```
------------------------------------------------------------------
graph1_usage.cpp: basic graph class based on adjacency lists
------------------------------------------------------------------

#include <...>
using namespace std;

template <typename Tkey>
class graph {
  public:
    graph() {}
    void read(const char *);
    void print();

  private:
    enum { UNDIRECTED, DIRECTED } graph_type;
    vector< Tkey > V;          // vertex list
    vector< vector<int> > E;   // edge matrix
    map<Tkey,int> key_map;     // key-to-index map
};

template <typename Tkey>
void graph<Tkey>::read(const char *fname) {
  ifstream in(fname);

  string input;

  getline(in, input);
  if (input.compare("# UNDIRECTED") == 0)
    graph_type = UNDIRECTED;
  else if (input.compare("# DIRECTED") == 0)
    graph_type = DIRECTED;
  else {
    cerr << "error: graph type not known\n";
    exit(0);
  }


  // Create mapping from key to index

  Tkey key1, key2;
  vector< pair<int,int> > Eij;

  while (in >> key1 >> key2) {
    key_map.insert(make_pair(key1, key_map.size()));
    key_map.insert(make_pair(key2, key_map.size()));
    Eij.push_back(make_pair(key_map[key1], key_map[key2]));
  }

  in.close();


  // Create vertex list and sparse edge matrix

  V.resize(key_map.size());
  E.resize(key_map.size());

  typename map<Tkey,int>::iterator kmp;
  for (kmp=key_map.begin(); kmp!=key_map.end(); ++kmp)
    V[kmp->second] = kmp->first;

  for (int k=0; k<(int)Eij.size(); k++) {
    int i = Eij[k].first;
    int j = Eij[k].second;
    E[i].push_back(j);
    if (graph_type == UNDIRECTED)
      E[j].push_back(i);
  }

  vector<int>::iterator Uijp;
  for (int i=0; i<(int)V.size(); i++) {
    sort(E[i].begin(), E[i].end());
    Uijp = unique(E[i].begin(), E[i].end());
    E[i].resize(Uijp - E[i].begin());
  }
}

template <typename Tkey>
void graph<Tkey>::print() {
  for (int i=0; i<(int)V.size(); i++) {
    cout << setw(3) << right
         << i << " "
         << V[i] << ": ";
    for (int k=0; k<(int)E[i].size(); k++) {
      int j = E[i][k];
      cout << V[j] << "  ";
    }
    cout << "\n";
  }
}

int main(int argc, char *argv[]) {
  if (argc != 2)
    return 0;

  graph<string> G;

  G.read(argv[argc-1]);
  G.print();
}
```

```
-------------------------------------------------------------
Hint: Vertex key labels are mapped to integer indices for easy
lookup. Edge indices are sorted and unique values are stored.

Hint: The difference between an undirected and a directed graph
is that the former stores both (i,j) and (j,i) edges while the
latter only stores edges actually present in the input graph.

Hint: The list of header files needed to compile the above is
long: cstdlib, algorithm, fstream, iomanip, iostream, map,
string, utility, and vector.
-------------------------------------------------------------

unix> cat letters_undirected.txt

# UNDIRECTED
A B
B C
B D
C E
D E
D A

unix> ./graph1_usage letters_undirected.txt

  0 A : B D
  1 B : A C D
  2 C : B E
  3 D : A B E
  4 E : C D

unix> cat letters_directed.txt

# DIRECTED
A B
B C
B D
C E
D E
D A

unix> ./graph1_usage letters_directed.txt

  0 A : B
  1 B : C D
  2 C : E
  3 D : A E
  4 E :
```

```
-------------------------------------------------------------------
graph2_usage.cpp: graph class augmented with dfs, bfs algorithms
-------------------------------------------------------------------

#include <...>
using namespace std;

template <typename Tkey>
class graph {

  see graph1_usage.cpp for basic definitions

  public:
    void dfs(Tkey &);     // dfs traversal
    void bfs(Tkey &);     // bfs traversal

  private:
    void dfs(int);
    void bfs(int);

    typedef enum { WHITE, BLACK } vcolor_t;
    vector<vcolor_t> vcolor;
};


see dfs and bsf implementations on the next page


int main(int argc, char *argv[]) {
  if (argc == 4) {
    cerr << "usage: " << argv[0]
         << " -dfs|bfs source"
         << " graph.txt\n";
    return 0;
  }

  graph<string> G;
  G.read(argv[3]);

  if (strcmp("-dfs", argv[1]) == 0) {
    G.dfs(argv[2]);
  } else
  if (strcmp("-bfs", argv[1]) == 0) {
    G.bfs(argv[2]);
  }
}
```

```
template <typename Tkey>
void graph<Tkey>::dfs(Tkey &source_key) {
  if (key_map.find(source_key) == key_map.end()) {
    cerr << "error: " << source_key << " not found!\n";
    exit(0);
  }

  dfs(key_map[source_key]);
}

template <typename Tkey>
void graph<Tkey>::dfs(int source) {
  vcolor.assign(V.size(), WHITE);

  stack<int> S;
  S.push(source);

  while (!S.empty()) {
    int i=S.top();
    S.pop();

    if (vcolor[i] == BLACK)
      continue;

    vcolor[i] = BLACK;
    cout << setw(3) << i << " "
         << V[i] << "\n";

    for (int k=E[i].size()-1; 0<=k ; k--)
      S.push(E[i][k]);
  }
}
```

```
unix> ./graph2_usage -dfs A letters_undirected.txt

  0 A
  1 B
  2 C
  4 E
  3 D
```

```
template <typename Tkey>
void graph<Tkey>::bfs(Tkey &source_key) {
  if (key_map.find(source_key) == key_map.end()) {
    cerr << "error: " << source_key << " not found!\n";
    exit(0);
  }

  bfs(key_map[source_key]);
}

template <typename Tkey>
void graph<Tkey>::bfs(int source) {
  vcolor.assign(V.size(), WHITE);

  queue<int> Q;
  Q.push(source);

  while (!Q.empty()) {
    int i=Q.front();
    Q.pop();

    if (vcolor[i] == BLACK)
      continue;

    vcolor[i] = BLACK;
    cout << setw(3) << i << " "
         << V[i] << "\n";

    for (int k=0; k<(int)E[i].size(); k++)
      Q.push(E[i][k]);
  }
}
```

```
unix> ./graph2_usage -bfs A letters_undirected.txt

  0 A
  1 B
  3 D
  2 C
  4 E
```