Name: Chitwan Goel
Roll No. 210295

CS610 Assignment 5

1 Problem 1

The directory `problem1-dir` contains the source files `hashset.hpp` and `problem1.cpp`. `hashet.hpp` hsa the class definition of `hashmap`, along with the definitions of various hash functions. The file `problem1.cpp` has the driver code along with 3 unit test cases named `unit_tc<i>`. There is a script `run.sh`, which takes 5 arguments and compiles and executes the code based on those.

```
chmod +x run.sh
./run.sh <TBB_FLAG> <NUM_OPS> <RUNS> <PERCENT_ADD_OPS> <PERCENT_REMOVE_OPS>
```

`TBB_FLAG` has to be set 1 or 0 depending on whether we want to use the `TBB` implementation or our custom implementation.

Please ensure that the `bin` files for random data are in the current working directory before running the code.

The execution will print the result of the unit test cases and also the result for the files given with random values. For the files given, if we run our custom implementation, it will run for 36 combinations of hash functions, and print the results accordingly.

Below table summarises the results for `NUM_OPS = 1e7`, `ADD = 40`, `REM = 30` and 8 threads on `csews172`.

Hash1	**Hash2**	**Insert Time (ms)**	**Delete Time (ms)**	**Search Time (ms)**	**Insert Rate (ops/ms)**	**Delete Rate (ops/ms)**	**Search Rate (ops/ms)**
INTEL_TBB_HASHMAP		7	5	5	571428.57	600000	600000
Modulo Hash	Modulo Hash	309	464	11	12945.0	6465.5	272727.3
Modulo Hash	Modulo Hash 2	198	425	11	20202.0	7058.8	272727.3
Modulo Hash	Multiplicative	222	446	11	18018.0	6724.2	272727.3
Modulo Hash	XOR Shift Hash	312	461	11	12820.5	6507.6	272727.3
Modulo Hash	Jenkins Hash	268	470	11	14925.4	6382.9	272727.3
Modulo Hash	FNV1A Hash	271	494	11	14760.1	6072.7	272727.3
Modulo Hash 2	Modulo Hash	247	449	11	16112.9	6679.6	272727.3
Modulo Hash 2	Modulo Hash 2	353	499	11	11342.5	6012.0	272727.3
Modulo Hash 2	Multiplicative	254	448	11	15748.0	6705.4	272727.3
Modulo Hash 2	XOR Shift Hash	252	455	11	15873.0	6593.4	272727.3
Modulo Hash 2	Jenkins Hash	259	439	11	15403.5	6830.3	272727.3
Modulo Hash 2	FNV1A Hash	255	464	11	15686.3	6474.1	272727.3
Multiplicative	Modulo Hash	302	505	11	13245.0	5940.0	272727.3
Multiplicative	Modulo Hash 2	236	430	12	16948.9	6988.4	250000.0
Multiplicative	Multiplicative	232	451	11	17241.4	6645.0	272727.3
Multiplicative	XOR Shift Hash	263	473	11	15287.8	6341.0	272727.3

Hash1	**Hash2**	**Insert Time (ms)**	**Delete Time (ms)**	**Search Time (ms)**	**Insert Rate (ops/ms)**	**Delete Rate (ops/ms)**	**Search Rate (ops/ms)**
Multiplicative	Jenkins Hash	254	462	11	15748.0	6498.9	272727.3
Multiplicative	FNV1A Hash	238	459	11	16849.6	6549.7	272727.3
XOR Shift Hash	Modulo Hash	265	474	11	15094.3	6330.8	272727.3
XOR Shift Hash	Modulo Hash 2	271	465	11	14757.0	6451.6	272727.3
XOR Shift Hash	Multiplicative	286	498	11	13986.0	6032.1	272727.3
XOR Shift Hash	XOR Shift Hash	212	431	11	18867.9	6967.1	272727.3
XOR Shift Hash	Jenkins Hash	224	446	11	17857.1	6726.5	272727.3
XOR Shift Hash	FNV1A Hash	220	428	11	18181.8	7028.0	272727.3
Jenkins Hash	Modulo Hash	239	454	11	16736.4	6617.2	272727.3
Jenkins Hash	Modulo Hash 2	260	460	11	15384.6	6521.7	272727.3
Jenkins Hash	Multiplicative	384	468	11	10416.7	6405.6	272727.3
Jenkins Hash	XOR Shift Hash	293	465	11	13683.7	6451.6	272727.3
Jenkins Hash	Jenkins Hash	266	483	11	15037.6	6215.3	272727.3
Jenkins Hash	FNV1A Hash	301	458	11	13322.6	6548.7	272727.3
FNV1A Hash	Modulo Hash	290	464	11	13793.1	6474.1	272727.3
FNV1A Hash	Modulo Hash 2	200	438	11	20000.0	6835.8	272727.3
FNV1A Hash	Multiplicative	277	451	11	14487.4	6650.3	272727.3
FNV1A Hash	XOR Shift Hash	243	460	11	16460.9	6521.7	272727.3
FNV1A Hash	Jenkins Hash	264	449	11	15151.5	6670.4	272727.3
FNV1A Hash	FNV1A Hash	289	462	11	13802.1	6497.8	272727.3

The best throughput for insert is observed when we use the `modulo hash` and `modulo hash 2` as index and step functions. However, the difference between the performances of various hash functions is not much pronounced.
Search has the largest throughput because I didn't use locks in my implementation of search. As it was given in the problem that the concurrency is only limited to batch operations, search operation doesn't need a lock. Since any concurrent thread would anyhow be searching only, there is no concurrent read and write.
Moreover, I used reader-writer locks along with mutexes for every entry to handle other scenarios. During resize, the thread which goes into the `resize` function, takes the writer lock and the threads going into all other operations take the reader locks. This ensures that only one thread is resizing at any moment.

2 Problem 2

The directory `problem2-dir` contains the source files. `cstack.hpp` has the implementation of the lock-free stack and `problem2.cpp` contains the driver code. Just run the script `run.sh`, it will generate `problem2.out`, as well as run the executable for various combination of `NUM_OPS` and `NUM_THREADS`.

```
chmod +x run.sh
./run.sh
```

OR

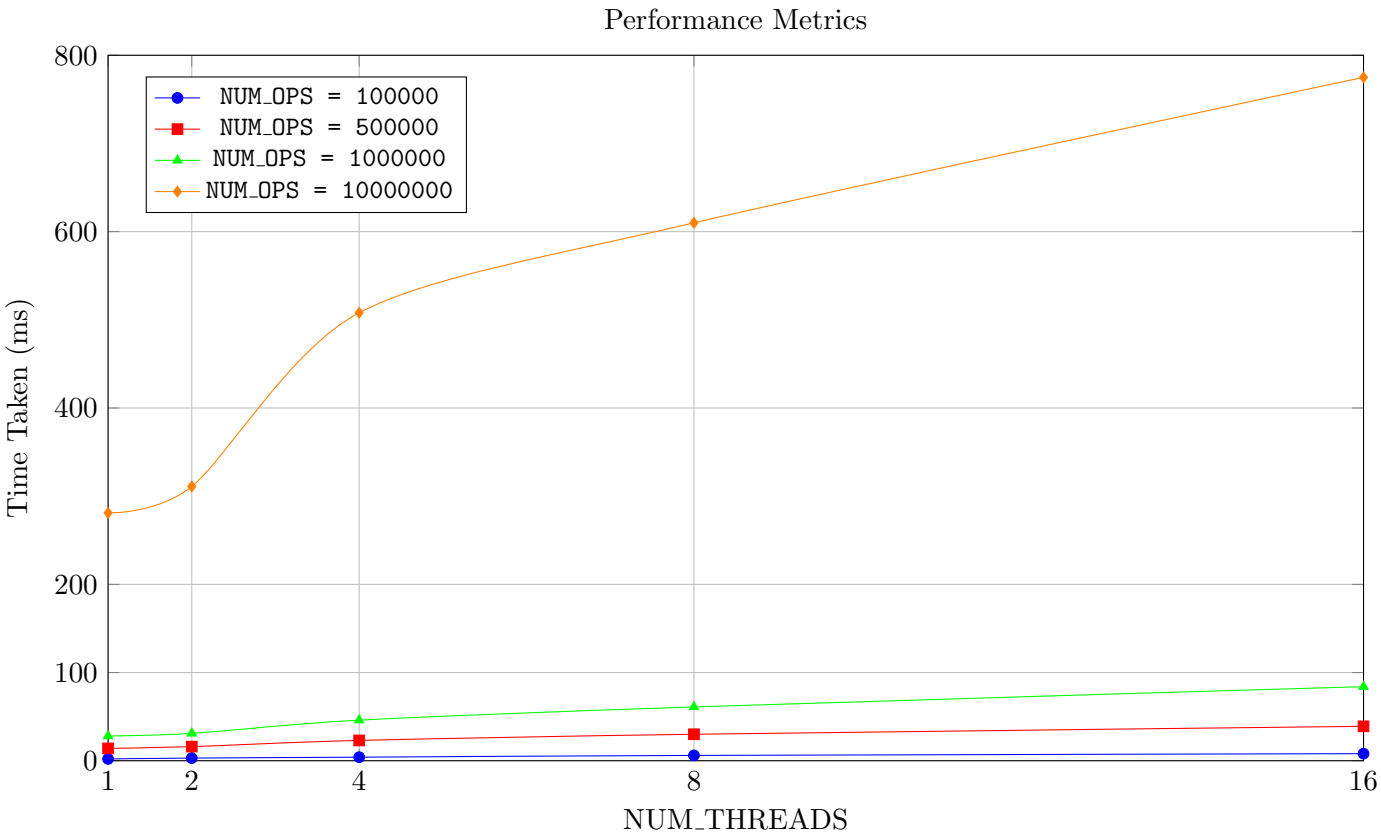
```
g++ -std=c++17 -O3 problem2.cpp -o problem2.out -latomic
./problem2.out <NUM_OPS> <NUM_THREADS>
```

Please ensure that the `bin` files for random data are in the current working directory before running the code.

Below graph and table represents the results I obtained by running the code on `csews172`.

NUM_THREADS	NUM_OPS	TIME TAKEN (ms)
1	100,000	2
2	100,000	3
4	100,000	4
8	100,000	6
16	100,000	8
1	500,000	14
2	500,000	16
4	500,000	23
8	500,000	30
16	500,000	39
1	1,000,000	28
2	1,000,000	31
4	1,000,000	46
8	1,000,000	61
16	1,000,000	84
1	10,000,000	281
2	10,000,000	311
4	10,000,000	508
8	10,000,000	610
16	10,000,000	775

Table 2: Performance Metrics for Different NUM_OPS and NUM_THREADS


Performance Metrics
Time Taken (ms)
NUM_THREADS
NUM_OPS = 100000
NUM_OPS = 500000
NUM_OPS = 1000000
NUM_OPS = 10000000
800
600
400
200
100
0
1
2
4
8
16