MAC0323 Algoritmos e Estruturas de Dados II

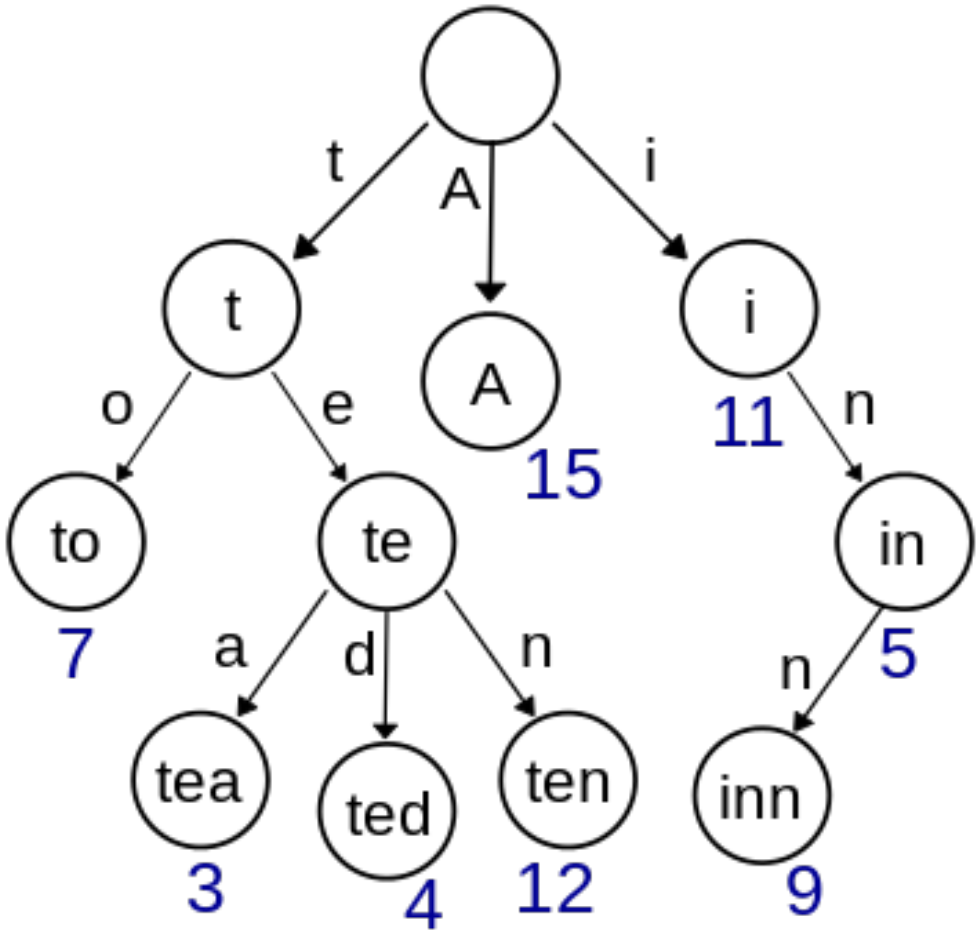
Edição 2020 – 2

Fonte: ash.atozviews.com

Compacto dos melhores momentos

AULA 17

Tries

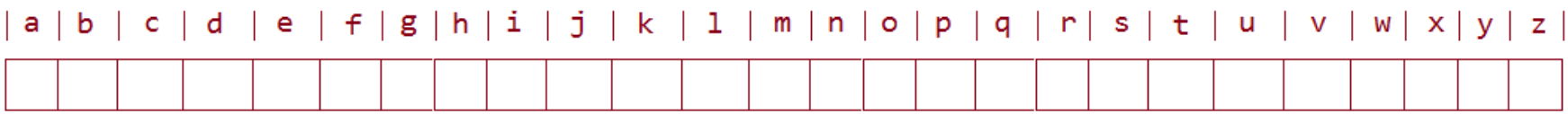


Fonte: Wikipedia

Estrutura do nó de uma trie

Os links correspondem a caracteres.
Tries são compostas por nós do tipo struct node.

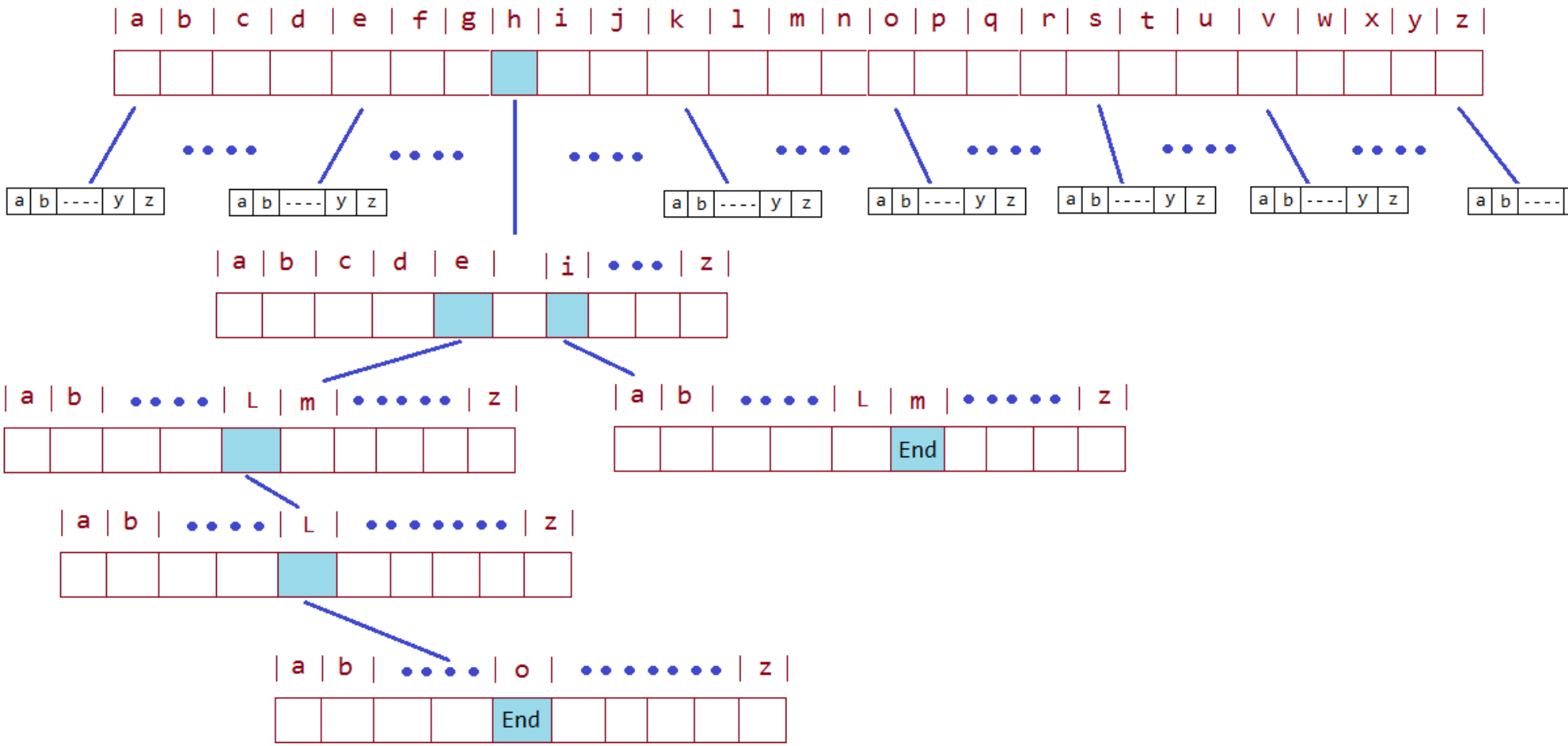
```
typedef struct node *Node;
struct node {
   Value val;
   Node next[R];
}
```



Fonte: JavaByPatel

Muitos dos R ponteiros podem ser NULL.

Trie para "hello" e "him"



Fonte: JavaByPatel

AULA 18

Tries ternárias

Fonte: The Little Prince, Antoine de Saint-Exupéry

Referências: Tries (árvores digitais) (PF); Tries (S&W); slides (S&W); Vídeo (S&W); TAOCP, vol 3, cap. 6.3

Tries ternárias

O maior problema das `tries` é possivelmente o espaço, já que cada nó contém `R` referências.

Assim, cada nó utiliza pelo menos $8 \times$ `R` bytes.

Veja a seguir alguns valores de `R` para alguns alfabetos comuns em aplicações.

Para evitar o custo excessivo de espaço de uma `R-trie`, consideramos uma representação como uma **ternary search trie** (`TST`).

Alphabet

nome	R()	lg(R)	conjunto de caracteres
BINARY	2	1	'01'
DNA	4	2	'ACTG'
OCTAL	8	3	'01234567'
DECIMAL	10	4	'0123456789'
HEXADECIMAL	16	4	'0123456789ABCDEF'
PROTEIN	20	5	'ACDEFGHIKLMNPQRSTVWY'
LOWERCASE	26	5	'abcd...wxyz'
UPPERCASE	26	5	'ABCD...WXYZ'
ASCII	128	7	alfabeto ASCII
EXTENDED_ASCII	256	8	alfabeto ASCII estendido
UNICODE16	65536	16	alfabeto Unicode

Ilustração

Para os pares `key`-`val`

`key`	`val`
are	12
by	4
sea	2
sells	1
she	0
shells	3
the	5
shore	6

temos a `TST` a seguir.

key	val
are	12
by	4
sea	2
sells	1
she	0
shells	3
the	5
shore	6

TSTs

De maneira semelhante ao que ocorre com `tries`, nas `tries` ternárias:

- chaves ficam codificadas nos caminhos que começam na raiz;
- prefixos de chaves, que nem sempre são chaves, estão representados na `TST`.

Estrutura de uma trie ternária

Os links da estrutura correspondem a caracteres.
Nas figuras, o caractere escrito dentro de um nó
é o caractere do link que sai pelo meio do nó.
TSTs são compostas por nós do tipo `struct node`.

```
typedef struct node *Node;

struct node {
   char c;      /* caracter */
   Value val;
   Node left;   /*  < c */
   Node mid;    /* == c */
   Node right;  /*  > c */
};
```

Ilustração

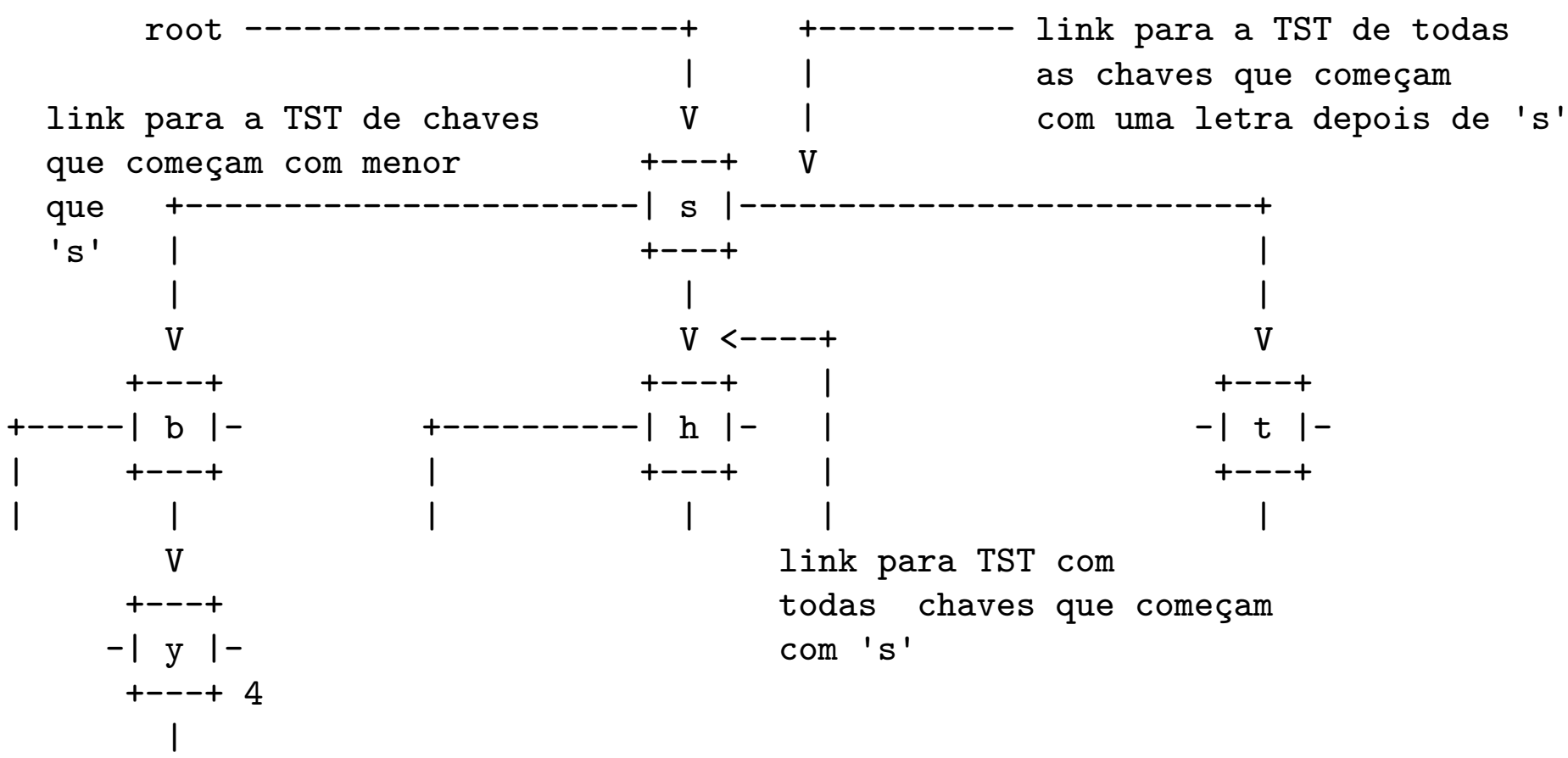
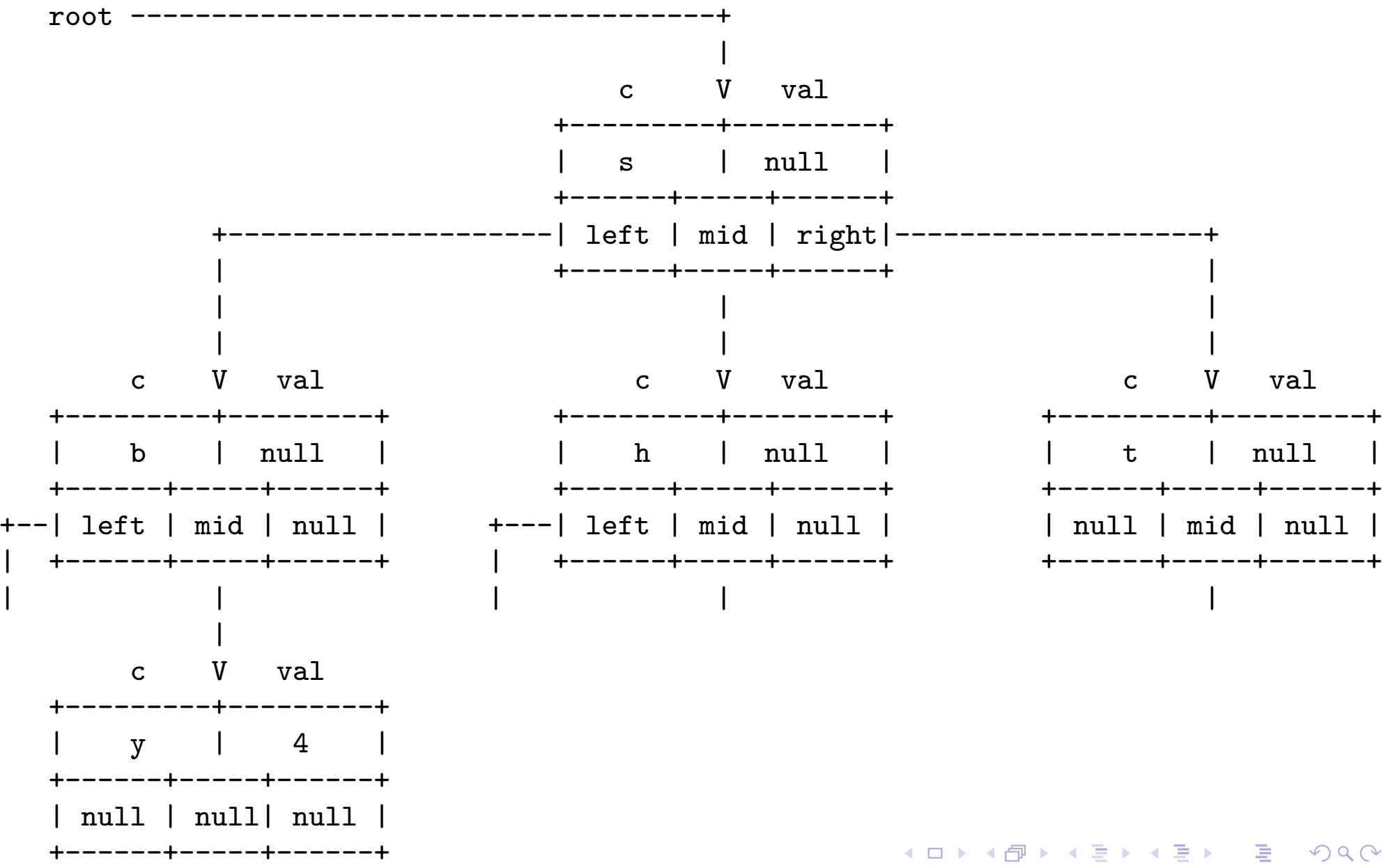
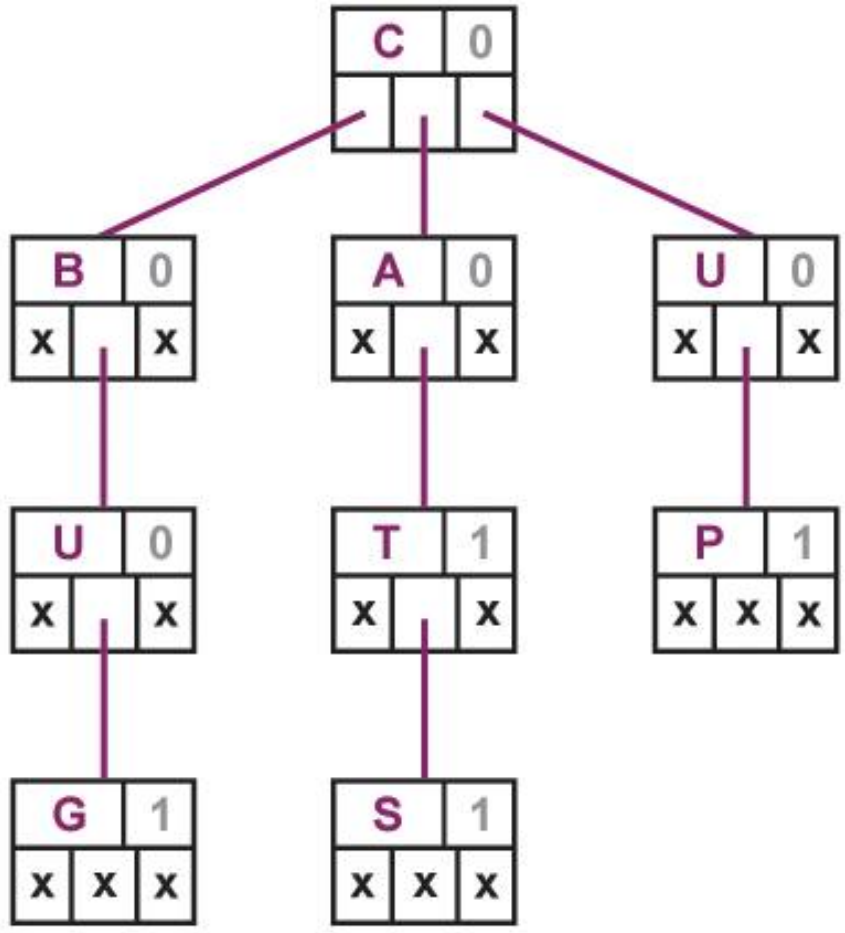

root
link para a TST de todas as chaves que começam com uma letra depois de 's'
link para a TST de chaves que começam com menor que 's'
s
b
h
t
y
4
link para TST com todas chaves que começam com 's'

Ilustração com zoom



Outra ilustração



Ternary Search Tree for CAT, BUG, CATS, UP

Fonte: Ternary search trees for autocompletion and spell checking

TST.c: esqueleto

```
static int R = 128;                /* tamanho do alfabeto */
static int n;                  /* número de pares chave-valor */

/* definição e rotinas de Node */
{...}
static Node r; /* raiz da tst */

Value  get(char *key) {...}
static Node getT(Node x, char *key, int d) {...}
void   put(char *key, Value val) {...}
static Node putT(Node x, char *key,
                 Value val, int d) {...}
void   delete(char *k) {...}
Queue  keys() {...}   /* Iterador */
```

get(key): método clássico

A string que leva a um nó x é uma chave se e somente se x->c é o último caractere da chave e x->val != NULL.

```
/* Trie e TST */
Value get(char *key) {
   Node x = getT(r, key, 0);
   if (x == NULL) return NULL;
   return x->val;
}
```

get(key): método clássico

```
static Node getT(Node x, char *key, int d) {
   char c = key[d];
   if (x == NULL) return NULL;
   if (c < x->c)
      return getT(x->left, key, d);
   if (c > x->c)
      return getT(x->right, key, d);
   if (d < strlen(key)-1)
      return getT(x->mid, key, d+1);
   return x;
}
```

put(key, val): método clássico

É feita uma busca.
Se a key é encontrada, o valor val é substituido.
Caso contrário, chegamos a um NULL e devemos continuar a inserção, ou chegamos no último caractere de key.

```
/* Trie e TST */
void put(char *key, Value val) {
   r = putT(r, key, val, 0);
}
```

put(key, val): método clássico

```
static Node putT(Node x, char *key,
                 Value val, int d) {
    char c = key[d];
    if (x == NULL)
        x = newNode(c);
    if (c < x->c)
        x->left = putT(x->left, key, val, d);
    else if (c > x->c)
        x->right = putT(x->right, key, val, d);
    else if (d < strlen(key)-1)
        x->mid = putT(x->mid, key, val, d+1);
    else x->val = val;
    return x;
}
```

collect(): método auxiliar

O método coloca na fila q todas as chaves da subtrie cuja raiz é x, depois de acrescentar o prefixo pre a todas essas chaves.

```
static void collect(Node x, char *pre, Queue q) {
    int n = strlen(pre);
    if (x == NULL) return;
    collect(x->left, pre, q);
    /* ordem lexicográfica */
    pre[n] = x->c;   pre[n+1] = '\0';
    if (x->val != NULL) enqueue(q, pre);
    collect(x->mid, pre, q);
    pre[n] = '\0';
    collect(x->right, pre, q);
}
```

keys(): método clássico

```
Queue keys() {
    Queue q = queueInit();
    collect(r, "", q);
    return q;
}
```

keysWithPrefix(): método especial

Devolve todas as chaves na ST que têm prefixo pre.

```
Queue keysWithPrefix(char *pre) {
   Queue q = queueInit();   /* fila de chaves */
   Node x = getT(r, pre, 0);
   if (x == NULL) return q;
   if (x->val != NULL) enqueue(q, pre);
   collect(x->mid, pre, q);
   return q;
}
```

`longestPrefixOf()`: método especial

Devolve a maior chave que é prefixo de `s`.

```
char *longestPrefixOf(char *s) {
   int max, i; char c, *p;
   Node x = r;

   if (x == NULL || strlen(s) == 0)
      return NULL;

   max = i = 0;
```

longestPrefixOf(): método especial

```
    while (x != NULL && i < strlen(s)) {
       c = s[i];
       if (c < x->c) x = x->left;
       else if (c > x->c) x = x->right;
       else {
          i++;
          if (x->val != NULL) max = i;
          x = x->mid;
       }
    }
    p = mallocSafe((max+1)*sizeof(char));
    strncpy(p, s, max);  p[max] = '\0';
    return p;
}
```

keysThatMatch(): método especial

Devolve todas as chaves que
casam com o padrão pat.

Os caracteres '.' em pat são curingas.

```
Queue keysThatMatch(char *pat) {
   Queue q = queueInit();
   collectC(r, "", 0, pat, q);
   return q;
}
```

collectC: versão do collect com curinga.

Mais um collect()

Coloca em q todas as chaves da trie que
têm prefixo pre e casam com o padrão pat.

```
static void collectC(Node x, char *pre,
                     int i, char *pat, Queue q) {
   int n = strlen(pre);
   char c = pat[i];

   if (x == NULL) return;
   if (c == '.' || c < x->c)
      collectC(x->left, pre, i, pat, q);
```

Mais um collect()

```
    pre[n] = x->c;   pre[n+1] = '\0';
    if (c == '.' || c == x->c) {
       if (i == strlen(pat) - 1
          && x->val != NULL)
          enqueue(q, pre);
       else if (i < strlen(pat) - 1)
          collectC(x->mid, pre, i+1, pat, q);
    }
    pre[n] = '\0';
    if (c == '.' || c > x->c)
       collectC(x->right, pre, i, pat, q);
}
```

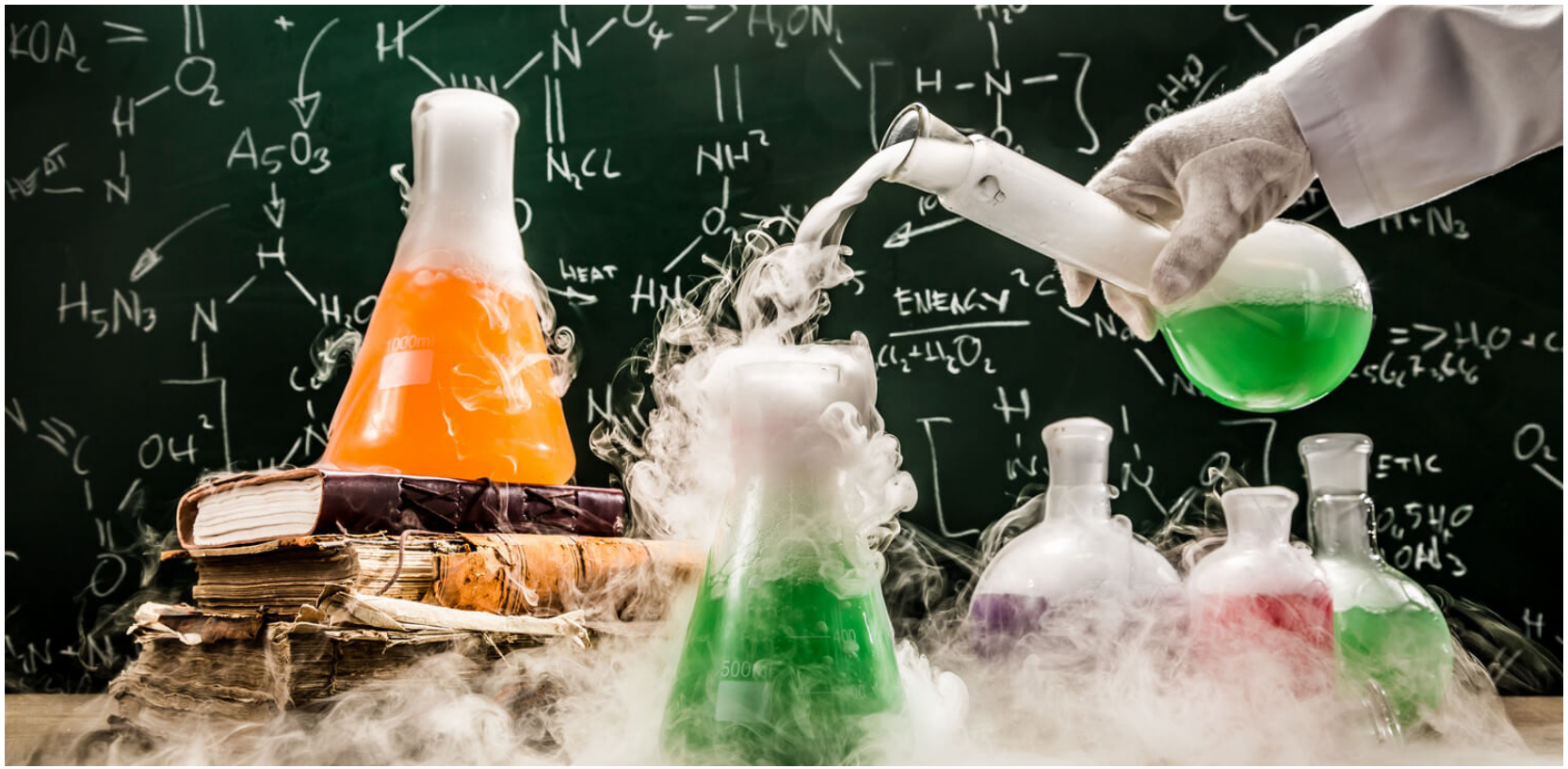
Consumo de espaço e tempo

Espaço. A propriedade mais importante de uma TST é que ela tem apenas três links por nó.

Proposição J: O número de links em uma TST com n chaves de comprimento médio w é entre 3n e 3nw.

Proposição K: O número esperado de nós visitados durante uma busca mal sucedida em uma TST com n chaves aleatórias é aproximadamente $\lg n$.

Alguns experimentos



Fonte: https://singularityhub.com/

Experimentos: les_miserables.txt

ST com 26764 itens

```
% wc les-miserables.txt
68116 568531 3322649 les-miserables.txt
```

ST	criada (s)
BST	0.626
RedBlackBST	0.577
SeparateChainST	0.495 (4096, 18, $\alpha = 6$)
LinearProbingST	0.411 (65536, 67, $\alpha = 0.4$)
TrieST	0.574
TST	0.497

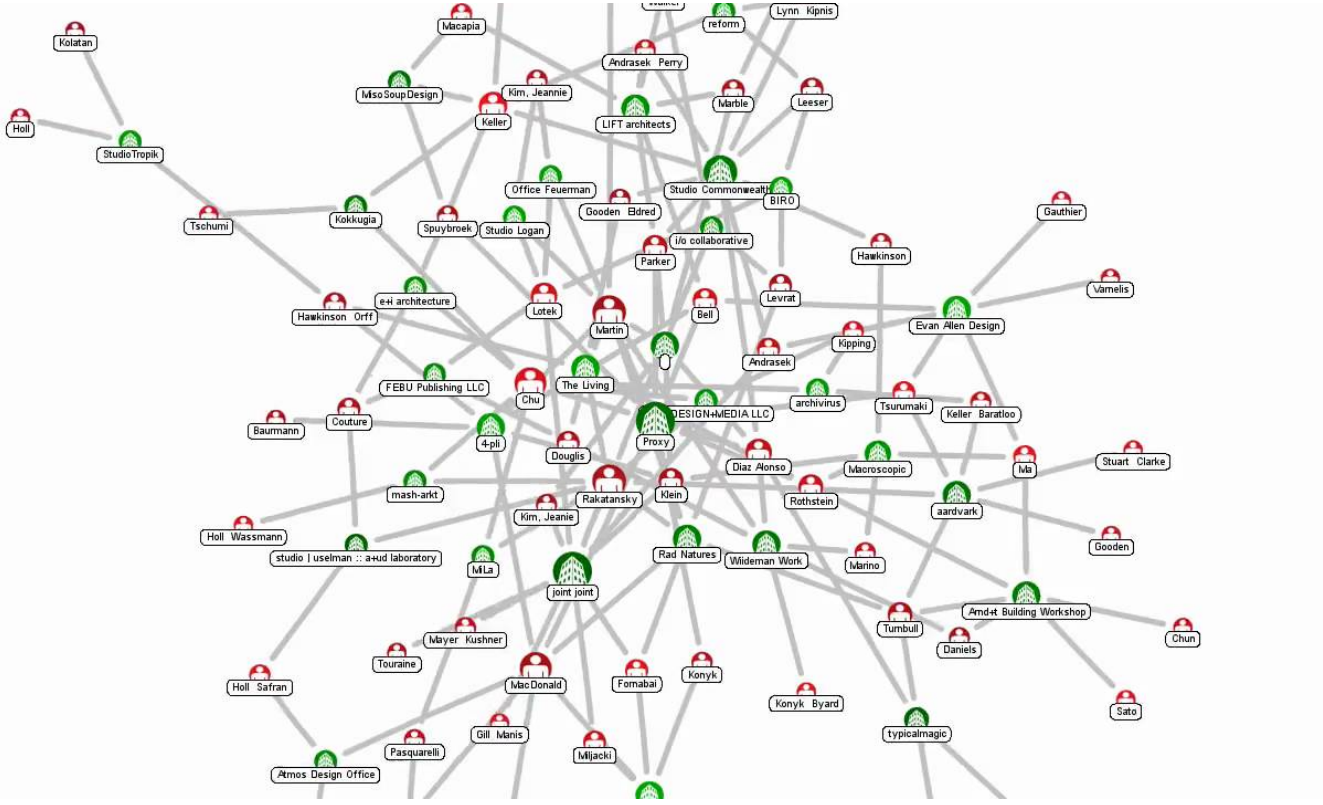
Experimentos: actors.list

ST com 1482495 itens

```
% wc actors.list
16612200 124796815 932688622 actors.list
```

ST	criada (s)
BST	141.968
RedBlackBST	168.723
SeparateChainST	110.035 (262144, 19, $\alpha = 5$)
LinearProbingST	73.123 (4194304, 4787, $\alpha = 0.35$)
TrieST	OutOfMemoryError
TST	107.223

Digrafos



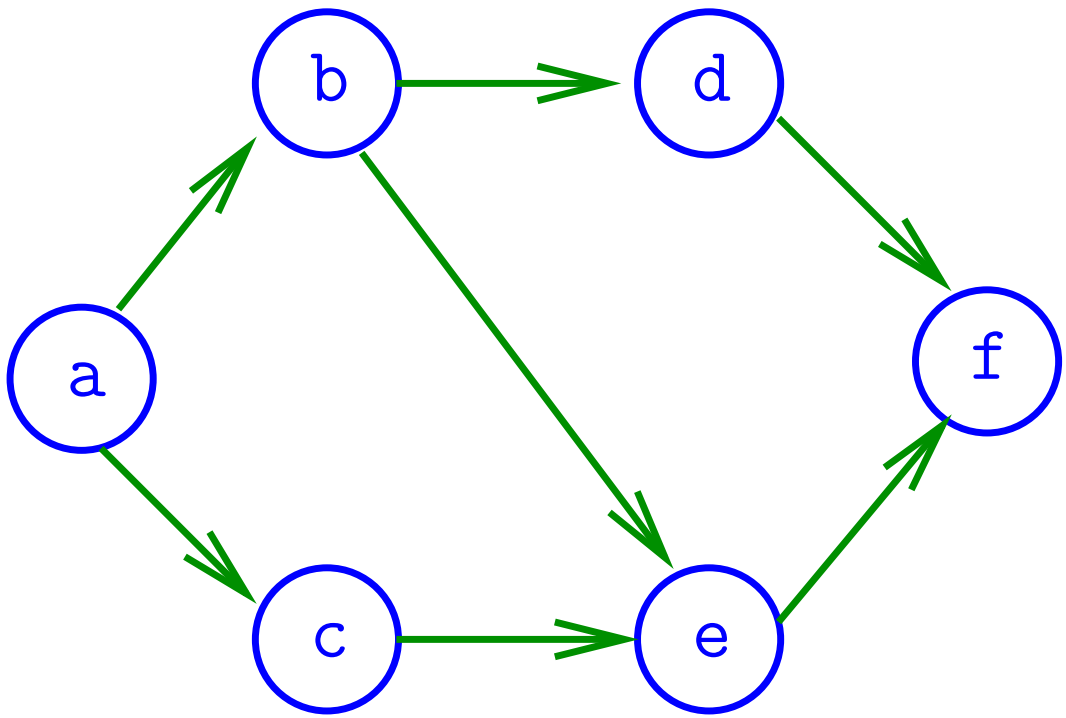
Fonte: `Force Directed Graph`

Referências: Directed graphs (SW): slides, vídeo.

Digrafos

Um **digrafo** (*directed graph*) consiste de um conjunto de vértices (bolas) e um conjunto de arcos (flechas).

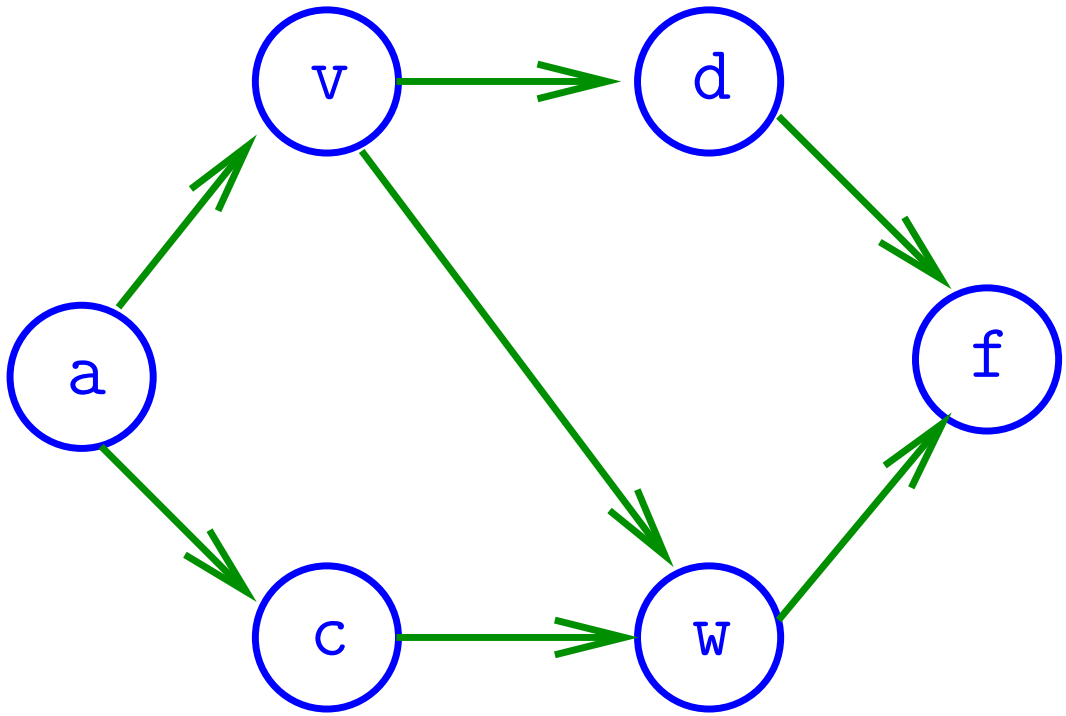
Exemplo: representação de um digrafo



Arcos

Um **arco** é um par ordenado de vértices.

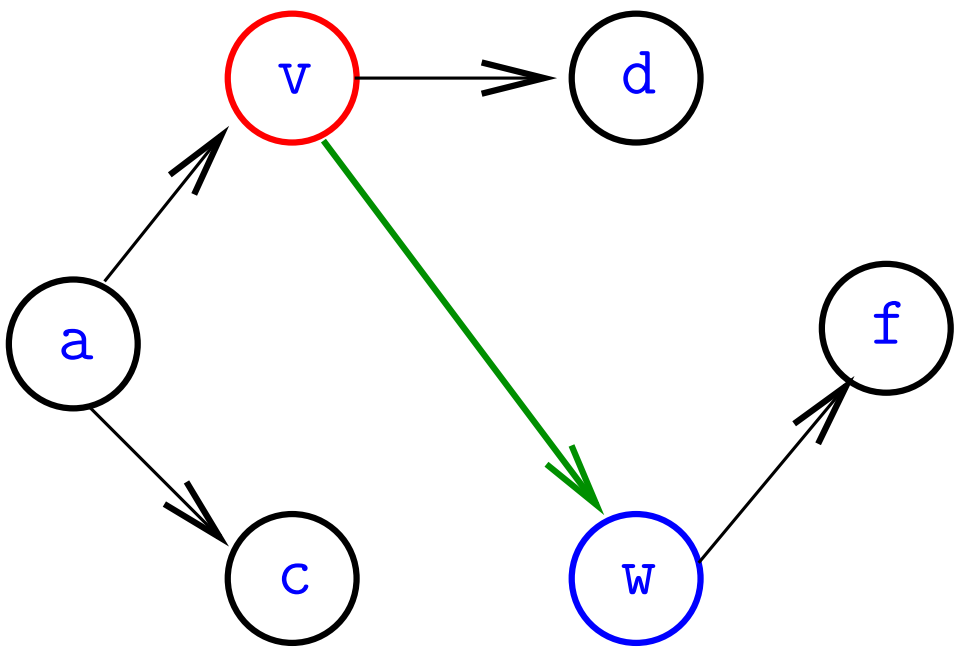
Exemplo: `v` e `w` são vértices e `v-w` é um arco



Ponta inicial e final

Para cada arco v-w,
o vértice v é a **ponta inicial** e w é a **ponta final**.

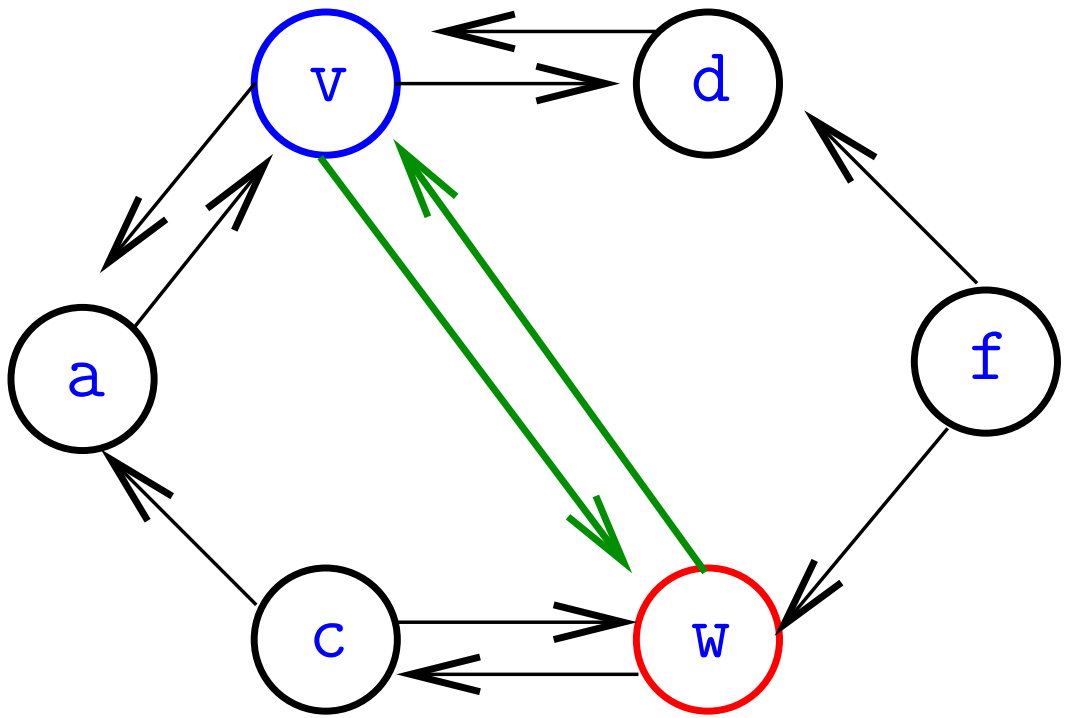
Exemplo: v é ponta inicial e w é ponta final de v-w



Arcos anti-paralelos

Dois arcos são **anti-paralelos**
se a ponta inicial de um é ponta final do outro.

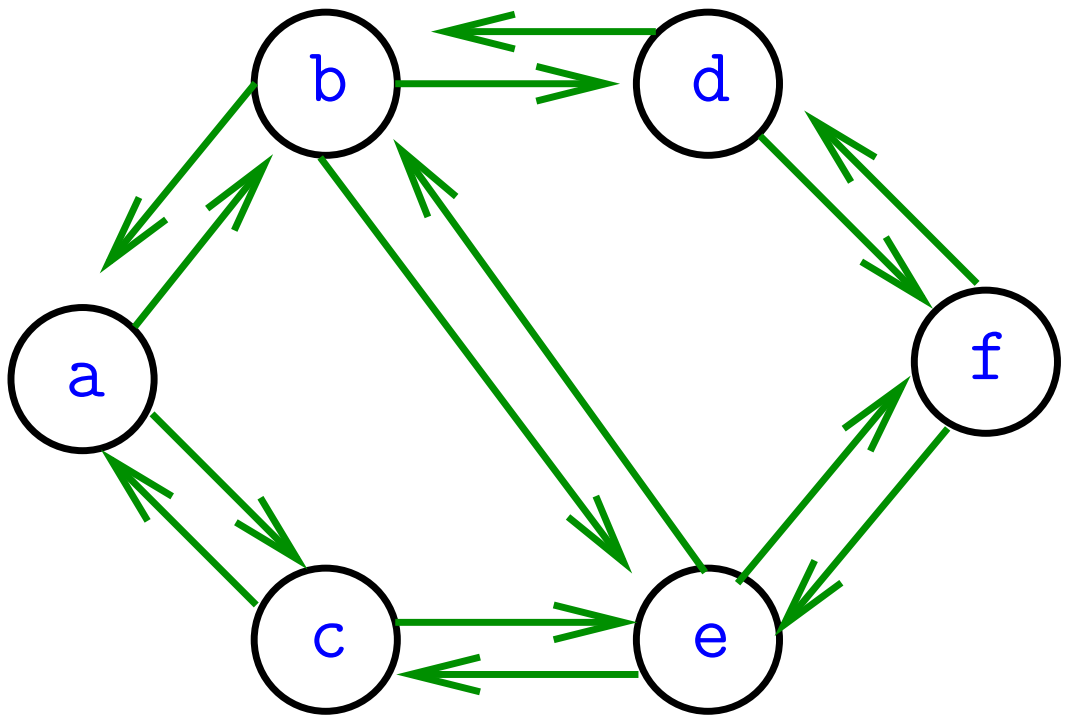
Exemplo: `v`-`w` e `w`-`v` são anti-paralelos



Digrafos simétricos

Um digrafo é **simétrico**
se cada um de seus arcos é anti-paralelo a outro.

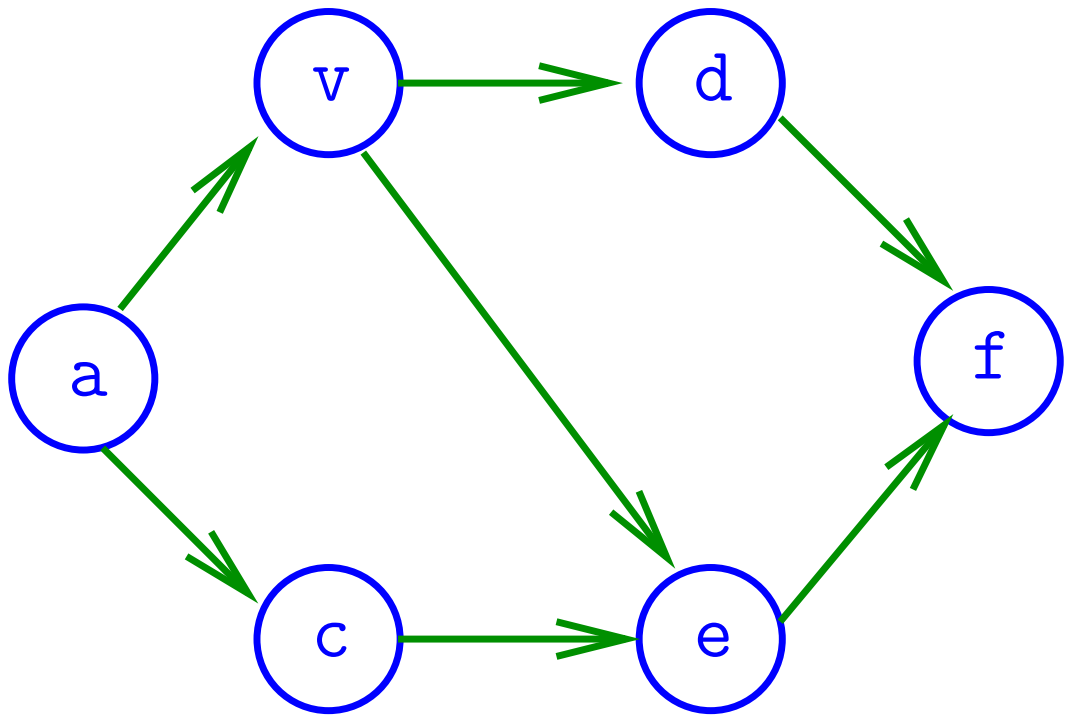
Exemplo: digrafo simétrico



Graus de entrada e saída

grau de entrada de v = no. arcos com ponta final v
grau de saída de v = no. arcos com ponta inicial v

Exemplo: v tem grau de entrada 1 e de saída 2



Número de arcos

Quantos arcos, no máximo,
tem um digrafo com `V` vértices?

Número de arcos

Quantos arcos, no máximo,
tem um digrafo com $\mathtt{V}$ vértices?

A resposta é $\mathtt{V} \times (\mathtt{V} - 1) = \Theta(\mathtt{V}^2)$.

digrafo **completo** = todo par ordenado de vértices distintos é arco

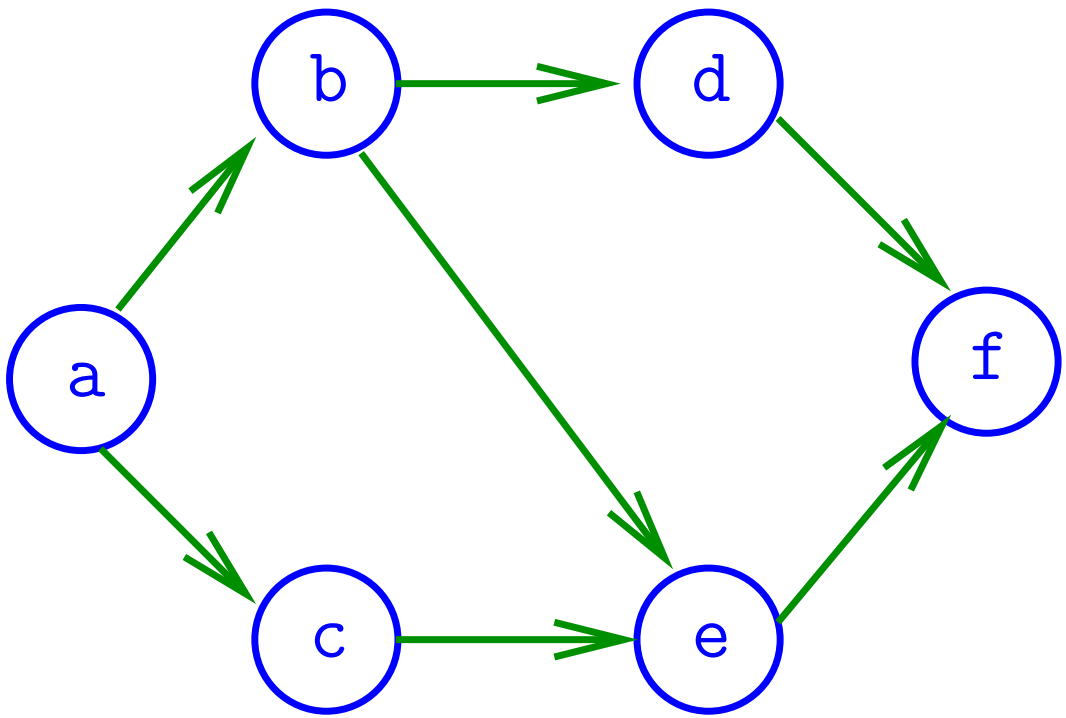
digrafo **denso** = tem "muitos" arcos

digrafo **esparso** = tem "poucos" arcos

Especificação

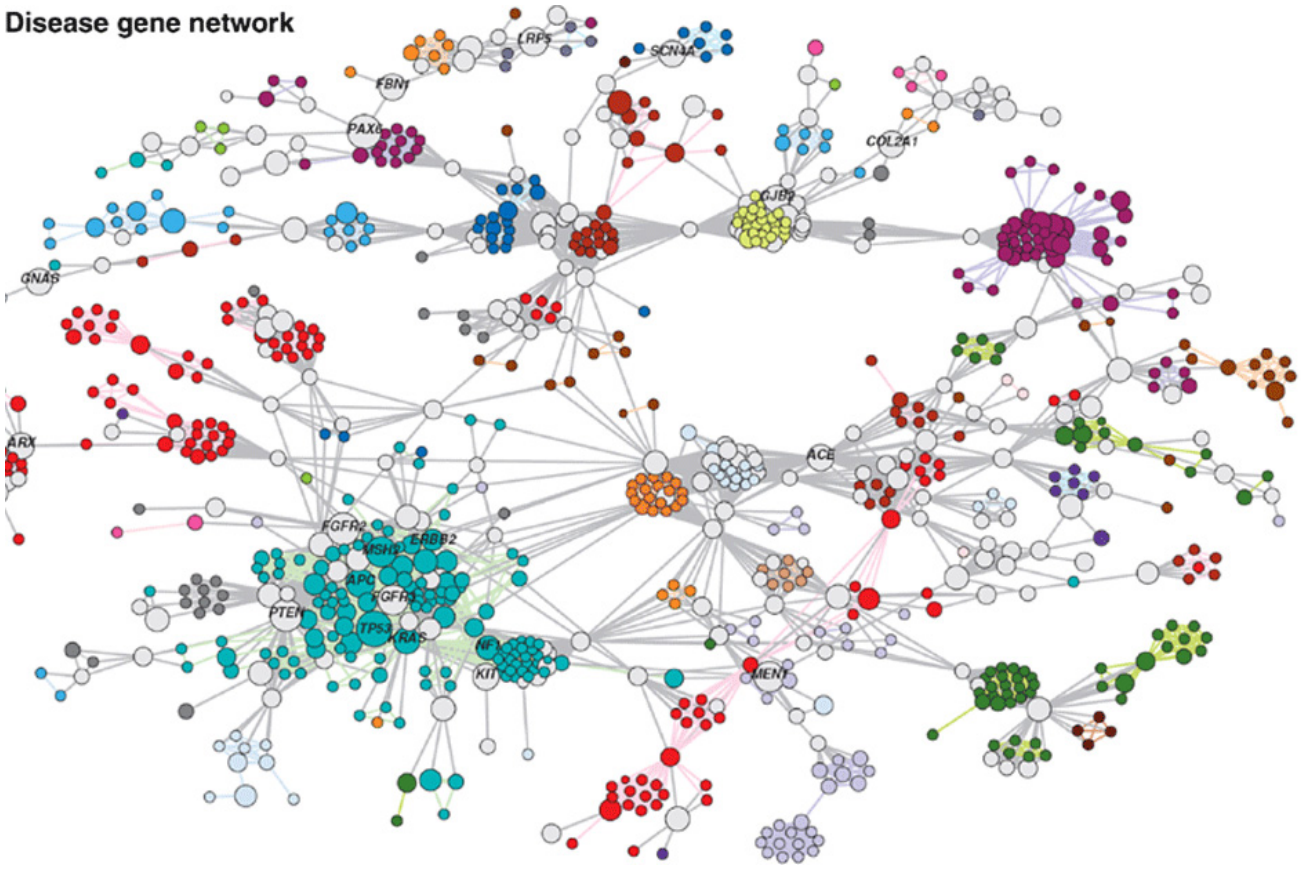
Digrafos podem ser especificados através de sua lista de arcos.

Exemplo:




```
d-f
b-d
a-c
b-e
e-f
a-b
```

Grafos



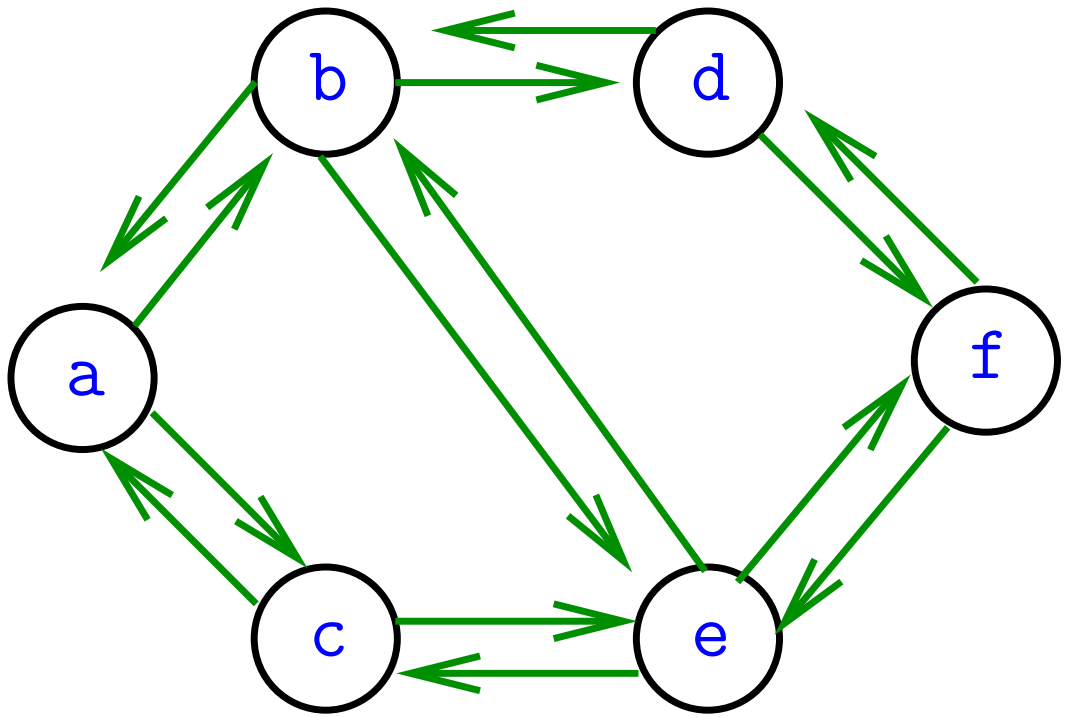
Fonte: Scaling Computation of Graph Structured Data with NScale

Referências: Undirected graphs (SW): slides, vídeo.

Grafos

Um **grafo** é um digrafo **simétrico**.

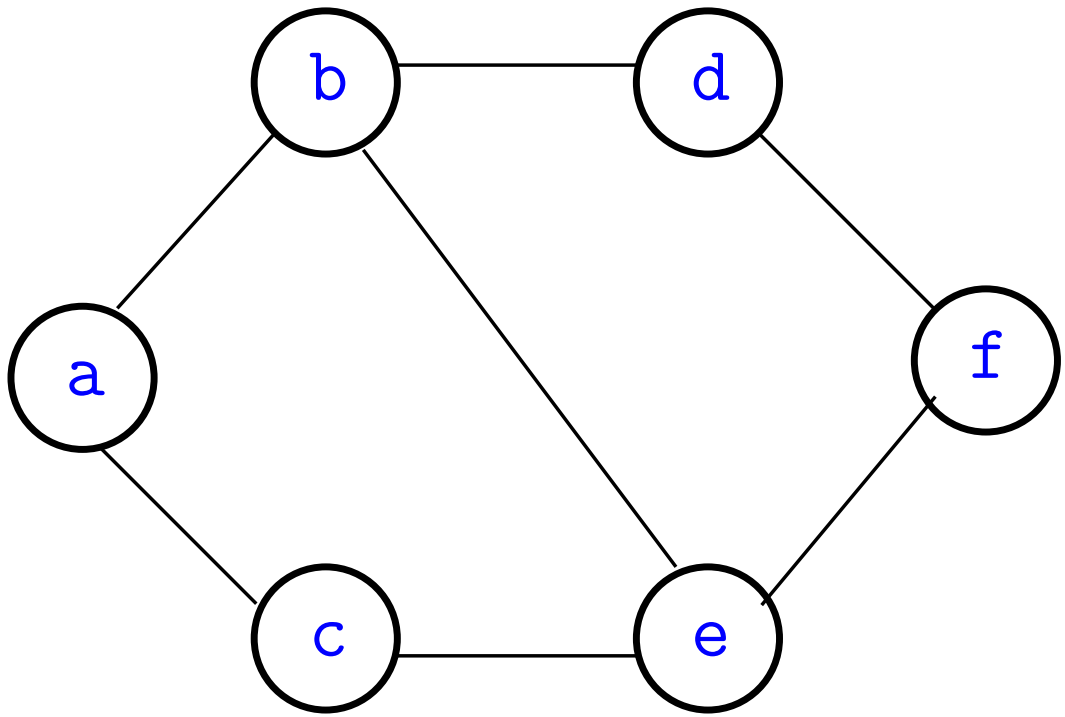
Exemplo: um grafo



Grafos

Um **grafo** é um digrafo **simétrico**.

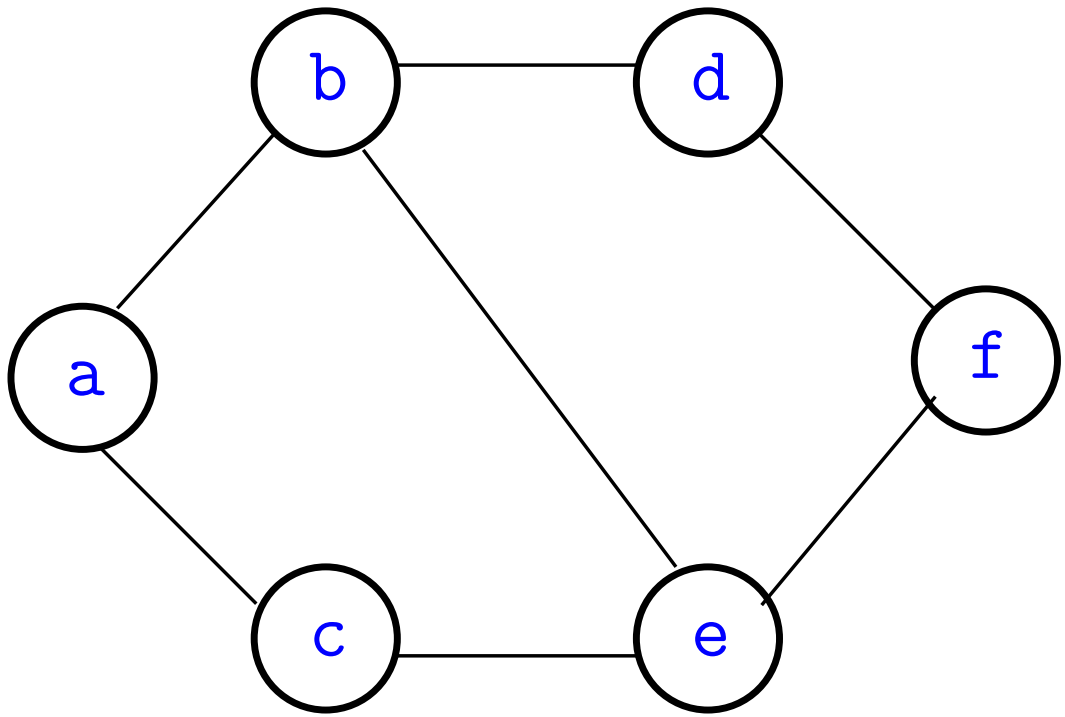
Exemplo: representação usual



Arestas

Uma **aresta** é um par de arcos anti-paralelos.

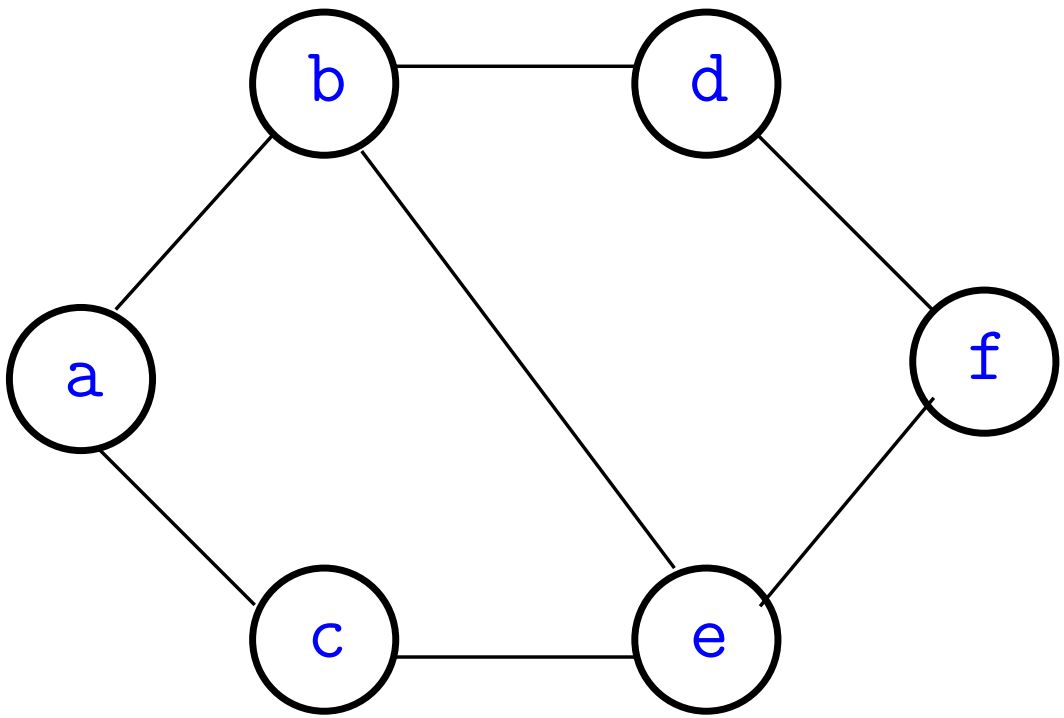
Exemplo: `b-a` e `a-b` são a **mesma** aresta



Especificação

Grafos podem ser especificados através de sua lista de arestas.

Exemplo:



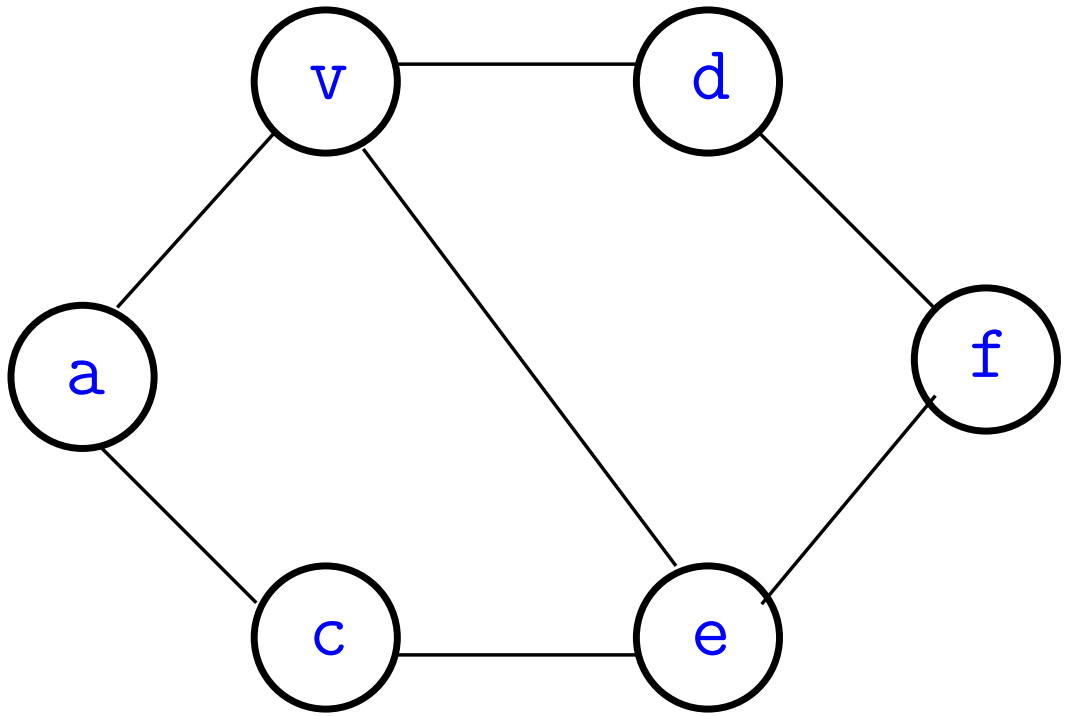

```
f-d
b-d
c-a
e-b
e-f
a-b
```

Graus de vértices

Em um grafo

grau de v = número de arestas com ponta em v

Exemplo: v tem grau 3



Número de arestas

Quantas arestas, no máximo,
tem um grafo com `V` vértices?

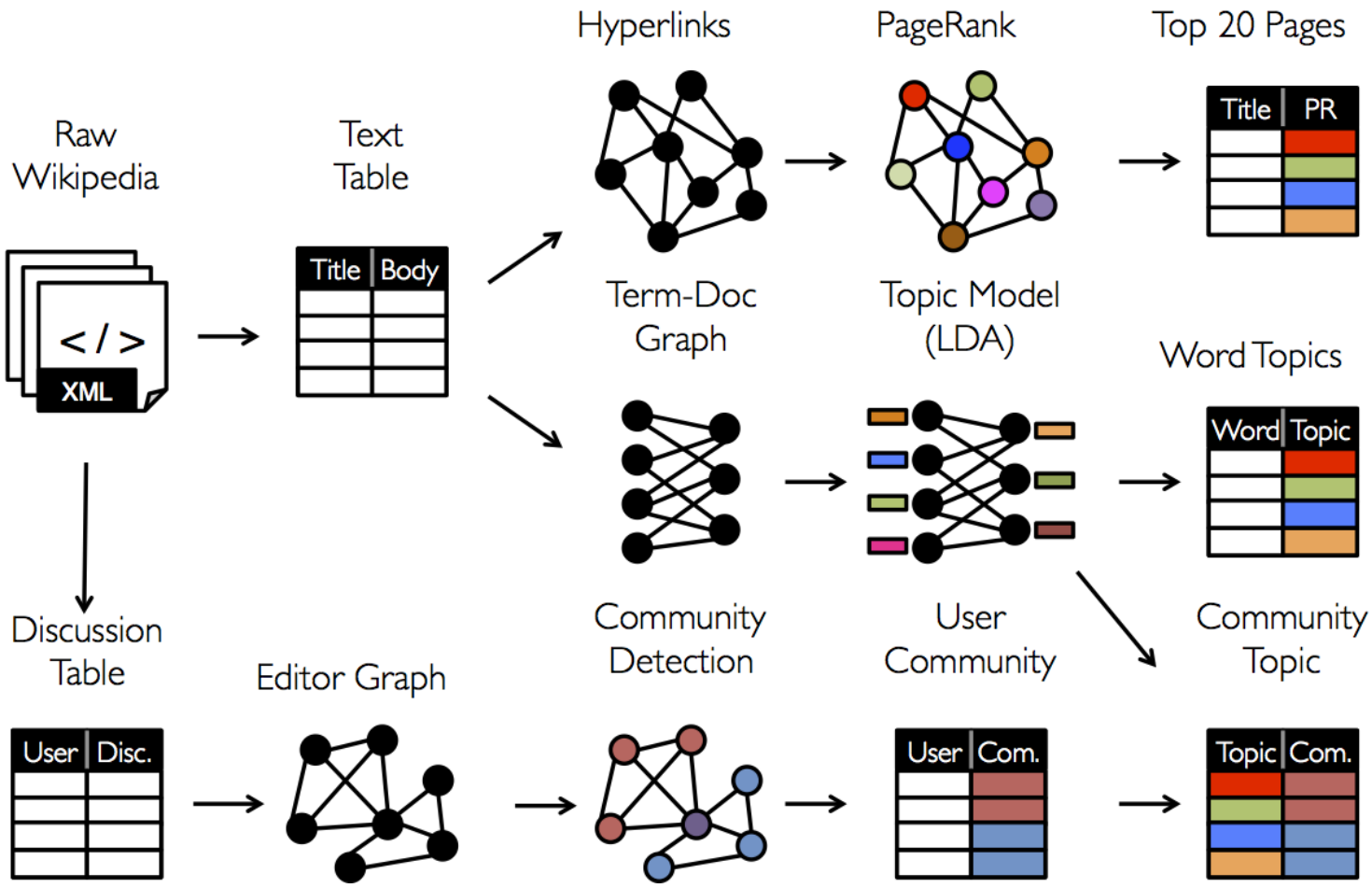
Número de arestas

Quantas arestas, no máximo,
tem um grafo com $\mathtt{V}$ vértices?

A resposta é $\mathtt{V} \times (\mathtt{V} - 1)/2 = \Theta(\mathtt{V}^2)$.

grafo **completo** = todo par **não**-ordenado de vértices distintos é aresta

Digrafos no computador



Fonte: GraphX Programming Guide

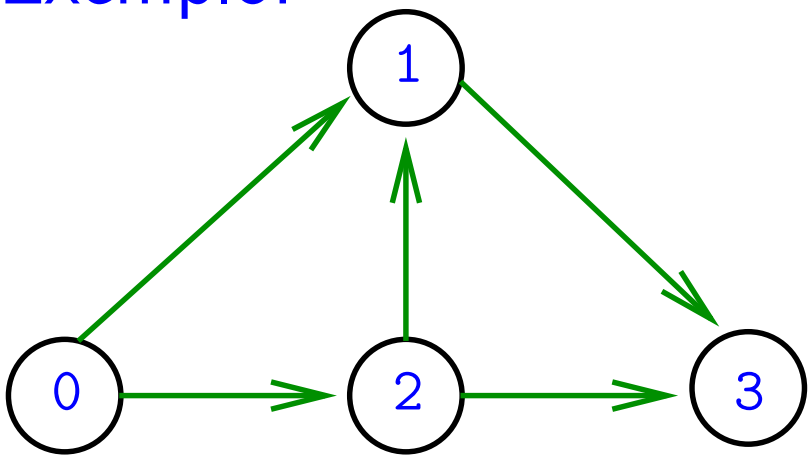
Matriz de adjacência de digrafos

Matriz de adjacência de um digrafo tem linhas e colunas indexadas por vértices:

`adj[v][w]` = 1 se `v`-`w` é um arco

`adj[v][w]` = 0 em caso contrário

Exemplo:



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

Consumo de espaço: $\Theta(V^2)$ fácil de implementar

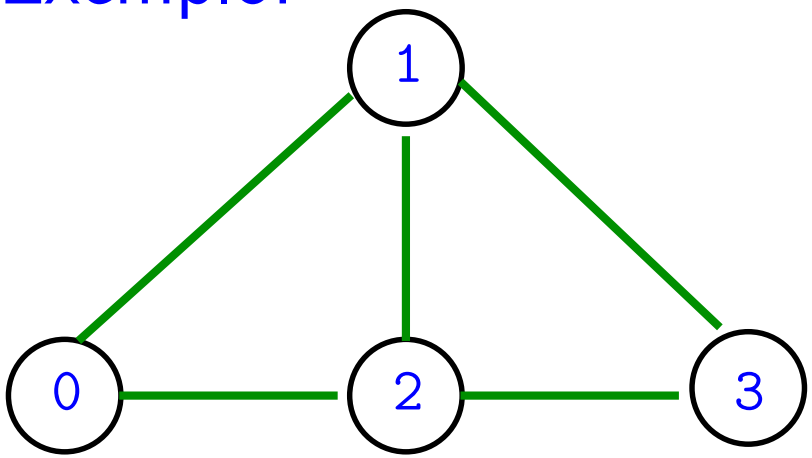
Matriz de adjacência de grafos

Matriz de adjacência de um grafo tem linhas e colunas indexadas por vértices:

`adj[v][w]` = 1 se `v`-`w` é um aresta

`adj[v][w]` = 0 em caso contrário

Exemplo:

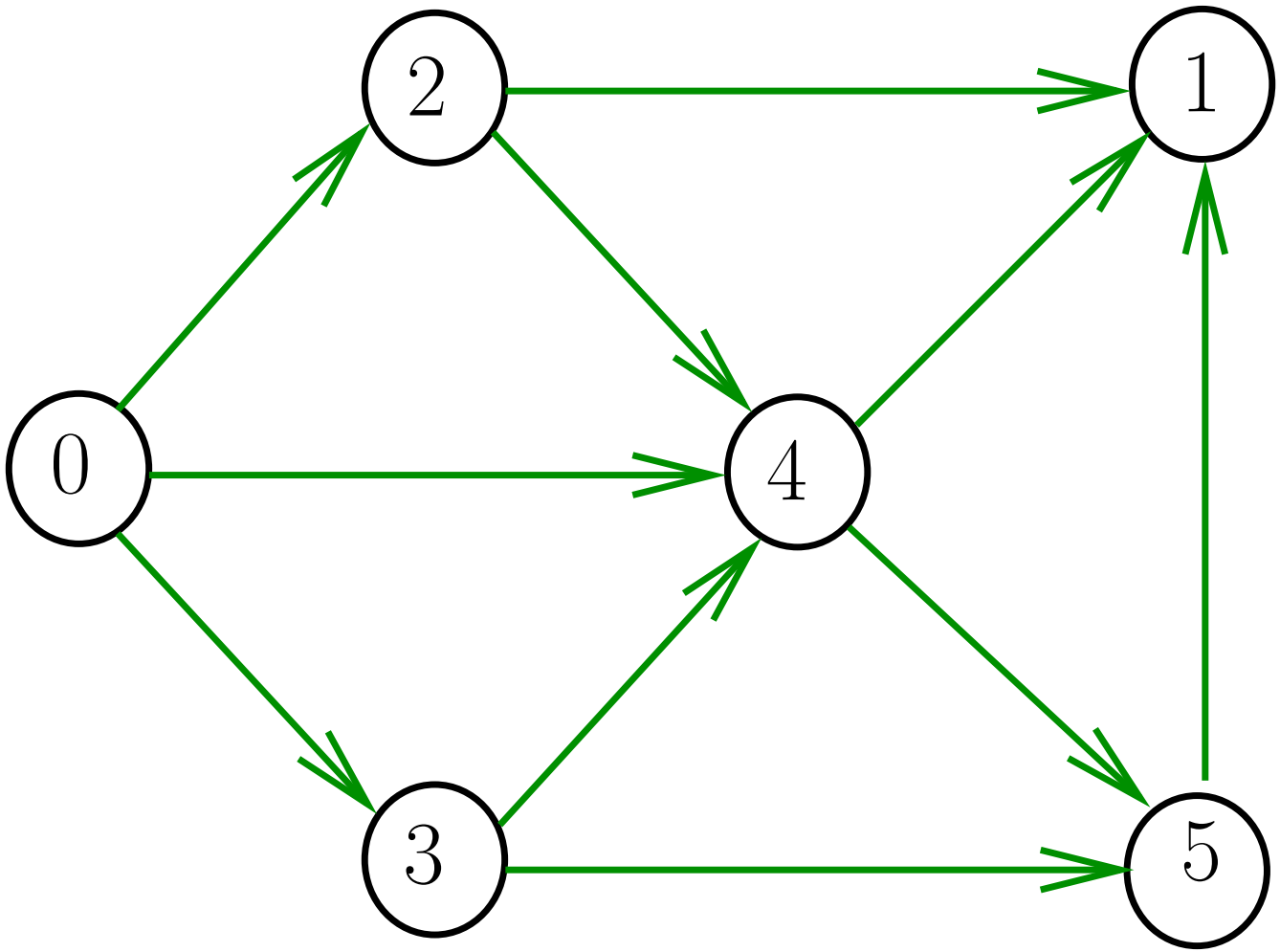


	0	1	2	3
0	0	1	1	0
1	1	0	1	1
2	1	1	0	1
3	0	1	1	0

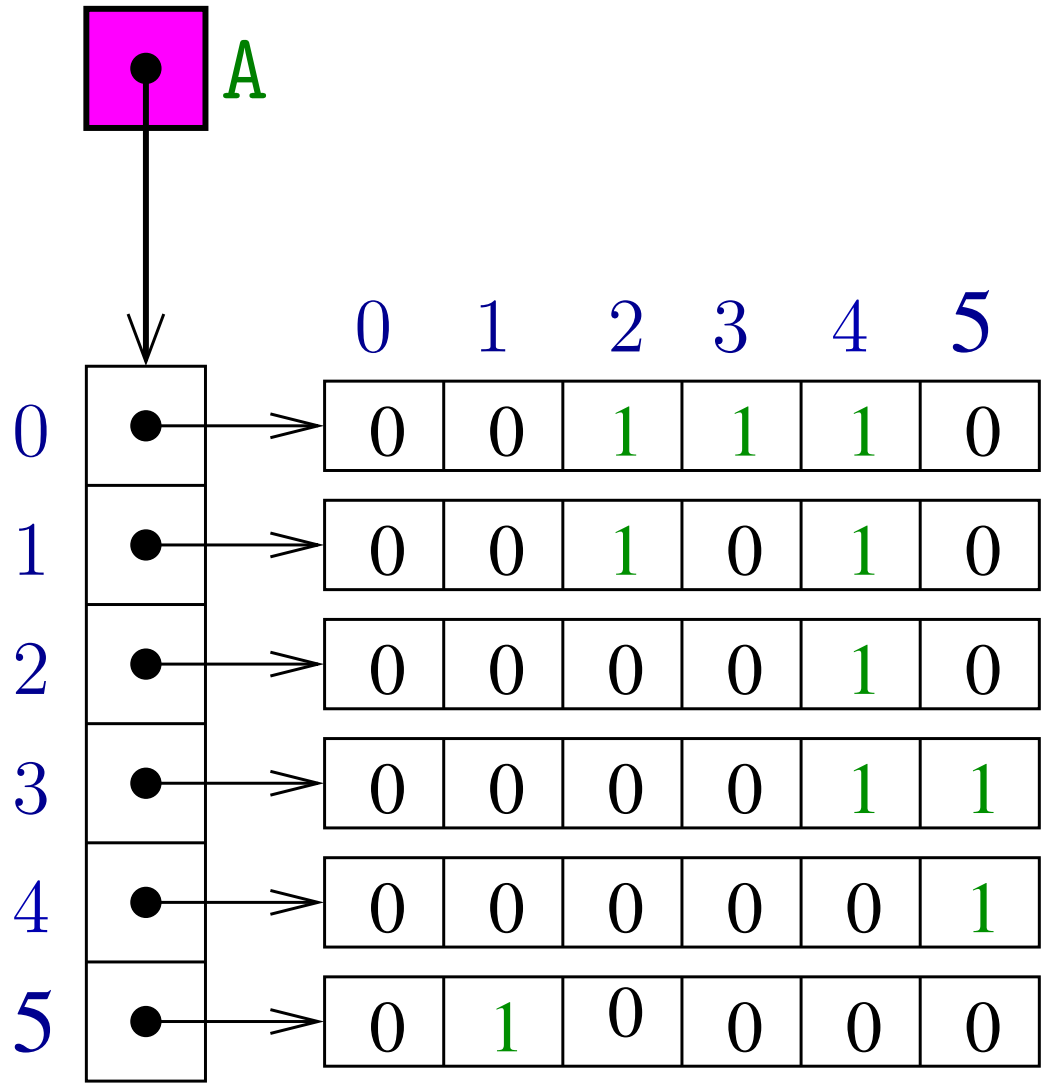
Consumo de espaço: $\Theta(V^2)$ fácil de implementar

Digrafo

Digraph G

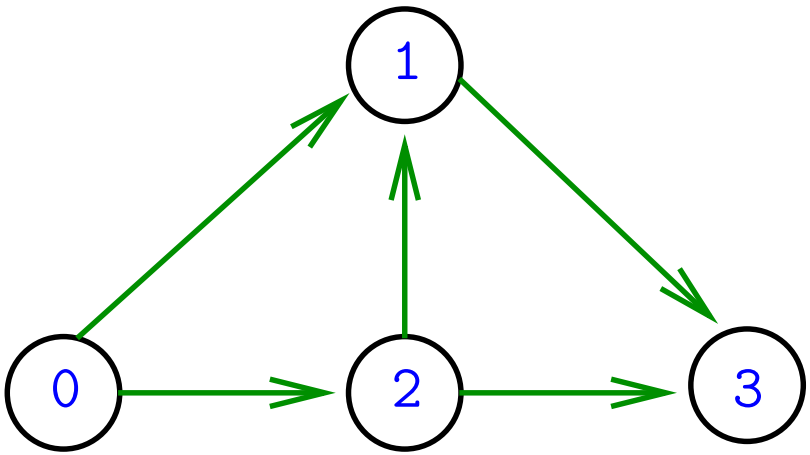


Estruturas de dados



Matriz de incidência de digrafos

Uma **matriz de incidências** de um digrafo
tem linhas indexadas por vértices e
colunas por arcos e cada entrada [k] [vw]
é -1 se k = v, $+1$ se k = w, e 0 em caso contrário.



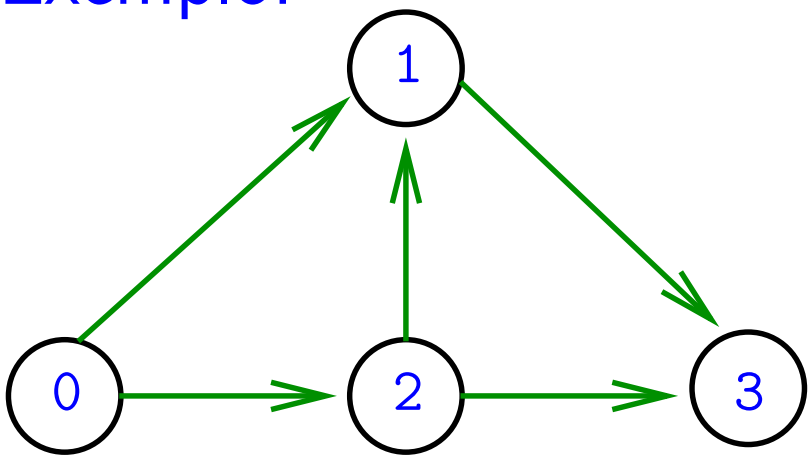
	0-1	0-2	2-1	2-3	1-3
0	-1	-1	0	0	0
1	$+1$	0	$+1$	0	-1
2	0	$+1$	-1	-1	0
3	0	0	0	$+1$	$+1$

Consumo de espaço: $\Theta(nm)$
Interessante do ponto de vista de otimização linear.

Vetor de listas de adjacência de digrafos

Na representação de um digrafo através de **listas de adjacência** tem-se, para cada vértice v, uma lista dos vértices que são vizinhos v.

Exemplo:




```
0:  1,  2
1:  3
2:  1,  3
3:
```

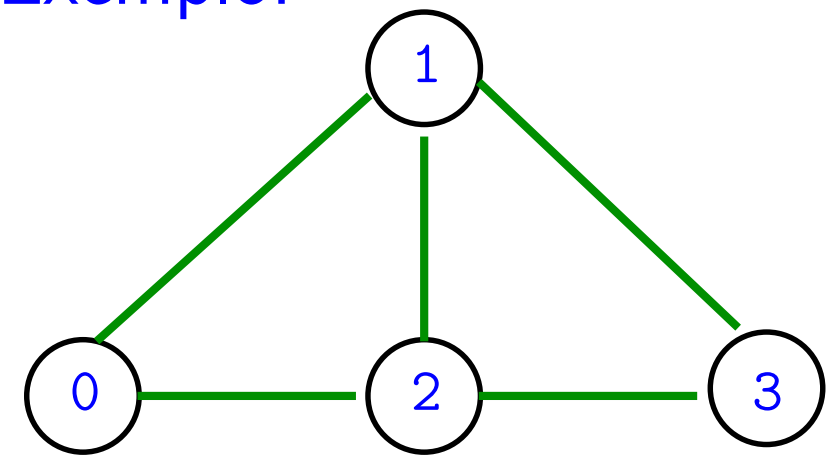
Consumo de espaço: $\Theta(V + A)$ (linear)

Manipulação eficiente

Vetor de lista de adjacência de grafos

Na representação de um grafo através de **listas de adjacência** tem-se, para cada vértice `v`, uma lista dos vértices que são pontas de arestas incidentes a `v`.

Exemplo:



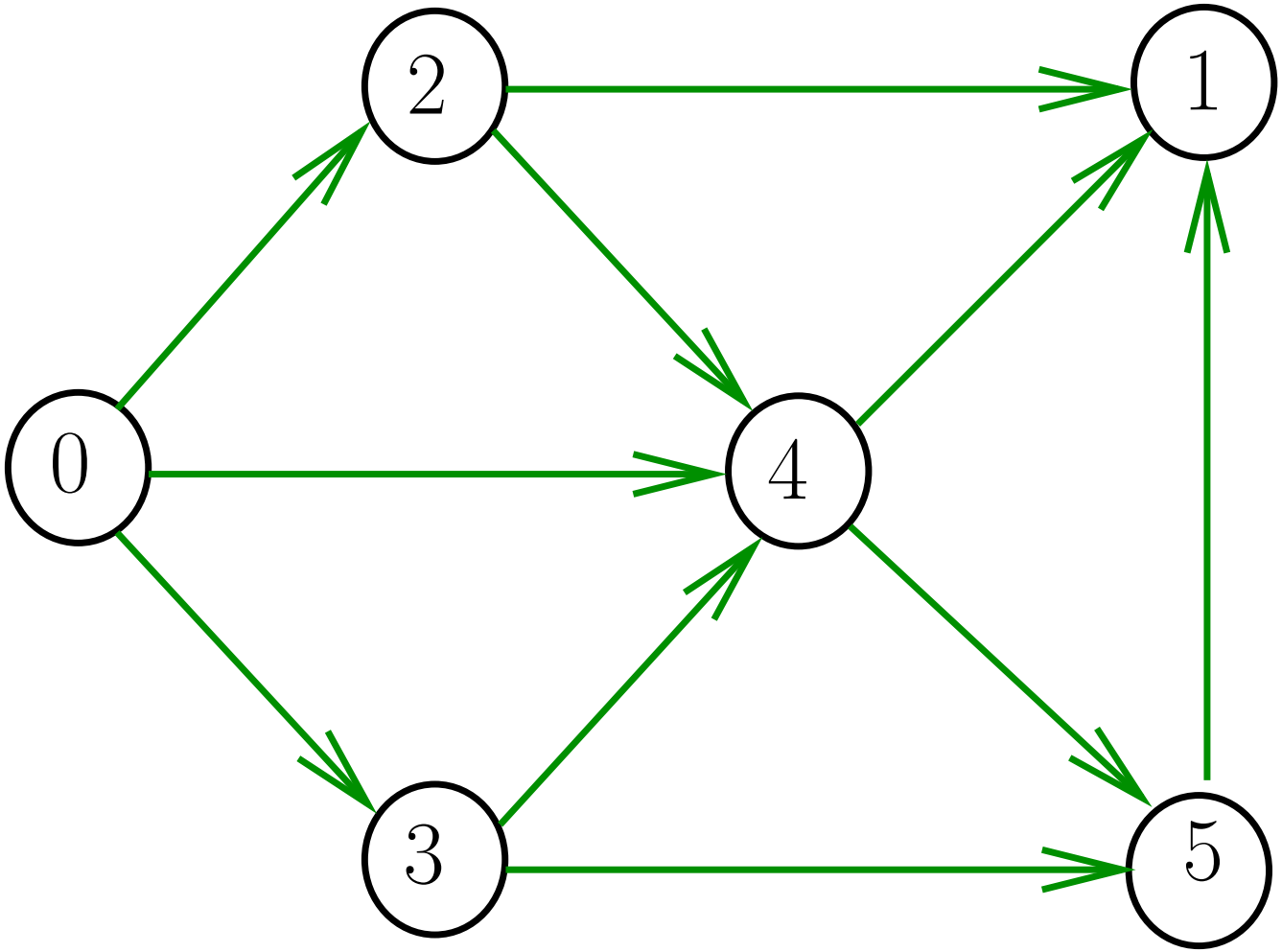

```
0:  1,  2
1:  3,  0,  2
2:  1,  3,  0
3:  1,  2
```

Consumo de espaço: $\Theta(\mathtt{V} + \mathtt{A})$ (linear)

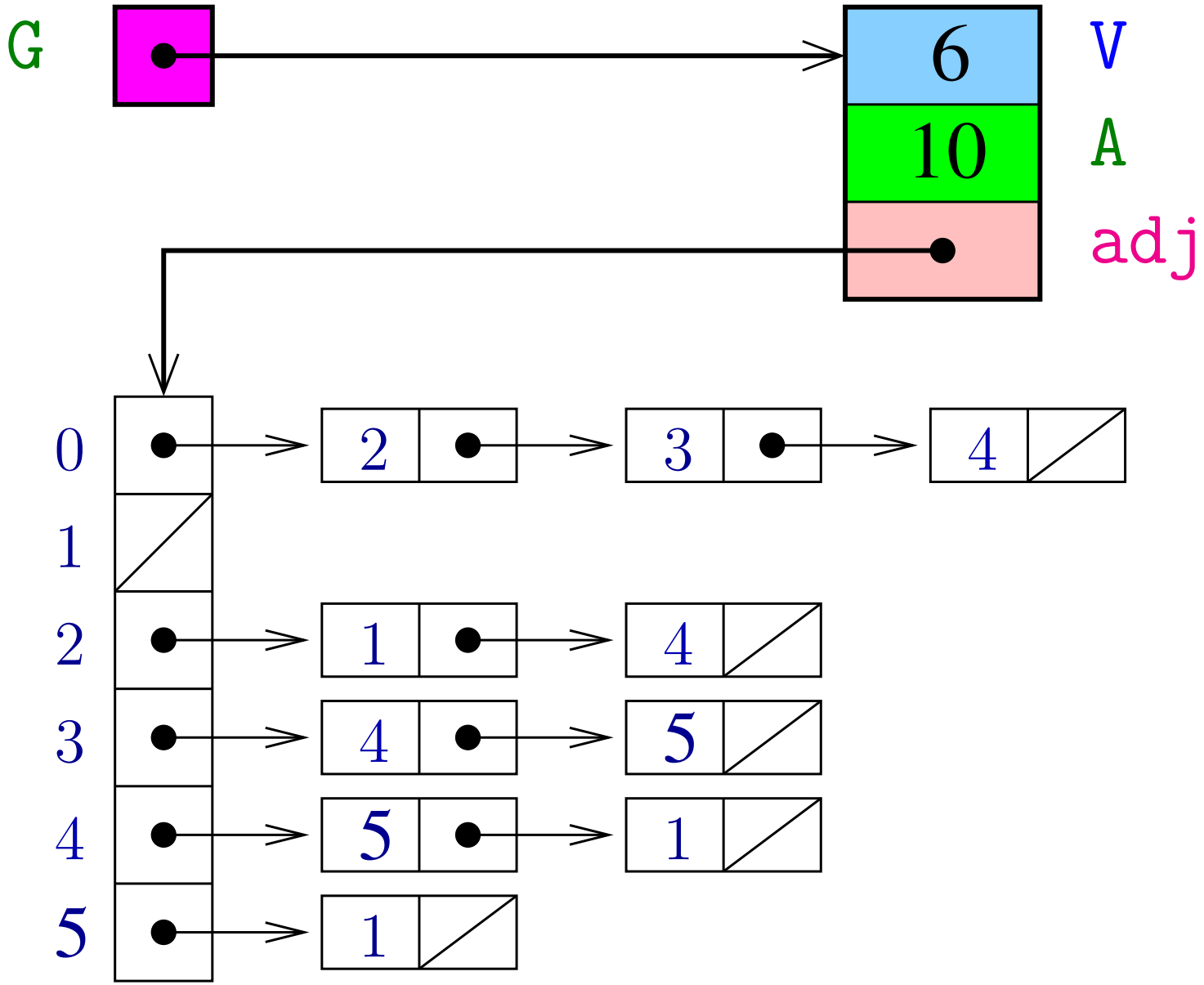
Manipulação eficiente

Digrafo

Digraph G



Estruturas de dados



Arquivo Digraph.h

```
typedef void *Digraph;

Digraph newDigraph(int);

void addEdge(Digraph, int, int);

int *adj(Digraph, int);  /* lista dos vizinhos */

int outdegree(Digraph, int);
int indegree(Digraph, int);

/* digrafo com os arcos invertidos */
Digraph reverse(Digraph);
```

Esqueleto do arquivo Digraph.c

```
static struct digraph {
   int V;                          /* no. vértices */
   int E;                             /* no. arcos */
   Link *adj;                  /* listas de inteiros */
   int *indegree;                /* grau de entrada */
};

static typedef struct digraph *TDigraph;
```

Esqueleto do arquivo Digraph.c

```
/* digrafo de ordem V */
Digraph newDigraph(int V) {...}

void addEdge(Digraph D, int v, int w) {...}

Link adj(Digraph D, int v) {...}

int outdegree(Digraph D, int v) {...}
int indegree(Digraph D, int v) {...}

Digraph reverse(Digraph D) {...}
```

Digraph

```
Digraph newDigraph(int V) {
   TDigraph D = mallocSafe(sizeof(*D));
   D->V = V;
   D->E = 0;
   D->indegree = mallocSafe(V*sizeof(int));
   D->adj = mallocSafe(V*sizeof(Link));
   for (int v = 0; v < V; v++) {
      D->indegree[v] = 0;
      D->adj[v] = NULL;
   }
   return D;
}
```

Digraph

```
/* insere um arco */
void addEdge(Digraph D, int v, int w) {
   insert(D->adj[v], w); /* insere na lista */
   (D->indegree[w])++;
   (D->E)++;
}

/* retorna a lista de adjacência de v */
Link adj(Digraph D, int v) {
   return D->adj[v];
}
```

Digraph

```
/* retorna o grau de entrada de v */
int indegree(Digraph D, int v) {
   return D->indegree[v];
}
```

Se for usar o grau de saída também na aplicação, deve criar um vetor extra `outdegree` ou algo assim.

```
/* retorna o grau de saída de v */
int outdegree(Digraph D, int v) {
   return D->outdegree[v];
}
```

Digraph

```
/* retorna o digrafo reverso */

Digraph reverse(Digraph D) {
   Digraph reverse = newDigraph(D->V);
   int v, Link u;
   for (v = 0; v < D->V; v++) {
      for (u = D->adj(v); u != NULL; u = u->next)
         addEdge(reverse, u->vertex, v);
   }
   return reverse;
}
```