CS106B Handout #38
Spring 99-00 May 29, 2000

Section Exercises #9

Problem 1: Minimal Spanning Tree

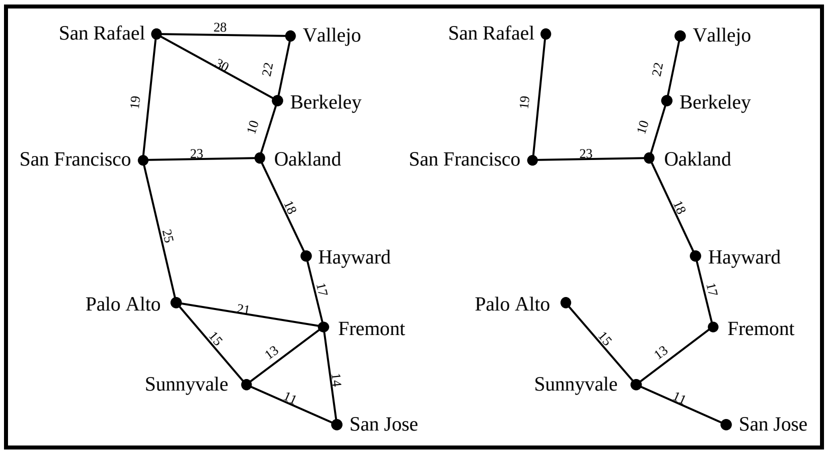
Although Dijkstra's algorithm for finding minimum-cost paths has considerable practical importance, there are numerous other graph algorithms that have comparable commercial significance. In many cases, finding a minimum-cost path between two specific nodes is not as important as minimizing the cost of a network as a whole.

As an example, suppose that you are working for a company that is building a new cable system that connects 10 large cities in the San Francisco Bay area. Your preliminary research has provided you with cost estimates for laying new cable lines along a variety of possible routes. Those routes and their associated costs are shown in the graph on the left side of Figure 16-12. Your job is to find the cheapest way to lay new cables so that all the cities are connected through some path.

To minimize the cost, one of the things you need to avoid is laying a cable that forms a cycle in the graph. Such a cable would be unnecessary, because the cities it connects are already linked by some other path. If your goal is to find a set of arcs that connects the nodes of a graph at a minimum cost, you might as well leave such edges out. The remaining graph, given that it has no cycles, forms a tree. A tree that links all the nodes of a graph is called a **spanning tree**. The spanning tree in which the total cost associated with the arcs is as small as possible is called a **minimum spanning tree.** The cable-network problem described earlier in this exercise is therefore equivalent to finding the minimum spanning tree of the graph, which is shown in the right side of Figure 16-12.

There are many algorithms in the literature for finding a minimum spanning tree. Of these, one of the simplest was devised by Joseph Kruskal in 1956. In Kruskal's algorithm, all you do is consider the arcs in the graph in order of increasing cost. If the nodes at the endpoints of the arc are unconnected, then you include this arc as part of the spanning tree. If, however, the nodes are already connected by a path, you can discard this arc from the final graph. The steps in the construction of the minimum spanning tree for the graph in Figure 16-12 are shown in the sample run on the next page.

A graph and its minimum spanning tree



```
Process edges in order of cost:
10: Berkeley -> Oakland
11: San Jose -> Sunnyvale
13: Fremont -> Sunnyvale
14: Fremont -> San Jose (not needed)
15: Palo Alto -> Sunnyvale
17: Fremont -> Hayward
18: Hayward -> Oakland
19: San Francisco -> San Rafael
21: Fremont -> Palo Alto (not needed)
22: Berkeley -> Vallejo
23: Oakland -> San Francisco
25: Palo Alto -> San Francisco (not needed)
28: San Rafael -> Vallejo (not needed)
30: Berkeley -> San Rafael (not needed)
```

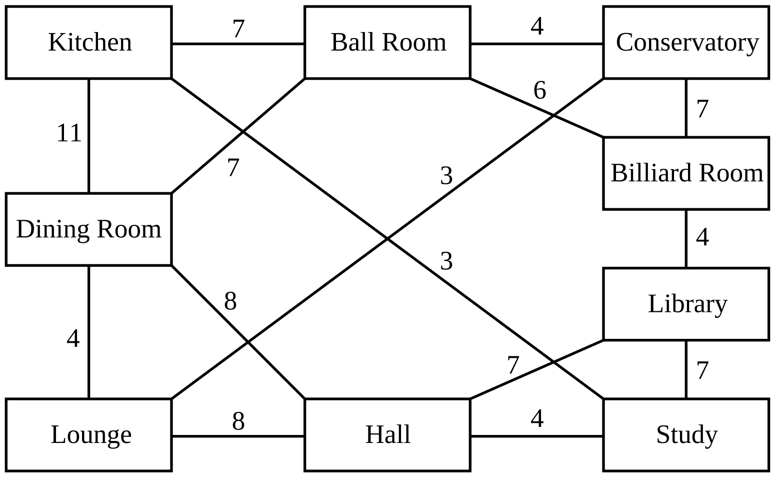
Write a function

```
static graphADT MinimumSpanningTree(graphADT graph);
```

that implements Kruskal’s algorithm to find the minimum spanning tree. The function modifies the original graph, but includes only the arcs that are part of the minimum spanning tree. Also, unlike in Pathfinder, you can assume that all arcs represent the same medium, i.e. there are not parallel arcs representing different modes of transportation. Thus, you do not need to worry that your algorithm creates a minimum spanning tree where some arcs represent flying and other represent train travel, etc, since this contradicts the basic idea behind a minimum spanning tree. Though, should you want to create such a tree, note that the algorithm implemented in the solutions is robust enough to do so.

Problem 2: Understanding graph algorithms

Those of you who have played Clue will recognize the following undirected graph, which shows the connections between the various rooms on the game board:



The numbers on the various arcs show the distance (measured in spaces on the board) between pairs of rooms. For example, the distance from the Hall to the Lounge is 4 steps, and the distance from the Ball Room to the Billiard Room is 6 steps. In this problem, the "secret passages" that connect the rooms at the corners of the board (the Kitchen-Study and Lounge-Conservatory arcs) are arbitrarily assumed to have distance 3.

2a) Indicate the order of traversal for a depth-first search starting at the Lounge. Assume that iteration over a set chooses nodes in alphabetical order. Thus, the first step in the depth-first search will be to the Conservatory, rather than to the Dining Room or Hall, which come later in the alphabet.

2b) Indicate the order of traversal for a breadth-first search starting at the Kitchen. As before, assume that nodes in any set are processed in alphabetical order.

2c) Trace the operation of Dijkstra's algorithm to find the minimum path from the Lounge to the Library. For convenience, Dijkstra's algorithm is shown below.

Figure 1. Code for Dijkstra's algorithm

```
static pathADT FindShortestPath(nodeADT start, nodeADT finish)
{
    queueADT queue;
    pathADT path, newPath;
    arcADT arc;

    queue = NewQueue();
    path = NewPath();
    while (start != finish) {
        if (!IsDistanceFixed(start)) {
            FixNodeDistance(start, TotalPathDistance(path));
            foreach (arc in ArcsFrom(start)) {
                if (!IsDistanceFixed(EndOfArc(arc))) {
                    newPath = NewExtendedPath(path, arc);
                    PriorityEnqueue(queue, newPath,
                                    TotalPathDistance(newPath));
                }
            }
        }
        if (QueueIsEmpty(queue)) return (NULL);
        path = Dequeue(queue);
        start = EndOfPath(path);
    }
    return (path);
}
```

Problem 3: Traversal Strategies for Graphs

The `DepthFirstSearch` and `BreadthFirstSearch` traversal functions given in chapter 16 of your textbook are written to emphasize the structure of the underlying algorithms. If you wanted to include these traversal strategies as part of the graph package, you would need to reimplement the functions so that they no longer depended on a client-supplied `Visit` function. One approach is to implement these two algorithms as the following mapping functions:

```
void MapDFS(nodeFnT fn, nodeADT start, void *clientData);
void MapBFS(nodeFnT fn, nodeADT start, void *clientData);
```

In each case, the functions should call `fn(node, clientData)` for every node reachable from `start` in the specified traversal order.

To make these functions even more general, write the code for `MapDFS` and `MapBFS` so that the mapping function itself keeps track of the nodes that have been visited—presumably by maintaining a set of visited nodes—instead of having the callback function set the visited flag in the node structure. This change will allow nested mapping operations on a graph. For convenience, the code from the book that you will have to modify is included on the next page.

Figure 2. Code for graph traversal

```
/*
 * Functions: DepthFirstSearch, BreadthFirstSearch
 * Usage: DepthFirstSearch(start);
 *        BreadthFirstSearch(start);
 * ------------------------------
 * These functions visit each node in the graph containing start,
 * beginning at the start node.  DepthFirstSearch recursively explores
 * each path as far as it can; BreadthFirstSearch explores outward in
 * order of increasing distance from the start node.
 */

void DepthFirstSearch(nodeADT start)
{
    iteratorADT iterator;
    nodeADT node;

    if (HasBeenVisited(start)) return;
    Visit(start);
    iterator = NewIterator(ConnectedNodes(start));
    while (StepIterator(iterator, &node)) {
        DepthFirstSearch(node);
    }
    FreeIterator(iterator);
}


void BreadthFirstSearch(nodeADT start)
{
    iteratorADT itSet, itNode;
    setADT frontier;
    nodeADT node, target;

    frontier = NewPtrSet(PtrCmpFn);
    AddPtrElement(frontier, start);
    while (NElements(frontier) > 0) {
        itSet = NewIterator(frontier);
        frontier = NewPtrSet(PtrCmpFn);
        while (StepIterator(itSet, &node)) {
            if (!HasBeenVisited(node)) {
                Visit(node);
                itNode = NewIterator(ConnectedNodes(node));
                while (StepIterator(itNode, &target)) {
                    AddPtrElement(frontier, target);
                }
                FreeIterator(itNode);
            }
        }
        FreeIterator(itSet);
    }
}
```