15-418/618 Spring 2017: Apr 7 Recitation – Practice Problems

A Concurrent Heap

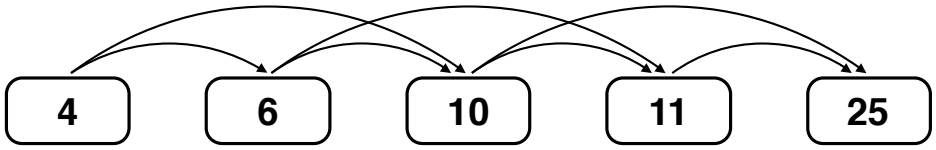
Consider deletion from a max-heap data structure, which is implemented below. Please insert the appropriate locks to ensure race-free, concurrent deletion from the heap. You should assume deletion is the **only operation** performed on the heap, and may ignore edge cases like deletion from an empty or 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);


    }


}
```

Another Fine-Grained Locking Question



Consider the semi-skip list structure pictured above. Each node maintains a pointer to the next node and the next-next node in the list. **The list must be kept in sorted order.** A node struct is given below.

```
struct Node {
   int value;
   Node* next;
   Node* skip;   // note that skip == next->next
};
```

Please describe a thread-safe implementation of **node deletion** from this data structure. You may assume that deletion is the only operation the data structure supports. Please write C-like pseudocode.

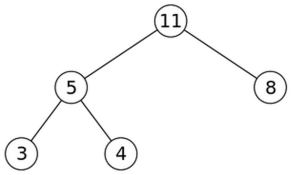
To keep things simple, you can ignore edge-cases near the front and end of the list (assume that you're not deleting the first two or last two nodes in the list, and the node to delete is in the list). If you define local variables like `curNode`, `prevNode`, etc. just state your assumptions about them. **However, please clearly state what per-node locks are held at the start of your process. E.g., "I start by holding locks on the first two nodes.".** Full credit will only be given for solutions that maximize concurrency.

```
// delete node containing value
void delete_node(Node* head, int value) {
```

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.)

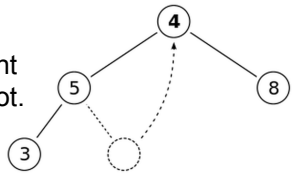
DELETION (OF THE MAX ELEMENT) FROM A MAX-HEAP:

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;