```
------------------------------------------------------------------
graph_prim.cpp: prim's minimum spanning tree algorithm
------------------------------------------------------------------

#include <...>
using namespace std;

template <typename Tkey, typename Twgt>
class graph {

  see graph_wgt.cpp for basic definitions

  public:
    void prim_mst();

  private:
    typedef enum { WHITE, BLACK } vcolor_t;
    vector<vcolor_t> vcolor;
    vector<Twgt> vdist;
    vector<int> vlink;
};

int main(int argc, char *argv[]) {
  if (argc != 2) {
    cerr << "usage: " << argv[0]
         << " graph.txt\n";
    return 0;
  }

  graph<string,int> G;

  G.read(argv[1]);
  G.prim_mst();
}

------------------------------------------------------------------
Hint: Prim's algorithm can be viewed as a Dijkstra modification.

Hint: For sparse graphs (E < V log V), it is more efficient to
use a priority queue just as it was for Dijkstra's algorithm.
In fact, a standard min heap can be used. Push source onto heap.
While heap not empty and all vertices not processed, extract
next vertex from top of heap. If BLACK, skip and try next vertex.
If WHITE, color vertex BLACK and push adjacent lower edge weight
vertices onto heap. Could use std::push_heap and std::pop_heap.
------------------------------------------------------------------
```

```
template <typename Tkey, typename Twgt>
void graph<Tkey,Twgt>::prim_mst() {
  vcolor.assign(V.size(), WHITE);

  vdist.assign(V.size(), numeric_limits<Twgt>::max());
  vdist[0] = 0;

  vlink.assign(V.size(), -1);
  vlink[0] = 0;

  while (1) {
    int next_i = -1;

    Twgt mindist = numeric_limits<Twgt>::max();

    for (int i=0; i<(int)vcolor.size(); i++) {
      if (vcolor[i] == WHITE && mindist > vdist[i]) {
        next_i = i;
        mindist = vdist[i];
      }
    }

    int i = next_i;

    if (i == -1)
      return;

    vcolor[i] = BLACK;

    if (i != vlink[i]) {
      cout << V[min(i,vlink[i])] << " "
           << V[max(i,vlink[i])] << " "
           << vdist[i] << "\n";
    }

    for (int k=0; k<(int)E[i].size(); k++) {
      int j = E[i][k];
      Twgt wij = W[i][k];
      if (vcolor[j] == WHITE) {
        if (vdist[j] > wij) {
          vdist[j] = wij;
          vlink[j] = i;
        }
      }
    }
  }
}
```

```
-----------------------------------------------------------------
graph_kruskal.cpp: kruskals's minimum spanning tree algorithm
-----------------------------------------------------------------

#include <...>
using namespace std;

#include "dset.h"

template <typename Tkey, typename Twgt>
class graph {

  see graph_wgt.cpp for basic definitions

  struct EW {
    EW(int n_i, int n_j, Twgt n_wij) {
      i = n_i;
      j = n_j;
      wij = n_wij;
    }

    bool operator<(const EW &rhs) const {
      if (wij == rhs.wij) {
        if (i == rhs.i) return j < rhs.j;
        return i < rhs.i;
      }

      return wij < rhs.wij;
    }

    int i, j;
    Twgt wij;
  };

  public:
    void kruskal_mst();
};
```

```
template <typename Tkey, typename Twgt>
void graph<Tkey,Twgt>::kruskal_mst() {
  dset DS(V.size());
  vector<EW> H;

  for (int i=0; i<(int)V.size(); i++) {
    for(int k=0; k<(int)E[i].size(); k++) {
      int j = E[i][k];
      Twgt wij = W[i][k];
      if (i<j)
        H.push_back(EW(i, j, wij));
    }
  }

  sort(H.begin(), H.end());

  for (int k=0; k<(int)H.size(); k++) {
    if (DS.find(H[k].i) != DS.find(H[k].j)) {
      cout << V[H[k].i] << " "
           << V[H[k].j] << " "
           << H[k].wij << "\n";

      DS.merge(H[k].i, H[k].j);
    }
  }
}

int main(int argc, char *argv[]) {
  if (argc != 2) {
    cerr << "usage: " << argv[0]
         << " graph.txt\n";
    return 0;
  }

  graph<string,int> G;

  G.read(argv[1]);
  G.kruskal_mst();
}
```

```
-----------------------------------------------------------------
Hint: Kruskal's algorithm processes edges in order of their
weights. Order can also be established using a binary heap.
-----------------------------------------------------------------
```