Lab 6: Heaps and Hashes

ELEC 278: Fundamentals of Information Structures

Learning Goals

- Demonstrate the heap structure and core heap operations such as insertion, deletion, and extraction of the minimum or maximum element.
- Recognize heaps' usage in practical contexts and use heaps to efficiently solve several problems.
- Explore the implementation of hashing strings.

Instructions

Download the zipped folder `lab6.zip` from OnQ and unzip it. Next, open the lab6 folder that you extracted in your code editor. There are four folders inside, each is used for one of the exercises in this lab.

To receive full marks, your code must be able to correctly handle corner cases. Comment your code and be ready to discuss how your code handles corner cases with your TA. Generate a list a cases that you will test your code against with your TA.

Grading (fraction of marks for each exercise)

0	No code.
1/2	Functioning code, but fails corner cases, if any, or not readable/not well-explained.
1	Functioning code that handles corner cases

Exercise 1. (0.5 mark)

In the folder *intHeap*, the file *main.c* contains the following C structure for a min-heap.

```
#define MAX_SIZE 100

typedef struct MinHeap {
    int size;
    int data[MAX_SIZE];
} MinHeap;
```

and the functions:

- `insertMinHeap()` to insert a new element and restore the min-heap property of the heap.
- `extractMin()` to remove the minimum element (the root) of the heap.

- `heapify()` to restore the min-heap property of the heap.

Study the code and comment it to explain the operations. Prepare short descriptions of the operations of each function.

Exercise 2. (1 mark)

The file *main.c* in the folder *smallest* contains an empty function `int findKthSmallest(int nums[], int sizeNums, int k)`. Implement the function such that, given an unsorted array of integers, and an integer `k`, it returns the kth smallest element in the array.

For example, if `nums` = [2,3,1,5,4], `sizeNums` = 5, and `k` = 2, the function returns 2.

Since this is a lab about heaps, use a heap to solve the exercise. Feel free to use the appropriate code from exercise 1. Add appropriate test cases to the main function to ensure your code works.

Exercise 3. (4 marks)

Backbone network routers handle millions of packets every second. Some network packets have higher priority than others. The structure of a packet is defined in the file *main.c* in the folder *router* as:

```
typedef struct {
    int src;      // src address of packet
    int dst;      // dst address of packet
    int priority; // packet priority (low value = high priority)
    char * data;  // packet data
} Packet;
```

The activity of the router is represented as a set of commands that is processed by the main function. The command data structure is defined as:

```
typedef enum {
    insertPacket, getPacket, endOfCommands
} CommandName;

typedef struct {
    CommandName comm;
    Packet packet;
} Command;
```

Use a C struct to implement a heap of packets to represent the operation of the router with the following functions:

- `void initRouter(Router * r)` – initialize the heap representing the router.
- `bool addPacket(Router * r, Packet p)` – add a packet to the heap;
- `bool getNextPacket(Router * r, Packet *p)` – get the next packet from the heap
- `void heapify(Router * r, int parent)` - restore the heap structure of the router;

Your main function will process an array of commands, and call the above functions as appropriate. It will print, as output, details of each packet as it is sent on to its destination.

Exercise 4. (4 marks)

There are many ways in which a hash value may be generated for a given string. This exercise explores three of them:

1. The hash is the sum of the ascii values of the characters in the string. The hash of "abc" is the sum of the values 97, 98 and 99, which sums to 294.
2. The same as (1), but multiple the running total by some constant at each step. For example, for a constant mutliple 31, the hash of "abc" is $(97 * 31 + 98) * 31 + 99 = 96354$.
3. The `djb2` hash algorithm. This uses an initial hash value of 5381. Next, the algorithm iterates through the string. At each step, it multiplies the hash by 33 and then adds the acsii value of the current character to it. Since 33 is $2^32 + 1$, this can be efficiently implemented in bit operations, but you will use the simpler approach of multiplying by 33. The hash of "abc" is 193485963.

In all cases, the value computed from the string must be reduced to the size of the hash array using the modulo (%) operator.

The performance of a hash table is dependent on the performance of the hash function used. You will complete a program that will count the number of hash collisions for each of these hash tables from a representative sample of strings.

The file *main.c* in the folder *hashes* has a list of 40 unique words extracted from the solution to a 3rd year programming assignment. It declares a hash table that contains 60 entries.

It contains empty function bodies for each of the hash functions as well as a function to erase the hash table. For each hash function, you will hash each of the 40 strings and increment the appropriate element of the *hashTable* to count the number of words that hash to that location (bucket). You do not have to store the strings or the computed values in the table, just count the number of strings that hash to that location. Use the *eraseTable()* function to erase the table between functions.

Find the maximum number of collisions that occurs at any one location (i.e. the maximum bucket size) as well as the total number of collisions. Also count how many locations (buckets) were not used by each hash function. Use the following cluster metric formula to get an indication of the degree in which the values clustered:

$$\sum_i x_i^2/n$$

where x_i is the number of entries that hash to location (or bucket) i, and n is the total number of locations.

A clustering metric estimates how evenly a hash function spreads elements across buckets in a hash table. If the clustering measure is less than 1, the hash function is spreading elements more evenly than a random hash function. If the clustering measure is significantly greater than one, the hash function is not utilizing a substantial fraction of buckets.

Empty functions have been provided both for the counting function and for the cluster metric.

Thus for each of the three hash functions, your program will report the maximum number of collisions at any one bucket, the total number of collisions, the number of empty buckets, and the cluster metric.