ECE 2035	Programming HW/SW Systems	Fall 2015
4 problems, 5 pages	Exam Three	18 November 2015

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.

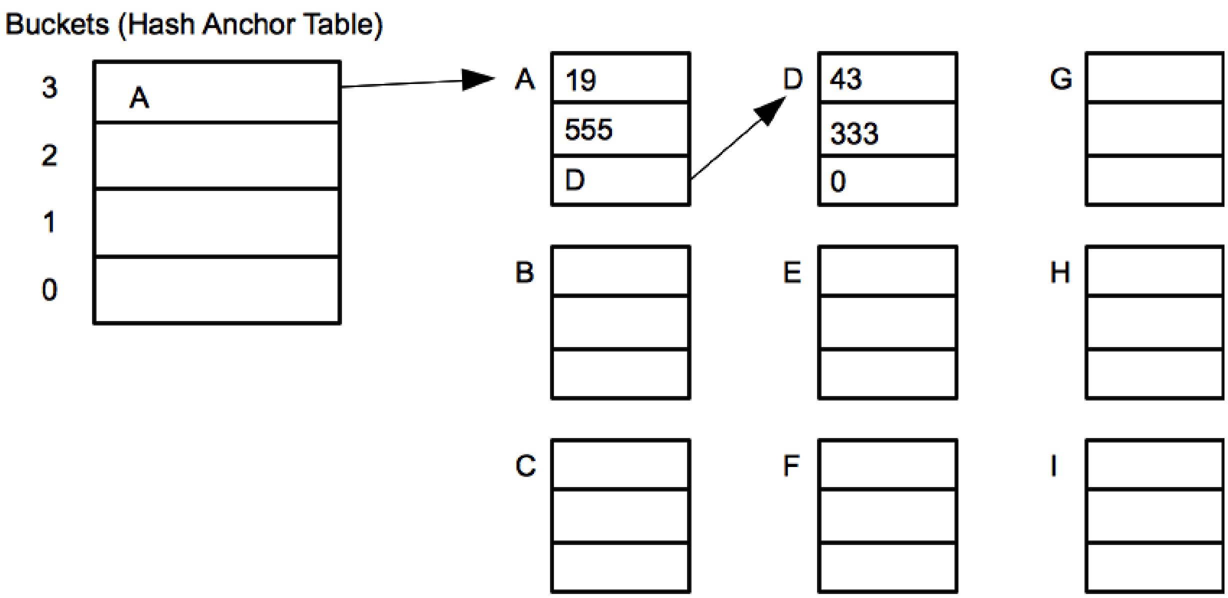
Signed ____

1	2	3	4		Total
20	20	30	30		100

Problem 1 (20 points) **Hash Tables**

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	100	111	5	remove	43	n/a
2	insert	43	222	6	insert	42	555
3	insert	35	333	7	remove	100	n/a
4	insert	0	444	8	insert	14	999

Problem 2 (3parts, 20 points) **Associative Set Performance**

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

```
#define NUMBUCKETS   11
typedef struct Entry {
   int           Key;
   int           Value;
   struct Entry  *Next;
} Entry;

typedef struct {
   Entry         *Buckets[NUMBUCKETS];
   int           Size;
} HashTable;
```

Suppose the entries are maintained in an *unsorted* linked in each bucket.

Part A (8 points) Complete the C function `Find_Key` that searches the hash table for an entry corresponding to a specified key. It should return a pointer to the matching `Entry` if `Key` is found or return `NULL` if `Key` is not found in the hash table.

```
Entry *Find_Key(HashTable *HT, int Key) {
   Entry    *ThisEntry;
   int      Index;
   int      Hash(int Key); /* function prototype for hash function */









}
```

Part B (8 points) Suppose a hash table created using the structs above contains **154** entries total and the entries are evenly distributed across the **11** hash table buckets, each implemented as an **unsorted** linked list of `Entry` structs. An application performs **450** lookups of various keys, some of which are found and some not. The keys that are found are distributed throughout the bucket lists so that each bucket and each position in the bucket lists is equally likely to be where a key is found. Suppose the average number of key comparisons that are needed for a lookup is **8.8**. In what percentage of the lookups is the key found? (Show work.)

Percentage of key lookups in which key is found: ______________________ **%**

Part C (4 points) Suppose the hash table (with the same **154** entries evenly distributed in **11** buckets) maintains a **sorted** link list of `Entry` structs in each bucket. What is the average number of key comparisons that would be needed for a lookup in this hash table implementation? (Show work.)

Number of key comparisons: ______________________

Problem 3 (4 parts, 30 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 **sorted** 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**	8016	**8104**	4	**8136**	4	**8168**	22
8012	16	**8044**	12	**8076**	8144	**8108**	2	**8140**	8	**8172**	7000
8016	8036	**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` = 8152 and `freePtr` = 8072. Consider each part below independently.

a) (4) How many **blocks and useable bytes** are on the free list? blocks = _____ bytes = ______

b) (9) What value would be returned by the call **`malloc(30);`** __________________

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(8);`**_____________________

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) (8) Which (if any) values in the above map would be changed by the call **`free(8116)`**?

addr _____ value _____ addr_____ value______ addr_____ value______ No change (✓) ______

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

Problem 4 (2 parts, 30 points) **Linked Lists**

Suppose we have the following definition which is used to create singly linked lists.

```
typedef struct Link {
   int          ID;
   int          Value;
   struct Link *Next;
} Link;
```

Part A (6 points) Complete the following subroutine which inserts a `Link` (pointed to by the input parameter `NewLink`) into the list just after the `Link` pointed to by the input parameter `Before`. You may assume that neither input parameter is `NULL`. `Before's Next` field may point to another `Link` or it may be `NULL`. `NewLink's Next` field is `NULL`.

```
void SpliceIn(Link *NewLink, Link *Before){
  ______________________________________________; /* part A*/
  ______________________________________________; /* part A*/
}
```

Part B Complete the following recursive subroutine which takes a pointer to the head of a linked list and returns a pointer to a copy of the linked list. Follow the steps specified below.

```
Link * CopyList(Link *Head) {
  if (Head == NULL) return ______________________ _; /* part B.1 */
  ______________________________________________; /* part B.2 */
  ______________________________________________; /* part B.3 */
  if (__________________________________________){  /* part B.4 */
     printf("Error: Insufficient space.");
     exit(1);
  }
  ______________________________________________; /* part B.5 */
  ______________________________________________; /* part B.5 */
  ______________________________________________; /* part B.6 */
  return LinkCopy;
}
```

Part B.1 (3 points) Fill in what should be returned if the list is empty.

Part B.2 (3 points) Add a local variable called `LinkCopy` that is a pointer to a `Link` object.

Part B.3 (5 points) Allocate space for a `Link` structure using `malloc` and make `LinkCopy` point to the object allocated. Be sure to include appropriate type casting to avoid type errors.

Part B.4 (3 points) Fill in the test for whether `malloc` found enough space which controls the print statement.

Part B.5 (5 points) Copy the values of `Head`'s `ID` and `Value` fields to `LinkCopy`.

Part B.6 (5 points) Call `CopyList` recursively to copy the rest of the list and assign the result to `LinkCopy's Next` field.