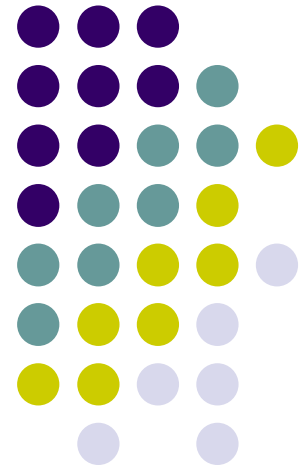


Estruturas de Dados

Marcelo B. Tenorio



Bibliografia



PEREIRA, Silvio do Lago. Estruturas de Dados Fundamentais. Conceitos e Aplicações. 11ª Edição. São Paulo: Érica, 2008.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. Algoritmos: teoria e prática. Rio de Janeiro: Elsevier, 2002.

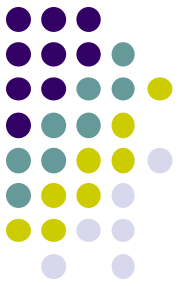
ZIVIANI, Nivio. Projeto de algoritmos : com implementações em Pascal e C. 2ª edição. São Paulo: Thomson Learning, 2007.

PREISS, Bruno R. Estrutura de dados e algoritmos: padrões de projetos orientados a objeto com Java.

FORBELONE, André Luiz Villar. Lógica de Programação: a construção de algoritmos e estruturas de dados. São Paulo: Makron Books, 1993.

KOFFMAN, Elliot B. Objetos, Abstração, Estrutura de Dados e Projeto Usando o C++. Rio de Janeiro: LTC, 2006.

Conteúdo



- Introdução a IDE
- Recursividade
- Lista linear sequencial
- Lista linear encadeada
- Lista não linear
- Hash
- Ordenação, Busca, Grafos e Complexidade

Introdução a IDE



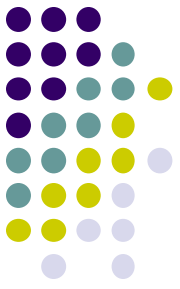
- Instalar o VS Code
 - <https://code.visualstudio.com>
- Instalar o .NET SDK 6.0
 - O .NET SDK já inclui o .NET Runtime
 - <https://dotnet.microsoft.com/en-us/download>
- No VS Code, instalar a extensão C#

Criando um projeto



- No SO
 - Criar uma pasta para o projeto
- No VS Code
 - Na opção Abrir Pasta, selecione a pasta criada
 - Na opção Terminal, escolha Novo Terminal
 - Para criar o projeto, digite `dotnet new console`
 - Edite o arquivo `Program.cs`
 - Para executar, digite `dotnet run`

Abrindo um projeto



- No VS Code
 - Na opção Abrir Pasta, selecione a pasta do projeto
 - Edite o arquivo Program.cs
 - Na opção Terminal, escolha Novo Terminal
 - Para executar, digite dotnet run

Comandos básicos



Conceitos	Python	C#
Tipo primitivo de dado	int, float	int, float, char, string, bool
Variável e Constante (homogênea e não indexada)	Não se declara variável, o tipo é definido numa atribuição ou entrada. Não existe constante.	int x, y; float z; bool a; const int X = 10;
Operador e Expressão	Aritmético (+, -, *, /, %) Relacional (>, >=, !=, ==) Lógico (and, or, not) Atribuição (=) z = x / y a = x > y and z == 0.5	Aritmético (+, -, *, /, %) Relacional (>, >=, !=, ==) Lógico (&&, , !) Atribuição (=) z = x / y; a = x > y && z == 0.5;
Entrada e Saída	n = input() print("Nome", n)	n = Console.ReadLine(); Console.WriteLine("Nome"+n);

Comandos básicos



Conceitos	Python	C#
Decisão e Repetição	<pre>if x > y: print("x maior que y") elif x == y: print("x igual a y") else: print("x menor que y") print("outro comando") for i in range(1,6): print(i) i = 1 while i <= 5: print(i) i = i + 1</pre>	<pre>if (x > y) Console.WriteLine("x maior que y"); else if (x == y) Console.WriteLine("x igual a y"); else { Console.WriteLine("x menor que y"); Console.WriteLine("outro comando"); } for (i = 1; i <= 5; i = i + 1) Console.WriteLine(i); i = 1; while (i <= 5) { Console.WriteLine(i); i = i + 1; }</pre>

Comandos básicos



Conceitos	Python	C#
Variável indexada (homogênea)	<pre>b = [] b.append(10) b.append(20)</pre>	<pre>int[] b = new int[5]; b[0] = 10; b[1] = 20;</pre>
Função	<pre>def soma(x, y): x = 30 y = 40 return x + y a = 10 b = 20 z = soma(a, b) print(a, b, z) Parâmetro imutável: tipo primitivo (int, float, ...) Parâmetro mutável: tipo abstrato (lista ou objeto)</pre>	<pre>int soma(int x, ref int y) { x = 30; y = 40; return x + y; } int a = 10, b = 20; z = soma(a, b); Console.WriteLine(a, b, z);</pre>

Comandos básicos

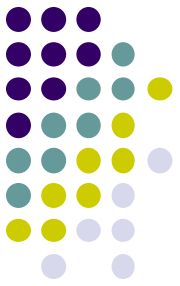


Conceitos	Python	C#
Tipo abstrato de dado (heterogêneo)	<pre>class Tipo: codigo = 0 nome = "" d = [] c = Tipo() c.codigo = 1 c.nome = "Enzo" d.append(c) c = Tipo() c.codigo = 2 c.nome = "Sofia" d.append(c)</pre>	<pre>class Tipo { int codigo; string nome; } Tipo[] d = new Tipo[5]; d[0] = new Tipo(); d[0].codigo = 1; d[0].nome = "Enzo"; d[1] = new Tipo(); d[1].codigo = 2; d[1].nome = "Sofia"; Tipo c = new Tipo(); c.codigo = 1; c.nome = "Patricia";</pre>

Introdução a Estrutura de Dados



- Estuda as organizações dos dados utilizadas pelo computador para controle de diversas atividades.
- Exemplos: Chamadas das funções, Escalonamento de processos, Criptografia, Localização dos dados, Fila de Impressão, etc.



Recursividade

- Dividir para conquistar
- A função chama ela mesma
- Divide o problema em subproblemas menores



Recursividade

- **Trivial:** dado por definição; isto é, **não necessita da recursão para ser obtida.**
- **Geral:** parte do problema que em essência é igual ao problema original, sendo porém menor.

Recursividade



- Exemplo: Fatorial

// Dada por definição


Trivial: $0! = 1$

// Requer aplicação da função para $(n-1)!$

Geral: $n! = n * (n-1)!$

Fatorial

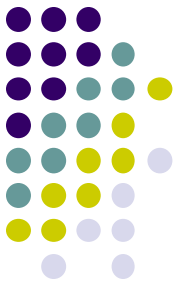
- Solução iterativa

int fat(int n) 

```
{  
    int f = 1;  
    while (n > 0)  
    {  
        f = f * n;  
        n = n - 1;  
    }  
    return f;  
}
```

- Solução recursiva 

```
int fat(int n)  
{  
    if (n == 0)  
        return 1;  
    else  
        return n * fat(n-1);  
}
```

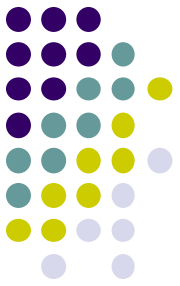


Recursividade - Ilustração



				n = 0 return 1
			n = 1 return 1 * fat(0)	n = 1 return 1 * fat(0)
		n = 2 return 2 * fat(1)	n = 2 return 2 * fat(1)	n = 2 return 2 * fat(1)
	n = 3 return 3 * fat(2)	n = 3 return 3 * fat(2)	n = 3 return 3 * fat(2)	n = 3 return 3 * fat(2)
n = 4 return 4 * fat(3)	n = 4 return 4 * fat(3)	n = 4 return 4 * fat(3)	n = 4 return 4 * fat(3)	n = 4 return 4 * fat(3)
fat(4)	fat(3)	fat(2)	fat(1)	fat(0)

Lista linear sequencial



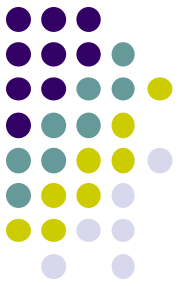
- Dados organizados de maneira sequencial
- Vetor
- Tipos
 - Pilha
 - Fila

Pilha



- Tipo de lista linear em que todas as operações de **inserção e remoção** são realizadas numa mesma extremidade, denominada **topo**.
- LIFO (Last-in / First-out)
- Ilustração

Pilha



- Aplicações
 - Análise de expressões e sintaxe
 - Compiladores
 - Calculadoras
 - Chamada das funções
 - Inversão de dados

Pilha



- Operações básicas

Inserir: insere um elemento no topo da pilha

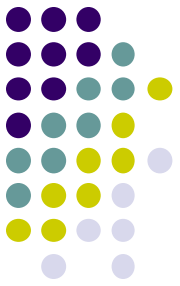
Remove: remove um elemento do topo da pilha

- Operações complementares

EstaVazia: verifica se a pilha está vazia

EstaCheia: verifica se a pilha está cheia

Implementação de Lista Linear Sequencial - Pilha



```
const int MAX = 20;
```

```
void Insere(char[] p, ref int t, char v)
{
    p[t] = v;
    t = t + 1;
}
```

```
char Remove(char[] p, ref int t)
{
    t = t - 1;
    return (p[t]);
}
```

Implementação de Lista Linear Sequencial - Pilha



```
bool EstaVazia(int t)
{
    if (t == 0)
        return true;
    else
        return false;
}
```

```
bool EstaCheia(int t)
{
    if (t == MAX)
        return true;
    else
        return false;
}
```

```
char[] pilha = new char[MAX];
int topo = 0;
```



Fila

- Tipo de lista linear em que as **inserções** são realizadas num extremo (**fim**), ficando as **remoções** restritas ao outro extremo (**início**).
- FIFO (First-In / First-Out)
- Ilustração



Fila

- Operações básicas

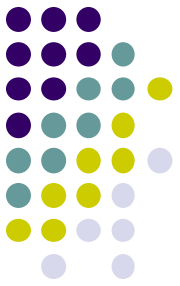
Inserir: insere um elemento no fim da fila

Remover: remove um elemento do início da fila

- Aplicações

- Escalonamento de processos do SO
- Controle de requisições SQL no BD
- Fila de impressão do SO

Implementação de Lista Linear Sequencial - Fila



```
const int MAX = 20;
```

```
void Insere(int[] q, ref int f, int v)
{
    q[f] = v;
    f = f + 1;
}
```

```
int Remove(int[] q, ref int i)
{
    int v = q[i];
    i = i + 1;
    return (v);
}
```

Implementação de Lista Linear Sequencial - Fila



```
bool EstaVazia(int i, int f)
{
    if (i == f)
        return true;
    else
        return false;
}
```

```
bool EstaCheia(int f)
{
    if (f == MAX)
        return true;
    else
        return false;
}
```

```
int[] fila = new int[MAX];
int inicio = 0, fim = 0;
```

Alocação estática de memória



- Vetor: Aloca-se a quantidade máxima de memória em tempo de programação. Esta alocação NÃO PODE ser alterada em tempo de execução do software.
- Aplica-se quando é conhecido, em tempo de programação, a quantidade máxima de memória necessária para solução do problema.
- Vantagem
 - Agilidade no acesso aos valores (índices)
- Desvantagem
 - Desperdício de memória

Alocação dinâmica de memória




- Ponteiro: Aloca-se a quantidade de memória necessária, conforme a execução do software. Esta alocação PODE ser alterada em tempo de execução.
- Aplica-se quando NÃO é conhecido, em tempo de programação, a quantidade máxima de memória necessária para solução do problema.
- Vantagem
 - Poupa memória
- Desvantagem
 - Não há acesso aos valores por índice (em linguagens de alto nível, o índice também existe, assim como em vetor)

Introdução a Ponteiros



Parâmetros - por referência (entrada/saída) e por valor (entrada)

```
void funcao(ref int px, int y2)
{
    Console.WriteLine(px+" "+y2);    // 10 20
    px = 30;
    y2 = 40;
    Console.WriteLine(px+" "+y2);    // 30 40
}
```

```
int x = 10, y = 20;
Console.WriteLine(x+" "+y);          // 10 20
funcao(ref  x, y);
Console.WriteLine(x+" "+y);          // 30 20
```

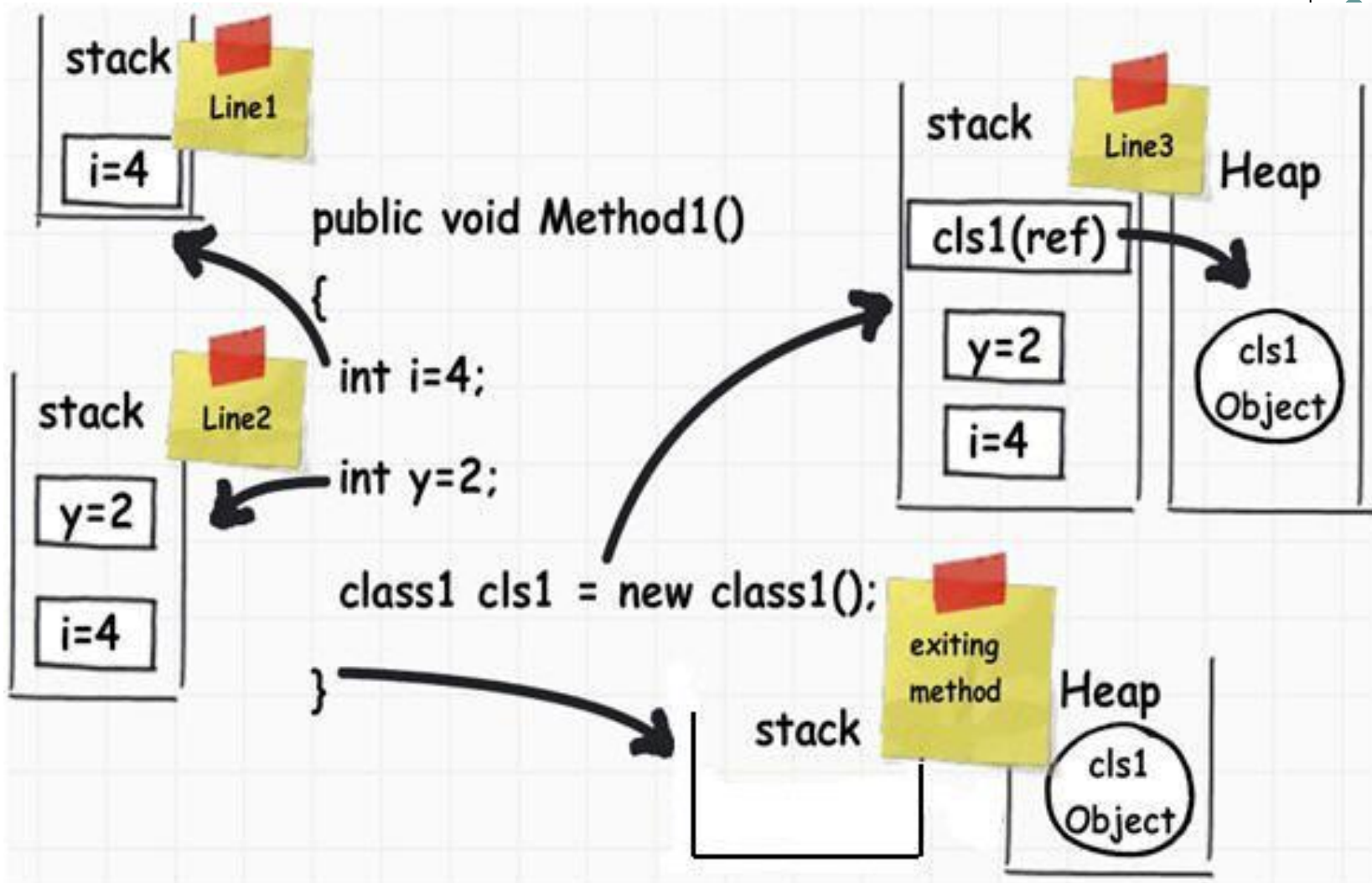
Memória



Ponteiro (referências / endereços) Valor Hexadecimal (base 16)

Endereço	...	3E8	3EB	3EC	3ED	...
Conteúdo	...	3EB	10	20	20	...
Nome	...	px	x	y	y2	...

Memória (stack e heap)



Lista linear encadeada



- Diferença entre lista sequencial e encadeada
- Ponteiro
- Ilustração

Implementação de Lista Linear Encadeada

- Pilha



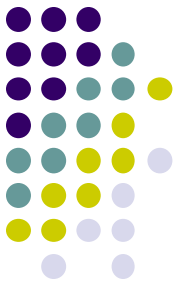
```
tp_no Remove(ref tp_no l)
{
    tp_no no = null;
    if (l != null)
    {
        no = l;
        l = l.prox;
        no.prox = null;
    }
    return no;
}
```

```
void Insere(ref tp_no l, int v)
{
    tp_no no = new tp_no();
    no.valor = v;
    if (l != null)
        no.prox = l;
    l = no;
}

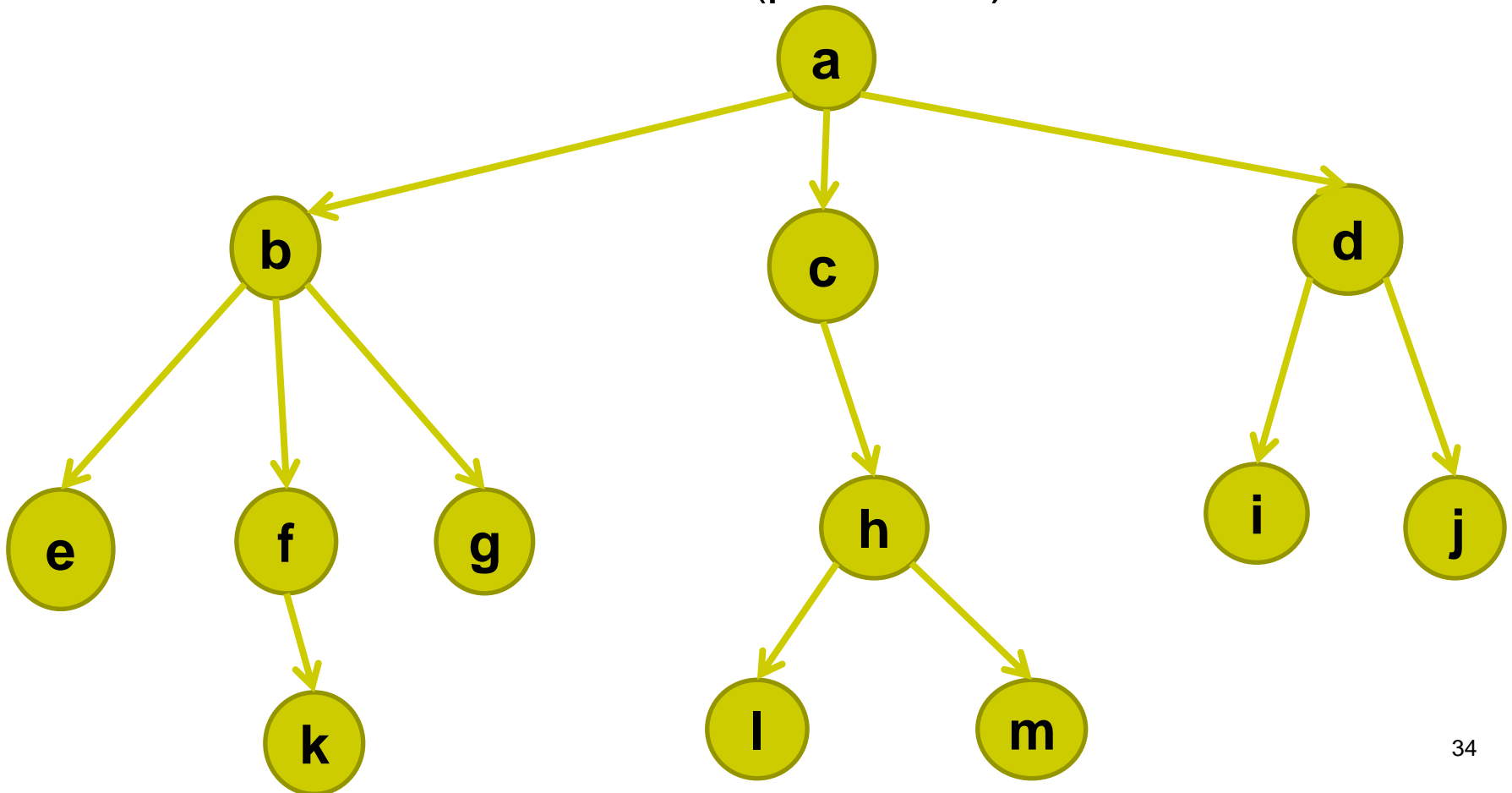
tp_no lista = null;

class tp_no
{
    public int valor;
    public tp_no prox;
}
```

Lista não linear (ou generalizada)



- Diferença entre lista linear e não linear
 - Exemplos (árvores e grafos)
 - Também são encadeadas (ponteiros)



Árvore

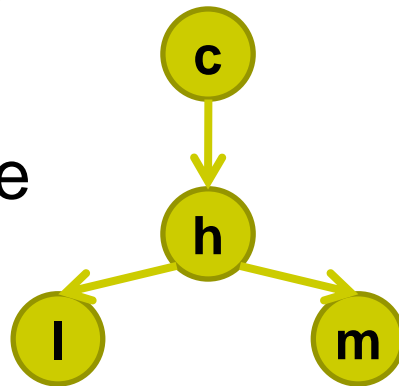


- Coleção finita de nós

- Características:

- Raiz 

- Subárvore



- Grau: quantidade de subárvores
 - O nó **a** tem grau 3.
 - O nó **d** tem grau 2.

Árvore



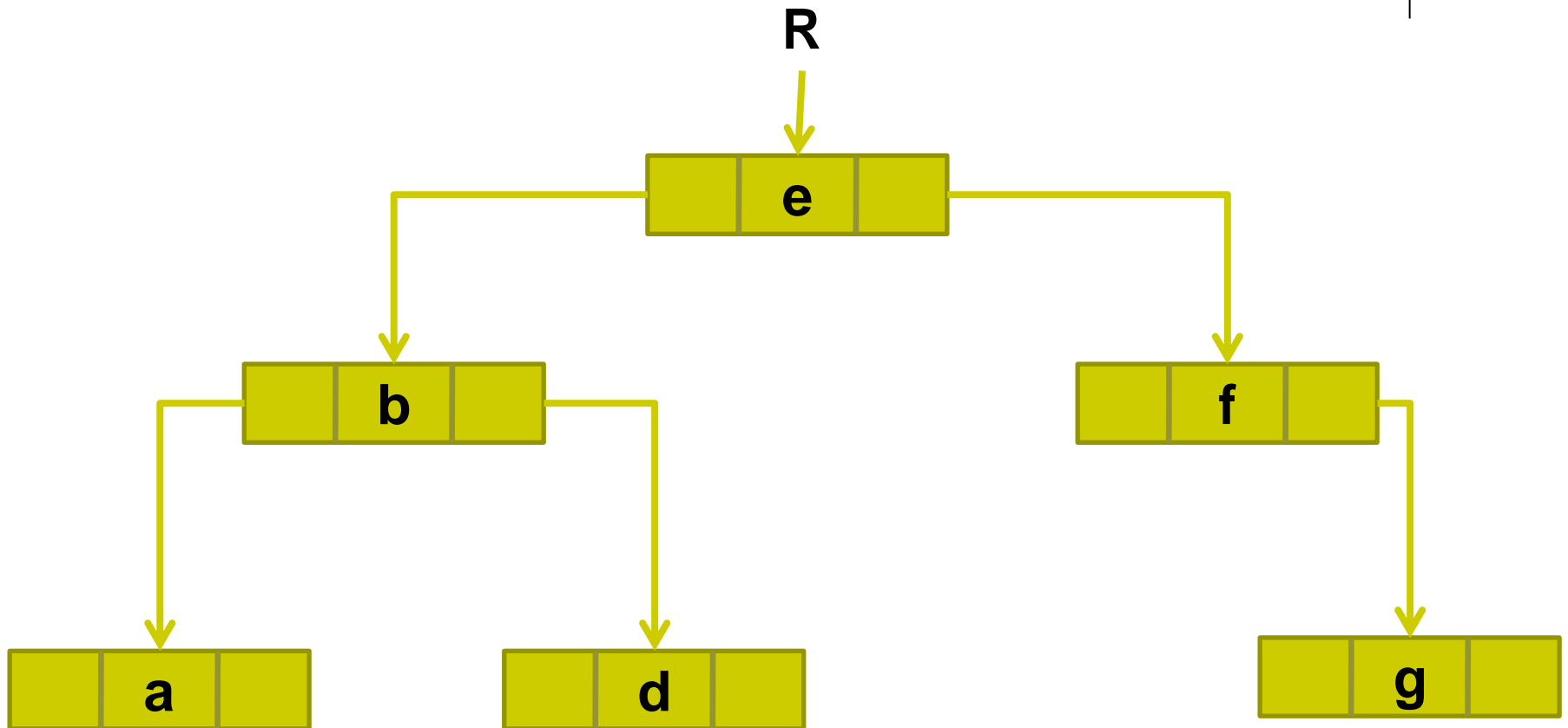
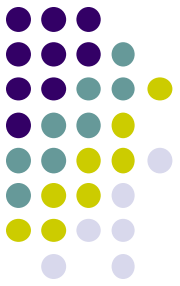
- Nó de grau 0 é uma folha.
- Nós pai e filhos.
 - Filhos do nó **b** são: **e**, **f** e **g**
 - O nó **h** é o pai de **l** e **m**
- Altura de uma árvore
 - O máximo dos níveis de todos os seus nós.
 - Altura desta árvore exemplo: 4

Árvores

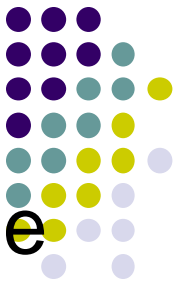


- Aplicações:
 - Jogos (análise de possibilidades), Dicionário de linguagem de programação, Geometria computacional, Conjuntos, Banco de dados, Sistemas de arquivos, etc
- Tipos
 - Binária: Cada nó com no máximo 2 filhos
 - AVL: Binária balanceada (busca mais rápida)
 - Rubro-Negra: Binária balanceada com alguns diferenciais
 - B: Cada nó com mais de 2 filhos

Árvore Binária

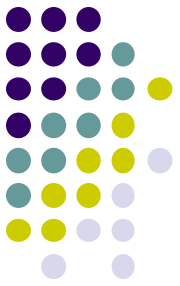


Árvore Binária



- Inserção: Valores menores para a esquerda e valores maiores para a direita
- Remoção
 - Nó sem filhos: Atribui nulo para o ponteiro do pai
 - Nó com 1 filho: Liga com o ponteiro do pai
 - Nó com 2 filhos: Iniciando a busca pela esquerda, substitua-o pelo nó de maior valor. Faça os ajustes necessários com o nó de maior valor (nó sem filhos ou nó com 1 filho)

Implementação de Árvore Binária



```
void Insere(ref tp_no r, int x)
{
    if (r == null)
    {
        r = new tp_no();
        r.valor = x;
    }
    else if (x < r.valor)
        Insere(ref r.esq, x);
    else
        Insere(ref r.dir, x);
}
```


Implementação de Árvore Binária



```
tp_no Busca(tp_no r, int x)
{
    if (r == null)
        return null;
    else if (x == r.valor)
        return r;
    else if (x < r.valor)
        return Busca(r.esq, x);
    else
        return Busca(r.dir, x);
}
```

Implementação de Árvore Binária



```
tp_no RetornaMaior(ref tp_no r)
{
    if (r.dir == null)
    {
        tp_no p = r;
        r = r.esq;
        return p;
    }
    else
        return RetornaMaior(ref r.dir);
}
```

```
tp_no Remove(ref tp_no r, int x)
```

```
{
```

```
    if (r == null)
```

```
        return null;
```

```
    else if (x == r.valor)
```

```
    {
```

```
        tp_no p = r;
```

```
        if (r.esq == null)      // nao tem filho esquerdo
```

```
            r = r.dir;
```

```
        else if (r.dir == null) // nao tem filho direito
```

```
            r = r.esq;
```

```
        else                    // tem ambos os filhos
```

```
    {
```

```
        p = RetornaMaior(ref r.esq);
```

```
        r.valor = p.valor;
```

```
    }
```

```
    return p;
```

```
}
```

```
else if (x < r.valor)
```

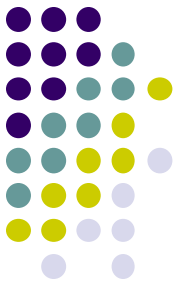
```
    return Remove(ref r.esq, x);
```

```
else
```

```
    return Remove(ref r.dir, x);
```

```
}
```

Implementação de Árvore Binária



Implementação de Árvore Binária



```
void EmOrdem(tp_no r)
{
    if (r != null)
    {
        EmOrdem(r.esq);
        Console.WriteLine(r.valor);
        EmOrdem(r.dir);
    }
}
```

```
void PreOrdem(tp_no r)
{
    if (r != null)
    {
        Console.WriteLine(r.valor);
        PreOrdem(r.esq);
        PreOrdem(r.dir);
    }
}
```

Implementação de Árvore Binária



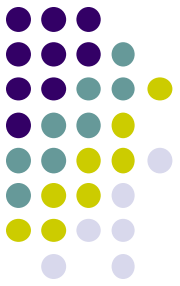
```
void PosOrdem(tp_no r)
{
    if (r != null)
    {
        PosOrdem(r.esq);
        PosOrdem(r.dir);
        Console.WriteLine(r.valor);
    }
}
```

```
class tp_no
{
    public tp_no esq;
    public int valor;
    public tp_no dir;
}

tp_no raiz = null;
```

Hash

(dispersão ou espalhamento)



- Principal característica:
 - Maior agilidade na recuperação do dado.
- Há uma função que mapeia o valor a ser armazenado e a posição de armazenamento.
 - Função hash

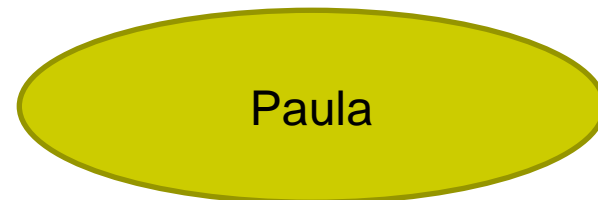
Hash – Redução do espaço de busca



$k = \text{'Maria'}$



$k = \text{'Maria'}$
 $f(k) = 2$



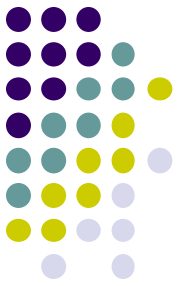


Tabela Hash

- Funções
 - Resto da divisão inteira (ilustração)
 - Meio do Quadrado
 - Método da Dobra
 - Método da Multiplicação
 - Hashing Universal

Implementação de Hash



```
const int N = 5;
```

```
int Hash(int chave)
{
    return (chave % N);
}
```

```
void Insere(int[] v, int c)
{
    int pos = Hash(c);
    v[pos] = c;
}
```

```
int Busca(int c)
{
    int pos = Hash(c);
    return pos;
}
```

```
int[] vetor = new int[N];
```



Hash - Colisão

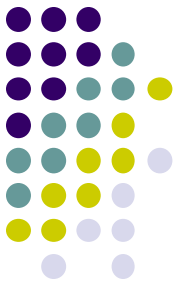
- A colisão ocorre quando uma chave deve ser armazenada numa posição já ocupada.
- Métodos para tratamento
 - Endereçamento Aberto
 - Tratamento linear
 - Endereçamento Fechado
 - Lista encadeada

Implementação de Hash



```
void InsereLinear(int[] v, int c)
{
    int pos = Hash(c);
    while (v[pos] != 0)
    {
        pos++;
        pos = pos % N;
    }
    v[pos] = c;
}
```

Implementação de Hash



```
void InsereEncadeado(tp_no[] v, int c)
{
    tp_no no = new tp_no()
    no.chave = c;
    int pos = Hash(c);
    if (v[pos] != null)
        no.prox = v[pos];
    v[pos] = no;
}
```

```
tp_no[] vetor = new tp_no[N];
```

```
class tp_no
{
    public int chave;
    public tp_no prox;
}
```

Ordenação



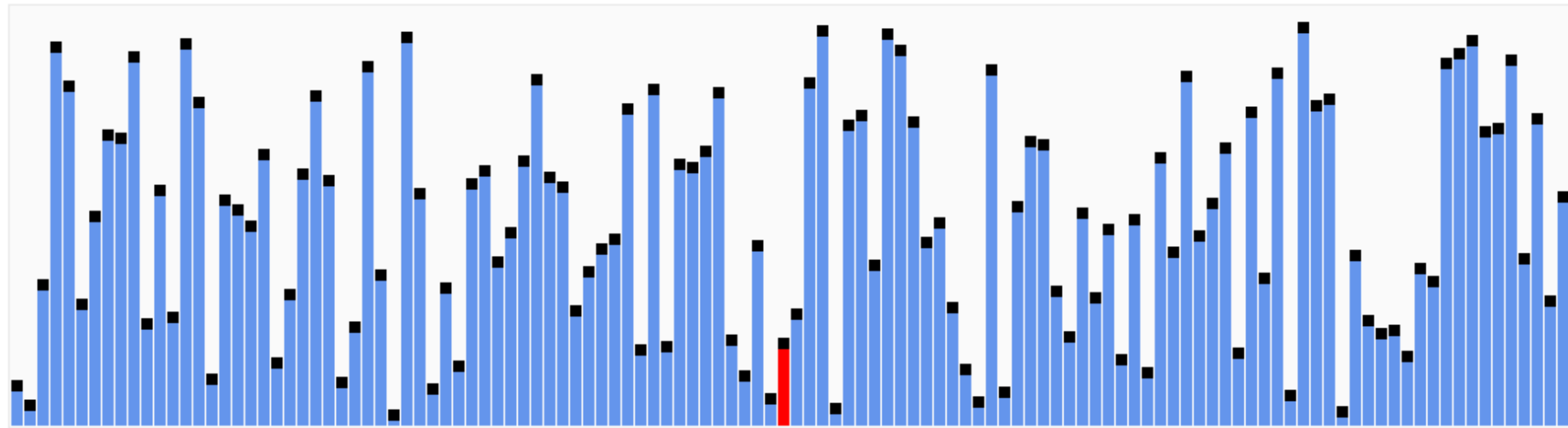
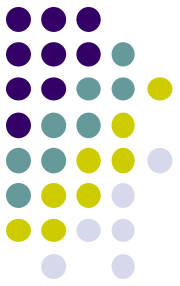
- Existem vários algoritmos de ordenação
 - O mais rápido é o Quicksort
- A ordenação dos dados é essencial quando algo é procurado (busca)
- Em bancos de dados, por exemplo, utiliza-se algoritmo de ordenação

Ordenação - Algoritmo Quicksort



```
void quick_sort(int[] a, int left, int right) {  
    int i, j, x, y;  
    i = left;  
    j = right;  
    x = a[(left + right) / 2];  
    while(i <= j) {  
        while(a[i] < x && i < right) i+=1;  
        while(a[j] > x && j > left) j-=1;  
        if(i <= j) {  
            y = a[i];  
            a[i] = a[j];  
            a[j] = y;  
            i+=1;  
            j-=1;  
        }  
    }  
    if(j > left) quick_sort(a, left, j);  
    if(i < right) quick_sort(a, i, right);  
}
```

Algoritmo Quicksort - Ilustração



Busca



- Existem dois tipos
- Busca Linear (sequencial / exaustiva)
 - Os dados podem estar desordenados
 - Lenta
- Busca Binária
 - Os dados precisam estar ordenados
 - Rápida

Busca Linear - Algoritmo



```
// A variável qtd significa a quantidade de valores no vetor  
// A variável chave é o valor procurado
```

```
int i = 0;  
while (i < qtd && chave != vetor[i])  
    i = i + 1;  
  
if (i < qtd)  
    exhibe "Encontrou na posição: " + i;  
else  
    exhibe "Não encontrou";
```

Busca Binária - Algoritmo

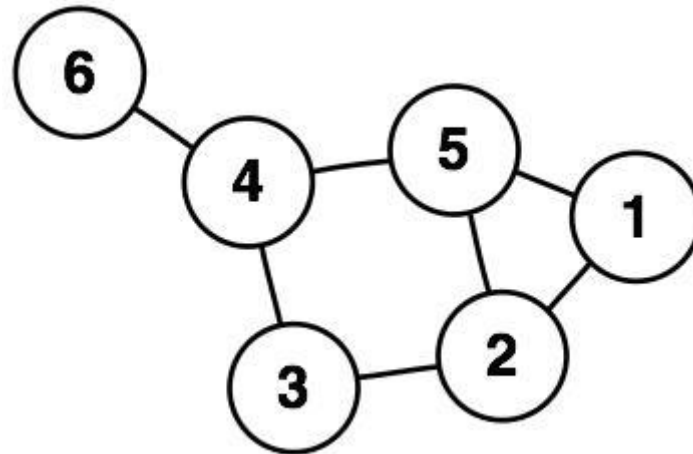


```
int BuscaBinaria(int[] vetor, int chave)
{
    int meio, Min = 0, Max = vetor.Length - 1;
    do
    {
        meio = (int)(Min + Max) / 2;
        if (vetor[meio] == chave)
            return meio;        // Retorna a posição que encontrou
        else if (chave > vetor[meio])
            Min = meio + 1;
        else
            Max = meio - 1;
    }
    while (Min <= Max);
    return -1;        // Retorno -1 significa que não encontrou
}
```

Grafos



- Conjunto de elementos interligados (não linear)



- Os algoritmos de grafos são utilizados quando o problema envolve caminhos



Grafos

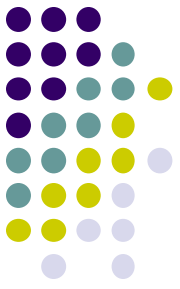
- Aplicações
 - Logística de transporte, Movimentação em jogos digitais, Redes sociais (seguidores), etc.
- Algoritmos conhecidos:
 - Dijkstra
 - Kruskal

Complexidade de Algoritmo



- Utilizada para analisar a eficiência do algoritmo
- Observa-se a quantidade de operações relevantes, por exemplo, comandos de decisão (if) e repetição (for ou while)
- Geralmente, analisa-se o pior caso
- Notação O

Notação O Grande



Notação	Nome	Exemplo
$O(1)$	Constante	Determinar se um número é par ou ímpar
$O(\log n)$	Logarítmico	Encontrar um item numa matriz ordenada
$O(n)$	Linear	Encontrar um item numa lista desordenada
$O(n^2)$	Quadrático	Quicksort
$O(2^n)$	Exponencial	Caixeiro viajante (programação dinâmica)
$O(n!)$	Fatorial	Caixeiro viajante (força bruta)