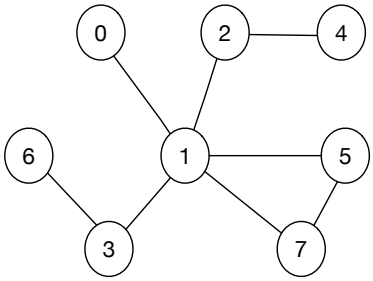
PUC-Rio – Estruturas de Dados – INF1010
Prova 3 – Turma 3wb – 29/11/2017

Responda as questões da prova nas folhas em separado distribuídas junto com a prova. As respostas podem ser escritas a lápis ou caneta. Indique claramente a questão que está respondendo e, quando cabível, os passos que seguiu para chegar a sua resposta.

Por favor guarde seu celular/tablet/o_que_for antes de começar a prova e não o utilize até sair da sala.

1. (1 ponto) Considere o grafo a seguir:



   Diga qual seria uma possível ordem de visitação a partir do nó 2 com cada uma das técnicas:

   (a) percurso em profundidade

   (b) percurso em largura

2. (2 pontos) Considere a implementação de tabelas hash explorada no laboratório 8, A estrutura de dados usada era a seguinte:

```
typedef struct smapa Mapa;
typedef struct {
  int chave;
  int dados;
  int prox;
} ttabpos;
struct smapa {
  int tam;
  ttabpos *tabpos;
};
```

   A implementação trata conflitos através de encadeamento interno: Cada posição da tabela contém, além da chave e dos dados, um campo prox, apontando para o índice da tabela contendo a próxima chave mapeada para o mesmo valor de hash que a chave na posição atual. Se não houver essa "próxima"chave, o campo prox deve conter o valor -1.

   Escreva uma função com a assinatura a seguir, que dada uma tabela hash determine qual o comprimento da maior cadeia de colisões contida na tabela. (Essa função seria parte do arquivo `mapa.c` e "enxerga" a implementação da tabela de dispersão.)

```
static int maiorcadeia (Mapa* m);
```

sugestão: Percorra a tabela e, para cada índice ocupado onde o hash da chave inserida coincida com o índice, verifique o tamanho da cadeia.

3. (2 pontos) Considere a representação de grafos vista nos nossos vários laboratórios sobre o assunto:

```
typedef struct _grafo Grafo;
typedef struct _viz Viz;
struct _viz {
  int noj;
  float peso;
  Viz*  prox;
};
struct _grafo {
  int nv;           /* numero de nos ou vertices */
  int na;           /* numero de arestas */
  Viz** viz;         /* viz[i] aponta para a lista de arestas vizinhas do no i */
};
```

Suponha que estamos tratando de grafos *direcionados*, isto é, as arestas têm nó origem e destino. Em nossa representação, se existe uma aresta com origem i e destino j, j aparecerá na lista de vizinhos de i. Escreva uma função `mostraCarentes` que imprima os nós que não têm aresta incidente alguma, ou seja, não são destino em aresta alguma do grafo. (*sugestão:* Percorra todo o grafo marcando os nós que são destino de alguma aresta.)

```
void mostraCarentes (Grafo *g);
```

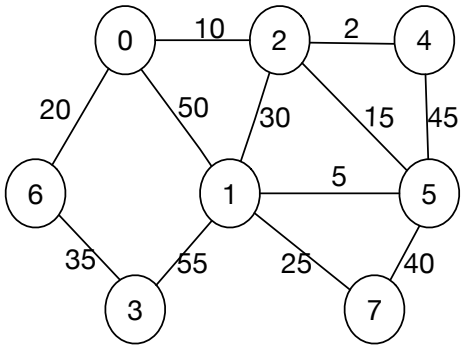
4. (2 pontos) Ainda considerando a representação de grafos mostrada na questão anterior, agora suponha que os grafos são não direcionados. Suponha a disponibilidade de um módulo de união e busca com a interface a seguir.

```
typedef struct suniaoBusca UniaoBusca;
UniaoBusca* ub_cria(int tam);
/* cria particao de conjunto com tam elementos */
/* cada elemento está inicialmente em parte separada */
int ub_busca (UniaoBusca* ub, int u);
/* retorna o representante da parte em que está u */
int ub_uniao (UniaoBusca* ub, int u, int v);
/* retorna o representante do resultado */
void ub_libera (UniaoBusca* ub);
/* libera a estrutura */
```

Escreva uma função `temCiclo` que retorne 1 caso o grafo possua ciclos e 0 caso contrário. (*observação:* Para não considerar a mesma aresta duas vezes, ignore arestas onde o nó destino tem índice menor do que o nó origem.)

```
int temCiclos (Grafo *g);
```

5. (1 ponto) Mostre o passo a passo seguido pelo algoritmo de Kruskal para a construção da árvore geradora mínima para o grafo a seguir. (Não é necessário desenhar o grafo n vezes desde que a ordem dos passos fique clara.)

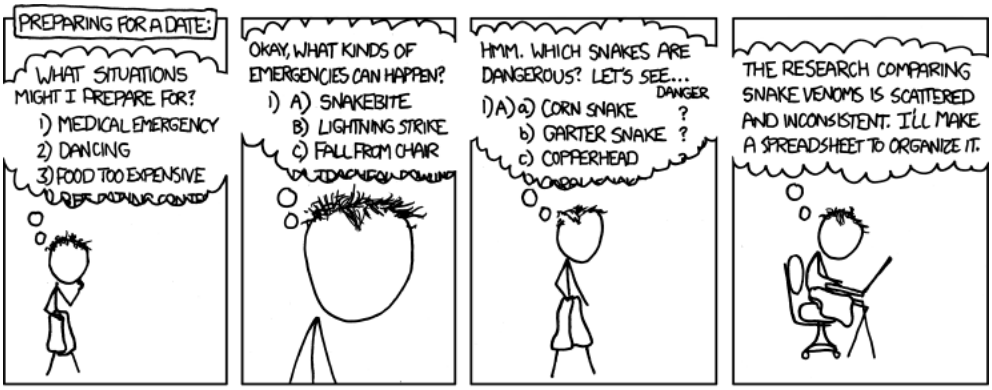


6. (2 pontos) Considere novamente as declarações vistas em laboratório para implementação de grafos, com uma alteração nas funções de enfileiramento e retirada da fila:

```
static SQ* enqueue(SQ* queue, int no, float custo);
/* enfileira um no e seu custo - retorna fila resultante*/
static SQ* dequeue(SQ* queue, int* popped_info, float* custo);
/* retira no e custo mais antigos - retorna fila resultante e esses dois dados */
```

Escreva uma função, com a assinatura mostrada a seguir, que recebe um grafo e um nó inicial e faz um percurso em largura a partir desse nó inicial, mostrando (printf) cada nó visitado e o custo de chegar a ele (em peso de arestas percorridas), nesse percurso, a partir do nó inicial. (*Sugestão*: Faça o percurso em largura normal e pense em como modificá-lo para gerar essa informação a mais.)

```
void grafoPercorreLarguraMostrandoCusto (Grafo *grafo, int no_inicial);
/* mostra percurso em largura e custo de chegada a cada nó neste percurso */
```



I REALLY NEED TO STOP USING DEPTH-FIRST SEARCHES.