1 Struct

```c
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 */
};
```

2 Headers

```c
typedef struct _grafo Grafo;

Grafo* grafoLe( char* filename );
Grafo* grafoLibera(Grafo* grafo);
void grafoMostra(char* title, Grafo* grafo);
int* menoresCaminhos (Grafo *grafo, int no_inicial);
```

3 Implementation

3.1 Initializing

3.1.1 Cria Vizinhos

```c
static Viz* criaViz(Viz* head, int noj, float peso) {
/* insere vizinho no inicio da lista */
  Viz* no = (Viz*) malloc(sizeof(Viz));
  assert(no);
  no->noj = noj;
  no->peso = peso;
  no->prox = head;
  return no;
}
```

3.1.2 Grafo Cria

```c
static Grafo* grafoCria(int nv, int na) {

  int i;
  Grafo* g = (Grafo *) malloc(sizeof(Grafo));
  g->nv = nv;
  g->na = na;
  g->viz = (Viz **) malloc(sizeof(Viz *) * nv);
  for (i = 0; i < nv; i++)
    g->viz[i] = NULL;
  return g;
}
```

3.2 Utilities

3.2.1 Grafo Le

```
Grafo* grafoLe( char* filename ) {

  FILE *arq = fopen(filename,"rt");
  int nv, na, no1, no2 = 0;
  float peso;
  Grafo* novo;

  fscanf(arq, "%d %d", &nv, &na);
  novo = grafoCria(nv, na);
  assert(novo);
  while (fscanf(arq, "%d %d %f", &no1, &no2, &peso) == 3) {
    novo->viz[no1] = criaViz(novo->viz[no1], no2, peso);
    novo->viz[no2] = criaViz(novo->viz[no2], no1, peso);
  }
  return novo;
}
```

3.2.2 Libera

```
Grafo*  grafoLibera(Grafo* grafo) {
  if (grafo) {
    int i = 0;
    Viz* no,*aux;
    for (i = 0; i < grafo->nv; i++){
      no = grafo->viz[i];
      while (no){
        aux = no->prox;
        free(no);
        no = aux;
      }
    }
    free(grafo->viz);
    free(grafo);
  }
  return NULL;
}
```

3.2.3 Mostra

```
void grafoMostra (char* title, Grafo * grafo) {
  int i;
  if (title)
    printf("%s", title);
  if (grafo) {
    printf("NV: %d, NA: %d\n", grafo->nv, grafo->na);
    for (i = 0; i < grafo->nv; i++){
    Viz* viz = grafo->viz[i];
    printf("[%d]->", i);
    while (viz) {
      printf("{%d, %g}->", viz->noj, viz->peso);
      viz = viz->prox;
    }
    printf("NULL\n");
    }
  }
}
```

3.3 Path Finder

3.3.1 Percorre Profundidade Recursiva

```
static void visitaprof (Viz** vizinhos, int no, char *visitado){
  Viz *noviz = vizinhos[no];
  printf("%d ", no);
  visitado[no] = 1;
  while (noviz!=NULL){
    if (!visitado[noviz->noj])
      visitaprof(vizinhos, noviz->noj, visitado);
    noviz = noviz->prox;
  }
}

void grafoPercorreProfundidade(Grafo *grafo, int no_inicial){
  int no;
  char *visitado;
  if (grafo == NULL) return;
  visitado = (char*) malloc(sizeof(int)*grafo->nv);
  assert(visitado);
  for (no=0;no<(grafo->nv);no++)
    visitado[no] = 0;
  visitaprof (grafo->viz, no_inicial, visitado);
  printf ("\n");
}
```

3.3.2 Percorre Largura

```
void grafoPercorreLargura(Grafo *grafo, int no_inicial){
  SQ *q = newList();
  Viz *v; int no;
  int *enfileirado;
  if (grafo == NULL) return;
  enfileirado = (int*) malloc(sizeof(int)*grafo->nv);
  for (no=0;no<(grafo->nv);no++) enfileirado[no] = 0;
  q = enqueue (q, no_inicial);
  enfileirado[no_inicial] = 1;
  q = dequeue(q, &no);
  while (no>=0) {
    printf ("%d-", no);
    v = grafo->viz[no];
    while (v!=NULL) {
      if (!enfileirado[(v->noj)]) {
        q = enqueue (q, v->noj);
        enfileirado[v->noj] = 1;
      }
      v = v->prox;
    }
    q=dequeue(q, &no);
  }
 printf ("\n");
}
```

3.3.3 Percorre Profundidade Iterativa

```
void grafoPercorreProfundidade2 (Grafo *grafo, int no_inicial){
  SQ *q = newList();
  Viz *v; int no;
  int *visitado;
```

```
  if (grafo == NULL) return;
  visitado = (int*) malloc(sizeof(int)*grafo->nv);
  for (no=0;no<(grafo->nv);no++) visitado[no] = 0;
  q = push (q, no_inicial);
  q = pop(q, &no);
  while (no>=0) {
    if (!visitado[no]){
      visitado[no] = 1;
      printf ("%d-", no);
      v = grafo->viz[no];
      while (v!=NULL) {
        q = push (q, v->noj);
        v = v->prox;
      }
    }
    q=pop(q, &no);
  }
 printf ("\n");
}
```

3.3.4 Menor Caminho

```
int menordist( int* caminhos, int* visitados) {
  int minimo = INT_MAX; int nomin = -1;
  int i;
  for (i=0;caminhos[i]!=-1;i++)
    if (!visitados[i] && caminhos[i]<minimo) {
      nomin = i;
      minimo = caminhos[i];
    }
  return nomin;
}
```

3.3.5 Menores Caminhos

```
int* menoresCaminhos (Grafo *grafo, int no_inicial){
  if (no_inicial >= grafo->nv) return NULL;
  #if 1
  int * visitados = (int*)malloc(sizeof(int)*grafo->nv);
  int * caminhos = (int*)malloc((sizeof(int)*grafo->nv)+1);
  int * cmc = (int*)malloc((sizeof(int)*grafo->nv)+1);

  for(int i = 0; i < grafo->nv; i++)
    visitados[i] = 0;
  for(int i = 0; i < grafo->nv; i++)
    caminhos[i] = INT_MAX;
  caminhos[no_inicial] = 0;
  caminhos[grafo->nv] = -1;
  cmc[grafo->nv] = -1;
  cmc[no_inicial] = no_inicial;

  int val = no_inicial;
  while(val!=-1){
    printf("val = %d\n",val);
    Viz* viz = grafo->viz[val];
    while (viz) {
      if((caminhos[val]+viz->peso) < caminhos[viz->noj]){
        caminhos[viz->noj] = caminhos[val]+viz->peso;
        cmc[viz->noj] = val;
      }
```

```
      viz = viz->prox;
    }
      visitados[val] = 1;
      val = menordist(caminhos,visitados);
  }
  #endif
  free(visitados);
  free(caminhos);
 return cmc;
}
```

4 Heap Implementation

4.1 Struct

```
typedef struct heap Heap;
typedef struct _item Item;

struct _item {
  int dist;
  int idno;
};

struct heap {
  int max; /* tamanho maximo do heap */
  int pos; /* proxima posicao disponivel no vetor */
  Item *itens; /* vetor de itens */
  int* posnos;
};
```

4.2 Initializing Heap

```
static Heap *heap_cria (int max) {
  int i;
  Heap* heap=(Heap*)malloc(sizeof(struct heap));
  heap->max=max;
  heap->pos=0;
  heap->itens = (Item *)malloc(max*sizeof(struct _item));
  heap->posnos = (int *)malloc(max*sizeof(int));
  for (i = 0; i < max; i++) heap->posnos[i] = -1;
  return heap;
}
```

4.3 Inserting and removing in Heap

4.3.1 Remove

```
static int heap_remove(Heap *h) {
  int idno;
  if (h->pos == 0) return -1;
  idno = h->itens[0].idno;
  h->posnos[idno] = -1;

  h->itens[0].idno = h->itens[h->pos-1].idno;
  h->itens[0].dist = h->itens[h->pos-1].dist;
  h->posnos[h->itens[0].idno] = 0;

  h->pos--;
```

```
  corrige_abaixo(h, 0);

  return idno;
}
```

4.3.2 Insere

```
static void heap_insere (Heap *h, int distancia, int idno) {
  if (h->pos >= h->max) {
    printf("Heap CHEIO!\n");
    exit(1);
  }
  h->itens[h->pos].dist = distancia;
  h->itens[h->pos].idno = idno;
  h->posnos[idno] = h->pos;
  h->pos++;
 }
```

4.4 Utilities

4.4.1 Troca

```
static void troca(Heap *h, int a, int b) {
  int ida = h->itens[a].idno;
  int idb = h->itens[b].idno;
  Item f = h->itens[a];
  h->itens[a] = h->itens[b];
  h->itens[b] = f;
  h->posnos[ida] = b;
  h->posnos[idb] = a;
}
```

4.4.2 Corrige Abaixo

```
static void corrige_abaixo(Heap *h, int atual) {

  int pai=atual;
  int filho_esq, filho_dir, filho;
  while (2*pai+1 < h->max){
    filho_esq=2*pai+1;
    filho_dir=2*pai+2;
    if (filho_dir >= h->max) filho_dir=filho_esq;
    if (h->itens[filho_esq].dist < h->itens[filho_dir].dist)
        filho=filho_esq;
    else
        filho=filho_dir;
    if (h->itens[pai].dist > h->itens[filho].dist)
        troca(h,pai,filho);
    else
        break;
    pai=filho;
  }
}
```

4.4.3 Corrige

```
static void heap_corrige (Heap* h, int novadist, int idno) {
  int pos = h->posnos[idno];
```

```
  h->itens[pos].dist = novadist;
  for(int i = pos; i >= 0;i--){
    Item pai = h->itens[(i-1)/2];
    int pai_pos = (i-1)/2;
    int filho_pos = i;
    Item filho = h->itens[i];
    if(filho.dist < pai.dist)
      troca(h,pai_pos,filho_pos);
  }
}
```

4.4.4 Debug

```
static void debug_heap_show (Heap *h, char* title) {
  int i;
  printf("%s={",title);
  for (i=0; i<(h->pos); i++)
    printf("[%d , %d] ",h->itens[i].idno, h->itens[i].dist);
  printf("}\n");
}
```

4.5 Menores Caminhos

```
int* menoresCaminhos (Grafo *grafo, int no_inicial){
  if (no_inicial >= grafo->nv) return NULL;
  #if 1
  int * visitados = (int*)malloc(sizeof(int)*grafo->nv);
  int * caminhos = (int*)malloc((sizeof(int)*grafo->nv)+1);
  int * cmc = (int*)malloc((sizeof(int)*grafo->nv)+1);//Caminhos mais curtos

  for(int i = 0; i < grafo->nv; i++)
    visitados[i] = 0;
  for(int i = 0; i < grafo->nv; i++)
    caminhos[i] = INT_MAX;

  caminhos[no_inicial] = 0;
  caminhos[grafo->nv] = -1;
  cmc[grafo->nv] = -1;
  cmc[no_inicial] = no_inicial;

  Heap * h = heap_cria(grafo->nv);
  for(int i = 0; i < grafo->nv; i++){
    if (i != no_inicial)
      heap_insere(h,caminhos[i],i);
  }
  int val = no_inicial;
  while(val!=-1){
    printf("val = %d\n",val);
    Viz* viz = grafo->viz[val];
    while (viz) {
      int pesao = caminhos[val]+viz->peso;
      if((pesao) < caminhos[viz->noj]){
        caminhos[viz->noj] = pesao;
        cmc[viz->noj] = val;
        //printf("before heap_corrige\n");
        heap_corrige(h,pesao,viz->noj);
      }
      viz = viz->prox;
    }
      visitados[val] = 1;
```

```
      val = heap_remove(h);
  }
  #endif
  free(visitados);
  free(caminhos);
  heap_libera(h);
 return cmc;
}
```

5 Kruskal

5.1 Árvore custo mínimo

```
Grafo* arvoreCustoMinimo (Grafo* g) {

  Heap * minEdges;

  minEdges = heap_cria(g->na);

  int val = g->nv;
  while(val--){
    printf("val: %d\n",val);
    Viz * viz = g->viz[val];
    while(viz){
      if(val < viz->noj){
        heap_insere(minEdges,viz->peso,val,viz->noj);
        }
      viz = viz->prox;
      }
  }

  int ktam = g->nv-1;
  int atual;
  Grafo * kruskal = grafoCria(g->nv,ktam);
  UniaoBusca * ub = ub_cria(g->nv);

  int i1,i2;
  int resp1,resp2;

  while(ktam){
    atual = heap_remove(minEdges,&i1,&i2);
    resp1 = ub_busca(ub,i1);
    resp2 = ub_busca(ub,i2);
    if(resp1 != resp2){
      ub_uniao(ub,i1,i2);
      ktam--;
      kruskal->viz[i1] = criaViz(kruskal->viz[i1], i2, atual);
      kruskal->viz[i2] = criaViz(kruskal->viz[i2], i1, atual);
    }
  }
  ub_libera(ub);
  heap_libera(minEdges);

  return kruskal;
}
```

5.2 União e busca

5.2.1 Struct

```
struct suniaoBusca {
  int n;
  int *v;};
```

5.2.2 Cria

```
UniaoBusca* ub_cria(int tam) {
  int i;
  UniaoBusca *ub = (UniaoBusca *) malloc (sizeof(UniaoBusca));
  assert(ub);
  ub->n = tam;
  ub->v = (int *) malloc (tam*sizeof(int));
  assert(ub->v);
  for (i=0;i<tam;i++)
    ub->v[i] = -1;
  return ub;
}
```

5.2.3 Busca

```
int ub_busca (UniaoBusca* ub, int u){
  int x = u;
  int aux;
  if ((u < 0) || (u > ub->n)) return -1;
  while (ub->v[u] >= 0) u = ub->v[u];
  while (ub->v[x] >= 0) {
    aux = x;
    x = ub->v[x];
    ub->v[aux] = u;
  }
  return u;
}
```

5.2.4 União

```
int ub_uniao (UniaoBusca* ub, int u, int v) {
  u = ub_busca (ub, u);
  v = ub_busca (ub, v);
  if ((u<0) || (v<0)) return -1;
  if (ub->v[u] > ub->v[v]) { /* negativos: v[u] menor em modulo! */
    ub->v[v] += ub->v[u];
    ub->v[u] = v;
    return v;
  }
  else {
    ub->v[u] += ub->v[v];
    ub->v[v] = u;
    return u;
  }
}
```

5.2.5 Libera

```
void ub_libera (UniaoBusca *ub) {
```

```
  free(ub->v);
  free(ub);
}
```

5.2.6 Debug

```
void debug (UniaoBusca *ub) {
  int i;
  for (i=0;i<ub->n;i++) printf ("ub[%d]=%d\n", i, ub->v[i]);
}
```