CSE 3100 Systems Programming	Fall 2023
Homework #4	

Please submit your work to gradescope. Note a Makefile is provided.

Exercise 1. (100 points) Epidemic Model

In this exercise, we write code to simulate the spread of contagious diseases. In particular, we assume:

1. There are m hosts of a certain contagious disease.
2. These hosts are located on a two dimensional grid.
3. There are three different types of hosts:
   (a) *susceptible* hosts - have not been infected by the disease
   (b) *infected* hosts - have been infected by the disease and have not yet recovered
   (c) *recovered* hosts - have recovered from the disease (Note: recovered hosts will not be infected by the disease any more)

Initially there is a single infected host located at $(0, 0)$, and the rest of the $m - 1$ hosts are susceptible hosts, and they are randomly located on a 2D grid with edges that warp around (like Pacman). The center of the grid is at $(0, 0)$, and the four corners are at $(-k, -k), (k, -k), (k, k)$ and $(-k, k)$, respectively. When a host moves outside of a border, they will appear on the opposite side. For example, if a host at (x, k) moves up, they will appear at $(x, -k)$, where $-k \leq x \leq k$. The hosts move according to a 2D random walk. Use `rand() % 4` to generate a direction according to the following mapping:

- 0 - up
- 1 - right
- 2 - down
- 3 - left

When a susceptible host is at the same location as one or more infected hosts, the susceptible host becomes infected and will be contagious at the next time step. An infected host will become a recovered host T time steps after they are infected. As time goes by, the number of infected hosts will eventually drop to 0. When this happens, the number of susceptible and recovered hosts will not change any more. The simulation will be stopped at this moment. In the end, we print out the proportions of hosts which are susceptible, infected, and recovered. Hint: the number of infected should always be zero at the end of the simulation!

We use a structure Host to describe a host. Note how we use enum in the code.

```
enum TYPE {S, I, R};

typedef struct Host
{
    int id;
    int x, y;
    int t;
    TYPE type;
} THost;
```

Here, id is the id of a host, (x, y) is the coordinates of a host, t is the duration since a host became infected, and type is the type of a host.

To speed up the simulation, we implement a hash table to save the location information of all the infected nodes. The hash table is implemented as an array of linked lists. The size of the hash table is N.

In each round of the simulation, for each infected host, a copy of this host is inserted into the hash table according to the hash value calculated based on the coordinate of the host.

To insert an infected host into the hash table, we first map its (x, y) coordinate to an integer. We then apply a hash function (that is given in the code) to the resulting integer. A modulo operation is applied on the result to ensure it fits into the range of the index of an array of size N. The infected node will be inserted into the front of the corresponding linked list.

For each susceptible host, we apply the same mapping and hash function to look up the corresponding linked list and see whether there is an infected host with the same coordinates. If there is, this susceptible host becomes an infected host.

Below is the main() function for this program. Note how we use command line arguments.

Note k is the grid size parameter; m is the number of hosts; T is the duration parameter for an infected host to recover; and N is the size of the hash table.

```
int main(int argc, char *argv[])
{

    if(argc != 5)
    {
        printf("Usage: %s k m T N\n", argv[0]);
        return 0;
    }

    int k = atoi(argv[1]);
    int m = atoi(argv[2]);
    int T = atoi(argv[3]);
    int N = atoi(argv[4]);

    assert(k >= 0 && k <= 1000);
    assert(m >= 1 && m <= 100000);
    assert(T >= 1);
    assert(N > 0 && N <= 100000);
    srand(12345);

    //initialize hosts
    THost hosts[m];

    hosts[0].id = 0;
    hosts[0].x = 0;
    hosts[0].y = 0;
    hosts[0].t = 0;
    hosts[0].type = I;
```

```
    for(int i = 1; i < m; i ++)
    {
        hosts[i].id = i;
        hosts[i].x = rand() % (2*k + 1) - k;
        hosts[i].y = rand() % (2*k + 1) - k;
        hosts[i].t = 0;
        hosts[i].type = S;
    }

    //initialize linked lists
    node *p_arr[N];

    for(int i = 0; i < N; i++)
    {
        p_arr[i] = NULL;
    }
    node *r = create_node(hosts[0]);
    int index = hash(idx(hosts[0].x, hosts[0].y, k)) % N;
    add_first(&(p_arr[index]), r);

    //simulation
    while(one_round(hosts, m, p_arr, N, k, T));

    return 0;
}
```

Below are outputs from a few sample runs. We can use these outputs to check our code.

```
$ time ./epidemic 150 100000 10 100000
    S        I        R
0.015700 0.000000 0.984300

real 0m2.991s
user 0m2.988s
sys 0m0.000s
$ time ./epidemic 150 100000 10 10000
    S        I        R
0.015700 0.000000 0.984300

real 0m2.765s
user 0m2.764s
sys 0m0.000s
$ time ./epidemic 150 100000 10 1000
    S        I        R
0.015700 0.000000 0.984300

real 0m3.424s
user 0m3.420s
sys 0m0.000s
$ time ./epidemic 150 100000 10 100
    S        I        R
0.015700 0.000000 0.984300
```

```
real 0m5.868s
user 0m5.864s
sys 0m0.000s
$ time ./epidemic 150 100000 10 10
    S        I        R
0.015700 0.000000 0.984300

real 0m27.748s
user 0m27.744s
sys 0m0.000s
$ time ./epidemic 150 100000 10 1
    S        I        R
0.015700 0.000000 0.984300

real 2m49.609s
user 2m49.604s
sys 0m0.004s
```

The above outputs show how the size of the hash table N affects the time needed for the simulation. Note the simulation times range from 2.765 seconds to 2 minutes and 50 seconds. The longest time corresponds to using a single linked list, while the shortest time corresponds to using a hash table of size $10,000$. Note when we further increase the hash table size, the performance starts to drop. Think why this is the case.