Worksheet 9

Overview

The workshop for week 10 will start with a short tutorial on Dijkstra's Algorithm and will involve writing a simple priority queue.

Tutorial Questions

For a directed graph with the following edges (from - to - weight):

a b 3

a d 7

b d 2

c e 6

d b 2

d c 5

d e 4

e d 2

Draw the graph and then run through Dijkstra's Algorithm starting from the vertex a.

Programming Exercises

Programming 9.1 Given a set of data and key pairs, write a priority queue, this will be directly useful for next week's workshop. It should have a *makeQueue* function, which creates an empty priority queue, an *enqueue* function, which adds a data, key pair in the correct place in the queue, a *dequeue* function, which takes the highest priority item from the queue (for usefulness next week, you can make the highest priority the item with the *lowest* key), an *update* function, which moves an item with a certain index up in the queue to where it belongs (this function can assume the rest of the keys remain unchanged) and an *empty* function, which checks whether the queue is empty.

NOTE: You don't need to make this task harder than it needs to be, you can implement this as an unsorted array. When a dequeue is called, you can search through all the items to find the cheapest item to dequeue and then remove it when a dequeue happens, replacing it with the last item in the array (so the items don't need to be shuffled). Likewise, in this case, the update function needn't do anything either. If you like, you can switch the underlying structure to a heap later and implement a lot of that more efficiently, but the input will be sufficiently small that this isn't much of a problem for the moment.

```
/* pqueue.h */
struct pqueue;

/* Creates an empty priority queue. */
struct pqueue *makeQueue();

/* Adds an item to a priority queue. */
void enqueue(struct pqueue *queue, int priority, void *data);

/* Takes the highest priority item from the queue. */
void *dequeue(struct pqueue *queue);
```

```
/* Updates the position of the given data item.
    Note: if you were writing this generally, you
    might like to include a priority data update
    function in your pqueue creation (like the
    hash table exercise) that allows you to read
    and update priority information and set and
    retrieve heap position information from
    an auxillary data structure (there's a lot
    of options here which are all fairly reasonable
    choices).
    For now, presume this does nothing (or make it
    read through the array and update priorities,
    it depends on your dequeue implementation). */
void update(struct pqueue *queue, int index);

/* Returns 1 if the queue is empty, 0 otherwise. */
int empty(struct pqueue *queue);

/* Frees all memory allocated by the queue given. */
void freeQueue(struct pqueue *queue);

/* pqueue.c */
#include <stdlib.h>
#include <assert.h>
/* #include "pqueue.h" */
struct pqueue {
    /* FILL IN THIS PART */

};

struct pqueue *makeQueue(){
    /* Write this */


    return NULL;
}

void enqueue(struct pqueue *queue, int priority, void *data){
    /* Write this */



}

void *dequeue(struct pqueue *queue){
    /* Write this */

    return NULL;
}

void update(struct pqueue *queue, int index){
    /* Do whatever is appropriate. */
    return;
}

int empty(struct pqueue *queue){
    /* Write this. */
    return 0;
}

void freeQueue(struct pqueue *queue){

}

/* main.c */
#include <stdio.h>
#include <stdlib.h>
/* #include "pqueue.h" */
#define SIZE 10
#define ARBITRARYTAKE 3

int main(int argc, char **argv){
    int inputs[SIZE] = {7, 1, 3, 5, 6, 8, 4, 2, 9, 11};
    int i;
```

```
    struct pqueue *queue = makeQueue();

    printf("Adding all items to priority queue\n");
    for(i = 0; i < SIZE; i++){
        printf("%d ",inputs[i]);
        enqueue(queue, inputs[i], &(inputs[i]));
    }
    printf("\n");
    if(empty(queue)){
        printf("ERROR: Queue empty after all items added\n");
    }

    printf("Removing first three items from priority queue\n");
    for(i = 0; i < ARBITRARYTAKE; i++){
        printf("%d ",*(int *) (dequeue(queue)));
    }
    printf("\n");

    printf("Updating priority 11 item to 1\n");
    inputs[SIZE - 1] = 1;
    /* Normally you'd find this information from somewhere that
        the priority queue will update, but as the update function
        doesn't do anything anything with this information, that
        doesn't really matter. */
    update(queue, SIZE - ARBITRARYTAKE - 1);
    printf("Taking rest of items from priority queue.\n");
    for(i = 0; i < SIZE - ARBITRARYTAKE; i++){
        if(!empty(queue)){
            printf("%d ",*(int *) (dequeue(queue)));
        }
    }
    printf("\n");

    if(! empty(queue)){
        printf("ERROR: Queue not empty after all items dequeued\n");
    }

    freeQueue(queue);

    return 0;
}
```

Programming Challenge 9.1 For this problem, you're the companion (voiced by Eddy Murphy) to a big brown bugbear who has been tasked by King Git to rescue a supposedly helpless princess from her castle in exchange for the rights to your marshland. [1]

The problem to solve is that you will need to find a path to the castle, but, due to a bad habit of getting distracted while talking, you can only move in an L shape (two spaces forward in up, down, left or right directions, followed by a single space in a direction perpendicular, or one space forward followed by two spaces perpendicular). The map is given as a 2D array, where empty spaces are marked with ' ' and spaces you can't traverse are marked with '#', your starting location is marked with 'S' and the castle is marked with 'C'. Print out a path to the castle. This needn't be the shortest path, but you should ensure that the path doesn't return to the same location at any point (no moves are redundant). [2]

```
#include <stdio.h>
#include <stdlib.h>
#define UNTRAVERSABLE '#'
#define START 'S'
#define CASTLE 'C'
#define XSIZE 6
#define YSIZE 6

/* Prints the path from the S symbol in the map
    to the C symbol which doesn't ever stop on
    any # symbols. The map given, of size xSize by
    ySize can be modified freely and needn't be
    kept in any way consistent with the original
    map. */
```

```
void printPath(char map[XSIZE][YSIZE + 1], int xSize, int ySize);

void printPath(char map[XSIZE][YSIZE + 1], int xSize, int ySize){
    int i, j;
    struct coordinatePair {
        int x;
        int y;
    };
    struct coordinatePair startPair = { .x = -1, .y = -1 };

    for(i = 0; i < XSIZE; i++){
        for(j = 0; j < YSIZE; j++){
            if(map[i][j] == START){
                startPair.x = i;
                startPair.y = j;
                /* Could also just let the loops
                    run, but may as well break
                    out of both. */
                break;
            }
        }
        if(startPair.x != -1){
            break;
        }
    }
    /* FILL IN */

}

int main(int argc, char **argv){
    char map[XSIZE][YSIZE + 1] = {
        "     C",
        "  #  #",
        "# # ##",
        "#  # #",
        "# #   ",
        " S    "
    };

    printPath(map, XSIZE, YSIZE);

    return 0;
}
```

[1]: The story, all names, characters, and incidents portrayed in this question are fictitious. No identification w ith actual persons (living or deceased), places, buildings, and products is intended or should be inferred.

[2]: This problem w as inspired by Tuenti Challenge 8's Challenge 4 ([https://contest.tuenti.net/Challenges?id=4])