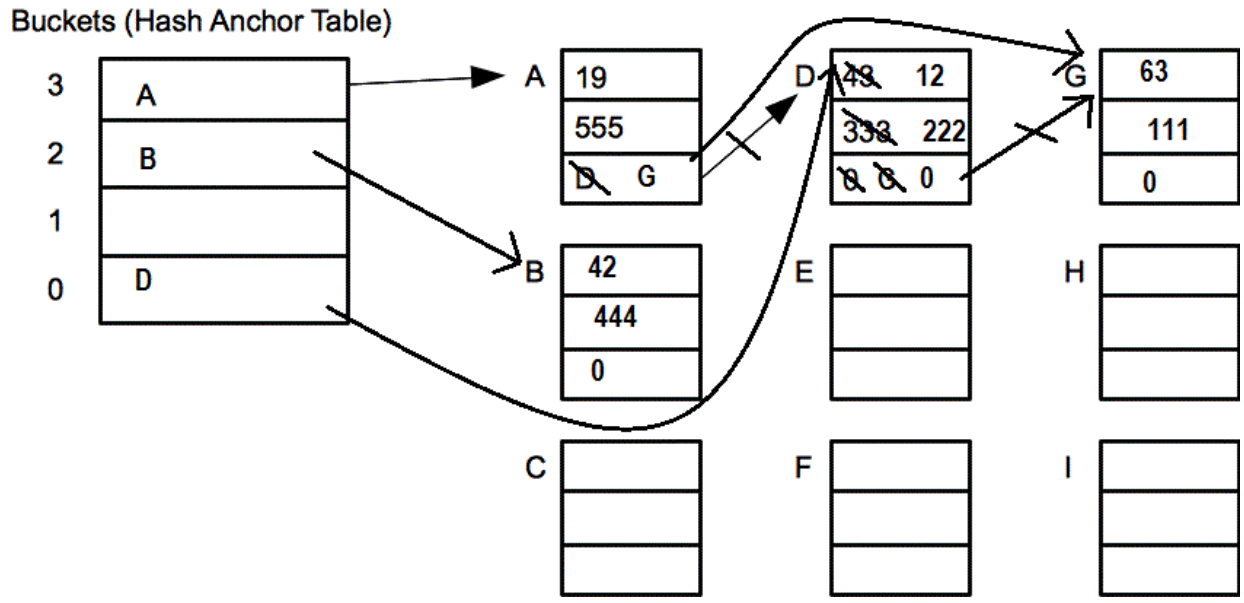
Problem 1 (2 parts, 32 points) **Hash Tables**

Part A (16 points) Consider an open hash table composed of a four-bucket table, with each bucket containing a variable length list. Each list entry has three slots <key, value, next> corresponding to the three word groupings in the entries section. The hash function is `key mod four`. Inserted entries are *appended to the end* of a bucket list. The initial state of the hash table is shown. List elements as allocated by malloc are shown in the figure. The symbol to the left of each list element (A, B, C,...) is the address returned by malloc. Entries that are freed are maintained in a last-in-first-out (LIFO) free list.

Execute the access trace shown in the table below. For ease of representation, you may use the allocated blocks in any order. Show pointers both by their (symbolic) value, and with an arrow. If a value changes, cross it out and place the new value to the right. If a pointer changes, also place an x on the corresponding arrow, and draw the new arrow.



Hash Table Access Trace

#	op	key	value	#	op	key	value
1	insert	63	111	3	insert	12	222
2	remove	43	n/a	4	insert	42	444

Part B (16 points) Consider a *different* hash table that uses **10 buckets**, each containing a singly linked list of entries. The hash table contains a total of **140 entries** evenly distributed across the hash table buckets. An application performs **1000** lookups of various keys: **800** of the lookups find the key in the hash table and **200** lookups fail to find the key. The keys that are found are distributed throughout the buckets so that each position is equally likely to be where a key is found. How many key comparisons would be required for the average lookup in the hash table if each bucket list is unsorted versus sorted?

L = 140/10 = 14

Hit rate (L+1)/2 + miss rate(L) = (8/10)(15/2) + (2/10)(14) = 6 + 2.8 = 8.8

number of comparisons when each bucket list is *unsorted:* **8.8**

number of comparisons when each bucket list is *sorted:* **(L+1)/2 = 15/2 = 7.5**

Problem 2 (2 parts, 36 points) **Doubly Linked Lists**

Consider a doubly linked list that is implemented using the following struct definitions.

NOTE: These are the same as the structs used in Project 2-1, except the `data` field in `llnode_t` is of type `int` and the `DLinkedList` has no `size` field.

```
typedef struct llnode_t {
    int              data;
    struct llnode_t* previous;
    struct llnode_t* next;
}LLNode;

typedef struct dll_t {
    struct llnode_t* head;
    struct llnode_t* tail;
    struct llnode_t* current;
} DLinkedList;
```

Part A (18 points) Suppose the C function `Find_Median` takes a pointer to a **sorted, nonempty** doubly linked list that has an **odd** number of nodes. Complete the function to find the middle node and return its `data` value. (Hint: take an equal number of steps from the head and tail until you hit the same node.)

```
int Find_Median(DLinkedList *DLL) {
   LLNode *H = DLL->head;
   LLNode *T = DLL->tail;
   while (H != T) {
     H = H->next;
     T = T->previous;
   }
   return(H->data);
}
```

Part B (18 points) Complete the C function `Insert_Node_Before` that takes a pointer to an `LLNode` and inserts it before the `current` node in the doubly linked list pointed to by the input parameter `DLL`. Return 0 if the current node is `NULL`, otherwise return 1 (this code is already provided). Be sure to update the head of `DLL` if `N` becomes the new head. DLL's `current` field should not change.

```
int Insert_Node_Before (LLNode *N, DLinkedList *DLL) {
  if(DLL->current == NULL){
    return 0;
  }else{
    LLNode *C = DLL->current;
    N->next = C;
    N->previous = C->previous;
    if (C == DLL->head)
       DLL->head = N;
    else
       (C->previous)->next = N;
    C->previous = N;
    return 1;
  }
}
```

Problem 3 (4 parts, 32 points) **Heap Management**

Consider a memory allocator (malloc and free), such as described in class. Inside the C-code for the allocator, `unsigned *heapPtr` is the address of the next word that could be allocated in the heap, and `unsigned **freePtr` is the address of the first block (or object) on the free list (and the word at the address of each block on the free list is a pointer to the next block on the free list). The allocator uses a **first fit** strategy with a free list **sorted by decreasing block size**, and never splits blocks.

addr	value	addr	value	addr	value	addr	value	addr	value	addr	value
8000	8	**8032**	20	**8064**	4	**8096**	8048	**8128**	8	**8160**	0
8004	8072	**8036**	0	**8068**	12	**8100**	8104	**8132**	8004	**8164**	0
8008	4	**8040**	43	**8072**	8088	**8104**	4	**8136**	4	**8168**	22
8012	16	**8044**	12	**8076**	8144	**8108**	2	**8140**	8	**8172**	7000
8016	8072	**8048**	8096	**8080**	8	**8112**	12	**8144**	43	**8176**	12
8020	8052	**8052**	12	**8084**	4	**8116**	0	**8148**	427	**8180**	41
8024	8132	**8056**	8	**8088**	0	**8120**	4	**8152**	8	**8184**	40
8028	8116	**8060**	8116	**8092**	16	**8124**	30	**8156**	0	**8188**	0

Suppose `heapPtr` = 8140 and `freePtr` = 8016. Consider each part below **independently**.

a) (5) How many **blocks and useable bytes** are on the free list? blocks = **3** bytes = **32**

b) (9) What value would be returned by the call **`malloc(10);`** **8016**

Which (if any) values in the above map would be changed by the call in (b)?

addr _____ value _____ addr_____ value_____ addr_____ value_____ No change (✓) ✓
(fill in the address/value pairs above. There may be more pairs than needed.)

Fill in the values at this point: `heapPtr` = **8140** `freePtr`= **8072**

c) (9) What value would be returned by the call **`malloc(18);`** **8144**

Which (if any) values in the above map would be changed by this call?

addr **8140** value **20** addr_____ value_____ addr_____ value_____ No change (✓) _____

Fill in the values at this point: `heapPtr` = **8164** `freePtr`= **8016**

d) (9) Which (if any) values in the above map would be changed by the call **`free(8004)`**?

addr **8072** value **8004** addr **8004** value **8088** addr_____ value_____ No change (✓) _____

Fill in the values at this point: `heapPtr` = **8140** `freePtr`= **8016**