CSE 351 Welcome back to section

Welcome back to section!

Section 9 Worksheet: Memory Allocation

To understand the computer, you must become the computer.

For this problem, you will use a heap simulator to answer questions about heap memory allocation in C using malloc and free. First, access the heap simulator, either by going to the Lab 5 page on the course website, or by following the link:

https://courses.cs.washington.edu/courses/cse351/heapsim/

Read through the information on the page so you can understand how the simulator works. If you check "simulation mode" it will walk you through each step in the process of allocating a block of memory on the heap. Notice that each header (and footer) contains three fields. They correspond to:

```
block size : preceding block allocated? : this block allocated?
```

where 1 means "allocated" and 0 means "unallocated." Remember that block size **includes** the size of the headers and footers. In Lab 5, all of this will be stored within 8 bytes using the last few bits of the size (see the spec for more information).

Now, answer the following questions:

1. Starting with an empty heap (you can empty the heap by refreshing the page), "Execute" the following code:

```
void *ptr1 = malloc(30);

void *ptr2 = malloc(40);

void *ptr3 = malloc(70);
```

   a. What pointer is returned if we execute another `malloc` now?

   b. Which block(s) could you free that would cause fragmentation in the heap?

   c. Which block(s) could you free that would cause coalescing to occur?

   d. Suppose `free(ptr2)` is run immediately after `malloc(70)`. Draw a diagram of what the free list looks like afterwards.

   e. What is the maximum size **payload** that we could allocate (i.e. the argument to `malloc`) such that we are returned a pointer to the address 48 (0x30)

In Lab 5, we will implement a memory management system that uses an explicit free list. Each block has pointers to the next and previous blocks. This is the block struct we will use:

```
struct BlockInfo {
  // Size of the block (in the high bits) and tags for whether the
  // block and its predecessor in memory are in use.
  size_t sizeAndTags;
  struct BlockInfo* next;
  struct BlockInfo* prev;
};
typedef struct BlockInfo BlockInfo;
```

Implement the following functions. Try using bitwise operators to access the tags in sizeAndTags.

```
// Bit masks used to retrieve tags from sizeAndTags.
#define TAG_USED 1
#define TAG_PRECEDING_USED 2
// SIZE(blockInfo->sizeAndTags) extracts the size of a 'sizeAndTags'
field.
#define SIZE(X) ((X) & ~(TAG_USED+TAG_PRECEDING_USED))
// Copies the tags (TAG_PRECEDING_USED and TAG_USED) from blockToCopy
// to originalBlock. Leaves the size of originalBlock unchanged.
void copyTags(BlockInfo* originalBlock, BlockInfo* blockToCopy) {

      size_t copyUsed = _____________________________;

      size_t copyPrecedingUsed = _____________________________;

      originalBlock->sizeAndTags = _____________________________;
}

BlockInfo *FREE_LIST_HEAD;
// Removes a block from the free list.
void removeFreeBlock(BlockInfo* freeBlock) {
      BlockInfo *nextFree, *prevFree;

      nextFree = freeBlock->next;
      prevFree = freeBlock->prev;

      // Your implementation goes here...









}
```