Memory Allocation Problems

Malloc implementation:

Consider an allocator that uses an implicit free list. Each memory block, either allocated or free, has a size that is a multiple of eight bytes. Thus, only the 29 higher order bits in the header and footer are needed to record block size, which includes the header and footer and is represented in units of bytes. The usage of the remaining 3 lower order bits is as follows:

- bit 0 indicates the use of the current block: 1 for allocated, 0 for free.
- bit 1 indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- bit 2 is unused and is always set to be 0.

Five helper routines are defined to facilitate the implementation of `free(void *p)`. The functionality of each routine is explained in the comment above the function definition. Fill in the missing code for the body of each helper routine.

HINT1: each solution can be done in a single line of C code, but you are allowed to use more than one line if you want

HINT2: use lots of parentheses "()" to clarify your order of operations

```
/* given a pointer p to an allocated block, i.e., p is a
pointer returned by some previous malloc()/realloc() call;
returns the pointer to the header of the block */
void * header(void* p)
{
  void *ptr;




  return ptr;
}

/* given a pointer to a valid block header or footer,   returns
the size of the block */
int size(void *hp)
{
  int result;




  return result;
}
```

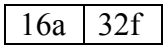
```
/* given a pointer p to an allocated block,i.e. p is a pointer
returned by some previous malloc()/realloc() call; returns the
pointer to the footer of the block*/
void * footer(void *p)
{
  void *ptr;









  return ptr;
}

/* given a pointer to a valid block header or footer,   returns
the usage of the currect block,
   1 for allocated, 0 for free */
int allocated(void *hp)
{
  int result;









  return result;
}

/* given a pointer to a valid block header,  returns the
pointer to the header of previous block in memory */
void * prev(void *hp)
{
  void *ptr;









  return ptr;
}
```

Dynamic Memory Allocation #1

Consider an allocator with the following:

- Uses a single implicit free list.
- All memory blocks have a size that is a multiple of 4 bytes and is at least 8 bytes.
- All headers, footers, and pointers are 4 bytes in size
- Allocated blocks consist of a header and a payload (no footer)
- Free blocks consist of a header and a footer at the end of the block.
- All freed blocks are immediately coalesced if possible.
- The heap starts with 0 bytes, never shrinks, and only grows large enough to satisfy memory requests.
- The heap contains only allocated blocks and free blocks. There is no space used for other data or special blocks to mark the beginning and end of the heap.
- When a block is split, the lower part of the block becomes the allocated part and the upper part becomes the new free block (and is coalesced if possible).
- The **best-fit** free block is used when searching the free list.
- If no free block is large enough then the heap is extended only enough to fulfill the request, including using any remaining free block at the end of the free list.

Below you are given a series of memory requests. You are asked to show what the heap looks like after each request is completed using the best-fit placement policy. The heap is represented as a series of boxes, where each box is a single block on the heap, and the bottom of the heap is the left most box. In each block, you should write the total size (including headers and footers) of the block in bytes and either 'f' or 'a' to mark it as free or allocated, respectively. For example, the following heap contains an allocated block of size 16, followed by a free block of size 32.



Assume that the heap is empty before each of the sequences is run, with a single 400B free block (shown for you). You do not necessarily have to use all of the columns provided for the heap. There are three copies here in case you make a mess. Clearly cross out the ones you don't want graded.

COPY #1: (they are identical, cross out the ones you don't want graded)

	400f						
ptr1 = malloc(8)							
ptr2 = malloc(31)							
free(ptr1)							
ptr3 = malloc(29)							
ptr4 = malloc(4)							
ptr5 = malloc(312)							
free(ptr4)							
ptr6 = malloc(9)							
ptr7 = malloc(4)							
free(ptr3)							
free(ptr2)							
free(ptr6)							

COPY #2: (they are identical, cross out the ones you don't want graded)

	400f						
ptr1 = malloc(8)							
ptr2 = malloc(31)							
free(ptr1)							
ptr3 = malloc(29)							
ptr4 = malloc(4)							
ptr5 = malloc(312)							
free(ptr4)							
ptr6 = malloc(9)							
ptr7 = malloc(4)							
free(ptr3)							
free(ptr2)							
free(ptr6)							

COPY #3: (they are identical, cross out the ones you don't want graded)

	400f						
ptr1 = malloc(8)							
ptr2 = malloc(31)							
free(ptr1)							
ptr3 = malloc(29)							
ptr4 = malloc(4)							
ptr5 = malloc(312)							
free(ptr4)							
ptr6 = malloc(9)							
ptr7 = malloc(4)							
free(ptr3)							
free(ptr2)							
free(ptr6)							

Dynamic Memory Allocation #2

Consider an allocator with the following specification:

- Uses a single explicit free list.
- All memory blocks have a size that is a multiple of 8 bytes and is at least 16 bytes.
- All headers, footers, and pointers are 4 bytes in size
- Allocated blocks consist of a header and a payload (no footer)
- Free blocks consist of a header, two pointers for the next and previous free blocks in the free list, and a footer at the end of the block.
- All freed blocks are immediately coalesced if possible, regardless of position in the free list.
- The heap starts with 0 bytes, never shrinks, and only grows large enough to satisfy memory requests.
- The heap contains only allocated blocks and free blocks. There is no space used for other data or special blocks to mark the beginning and end of the heap.
- When a block is split, the lower part of the block becomes the allocated part and the upper part becomes the new free block.
- Any newly created free block (whether it comes from a call to free, the upper part of a split block, or the coalescing of several free blocks) is inserted at the beginning of the free list.
- All searches for free blocks start at the head of the list and walk through the list in order (i.e., first-fit).
- If a request can be fulfilled by using a free block, that free block is used. Otherwise the heap is extended only enough to fulfill the request. If there is a free block at the end of the heap, this can be used along with the new heap space to fulfill the request.

Below you are given a series of memory requests. You are asked to show what the heap looks like after each request is completed using a first fit placement policy. The heap is represented as a series of boxes, where each box is a single block on the heap, and the bottom of the heap is the left most box. In each block, you should write the total size (including headers and footers) of the block in bytes and either 'f' or 'a' to mark it as free or allocated, respectively. For example, the following heap contains an allocated block of size 16, followed by a free block of size 32.

16a	32f

Assume that the heap is empty before each of the sequences is run, with a single 200B free block (shown for you). You do not necessarily have to use all the boxes provided for the heap. Some of the boxes are already filled in to help you. It is recommended to solve this on a scrap paper then copy your final answer into the boxes when you are satisfied. There are two copies here in case you make a mess. Clearly circle the one you want graded and cross out the one you don't want graded.

COPY #1: (they are identical, we will only grade the one you circle)

	200f						
ptr1 = malloc(1)							
ptr2 = malloc(12)							
ptr3 = malloc(17)							
ptr4 = malloc(36)							
free(ptr2)							
ptr5 = malloc(37)							
free(ptr4)							
ptr6 = malloc(20)							
free(ptr5)							
ptr7 = malloc(120)							
free(ptr3)							

COPY #2: (they are identical, we will only grade the one you circle)

	200f						
ptr1 = malloc(1)							
ptr2 = malloc(12)							
ptr3 = malloc(17)							
ptr4 = malloc(36)							
free(ptr2)							
ptr5 = malloc(37)							
free(ptr4)							
ptr6 = malloc(20)							
free(ptr5)							
ptr7 = malloc(120)							
free(ptr3)							