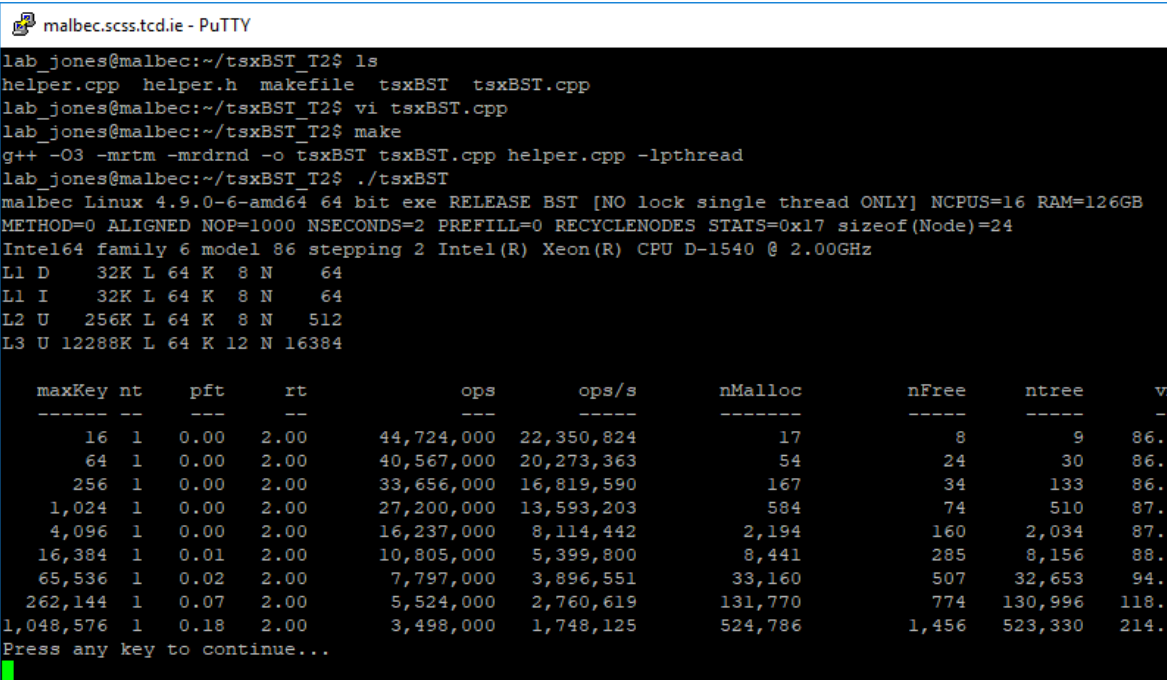


## CS4021/4521 Tutorial 2

- Q1. Implement and compare the performance of (i) a binary search tree (BST) protected by a lock (ii) a *lockless* BST using HLE and (iii) a *lockless* BST using RTM.

The source code for a test framework can be downloaded from the CS4021/4521 web site. There are four files `tsxBST.cpp`, `helper.h`, `helper.cpp` and a simple makefile. The easiest way to get the test framework running is to download the four files into a directory on malbec, type `make` and run the executable `./tsxBST`. Remember that the final code for this project needs to run on a CPU that supports TSX such as malbec.

Alternatively, the files can be placed in a Visual Studio or Eclipse project. If the “Visual C++ for Linux Development” feature is installed [Tools][Get Tools and Features...], the files can be compiled and run remotely on malbec using Visual Studio on a Windows PC.



```
malbec.scss.tcd.ie - PuTTY
lab_jones@malbec:~/tsxBST_T2$ ls
helper.cpp helper.h makefile tsxBST tsxBST.cpp
lab_jones@malbec:~/tsxBST_T2$ vi tsxBST.cpp
lab_jones@malbec:~/tsxBST_T2$ make
g++ -O3 -mrtm -mrdrnd -o tsxBST tsxBST.cpp helper.cpp -lpthread
lab_jones@malbec:~/tsxBST_T2$ ./tsxBST
malbec Linux 4.9.0-6-amd64 64 bit exe RELEASE BST [NO lock single thread ONLY] NCPUS=16 RAM=126GB
METHOD=0 ALIGNED NOP=1000 NSECONDS=2 PREFILL=0 RECYCLENODES STATS=0x17 sizeof(Node)=24
Intel64 family 6 model 86 stepping 2 Intel(R) Xeon(R) CPU D-1540 @ 2.00GHz
L1 D 32K L 64 K 8 N 64
L1 I 32K L 64 K 8 N 64
L2 U 256K L 64 K 8 N 512
L3 U 12288K L 64 K 12 N 16384

  maxKey nt    pft    rt          ops          ops/s          nMalloc          nFree          ntree          v
  ----- --    ---    --          ---          -----          -
  16 1 0.00 2.00    44,724,000    22,350,824          17             8             9             86.
  64 1 0.00 2.00    40,567,000    20,273,363          54             24            30             86.
  256 1 0.00 2.00    33,656,000    16,819,590          167            34            133            86.
  1,024 1 0.00 2.00    27,200,000    13,593,203          584             74            510            87.
  4,096 1 0.00 2.00    16,237,000    8,114,442           2,194           160           2,034           87.
  16,384 1 0.01 2.00    10,805,000    5,399,800           8,441           285           8,156           88.
  65,536 1 0.02 2.00    7,797,000    3,896,551           33,160          507           32,653           94.
  262,144 1 0.07 2.00    5,524,000    2,760,619           131,770          774           130,996          118.
  1,048,576 1 0.18 2.00    3,498,000    1,748,125           524,786          1,456          523,330          214.
Press any key to continue...
```

The code in `tsxBST.cpp` is written to run on Windows in the first instance. If compiled on a linux machine, however, the window specific functions are mapped to linux equivalents by `helper.h` and `helper.cpp`.

The file `tsxBST.cpp` defines `METHOD`. If `METHOD` is set to 0, the test framework uses a single thread to randomly add and remove random keys from a BST for `NSECONDS` with `maxKey` ranging from 16 (keys 0 to 15) up to 1,048,576 (keys 0 to 1,048,575). The framework reports the number of operations per second, the average search depth and the max search depth amongst others (see figure above). As would be expected, the larger the key range the fewer operations can be performed per second.

If `METHOD` is set to 1, the same series of tests are performed with the number of threads ranging from 1 to  $2 * \text{NCPUS}$ . The BST is protected by a lock.

Code must to be added for METHOD 2 and 3. For METHOD 2, the testAndTestAndSet lock is replaced with an HLE testAndTestAndSet lock and for METHOD 3 the BST is updated using transactional memory (RTM).

**Rand function:** the standard VC++ rand() function only generates a 15 bit pseudo random number (0 .. 32767). For key ranges greater than 32768, this version of rand() is unsuitable. Alternative rand functions are provided in helper.cpp (for example UINT rand(UINT &r)). Note also that the VC++ rand() function is thread safe, but clearly the ones in helper.cpp are not. This means that the variable used to hold the *by address* parameter r must be thread local (for example declared in worker, not globally).

**Prefilling BST:** a large tree will take time to fill, consequently the ops/sec may be higher than expected because the most of the operations will occur on a smaller tree than that in steady state. It can take seconds for a large BST to become *full* and the ops/sec to reach a steady state. One way around this is to prefill the BST. Set PREFILL = 0 to fill a BST perfectly with odd integers, set PREFILL to 1 to prefill BST with a list of ascending odd integers (a linked list) and PREFILL = 2 for a list of descending odd integers (again a linked list). Large trees are filled using multiple threads.

**Transaction size:** the transaction size should be kept as small as possible. Larger read and write sets increase the probability of conflicts and hence transaction aborts. The code for adding and removing keys is iterative (rather than recursive) in an attempt to reduce the size of the read and write sets. It should also be apparent that it is unwise to call malloc and free (new and delete) within a transaction. Although is not possible to know exactly which memory locations these routine access, it is sensible to assume they read and write many memory locations and consequently, if include in a transaction, would increase the probability of transaction aborts. Consequently, calls to malloc and free (new and delete) should be kept outside of a transaction.

**Recycling nodes:** calls to malloc and free (to allocate nodes) from multiple threads may be a bottleneck because they normally cannot be executed concurrently (protected by a lock). Setting RECYCLENODES make uses of a per thread allocator (alloc and dealloc). Each thread maintains a list of free nodes (see PerThreadData free). When a thread allocates a node, if the per thread free list is not empty it takes a node for the head of the list otherwise it calls malloc. When a node is freed by a thread, it is placed on its per thread free list. This approach allows nodes to be allocated and freed concurrently most of the time.

**Per thread data:** a pointer to a thread's per thread data is stored at index tlsPtIndx in thread local storage (see the start of the worker function)

Hand in a short report describing what you have done, your results including graphs (max four A4 pages) and a code listing.

Consult CS4021/4521 web page for the project deadline.