

BFS nó inicial 0

processado	distância	antecessor
0	0	-1
1	1	0
3	1	0
2	2	1

BFS nó inicial 4

processado	distância	antecessor
4	0	-1
5	1	4

```

1 procedure BFS(G, start_v)
2   Seja Q uma queue
3   rotule start_v como cinza, dist = -1
4   Q.enqueue(start_v)
5   while Q não está vazia do
6     v := Q.dequeue(), dist = dist + 1
7     rotule v com preto
8     print(v, dist)
9     for all arestas (v, w) in G.adjacentEdges(v) do
10      if w não está rotulado como cinza then
11        rotule w como cinza
12        Q.enqueue(w)
  
```

Estrutura de dados Grafo

```
#define MaxNumVertices 100
```

```
typedef int elem;
```

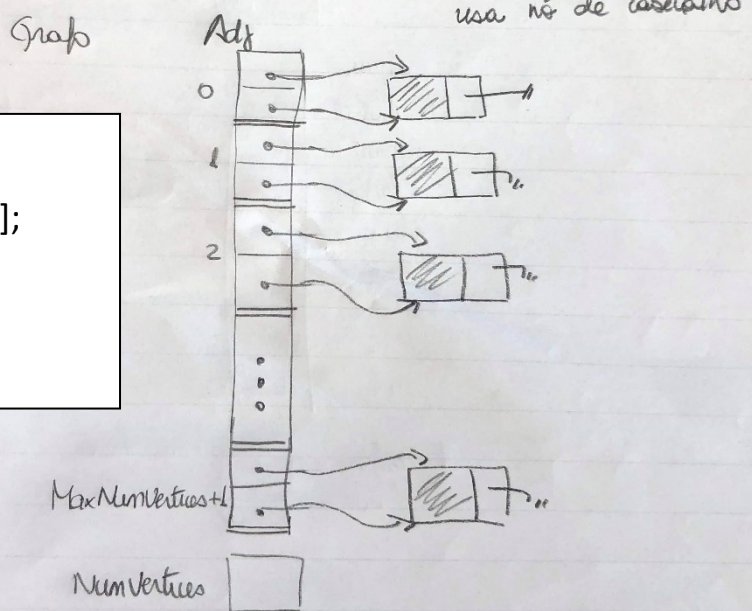
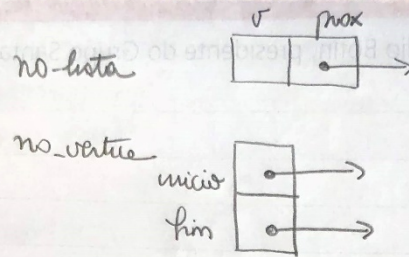
```
typedef enum {  
    branco, cinza, preto  
} TipoCor;
```

```
typedef struct no_lista {  
    elem v;  
    struct no_lista *prox;  
} no_lista;
```

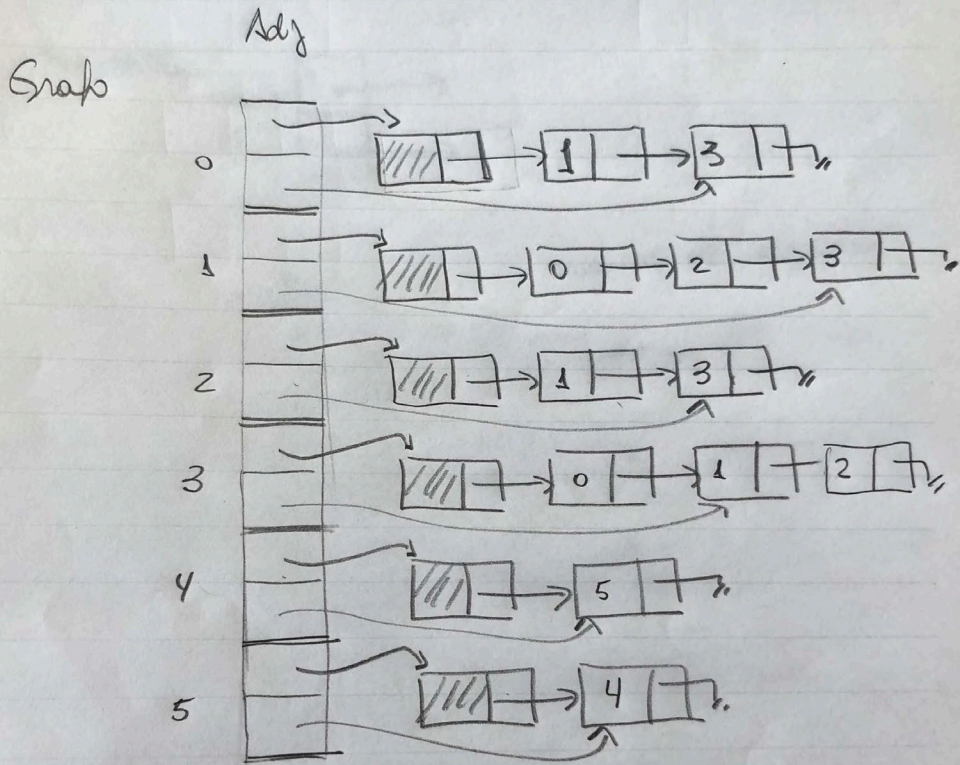
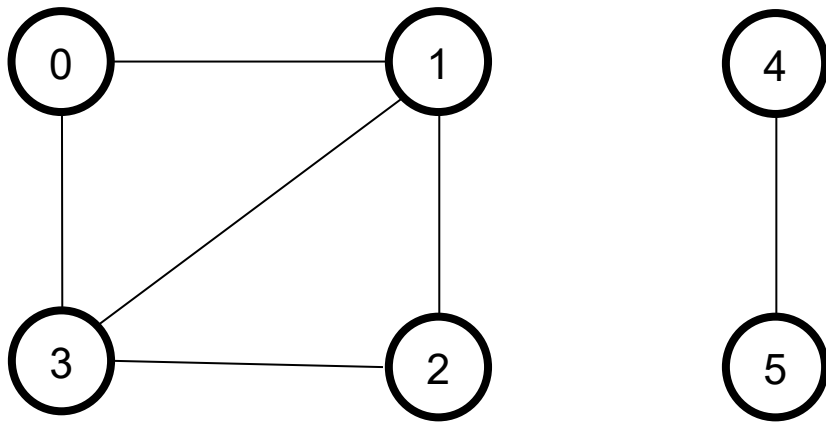
```
typedef struct {  
    no_lista *inicio, *fim;  
} no_vertice;
```

```
typedef struct {  
    no_vertice Adj[MaxNumVertices];  
    int NumVertices;  
} Grafo;
```

```
typedef struct no_lista {  
    elem v;  
    struct no_lista *prox;  
} no_lista;  
  
typedef struct {  
    no_lista *inicio, *fim;  
} no_vertice;
```



```
typedef struct {  
    no_vertice Adj[MaxNumVertices];  
    int NumVertices;  
} Grafo;
```



Number of vertices 6

Funções auxiliares:

Para percorrer a Lista de Adjacências de um vértice V

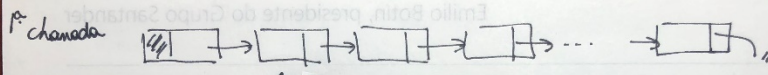
```
/* retorna o endereço do primeiro vértice na lista de adjacentes do vértice V */
```

```
no_aresta* PrimeiroListaAdj(Grafo *G, int *V, int *erro) {  
    if (*V >= G->NumVertices) {  
        *erro= 1;  
        return(NULL);  
    }  
    else {  
        *erro= 0;  
        return(G->Adj[*V].inicio->prox);  
    }  
}
```

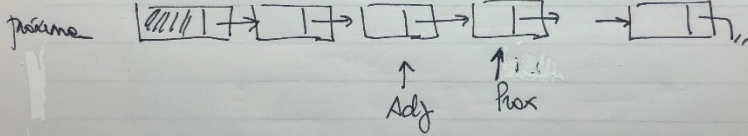
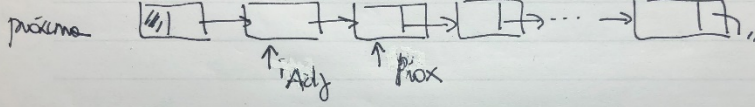
```
/* na lista de adjacentes de V: retorna o vértice atualmente apontado por Prox (retorna na variável Adj) e  
retorna também Prox já posicionado no próximo vértice da lista; FimListaAdj retorna 1 se chegou no final  
da lista */
```

```
void ProxAdj(Grafo *G, no_aresta **Adj, no_aresta **Prox, int *FimListaAdj) {  
    *Adj= *Prox;  
    *Prox= (*Prox)->prox;  
  
    if (*Prox == NULL)  
        *FimListaAdj= 1;  
}
```

void ProxAdj (Grafo *G, no_aresta **Adj,
no_aresta **Prox, int *FimAdj)

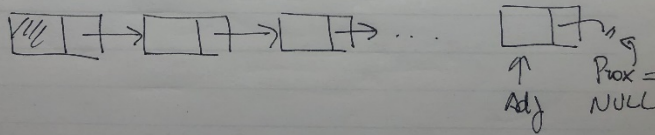


Adj?



:

ultima



TAD Fila

```
#define TamFila 100
```

...

```
void Create(Fila*);
```

```
void Empty(Fila*);
```

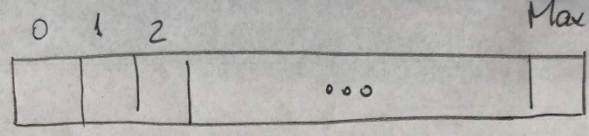
```
int IsEmpty(Fila*);
```

```
int IsFull(Fila*);
```

```
void Entra(Fila*, elemF*, int*);
```

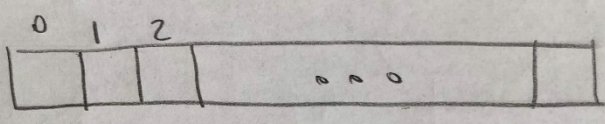
```
void Sai(Fila*, elemF*, int*);
```

distancia:

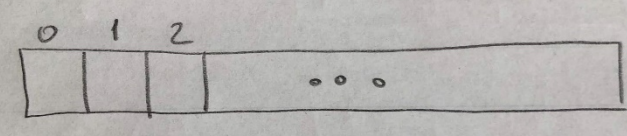


Max Num Vertices

antecessor



tipo Cor



```
/* função para busca em largura, utiliza a função visita_bfs */
```

```
void busca_largura(Grafo *G) {
    int V, distancia[MaxNumVertices+1], antecessor[MaxNumVertices];
    TipoCor cor[MaxNumVertices];

    printf("*** Sequencia de nos visitados na busca em largura ***\n\n");

    for (V= 1; V <= G->NumVertices; V++) {
        cor[V]= branco;
        distancia[V]= -1;
        antecessor[V]= -1;
    }

    for (V= 1; V <= G->NumVertices; V++)
        if (cor[V] == branco)
            visita_bfs(G, V, distancia, cor, antecessor);
}

void visita_bfs(Grafo *G, int V, int distancia[], TipoCor cor[], int antecessor[]) {
    int FimListaAdj, erro;
    no_lista *Adj, *Aux;

    Fila F;
    Create(&F);

    cor[V]= cinza;
    distancia[V]= 0;
    Entra-Fila(&F, &V, &erro);
    printf("No %d, distancia = %d, antecessor = %d\n", V, distancia[V], antecessor[V]);

    while (!IsEmpty(&F)) {
        Sai-Fila(&F, &V, &erro);
        if (!ListaAdjVazia(G, V, &erro)) {
            Aux= PrimeiroListaAdj(G, V, &erro);
            FimListaAdj= 0;
            while (!FimListaAdj) {
                ProxAdj(G, &Adj, &Aux, &FimListaAdj);
                if (cor[Adj->v] == branco) {
                    cor[Adj->v]= cinza;
                    distancia[Adj->v]= distancia[V]+1;
                    antecessor[Adj->v]= V;
                    Entra-Fila(&F, &Adj->v, &erro);
                    printf("No %d, distancia=%d, antecessor=%d\n", Adj->v, distancia[Adj->v], antecessor[Adj->v]);
                }
            }
        }
        cor[V]= preto;
    }
}
```