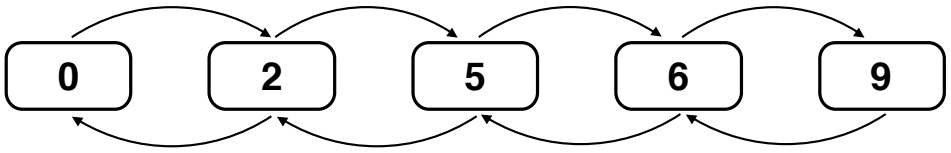
CMU 15-418/618: Parallel Computer Architecture and Programming
Practice Exercise 5

Problem 1: Concurrent Linked Lists (10 pts)

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

   - Test 1: `insert_head(1)`, `delete_head(9)`
   - Test 2: `insert_head(8)`, `delete_head(2)`
   - Test 3: `insert_head(8)`, `insert_tail(1)`

   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. (2 pts) 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? **Warning, your answer should avoid livelock, but it is acceptable in this problem to allow for the possibility of starvation.**

C. (1 pt) Imagine you removed all locks and implemented `insert_head`, `delete_head`, and `insert_tail` by placing the entire body of these functions in an atomic block for execution on a system **supporting optimistic transactional memory**. Does this fix the correctness problem described in part A? Why or why not?

D. (2 pts) Consider two transactions simultaneously performing `insert_head(3)` and `delete_head(9)`. Assume both transactions start at the same time on different cores and the transaction for `insert_head(3)` proceeds to commit while the `delete_head(9)` transaction has just iterated to the node with value 6. Must either of the two transactions abort in this situation? Why? **(Remember this is an optimistic transactional memory system!)**

E. (1 pt) Must either transaction abort if the transaction for `delete_head(9)` proceeds to commit before the transaction for `insert_head(3)` does? Why? **Please assume that at the time of the attempted commit, `insert_head(3)` has iterated to node 2, but has not begun to modify the list.**

F. (2 pts) Must either transaction abort if the situation in part E is changed so that `delete_head(9)` attempts to commit first, but by this time `insert_head(3)` has made updates to the list (although not yet initiated its commit)? Why?

Problem 2: Heap, Heap Hurray! A Concurrent Heap (10 pts)

Consider deletion from a max-heap data structure, which is implemented below. Please insert the appropriate locks to ensure race-free, concurrent deletion from the heap. You should assume deletion is the **only operation** performed on the heap, and may ignore edge cases like deletion from an empty or single element heap or the prealloced heap storage overflowing. You may add any additional locks if necessary, but `locks[i]` is meant to be the lock for node `A[i]` in the max-heap. **NOTE: for reference, the last page of this exercise contains an illustration of deletion from a max-heap.**

```
struct Heap {
   int   size;
   int   A[MAX_SIZE];
   Lock  locks[MAX_SIZE];


};

// The heap is a complete binary tree stored as an array
// Assume the root node of heap (maximum element) is at index 1 in the array, and the
// last valid node in heap is at index heap->size.
// Each call to this function pops the max element (root) from
// the heap, and the modifies the resulting structure to maintain
// the heap property: that every node is greater than its children.
int maxheap_delete(Heap* heap) {


   int top = heap->A[1];  // grab top of heap


   heap->A[1] = heap->A[heap->size];  // set last element to root position


   heap->size--;


   bubble_down(heap, 1);  // modify structure to regain heap property


   return top;            // return top
}
```

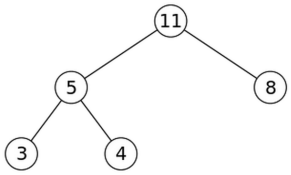
The function `bubble_down` is given on the next page. You may wish to modify it...

```
void bubble_down(Heap* heap, int cur) {
    int left = 2 * cur;          // compute index of left child
    int right = 2 * cur + 1;     // compute index of right child
    int largest = cur;           // which of the three nodes in question is the largest



    // determine if left child (if valid) is bigger than current node
    if (left <= heap->size && heap->A[left] > heap->A[largest]) {

       largest = left;

    }

    // determine if the right child (if valid) is bigger than the current or left node
    if (right <= heap->size && heap->A[right] > heap->A[largest]) {


       largest = right;


    }


    // if either left or right child is larger than current node,
    // then recursively bubble down
    if (largest != cur) {




       swap(heap->A[cur], heap->A[largest]);



       bubble_down(heap, largest);


    }


}
```

A maxheap is a **complete binary tree** where every node is greater than its two children. The figure below illustrates deletion from a binary heap (removing the greatest valued node, which resides at the root of the tree.)

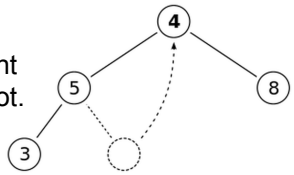
DELETION (OF THE MAX ELEMENT) FROM A MAX-HEAP:

Original heap state:



We remove the 11 and replace it with the 4.

Step 1: pop the root, place the last element in the heap at the root.



Step 2: restore heap property by swapping with childen (if necessary). In this case, swap 4 and 8. Then recurse on right child;