CS 241 | Lecture #9 Building an allocator

1. The following allocator will use this linked list structure:

```
typedef struct _metadata_entry_t {
   void *ptr;
   int size;
   int free; //0(in use) or 1(available)
   struct _metadata_entry_t *next;
} metadata_entry_t;
```

Global variable:

```
Static metadata_entry_t * head = NULL;
```

1. Complete `malloc()`

```
void *malloc(size_t size) {

   /* See if we have free space of enough size. */
   metadata_entry_t *p = head;
   metadata_entry_t *chosen = NULL;

   while (p != NULL) {
      if (p->free && ____________________________) {
         if (chosen == NULL || (chosen && p->size < chosen->size)) {
            chosen = p;
         }
      }
      p = p->next;
   }

   if (chosen) {
      chosen->free = 0;
      return chosen->ptr;
   }

  /* Add our entry to the metadata */
   chosen = sbrk(0);
   sbrk(sizeof(metadata_entry_t));
   chosen->ptr = sbrk(0);
   if (sbrk(size) == (void*)-1) {
      return NULL;
   }
   chosen->size = size;
   chosen->free = 0;

   chosen->next = head;
   head = chosen;
   return chosen->ptr;
}
```

2. Complete `free()`

```
void free(void *ptr) {
  if (!ptr) return;


    metadata_entry_t *p = ________________
    while (p) {
       if (p->ptr == ptr) {






       }
       p = p->next;
    }

    return;
}
```

2. Which placement algorithm does this malloc()use?

3. Does this implementation use explicit or implicit linked list?

Advantages?

Disadvantages?

4. Why does this implementation suffer from false fragmentation?

5. How would you change `malloc()` to use a *first-fit* placement allocation?

```
 while (p != NULL) {
      if (p->free && ____________________________) {
         if (chosen == NULL || (chosen && p->size < chosen->size)) {
            chosen = p;
         }
      }
      p = p->next;
   }
```

6. Towards a better allocator

Implementing `realloc` & improving performance of `free`()

Hint: Can we ensure this structure is immediately before the user's pointer?

```
typedef struct _metadata_entry_t {
   void *ptr;
   int size;
   int free;
   struct _metadata_entry_t *next;
} metadata_entry_t;
```

We want an O(1) deallocator!

```
void free(void*user) {
  if(user == NULL) return; // No-op
  ?
```

End of the allocator challenge?

1. Block Spitting & Block Coalescing
2. Memory pools
3. Advanced: Slab allocator and Buddy allocator
4. Internal vs External Fragmentation
5. How we use Boundary Tags to implement coalescing?

7. Puzzle:

Complete this code to read in values from stdin into heap memory. Can you beat CS225 code by using C and `realloc` to increase the size of the array? Fix any errors you notice.

```
#define quit(mesg) {puts(mesg); exit(1);}

size_t capacity = 256;
size_t count =  0;
int* data = malloc( capacity );
if( ! data ) quit("Out of memory");

while( !feof(stdin) && !ferror(stdin)) {
  if( count == capacity) {
     capacity *= 2;


  }
  if( fscanf(stdin, "%d", data+count) != 1) break;
  count++;
}
// can now reduce capacity to the number actually read
printf("%d values read",(int) count);
data = realloc(data, count);
```