Grafos – Representação

Implemente a operação **join()** para um grafo não ponderado de forma similar à anterior.

```
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;
  }
```

UNIVASF

Grafos – Representação

```
  r = getnode(listaDeNodosVazios, node);
  node[r].point = q;
  node[r].next = -1;
  (r2 < 0 ) ? (node[p].point = r) : (node[r2].next = r);
}
```

Grafos – Representação

Implemente a operação **remv()** que aceita ponteiros para dois nós de cabeçalho e remove o arco entre eles, caso este exista.

Observação: Considere a existência de uma função freenode().

UNIVASF

```
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 */
}
```

UNIVASF

Grafos – Representação

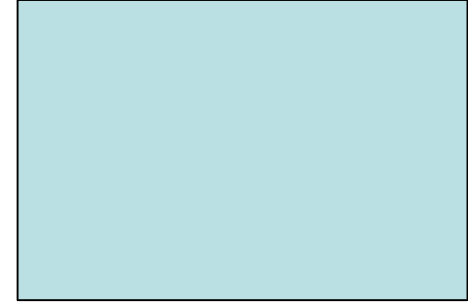
Implemente a operação **freenode()**.

```
void freenode (int *listaDeNodosVazios, listaDeNodos node, int r)
{
  node[r].next = *listaDeNodosVazios;
  *listaDeNodosVazios = r;
}
```

UNIVASF

Grafos – Representação

Implemente a operação **remvwt (node, listaDeNodosVazios, p, q, x)** que define **x** com o peso do arco **<p,q>** num grafo ponderado e depois remove o arco do grafo, caso este exista.



```
void remvwt (listaDeNodos node, int *listaDeNodosVazios, int p,
int q, int x) {
   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;
      node[r].info = x;
      freenode(listaDeNodosVazios, node, r);
   }
   /*na inexistência de um arco, nenhuma ação precisa ser tomada*/
}
```

Grafos – Representação

Implemente a operação **adjacent (node, p, q)** a qual recebe ponteiros para dois nós de cabeçalho e determina se **node(q)** é adjacente a **node(p)**.

```
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);
}
```

Grafos – Representação

Implemente a operação **findnode(node, graph, x)** que retorna um ponteiro para um nó de cabeçalho com o campo de informação **x**, caso exista esse nó de cabeçalho, e retorna o ponteiro nulo (com valor -1), caso contrário.

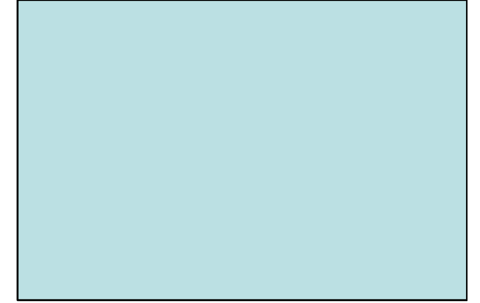
```
int findnode (listaDeNodos node, int graph, int x) {
   int p = graph;
   while (p >= 0)
      if (node[p].info == x)
         return(p);
      else
         p = node[p].next;
   return (-1);
}
```

UNIVASF

Grafos – Representação

Implemente a operação **addnode (node, &listaDeNodosVazios, &pgraph, x)** inclui um nó com o campo de informação **x** num grafo e retorna um ponteiro para esse nó.

```
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);
}
```



Grafos – Representação

É importante salientar outra diferença relevante entre a representação de matriz de adjacência e a representação ligada de grafos.

Na representação de matriz está implícita a possibilidade de percorrer uma linha ou coluna da matriz. Percorrer uma linha equivale a identificar todos os arcos emanando de determinado nó. Isso pode ser feito com eficiência na representação ligada, percorrendo a lista de nós de arco, a partir de determinado nó de cabeçalho.

Entretanto, percorrer uma coluna de uma matriz de adjacência é equivalente a identificar todos os arcos que terminam em determinado nó;

UNIVASF

Grafos – Representação

não existe um método correspondente para fazer isso sob a representação ligada que trabalhamos.

A representação ligada poderia ser alterada de modo a incluir duas listas emanando de cada nó de cabeçalho: uma para os arcos emanando do nó de grafo e outra para os arcos terminando no nó de grafo.

Entretanto, isso exigiria a alocação de dois nós para cada arco, aumentando, por conseguinte, a complexidade da inclusão ou eliminação de um arco.

Como alternativa, cada nó de arco poderia ser colocado em duas listas.

UNIVASF

Grafos – Representação

Nesse caso, um nó de arco conteria quatro ponteiros: um para o próximo arco emanando do mesmo nó, um para o próximo arco terminando no mesmo nó, um para o nó de cabeçalho no qual ele termina e um para o nó de cabeçalho a partir do qual ele emana. Um nó de cabeçalho conteria três ponteiros: um para o próximo nó de cabeçalho, um para a lista de arcos emanando a partir dele e outro para a lista de arcos terminando nesse nó.

Evidentemente, o programador precisará escolher entre essas representações examinando as necessidades do problema específico e considerando a eficiência em termos de tempo e espaço de armazenamento.

UNIVASF

Grafos – Representação

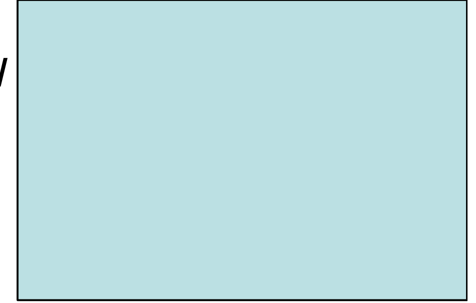
Implemente uma rotina **remvnode(...,&graph,p)** que remova um nó de cabeçalho apontado por **p** de um grafo apontado por **graph**, usando a representação ligada em vetor.

Evidentemente, quando um nó é removido de um grafo, todos os arcos emanando e terminando nesse nó precisarão ser removidos também.

Na representação ligada que estamos utilizando, não existe um método fácil de remover um nó de um grafo, porque os arcos terminando no nó não podem ser obtidos diretamente.

UNIVASF

```
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 no vértice e o próprio vértice*/
      int nodoAux, nodoAux2;
      if (nodoAtual == *graph) /*eliminado o 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);
}
```

Grafos – Representação

Em resumo, implementamos o TAD grafo com re-presentação encadeada em vetor com a estrutura e operações a seguir:

```
#define MAXNODES 500
typedef struct nodetype {
  int info;
  int point ;
  int next;
}tipoNodo;
typedef tipoNodo listaDeNodos[MAXNODES];
void inicializarGrafo (int *graph);
void criaListaDeNodosVazios (int *listaDeNodosVazios, listaDeNodos node);
int getnode (int *listaDeNodosVazios, listaDeNodos node);
void freenode (int *listaDeNodosVazios, listaDeNodos node, int r);
```

UNIVASF

Grafos – Representação

```
void joinwt (listaDeNodos node,
int *listaDeNodosVazios, int p, int q, int wt);
void join (listaDeNodos node, int *listaDeNodosVazios, int p, int q);
void remv (listaDeNodos node, int *listaDeNodosVazios, int p, int q);
void remvwt (listaDeNodos node, int *listaDeNodosVazios,
int p, int q, int x);
char adjacent (listaDeNodos node, int p, int q);
int findnode (listaDeNodos node, int graph, int x);
int addnode (listaDeNodos node, int *listaDeNodosVazios,
int* pgraph, int x);
int remvnode(int *listaDeNodosVazios, listaDeNodos node,
int *graph, int p);
```

UNIVASF

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;
```

UNIVASF

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

UNIVASF