Instructions: This is a closed book, closed note exam. Calculators are not permitted. If you have a question, raise your hand and I will come to you. Please work the exam in pencil and do not separate the pages of the exam. For maximum credit, show your work.
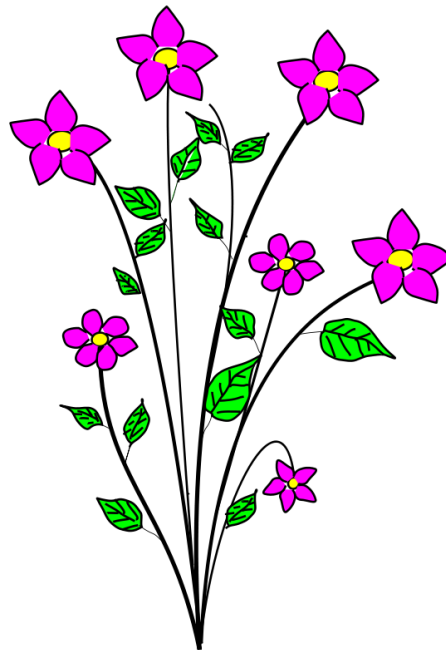*Good Luck!*

Your Name (***please print***) _____

This exam will be conducted according to the Georgia Tech Honor Code. I pledge to neither give nor receive unauthorized assistance on this exam and to abide by all provisions of the Honor Code.

Signed _____

| 1	2	3	
32	36	32	100

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.

Buckets (Hash Anchor Table)
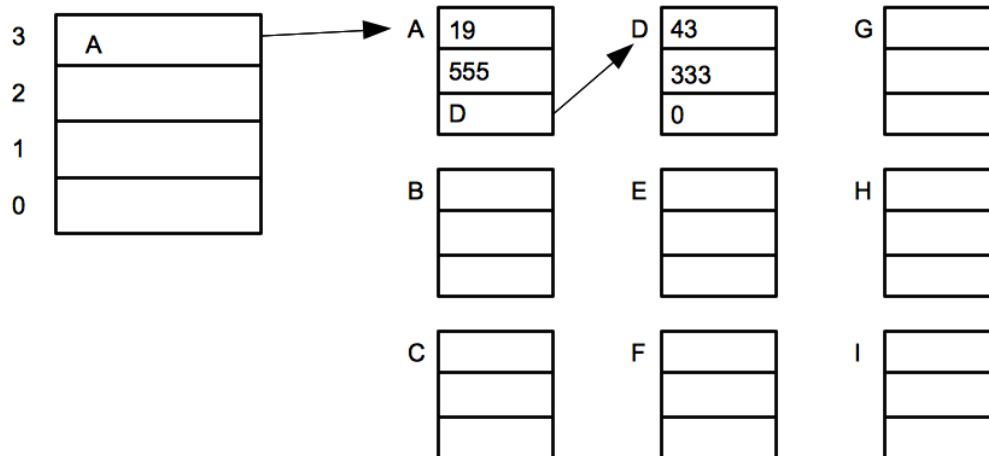
Hash Table Access Trace

#	op	key	value	#	op	key	value
1	insert	43	222	3	insert	17	111
2	remove	19	n/a	4	insert	21	444

Part B (16 points) Consider a *different* hash table that contains a total of **270 entries**, evenly distributed across the hash table buckets. The buckets each containing an **unsorted singly linked list** of entries. An application performs lookups of various keys: **3 out of 4** lookups find the key in the hash table. The keys that are found are distributed throughout the buckets so that each position is equally likely to be where a key is found. The number of key comparisons required for the average lookup in the hash table is **17.25**.

How many buckets are in this hash table?

_____

This hashtable is **resized** so that the average number of key comparisons per lookup is **6**. The resized hash table still has a total of 270 entries that are evenly distributed across the buckets, which are unsorted singly linked lists. The same application is run in which 3 out of 4 lookups find the key in the resized hash table.

How many buckets are in the resized hash table?   _____

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

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

```
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;
```

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.

Part A (12 points) The following C function **Insert_Node_After** is buggy. It is supposed to take a pointer to an `LLNode` and insert it after the `current` node in the doubly linked list pointed to by the input parameter `DLL`. It should return 0 if the current node is `NULL`, otherwise return 1. DLL's `current` field should not change. What is the bug and what is the bug fix (show code modifications to the right)?

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

// Show the bug fix here:

Description of the bug:

Part B (24 points) The C function `Concatenate` takes two pointers (`DLL1` and `DLL2`) to two doubly linked lists. Complete this function to append `DLL2` to the end of `DLL1`. When the function returns, `DLL1` should contain all of its original nodes followed by the nodes of `DLL2` (all nodes should be in the same order as in the original lists). `DLL1`'s `current` should point to the head of the resulting list. Be sure to handle the cases where `DLL1` and/or `DLL2` are empty. Also, free up the struct pointed to by `DLL2`.

```
 void Concatenate(DLinkedList *DLL1, DLinkedList *DLL2) {
```

```
 }
```

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 **best fit** strategy with an **unsorted** free list, 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	12	8172	7000
8016	8072	8048	8096	8080	8	8112	12	8144	8004	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` = 8156 and `freePtr` = 8144. Consider each part below **independently**.

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

b) (9) What value would be returned by the call **malloc(5);** _____

   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` = _____ `freePtr`= _____

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

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

   addr _____ value _____ addr_____ value_____ addr_____ value_____ No change (✓) _____

   Fill in the values at this point: `heapPtr` = _____ `freePtr`= _____

d) (9) Which (if any) values in the above map would be changed by the call **free(8016)**?

   addr _____ value _____ addr_____ value_____ addr_____ value_____ No change (✓) _____

   Fill in the values at this point: `heapPtr` = _____ `freePtr`= _____