

CS 2213, Spring 2009
Assignment 8: Priority Queues and the Dijkstra Algorithm (v2)
Due: 5pm, Friday, April 24, 2009
Late Submissions: 5pm, Wednesday, April 29, 2009

Goal: The purpose of this assignment is to get practice working with heaps and adjacency lists.

Objectives: be able to write code to implement a priority queue using a heap; be able to write code to implement a edge-labeled undirected graph using adjacency lists; be able to implement Dijkstra's Shortest Path Algorithm.

Heap Test Program and Shortest Path Finder

This assignment provides the specification of two programs, the first `exercise-heap` is required the second `path-finder` is optional for extra credit.

The Heap Test Program (`exercise-heap`) will read in a sequence of pairs of city names and distances from a file into an array. Then change the array into a minimum-heap priority queue. Then, based on the standard input, be able to dequeue and print out the closest city, and be able to enqueue additional city-distance pairs.

For extra credit, you may write a `path-finder` program, reads in a list of distances between pairs of cities, puts those into a graph data structure, is then able to find the shortest path between any two cities (specified on the standard input), and shortest path and its distance.

Heap Test Program Input/Output Specification

The program should work with input and produce output exactly as described here. (Automated tools that do a character by character comparison may be used for grading.) For inputs that do not comply with this specification, your program may print an error message and exit; there should not be any inputs which would cause your program to have undefined behavior.

exercise-heap command line. The program executable should expect as a command-line argument the name of a file containing pairs of cities and their distance. Example:

```
./exercise-heap cities.txt
```

data file. The file specified on the command-line (e.g., `cities.txt`) will contain lines of up to 80 non-null characters. (If there are more than 80-characters—not counting '`\n`'—on a line, the additional characters should be silently discarded, producing the same result as if all of the lines were truncated to 80 characters.)

Each pair of lines contains the name of a city and a number which represent the distance of the city from some particular point). Lower numbers represent higher-priority. For example:

```
College_Station
201
Houston
210
Dallas
278
El_Paso
539
```

Standard Input. On the standard input the program can expect to receive lines denoting enqueue or dequeue commands. Dequeue commands consist solely of the characters dequeue, and enqueue commands consist of the word enqueue, a tab, the name of a city, a tab, and a distance. For example:

```
dequeue
enqueue→Austin→83
dequeue
dequeue
```

Standard Output For each dequeue, command, the exercise-heap program should remove and printout the name and distance (separated by a tab) of the city in the priority queue with the lowest associated distance. So for the inputs above, the program should output:

```
College_Station→201
Austin→83
Houston→210
```

Shortest Path Finder Input/Output Specification (Extra Credit)

The program should work with input and produce output exactly as described here. (Automated tools that do a character by character comparison may be used for grading.) For inputs that do not comply with this specification, your program may print an error message and exit; there should not be any inputs which would cause your program to have undefined behavior.

path-finder command line. The program executable should expect as command-line arguments (1) the name of a file containing pairs of cities and their distance and (2) the name of a city you are interested in finding distances from. Example:

```
./path-finder_cities-pairs.txt "El_Paso"
```

data file. The file specified on the command-line (e.g., city-pairs.txt) will contain lines of up to 80 non-null characters. (If there are more than 80-characters—not counting '\n'—on a line, the additional characters should be silently discarded, producing the same result as if all of the lines were truncated to 80 characters.)

Each group of three lines contains the name of one city, the name of a second city, and a number which represents the distance between those city. Lower numbers represent higher-priority. For example:

```

Austin
San Antonio
82
Austin
College Station
121
Houston
San Antonio
210
College Station
Houston
99
El Paso
San Antonio
539
Austin
Dallas
196
Houston
Dallas
239
El Paso
Dallas
637
Dallas
College Station
188

```

Your program should not have a fixed limit on the number of cities that can be considered.

Standard Output Based on the information in `city-pairs.txt`, the path-finder program should determine the shortest path to each of the destination from the city specified on the command-line, and then print out pairs of lines, the first of which, contains the distance and destination, and the second, prefixed by a tab, contains the full path of intermediate cities taken to reach the destination:

The output should be formatted as in the following example:

```

537 to San Antonio
----->El Paso->San Antonio
619 to Austin
----->El Paso->San Antonio->Austin
637 to Dallas
----->El Paso->Dallas
740 to College Station
----->El Paso->San Antonio->Austin->College Station
747 to Houston
----->El Paso->San Antonio->Houston

```

Implementation.

The `exercise-heap` program should store the information about cities/distances in a minimum heap stored in an array and keyed on the distance number. (Note: since this is a minimum heap, the smallest rather than the biggest number should be at the root.) For efficiency, the data from the `cities.txt` should be read in in order and then transformed into a heap. To 'heapify' the input from `cities.txt`, traverse the array backwards sifting down each element (see page 182 of the text book); this turns an n -element array into a heap in $O(n)$ time. Once the information is converted into a heap, the standard input commands should be processed using the heap as a priority queue (where smallest distance means highest priority).¹

The `path-finder` should read the data stored in the `city-pairs.txt` file and place into into an adjacency list, and then use Dijkstra's algorithm (see Section 7.3 in the text book), to calculate the shortest path to each of the cities.

You may find it useful to number the cities as you encounter them in the input and maintain an AA Tree mapping names to index numbers. You can then use those index numbers as indexes into dynamic array lists, vertex numbers, as a value to store in the heap, etc. It should be possible to use the same heap implementation for both programs.

Provide a Makefile that produces an executable `exercise-heap` and `path-finder` with the proper dependencies on your source files.

Test your program thoroughly using `valgrind` and a variety of valid and invalid inputs.

Submission

Please create a README file divided into three sections: NOTES TO GRADER, EXTRA CREDIT (optional), PROGRAM VALIDATION, and CERTIFICATION.

1. Under NOTES TO GRADER, add any notes, you wish the TA/instructor to consider while grading.
Please give a general overview of the state of your `exercise-heap` program. Describe the major data structures, and explain how the program is decomposed into files and functions. Describe any known problems with it program.
If `gcc` produces any warning or error messages (with the switches: `-std=c99 -pedantic -Wall -Wextra -Wstrict-prototypes -Wno-unused-parameter`), or if `valgrind` reports any problem, please list them, describe what you think they mean, and say whether you think it indicates a problem in your code or if `gcc/valgrind` is just being 'too picky.'
Please, also note approximately how many hours you spent to complete the assignment. This information will not effect your grade, but will be used for future course planning.
2. If you implemented the `path-finder` program (or attempted to), please add a section entitled EXTRA CREDIT. In this section describe the current state of your `path-finder` program and how it is implemented. Include a description of any known defects.
3. Under PROGRAM VALIDATION, please describe what you did to test your program or otherwise check that it is behaving correctly. In particular: What inputs did you try? How did you determine if the outputs were correct? Did you try any invalid inputs and make sure that your program doesn't crash? Did you use `valgrind` when running these tests? (See also the hints of Testing and Debugging given in the instructions to Assignment 5.)

1. You might find it easiest, to initially skip the input file and heapify algorithm, and just do the enqueue/dequeue commands first, and then come back to heapify later.

4. Under CERTIFICATION, please answer the following two questions in your README file:
 - (a) Did you use any code/materials other than those provided by or referenced by the instructor or TA in completing this assignment? If so, please cite the source (including its URL if it is online) and the nature and extent of your use of that material.
If such material is prohibited by the directions for this assignment, points may be deducted, but as long as such use is completely disclosed in the README file (including source, nature, and extent of use), it will not be considered to constitute scholastic dishonesty.
 - (b) Did you work with anyone or get help in completing this assignment from anybody other than the instructor or TA? (This includes mentors and tutors.) If so, please describe the nature and extent of the help received.
If such help or collaboration exceeds what is allowed for this assignment according to these directions, points may be deducted, but as long as the help or collaboration is completely disclosed in the README file, it will not be considered to constitute scholastic dishonesty.

Below your answer to those two questions, include the following statement:

```
I certify that except as noted above, the work I submit for
this assignment has been completed solely by me without any
outside help and without looking at any code for solving
this problem other than what is contained in
this document or included in other material/links provided
by the instructor or TA.
```

Below this statement, please write your name and date.

Use svn to commit your final submission by 5pm on the due date.

1. Make sure that you have all of the necessary files and they've all been added to svn.
2. You can continue to make changes, and re-commit as often as you want. The last version committed before 5pm on the due date will be graded.

Permissible Collaboration and Resources. This assignment is to be completed *individually*. You are allowed to discuss general strategies for solving assignments with fellow students or other individuals, but it is no longer a 'general strategy' if the discussion gets to the level of detail of what would be done in actual lines of code found in your programs. In addition, discussions of 'general strategy' should not be taking place while either party is editing their source code. (It is, however, perfectly acceptable to discuss C language constructs and concrete examples of them which are not taken directly from anyone's solution to an assignment.) In addition, you may seek more specific help from mentors and lab tutors in trying to understand why their code doesn't work. Furthermore, you may also seek help and guidance of any kind from the TA and instructor.

For this assignment, you may not consult any heap or Dijkstra's shortest path code other than that provided by the instructor, TA, or in the text book.