

Exercise 3

EE312 Fall 2014

TWO POINTS

NOTE: I've left all the type casts off for the code examples/questions below. We do not consider type casts to be materially significant to your understanding of pointers. Pointer types **are** important, *casting* one type of pointer to another is (mostly) style and not substance.

1. Write a function that searches the heap for the closest fitting available chunk for the specified size. Specifically return the index within the heap of the bottom signature for the smallest available chunk that has at least *size* words (a word is 4 bytes, and is the amount of memory required to hold an **int** variable). You can assume that the heap is maintained in a global variable (an array of ints) called *heap*. The number of elements in this array is *heap_size*. Do not change the heap in any way, simply identify and return the best fitting chunk currently in the heap. You should assume that there is at least one chunk large enough to satisfy the request.

```
const int heap_size = ...; // some value
int heap[heap_size];

// return the index (not the address) of the bottom signature of the chunk
int bestFit(int size) {
```

2. For each program, draw a diagram of the heap and answer the question(s) posed. You should assume that the heap is an array of 10 **int** locations beginning at address 1000. The following diagram shows the initial state of the heap. You should assume that each of the following questions starts from this state. The “?” indicates that the value stored in that memory location is unknown. Please show only the final state of the heap. If you need scratch paper, please write on the back of the page **WE NEED TO BE ABLE TO READ YOUR DIAGRAM**. No credit will be given for illegible answers. Be sure to both draw the diagram of the heap and answer the question. You must include both an answer and a diagram to receive credit. REMEMBER the argument to malloc is in units of BYTES

the heap	
1036	8
1034	?
1028	?
1024	?
1020	?
1016	?
1012	?
1008	?
1004	?
1000	8
address	contents

a. (6 pts)

```
int* p = malloc(1);
int k;
for (k = 0; k < 5; k = k + 1) {
    p[k] = -1;
}
for (k = 5; k < 9; k = k + 1) {
    p[k] = 2;
}
int* q = malloc(1);
```

What address is stored in the variable q?

the heap	
1000	
address	contents

b. (6 pts)

```
int* p = malloc(1);  
*p = 42;  
int* q = malloc(1);  
free(p);  
int* r = malloc(1);  
*q = *r;
```

How large (in bytes) is the largest chunk
still available?

1000
address

the heap

contents

c. (4 pts) (arg to malloc is bytes!)

```
int* p = malloc(6);  
int* q = malloc(4);  
int* r = malloc(4);  
free(r);  
free(q);  
free(p);
```

1000
address

the heap

contents

Beta/Challenge Problem

We use “first fit” as our allocation strategy for our heap. This is fine, but it’s possible that we’ll look at a lot of busy chunks before we find any free ones. I’d like you to contemplate (and design) an optimization to our heap where we keep all of the available blocks on the heap in a linked list. Then, rather than doing traditional “first fit” looking at every block in the heap, we simply search the linked list to find the first block on the free list that’s big enough. Contemplate and give answers to the following questions (discussion in recitation is an excellent idea).

So, in summary, we will keep a linked list of all the available blocks on the heap. Blocks that are in-use will not be in the linked list (it’s the client’s job to track them).

1. Ignoring (for now) the issues associated with splitting large blocks into smaller blocks when allocating, and combining smaller blocks into larger blocks when deallocating – what is the basic mechanic of the linked list
 - a. How would the list be initialized?
 - b. How would malloc work with the linked list, could malloc work if the list was singly linked, or does the list need to be doubly linked? Why?
 - c. How would free work with the linked list? When a block is added to the linked list, where should the block be placed on the list – at the beginning, at the end, in the middle?
2. How does the introduction of this linked list impact the meta-data that a Knuth heap would use? Are there additional meta-data that must be stored? How much larger would a “chunk” be (e.g., with a traditional Knuth heap, an 8-byte block of usable memory requires a 16-byte chunk in the heap – 100% overhead. How much worse does this problem get if we add the linked list idea?). Is there any Knuth meta-data that we no longer need? Can we reduce some of the traditional overhead by using this linked list?
3. If, during malloc, we find a block that we want to use on the linked list, we remove the block from the list and then decide we can split the block, how do we return the left-over part of the split block to the linked list?
4. If, during free, we find that that block we’re deallocating can be merged with its predecessor, what (if anything) do we do? Do we remove the predecessor from the linked list? Do we add the new block to the linked list?
5. Same as question 4, but what if the block can be merged with its successor?