Your Name (please PRINT clearly) _____

Your Student Number (9 digit) _____

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	total
30	30	40	100

Instructions: This is a closed book, closed note exam. Calculators are not permitted.

Please work the exam in dark pencil or pen and do not separate the pages of the exam. Note that this exam is double sided. If you run out of room, please mark on the problem that you will continue on the pages at the end of the exam. Also please identify the problem that is being continued, and draw a box around it. When writing your answer in a given answer box, please just write the answer and do not add extraneous marks etc. which confuse our grading tools.

Read each question over before you start to work.

If you have a question, raise your hand and we will come to you; do not leave your seat.

For maximum credit, show your work.

Good Luck!

Problem 1 (2 parts, 30 points) **Hash Tables and Linked Lists**

Consider a hash table that is implemented using the following `struct` definitions.

```
typedef unsigned int (*HashFunction)(unsigned int key);

typedef struct _HTE {
    unsigned int     key;
    void*            value;
    struct _HTE*     next;
} HashTableEntry;

typedef struct _HT {
    HashTableEntry** buckets;
    HashFunction     hash;
    unsigned int     num_buckets;
} HashTable;
```

Part A (12 points) Complete the following function, which takes a pointer to a `HashTableEntry`, which is the head of a linked list of entries, and computes the length of the list. Be sure to declare and initialize any local variables. **Write only C-code (no comments) in the given spaces.**

```
int Length(HashTableEntry* head) {
```

```

```

```
}
```

Part B (18 points) Complete the following function, which takes a pointer to a `HashTable` and computes the average bucket length (i.e, the average number of entries per bucket). It should call the **`Length`** function you wrote in Part A. **Write only C-code (no comments) in the given spaces.**

```
int AvgBucketLength(HashTable* myHT) {
  // Declare & initialize any local variables:
```

```

```

```
  // Loop through each bucket of myHT to count all entries:
```

```

```

```
  // Compute and return the average bucket length:
```

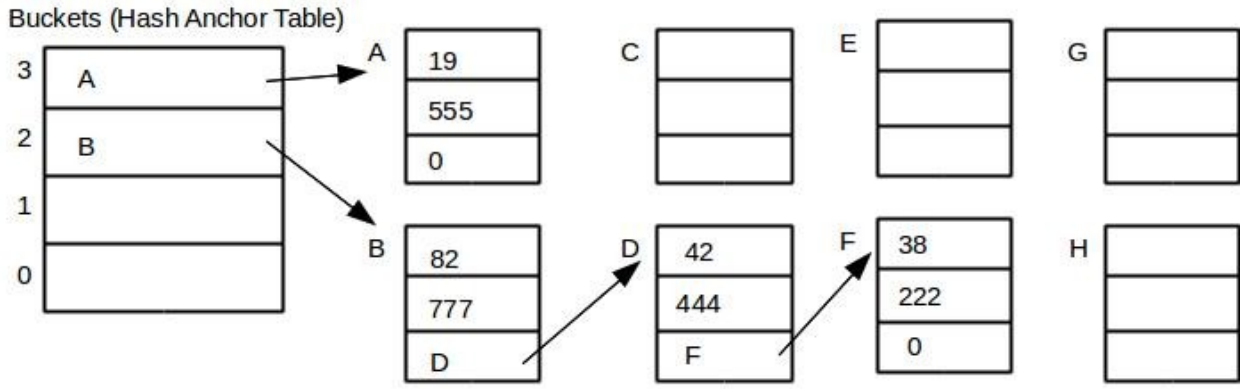
```

```

```
}
```

Problem 2 (2 parts, 30 points) **Hash Table Trace and Performance**

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 which starts out empty. *Be sure to reuse any blocks added to the free list before allocating unused blocks.*

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.



Trace #	op	key	value	Trace #	op	key	value
1	remove	82	n/a	3	insert	35	333
2	insert	47	111	4	insert	38	888

Part B (14 points) Consider an associative set of **700 entries** each containing a (key, value) pair. Suppose the set is implemented by a hash table with **20** buckets. An application performs **6000** lookups of various keys: **4800** of the lookups find the key in the list and **1200** lookups fail to find the key. The keys that are found are distributed throughout the list so that each position is equally likely to be where a key is found.

B.1 How many key comparisons would be required for the average lookup if the buckets are **sorted singly linked lists** of entries?

Answer: _____ key comparisons

B.2 How many key comparisons would be required for the average lookup if the buckets are **unsorted singly linked lists** of entries?

Answer: _____ key comparisons

Problem 3 (4 parts, 40 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 a free list **sorted by increasing block size**, and never splits blocks.

Incomplete code for malloc() is:

```
L1  void * malloc(unsigned size){
L2      unsigned **ptr = &freeptr;
L3      unsigned *block;
L4      while (*ptr) {
L5          if (                ) {    //Part A
L6              block = *ptr;
L7              *ptr = **ptr;
L8              return block;
            }
L9          ptr = *ptr;
        }
L10     block = heapptr + 1;
L11     *heapptr = size =                       ; //Part B
L12      heapptr =                              ; //Part C
L13     return block;
    }
```

Part A (5 pts) Complete the code for line L5 so the predicate is true when the given block is able to satisfy the request.

```
L5      if (                                    ) {
```

Part B (5 pts) Complete the code for line L11 to set the block size properly.

```
L11   *heapptr = size =
```

Part C (5 pts) Complete the code for line L12 to adjust the heapptr.

```
L12   heapptr =
```

Part D (25 pts) Suppose a snapshot of the heap is:

addr	value	addr	value	addr	value	addr	value	addr	value	addr	value
8000	16	**8032**	12	**8064**	4	**8096**	8048	**8128**	8	**8160**	0
8004	0	**8036**	0	**8068**	12	**8100**	8104	**8132**	8072	**8164**	0
8008	4	**8040**	8072	**8072**	8016	**8104**	4	**8136**	8	**8168**	22
8012	16	**8044**	12	**8076**	8144	**8108**	2	**8140**	4	**8172**	7000
8016	0	**8048**	16	**8080**	8	**8112**	12	**8144**	43	**8176**	12
8020	8	**8052**	0	**8084**	16	**8116**	8004	**8148**	28	**8180**	41
8024	8132	**8056**	8	**8088**	0	**8120**	4	**8152**	8	**8184**	40
8028	116	**8060**	8124	**8092**	16	**8124**	30	**8156**	0	**8188**	0

and `heapPtr` = 8148 and `freePtr` = 8132. Consider each part below **independently**.

Part D1 (4 pts) How many **blocks and useable bytes** are on the free list?

blocks		**bytes**	

Part D2 (6 pts) Consider the call: **`malloc(11);`** Give the values of `*ptr` and `**ptr` at line **L3**.

`*ptr`		`**ptr`	

Part D3 (9 pts) Now give the value of `ptr, *ptr` and `block` at the point where block is returned:

`ptr`		`*ptr`		`block`	

Part D4 (6 pts) Which (if any) values in the above map would be changed by the call **`free(8144)`**?

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

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