Lab 1 - CS4532

180192L - N.M. Gamlath
180474E - I.S. Pietersz

1. To pre-generate the workload,
   - A random number generator function was implemented, which yields a list of unique random numbers in the given range. This was used to generate arguments for each type of operation.
   - The arguments for Member and Insert operations were generated independently.
   - Half of the arguments for Delete operations were taken from the Insert arguments, and the other half were newly generated.
     (This was done to increase the chances of actually deleting an existing item)
   - The operation names and arguments were put into a vector and shuffled.
   - Each thread would receive an equal slice of this vector.

2. Linked lists have been implemented accordingly in the code.
   The linkedist.h header file defines the LinkedList class and functions, which are then implemented by the relevant .cpp files.
   See README.txt for an explanation of how to compile and run the code.

3. Calculating sample size

$$Confidence\ interval = Sample\ mean \mp Error$$

$$\bar{x} \mp z\frac{s}{\sqrt{n}} = \bar{x}\left(1 \mp \frac{r}{100}\right)$$

This yields $$n = \left(\frac{100 \times z \times s}{r \times \bar{x}}\right)^2$$

z = 1.960 for 95% confidence level
r = Accuracy percentage = 5
s = Sample standard deviation
n = No of observations

Start with initial sample of 100, then calculate an appropriate sample size using this equation.
This workflow is implemented in the code.

Machine specifications:
CPU : Intel(R) Core(™) i7-8550U CPU @ 1.80GHz
Cores : 4
Logical processors : 8
Memory : 7.9 GB
Memory speed : 2400 MHz
OS: Windows 10 Pro, version 22H2
System type : 64-bit operating system, x64-based processor

Results

Times are given in milliseconds

Case 1 : n = 1,000 and m = 10,000, m_Member = 0.99, m_Insert = 0.005, m_Delete = 0.005

Implementation	**No. of threads**							
	1		**2**		**4**		**8**	
	Avg	**Std**	**Avg**	**Std**	**Avg**	**Std**	**Avg**	**Std**
Serial	31.01	7.81						
One mutex for entire list	32.7	8.77	47.3	10.8	66.40	12.45	137.14	20.51
Read-Write lock	38.7	10.3	22.1	8.83	15.9	10.05	21.3	13.34

Case 2 : n = 1,000 and m = 10,000, m_Member = 0.90, m_Insert = 0.05, m_Delete = 0.05

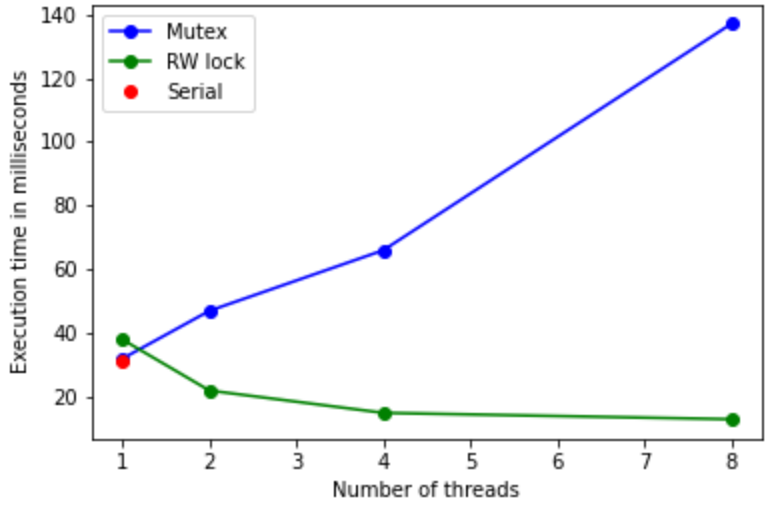
Implementation	**No. of threads**							
	1		**2**		**4**		**8**	
	Avg	**Std**	**Avg**	**Std**	**Avg**	**Std**	**Avg**	**Std**
Serial	47.2	8.12						
One mutex for entire list	41.11	9.31	51.56	11.13	54.0	12.4	66.1	15.2
Read-Write lock	40.8	12.29	22.1	9.43	18.5	10.1	15.61	11.4

Case 3 : n = 1,000 and m = 10,000, m_Member = 0.50, m_Insert = 0.25, m_Delete = 0.25

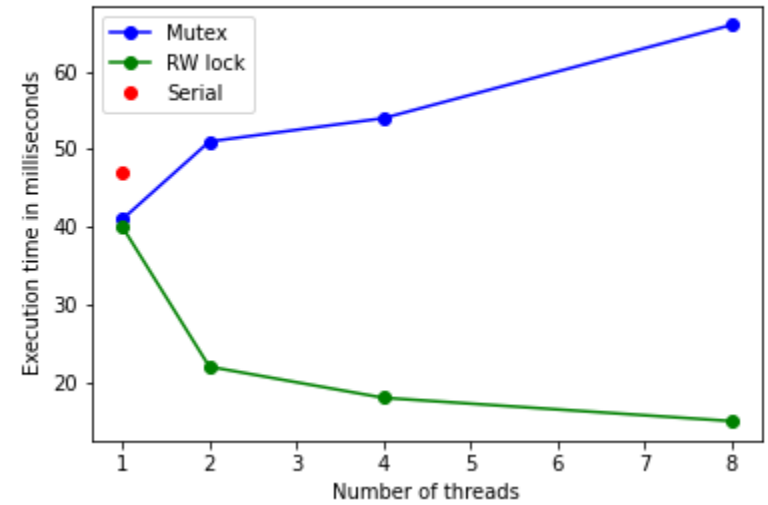
Implementation	**No. of threads**							
	1		**2**		**4**		**8**	
	Avg	**Std**	**Avg**	**Std**	**Avg**	**Std**	**Avg**	**Std**
Serial	71.8	13.08						
One mutex for entire list	68.4	9.49	85.1	12.49	104.7	18.89	113.0	40.5
Read-Write lock	147.1	18.62	80.3	20.12	71.4	9.41	60.31	12.1

4. Plots of execution time for each case

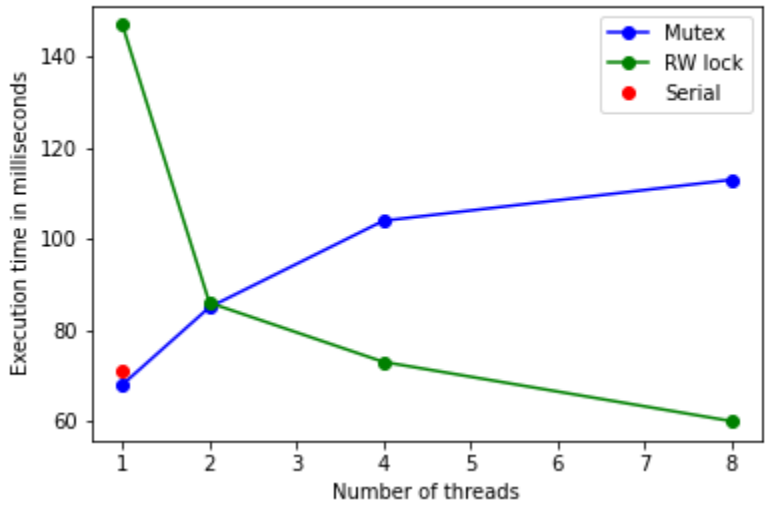
Case 1: n = 1,000 and m = 10,000, m_Member = 0.99, m_Insert = 0.005, m_Delete = 0.005



Case 2: n = 1,000 and m = 10,000, m_Member = 0.90, m_Insert = 0.05, m_Delete = 0.05



Case 3: n = 1,000 and m = 10,000, m_Member = 0.5, m_Insert = 0.25, m_Delete = 0.25



5. Comments on the observations:

   - Considering the execution times across cases, the results are almost always in the pattern Case 1 < Case 2 < Case 3
     This is due to the increasing ratio of insert and delete functions from Case 1 to 2 to 3. Insert and delete take slightly more time than member operations.
     In the multithreaded implementations, this effect is more noticeable, because the insert and delete operations will trigger locking / unlocking actions, which add more overhead.

   - The mutex implementation shows no improvement in execution time as the number of threads increase. In fact, it increases with the number of threads.
     This is because having a mutex on the entire list makes it functionally similar to a serial implementation. Threads trying to operate on the list simultaneously will be forced to wait until the mutex lock can be acquired from the thread currently holding it.
     The overhead due to the mutex implementation, thread waiting and switches therefore causes the execution time to increase, not decrease.

   - The read-write lock implementation shows the expected behaviour of decreasing execution time with increasing number of threads.
     This is because multiple readers are allowed in at the same time. Threads will be suspended only when a write operation (insert or delete) begins.

   - Compared to the serial implementation, execution time is consistently improved only by the read-write lock implementation with 4 or 8 threads.