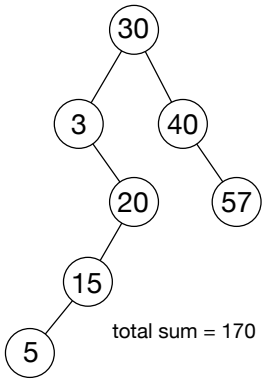
Stanford CS149: Parallel Computing
Written Assignment 5

Transactions on Trees

Problem 1:

Consider the binary search tree illustrated below.



The operations `insert` (insert value into tree, assuming no duplicates) and `sum` (return the sum of all elements in the tree) are implemented as transactional operations on the tree as shown below.

```
struct Node {
   Node *left, *right;
   int value;
};
Node* root;  // root of tree, assume non-null

void insertNode(Node* n, int value) {
  if (value < n->value) {
    if (n->left == NULL)
       n->left = createNode(value);
    else
       insertNode(n->left, value);
  } else if (value > n->value) {
    if (n->right == NULL)
       n->right = createNode(value);
    else
       insertNode(n->right, value);
  }  // insert won't be called with a duplicate element, so there's no else case
}

int sumNode(Node* n) {
   if (n == null) return 0;
   int total = n->value;
   total += sumNode(n->left);
   total += sumNode(n->right);
   return total;
}

void insert(int value) {  atomic { insertNode(root, value); }   }
int sum()              {  atomic { return sumNode(root); )      }
```

Consider the following four operations are executed against the tree in parallel by different threads.

```
insert(10);
insert(25);
insert(24);
int x = sum();
```

A. Consider different orderings of how these four operations could be evaluated. Please draw all possible trees that may result from execution of these four transactions. (Note: it's fine to draw only subtrees rooted at node 20 since that's the only part of the tree that's effected.)

B. Please list all possible values that may be returned by `sum()`.

C. Do your answers to parts A or B change depending on whether the implementation of transactions is optimistic or pessimistic? Why or why not?

D. Consider an implementation of **lazy, optimistic** transactional memory that manages transactions at the granularity of tree nodes (the read and writes sets are lists of nodes). Assume that the transaction `insert(10)` commits when `insert(24)` and `insert(25)` are currently at node 20, and `sum()` is at node 40. Which of the four transactions (if any) are aborted? **Please describe why.**

E. Assume that the transaction `insert(25)` commits when `insert(10)` is at node 15, `insert(24)` has already modified the tree but not yet committed , and `sum()` is at node 3. Which transactions (if any) are aborted? **Again, please describe why.**

F. Now consider a transactional implementation that is **pessimistic** with respect to writes (check for conflict on write) and **optimistic** with respect to reads. The implementation also employs a "writer wins" conflict management scheme – meaning that the transaction issuing a conflicting write will not be aborted (the other conflicting transaction will). Describe how a **livelock problem** could occur in this code.

G. Give one livelock avoidance technique that an implementation of a pessimistic transactional memory system might use. You only need to summarize a basic approach, but make sure your answer is clear enough to refer to how you'd schedule the *transactions*.

Problem 2:

Below is an implementation of an **optimistic read, pessimistic write, lazy versioning** STM that tracks reads and writes at the granularity of objects.

```
// Assume TMDesc is a transaction descriptor data structure
//    -- use GetDataVersion(obj) to access the version
// Assume each object maintains a lock, which supports these two ops:
//    -- LockObj(obj) returns true if success, false if failure
//    -- CheckLock(obj) returns true if locked, false otherwise
//    -- ReleaseLock(obj)

// helper routines ////////////////////////////////////////////////

// add object to read set
void OpenForReadTx(TMDesc tx, object obj) {
  tx.readSet.obj = obj;
  tx.readSet.version = GetDataVersion(obj);
  tx.readSet++;
}

// add object to write set
OpenForWriteTx(TMDexc tx, object obj) {
  if(!LockObj(obj) {      // try to lock object for writing
     AbortTx(tx);         // abort if someone else holds the lock
  }
  tx.writeSet.obj = obj;
  tx.writeSet.version = GetDataVersion(obj);
  tx.writeSet++;
}

// offset denotes what field of the object is being written to and needs to be buffered
// aka... conflict detection is at object granularity, but write logging at field granularity.
void writeBuffIntInsertTx(TMDesc tx, object obj, int offset, int value) {
  tx.writeBuff.obj = obj;
  tx.writeBuff.offset = offset;
  tx.writeBuff.value = value;  // buffer the value
  tx.writeBuff++;
}

  // helper: returns the int corresponding to the appropriate offset in object obj
int ReadDataFromMem(object obj, int offset);

// helper: writes value to the appropriate location in memory
void WriteDataToMem(object obj, int offset, int value);

void AbortTx(TMDesc tx); // call this internal helper to abort the transaction

// you will implement ReadIntTx in part A
int ReadIntTx(TMDesc tx, object obj, int offset) { }

// you will implement CommitTx in part A
bool CommitTx(TMDesc tx) { }

// you will implement UnlockObj in part A
void UnlockObj(object obj, TMVersion version) { }
```

A. Provide implementations for `ReadIntTx` (which reads an `int` value), `CommitTx()` (which commits transactions) and `UnlockObj()` (which commits writes) to ensure correct operation of this STM. Pseudocode is fine, but it must be sufficiently precise for the grader. **Hint: Be careful: how do you ensure your reads get the most up to date data?**

```
int ReadIntTx(TMDesc tx, object obj, int offset) {
  // transaction is performing a read to a field of obj (given by offset)









}

void UnlockObj(object obj, TMVersion version) {
  // this is the symmetric call to LockObj. Remember the lock is taken when
  // a pending transaction writes so unlock must be called when the writing
  // transaction commits.









}

bool CommitTx(TMDesc tx) {
   // recall *optimistic* reads, *pessimistic* writes
   // if there is a conflict, this transaction should abort









}
```

B. **THE REST OF THIS PROBLEM IS INDEPENDENT OF YOUR ANSWERS FROM PART A. NOTICE THAT THE TM SYSTEM IS NOW EAGER.** Assuming **optimistic reads, pessimistic writes and eager versioning**, fill in the tables below for the two concurrent transactions X1 and X2. **Assume all data and version numbers are initialized to zero.** Assume the transactions proceed concurrently, and the operations occur in order listed to the left of each statement. e.g., (1) is the first operation to occur in time, (2) is the second, etc. (5) and (7) are the commit times.

```
     X1                          X2
     ------------------------------------------
     atomic {                    atomic {
(1)    obj1.x = 5          (2)    t1 = obj1.x
(4)    obj2.x = 6          (3)    t2 = obj2.x
(5)  }                     (6)    obj3.x = t2 + 1;
                           (7)  }
```

After step (1) in the figure above, the state of the system looks like this: (The table below shows the metadata for all objects, as well as the read/write sets for all transactions, as well as the state of the transaction undo logs.

	X	version	Locked by
Obj1	5	0	X1
Obj2	0	0	-
Obj3	0	0	-

	Read Set	Write Set	Undo Log
X1	{}	{(obj1, 0)}	{(obj1.x, 0)}
X2	{}	{}	{}

Now please fill in the table to describe the state of the system after step (6) in the figure above:

	X	version	Locked by
Obj1			
Obj2			
Obj3			

	Read Set	Write Set	Undo Log
X1			
X2			

Hash Table Parallelization

PRACTICE PROBLEM 1:

A. Consider the following sequence of locking/unlocking operations by two threads.

```
        T0                              T1
==================              ==================

lock(l1);                         lock(l3);
lock(l2);                         lock(l2);
lock(l3);                         lock(l1);

// critical section               // critical section

unlock(l3);                       unlock(l1);
unlock(l2);                       unlock(l2);
unlock(l1);                       unlock(l3);
```

Assuming that both threads must acquire all three locks prior to entering the critical section, please describe the **correctness problem** that can occur when running these two threads. Please also describe a modification to the code that fixes the problem, while preserving mutual exclusion (protects the critical section).

B. Below you will find an implementation of a hash table (a linked list per bin). The hash table has a function called `tableInsert` that takes two strings, and inserts both strings into the table **only if neither string already exists in the table**. Please implement `tableInsert` below in a manner that enables maximum concurrency. You may add locks wherever you wish. (Update the structs as needed.) To keep things simple, your implementation SHOULD NOT attempt to achieve concurrency within an individual list (notice we didn't give you implementations for `findInList` and `insertInList`). **Careful, things are a little more complex than they seem. You should assume nothing about `hashFunction` other than it distributes strings uniformly across the 0 to `NUM_BINS` domain. (HINT: deadlock!)**

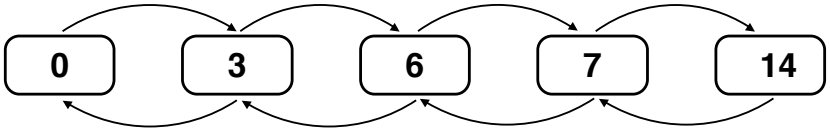
```
struct Node {
  string value;
  Node* next;

};

struct HashTable {
   Node* bins[NUM_BINS];  // each bin is a singly-linked list


};

int   hashFunction(string str);           // maps strings uniformly to [0-NUM_BINS]
bool  findInList(Node* n, string str);    // return true is str is in the list
void  insertInList(Node* n, string str);  // insert str into the list

bool tableInsert(HashTable* table, string s1, string s2) {
   int idx1 = hashFunction(s1);
   int idx2 = hashFunction(s2);
   bool result = false;




   if (!findInList(table->bins[idx1], s1) &&
       !findInList(table->bins[idx2], s2)) {



     insertToList(table->bins[idx1], s1);



     insertToList(table->bins[idx2], s2);



     result = true;
   }




   return result;
}
```

More on Concurrent Linked Lists

PRACTICE PROBLEM 2:

Consider a **SORTED doubly-linked list** that supports the following operations.

- `insert_head`, which traverses the list from the head. The implementation uses hand-over-hand locking just like in class.
- `delete_head`, which deletes a node by traversing from the head, using hand-over-hand locking just like in class.
- `insert_tail`, which traverses the list **backwards from the tail** to insert a node using hand-over-hand locking in the opposite order as `insert_head`.



A. Your friend writes three unit tests that each execute a pair of operations concurrently on the list shown above.

   - Test 1: `insert_head(2)`, `delete_head(14)`
   - Test 2: `insert_head(12)`, `delete_head(6)`
   - Test 3: `insert_head(13)`, `insert_tail(4)`

   The first two unit tests complete without error, but the third test goes badly and it does not terminate with the right answer. Describe what behavior is observed and why the problem occurs. (All unit tests start with the list in the state shown above.)

B. Imagine that locks in this system supported not only `lock()` and `unlock()`, but the ability to query the state of the lock via the call `trylock()` (this call takes the lock if the lock is free, but immediately returns false if the lock is currently locked – it does not block). Given this functionality, describe a fix to the problem you identified in part A? **Your answer should avoid livelock, but it is acceptable to allow for the possibility of starvation.**

Tricky Little Graphs

PRACTICE PROBLEM 3:

```
struct Graph_node {
   Lock   lock;
   float  value;
   int    num_edges;    // number of edges connecting to node (its degree)
   int*   neighbor_ids; // array of indices of adjacent nodes
};

// a graph is a list of nodes, just like in assignment 3
Graph_node graph[MAX_NODES];
```

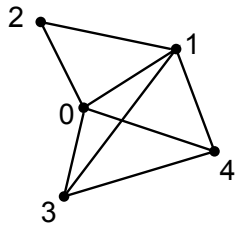
Consider the undirected graph representation shown in the code above.

Your boss asks you to write a program that atomically updates each graph node's `value` field by setting it to the average of all the values of neighboring nodes. The program must obtain a lock on the current node and all adjacent nodes to perform the update. It does so as follows...

```
void update(int id) {
    Graph_node* n = &graph[id];
    LOCK(n->lock);
    for (int i=0; i<n->num_edges; i++)
      LOCK(graph[n->neighbor_ids[i]].lock);
    // now perform computation...
```

Consider running the update code in parallel on nodes 0 and 1 in the two graphs below. For each graph, determine if deadlock occurs. Please describe why or why not. (Note: we do not ask you to solve the deadlock problem, but think about you might avoid it, assuming you must still only use locks. Consider changing the order in which you take the locks.

Graph A



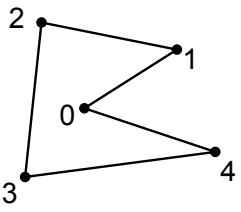
Values in neighbor_ids[]:

0: 3 4 2 1 3: 0 4 1

1: 0 2 3 4 4: 0 3

2: 0 1

Graph B



Values in neighbor_ids[]:

0: 1 4 3: 2 4

1: 0 2 4: 0 3

2: 1 3

A Concurrent Binary Search Tree

PRACTICE PROBLEM 4:

In this problem you'll work with a version of a binary search tree (BST where **locks are associated with the edges of the tree, rather than the nodes**. Edges are represented as a C++ class `Edge`, declared as follows:

```
class Edge {
 private:
    Node *n;           // Pointer to BST node reachable along edge (or NULL)
    Lock plock;        // Lock associated with arc
 public:
    Node *get();       // Retrieve node pointer
    void set(Node *n); // Set node pointer
    void lock();       // Acquire lock
    void unlock();     // Release lock
};
```

The node data structure has a per-node value, plus edges to its two children

```
class Node {           // Nodes in BST
  public:
    Edge left, right;  // Edges to subtrees
    int value;         // Node value

    Node(int v) {      // constructor
        value = v;
        left.set(NULL);
        right.set(NULL);
    }
};
```

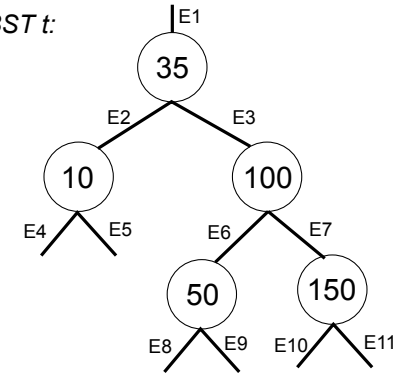
and the tree contains an "edge" to the root: (For an empty tree, the `n` field of the root edge is `NULL`.)

```
class BST {     // BST representation
 private:
    Edge root;
    public:
    // Insert value into BST
    bool insert(int val);

    // Remove maximum value node from BST, and assign its value to *val.
    // Return false if empty.
    bool remove_max(int *val);
};
```

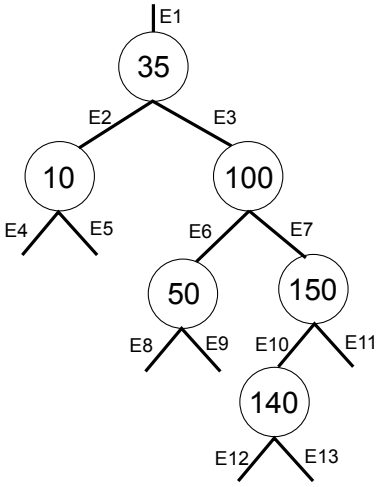
The following BST, which we will call `t` includes labels for all of its arcs. Notice that in a binary search tree, the left subtree of a node n contains nodes with values LESS than N, and the right subtree of a node n contains nodes with values GREATER than n.

As a reference, a correct insertion of the value 140 into t would yield the tree on the bottom-left. A correct removal of the maximum value would result in the tree on the bottom-right. We also show the result of removing the maximum element twice.

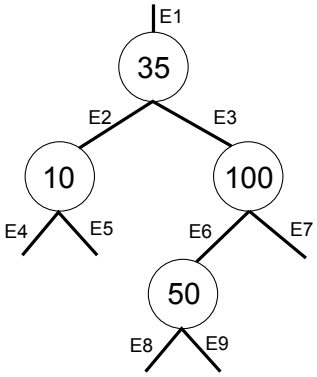
original BST t:



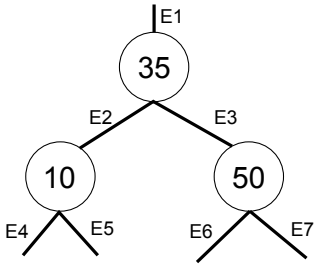
after inserting 140 to t:



after removing max from t:



after removing max from t, and then removing max again:



The following is a function for inserting elements into the tree. **It is intended to be thread safe (but may or may not be).**

```
// Top level insertion code (insert val into BST)
bool BST::insert(int val) {
    bool result = false;

    root.lock();
    result = insert_sync(&root, val);

    return result;
}

// insertion subroutine
bool BST::insert_sync(Edge *e, int val) {
    Node *n = e->get();
    if (n == NULL) {
        e->set(new Node(val));
        e->unlock();
        return true;
    }
    e->unlock();
    if (n->value == val) {
        return false;
    }
    Edge *next = (val < n->value) ? &n->left : &n->right;
    next->lock();
    return insert_sync(next, val);
}
```

The following is a function for removing the maximum element in the tree. (Notice it always traverses to the right child until there are no more right children.) **It is intended to be thread safe (but may or may not be).**

```
// Top level remove code. Returns true if a node exists, and fills in val
bool BST::remove_max(int* val) {
  root.lock();
  return remove_max_sync(root, *val);
}

// removal subroutine
bool BST::remove_max_sync(Edge *e, int *val) {
    Node *n = e->get();
    if (n == NULL) {
        e->unlock();
        return false;
    }
    Edge *next = &n->right;
    next->lock();

    bool found = remove_max_sync(next, val);

    if (!found) {
        // Current node holds the maximum value since there
        // is no right child

        *val = n->value;

        // Replace this node with its left subtree
        Edge *left = &n->left;
        left->lock();
        e->set(left->get());
        left->unlock();
        delete n;
    }
    e->unlock();
    return true;
}
```

A. For BST `t`, assume a thread executes the call `t.insert(40)`. What sequence of lock acquisitions and releases would it cause to occur? (Use the notation `L1` to indicate locking of edge E1, `U2` to indicate unlocking of edge E2, etc.)

B. For the original BST `t` (without any additional insertions), assume a thread executes the call `t.remove_max()`. What sequence of lock acquisitions and releases would occur?

C. Starting with BST `t`, suppose two threads execute the following:

Thread 1: `t.insert(140);`

Thread 2: `int v; t.remove_max(&v);`

Assume that Thread 1 acquires the lock on edge E1 first. Identify sequences of actions by the two threads that could cause the resulting tree to contain only four nodes, and then answer the following:

(a) Describe the specific locking, unlocking, and update operations:

(b) Draw (or describe in text) the resulting tree.

D. Starting with BST `t`, suppose two threads execute the following:

Thread 1: `int v; t.remove_max(&v);`

Thread 2: `t.insert(200);`

Assume Thread 1 acquires the lock on edge E1 first. List all possible value(s) that could be assigned to `v`. Explain why this is the complete set of possibilities.

E. Modify the insertion code below to eliminate the problem you identified earlier, **while still allowing fine-grained concurrency.**

```
bool BST::insert_sync(Edge *e, int val) {

  Node *n = e->get();

    if (n == NULL) {


        e->set(new Node(val));


        e->unlock();


        return true;
    }


    e->unlock();


    if (n->value == val) {


        return false;


    }


    Edge *next = val < n->value ? &n->left : &n->right;


    next->lock();


    return insert_sync(next, val);
}
```

PRACTICE PROBLEM 5:

A. Consider a cache coherence protocol that implements an **eager, pessimistic hardware transactional memory**. Each cache line can be one of three states: Invalid (I), Shared (S), or Exclusive (E). The protocol has the following rules:

- A cache miss occurs if the cache line is not present or is in the wrong state.
- Reads change the line in the cache to shared (S) state if it is (I) or not present.
- A line can be in the shared (S) state in multiple caches.
- Writes change the line in the cache to exclusive state
- Only one cache can have the line in exclusive state, in all other caches the line must be invalid (I) or not present.
- **On a cache miss data comes from memory... Unless another processor's cache has the line in exclusive state in which case data comes from that cache.**
- There are processor actions **Tbeg:** begin transaction, **Tend:** end and commit transaction. Remember that when a transaction commits, that might allow a stalled transaction to continue.
- Aborts cause the processor's cache's read and write cache state to be invalidated.
- **If a conflict is detected on a write, the transaction issuing the current action "wins" (abort the other conflicting transaction). If a conflict is detected on a read, the transaction issuing the read should stall waiting for the conflicting transaction to commit. (This is exactly as we discussed in the pessimistic example diagrams in class.**

Given the rules above, show what happens to the cache line state for address X for references made by three processors (P1, P2, P3) by filling in the table below (the first two rows are given). Initially none of the caches contain address X. **If an action causes a processor P to abort or stall a transaction, write "abort" or "stall" in the table entry at the row for that action and the column for P together with the state of P (e.g. "S, stall")**. If a transaction aborts, for simplicity assume that it never resumes for the rest of the duration of the table. (Also, if a transaction has already aborted, assume that the processor executing a **Tend** in a later row of the table does nothing.)

Processor Action	Hit / Miss	P1 state	P2 state	P3 state	Data comes from
P1,P2, P3 Tbeg		--	--	--	
P1 read x	miss	S	--		mem
P3 read x	miss	S	--	S	mem
P2 read x					
P1 Tend, Tbeg					
P2 Tend, Tbeg					
P1 read x					
P3 write x					
P2 read x					
P1 Tend					
P3 Tend					
P2 Tend					

B. Now imagine a cache coherence protocol that implements a **lazy, optimistic** hardware transactional memory system. Each cache line can be one of four states: Invalid (I), Shared (S), Shared Write (SW) or Exclusive (E). The protocol has the following rules:

- A cache miss occurs if the cache line is not present or is in the wrong state.
- Reads change the line in the cache to shared (S) state if it is I or not present.
- A line can be in shared (S) state in multiple caches.
- Writes change the line in the cache to shared write (SW) state; allowed in multiple caches. (Note the SW state is functioning as the write log.)
- SW and S can co-exist in different caches (convince yourself why this is true!)
- Only one cache can have the line in exclusive state, in all other caches the line must be invalid (I) or not present.
- On a cache miss data comes from memory... **unless another processor's cache has the line in exclusive state in which case data comes from that cache.**
- There are processor actions **Tbeg:** begin transaction, **Tend:** end and commit transaction
- Aborts cause read and write cache state to be invalidated
- Transaction commit (Tend) causes cache lines to transition from shared write (SW) to exclusive (E) state and may cause other transactions to abort.

Given the rules above, show what happens to the cache line state for address X for references made by three processors (P1, P2, P3) by filling in the table below. Initially none of the caches contain address X. If an action causes a processor P to abort or stall a transaction, write "abort" or "stall" in the table entry at the row for that action and the column for P together with the state of P (e.g. "S, stall"). If a transaction aborts, assume that Tend does nothing.

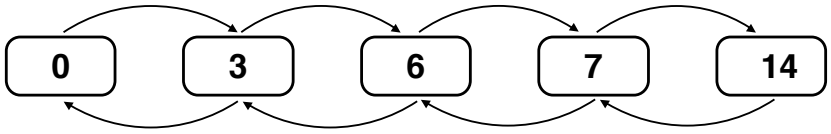
Processor Action	Hit / Miss	P1 state	P2 state	P3 state	Data comes from
P1,P2, P3 Tbeg		--	--	--	
P1 read x					
P3 read x					
P2 read x					
P1 Tend, Tbeg					
P2 Tend, Tbeg					
P1 read x					
P3 write x					
P2 read x					
P1 Tend					
P3 Tend					
P2 Tend					

Transactions on a Doubly Linked List

PRACTICE PROBLEM 6:

Consider a **SORTED doubly-linked list** that supports the following operations.

- `insert_front`, which traverses the list from the front.
- `delete_front`, which deletes a node by traversing from the front
- `insert_back`, which traverses the list **backwards from the end** to insert a node in the opposite order as `insert_front`.



In this problem, assume that the entire body of each function `insert_front`, `delete_front`, and `insert_back` is placed in its own atomic block, and the code is run on a system **supporting optimistic (for both reads and writes) transactional memory**.

A. Your friend writes three unit tests that each execute a pair of operations concurrently on the list shown above.

   - Test 1: `insert_front(2)`, `delete_front(14)`
   - Test 2: `insert_front(12)`, `delete_front(6)`
   - Test 3: `insert_front(13)`, `insert_back(4)`

   Assuming all unit tests start with the list in the state shown above, is the code correct? (By correct, we mean there are no race conditions and so all operations will modify the data structure according to their specification.) Why or why not?

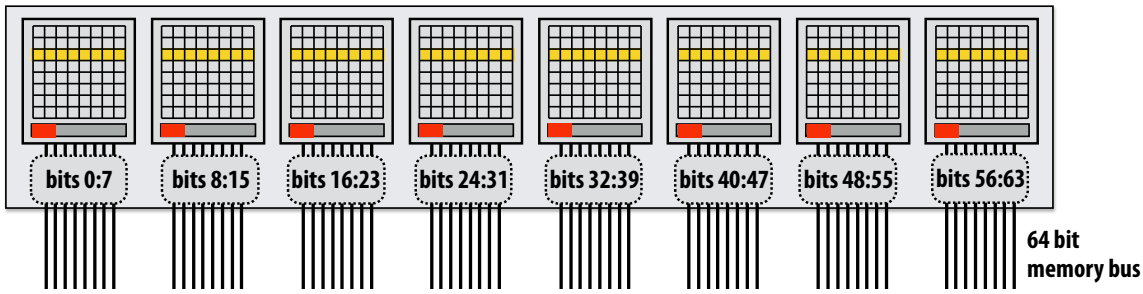
B. Consider two transactions performing `insert_front(4)` and `delete_front(14)`. Assume both transactions start at the same time on different cores and the transaction for `insert_front(4)` proceeds to commit while the `delete_front(14)` transaction has just iterated to the node with value 7. Must either of the two transactions abort in this situation? Why? **(Remember this is an optimistic transactional memory system!)**

C. Must either transaction abort if the transaction for `delete_front(14)` proceeds to commit before the transaction for `insert_front(4)` does? Why? **Please assume that at the time of the attempted commit, `insert_front(4)` has iterated to node 3, but has not begun to modify the list.**

D. Must either transaction abort if the situation in part C is changed so that `delete_front(14)` attempts to commit first, but by this time `insert_front(4)` has made updates to the list (although not yet initiated its commit)? Why?

PRACTICE PROBLEM 7:

Consider a DRAM DIMM with 8 chips (8-bit interface per chip) just like what we talked about in class. Physical memory addresses are strided across the chips as in the figure below, so that 64 consecutive bits from the address space can be read in a single clock over the bus. The DRAM row size is 2 kilobits (256 bytes). There is only a single bank per chip. (We ignore banking in this problem.)



The memory controller processes requests with the following logic:

```
int active_row;   // stores active row

handle_64bit_request(void* addr) {

   int row, col;

   compute_row_col(addr, &row, &col);   // compute row/col from addr (0 cycles)

   if (row != active_row)
      activate_row(row);     // this operation takes 15 cycles

   transfer_column(col);     // this operation takes 1 cycle
}
```

Questions are on the next page...

Now consider the following C-program, which executes using two threads on a dual-core processor with a single shared cache.

```
struct ThreadArg {
  int threadId;
  double sum;  // thread-local variable
  int N;       // assume this is very large
  double* A;   // pointer to shared array
};

// each thread processes one half of array A
void myfunc(ThreadArg* arg) {
  arg->sum = 0.f;
  int offset = arg->threadId * arg->N / 2;
  for (int i=0; i<arg->N / 2; i++)
    arg->sum += arg->A[offset + i];
}

/* main code */

ThreadArg args[2];
args[0].threadId = 0; args[1].threadId = 1;
args[0].A = args[1].A = new double[N];

// initialize args[].sum, args[].N, args[].A, and launch two threads here that run myfunc
// Then wait for threads to complete

print("%f\n", args[0].sum + args[1].sum);
```

A. Assume that the two threads run at approximately the same speed, so the memory controller receives requests from the two threads in interleaved order: thread0_req0, thread1_req0, thread0_req1, thread1_req1, etc. Given this stream, what is the effective bandwidth of the memory system as observed by the processor (the rate at which it receives data)? Assume that:

   - The program is bandwidth bound so that the memory system always has a deep queue of requests to process.
   - The granularity of transfer between the memory controller and the cache is 64 bits. (e.g., 8-byte cache line size)
   - Note that array elements are DOUBLES (8 bytes).

B. Modify the program code to significantly improve the effective memory system bandwidth. What is the new bandwidth you observe?

C. Return to the original code given in this assignment (ignore your solution to part B), and assume that requests now arrive at the memory controller every ten cycles. For example...

```
cycle 0: thread 0 req 0
cycle 10: thread 1 req 0
cycle 20: thread 0 req 1
cycle 30: thread 1 req 1
cycle 40: thread 0 req 2
cycle 50: thread 1 req 2
cycle 60: thread 0 req 3
...
```

Write (rough) pseudocode for a memory request scheduling algorithm that allows the memory system to keep up with this request stream. **Your implementation can assume there is an incoming request buffer called `request_buf` that holds up to 4 requests.** (The processor stalls if the request buffer is full.)

D. (TRICKY!) You add hardware multi-threading to your dual-core processor (2 threads-per core) and modify your code to spawn four threads. You assign contiguous blocks of the input array to each thread. Assuming the request arrival rate stays the same (but now requests from four threads, rather than two, are interleaved), how would you change your solution in part C to keep up with the request stream? (you may modify the buffer size if need be). Is overall memory latency higher or lower than in part C? Why?