

## Grafos – Representação

Agora iremos implementar o TAD grafo com representação encadeada com a estrutura e operações a seguir:

```
typedef struct _nodetype {  
    int info;  
    struct _nodetype *point;  
    struct _nodetype *next;  
} nodetype;  
typedef nodetype *TADgraph;
```

## Grafos – Representação

Implemente as operações a seguir sobre o TAD apresentado no slide anterior.

```
void inicializaGrafo(TADgraph *);  
void joinwt (nodetype *, nodetype *, int);  
void join (nodetype *, nodetype *);  
void remvwt (nodetype *, nodetype *);  
void remv (nodetype *, nodetype *);  
char adjacent (nodetype *, nodetype *);  
nodetype *findnode (TADgraph, int);  
nodetype *addnode (TADgraph*, int);  
int remvnode(TADgraph *, nodetype *);
```

```
void inicializaGrafo(int *grafo, listaDeNodos node)
{
    int i;
    for (i=0; i<MAXNODES; i++)
        node[i].livre = 1;
    *grafo = -1;
}
```

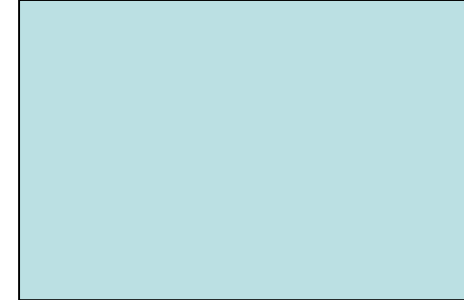


```
void inicializaGrafo(TADgraph *grafo)
{
    *grafo = NULL;
}
```

```
void joinwt (listaDeNodos node, int *listaDeNodosVazios, int p,
int q, int wt) {
    int r, r2;
    r2 = -1;
    r = node[p].point;
    while (r >= 0 && node[r].point != q) {
        r2 = r;
        r = node[r].next;
    }
    if (r >= 0) {
        node[r].info = wt;
        return;
    }
    r = getnode(listaDeNodosVazios, node);
    node[r].point = q;
    node[r].next = -1;
    node[r].info = wt;
    (r2 < 0 ) ? (node[p].point = r) : (node[r2].next = r);
}
```



```
void joinwt (nodetype *p, nodetype *q, int wt) {
    nodetype * r, *r2;
    r2 = NULL;
    r = p->point;
    while (r && r->point != q) {
        r2 = r;
        r = r->next;
    }
    if (r) {
        r->info = wt;
        return;
    }
    if (!(r = (nodetype *) malloc (sizeof(nodetype)))) exit(1);
    r->point = q;
    r->next = NULL;
    r->info = wt;
    (!r2) ? (p->point = r) : (r2->next = r);
}
```



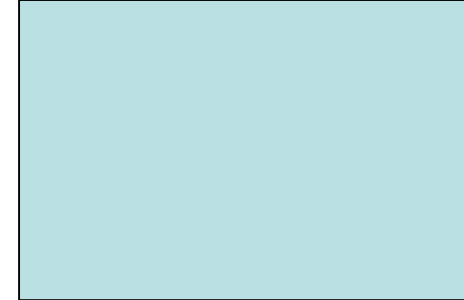
```
void join (listaDeNodos node, int *listaDeNodosVazios,
int p, int q){
    int r, r2;
    r2 = -1;
    r = node[p].point;
    while (r >= 0 && node[r].point != q) {
        r2 = r;
        r = node[r].next;
    }
    if (r >= 0) {
        return;
    }
    r = getnode(listaDeNodosVazios, node);
    node[r].point = q;
    node[r].next = -1;
    (r2 < 0 ) ? (node[p].point = r) : (node[r2].next = r);
}
```



```
void join (nodetype *p, nodetype *q)
{
    nodetype * r, *r2;
    r2 = NULL;
    r = p->point;
    while (r && r->point != q)
    {
        r2 = r;
        r = r->next;
    }
    if (r)
        return;
    if (!(r = (nodetype *) malloc (sizeof(nodetype)))) exit(1);
    r->point = q;
    r->next = NULL;
    (!r2) ? (p->point = r) : (r2->next = r);
}
```



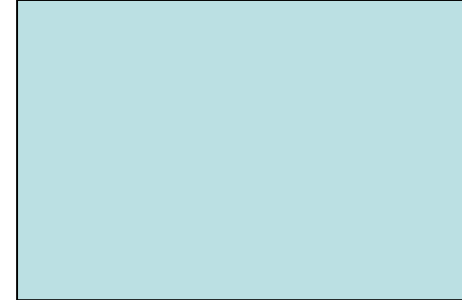
```
void remv (listaDeNodos node, int *listaDeNodosVazios, int p,
int q) {
    int r , r2;
    r2 = -1 ;
    r = node[p].point;
    while (r >= 0 && node[r].point != q) {
        r2 = r;
        r = node[r].next;
    }
    if (r >= 0) {
        /* r aponta para um arco de node[p] para node[q] */
        if (r2 < 0)
            node[p].point = node[r].next;
        else
            node[r2].next = node[r].next;
        freenode(listaDeNodosVazios, node, r);
        return;
    } /* na inexistência de um arco, nenhuma ação precisa ser tomada */
}
```



```
void remv (nodetype *p, nodetype *q) {
    nodetype *r , *r2;
    r2 = NULL ;
    r = p->point;
    while (r && r->point != q) {
        r2 = r;
        r = r->next;
    }
    if (r) {
        if (!r2)
            p->point = r->next;
        else
            r2->next = r->next;
        free(r);
        return;
    }
}
```



```
void remvwt (listaDeNodos node, int *listaDeNodosVazios, int p,
int q) {
    int r , r2;
    r2 = -1 ;
    r = node[p].point;
    while (r >= 0 && node[r].point != q) {
        r2 = r;
        r = node[r].next;
    }
    if (r >= 0) {
        /* r aponta para um arco de node[p] para node[q] */
        if (r2 < 0)
            node[p].point = node[r].next;
        else
            node[r2].next = node[r].next;
        freenode(listaDeNodosVazios, node, r);
    }
    /*na inexistência de um arco, nenhuma ação precisa ser tomada*/
}
```



```
void remvwt (nodetype *p, nodetype *q) {
    nodetype *r , *r2;
    r2 = NULL ;
    r = p->point;
    while (r && r->point != q) {
        r2 = r;
        r = r->next;
    }
    if (r) {
        if (!r2)
            p->point = r->next;
        else
            r2->next = r->next;
        free(r);
        return;
    }
}
```



```
char adjacent (listaDeNodos node, int p, int q) {
    int r;
    r = node[p].point;
    while (r >= 0)
        if (node[r].point == q)
            return(1);
        else
            r = node[r].next ;
    return (0);
}
```

```
char adjacent (nodetype *p, nodetype *q) {
    nodetype *r;
    r = p->point;
    while (r)
        if (r->point == q)
            return(1);
        else
            r = r->next ;
    return (0);
}
```

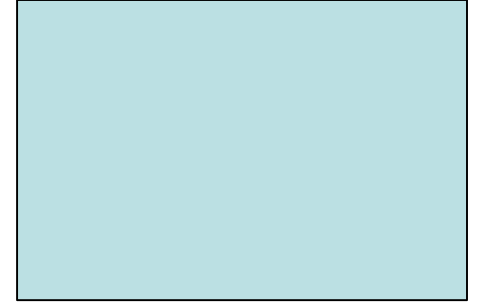


```
int findnode (listaDeNodos node, int graph, int x) {  
    int p;  
    p = graph;  
    while (p >= 0)  
        if (node[p].info == x)  
            return(p);  
        else  
            p = node[p].next;  
    return (-1);  
}
```

```
nodetype *findnode (TADgraph graph, int x) {  
    nodetype * p;  
    p = graph;  
    while (p)  
        if (p->info == x)  
            return(p);  
        else  
            p = p->next;  
    return (NULL);  
}
```



```
int addnode (listaDeNodos node,  
int *listaDeNodosVazios, int* pgraph, int x) {  
    int p;  
  
    p = getnode(listaDeNodosVazios, node);  
  
    node[p].info = x;  
  
    node[p].point = -1;  
  
    node[p].next = *pgraph;  
  
    *pgraph = p;  
  
    return (p);  
}
```



```
nodetype *addnode (TADgraph* pgraph, int x) {  
    TADgraph p;  
    if (!(p = (TADgraph) malloc (sizeof(nodetype))))  
        exit(1);  
    p->info = x;  
    p->point = NULL;  
    p->next = *pgraph;  
    *pgraph = p;  
    return (p);  
}
```



```
int remvnode(int *listaDeNodosVazios, listaDeNodos node,  
int *graph, int p) {
```

```
    int i;
```

```
    int nodoAtual, nodoAnterior, retorno = 0;
```

```
    nodoAtual = nodoAnterior = *graph;
```

```
    while (nodoAtual >= 0) {
```

```
        if (nodoAtual == p)
```

```
            /*remover todas as arestas com origem em vértice e o próprio vértice*/
```

```
            int nodoAux, nodoAux2;
```

```
            if (nodoAtual == *graph) /*eliminado vértice do conjunto de vértices*/
```

```
                *graph = node[*graph].next;
```

```
            else
```

```
                node[nodoAnterior].next = node[nodoAtual].next;
```

```
            nodoAux = node[nodoAtual].point;
```

```
            while (nodoAux >= 0) {
```

```
                nodoAux2 = nodoAux;
```

```
                nodoAux = node[nodoAux].next;
```

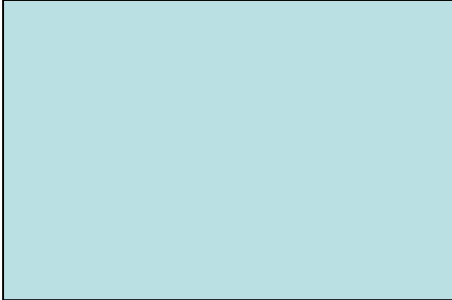
```
                freenode(listaDeNodosVazios, node, nodoAux2);/*eliminado arestas*/
```

```
            }
```



```
nodoAnterior = nodoAtual;
nodoAtual = node[nodoAtual].next;
freenode(listaDeNodosVazios, node, p); /*eliminando vértice*/
retorno = 1;
}
else
{ /*remover as arestas que levam ao vértice sendo removido, caso existam*/
int auxAnterior, auxAtual, nodeAux;
auxAnterior = auxAtual = node[nodoAtual].point;
while (auxAtual >=0) {
if (node[auxAtual].point == p) {
if (auxAtual == auxAnterior)
node[nodoAtual].point = node[auxAtual].next;
else
node[auxAnterior].next = node[auxAtual].next;
nodeAux = auxAtual;
auxAnterior = auxAtual;
auxAtual = node[auxAtual].next;
freenode(listaDeNodosVazios, node, nodeAux);
}
}
```





```
else {  
    auxAnterior = auxAtual;  
    auxAtual = node[auxAtual].next;  
}  
}  
nodoAnterior = nodoAtual;  
nodoAtual = node[nodoAtual].next;  
}  
}  
return (retorno);  
}
```

```
int remvnode(TADgraph *graph, nodetype *p) {
    int retorno = 0;
    nodetype *nodoAtual, *nodoAnterior;
    nodoAtual = nodoAnterior = *graph;
    while (nodoAtual) {
        if (nodoAtual == p)
            /*remover todas as arestas com origem em vértice e o próprio vértice*/
            nodetype *nodoAux, *nodoAux2;
            if (nodoAtual == *graph) /*eliminado vértice do conjunto de vértices*/
                *graph = (*graph)->next;
            else
                nodoAnterior->next = nodoAtual->next;
            nodoAux = nodoAtual->point;
            while (nodoAux) {
                nodoAux2 = nodoAux;
                nodoAux = nodoAux->next;
                free(nodoAux2);/*eliminado arestas*/
            }
    }
}
```



```
nodoAnterior = nodoAtual;  
nodoAtual = nodoAtual->next;  
free(p); /*eliminando vértice*/  
retorno = 1;
```

```
}
```

```
else
```

```
{/*remover as arestas que levam ao vértice sendo removido, caso existam*/
```

```
  nodetype *auxAnterior, *auxAtual, *nodeAux;
```

```
  auxAnterior = auxAtual = nodoAtual->point;
```

```
  while (auxAtual) {
```

```
    if (auxAtual->point == p) {
```

```
      if (auxAtual == auxAnterior)
```

```
        nodoAtual->point = auxAtual->next;
```

```
      else
```

```
        auxAnterior->next = auxAtual->next;
```

```
      nodeAux = auxAtual;
```

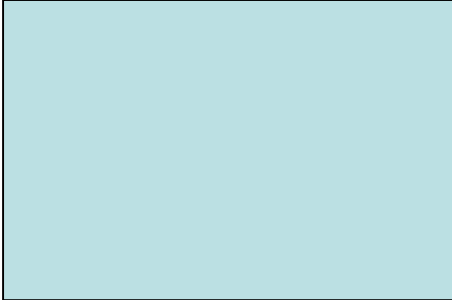
```
      auxAnterior = auxAtual;
```

```
      auxAtual = auxAtual->next;
```

```
      free(nodeAux);
```

```
    }
```





```
else {  
    auxAnterior = auxAtual;  
    auxAtual = auxAtual->next;  
}  
}  
nodoAnterior = nodoAtual;  
nodoAtual = nodoAtual->next;  
}  
}  
return (retorno);  
}
```

## Grafos – Busca em largura

A busca em largura é um dos algoritmos mais simples para se pesquisar um grafo e é arquétipo de muitos algoritmos de grafos importantes.

O algoritmo de caminhos mais curtos de origem única de Dijkstra e o algoritmo de árvore espalhada mínima de Prim utilizam ideias semelhantes às que aparecem na busca em largura.

A busca em largura baseia-se em partindo de um determinado vértice  $s$  (origem) explorar sistematicamente as arestas do grafo determinando cada vértice acessível a partir de  $s$ .

## Grafos – Busca em largura

O algoritmo calcula a distância (menor número de arestas) desde  $s$  até todos os vértices acessíveis partindo do mesmo.

Ele também produz uma "árvore primeiro na extensão" com raiz  $s$  que contém todos os vértices acessíveis. Para qualquer vértice acessível  $v$  a partir de  $s$ , o caminho na árvore primeiro na extensão de  $s$  até  $v$  corresponde a um "caminho mais curto" de  $s$  até  $v$  em  $G$  (grafo), ou seja, um caminho que contém o número mínimo de arestas.

O algoritmo funciona sobre grafos orientados e também não orientados.

## Grafos – Busca em largura

Para controlar o andamento, a busca em largura pinta cada vértice de branco, cinza ou preto.

No início, todos os vértices são brancos, e mais tarde eles podem se tornar acinzentadas e depois pretos.

Um vértice é descoberto na primeira vez em que é encontrado durante a pesquisa, e nesse momento ele se torna não branco. Portanto, vértices em cor cinza e preta foram descobertos, mas a busca em largura faz distinção entre eles para assegurar que a pesquisa continuará de maneira a seguir primeiro na extensão.

## Grafos – Busca em largura

Considerando um grafo  $G = (V, E)$ . Se  $(u, v) \in E$  e o vértice  $u$  é preto, então o vértice  $v$  é cinza ou preto; isto é, todos os vértices adjacentes a vértices pretos foram descobertos. Vértices de cor cinza podem ter alguns vértices adjacentes brancos; eles representam a fronteira entre vértices descobertos e não descobertos.

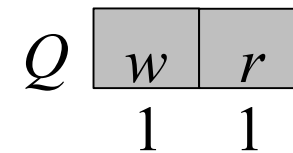
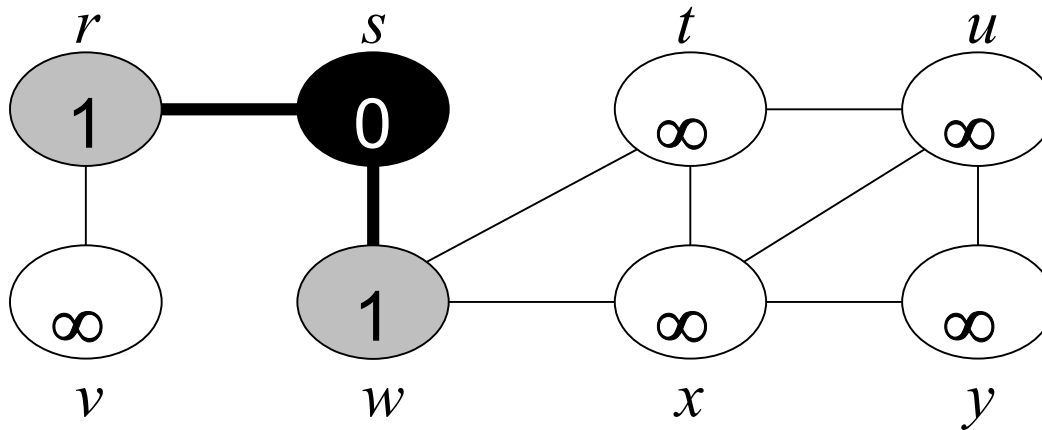
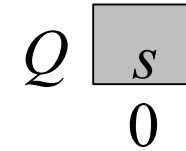
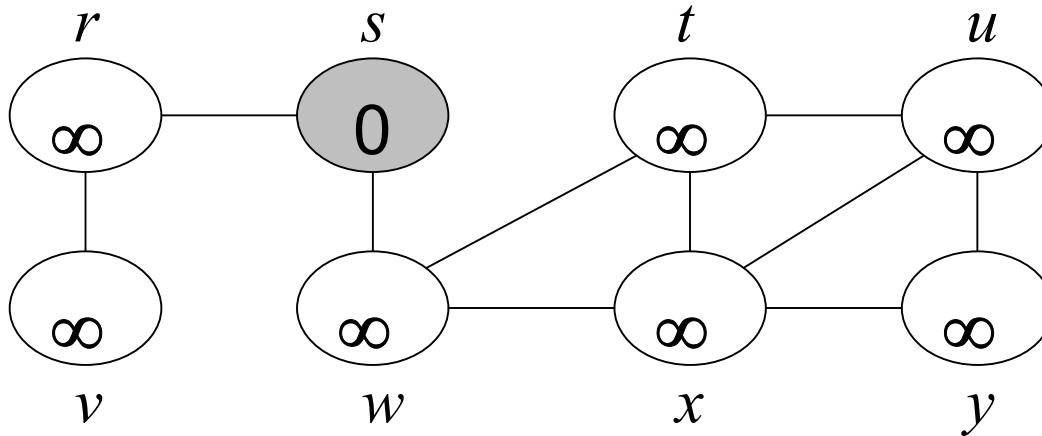
Sendo assim, a busca em largura constrói uma árvore primeiro na extensão, contendo inicialmente apenas sua raiz, a qual é o vértice de origem  $s$ . Sempre que um vértice branco  $v$  é descoberto no curso da varredura da lista de adjacências de um vértice  $u$  já descoberto, o vértice  $v$  e a aresta  $(u, v)$  são adicionados à árvore.

## Grafos – Busca em largura

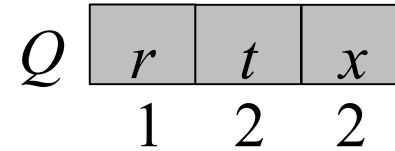
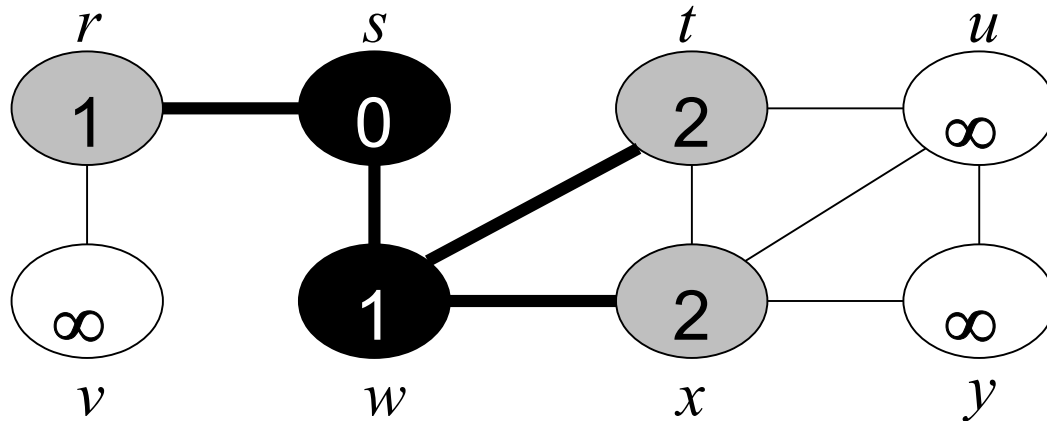
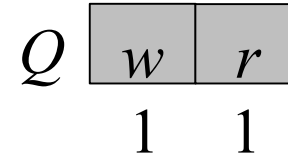
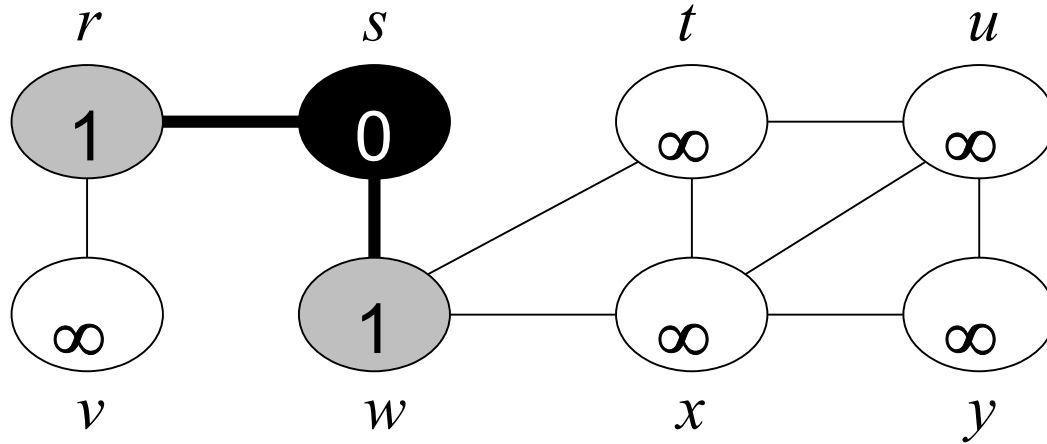
Dizemos que  $u$  é o predecessor ou pai de  $v$  na árvore primeiro na extensão. Tendo em vista que um vértice é descoberto no máximo uma vez, ele tem no máximo um pai. Relacionamentos de ancestral e descendente na árvore primeiro na extensão são definidos em relação à raiz  $s$  da maneira usual: se  $u$  está em um caminho na árvore a partir da raiz  $s$  até o vértice  $v$ , então  $u$  é um ancestral de  $v$ , e  $v$  é um descendente de  $u$ .

Para uma melhor compreensão vamos observar como ocorre este processo no grafo apresentado no próximo slide.

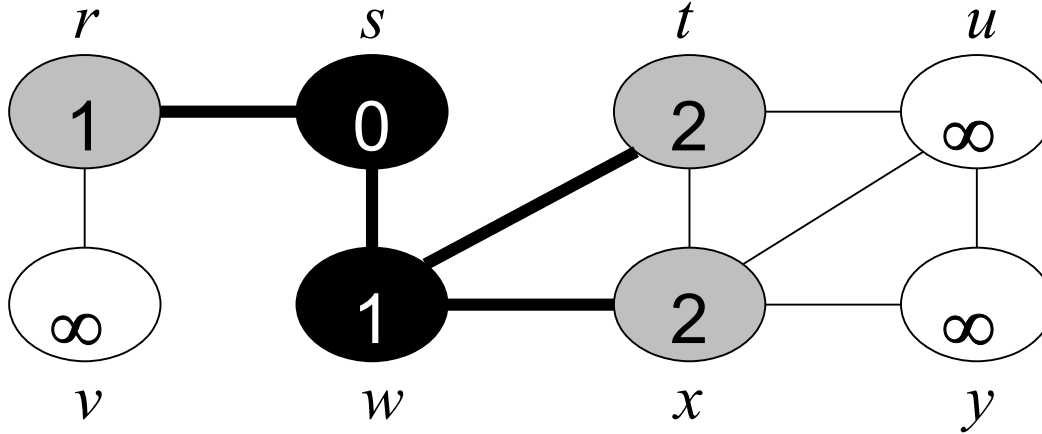
# Grafos – Busca em largura



# Grafos – Busca em largura

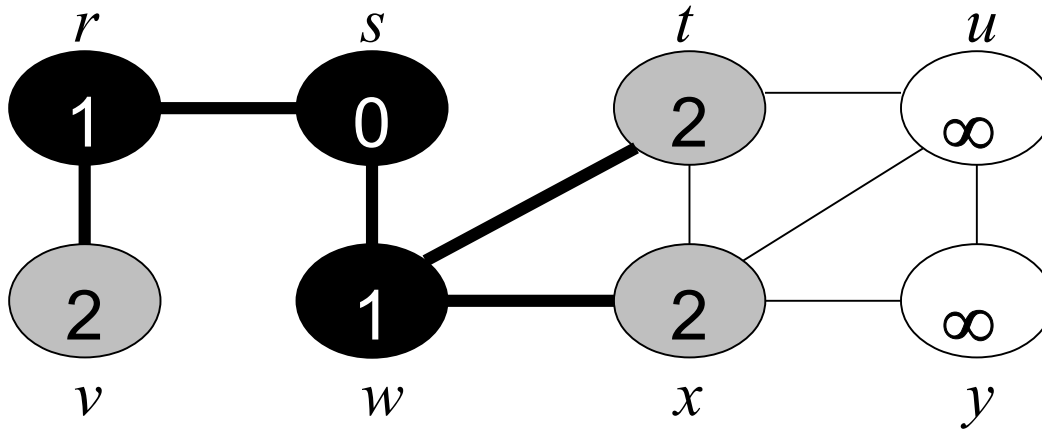


# Grafos – Busca em largura



$Q$

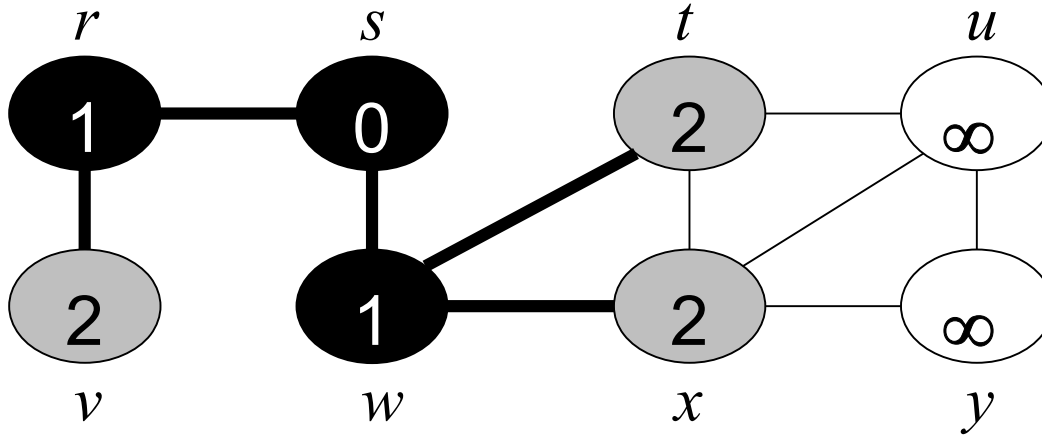
$r$	$t$	$x$
1	2	2



$Q$

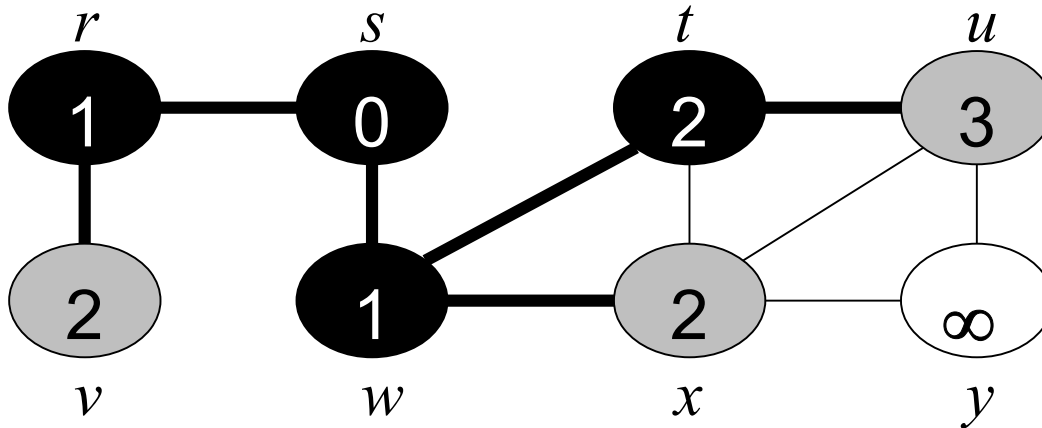
$t$	$x$	$v$
2	2	2

# Grafos – Busca em largura



$Q$

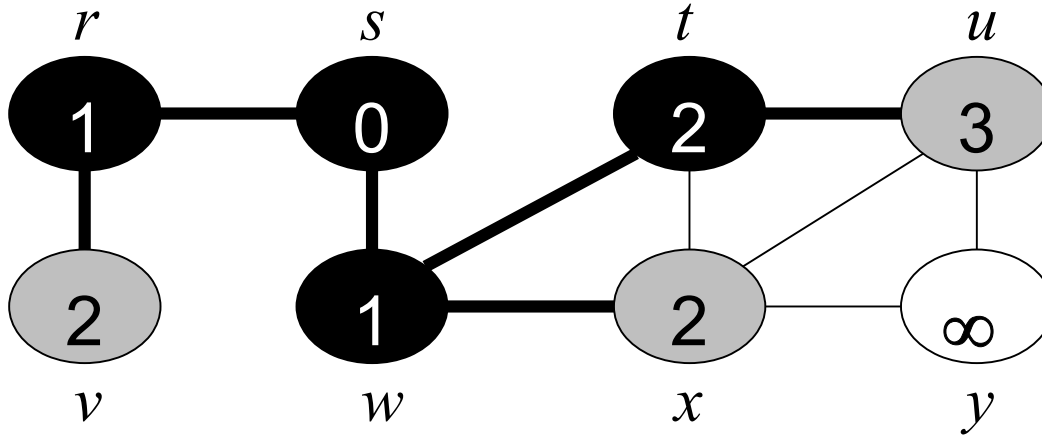
$t$	$x$	$v$
2	2	2



$Q$

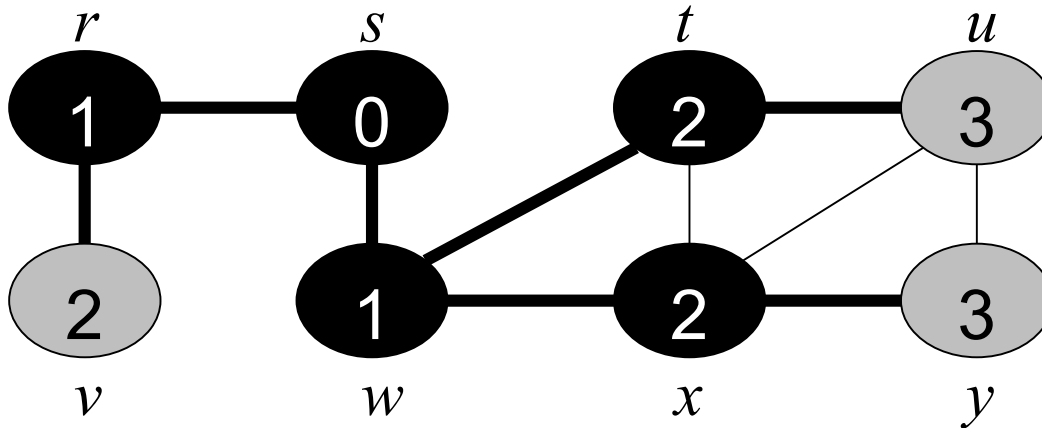
$x$	$v$	$u$
2	2	3

# Grafos – Busca em largura



$Q$

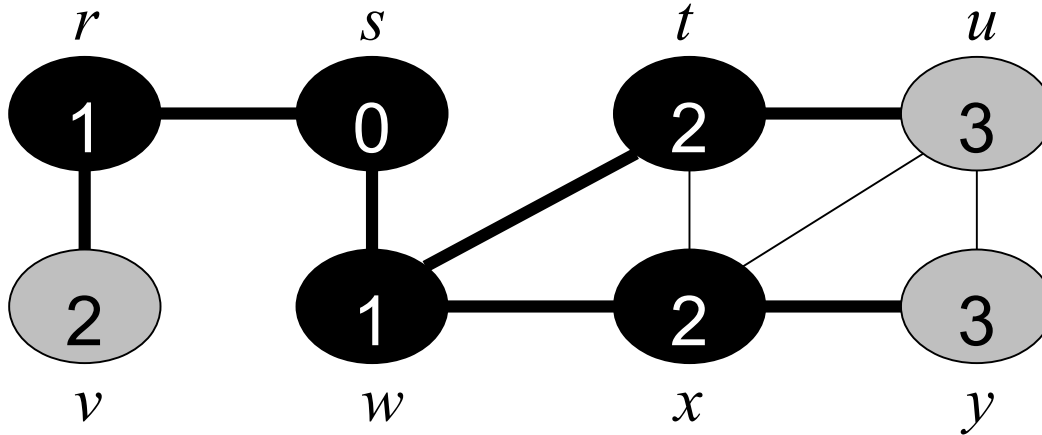
$x$	$v$	$u$
2	2	3



$Q$

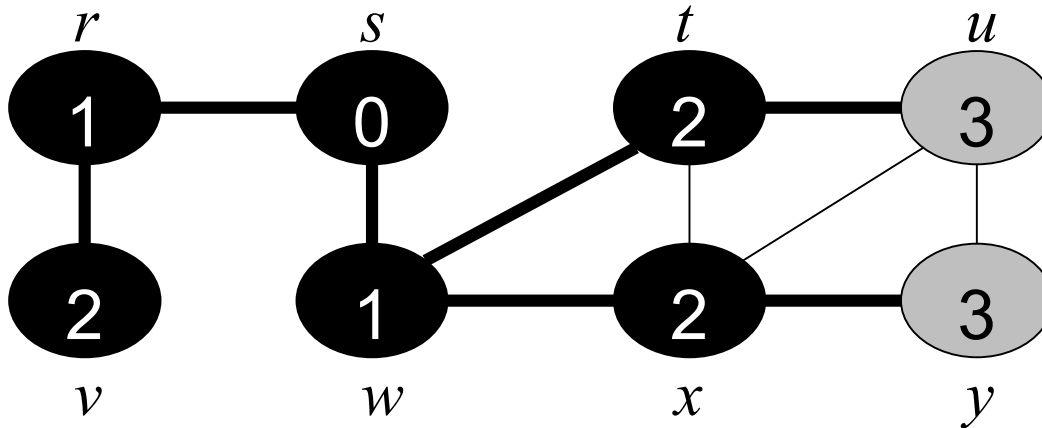
$v$	$u$	$y$
2	3	3

# Grafos – Busca em largura



$Q$

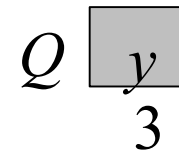
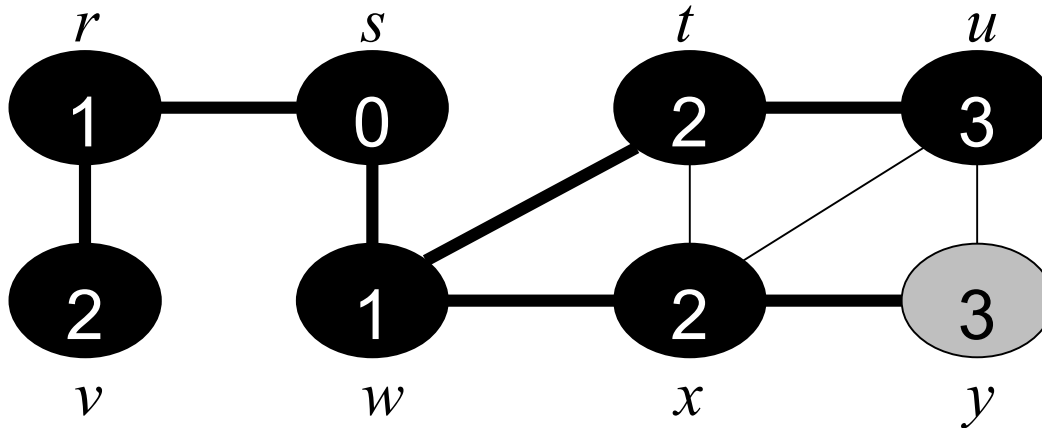
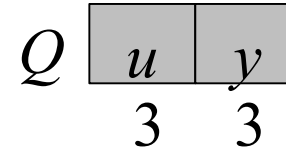
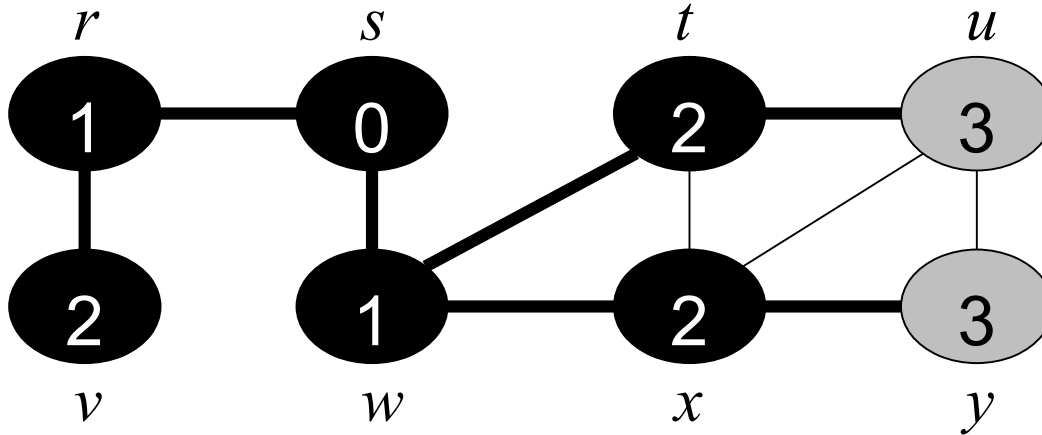
$v$	$u$	$y$
2	3	3



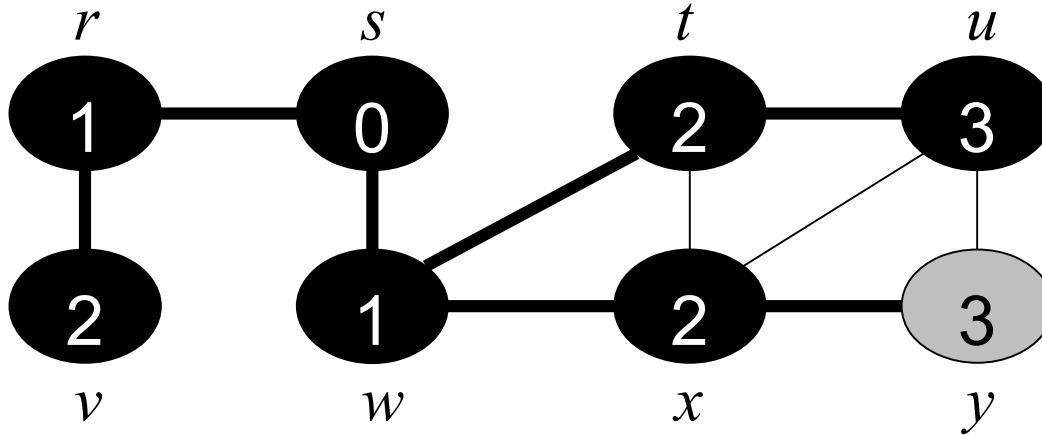
$Q$

$u$	$y$
3	3

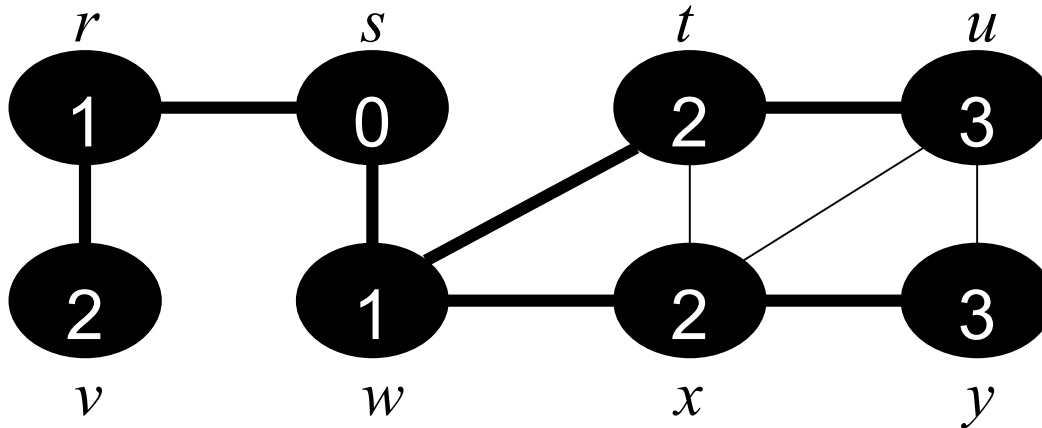
# Grafos – Busca em largura



# Grafos – Busca em largura



$Q$   $\begin{matrix} y \\ 3 \end{matrix}$



$Q$   $\emptyset$

## Grafos – Busca em largura

Com base no que foi estudado, codifique uma função na linguagem C que implemente o procedimento de busca em largura discutido, pressuponha que o grafo de entrada  $\mathbf{G} = (\mathbf{V}, \mathbf{E})$  é representado com o uso de listas de adjacências (representação encadeada em vetor).

**Dica:** Mantenha estruturas de dados adicionais para armazenar informações relativas a cada vértice no grafo. A cor de cada vértice  $u \in \mathbf{V}$  pode ser armazenada na variável  $cor[u]$ , e o predecessor de  $u$  pode ser armazenado na variável  $pai[u]$ . Se  $u$  não tem nenhum predecessor (por exemplo, se  $u = s$  ou se  $u$  não foi descoberto), então  $pai[u] = \text{NULL}$ . A distância desde a origem  $s$  até o vértice  $u$  calculada pelo algoritmo pode ser armazenada em  $d[u]$ . Para facilitar a implementação o algoritmo também pode se utilizar de uma fila  $Q$  (FIFO) para gerenciar o conjunto de vértices de cor cinza.

**BFS(G, s) /\*Procedimento apresentado no livro Algoritmos, Cormen et al\*/**

```
1  for cada vértice  $u \in V[G] - \{s\}$ 
2    do cor[u] <- BRANCO
3      d[u] <-  $\infty$ 
4      pai[u] <- NULL
5  cor[s] <- CINZA
6  d[s] <- 0
7  pai[s] <- NULL
8  Q <-  $\emptyset$ 
9  ENQUEUE(Q, s) /*Insere s na fila Q*/
10 while Q  $\neq \emptyset$ 
11do u <- DEQUEUE(Q) /*Consulta e remove o primeiro elemento da fila Q*/
12  for cada v <- Adj[u]
13    do if cor[v] = BRANCO
14      then cor[v] <-CINZA
15          d[v] <- d[u] + 1
16          pai[v] <- u
17          ENQUEUE (Q, v)
18  cor[u] <- PRETO
```

```
/*TAD FILA*/  
typedef struct {  
    int indMemoria;  
    int indInf;  
}DADOS;  
typedef struct nodo {  
    DADOS inf;  
    struct nodo * next;  
}NODO;  
typedef struct {  
    NODO *INICIO;  
    NODO *FIM;  
}DESCRITOR;  
typedef DESCRITOR * FILA_ENC;  
void cria_fila (FILA_ENC *);  
int eh_vazia (FILA_ENC);  
void ins (FILA_ENC, DADOS);  
DADOS cons (FILA_ENC);  
void ret (FILA_ENC);  
DADOS cons_ret (FILA_ENC);
```



```
void buscaEmLargura(listaDeNodos node, int G, int s) {
    int u, *d=NULL, *pai=NULL, *vertice=NULL, numVertices=0, v, ind;
    char *cor=NULL; /*'B'=Branco, 'C'=Cinza e 'P'=Preto*/
    FILA_ENC Q;
    DADOS aux;
    u = G;
    while (u>=0) {
        ++numVertices;
        d = (int *) realloc(d, numVertices*sizeof(int));
        pai = (int *) realloc(pai, numVertices*sizeof(int));
        cor = (char *) realloc(cor, numVertices*sizeof(char));
        vertice = (int *) realloc(vertice, numVertices*sizeof(int));
        /*apesar de suprimida é necessária a verificação das alocações dinâmicas*/
        if (u != s) {
            cor[numVertices-1] = 'B';
            d[numVertices-1] = -1; /*-1 equivale a infinito*/
            pai[numVertices-1] = -1; /*-1 equivale a NULL*/
            vertice[numVertices-1] = node[u].info;
        }
    }
}
```

```
else {
    ind = numVertices-1;
    cor[numVertices-1] = 'C';
    d[numVertices-1] = 0;
    pai[numVertices-1] = -1; /*-1 equivale a NULL*/
    vertice[numVertices-1] = node[u].info;
}
u = node[u].next;
}
cria_fila(&Q);
aux.indInf = s;
aux.indMemoria = ind;
ins(Q, aux); /*Insere s na fila Q*/
while (!eh_vazia(Q))
{
    aux = cons_ret(Q); /*Consulta e remove o primeiro elemento da fila Q*/
    u = aux.indInf;
    v = G;
    ind = -1;
}
```



```
while (v>=0) {  
    ind++;  
    if (adjacent (node, u, v)) {  
        if (cor[ind] == 'B')  
        {  
            DADOS aux2;  
            cor[ind] = 'C';  
            d[ind] = d[aux.indMemoria] + 1;  
            pai[ind] = u;  
            aux2.indInf = v;  
            aux2.indMemoria = ind;  
            ins (Q, aux2);  
        }  
    }  
    v = node[v].next;  
}  
cor[aux.indMemoria] = 'P';  
}  
}
```



```
typedef struct nodetype {  
    int info;  
    int point ;  
    int next;  
    /*Dados necessários para a busca em largura*/  
    char cor; /*'B'=Branco, 'C'=Cinza e 'P'=Preto*/  
    int d;  
    int pai;  
    /***/  
}tipoNodo;  
typedef tipoNodo listaDeNodos[MAXNODES];
```



```
/*TAD FILA*/  
typedef struct nodo  
{  
    int inf;  
    struct nodo * next;  
}NODO;  
typedef struct  
{  
    NODO *INICIO;  
    NODO *FIM;  
}DESCRITOR;  
typedef DESCRITOR * FILA_ENC;  
void cria_fila (FILA_ENC *);  
int eh_vazia (FILA_ENC);  
void ins (FILA_ENC, int);  
int cons (FILA_ENC);  
void ret (FILA_ENC);  
int cons_ret (FILA_ENC);
```



```
void buscaEmLargura(listaDeNodos node, int G, int s) {
    int u, v, ind;
    FILA_ENC Q;
    u = G;
    while (u >= 0) {
        if (u != s)
        {
            node[u].cor = 'B';
            node[u].d = -1; /*-1 equivale a infinito*/
            node[u].pai = -1; /*-1 equivale a NULL*/
        }
        else
        {
            node[u].cor = 'C';
            node[u].d = 0;
            node[u].pai = -1; /*-1 equivale a NULL*/
        }
        u = node[u].next;
    }
}
```



```
cria_fil(&Q);
ins(Q, s); /*Insere s na fila Q*/
while (!eh_vazia(Q)) {
    u = cons_ret(Q); /*Consulta e remove o primeiro elemento da fila Q*/
    v = node[u].point;
    while (v>=0)
    {
        if (node[node[v].point].cor == 'B')
        {
            node[node[v].point].cor = 'C';
            node[node[v].point].d = node[u].d + 1;
            node[node[v].point].pai = u;
            ins (Q, node[v].point);
        }
        v = node[v].next;
    }
    node[u].cor = 'P';
}
}
```



# Grafos – Busca em largura

Como determinar o caminho mínimo (menor caminho) entre um vértice A e B?

Com a busca em largura!

Como?

Conforme foi anteriormente mencionado, o procedimento de busca em largura constrói uma árvore primeiro na extensão à medida que pesquisa o grafo. A árvore é representada pelo campo **pai** em cada vértice. Mais formalmente, para um grafo  $\mathbf{G} = (\mathbf{V}, \mathbf{E})$  com origem  $\mathbf{s}$ , definimos o **subgrafo predecessor** de  $\mathbf{G}$  como:

$\mathbf{G}_{\text{pai}} = (\mathbf{V}_{\text{pai}}, \mathbf{E}_{\text{pai}})$ , onde

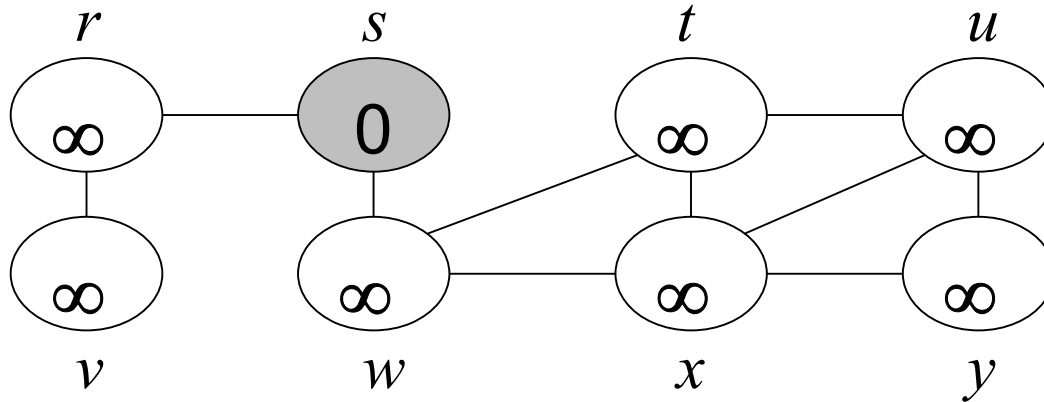
$\mathbf{V}_{\text{pai}} = \{v \in \mathbf{V} : \text{pai}[v] \neq \text{NULL}\} \cup \{\mathbf{s}\}$ .

e

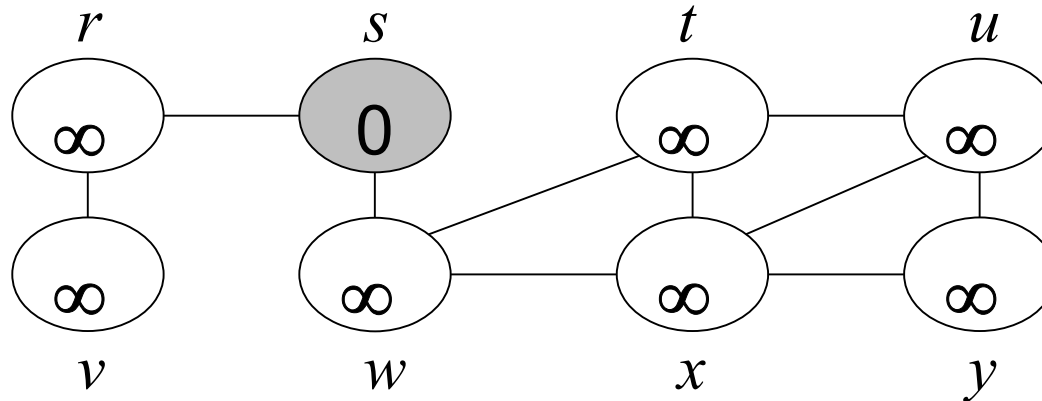
$\mathbf{E}_{\text{pai}} = \{ \{( \text{pai}[v], v) : v \in \mathbf{V}_{\text{pai}} - \{\mathbf{s}\} \}$ .

# Grafos – Busca em largura

O subgrafo predecessor  $G_{\text{pai}}$  é uma **árvore primeiro na extensão** ( BFS Tree ) se  $V_{\text{pai}}$  consiste nos vértices acessíveis a partir de  $s$  e, para todo  $v \in V_{\text{pai}}$  existe um caminho único simples desde  $s$  até  $v$  em  $G_{\text{pai}}$  que também é um caminho mais curto de  $s$  até  $v$  em  $G$ . As arestas em  $E_{\text{pai}}$  são chamadas arestas da árvore. Observe o grafo abaixo:



# Grafos – Busca em largura



A aplicação do algoritmo da busca em profundidade gera as seguintes informações nos respectivos campos:

**Vértices:** { y, x, w, v, u, t, s, r }

**Distâncias:** { 3, 2, 1, 2, 3, 2, 0, 1 }

**Pais:** { x, w, s, r, t, w, , s }

## Grafos – Busca em largura

Com base no que foi apresentado implemente uma função, na linguagem C, que com base no resultado da busca em largura apresente o caminho mínimo de um vértice  $s$  para um vértice  $v$ .