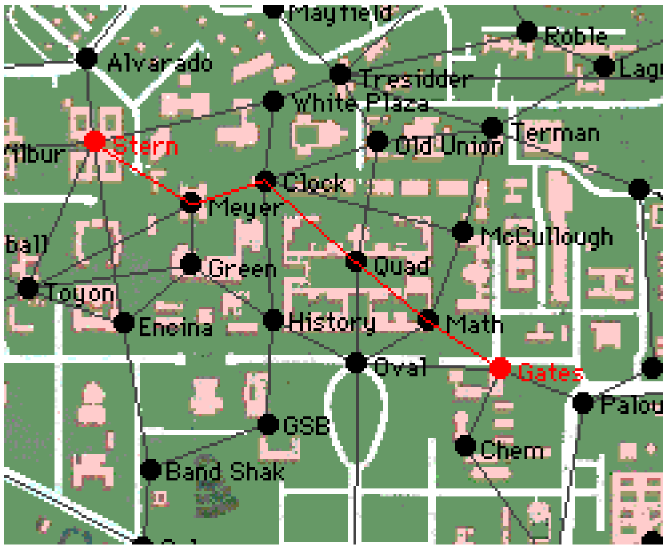
CS106B
Fall 2000

Handout #48
December 1, 2000

Assignment #9: Pathfinder

Due: Friday, December 8 at 5:00 p.m.

You may not use any late days on this assignment! Any assignments not received by 5:00 p.m. on Friday, December 8 will automatically receive a 0.



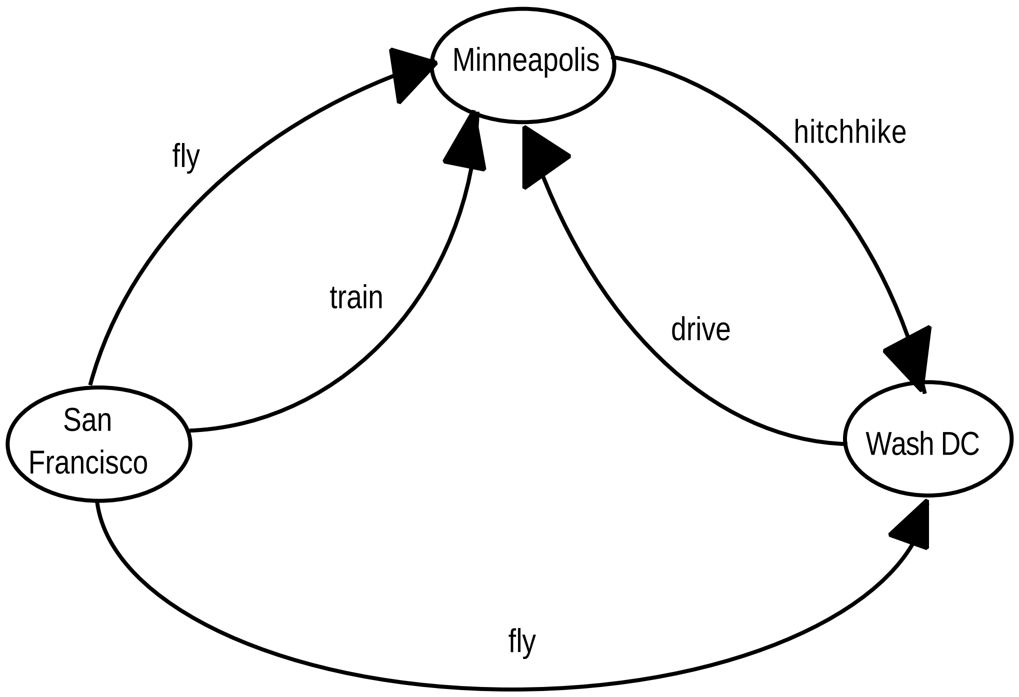
Have you ever wondered why it's so difficult to be on time for CS106B at 9:00 a.m.? Maybe you could shave off a few critical seconds by going around the Math corner rather than cutting through the Quad. Or maybe the answer is to give up walking and invest in a pair of Rollerblades or a bike. But then again, bikes and blades are expensive and money doesn't grow on trees. So, what if you truly needed to know what was the shortest or fastest route between your dorm and Gates? Or what about the similar task of plotting out the cheapest and fastest way to get you back home to Fargo for Winter Break?

This task of exploring choices in an attempt to optimize a path is a graph traversal problem of the type discussed in Chapter 16 of the text. With a bag full of ADTs, including priority queues and symbol tables, you can make quick work of writing a program to do this tedious task. It's a perfect problem for someone with finely-honed CS106B programming skills.

In this assignment, you will write a program that reads in a data file of sites and paths, asks the user for a source and destination, and searches the paths from the start seeking the optimal route to the destination. This is your last 106B assignment, and it involves one of the classic algorithms of computer science (not to be confused with a "great topic" in computer science). Good luck!

The basic plan for this assignment

A map is represented as a connected set of sites. Each site is called a *node*. A node has a name ("San Francisco") and set of connections (or *arcs*) that lead from that node to other nodes. Each arc has information associated with it that includes the mode of transportation (represented as a string such as "drive", "walk", "fly", etc.), the distance traveled along that route, the time it takes, and the cost of using that means of transportation.



This type of data structure is commonly referred to as a *graph*. A graph is a generalization of the recursive tree data structure. However, a graph doesn't have a root node like a tree does — all nodes are basically peers, and it is possible for graphs to have cycles—that is, tracing a path away from a node may at some point return back to that same node. Note that it is possible for there to be more than one arc connecting two nodes, such as the two different options shown above to get from San Francisco to Minneapolis.

Your program will read in the graph nodes and arcs from a data file specified by the user. You will then draw the nodes in the graphics window and connect them up with lines to show the arcs. The user will choose the start node and destination node by clicking on two sites in the graph. Your goal will be to find a route that is optimal for each of these different criteria: shortest travel time, least total cost, minimum distance, and minimum number of hops. The same path may be the best for all situations, but likely there will be different results for the different goals. It will be your job to find the best path for each criteria and draw and print that path.

For example, here is the arc data for the 3-node graph above:

```
  "fly" from "SF" to "Minn" distance: 1777  time: 4 cost: 485.00
  "train" from "SF" to "Minn" distance: 1890  time: 18 cost: 250.00
  "fly" from "SF" to "DC" distance: 2441  time: 5 cost: 540.00
  "hitchhike" from "Minn" to "DC" distance: 1600 time: 40 cost: 25.00
  "drive" from "DC" to "Minn" distance: 1076  time: 17 cost: 107.00
```

You're trying to get home from San Francisco to Washington DC. The optimal solution to minimize distance is to go directly via flying, for a total of 2441 miles. However, to spend the least money, you should take the train to Minneapolis, and then hitchhike to Washington, for just $275.

Optimal path search

One strategy you could employ to find the optimal path is to exhaustively construct all possible paths between the two nodes and then compare them all to find the best one. However, given that there are a lot of possible paths between any two sites, a completely exhaustive search would be time consuming; in general, the running time would be $O(2^N)$. Instead, you're going to make use of a new kind of queue called a priority queue to direct your search quickly and efficiently to find the best path.

We are going to use a method known as Dijkstra's Algorithm (after its inventor, Edsgar Dijkstra). The basic idea is to expand our search out from the starting node, in order of increasing total path cost. When this leads us to our destination, we must have the shortest path because we have been searching in order of increasing cost. Algorithms of this type are called "greedy", since they always focus attention on the best path found so far. They find the optimal path without exhaustive search. Dijkstra's Algorithm is discussed in more detail in Chapter 16 and later in this handout.

Graph data files

The graph is described in a data file that contains the information for each node and its arcs. The first line is the name of the map picture (used to draw the background picture in the graphics window; more on that later) and a list of the graph nodes, each identified by its name and its x-y coordinate. Then there is a list all of the graph arcs, where each arc connects two nodes via some means of transportation with an associated distance, time, and cost. The file format looks like this:

```
USA picture                   <-- name of map picture
NODES                         <-- word "NODES" marks beginning of node listing
Denver                        <-- name of node
2.54 3.25                     <-- x y coordinate for this node
San Francisco
1.05 2.66
ARCS                          <-- word "ARCS" marks beginning of arc listing
 "fly" from "Denver" to "SF" distance: 1780  time: 1 cost: 15.00
 "drive" from "LA" to "Denver" distance: 1890 time:  3 cost: .29
                                    <-- blank line indicates end of all arcs
```

The starter code provides an **sscanf** template to use in reading the file format to save you the trouble of messing with formatting details. Be sure to use it and spare yourself grief. For information about the scanf functions and their conversion specifications, see pp. 146-148 in the text (if you really care).

As you read in the data, you will build data structures representing it using the `graphADT`, `nodeADT`, and `arcADT` from Chapter 16. Later, you will also need a `pathADT`, which you will write (the .h file is provided). The node and arc ADTs maintain `void *` pointers for the client's use. You will use these to associate the necessary data (city names, coordinates, etc.) with nodes and arcs. You will have to design structures that hold the necessary information, and use functions exported by the ADTs to form the association. You will also need to build a symbol table that associates location names with pointers to the corresponding `nodeADT`. This will help you build the arcs as you read the data. Once your structures are all in place, you will be ready to solve graph traversal problems by letting the user select starting and ending nodes.

Interacting with the user

After you have read in the data and built the graph data structure, your program will draw the graph in the graphics window and allow the user to click on the starting and destination sites. We provide a function that will return the x-y coordinate of the click. It will be up to you to determine whether a node was under the mouse click. If the user did not click on a node, you will ask for another attempt.

Once you have gotten a valid starting and ending node, you will apply Dijkstra's Algorithm to explore the paths outward from the start node, at each stage choosing to follow the best path so far. Once your program arrives at the destination node, it will report the optimal solution by tracing it on the map and printing the steps to the text window. You will call your traversal procedure multiple times, one time for each of the different criteria you are trying to optimize to provide the different results.

Solution strategies

The discussion above is pretty general. So, straight from the 106X Task Breakdown Task Force, here's our suggested order of attack. We highly recommend following this plan. Each step is discussed below.

Task 1	Check out the following ADTs and get familiar with their interfaces: `graphADT`, `nodeADT`, `arcADT`, `pathADT`, and `symtabADT`.
Task 2	Read the data files and construct the graph, nodes, arcs, and symbol table.
Task 3	Draw all of the arcs and nodes for the graph in graphics window
Task 4	Write the code for the user to choose nodes by clicking on them in graphics window
Task 5	Finish implementing the `pathADT` and add the shortest path algorithm to your program.
Task 6	Traverse to find the quickest path between nodes selected by the user
Task 7	Generalize search to solve for other optimization criteria

Task 1: ADTs

Much of this assignment is made easier by use of pre-written ADTs. We supply you with the following ADTs from the text: the polymorphic symbol table ADT, `graphADT`, `nodeADT`, and `arcADT`. The interface for graphs, nodes, and arcs is from Figure 16-6, and the code is from Figure 16-7. We are using the set-based implementation, and include the .h and .c files in the starter files. To find out more about using the `setADT` refer to Chapter 15 in the textbook. As with all ADTs, you can make use of the `setADT` without having to understand the implementation in the file `set.c`.

The pathADT is discussed in the text but not fully implemented. We provide a .h file and most of the path functionality; you will need to write the remainder of the code. You will also use a priority queue ADT, as defined by pqueue.h and implemented in pqueue.c. Priority queues are discussed in the text on page 724 and on page 452, Problem 5 (there is additional discussion on pages 728 - 732, but this is beyond what you need to know for this assignment).

The idea of a priority queue is that associated with each item in the queue is a value that represents its priority, and that when you dequeue, instead of getting the oldest item in the queue, you get the one with the highest priority. An example might be an airline's queue of standby passengers, where frequent flyers might have higher priority and thus would be dequeued first. One thing to note: the lower the associated value, the higher the priority. This is just what we want for this assignment, where we will place paths in the queue, then want to dequeue them in increasing order of length. So the shortest path currently in the queue should have the highest priority. Read over the interface for the priority queue carefully so you understand how it works; you may find it useful to write some throwaway test code that exercises the priority queue, just to make sure you know how it behaves.

The symbol table ADT will be used to associate node names with pointers to `nodeADTs`. (Note that you will create a separate `nodeADT` for each node in the graph, and an `arcADT` for each arc). Using the table will allow you to look up nodes by name, something you will have to do when you read the arc data. The `pathADT` will be used by the graph traversal algorithm to keep track of the arcs that make up a path. It contains operations that allow you to append arcs, sum the path, and more. You should look over the header files to become familiar with the interfaces to these ADTs. They are reproduced at the end of this handout or in the text.

Since most of the ADTs are given, the thrust of this assignment is writing client code. You are given a set of tools, and you will use them to solve an interesting problem. This is a common situation in modern programming.

Task 2: Reading node and arc data from a file

In the file `graph.h`, you find that the ADTs export functions for creating new graphs, nodes, and arcs. When you read a data file, you will call `NewGraph` once for the whole file to create a `graphADT`. Then you will call `NewNode` for every node in the file (creating a `nodeADT` for each), then you will call `NewArc` for each arc that you read.

It is necessary, of course, to associate the data you read (such as distance between cities) with the nodes and arcs you create, and to access this data when needed. The ADTs make this possible via the following functions, which are discussed on page 706 of the text.

```
void SetNodeData(nodeADT node, void *data);
void *GetNodeData(nodeADT node);
void SetArcData(arcADT arc, void *data);
void *GetArcData(arcADT arc);
```

You will have to design data structures to hold the information you read from the file, then pass pointers to these structures to the "Set…Data" functions. For example, for nodes you will have to keep track of the name of the node and its coordinates. The `arcADT` keeps track of which `nodeADTs` an arc connects, but you will have to use the data pointer to associate the other information with each arc: the method of transportation, distance, time, and cost.

Let's make this a little more explicit. To read a file, the first thing you do is create a graph and a symbol table. Then you read the nodes, and for each one you do the following. First, allocate and fill a structure for the name and coordinate data (let's say you call this struct a `nodeDataT`). Second, call `NewNode` (it's in `graph.h`) to create a node and add it to the graph. Third, call `SetNodeData` to store a pointer to the `nodeDataT` in the node. Fourth, enter the node into the symbol table, using the location name as the key (e.g., "San Francisco"), and the pointer to the node (it was returned by `NewNode`) as the value. The reason for the symbol table will become apparent momentarily.

After reading all the nodes, you read the arcs. You will have defined another data type, `arcDataT`, to hold the arc information mentioned above. The file provides the names of the locations at the ends of the arcs, so the next thing to do is look the names up in the symbol table to retrieve pointers to the nodes. Then you can call `NewArc` to add the arc to the graph. Then you call `SetArcData` to associate the `arcDataT` with the arc. Note that you may have several arcs between the same two cities, one for each mode of transportation. The `graphADT` is quite happy with parallel arcs, as they are called.

A question that will immediately arise is how to know if you have successfully built the graph. A tool that will help you is the *iterator*. We provide `iterator.h` and `iterator.c` (they are from Chapter 15), and you can use an iterator to print out the information associated with each node like this:

```
iteratorADT iterator;
nodeADT node;
...
iterator = NewIterator(Nodes(myGraph));
while (StepIterator(iterator, &node))
    {
        PrintNodeData(node);
    }
```

The code for `PrintNodeData` will call `GetNodeData` to get the `nodeDataT` for the node, extract the information, and print it. You can use a similar approach to convince yourself that the arcs have been correctly added to the graph, and you could use `MapSymbolTable` to get a list of the locations in the symbol table.

Be sure you understand the iterator code shown above. You will use iterators in many places in this program! Remember from lecture that the `foreach` macro may make your code more readable when working with iterators.

Task 3: Drawing the graph

Recall that the first line of each graph data file is the name of picture to use as the background. Use the function in the extended graphics library called `DrawNamedPicture` that allows you to draw a named picture in the graphics window. After drawing this picture in the background, you will draw all of the nodes and arcs in the graph. Each node should be represented as a small filled circle labeled with the node name, each arc is drawn a straight line between two nodes. A sample is shown on the first page of this handout. To associate the picture name with the graph, you can use the `SetGraphData` function, called with a pointer to a data structure you define. (Keeping the symbol table here might also be a good idea.)

Since you don't have direct access to the nodes and arcs (they are behind the wall of abstraction), you will use the iterators, as suggested in Task 2, to get the information you need. Draw each node at its proper x-y coordinate and draw the arcs between nodes.

Task 4: Finding a node by clicking
You will ask the user to choose the starting site and desired destination by clicking on the displayed graph. We provide a function called `GetUserMouseClick` in `pathfinder.c` that will wait for the user to press the mouse and then return a `coord` struct which holds the x and y coordinates of where the mouse was clicked. Given this `coord`, you need to find if there is a node at that location, and if so, which one. Again, the appropriate tool is an iterator.

Task 5: Working with paths
We have provided you with an interface called `path.h` for a `pathADT` you can use to keep track of the arcs that make up a path. The `pathADT` allows you to create and free paths, append arcs to a path, get the first and last node of a path, etc. It also includes a mapping function that allows you to apply a function to each arc in the path—this will be useful when you are trying to print or draw a path.

Most importantly, the `pathADT` is capable of evaluating the path and reporting its sum (e.g. the total distance or total cost or whatever arc weighting you are using). When you create a path, you must specify a function pointer that can be applied to each arc to obtain its weight to add to the sum. By supplying different weighting functions, you can use the `pathADT` to sum over different criteria.

Most of the implementation of the `pathADT` has been given to you in `path.c` . However, the important functions `MapPath` and `TotalPathDistance` have not been implemented yet. You will need to study the provided `pathADT` implementation and then write the code for these functions before the `pathADT` will be complete.

During the search, you will be using the priority queue ADT we have provided to store the collection of paths to explore. The priority queue needs to know how to order queue elements relative to each other. In this version of the priority queue, you specify two things when you enqueue an item: a `void *` pointer to the item (in our case, an arc), and a `double` that represents the priority of the item. This priority queue dequeues in order of **lowest** priority value.

Task 6: Finding the quickest path
With all your infrastructure in place, you're finally ready to work on the heart of the program—the priority-driven traversal to find the optimal path. To complete Task 6, you only need one arc-weighting function, which returns the travel time for an arc. Your traversal should find the path that minimizes total travel time from the start to the destination. Print out the path determined to be the optimal solution and visually highlight the route by drawing the arcs of the path in a distinctive color.

In a little more detail, here is how the algorithm works. We will be creating a priority queue of paths, prioritized in terms of total distance (or time, etc.). When we have determined the shortest distance from the starting point to some other node (not necessarily the desired destination), we will say we have "fixed" the distance to that node. Using this terminology, our goal is to fix the distance to the desired destination. We will keep track of which nodes have been fixed, and the distance to each.

Suppose the starting node is called A. We begin by fixing the distance to A as 0. Then we place in the queue the paths from A to all nodes directly connected to it. Now we choose the shortest of these paths--suppose it is to node D. Guess what--we have just fixed to length of the path to node D. There can't possibly be a shorter way to D, because all the other paths leading out of A are at least as long as this one. We now mark D as fixed and note the distance. Now we add to the queue the paths that go from A to D to each of D's direct neighbors, omitting the path from A to D to A, of course.

Now we just repeat this process over and over. We dequeue a path, and since this is a priority queue, we will get the shortest path we have found so far. This allows us to fix the distance to the node at the end of the path, since no other path could be this short. Now we extend the path that we just dequeued by one node, placing the resulting paths (one for each neighbor) in the priority queue unless they end with a node whose distance we have already fixed. In this case, we discard the extended path. Then we dequeue the next path, fix the distance to the end, etc. When we finally dequeue a path that ends at the desired destination, we have the solution to the problem.

What we have just described is Dijkstra's Algorithm, which is also described in Section 16.5 of the text. The code shown in Figure 16-9 is provided in the file `dijkstra.c`, with the "foreach" shorthand of the text expanded into the explicit use of an iterator. We have also modified the code to use an arc weighting function for paths, which will be necessary for Task 7. You can paste this code into your program when you are ready for it.

One more detail about the algorithm: sometimes when we dequeue a path, we will find that the distance to the end node has already been fixed. In this case, just discard the dequeued path--it can't be as good as the one we already found.

In summary, you will be using a priority queue to keep track of your current paths, the `pathADT` to manage the paths and evaluate their costs, and you need to have a way to keep track of the nodes that have been fixed, along with their costs.

Task 7 Optimizing for other criteria

Now you will generalize your pathfinder to optimize for other criteria than just shortest travel time. This is actually just a small step: by adding a function parameter that specifies how to weight the arcs to the search, you can control how a path computes its sum, and thus how paths measure up relative to one another. Optimizing for a different criterion simply means writing a new arc weighting function and then making another call to your optimize function.

Write two additional weighting functions that can be used as the criteria to find:

1) the path with the minimum cost

2) the path with the minimum number of hops

Change your program to call the optimizer routine two more times, each time passing a different weight function to obtain the minimum path for a different criterion, printing and drawing the results.

Notes

Careful planning aids re-use.

This program has a lot of opportunity for code-unification and code-reuse, but it requires some careful up-front planning. You'll find it much easier to do it right the first time than to go back and try to unify it after the fact. Sketch out your basic attack before writing any code and look for potential code-reuse opportunities in advance so you can design your functions to be all-purpose from the beginning.

Freeing memory.

Dijkstra's algorithm can use up an incredibly large amount of memory when searching for the shortest path. You are bound to run out of memory after a few searches if you are not careful about freeing memory allocated in the `FindShortestPath` function. You are required to free all of the dynamic memory allocated within the `FindShortestPath` function, but you do not need to free memory anywhere else. Remember that it is always a good idea to get your program completely working without memory management first and then go back and try to free your memory.

Accessing Files

There are starter folders for the Mac and the PC. Each folder contains:

`pathfinder.h`	An incomplete interface file for the main program.
`pathfinder.c`	A shell of the main program.
`dijkstra.c`	The implementation of Dijkstra's Algorithm. Modified from the text to be compatible with our other files.
`path.h`	The interface file for the pathADT (a list of arcs).
`path.c`	A shell of the implementation of the pathADT.
`pqueue.h`	The interface file for the priority queue.
`pqueue.lib`	The implementation of the priority queue (pre-compiled as a library).
`symtab.h`	The interface file for a polymorphic symbol table.
`symtab.c`	The implementation for the symbol table.
`iterator.h`	The interface file for the polymorphic iterator from Chap. 15.
`iterator.c`	The implementation for the polymorphic iterator.
`graph.h`	The interface file for the graphADT of Chap. 16.
`graph.c`	The implementation of the graphADT.
`set.h`	The interface file for the setADT of Chap. 15.

`set.c`	The implementation of the setADT.
`bst.h`	The interface file for the bstADT used to implement sets.
`bst.c`	The implementation of the bstADT.
`cmpFn.h`	The interface file for the compare functions.
`cmpFn.c`	The implementation of the compare functions. Used by sets.
`foreach.h`	The interface file for the foreach macro from Chapt. 15.
`foreach.c`	The implementation of the foreach macro. Used by graphs.
`Small Data`, `Cities`, `Stanford`	Graph data files. The small one is helpful when you're trying to debug.
`Pathfinder.π.rsrc`	A Mac resource file containing the pictures used. Individual files are provided for PCs.

Your project should contain all the .c files listed above except `dijkstra.c`. When you are ready to work on Task 6, you can copy the function from `dijkstra.c` into `pathfinder.c`. **The only files you will modify are pathfinder.c and path.c.** Also, keep in mind that some of these files are only used in the `graphADT` implementation. You won't need to use these ADTs directly in your program but you must still add the files to your project in order for the program to run.

Deliverables

The final fully-written, fully-functional program source files should be submitted electronically by 5:00 p.m. on December 8th**. As mentioned previously, late days may not be used on this assignment!** You'll also need to get a printout to your section leader (or in class) by the same deadline. Make sure this printout is clearly marked with your name and section leader's name. We are being more strict with the paper copies because of the short time your section leader has to grade this assignment.

Note: Due to time constraints at the end of the quarter, this program will not be interactively graded.

Interface Files

Here are the following interface files: `path.h`, `pqueue.h`, and `iterator.h`. `graph.h` is in the text as Figure 16-6.

```
/*
 * File: path.h
 * -------------
 * Defines a abstraction for paths, which are ordered lists of arcs.
 */

#ifndef _path_h
#define _path_h

#include "genlib.h"
#include "graph.h"           /* for definitions of graph, node, arc */

/*
 * Type: pathADT
 * ------------------
 * Provides an abstract data type for manipulating paths, which are lists of
 * arcs. The type is a purely abstract type in this interface, defined
 * entirely in terms of the operations. The client has no access to the
 * the record structure used to implement the actual type.
 */
typedef struct pathCDT *pathADT;

/*
 * Type: arcWeightFn
 * --------------------
 * Defines the type space of functions that can be used to determine the
 * weight of a particular arc. The function should take an arc and
 * return a double value which represents the weight of this arc.
 */
typedef double (*arcWeightFn)(arcADT arc);

/*
 * Type: arcMappingFn
 * --------------------
 * Defines the type space of functions can be used to map over the list
 * of arcs in order from the first to the last. Each function is called
 * with an arc and a client data pointer passed in from the caller.
 */
typedef void (*arcMappingFn)(arcADT arc, void *clientData);
```

```
/* Operations */

/*
 * Function: NewPath
 * Usage: path = NewPath(Distance);
 * -------------------------------------------
 * Dynamically allocates enough memory for the path and initializes it to
 * to represent an empty path. The weight function will be used to compute
 * the path sum as arcs are appended.
 */
pathADT NewPath(arcWeightFn weightFn);

/*
 * Function: FreePath
 * Usage: FreePath(path);
 * ----------------------
 * Frees the storage associated with the path.*/
void FreePath(pathADT path);

/*
 * Function: NewExtendedPath
 * Usage: newPath = NewExtendedPath(path, arc);
 * --------------------------------------------
 * Copies the path, then appends the arc to it.
 */
pathADT NewExtendedPath(pathADT path, arcADT arc);

/*
 * Function: AppendToPath
 * Usage: AppendToPath(path, arc);
 * --------------------------------
 * Appends a new arc to the end of the path.
 */
void AppendToPath(pathADT path, arcADT element);

/*
 * Function: StartOfPath
 * Usage: node = StartOfPath(path);
 * -------------------------
 * Returns the first node in the path.
 * If the path is empty, NULL is returned.
 */
nodeADT StartOfPath(pathADT path);
```

```
/*
 * Function: EndOfPath
 * Usage: node = EndOfPath(path);
 * -------------------------
 * Returns the last node in the path (ie the node that was appended most
 * recently). If the path is empty, NULL is returned.
 */
nodeADT EndOfPath(pathADT path);

/*
 * Function: CopyPath
 * Usage: copy = CopyPath(path);
 * ----------------------------
 * Returns a newly allocated path which is a copy of the given path, same
 * arcs, same order, same weighting function.
 */
pathADT CopyPath(pathADT path);

/*
 * Function: TotalPathDistance
 * Usage: sum = TotalPathDistance(path);
 * -------------------------
 * Returns the sum of the arc weights of the path. The weighting function
 * supplied when creating this path is used to determine the weight of each arc.
 * If the path is empty, 0 is returned. YOU WILL NEED TO IMPLEMENT THIS FUNCTION.
 */
double TotalPathDistance(pathADT path);

/*
 * Function: MapPath
 * Usage: MapPath(path, fn, clientdata);
 * ---------------------------------------------
 * Iterates through the arcs in the path and calls the function fn for each arc.
 * The function is called with arguments of the current arc and the clientdata
 * pointer. The clientdata value allows the client to pass extra state information
 * to the client-supplied function, if necessary. If no client data is required,
 * this argument should be NULL. YOU WILL NEED TO IMPLEMENT THIS FUNCTION.
 */
void MapPath(pathADT path, arcMappingFn mapFn, void *clientData);

#endif
```

```
/*
 * File: pqueue.h
 * -----------------------------------------------------
 * Defines an abstraction for the priority queue ADT. This version defines
 * the highest priority to be the lowest value. Items placed in the queue
 * are represented by a void * (a pointer to the item),  and a double (the
 * priority of the item). Note that some of the function names differ from
 * those used in the priority queue assignment, for compatibility with the
 * code in Chpater 16.
*/

#ifndef _pqueue_h
#define _pqueue_h

#include "genlib.h"

/*
 * Type: pqueueADT
 * ----------------------
 * This is the abstract type for a priority queue. The type definition
 * below is "incomplete". Clients know that pqueueADT is a pointer to
 * a struct tagged "pqueueCDT" but that's it. No details of the size,
 * field names, and types of the structure are visible to the client (and the
 * client has no need to access this information).
 */
typedef struct pqueueCDT *pqueueADT;


typedef struct
{
       void *info;
       double value;
} pqElementT;

/*
 * Function: NewPriorityQueue
 * Usage: queue = NewPriorityQueue();
 * ---------------------------------
 * Returns a new empty pqueueADT with no elements.
 */
pqueueADT NewPriorityQueue(void);
```

```
/*
 * Function: FreeQueue
 * Usage: FreeQueue(queue);
 * ------------------------
 * Frees all the storage associated with the queue.
 */
void FreeQueue(pqueueADT queue);

/*
 * Function: IsEmpty
 * Usage: if (IsEmpty(queue)) . . .
 * --------------------------
 * Returns TRUE if queue has no entries.
 */
bool IsEmpty(pqueueADT queue);

/*
 * Function: IsFull
 * Usage: if (IsFull(queue)) . . .
 * --------------------------
 * Returns TRUE if queue has no more room for entries. Clients can use
 * this to check if any further Insert operation will overflow the
 * ability of the queue to handle it. Some versions of the pqueue may
 * never return TRUE if they can always accommodate more entries.
 */
bool IsFull(pqueueADT queue);

/*
 * Function: PriorityEnqueue
 * Usage: PriorityEnqueue(queue, arc, length);
 * --------------------------
 * Adds the specified value to the queue. No effort is made to
 * avoid duplicates. If the queue is full, this function raises an error.
 */
void PriorityEnqueue(pqueueADT queue, void *newInfo, double newValue);

/*
 * Function: PriorityDequeue
 * Usage: best = PriorityDequeue(queue);
 * -----------------------------------
 * Removes the smallest priority element from the queue and returns its info. If the queue
 * is empty, this function raises an error.
 */
void *PriorityDequeue(pqueueADT queue);

#endif
```

```
/*
 * File: iterator.h
 * ----------------
 * Many abstract data types -- including those for such common
 * structures as sets and symbol tables -- represent collections
 * of values. In many cases, clients need to perform some
 * operation on each of the values in that collection. This
 * file provides a polymorphic interface for iterators, which
 * provides that capability in a very general way.
 */

#ifndef _iterator_h
#define _iterator_h

#include "genlib.h"
/*
 * Type: iteratorADT
 * -----------------
 * This abstract type is used to iterate over the elements
 * of any collection.
 */
typedef struct iteratorCDT *iteratorADT;


/* Exported entries */
/*
 * Functions: NewIterator, StepIterator, FreeIterator
 * Usage: iterator = NewIterator(collection);
 *        while (StepIterator(iterator, &element)) {
 *            . . . body of loop involving element . . .
 *        }
 *        FreeIterator(iterator);
 * -----------------------------------------------------
 * These functions make it possible to iterate over the
 * elements in any collection that supports iteration. Each
 * call to StepIterator advances the iterator and returns the
 * next element using the reference parameter. StepIterator
 * returns TRUE until the keys are exhausted, after which it
 * returns FALSE. The FreeIterator function releases any
 * storage associated with the iterator.
 */
iteratorADT NewIterator(void *collection);
bool StepIterator(iteratorADT iterator, void *ep);
void FreeIterator(iteratorADT iterator);

#endif
```