Section Solutions #9

Problem 1: Minimal Spanning Tree

```
/* Function:  MinimumSpanningTree
 * Usage:       minGraph = MinimumSpanningTree(graph);
 * ----------------------------------------------
 * Note that this function alters the original graph passed in. In this
 * function, we declare a new priority queue. We initiate an iterator
 * over the arcs in the graph. We enter each arc into the queue based on
 * its associated cost (assuming that CostOfArc is a function capable of
 * extracting the cost from the void * returned by GetArcData).  We then
 * delete that arc from the graph.  This "disconnects" the two nodes.
 * Thus, when the iterator has finished, the graph has no arcs remaining.
 * Now, we evaluate each of the arcs (all of which have been stored in the
 * priority queue) and decide whether or not they should be entered back
 * into the graph.  We can use Dijkstra's algorithm as provided in the
 * book, since we are interested in the case in which the nodes are not
 * connected and in that case Dijksta's simply returns NULL.  Thus, if
 * Dijkstra's returns NULL, we add the arc back into the set, otherwise we
 * don't.  Note that the minimum spanning tree concept actually relies on
 * bidirectional arcs.  Thus, when we find an arc to add.  We add it
 * to the graph as well as the arc in the opposite direction, creating the
 * effect of a birectional arc.
 */

static graphADT MinimumSpanningTree(graphADT graph)
{
   pqueueADT pqueue;
   arcADT tmpArc, addedArc, arc;

   pqueue = NewPriorityQueue();
   foreach (arc in Arcs(graph)){
          PriorityEnqueue(pqueue, arc, CostOfArc(GetArcData(arc)));
          DeleteArc(graph, arc);
   }
   while (!IsEmpty(pqueue)) {
          tmpArc = ExtractMax(pqueue);
          if (!FindShortestPath(graph, StartOfArc(tmpArc), EndOfArc(tmpArc))){
                 addedArc = NewArc(StartOfArc(tmpArc), EndOfArc(tmpArc));
                 SetArcData(addedArc, GetArcData(tmpArc));
                 addedArc = NewArc(EndOfArc(tmpArc), StartOfArc(tmpArc));
                 SetArcData(addedArc, GetArcData(tmpArc));
          }
   }
}
```

Note: For the above to work, graph.h will need to be modified slightly. We need to export the function DeleteArc, which disconnects nodes. It can be implemented in graph.c as follows:

```
void DeleteArc (graphADT graph, arcADT arc)
{
   DeletePtrElement(Arcs(graph), arc);
}
```

Problem 2 : Understanding graph algorithms

2a) Lounge, Conservatory, BallRoom, BilliardRoom, Library, Hall, DiningRoom, Kitchen, Study

2b) Kitchen, BallRoom, DiningRoom, Study, BilliardRoom, Conservatory, Hall, Lounge, Library

2c)
Fix distance to Lounge at 0
Process the arcs out of Lounge (Conservatory, DiningRoom, Hall)
Enqueue the path: Lounge -> Conservatory (3)
Enqueue the path: Lounge -> DiningRoom (4)
Enqueue the path: Lounge -> Hall (8)
Dequeue the shortest path: Lounge -> Conservatory (3)
Fix distance to Conservatory at 3
Process the arcs out of Conservatory (BallRoom, BilliardRoom, Lounge)
Enqueue the path: Lounge -> Conservatory -> BallRoom (7)
Enqueue the path: Lounge -> Conservatory -> BilliardRoom (10)
Ignore Lounge because its distance is fixed
Dequeue the shortest path: Lounge -> DiningRoom (4)
Fix distance to DiningRoom at 4
Process the arcs out of DiningRoom (BallRoom, Hall, Kitchen, Lounge)
Enqueue the path: Lounge -> DiningRoom -> BallRoom (11)
Enqueue the path: Lounge -> DiningRoom -> Hall (12)
Enqueue the path: Lounge -> DiningRoom -> Kitchen (15)
Ignore Lounge because its distance is fixed
Dequeue the shortest path: Lounge -> Conservatory -> BallRoom (7)
Fix distance to BallRoom at 7
Process the arcs out of BallRoom (BilliardRoom, Conservatory, DiningRoom, Kitchen)
Enqueue the path: Lounge -> Conservatory -> BallRoom -> BilliardRoom (14)
Ignore Conservatory because its distance is fixed
Ignore DiningRoom because its distance is fixed
Enqueue the path: Lounge -> Conservatory -> BallRoom -> Kitchen (14)
Dequeue the shortest path: Lounge -> Hall (8)
Fix distance to Hall at 8
Process the arcs out of Hall (DiningRoom, Library, Lounge, Study)
Ignore DiningRoom because its distance is fixed
Enqueue the path: Lounge -> Hall -> Library (15)
Ignore Lounge because its distance is fixed
Enqueue the path: Lounge -> Hall -> Study (12)
Dequeue the shortest path: Lounge -> Conservatory -> BilliardRoom (10)
Fix distance to BilliardRoom at 10
Process the arcs out of BilliardRoom (BallRoom, Conservatory, Library)
Ignore BallRoom and Conservatory because their distances are fixed
Enqueue the path: Lounge -> Conservatory -> BilliardRoom -> Library (14)
Dequeue the shortest path: Lounge -> DiningRoom -> BallRoom (11)
Ignore this path because the distance to the BallRoom is fixed
Dequeue the shortest path: Lounge -> DiningRoom -> Hall (12)
Ignore this path because the distance to the Hall is fixed
Dequeue the shortest path: Lounge -> Hall -> Study (12)
Fix distance to Study at 12
Process the arcs out of Study (Hall, Kitchen, Library)
Ignore Hall because its distance is fixed
Enqueue the path: Lounge -> Hall -> Study -> Kitchen (15)
Enqueue the path: Lounge -> Hall -> Study -> Library (19)
Dequeue the shortest path: Lounge -> Conservatory -> BallRoom -> BilliardRoom (14)
Ignore this path because the distance to the BilliardRoom is fixed
Dequeue the shortest path: Lounge -> Conservatory -> BallRoom -> Kitchen (14)
Fix distance to Kitchen at 14
Process the arcs out of Kitchen (BallRoom, Study, DiningRoom)
Ignore BallRoom because its distance is fixed
Ignore DiningRoom because its distance is fixed
Ignore Study because its distance is fixed
Dequeue the shortest path: Lounge -> Conservatory -> BilliardRoom -> Library (14)

Shortest path: Lounge -> Conservatory -> BilliardRoom -> Library (14)

Problem 3: Traversal Strategies for Graphs

```
/*
 * Functions: MapDFS, MapBFS
 * Usage: MapDFS(fn, start, clientData);
 *        MapBFS(fn, start, clientData);
 * ------------------------------
 * These functions visit each node in the graph containing start,
 * beginning at the start node, and call the client-supplied fn
 * on each visited node.  These functions keep track of which nodes
 * have been visited internally using a set. MapDFS recursively explores
 * each path as far as it can; MapBFS explores outward in
 * order of increasing distance from the start node.
 */

typedef void (*nodeFnT)(nodeADT node, void *clientData);

void MapDFS(nodeFnT fn, nodeADT start, void *clientData)
{
  setADT visitedNodes = NewPtrSet(PtrCmpFn);
  RecMapDFS(fn, start, clientData, visitedNodes);
  FreeSet(visitedNodes);
}

void RecMapDFS(nodeFnT fn, nodeADT start, void *clientData
               setADT visitedNodes)
{
    iteratorADT iterator;
    nodeADT node;

    if (IsPtrElement(visitedNodes, start)) return;
    fn(start, clientData);
    AddPtrElement(visitedNodes, start);

    iterator = NewIterator(ConnectedNodes(start));
    while (StepIterator(iterator, &node)) {
        RecMapDFS(fn, node, clientData, visitedNodes);
    }
    FreeIterator(iterator);
}
```

```
void MapBFS(nodeFnT fn, nodeADT start, void *clientData)
{
    iteratorADT itSet, itNode;
    setADT frontier, visitedNodes;
    nodeADT node, target;

    visitedNodes = NewPtrSet(PtrCmpFn);
    frontier = NewPtrSet(PtrCmpFn);
    AddPtrElement(frontier, start);

    while (NElements(frontier) > 0) {
        itSet = NewIterator(frontier);
        frontier = NewPtrSet(PtrCmpFn);
        while (StepIterator(itSet, &node)) {
            if (!IsPtrElement(visitedNodes, node)) {
                fn(node, clientData);
                AddPtrElement(visitedNodes, node);

                itNode = NewIterator(ConnectedNodes(node));
                while (StepIterator(itNode, &target)) {
                    AddPtrElement(frontier, target);
                }
                FreeIterator(itNode);
            }
        }
        FreeIterator(itSet);
    }
}
```