Sorted singly linked list

You are going to write a program that implements a sorted singly linked list that supports insertion and deletion. The two most important aspects of this project are:

1. Using C and C++ structs in assembly language.
2. Perfecting memory management discipline (hint: your program will be subjected to valgrind on Linux and Leaks on the Mac).

You know how to do this. You've done it before if you have taken Data Structures and Algorithms. In fact, you might consider implementing this program in C *first*.

All behavior and output should closely follow this specification

Do not vary the messages. Do not be creative with your wording. You should try to match my output letter-for-letter. In fact, here is my text for you to use:

```
head_address:   .asciz  "head points to: %x\n"
node_info:      .asciz  "node at 0x%8x contains payload: %lu next: 0x%8x\n"
bad_malloc:     .asciz  "malloc() failed\n"
```

IMPORTANT

The following is deprecated. Ignore it.

~~Modern Linux systems have Address Space Randomization enabled by default. This is a security mechanism whereby the layout in memory of an application is randomized. Without the following step, your output will NOT match mine because the addresses returned by your `malloc()` will not match mine due to randomization.~~

The following is deprecated. Ignore it.

```
$ su
<<enter the root password - it is 'a' without the quotes>>
# echo 0 > /proc/sys/kernel/randomize_va_space
# exit
```

The following is deprecated. Ignore it.

~~This must be done just once for the lifetime of the course. If you want to reenable Address Space Randomization, change the 0 to 2 in the above steps.~~

The node

```
struct Node
{
```

```
    struct Node * next;
    unsigned int payload;
};
```

Notice the payload is an `unsigned int`.

No negative number will ever make it into the linked list.

Encountering a negative number amongst the command line arguments triggers deletion. Positive numbers cause insertion. See below.

You are strongly encouraged to test any assumptions about how the above struct is layed out in memory. It would really suck to be dead in the water before you even get in the actual water.

Think about what a pointer to a `Node` actually points to. Think of the pointer as being the base address of the struct. All of a struct's members are offsets from its base address.

The command line

Any positive number provided on the command line is an insertion. A node with that number in the node's payload member is inserted in sorted order.

Any negative number provided on the command line is a deletion of the node with that number's **absolute value**.

If there are multiple nodes with the same payload, only the first one found is deleted.

If a node with the negative number's **absolute value** cannot be found, simply ignore it.

```
user@comporg:~/p2$ ./a.out 40 30 10 20 -30
head points to: aaabc2a0
node at aaabc2a0 contains payload: 10 next: aaabc2c0
node at aaabc2c0 contains payload: 20 next: aaabc260
node at aaabc260 contains payload: 40 next:        0
user@comporg:~/p2$
```

This will have added nodes for 40, 30, 20 and 10 but in sorted order. Then, it will have deleted the node for 30.

Follow the output - it says the head is at ...2a0.

The node at ...2a0 contains 10 and points to ...2c0.

The node at ...2c0 contains 20 and points to ...260.

The node at ...260 contains 40 and is the last node.

Notice the address for the node containing 40 is also the smallest of the node addresses. This makes sense since it was the first node inserted.

`valgrind` for Linux

`valgrind` must produce NO errors. This listing shows another sample output showing that **All heap blocks were freed – no leaks are possible**.

```
user@comporg:~/p2$ valgrind ./a.out 40 30 10 20 -30 -20 -10 -40
==767== Memcheck, a memory error detector
==767== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==767== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==767== Command: ./a.out 40 30 10 20 -30 -20 -10 -40
==767==
head points to: 0
==767==
==767== HEAP SUMMARY:
==767==     in use at exit: 0 bytes in 0 blocks
==767==   total heap usage: 5 allocs, 5 frees, 1,088 bytes allocated
==767==
==767== All heap blocks were freed -- no leaks are possible
==767==
==767== For counts of detected and suppressed errors, rerun with: -v
==767== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
user@comporg:~/p2$
```

I believe I already installed `valgrind` in the VM for you. If I did not you can install it yourself.

```
sudo apt install valgrind
```

`leaks` for Macintosh

If you are programming directly on the Apple Silicon processor, `valgrind` will not be accessible. Instead use `leaks`.

```
leaks --atExit -- ./a.out
```

A lot of stuff is printed with leak information appearing at the end. The very last line should look like:

```
Process _____: 0 leaks for 0 total leaked bytes.
```

where a process number replaces the underscores.

`printf` is hard on Apple Silicon

Variadic functions like `printf` are handled quite differently by Mac OS compared to Linux. You will be printing some fancy things:

```
"node at 0x%8x contains payload: %lu next: 0x%8x\n"
```

Notice there are three slots for data. Once you have correctly set this up as if it were Linux (i.e. all in registers), then add the code to shift the data onto the

stack from right to left.

This can be hard to get right so I will provide a working example:

```
#if defined(__APPLE__)
        PUSH_R      x3
        PUSH_P      x1, x2
        CRT         printf
        add         sp, sp, 32
#else
        bl          printf
#endif
```

Previous to this code, I set things up the way Linux would expect. Then from right to left go `x3` along with a non-existent partner, then `x1` and `x2` in that order.

Work rules

All work is to be done solo. This is not a trivial assignment so efforts at plagiarism may be obvious.

What to hand in

Into Schoology, submit only your assembly language source code file. Make sure your name is at its top.

Suggested tests and expected output

```
user@comporg:~/p2$ # null test
user@comporg:~/p2$ ./a.out
head points to: 0

user@comporg:~/p2$ # single insertion
user@comporg:~/p2$ ./a.out 10
head points to: aaabc260
node at aaabc260 contains payload: 10 next:        0

user@comporg:~/p2$ # single insertion then deletion
user@comporg:~/p2$ ./a.out 10 -10
head points to: 0

user@comporg:~/p2$ # two insertions already in order
user@comporg:~/p2$ ./a.out 10 20
head points to: aaabc260
node at aaabc260 contains payload: 10 next: aaabc280
node at aaabc280 contains payload: 20 next:        0

user@comporg:~/p2$ # two insertions not in order
user@comporg:~/p2$ ./a.out 20 10
```

```
head points to: aaabc280
node at aaabc280 contains payload: 10 next: aaabc260
node at aaabc260 contains payload: 20 next:        0

user@comporg:~/p2$ # three insertions each updating head
user@comporg:~/p2$ ./a.out 30 20 10
head points to: aaabc2a0
node at aaabc2a0 contains payload: 10 next: aaabc280
node at aaabc280 contains payload: 20 next: aaabc260
node at aaabc260 contains payload: 30 next:        0

user@comporg:~/p2$ # inserting in the middle
user@comporg:~/p2$ ./a.out 10 30 20
head points to: aaabc260
node at aaabc260 contains payload: 10 next: aaabc2a0
node at aaabc2a0 contains payload: 20 next: aaabc280
node at aaabc280 contains payload: 30 next:        0

user@comporg:~/p2$ # deleting the head
user@comporg:~/p2$ ./a.out 10 30 20 -10
head points to: aaabc2a0
node at aaabc2a0 contains payload: 20 next: aaabc280
node at aaabc280 contains payload: 30 next:        0

user@comporg:~/p2$  # deleting from the middle
user@comporg:~/p2$ ./a.out 10 30 20 -20
head points to: aaabc260
node at aaabc260 contains payload: 10 next: aaabc280
node at aaabc280 contains payload: 30 next:        0

user@comporg:~/p2$ # deleting at the tail
user@comporg:~/p2$ ./a.out 10 30 20 -30
head points to: aaabc260
node at aaabc260 contains payload: 10 next: aaabc2a0
node at aaabc2a0 contains payload: 20 next:        0
user@comporg:~/p2$
```

You should also `valgrind` or `leaked` each of these.