# Project 3: Database Management – Preston Hansen

## I/O Demonstrations:

*Note that screenshot(s) are raw output from Mars pasted into a text document for easier readability.*

```
Enter the max DB size (no greater than 256): 5

Enter account for insertion/lookup/delete: 1

Inserted account.
Enter account for insertion/lookup/delete: 2

Inserted account.
Enter account for insertion/lookup/delete: 3

Inserted account.
Enter account for insertion/lookup/delete: 4

Inserted account.
Enter account for insertion/lookup/delete: 1000000000

Inserted account.
Enter account for insertion/lookup/delete: 0

Accounts (min to max):
1,2,3,4,1000000000,
Most to least recently used:
1000000000,4,3,2,1,
```

First, the user initializes the database size. For simplicity, I chose 5 for this demonstration. After each user input, the desired action (insertion, lookup, deletion, display) is performed. The above screenshot demonstrates that account number and LRU sorting functions as expected.

```
Enter account for insertion/lookup/delete: 98765

Account deleted.
Inserted account.
Inserted and replaced LRU
Enter account for insertion/lookup/delete: 0

Accounts (min to max):
2,3,4,98765,1000000000,
Most to least recently used:
98765,1000000000,4,3,2,
Enter account for insertion/lookup/delete: 2

Found account (LRU modified)
Account deleted.
Inserted account.
Inserted account.
Enter account for insertion/lookup/delete: 0

Accounts (min to max):
2,3,4,98765,1000000000,
Most to least recently used:
2,98765,1000000000,4,3,
Enter account for insertion/lookup/delete: -89

Error: node not found to delete
Enter account for insertion/lookup/delete: -2

Account deleted.
Enter account for insertion/lookup/delete: -98765
```

Continuing the same demonstration, the DB is now full (5 accounts), so subsequent insertions will replace the LRU. The LRU (1) is replaced by 98765 and the LRU structure is modified. Next, account lookup is demonstrated. Since account 2 already exists, it is accessed and the LRU structure is again modified. Next, attempt to delete a non-present account is demonstrated. The program simply notifies the user of the account's (lack of) existence, and expects a new input.

```
Account deleted.
Enter account for insertion/lookup/delete: 0

Accounts (min to max):
3,4,1000000000,
Most to least recently used:
1000000000,4,3,
Enter account for insertion/lookup/delete: -4

Account deleted.
Enter account for insertion/lookup/delete: -3

Account deleted.
Enter account for insertion/lookup/delete: -1000000000

Account deleted.
Enter account for insertion/lookup/delete: 0

Error: empty list
Accounts (min to max):

Most to least recently used:
```

Finally, the remaining accounts are deleted to demonstrate deletion of present accounts. Attempting to display accounts warns the user of the empty list, and simply displays a blank line for account numbers and LRU order.

## Program Features:

i. My program achieves levels 1 and 2. All of the specified features (account insertion, deletion, lookup, display, LRU policy, user defined DB size) are implemented.

ii. The most significant challenge I found in the project was writing functions manipulating linked lists that would perform properly for all cases. To ensure proper functionality, I wrote driver functions that would test sample cases before attempting to implement the linked list functions in the final project.

iii. N/A

## Implementation Details:

i. 1:
I used a linked list to store the database in memory. I kept the accounts ordered from least to most recently used, where the LRU account was always guaranteed to be the head of the list. To sort the accounts, whenever the user requests display (enters zero), a sorting function copies the linked list values to a buffer which is then sorted by selection sort (and displayed to the user). For displaying the LRU in most to least recently used order, I simply copied the aforementioned buffer in reverse order before it is sorted to a second buffer. These two buffers contain the min to max accounts and most to least recently used accounts after any given display (zero) input by the user.

2:
Since the database is stored as a linked list, there is potential for inefficient routines – if nodes are stored in a fixed buffer, one might have to search the buffer every time a node is to be inserted. To optimize my program in light of this fact, I generated a stack of addresses which are empty (and can be used for node insertion) at the beginning of the program runtime. As nodes are created, addresses are popped off of the stack, and as they are deleted, they are pushed on to the stack. Contrary to normal linked list insertion, which could perform as poorly as O(n), insertion into my database is a constant time operation – it requires only that a stack pointer be maintained. For a given insertion, 6 store/load instructions are used. This number is fixed and does not change with database size. For each insertion, however, the list

must be traversed to find the node (or confirm that it is not present). In the worst case, the lookup is linear in the database size (O(n)). Each iteration of the traversal requires two loads, and initialization of the function requires two loads. O(n) extra space is required for the buffers and stack used to improve performance.

**A:**
In light of the above analysis, for a database of size 8, worst case insertion performance would require two load instructions for initialization, 8*2 = 16 load instructions to traverse the 8 accounts, and 6 load/store instructions to insert the account. Thus, the worst case cost of insertion is $2 + $16 + $6 = $24. In the best case (i.e., the list is empty, nothing needs to be traversed), the cost is initialization + insertion, so $2 + $6 = $8.

Because deletion also requires traversal of the list, there is a cost of $2 per iteration. Deletion cost depends on which node (the head, the tail, or something in between) is being deleted. On average, the cost of deletion is approx. $6. Thus, worst case deletion (whole list must be traversed) costs $2 (initialization) + $2*8 + $6 = $24. Best case deletion (the first node is the target) simply costs $2 for initialization and $6 for deletion, so $8 again.

Finding an existing account uses the same traversal as insertion, so the costs are similar, minus the insertion cost. Initialization is still $2, and each traversal is $2. Thus, worst case performance is $2 + 8*$2 = $18. Best case performance (the target is the head) is initialization + one traversal, so $2 + $2 = $4.

**B:**
As described above, my database performance scales linearly with size. Best case performances for all three operations would not change, but worst case would scale with database size.

Insertion would cost $2 + $2*1M + $6, roughly $2M in the worst case (and still $8 best case). Deletion would, by equivalent analysis, cost $2M in the worst case and $8 in the best case. Lookup would also cost $2M by the same logic as insertion and deletion in the worst case and $4 in the best case.

ii. To maintain the LRU replacement policy, I simply counted the length of the list as I traversed it. If the length was equal to the max DB size as defined by the user, the program branched to replace the LRU. Since, as described previously, the LRU was guaranteed to be the head of my linked list, I nested a call to the delete function within the LRU replacement function. This ensured that the head would be deleted (and its address pushed to the stack

of empty nodes), and that the new head (the new LRU) would, correctly, be updated to whatever account the head was previously pointing to. Finally, the insertion function is called again with the target and subsequently appended to the tail of the linked list (i.e. it is the most recently used).

Similarly, if the target account already existed in the linked list, I simply nested a call to delete the node, and then re-insert it. Since my delete function maintained the structure in the event of a node deletion, the accessed account would be appended to the tail of the list and the pointer structure would be correct. That is, the node that pointed to the deleted node would instead point to the node *after* the target node (again, the address of the deleted node was pushed to the node stack).

iii. N/A