Assignment 5: Hash Tables & Dijkstra's
Due at 11:59 pm on Sunday, 3/17/2024

This assignment will not be demoed. Instead, you will need to submit a readme text file (see "README.txt" below). TAs will grade this assignment during final's week!!!

In this assignment, you will implement a hash table (HT). The requirements for the assignment are described below.

Part 0. Download the skeleton code and unzip

For this assignment, you are provided with some starter code that defines the structures you'll be working with and prototypes the functions you'll be writing and also provides some data structures upon which to build a HT implementation. You may download the skeleton code for this assignment using the wget command:
wget https://classes.engr.oregonstate.edu/eecs/winter2024/cs261-020/assignments/assignment5.zip

To unzip the file, use the following command:
```
unzip assignment5.zip
```

Part 1. Implement a hash table

Your task for this assignment is to implement a hash table (HT).

You must build your HT using either a dynamic array or an array of linked list. A dynamic array-based HT resolves collisions using open addressing whereas array of linked list based HT resolves collisions using chaining.

The interface for the HT you'll implement (i.e. the structures and the prototypes of functions a user of the HT interacts with) is already defined for you in the file `hash_table.h`. Your job is to implement definitions for the functions that comprise this interface in `hash_table.c`.

Note that you may not modify the interface definition with which you are provided. Specifically, do not modify any of the already-defined HT function prototypes. We will use a set of tests to verify your implementation, and if you change the HT interface, it will break these tests, thereby (negatively) affecting your grade. Beyond these things, though, feel free to add any additional functions or structures that your HT implementation needs. In particular, you'll have to specify a definition of the main HT structure, `struct ht`, in `hash_table.c`.

The HT functions you'll need to implement are outlined briefly below. All of these functions use the type `struct ht`, which represents the HT itself and which you'll have to define in `hash_table.c`. For more details, including information on function parameters and expected return values, see the documentation provided in `hash_table.c`. Here are the functions you'll have to implement:

- `ht_create()` – This function should allocate, initialize, and return a pointer to a new HT structure.

- `ht_free()` – This function should free all the memory held within a HT structure created by `ht_create()` without any memory leaks. Note that this function only needs to free the memory held by the HT itself. It does not need to free the individual elements stored in the HT. This is the responsibility of the calling function.

- `ht_isempty()` – This function should return 1 if the HT is empty and 0 otherwise.

- `ht_size()` – This function should return the size of a given hash table.

- `ht_hash_func()` – This function should take a key, map it to an integer index value in the HT, and return the index. The hash algorithm is provided through a function pointer.

- `ht_insert()` – This function should insert an element with a specified key into a HT. **Note: if the key already exists in the hash table, update the value**. Collisions must be handled, <u>either by chaining or open addressing</u>. If using chaining, double the number of buckets when the load factor is >= 4; If using open addressing, double the array capacity when the load factor is >= 0.75, where load factor = (number of elements) / (hash table capacity). **This operation must have O(1) average runtime complexity.**

- `ht_lookup()` – This function should search for a given element in the HT with a provided key. **This operation must have O(1) average runtime complexity.**

- `ht_remove()` – This function should remove an element in the HT with a key provided. **This operation must have O(1) average runtime complexity.**

You are provided with a dynamic array implementation in `dynarray.c` and `dynarray.h` as well as a linked list implementation in `list.c` and `list.h` that you can use to implement your hash table, if you'd like. In addition to this dynamic array and linked list implementation, you may implement any additional helper functions you need to make your hash table work.

<u>Testing your work</u>

In addition to the skeleton code provided here, you are also provided with some application code in `test_hash_table.c` to help verify that your HT implementation, is behaving the way you want it to. In particular, the testing code calls the functions from `hash_table.c`, passing them appropriate arguments, and then prints the results. You can use the provided `Makefile` to compile all of the code in the project together, and then you can run the testing code as follows:

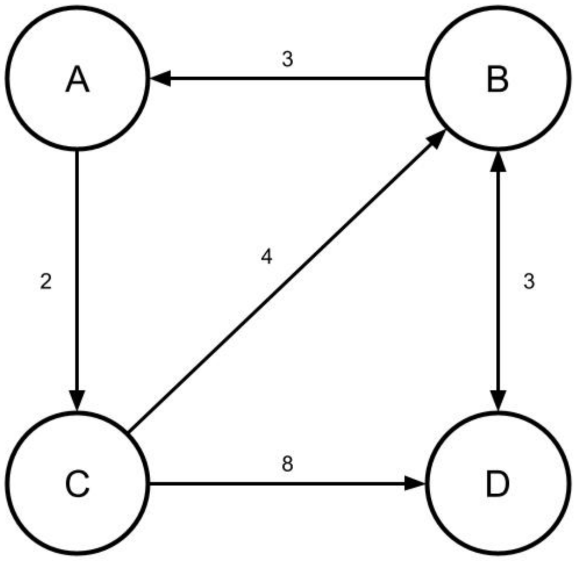
```
make
./test_ht
```

Example output of the testing program using a correct HT implementation is provided in the `example_output/` directory.

In order to verify that your memory freeing functions work correctly, it will be helpful to run the testing application through `valgrind`.

Part 2. Application: Dijkstra's algorithm

One of the most well-known applications of the priority queue data structure is in the implementation of Dijkstra's algorithm, which is used to find least expensive paths in a graph data structure. For example, below is an example of a graph:



In this graph, there are 4 nodes (A, B, C, and D) and there are edges connecting the nodes (the arrows). Each edge has a cost associated with it indicating how expensive it is to travel on that particular edge. For example, to move on the edge from C to B has a cost of 4. Note that the arrows representing some edges in this graph only point in one direction, and one can only travel in the direction indicated by the arrow. For example, in this graph you can travel directly from B to A (with a cost of 3), but you can't travel from A directly to B. Instead, to travel from A to B, you must go through C (i.e. taking the path A → C → B) or through both C and D (i.e. taking the path A → C → D → B).

Dijkstra's algorithm is used to find the least expensive path from one starting node to all other nodes in the graph. For example, starting from node A, Dijkstra's algorithm would find the following least-expensive paths to the other nodes in the graph:

- **B:** A → C → B with total cost 6

- **C:** A → C with total cost 2
- **D:** A → C → B → D with total cost 9

Dijkstra's algorithm is centered around a priority queue, which is used to help order the search for least-expensive paths. Here's pseudocode for Dijkstra's algorithm, where the starting node is `N_start`:

```
for every node N:
     cost[N] = infinity
     prev[N] = undefined

Q = new priority queue
insert N_start into Q with priority 0

while Q is not empty:
     // c is the total cost of the path to N
     c = Q.first_priority()

     // N_prev is the previous node on the path to N
     {N, N_prev} = Q.remove_first()

     if c < cost[N]:
          cost[N] = c
          prev[N] = N_prev

          // A neighbor is a node connected to N by an edge
          for each neighbor N_i of N:
               c_i = cost of edge from N to N_i

               // N is the previous node on the path to N_i
               insert {N_i, N} into Q with priority c + c_i
```

At the end of this algorithm, `cost[N]` will contain the cost of the least expensive path from `N_start` to N, and `prev[N]` will contain the node before `N` in that path. The entire path to `N` can be computed by tracing backwards:
`prev[N],prev[prev[N]],prev[prev[prev[N]]],` ...

If you prefer a video tutorial instead, here is one that explains how Dijkstra's algorithm is implemented using a priority queue.

Your job here is to implement Dijkstra's algorithm to find the shortest paths in the graph specified in the file `airports.dat`. Here are some things you'll have to do to make this work:

- Read the data from the file `airports.dat`. You can use the functions `fopen()` and `fscanf()`, respectively, to open and read data from the file. (Note: the provided skeleton code already opens the file and loads the first two integers into variables!) The file has a special format that will make it easier to read:

```
<num_nodes>
<num_edges>
<node_i> <node_j> <cost_i_j>
...
```

Specifically, the first line of the file contains the number of nodes `n` in the graph, and the second line contains the number of edges specified in the file. The remaining lines each specify one edge in the graph. Each edge specification consists of 3 values:

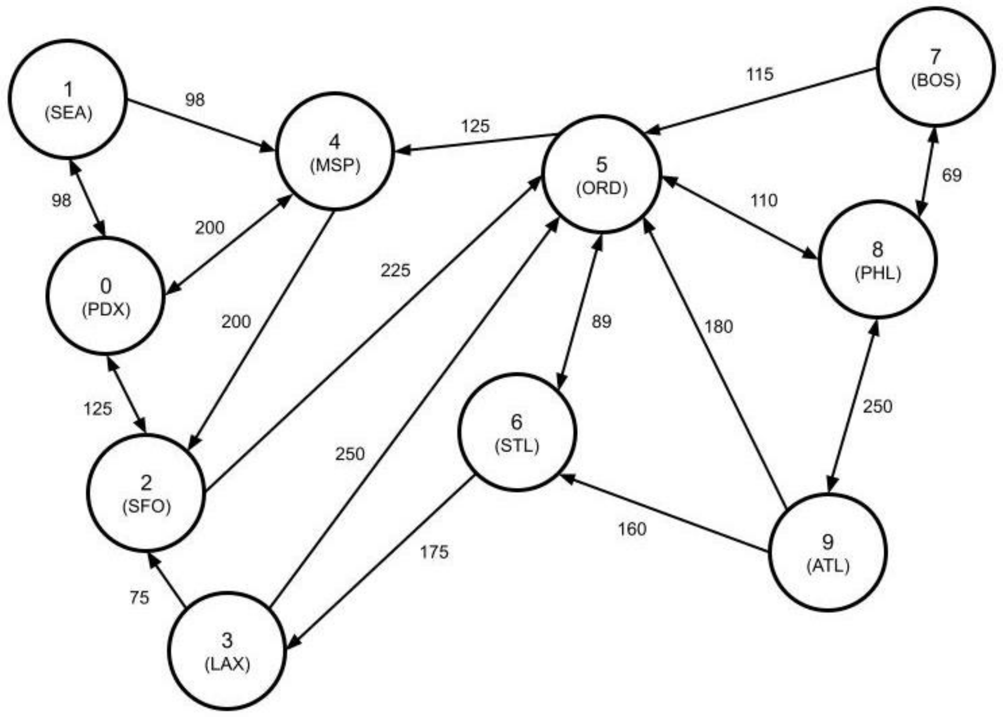
- An integer index between 0 and `n-1` representing the source node of the edge (i.e. the edge goes *from* this node).
- An integer index between 0 and `n-1` representing the destination node of the edge (i.e. the edge goes *to* this node).
- The cost associated with the edge.

- Store the graph data you read in an easily usable format. I'd recommend storing the graph data as an adjacency matrix. For a graph with `n` nodes, an adjacency matrix would be an `n` x `n` matrix (i.e. a 2D array), where the entry at location (`i`, `j`) is simply the weight of the edge between node `i` and node `j` or 0 if there is no edge between nodes `i` and `j`. For example, an adjacency matrix for the graph pictured above would look like this:

```
        A  B  C  D
      +------------
    A | 0  0  2  0
    B | 3  0  0  3
    C | 0  4  0  8
    D | 0  3  0  0
```

- Once the graph is read and stored, implement Dijkstra's algorithm as outlined above. Importantly, you'll have to figure out how to store the necessary data in your priority queue (*hint: storing both the current node and the previous node in your queue will probably require a custom* `struct`). For this assignment, you can assume that node 0 is the starting node. In other words, you want to find the least-cost path from node 0 to each other node in the graph. Once you've found these paths, print them out, along with their total cost.

Below is a depiction of the graph defined in airports.dat:



This graph represents a hypothetical set of flights between airports in the US. In this graph, the nodes represent airports in the US, and each edge weight represents the cost of a flight between two specific airports. Your goal in finding the least-cost paths using Dijkstra's algorithm here is to figure out the least expensive set of flights to get from PDX to each of the other airports. (As will become clear once you've solved the problem, we're not necessarily concerned with minimizing the number of layovers here.)

<u>README.txt</u>

Assignment 5 will not be demoed. TAs will grade on their own during Finals Week. Besides your code, you will submit a README.txt that outlines how your program is compiled and run. Failure to submit a README.txt will result in a deduction as well as any penalties that may be incurred as a result of incorrect use of your program. The README.txt needs to include the following information:

1. Your name and ONID
2. **Description**: One paragraph advertising what your program does (for a user who knows nothing about this assignment, does not know C, and is not going to read your code). Highlight any special features.
3. **Instructions**: Step-by-step instructions telling the user how to compile and run your program. If you expect a certain kind of input at specific steps, inform the user what the requirements are. Include examples to guide the user.
4. **Limitations**: Describe any known limitations for things the user might want or try to do but that program does not do/handle.

Submission

In order to submit your homework assignment, you must create a **zip file** that contains `assignment5/` folder with your implementation. This zip file will be submitted to TEACH . In order to create the zip file, go to the directory where you can access the `assignment5/`, and use the following command:

```
    zip assignment5.zip assignment5 -r
```

Do not forget to submit your README.txt !!!