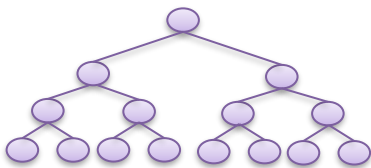
Em nossos estudos anteriores havíamos comentado sobre alguns algoritmos que apresentam complexidade da ordem de **$O(\log n)$** . Agora chegou o momento de entendermos isso melhor, saber a relação da complexidade logarítmica com a altura de uma árvore.

Nós	Altura	
1	0	$\log 1$
2	1	$\log 2$
3	1	$\log 3$
4	2	$\log 4$
5	2	$\log 5$
6	2	$\log 6$
7	2	$\log 7$
8	3	$\log 8$
9	3	$\log 9$
10	3	$\log 10$
11	3	$\log 11$
12	3	$\log 12$
13	3	$\log 13$
14	3	$\log 14$
15	3	$\log 15$
16	4	$\log 16$

No quadro ao lado percebemos a relação entre o logaritmo de n (quantidade de nós) e a altura da árvore.

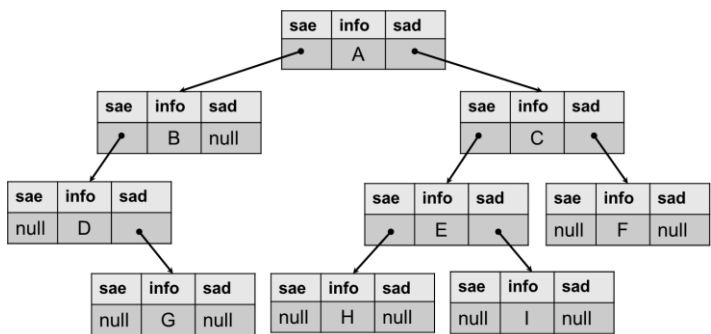
Fica fácil perceber que a altura da árvore de n nós é proporcional a **$\lfloor \log n \rfloor$** .

Daí, podemos afirmar que a complexidade do algoritmo de busca em uma árvore binária é da ordem de **$O(\log n)$**



7.4. Operações

Veremos a seguir os principais algoritmos de manipulação de árvore binária. Consideraremos apenas as representações de árvores binárias por encadeamento, onde cada nó é formado por uma estrutura composta de campo de informação, um ponteiro para a *sae* e outro ponteiro para a *sad*.



Todos os algoritmos que analisaremos terão como base a definição abaixo. Trata-se de uma árvore binária encadeada de números inteiros. Todos os códigos apresentados podem ser adaptados para qualquer outro tipo de dado básico ou estruturado.

Definição do nó da árvore

```
struct TipoArvore {
    int info;
    struct TipoArvore* sae;
    struct TipoArvore* sad;
};
typedef struct TipoArvore Arvore;
```

Protótipos de algumas funções

```
//cria uma árvore vazia
Arvore* inicializa();

//verifica se a arvore está vazia
int estaVazia(Arvore* a);

//cria um nó, dado a informação e as duas subárvores
Arvore* criaNo(int n, Arvore* sae, Arvore* sad);

//libera a estrutura da árvore
Arvore* libera(Arvore* a);

//Determinar se uma informação se encontra ou não na árvore
int busca(Arvore* a, int n);

//imprime a informação de todos os nós da árvore
void imprime(Arvore* a);
```

Programa principal

```
1 int main(){
2     Arvore *D = criaNo(4, inicializa(), inicializa());
3     Arvore *E = criaNo(5, inicializa(), inicializa());
4     Arvore *F = criaNo(6, inicializa(), inicializa());
5     Arvore *G = criaNo(7, inicializa(), inicializa());
6     Arvore *B = criaNo(2, D, E);
7     Arvore *C = criaNo(3, F, G);
8     Arvore *A = criaNo(1, B, C);
9
10    imprime(A);
11
12    if (!busca(A, 113)){
13        printf("\nInformacao INEXISTENTE!");
14    }else{
15        printf("\nInformacao ENCONTRADA COM SUCESSO!");
16    }
17
18    A = libera(A);
19
20    imprime(A);
21    return 0;
22 }
```

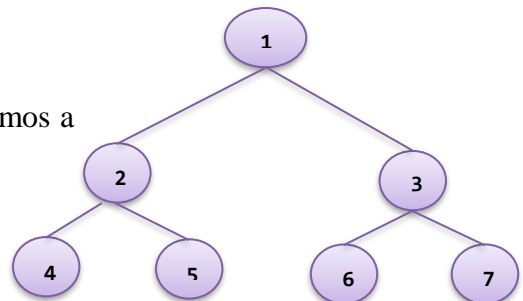
A seguir analisaremos detalhadamente cada função de acordo com a sua invocação pela função main listada acima.

Código invocado pela função main linhas 2,3,4,5,6,7 e 8

```
/* cria uma árvore vazia. */
Arvore* inicializa(){
    return NULL;
}

/* cria um nó com a informação, a sae e a sad. */
Arvore* criaNo(int n, Arvore* sae, Arvore* sad){
    Arvore* p = (Arvore*) malloc(sizeof(Arvore));
    p->info = n;
    p->sae = sae;
    p->sad = sad;
    return p;
}
```

Após a execução das linhas 1 até 8, da função main, teremos a seguinte árvore:



Código invocado pela função main linha 10: imprime(A)

```
/* imprime a informação corresponde a cada nó da árvore. */
void imprime(Arvore* a){
    if(!estaVazia(a)){
        printf("%d ", a->info); /* mostra raiz */
        imprime(a->sae); /* mostra sae */
        imprime(a->sad); /* mostra sad */
    }
}

/* verifica se a arvore está vazia. */
int estaVazia(Arvore* a){
    return a == NULL;
}
```

Código invocado pela função main linha 12: if (!busca(A, 113))...

```
/* busca pela chave */
int busca(Arvore* a, int n){
    if(estaVazia(a)){
        return 0;
    }else{
        if (a->info == n){
            return 1;
        }else{
            if (busca(a->sae, n)){
                return 1;
            }else{
                return busca(a->sad, n);
            }
        }
    }
}
```

Código invocado pela função main linha 18: A = libera(A)

```
/* Libera a árvore da memória */
Arvore* libera(Arvore* a){
    if(!estaVazia(a)){
        libera(a->sae); /* libera sae */
        libera(a->sad); /* libera sad */
        free(a); /* libera raiz */
    }
    return NULL;
}
```

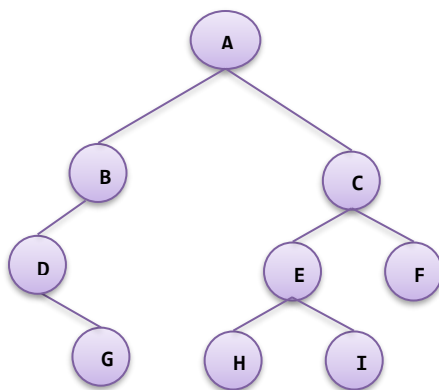
7.5. Percurso em profundidade

Uma das operações básicas relativas à manipulação de árvores é a visita sistemática a cada um dos seus nós. Para percorrer a árvore deve-se, então, visitar cada um de seus nós.

Visitar um nó significa operar com a informação do nó. Por exemplo: imprimir (conforme analisado anteriormente), atualizar informações, etc. Especificamente, o percurso em profundidade, corresponde a visitar um nó raiz e explorar tanto quanto possível cada um dos seus ramos, antes de retroceder. Estudaremos 3 tipos de percurso em profundidade:

- **Pré-ordem:**
 - ✓ **Visitar a raiz**
 - ✓ Percorrer a sub-árvore esquerda em pré-ordem
 - ✓ Percorrer a sub-árvore direita em pré-ordem
- **In-ordem:**
 - ✓ Percorrer a sub-árvore esquerda em in-ordem
 - ✓ **Visitar a raiz**
 - ✓ Percorrer a sub-árvore direita em in-ordem
- **Pós-ordem:**
 - ✓ Percorrer a sub-árvore esquerda em pós-ordem
 - ✓ Percorrer a sub-árvore direita em pós-ordem
 - ✓ **Visitar a raiz**

Veja os 3 algoritmos de percurso em profundidade:



Pre-ordem: A B D G C E H I F

```
/* imprime todos os nós da árvore em Pre-Ordem. */
void imprimePre(Arvore* a){
    if(!estaVazia(a)){
        printf("%d ", a->info); /* mostra raiz */
        imprimePre(a->sae); /* mostra sae */
        imprimePre(a->sad); /* mostra sad */
    }
}
```

In-ordem: D G B A H E I C F

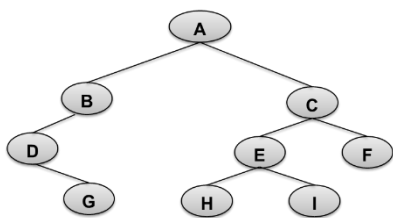
```
/* imprime todos os nós da árvore em In-Ordem. */
void imprimeIn(Arvore* a){
    if(!estaVazia(a)){
        imprimeIn(a->sae); /* mostra sae */
        printf("%d ", a->info); /* mostra raiz */
        imprimeIn(a->sad); /* mostra sad */
    }
}
```

Pos-ordem: G D B H I E F C A

```
/* imprime todos os nós da árvore em Pos-Ordem. */
void imprimePos(Arvore* a){
    if(!estaVazia(a)){
        imprimePos(a->sae); /* mostra sae */
        imprimePos(a->sad); /* mostra sad */
        printf("%d ", a->info); /* mostra raiz */
    }
}
```

7.6. Percurso em largura

Percorrer uma árvore em largura corresponde a visitar cada nó começando pelo nível mais baixo (ou mais alto) e movendo-se para baixo (ou para cima) nível a nível, visitando todos os nós em cada nível da esquerda para direita (ou da direita para a esquerda).



Para a árvore ao lado o percurso em largura produziria o seguinte caminho: A, B, C, D, E, F, G, H e I.

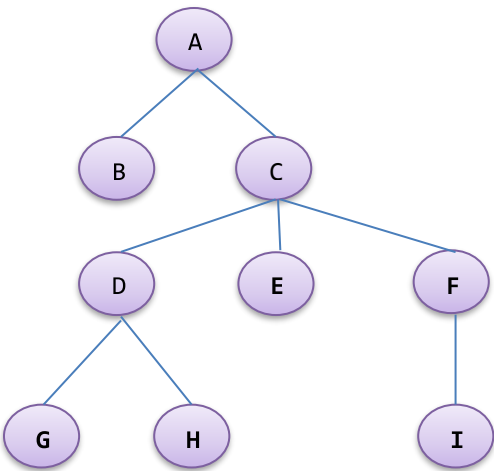
O percurso será realizado de cima para baixo e da esquerda para direita usando uma fila. Quando um nó é visitado, seus filhos (se houver) são colocados no final da fila e em seguida o nó no início da fila é visitado. Veja abaixo.

Visita A – insere B e C na fila	B C
Visita B – insere D na fila	C D
Visita C – insere E e F na fila	D E F
Visita D – insere G na fila	E F G
Visita E – insere H e I na fila	F G H I
Visita F	G H I
Visita G	H I
Visita H	I
Visita I	

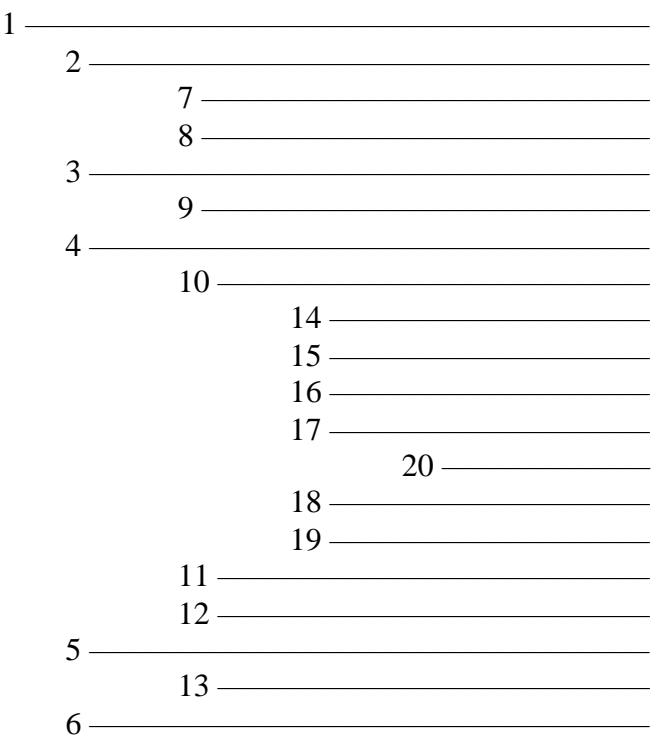
7.7. Atividades

1. Considerando a árvore ao lado, responda:

- a) Quais os nós folhas?
- b) Qual o grau de cada nó?
- c) Qual o grau da árvore?

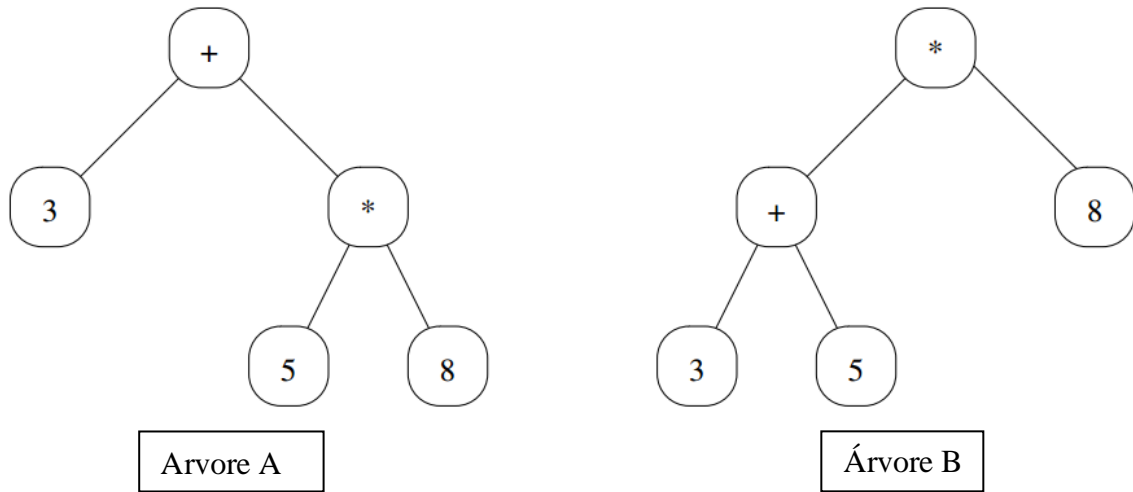


2. Redesenhe árvore abaixo na forma tradicional:

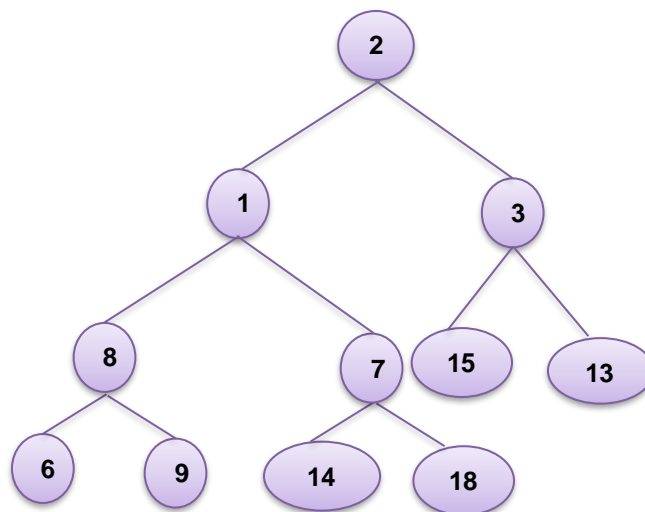


- 3. Quantos antecedentes tem um nó no nível n em uma árvore binária cheia?
- 4. Uma árvore estritamente binária com n nós folhas contém quantos nós?

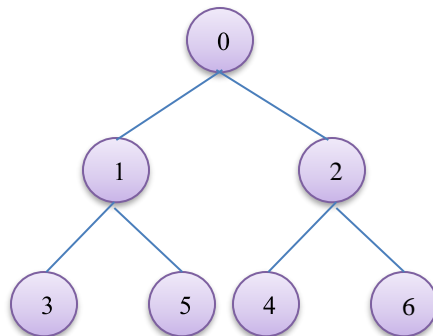
5. Qual a quantidade mínima e máxima de nós folhas que uma árvore estritamente binária pode apresentar?
6. Considere árvores binárias apresentadas abaixo.



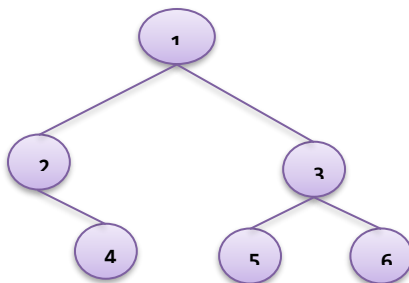
- a) Qual a expressão aritmética correspondente à árvore A? Qual é o seu valor?
 - b) Qual a expressão aritmética correspondente à árvore B? Qual é o seu valor?
7. Desenhe a árvore binária correspondente à seguinte operação: $3 * [10 / (2 * 1)]$.
 8. Indique o percurso em pré-ordem, in-ordem e pós-ordem da árvore abaixo.



9. Indique o percurso em largura para a árvore abaixo. Informe a ordem em que cada nó é visitado e o correspondente estado da fila.



10. Considere uma árvore binária cheia de altura h . Demostre que a quantidade total de nós da árvore é dada pela fórmula $2^{h+1} - 1$.
11. Demostre que uma árvore binária completa com $n > 0$ nós, possui altura h igual a $\lfloor \log n \rfloor$. Apresente sua função de complexidade $T(n)$ para o pior caso e a ordem da função de complexidade segundo a notação O .
12. Usando as funções `inicializa()` e `criaNo()`, crie uma estrutura que corresponda à seguinte árvore:



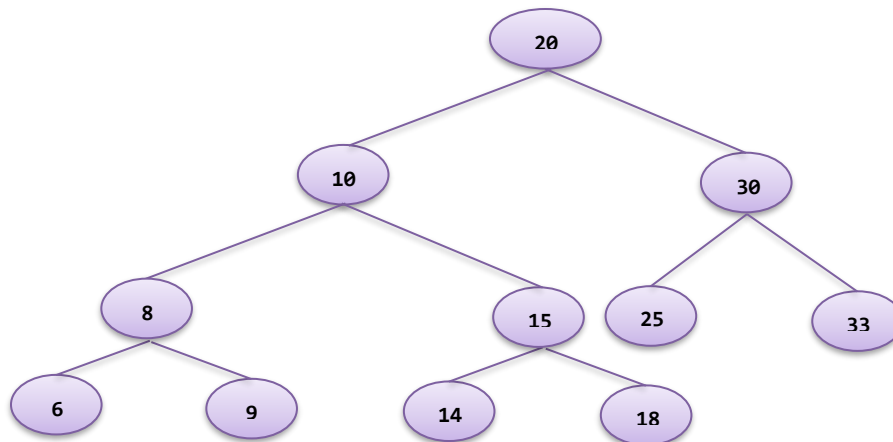
13. Implemente todas as funções analisadas neste capítulo, correspondente às operações com uma árvore binária.
14. Acrescente ao programa criado no item anterior duas funções: Segue abaixo os protótipos:

```
/* Determina a altura da árvore. */  
int getAltura(Arvore* a);
```

```
/* Calcula a quantidade de folhas de uma árvore. */  
int getFolhas(Arvore* a)
```

8. Árvore binária de busca

Trata-se de uma especialização de árvore binária, acrescentando a seguinte propriedade fundamental: *o valor da chave da raiz é maior do que o valor da chave da subárvore da esquerda (SAE) e menor do que o valor da chave da subárvore da direita (SAD)*. Para que uma árvore seja considerada binária de busca, todos os seus nós devem obedecer a esta propriedade fundamental.



8.2. Operações

Muitas funções, analisadas anteriormente quando estudamos árvores binárias, podem ser aproveitadas para árvores binárias de busca, como por exemplo as funções de inicialização, de criação de nós, verificação se a árvore está vazia e as funções de percurso.

Analisaremos, em detalhes, algumas funções que exploram a propriedade de ordenação das árvores binárias de busca, como a inserção e busca por um elemento na árvore.

BUSCA

A solução para o problema de busca em árvores binárias de busca pode ser apresentada de forma recursiva da seguinte forma:

- Duas situações de caso base:
 1. Quando chave = valor do nó → achou!
 2. Quando sair da árvore após busca → com certeza não achou!
- Duas situações de chamadas recursivas:
 1. Chave < valor do nó então buscar chave na subárvore da esquerda (SAE).
 2. Chave > valor do nó então buscar chave na subárvore da direita (SAD)

Se implementarmos a análise acima teremos o seguinte código:

Busca por uma informação na árvore

```
int busca(Arvore* a, int c){
    if(vazia(a)){ /* foi até o final e não achou.*/
        return 0;
    }else{
        if (a->info == c){ /* indica que achou.*/
            return 1;
        }else{
            if (c < a->info){
                return busca(a->sae, c);
            }else{
                if (c > a->info){
                    return busca(a->sad, c);
                }
            }
        }
    }
}
```

INSERÇÃO

Devemos inserir o novo nó como um nó folha, respeitando a propriedade fundamental da árvore binária de busca. Veja abaixo que a função de inserção é muito semelhante à função de busca, já que sua efetivação depende da busca pelo ponto correto de inserção do nó na árvore.

O ponto correto de inserção de um novo nó é quando percorrermos a árvore e achamos um nó que não possui sae ou sad e que obedeça a propriedade fundamental da árvore binária de busca em relação ao nó a ser inserido.

Inserção de um novo nó

```
Arvore* inserir(Arvore* a, int c){
    if (estaVazia(a)){ //achou o ponto correto de inserção
        return criaNo(c, inicializa(), inicializa());
    }else{
        if (c < a->info){
            a->sae = inserir(a->sae, c);
        }
        if (c > a->info){
            a->sad = inserir(a->sad, c);
        }
        return a; //nó a ser inserido já existe, nada muda!
    }
}
```

8.3. Atividades

Algumas atividades serão baseadas no programa abaixo, que implementa uma árvore binária cujos nós são números inteiros.

arvorebinaria.c

```
1 struct TipoArvore {
2     int info;
3     struct TipoArvore* sae;
4     struct TipoArvore* sad;
5 };
6
7 typedef struct TipoArvore Arvore;
8
9 //cria uma árvore vazia
10 Arvore* inicializa();
11
12 //verifica se a arvore está vazia
13 int estaVazia(Arvore* a);
14
15 //cria um nó dadas a informação e as duas subárvores
16 Arvore* criaNo(int n, Arvore* sae, Arvore* sad);
17
18 //libera a estrutura da árvore
19 Arvore* libera(Arvore* a);
20
21 //Determinar se uma informação se encontra ou não na árvore
22 int busca(Arvore* a, int n);
23
24 //imprime a informação de todos os nós da árvore em Pre-Ordem
25 void imprimePre(Arvore* a);
26
27 //imprime a informação de todos os nós da árvore em In-Ordem
```

```

28 void imprimeIn(Arvore* a);
29
30 //imprime a informação de todos os nós da árvore em Pos-Ordem
31 void imprimePos(Arvore* a);
32
33 int main(){
34     Arvore *D = criaNo(3, inicializa(), inicializa());
35     Arvore *E = criaNo(5, inicializa(), inicializa());
36     Arvore *F = criaNo(4, inicializa(), inicializa());
37     Arvore *G = criaNo(6, inicializa(), inicializa());
38     Arvore *B = criaNo(1, D, E);
39     Arvore *C = criaNo(2, F, G);
40     Arvore *A = criaNo(0, B, C);
41
42     imprimePre(A);
43     printf("\n");
44     imprimeIn(A);
45     printf("\n");
46     imprimePos(A);
47
48     if (!busca(A,111)){
49         printf("\nInformacao INEXISTENTE!");
50     }else{
51         printf("\nInformacao ENCONTRADA COM SUCESSO!");
52     }
53
54     A = libera(A);
55     imprimePre(A);
56     printf("\n");
57     imprimeIn(A);
58     printf("\n");
59     imprimePos(A);
60
61     return 0;
62 }
63
64 //cria uma árvore vazia
65 Arvore* inicializa(){
66     return NULL;
67 }
68
69 //verifica se a arvore está vazia
70 int estaVazia(Arvore* a){
71     return a == NULL;
72 }
73
74 //cria um nó dadas a informação e as duas subárvores
75 Arvore* criaNo(int n, Arvore* sae, Arvore* sad){
76     Arvore* p = (Arvore*) malloc(sizeof(Arvore));
77     p->info = n;

```

```

78     p->sae = sae;
79     p->sad = sad;
80     return p;
81 }
82
83 //libera a estrutura da árvore
84 Arvore* libera(Arvore* a){
85     if(!estaVazia(a)){
86         libera(a->sae); /* libera sae */
87         libera(a->sad); /* libera sad */
88         free(a); /* libera raiz */
89     }
90     return NULL;
91 }
92
93 //Determinar se uma informação se encontra ou não na árvore
94 int busca(Arvore* a, int n){
95     if(estaVazia(a)){
96         return 0;
97     }else{
98         if (a->info == n){
99             return 1;
100         }else{
101             if (busca(a->sae, n)){
102                 return 1;
103             }else{
104                 return busca(a->sad, n);
105             }
106         }
107     }
108 }
109
110 //imprime a informação de todos os nós da árvore em Pre-Ordem
111 void imprimePre(Arvore* a){
112     if(!estaVazia(a)){
113         printf("%d ", a->info); /* mostra raiz */
114         imprimePre(a->sae); /* mostra sae */
115         imprimePre(a->sad); /* mostra sad */
116     }
117 }
118
119 //imprime a informação de todos os nós da árvore em In-Ordem
120 void imprimeIn(Arvore* a){
121     if(!estaVazia(a)){
122         imprimeIn(a->sae); /* mostra sae */
123         printf("%d ", a->info); /* mostra raiz */
124         imprimeIn(a->sad); /* mostra sad */
125     }
126 }
127

```

```

128 //imprime a informação de todos os nós da árvore em Pos-Ordem
129 void imprimePos(Arvore* a){
130     if(!estaVazia(a)){
131         imprimePos(a->sae); /* mostra sae */
132         imprimePos(a->sad); /* mostra sad */
133         printf("%d ", a->info); /* mostra raiz */
134     }
135 }

```

1. Adaptar o programa disponibilizado acima para que possa manipular uma Árvore Binária de **Busca**. As funções `insere()` e `busca()` foram analisadas neste capítulo.
2. Crie uma função de nome `ImprimeAscendenteOrdem` que seja capaz de imprimir todos os nós da árvore em ordem ascendente.
3. Crie uma função de nome `ImprimeDescendenteOrdem` que seja capaz de imprimir todos os nós da árvore em ordem descendente.
4. Codificar uma função capaz de determinar o maior elemento armazenado em uma árvore. No caso de árvore vazia a função deve retornar 0. Lembre-se de que a busca deve se concentrar apenas na subárvore à direita. A função deve ter o seguinte protótipo:

```

//Determinar o maior elemento armazenado na árvore
int buscaMaior(Arvore* a);

```

5. Codificar uma função capaz de determinar o menor elemento armazenado em uma árvore. No caso de árvore vazia a função deve retornar 0. Lembre-se de que a busca deve se concentrar apenas na subárvore à esquerda. A função deve ter o seguinte protótipo:

```

//Determinar o menor elemento armazenado na árvore
int buscaMenor(Arvore* a);

```

6. Considere que o ponteiro `tree`, aponta para o nó raiz de uma árvore binária de busca vazia. Considere também a função `inserir` como aquela analisada neste capítulo. Faça um desenho que represente a árvore criada pelas instruções abaixo.

```

tree = inserir(tree, 7);
tree = inserir(tree, 9);
tree = inserir(tree, 5);
tree = inserir(tree, 1);
tree = inserir(tree, 8);
tree = inserir(tree, 4);
tree = inserir(tree, 10);
tree = inserir(tree, 61);
tree = inserir(tree, -1);

```

E ainda:

- a) Indique o caminho em profundidade pre-ordem, in-ordem e pós-ordem.
 - b) Qual desses percursos imprime os nós da árvore criada em ordem ascendente?
 - c) Qual desses percursos imprime os nós da árvore criada em ordem descendente?
7. Altere a função main deste programa, para criar uma árvore binária de busca com 100 números aleatórios (entre 1 e 1000 inclusive). Não poderá haver duplicidade de valores na árvore. Em seguida os números deverão ser impressos em ordem ascendente e descendente.
8. A função abaixo calcula a quantidade total de nós de uma árvore binária de busca.

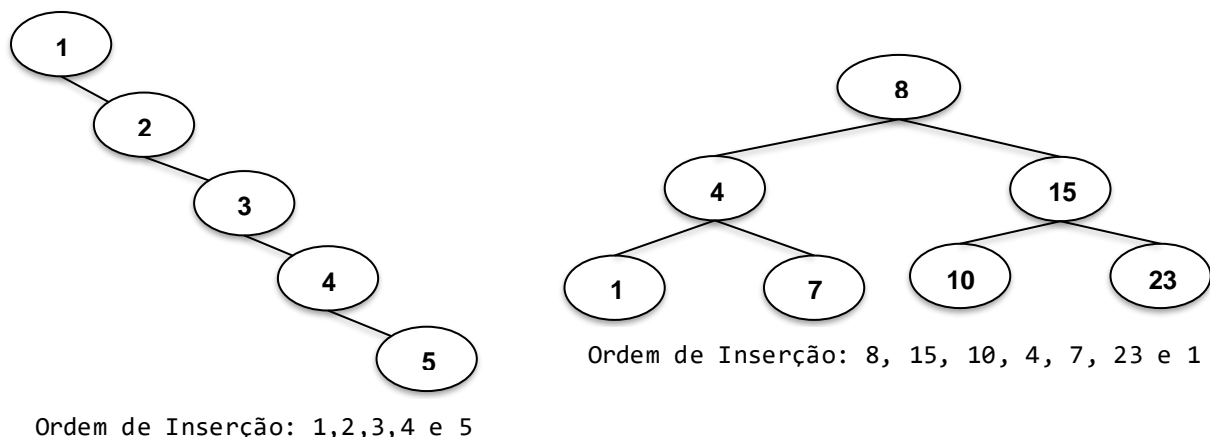
```
int getNos(Arvore* a){
    if(estaVazia(a)){
        return 0;
    }else{
        int n = 1;    //contabiliza o nó raiz
        n = n + getNos(a->sae); /* busca sae */
        n = n + getNos(a->sad); /* busca sad */
        return n;
    }
}
```

Adaptando a função acima, codificar uma função para calcular a quantidade de nós **folhas** e outra para calcular o **somatório** de todos os valores armazenados na árvore.

9. Árvore AVL

Vimos anteriormente que a altura da árvore binária completa e da cheia, com n nós, é proporcional a $\lfloor \log n \rfloor$ e que a complexidade do algoritmo de busca em uma árvore deste tipo está diretamente relacionado com a sua altura e é da ordem de $O(\log n)$. Trata-se de uma complexidade melhor do que a proporcionada pelas listas lineares que é da ordem de $O(n)$.

Essa eficiência pode se tornar nula se a complexidade deixar de ser logarítmica e passar a ser linear. As árvores binárias de busca, se construídas por inserção **aleatória** de elementos têm altura logarítmica, em n , na média. Portanto, **não** podemos assegurar que árvores binárias de busca construídas segundo qualquer ordem de inserção sempre terão altura logarítmica.



Queremos chamar sua atenção para a importância da ordem como a informação é armazenada em uma árvore. As duas árvores acima são consideradas árvores binárias de busca, porém uma delas está totalmente **desbalanceada** ou **desregulada**, devido a ordem de inserção.

A complexidade dos algoritmos de busca, inserção ou remoção, em uma árvore desbalanceada deixa de ser $O(\log n)$ e passa a ser $O(n)$. Por isso devemos evitar que uma árvore binária se torne desbalanceada ou deixe de ser completa. Então temos o seguinte problema: como manter uma árvore balanceada?

Uma solução é estabelecer um critério que garanta altura logarítmica ou falando de outra forma: que a árvore continue sendo completa ou que a complexidade continue proporcional a altura. *Adelson-Velskii* e *Landis* sugeriram um critério:

Uma árvore é dita balanceada se e somente se, para qualquer nó, a altura de suas duas subárvores diferem de no máximo uma unidade. As árvores que satisfazem esta condição são denominadas árvores AVL, em homenagem aos seus autores.

9.2. Mantendo-se AVL

Vimos que determinadas inserções podem desbalancear uma árvore, consequentemente deixando de ser uma árvore AVL e perdendo todas as vantagens já comentadas. Podemos ter o mesmo entendimento para as operações de remoção, já que causam o mesmo efeito.

Agora estudaremos uma solução para manter uma árvore balanceada. Trata-se de operações de rotações sempre que houver necessidade, após uma inserção ou remoção.

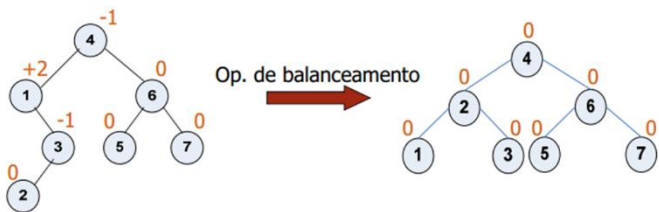
FATOR DE BALANCEAMENTO

O primeiro passo é verificar se a árvore está balanceada. Para tanto devemos calcular o fator de balanceamento (FB) para cada um dos seus nós. Considerando um determinado nó *a* de uma árvore, o seu fator de balanceamento é calculado assim:

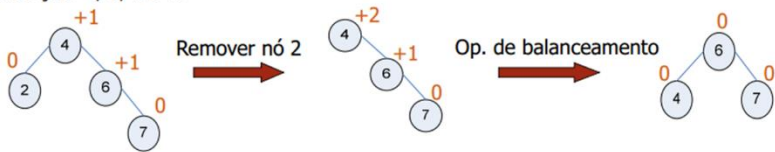
$$FB(a) = \text{Altura}(\text{sad}(a)) - \text{Altura}(\text{sae}(a))$$

Para que uma árvore seja considerada AVL, todos os seus nós devem ter um dos seguintes valores de FB: -1, 0, 1. Dizendo de outra forma: uma árvore é considerada AVL se todos os nós apresentam uma diferença máxima entre as sub-árvores esquerda e direita de no máximo 1 unidade. Veja alguns exemplos após a inserção e remoção.

Inserção: 4, 6, 1, 7, 5, 3 e 2.



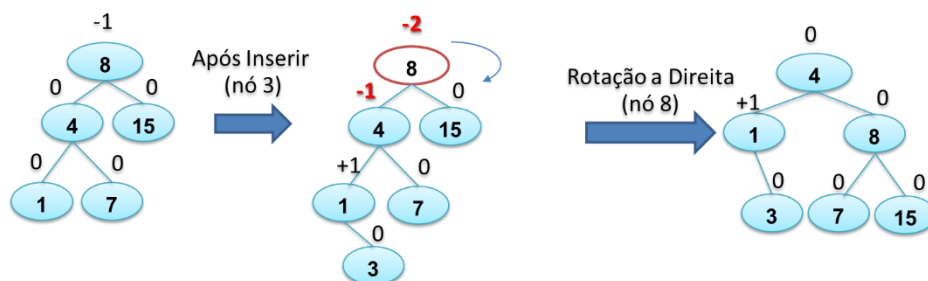
Inserção: 4, 6, 2 e 7.



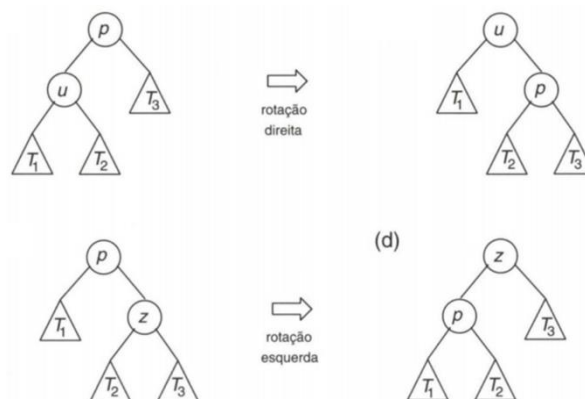
ROTAÇÃO SIMPLES

O segundo passo é realizar operações de rotação, com alguns nós da árvore, a fim de mantê-la balanceada. Veremos inicialmente a rotação simples, que possui as seguintes regras:

- A rotação é realizada no nó desregulado.
- Como os sinais dos FBs são os mesmos (nó 8 com $FB = -2$ e nó 4 com $FB = -1$) significa que precisamos fazer apenas uma **ROTAÇÃO SIMPLES**.
- O sinal do FB determina a direção da rotação: negativo indica rotação à direita e positivo na direção oposta.



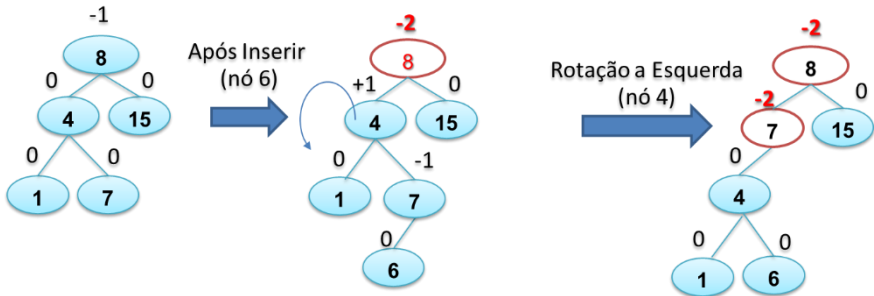
Considerando T_1 , T_2 , T_3 e T_4 árvores vazias ou não e p o nó desregulado, podemos generalizar:



ROTAÇÃO DUPLA

A rotação dupla será necessária apenas quando os sinais dos FBs, do nó desregulado e seu filho, são opostos. As seguintes regras devem ser obedecidas:

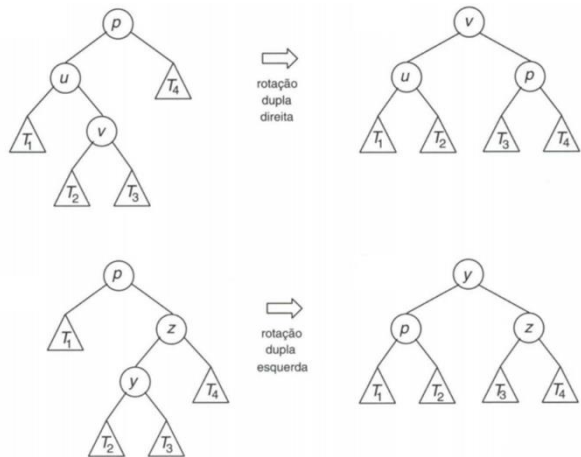
a) 1º rotação do nó filho (4) de acordo com as regras da rotação simples



b) 2º rotação do nó pai (8) na direção oposta.



Considerando T_1 , T_2 , T_3 e T_4 árvores vazias ou não e p o nó desregulado, podemos generalizar:



9.3. Operações

Veremos a seguir os principais algoritmos de manipulação de árvore AVL. Consideraremos apenas as representações por encadeamento. Todos os algoritmos que analisaremos terão como base a definição abaixo.

É aconselhável manter em cada nó um campo extra para registrar a altura da árvore ali enraizada. Na verdade, apenas a diferença de altura entre a subárvore esquerda e direita precisa ser mantida.

Definição da árvore AVL

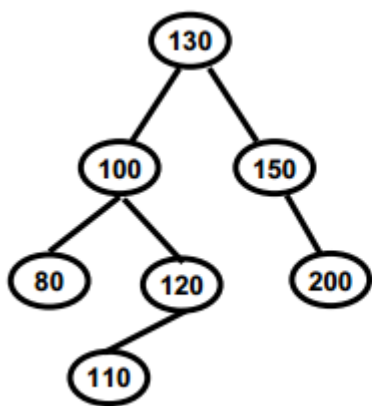
```
struct TipoArvore {  
    int info;  
    struct TipoArvore* sae;  
    struct TipoArvore* sad;  
    int altura;  
};  
typedef struct TipoArvore Arvore;
```

Os algoritmos de manipulação de uma árvore AVL são os mesmos para árvore binária de busca, porém devemos considerar:

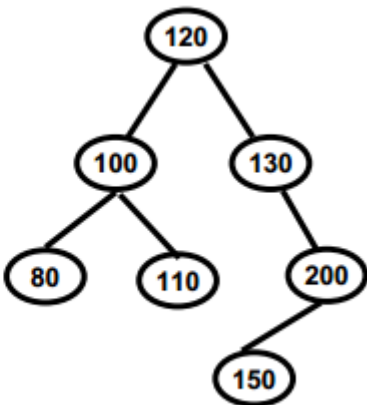
- a) Ao criar um novo nó, devemos atualizar a sua altura.
- b) Para atualizar a altura de um nó devemos calcular o fator de balanceamento.
- c) Realizar as diversas rotações tanto em inclusões quanto em remoções.

9.4. Atividades

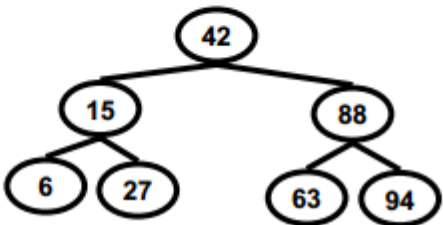
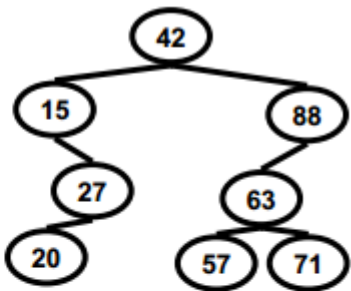
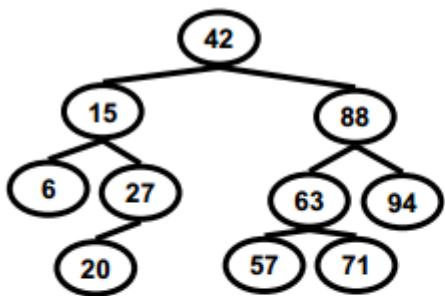
1. Analise as árvores abaixo e identifique as árvores AVL. Indique o fator de balanceamento de cada nó para todas as árvores.



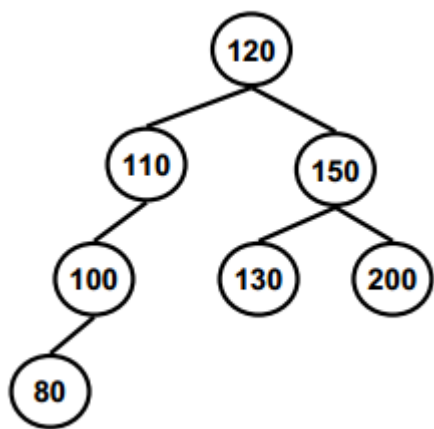
Árvore a



Árvore b

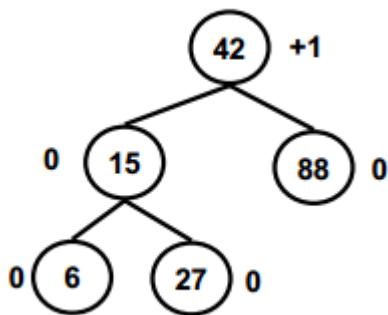


2. Indique as operações necessárias para tornar a árvore abaixo uma AVL.

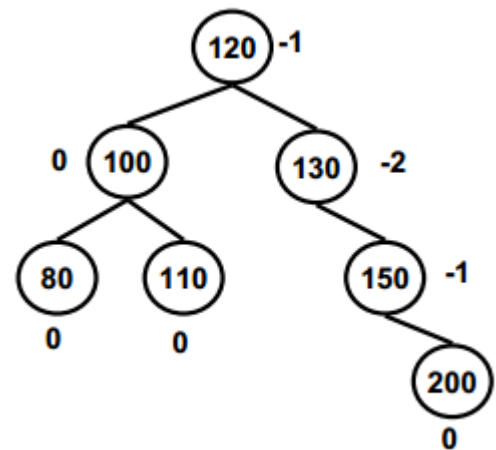


Consertar o FB dos exercícios abaixo.

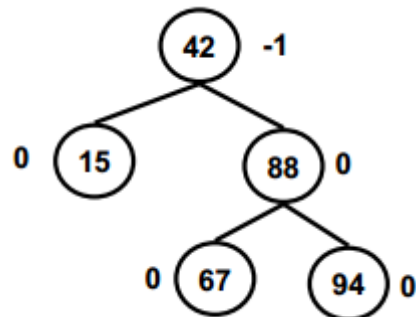
3. Desenhe a árvore abaixo após a inserção do nó 4. Em seguida mostre as operações necessárias para realizar o balanceamento da árvore.



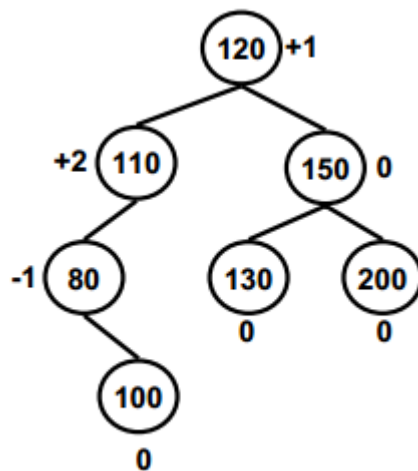
4. Indique as operações necessárias para tornar a árvore ao lado uma AVL.



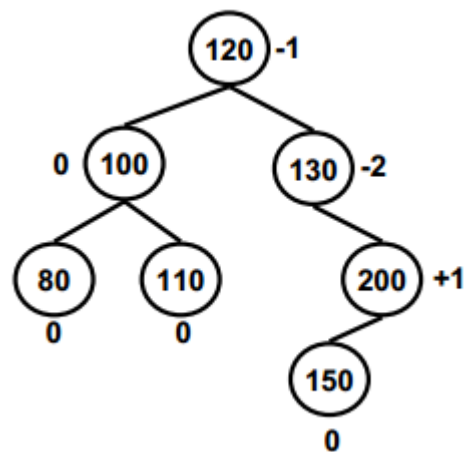
5. Desenhe a árvore abaixo após a inserção do nó 90.
Em seguida mostre as operações necessárias para realizar o balanceamento da árvore.



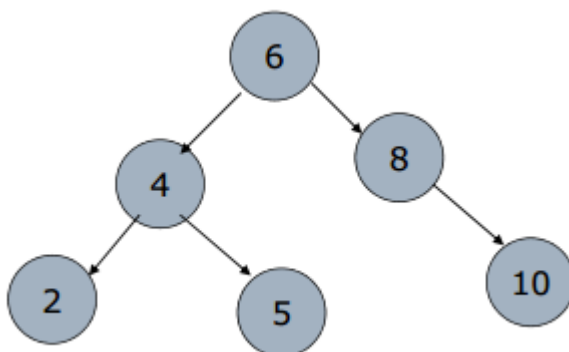
6. Indique as operações necessárias para realizar o balanceamento da árvore abaixo.



7. Indique as operações necessárias para realizar o balanceamento da árvore ao lado.



8. Inserir os nós 15, 7 e 28 na árvore abaixo. Em seguida remova os nós 4 e 8. Desenhe a árvore para cada operação de inserção e remoção e indique as rotações realizadas.



10. Árvore B

Existem situações onde o volume de informação é maior do que a capacidade de armazenamento da memória primária. Desta forma, inevitavelmente, haverá diversos acessos às informações armazenadas na memória secundária.

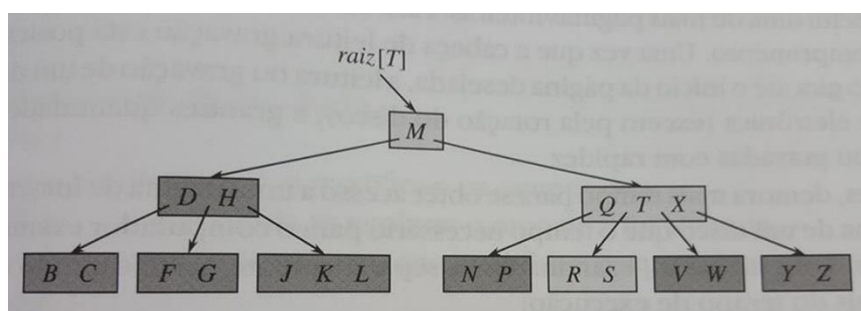
A árvore B é uma estrutura de dados sugerida por Rudolf Bayer e Edward Meyers (1971) como solução para otimizar o acesso a grande volume de informação em memória secundária. Por isso muitos SGBD (sistemas gerenciadores de banco de dados) utilizam árvore B.

Foi projetada de forma que apenas algumas partes do conjunto de dados armazenados em memória secundária, chamadas de **páginas**, possam ser carregadas para a memória primária para o devido processamento. O objetivo dessa estratégia é reduzir os acessos à memória secundária, maximizando o número de filhos de um nó, que podem chegar aos milhares. O importante é que a Árvore B garante bom desempenho, permite inserção, remoção e busca de chaves numa complexidade de tempo logarítmica.

Na realidade a árvore B é uma generalização da árvore binária de busca pois mantém sua propriedade fundamental: valores menores à esquerda da chave e valores maiores à direita da chave (ou vice-versa). Destacamos abaixo apenas as propriedades adicionais:

- a) Cada nó pode armazenar mais de uma chave.
- b) Termo mais usual para nó é página. Cada acesso ao disco carrega uma página na árvore.
- c) Cada página pode ter mais de 2 filhos
- d) Toda página deve ter no mínimo 50% de ocupação máxima. Exceto a raiz. Evita o desequilíbrio.
- e) Todas as folhas devem estar no mesmo nível. O crescimento é para cima e não para baixo como na árvore binária.

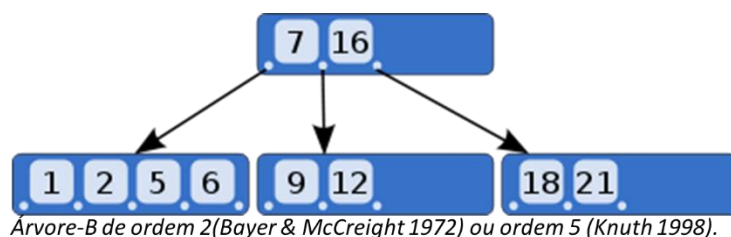
Veja na imagem abaixo um exemplo de árvore B onde as chaves são consoantes do alfabeto. As páginas levemente sombreadas são examinadas em uma pesquisa de busca a letra R.



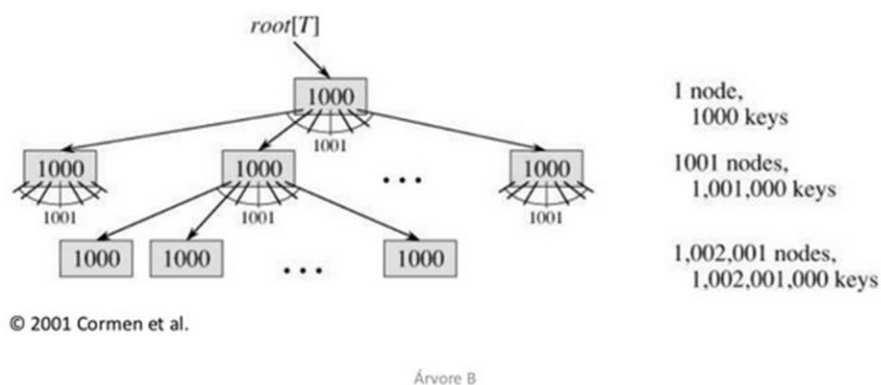
10.2. Ordem

Vimos que uma página é um conjunto de elementos que são agrupados e carregados para a memória primária para serem processados. Certamente que haverá um custo por esses acessos à memória secundária, entretanto buscamos otimizar ao máximo esse processo.

A ordem de uma árvore B tem relação com essa preocupação em otimizar o acesso à memória secundária. E há divergências entre autores ao definirem a ordem de um árvore B. **Para Cormen, Bayer e McCreight a ocupação mínima de cada página determina a ordem da árvore.** Já Knuth, a quantidade de filhos que uma página é que determina a ordem da árvore. Veja na imagem abaixo um exemplo dessa divergência.



A verdadeira aptidão de uma árvore B é armazenar grande volume de informação. Com apenas altura 2, o exemplo abaixo, armazena mais de 1 bilhão de chaves. Considerando que a raiz esteja sempre em memória primária, no máximo 2 acessos a disco são necessários para ler qualquer chave!

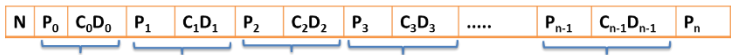


Resumindo as propriedades de uma árvore B de ordem n , segundo Cormen, Bayer e McCreight. Será a classificação que seguiremos daqui em diante.

- Cada página com no mínimo n chaves(50%) e $n+1$ filhos. Exceto a página raiz.
- Cada página com no máximo $2n$ chaves (100%) e $2n+1$ filhos.
- Páginas folhas no mesmo nível. Crescimento para cima.

10.3. Estrutura de uma página

A estrutura de uma página de uma árvore B poderia ser desta forma:

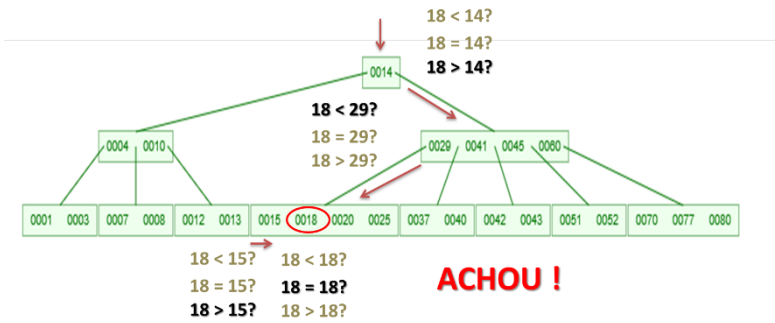


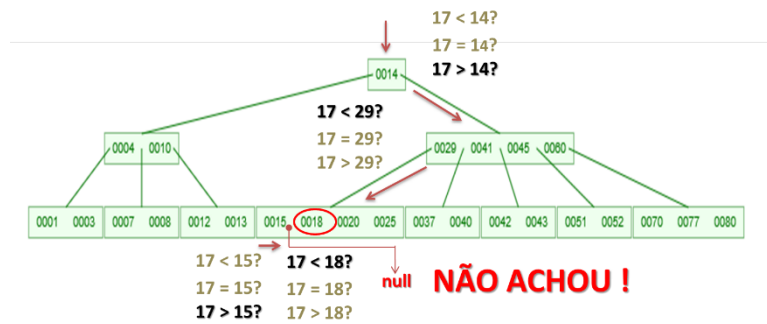
Onde,

- N** Número de elementos presentes na página. Auxilia na pesquisa por chaves. Nunca será maior do que o tamanho da página. Exemplo: N será no máximo 1000 para uma árvore de ordem 500.
- C_i** chave do registro
- D_i** dados vinculados à chave.
- P_i** Ponteiro para o i-ésimo filho. Aponta para a próxima página. Se for uma folha seu conteúdo será null.

10.4. Busca

A operação de busca em árvore B é semelhante a árvore binária de busca. A diferença está no fato de que devemos testar mais de uma chave por página. Veja abaixo os dois resultados possíveis em uma operação de busca.





10.5. Inserção

A inserção e árvore B pode ser resumida através do seguinte algoritmo genérico:

Algoritmos de busca em árvore B

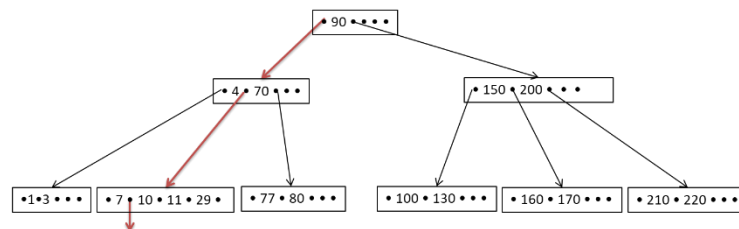
```

busca chave
se (chave já existe) então
    nada faz
senão
    se (chave couber na página) então
        incluir na página de forma ordenada.
    senão
        dividir a página em duas
        chave do meio deve ser promovida
fim-se
fim-se
  
```

Vamos analisar o algoritmo mais detalhadamente através de um exemplo de inserção da chave de valor 9 em uma árvore de ordem 2. Veremos a seguir as principais ações indicadas pelo algoritmo:

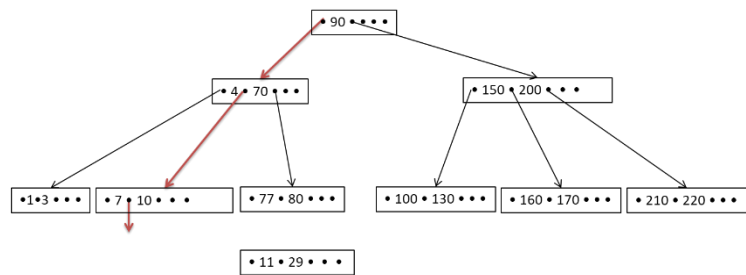
Primeira ação: busca pela chave a ser inserida

Caso a chave já exista, nada deve ser feito. Caso contrário, continuaremos o processo de inserção. Na imagem acima percebe-se uma indicação do local exato onde a chave deverá ser inserida.



Segunda ação: dividir a página em duas

Veja que a página onde será inserida a chave não possui espaço livre, já que é de ordem 2 e sua capacidade máxima é 4 elementos. Se a página estivesse com espaço livre, bastaria inserir na posição correta. Como a página está cheia realizaremos o crescimento da árvore para cima ao dividirmos a página em duas e promovermos a chave do meio para o nível acima. Será necessário criar uma página vazia e copiar metade da página atual para a nova, veja abaixo.



Terceira ação: Inserindo e promovendo a chave do meio

Na imagem abaixo percebemos a inserção da chave em sua posição correta. Em seguida verifica-se a promoção da chave do meio (10) para um nível acima, ela será responsável por apontar para a nova página que estava isolada da árvore.

Mas como identificar a chave a ser promovida? Se a chave foi inserida na esquerda (página existente) promover a maior chave para o nível acima. Se a chave foi inserida na direita (nova página) promover a menor chave para o nível acima.



E se a chave a ser promovida não couber na página do nível acima? Para essa página do nível acima, repetir o processo de dividir a página em duas e promover a chave do meio. Se for necessário propaga-se esse procedimento até chegar na raiz.

E se página raiz estiver cheia? Proceda da mesma forma: divide a página raiz em duas e promova a chave do meio. É nessa hora que a árvore B cresce para cima, aumentando sua altura e criando uma nova raiz.

Atenção: Acesse um simulador em <https://www.cs.usfca.edu/~galles/visualization/BTree.html>. Para árvores B de ordem 1 e 2, tente antecipar as mudanças realizadas na árvore antes de confirmar cada inserção.