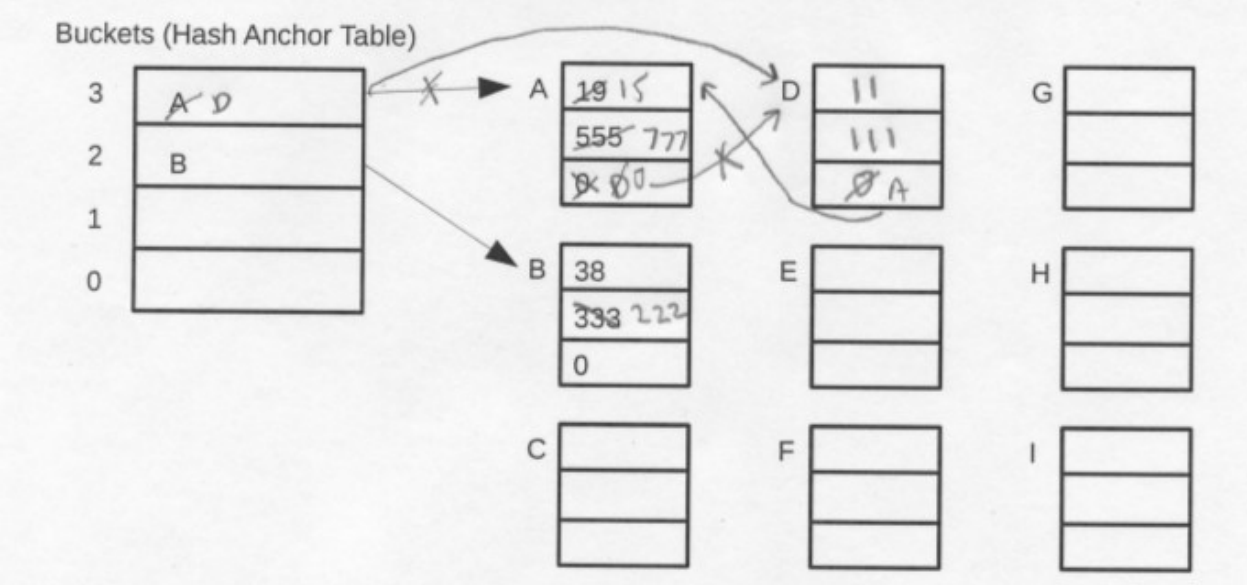
Problem 1 (2 parts, 22 points) **Hash Tables**

Part A (14 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. Assume the free list is initially empty.

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	11	111	3	remove	19	n/a
2	insert	38	222	4	insert	15	777

Part B (8 points)Consider a different hash table that uses **7 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: **600** of the lookups find the key in the hash table and **400** 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?

0.6(21/2) + (4/10)*20 = 6.3 + 8 = 14.3

number of comparisons when each bucket list is *unsorted:* **14.3**

number of comparisons when each bucket list is *sorted:* **21/2 = 10.5**

Problem 2 (4 parts, 28 points) **Dynamic Memory Allocation on Heap**

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 to the heap, and `unsigned **freePtr` is the address of the first block 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	1234	**8036**	0	**8068**	12	**8100**	8104	**8132**	8004	**8164**	0
8008	4	**8040**	43	**8072**	8036	**8104**	4	**8136**	4	**8168**	22
8012	16	**8044**	12	**8076**	8144	**8108**	2	**8140**	42	**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 = 8128`** and **`freePtr = 8016`**. *Consider each part below independently.*

Part A: (4 pts)	How many **blocks and useable bytes** are on the free list? blocks = **3** bytes = **48**
Part B: (12 pts)	B.1) What value would be returned by the call `malloc(10);` **8072** B.2) Which (if any) values in the above map would be changed by the call in B.1? addr **8016** value **8036** addr_____ value_____ addr_____ value_____ no change (fill in the address/value pairs above. There may be more pairs than needed) B.3) Fill in the values after the call in B.1: `heapPtr` = **8128** `freePtr` = **8016**
Part C: (8 pts)	C.1) What value would be returned by the call `malloc(22);` **8132** C.2) Which (if any) values in the above map would be changed by the call in C.1? addr **8128** value **24** addr_____ value_____ addr_____ value_____ no change
Part D: (4 pts)	Which (if any) values in the above map would be changed by the call `free(8116)`? addr **8116** value **8016** addr_____ value_____ addr_____ value_____ no change

Problem 3 (4 parts, 20 points) **Heap Management**

Consider the following three heap management strategies:

1. First fit with free list sorted by increasing size (smallest to largest).
2. First fit with free list sorted by decreasing size (largest to smallest).
3. Best fit with unsorted free list.

Part A (5 points) Which strategy (1, 2, or 3) has fastest average speed of `malloc`? _**2**_

Why?

With strategy 2, only the first element of the free list needs to be checked to see if it is large enough to accommodate the requested size. If it is not, then no other element on the free list will be large enough either, so space will simply be allocated at the top of the heap. If the first element is large enough, it will be used. The other two strategies involve searching through more of the free unless the first element is an exact size match.

Part B (5 points) Which strategy (1, 2, or 3) has slowest average speed of `malloc`? _**3**_

Why?

Strategy 3 needs to search the whole list to be sure it has the best fitting element unless it happens to find an exact size match.

Part C (5 points) Which strategy (1, 2, or 3) has fastest average speed of `free`? _**3**__

Why?

For an unsorted free list, the freed up object can just be pushed onto the front of the list. With a sorted list, the list must be searched for the proper place to insert the object and then needs to splice it in.

Part D (5 points) Which strategy (1, 2, or 3) has the worst internal fragmentation? _**2**___

Why?

Strategy 2 tends to select the largest sized block on the free list that is big enough, so there is a lot of wasted space (slack) in the allocated blocks.

Problem 4 (6 parts, 30 points) **Linked Lists**

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

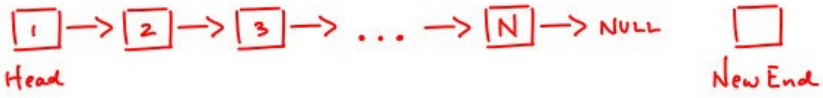
```
typedef struct Element {
   int             Num;
   struct Element *Next;
} Link;
```

Part A (4 points) Fill in the blank below to allocate space for a `Link` structure using `malloc` and make the variable `L1` point to the object allocated.

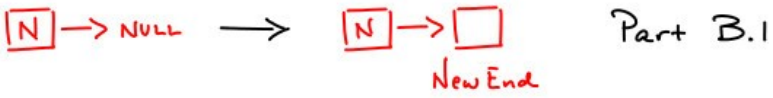
```
Link *L1 = ______(Link *) malloc (sizeof (Link))___;
```

Part B Complete the following recursive function which takes a pointer (called `Head`) to the first Link of a linked list and a pointer (called `NewEnd`) to a Link (which has a null Next field). The function places `NewEnd` on the linked list as its last element. Follow the steps specified below.

We would like to add NewEnd to the end of the list.

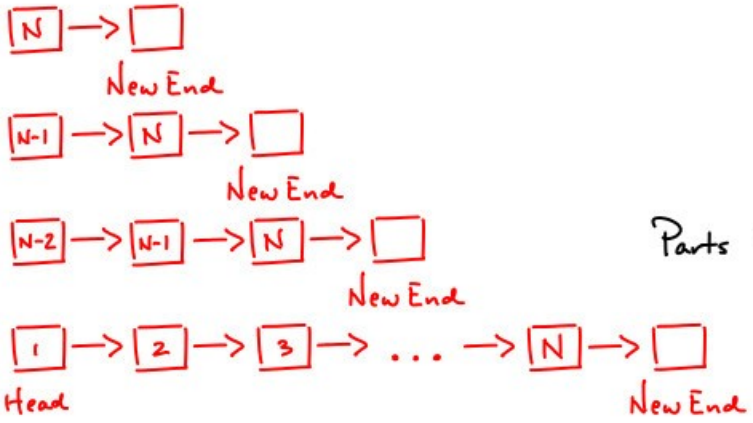


Without a tail pointer, we must traverse the list to append NewEnd. Utilizing recursion, we can traverse to the point where Next points to NULL. At that point, we will append NewEnd.



Part B.1

Because of how we have designed the recursive call, once the call stack begins unfurling, each return assigns Link's Next field to the successing Link.



Parts B.2-3

```
Link * Append(Link *Head, Link *NewEnd) {

  if (Head == NULL) return _NewEnd__________; /* part B.1 */

   Head->Next = Append(Head->Next, NewEnd)  ; /* part B.2 */

  return  Head                                ; /* part B.3 */
}
```

Part B.1 (5 points) Fill in what should be returned if the list is empty.

Part B.2 (8 points) Call `Append` recursively to place `NewEnd` on the end of the rest of the list and then push `Head` on the result of this recursive call.

Part B.3 (3 points) Fill in what should be returned.

Part C The following subroutine should free up all elements in the linked list whose first `Link` is pointed to by the input parameter `Head`. What error does it make? Write the correct code below.

```
void FreeElements(Link *Head)
{  Link *h;
   for (h = Head; h != NULL; h = h->Next)
      free(h);
}
```

Part C.1 (4 points) What is the error? **Dangling pointer: the program attempts to access h->Next after h has already been freed.**

Part C.2 (6 points) Write the correct version of FreeElements?

```
void FreeElements(Link *Head)
{  Link *h;
   Link *temp;
   for (h = Head; h != NULL; h = temp)
      temp = h->Next;
      free(h);
}
```