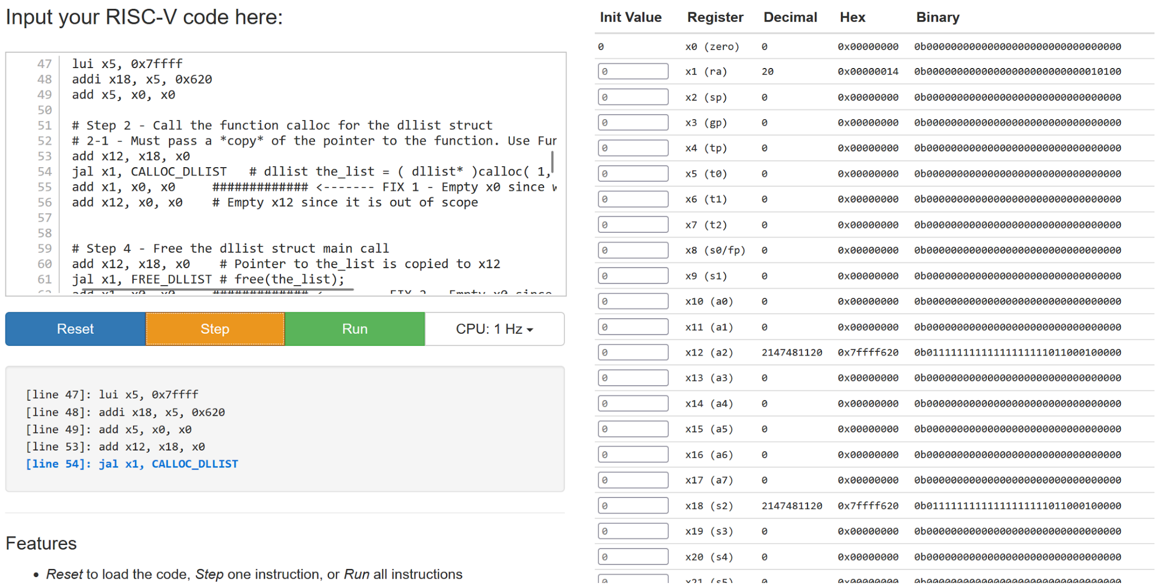
Report Deliverable: Ryan Nahm: rnahm@nd.edu, Andrew Gaylord: agaylord@nd.edu, Vinny Galassi: vgalassi@nd.edu, Matt Kennedy: mkenne24@nd.edu

Part 4.1

For the calloc of the dllist struct, we must first place the pointer to the dllist object into x12 so that the value can be passed to the CALLOC_DLLIST function. Then to calloc the memory we simply store the value 0 into all the bytes associated with the struct, which is 16 bytes worth. Next, we jalr back to the main function, using the value stored by jal in x1. Finally, set x1 and x12 to zero because they are out of scope.

We repeat this process for FREE_DLLIST, storing the address of the start of the list in x12 and returning to main with a jump and link. To free the list, we set all 16 bytes following the starting address to 0.

When calloc is initially called (x12 contains a copy of x18):



Input your RISC-V code here:

```
47  lui x5, 0x7ffff
48  addi x18, x5, 0x620
49  add x5, x0, x0
50
51  # Step 2 - Call the function calloc for the dllist struct
52  # 2-1 - Must pass a *copy* of the pointer to the function. Use Fur
53  add x12, x18, x0
54  jal x1, CALLOC_DLLIST   # dllist the_list = ( dllist* )calloc( 1,
55  add x1, x0, x0     ############# <------- FIX 1 - Empty x0 since w
56  add x12, x0, x0    # Empty x12 since it is out of scope
57
58
59  # Step 4 - Free the dllist struct main call
60  add x12, x18, x0    # Pointer to the_list is copied to x12
61  jal x1, FREE_DLLIST # free(the_list);
```

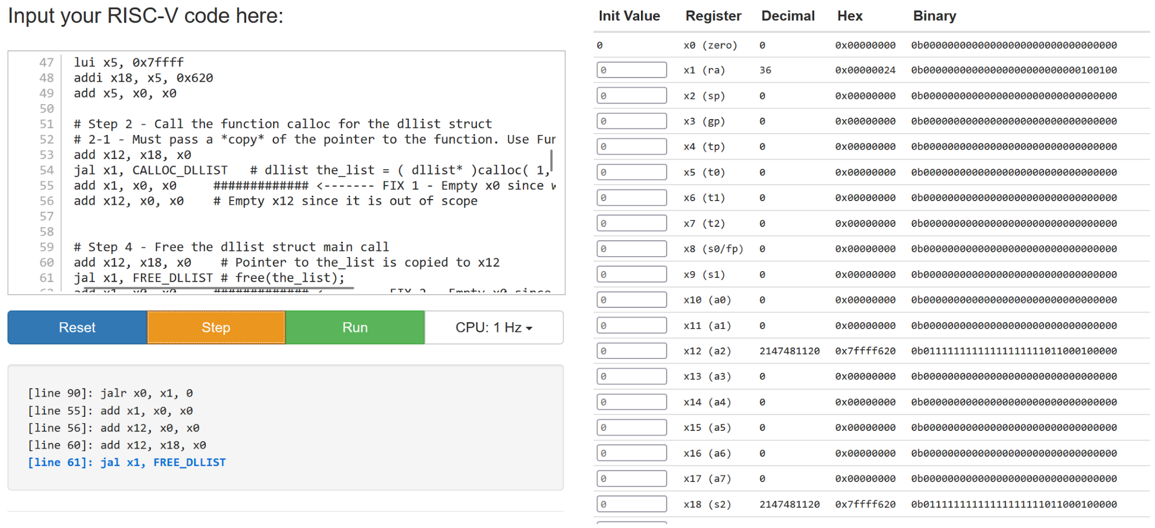
```
[line 47]: lui x5, 0x7ffff
[line 48]: addi x18, x5, 0x620
[line 49]: add x5, x0, x0
[line 53]: add x12, x18, x0
[line 54]: jal x1, CALLOC_DLLIST
```

Features

- *Reset* to load the code, *Step* one instruction, or *Run* all instructions

Init Value	Register	Decimal	Hex	Binary
0	x0 (zero)	0	0x00000000	0b00000000000000000000000000000000
0	x1 (ra)	20	0x00000014	0b00000000000000000000000000010100
0	x2 (sp)	0	0x00000000	0b00000000000000000000000000000000
0	x3 (gp)	0	0x00000000	0b00000000000000000000000000000000
0	x4 (tp)	0	0x00000000	0b00000000000000000000000000000000
0	x5 (t0)	0	0x00000000	0b00000000000000000000000000000000
0	x6 (t1)	0	0x00000000	0b00000000000000000000000000000000
0	x7 (t2)	0	0x00000000	0b00000000000000000000000000000000
0	x8 (s0/fp)	0	0x00000000	0b00000000000000000000000000000000
0	x9 (s1)	0	0x00000000	0b00000000000000000000000000000000
0	x10 (a0)	0	0x00000000	0b00000000000000000000000000000000
0	x11 (a1)	0	0x00000000	0b00000000000000000000000000000000
0	x12 (a2)	2147481120	0x7ffff620	0b01111111111111111111011000100000
0	x13 (a3)	0	0x00000000	0b00000000000000000000000000000000
0	x14 (a4)	0	0x00000000	0b00000000000000000000000000000000
0	x15 (a5)	0	0x00000000	0b00000000000000000000000000000000
0	x16 (a6)	0	0x00000000	0b00000000000000000000000000000000
0	x17 (a7)	0	0x00000000	0b00000000000000000000000000000000
0	x18 (s2)	2147481120	0x7ffff620	0b01111111111111111111011000100000
0	x19 (s3)	0	0x00000000	0b00000000000000000000000000000000
0	x20 (s4)	0	0x00000000	0b00000000000000000000000000000000
0	x21 (s5)	0	0x00000000	0b00000000000000000000000000000000

When Freelist is called (different x1 because different jal call):



Input your RISC-V code here:

```
47  lui x5, 0x7ffff
48  addi x18, x5, 0x620
49  add x5, x0, x0
50
51  # Step 2 - Call the function calloc for the dllist struct
52  # 2-1 - Must pass a *copy* of the pointer to the function. Use Fur
53  add x12, x18, x0
54  jal x1, CALLOC_DLLIST   # dllist the_list = ( dllist* )calloc( 1,
55  add x1, x0, x0     ############# <------- FIX 1 - Empty x0 since w
56  add x12, x0, x0    # Empty x12 since it is out of scope
57
58
59  # Step 4 - Free the dllist struct main call
60  add x12, x18, x0    # Pointer to the_list is copied to x12
61  jal x1, FREE_DLLIST # free(the_list);
```

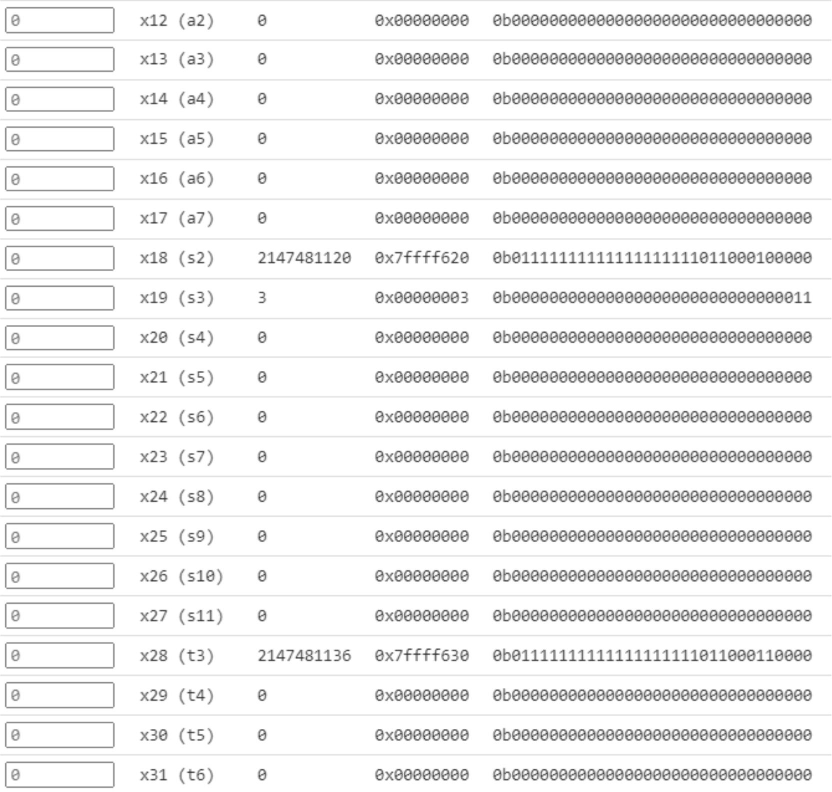
```
[line 90]: jalr x0, x1, 0
[line 55]: add x1, x0, x0
[line 56]: add x12, x0, x0
[line 60]: add x12, x18, x0
[line 61]: jal x1, FREE_DLLIST
```

Init Value	Register	Decimal	Hex	Binary
0	x0 (zero)	0	0x00000000	0b00000000000000000000000000000000
0	x1 (ra)	36	0x00000024	0b00000000000000000000000000100100
0	x2 (sp)	0	0x00000000	0b00000000000000000000000000000000
0	x3 (gp)	0	0x00000000	0b00000000000000000000000000000000
0	x4 (tp)	0	0x00000000	0b00000000000000000000000000000000
0	x5 (t0)	0	0x00000000	0b00000000000000000000000000000000
0	x6 (t1)	0	0x00000000	0b00000000000000000000000000000000
0	x7 (t2)	0	0x00000000	0b00000000000000000000000000000000
0	x8 (s0/fp)	0	0x00000000	0b00000000000000000000000000000000
0	x9 (s1)	0	0x00000000	0b00000000000000000000000000000000
0	x10 (a0)	0	0x00000000	0b00000000000000000000000000000000
0	x11 (a1)	0	0x00000000	0b00000000000000000000000000000000
0	x12 (a2)	2147481120	0x7ffff620	0b01111111111111111111011000100000
0	x13 (a3)	0	0x00000000	0b00000000000000000000000000000000
0	x14 (a4)	0	0x00000000	0b00000000000000000000000000000000
0	x15 (a5)	0	0x00000000	0b00000000000000000000000000000000
0	x16 (a6)	0	0x00000000	0b00000000000000000000000000000000
0	x17 (a7)	0	0x00000000	0b00000000000000000000000000000000
0	x18 (s2)	2147481120	0x7ffff620	0b01111111111111111111011000100000

Part 4.2

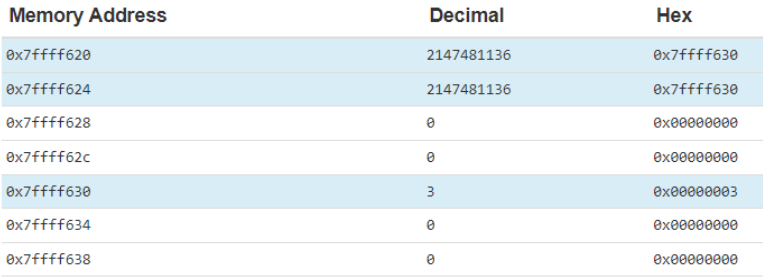
For setting up the for-loop, we began by setting register x19 (y) to 3 and then begin the for loop, breaking if y is less than 0. This will be checked at each iteration and will break after 4 loops, as -1 is added to x19 at the end of each iteration.

In order to add the first node to the DLL, we load the head pointer of the list into a register and check if it is null (or 0). If it is, we create and allocate the first node by adding 16 bytes to the address of the ddlist struct and storing the result in x28. We then load that new address into the head and tail pointer of the base list. The value of y (3) is loaded into the first 4 bytes of the new node.



0	x12 (a2)	0	0x00000000	0b00000000000000000000000000000000
0	x13 (a3)	0	0x00000000	0b00000000000000000000000000000000
0	x14 (a4)	0	0x00000000	0b00000000000000000000000000000000
0	x15 (a5)	0	0x00000000	0b00000000000000000000000000000000
0	x16 (a6)	0	0x00000000	0b00000000000000000000000000000000
0	x17 (a7)	0	0x00000000	0b00000000000000000000000000000000
0	x18 (s2)	2147481120	0x7ffff620	0b01111111111111111111011000100000
0	x19 (s3)	3	0x00000003	0b00000000000000000000000000000011
0	x20 (s4)	0	0x00000000	0b00000000000000000000000000000000
0	x21 (s5)	0	0x00000000	0b00000000000000000000000000000000
0	x22 (s6)	0	0x00000000	0b00000000000000000000000000000000
0	x23 (s7)	0	0x00000000	0b00000000000000000000000000000000
0	x24 (s8)	0	0x00000000	0b00000000000000000000000000000000
0	x25 (s9)	0	0x00000000	0b00000000000000000000000000000000
0	x26 (s10)	0	0x00000000	0b00000000000000000000000000000000
0	x27 (s11)	0	0x00000000	0b00000000000000000000000000000000
0	x28 (t3)	2147481136	0x7ffff630	0b01111111111111111111011000110000
0	x29 (t4)	0	0x00000000	0b00000000000000000000000000000000
0	x30 (t5)	0	0x00000000	0b00000000000000000000000000000000
0	x31 (t6)	0	0x00000000	0b00000000000000000000000000000000

Seen above, x28 is set to the address of the first node once the head pointer of the list is found to be null



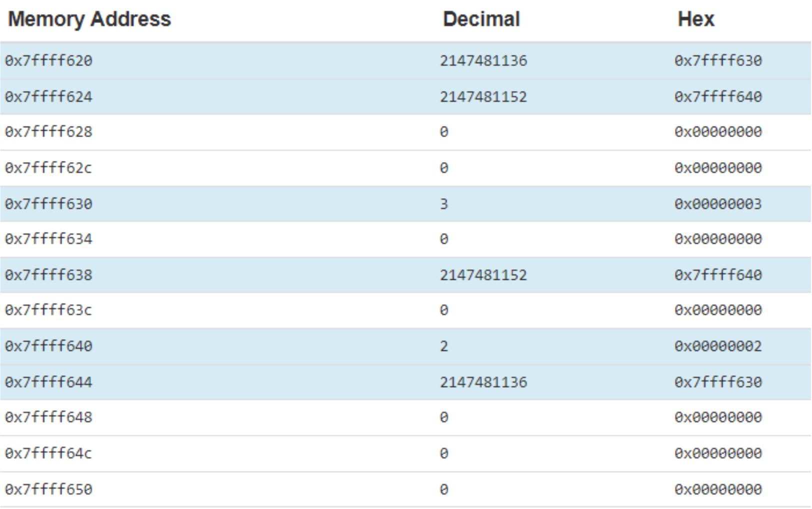
Memory Address	Decimal	Hex
0x7ffff620	2147481136	0x7ffff630
0x7ffff624	2147481136	0x7ffff630
0x7ffff628	0	0x00000000
0x7ffff62c	0	0x00000000
0x7ffff630	3	0x00000003
0x7ffff634	0	0x00000000
0x7ffff638	0	0x00000000

Seen above, the head and tail pointers both point to the first node whose value has been set to 3

Part 4.3

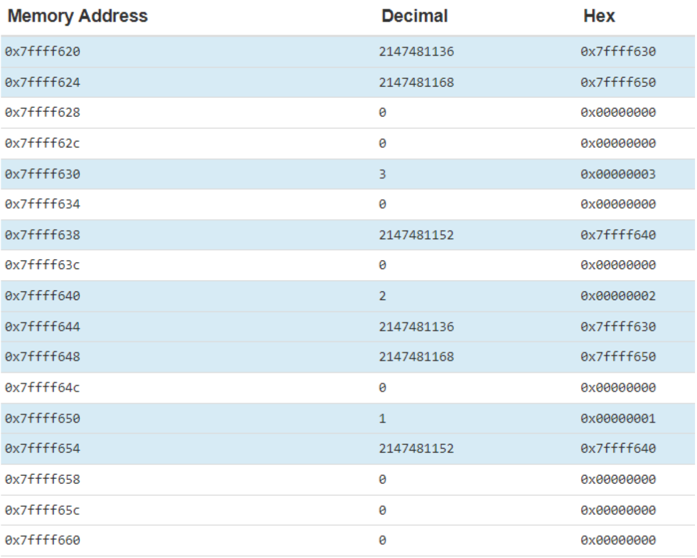
In order to insert new nodes, we first iterate through each node to find the last node of the list and load this address into a register. Then we add 16 bytes to the old final node address to get the address of our new final node. Then to create a new node, we clear all 16 bytes at the address of the new final node. After this, we store the value of y into the starting address of the new node. Then we load the address of the newly created node into the next_node value of the old final node. After this, we store the address of the old final node into the prev_node value (which is offset by 4 bytes) of the new final node. Finally, we store the address of the new final node into tail_ptr value by using the address of the head_ptr offset by 4 bytes.

This process repeats itself until the for-loop ends.



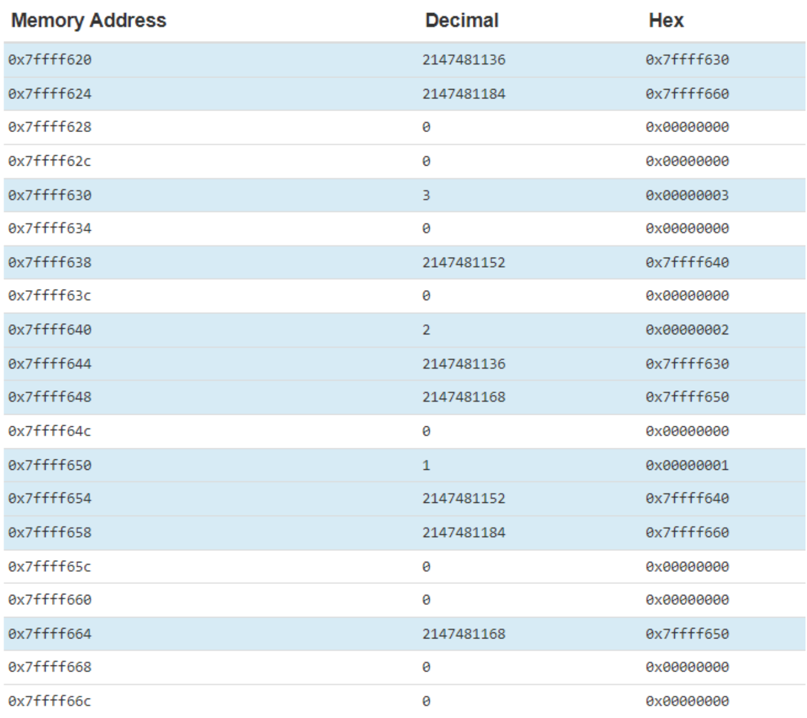
Memory Address	Decimal	Hex
0x7ffff620	2147481136	0x7ffff630
0x7ffff624	2147481152	0x7ffff640
0x7ffff628	0	0x00000000
0x7ffff62c	0	0x00000000
0x7ffff630	3	0x00000003
0x7ffff634	0	0x00000000
0x7ffff638	2147481152	0x7ffff640
0x7ffff63c	0	0x00000000
0x7ffff640	2	0x00000002
0x7ffff644	2147481136	0x7ffff630
0x7ffff648	0	0x00000000
0x7ffff64c	0	0x00000000
0x7ffff650	0	0x00000000

Memory after adding second node



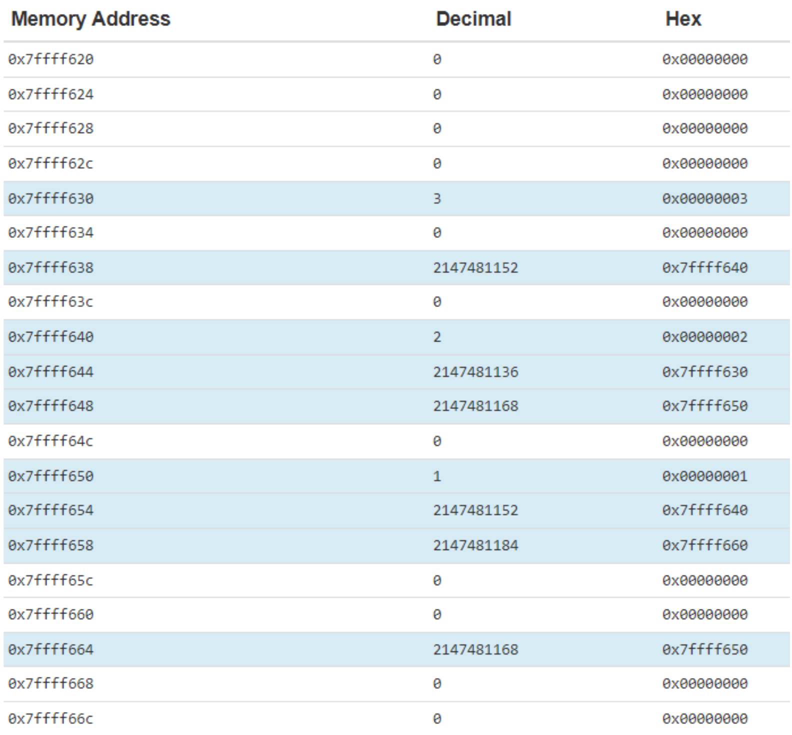
Memory Address	Decimal	Hex
0x7ffff620	2147481136	0x7ffff630
0x7ffff624	2147481168	0x7ffff650
0x7ffff628	0	0x00000000
0x7ffff62c	0	0x00000000
0x7ffff630	3	0x00000003
0x7ffff634	0	0x00000000
0x7ffff638	2147481152	0x7ffff640
0x7ffff63c	0	0x00000000
0x7ffff640	2	0x00000002
0x7ffff644	2147481136	0x7ffff630
0x7ffff648	2147481168	0x7ffff650
0x7ffff64c	0	0x00000000
0x7ffff650	1	0x00000001
0x7ffff654	2147481152	0x7ffff640
0x7ffff658	0	0x00000000
0x7ffff65c	0	0x00000000
0x7ffff660	0	0x00000000

After adding third node



Memory Address	Decimal	Hex
0x7ffff620	2147481136	0x7ffff630
0x7ffff624	2147481184	0x7ffff660
0x7ffff628	0	0x00000000
0x7ffff62c	0	0x00000000
0x7ffff630	3	0x00000003
0x7ffff634	0	0x00000000
0x7ffff638	2147481152	0x7ffff640
0x7ffff63c	0	0x00000000
0x7ffff640	2	0x00000002
0x7ffff644	2147481136	0x7ffff630
0x7ffff648	2147481168	0x7ffff650
0x7ffff64c	0	0x00000000
0x7ffff650	1	0x00000001
0x7ffff654	2147481152	0x7ffff640
0x7ffff658	2147481184	0x7ffff660
0x7ffff65c	0	0x00000000
0x7ffff660	0	0x00000000
0x7ffff664	2147481168	0x7ffff650
0x7ffff668	0	0x00000000
0x7ffff66c	0	0x00000000

After adding fourth node



Memory Address	Decimal	Hex
0x7ffff620	0	0x00000000
0x7ffff624	0	0x00000000
0x7ffff628	0	0x00000000
0x7ffff62c	0	0x00000000
0x7ffff630	3	0x00000003
0x7ffff634	0	0x00000000
0x7ffff638	2147481152	0x7ffff640
0x7ffff63c	0	0x00000000
0x7ffff640	2	0x00000002
0x7ffff644	2147481136	0x7ffff630
0x7ffff648	2147481168	0x7ffff650
0x7ffff64c	0	0x00000000
0x7ffff650	1	0x00000001
0x7ffff654	2147481152	0x7ffff640
0x7ffff658	2147481184	0x7ffff660
0x7ffff65c	0	0x00000000
0x7ffff660	0	0x00000000
0x7ffff664	2147481168	0x7ffff650
0x7ffff668	0	0x00000000
0x7ffff66c	0	0x00000000

Final memory state with four nodes and list freed

Part 4.4

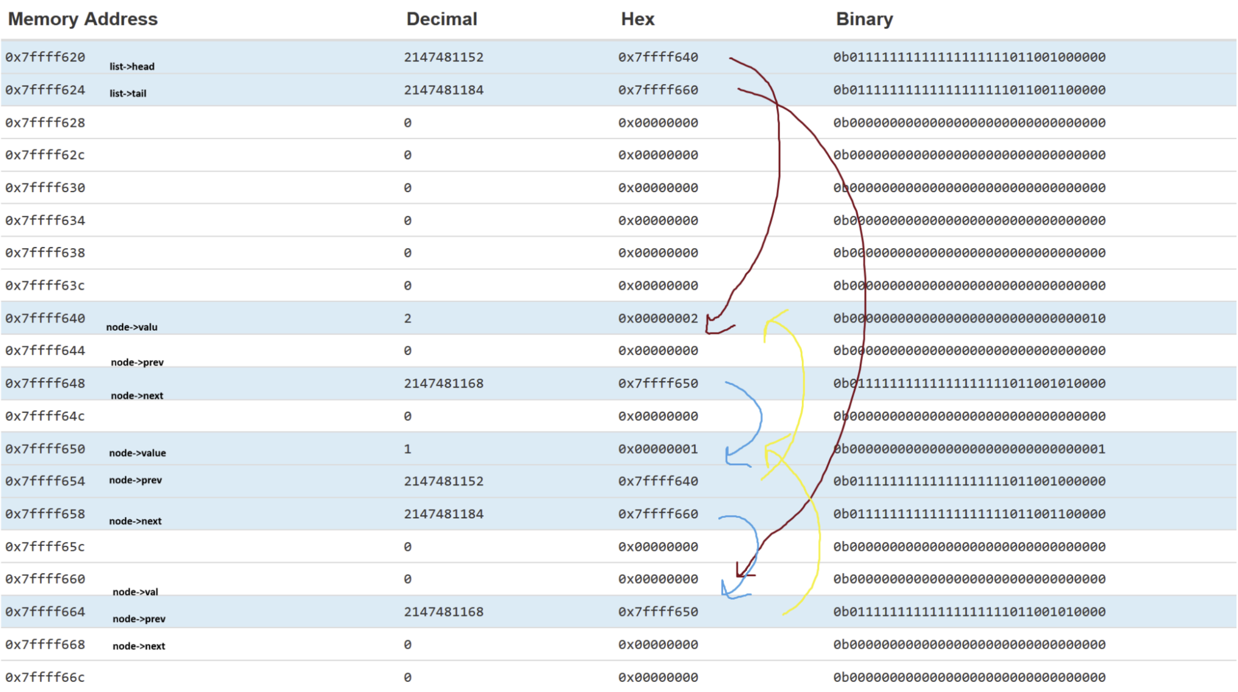
To free the nodes we begin by loading the address of the first node into a new register. We then set up a register to store the next_ptr of the current node and check if it is null (0). While it is not 0, we can free the current node by starting at the current address and setting the following 16 bytes to 0. Then, set the current node to the address of the second node and update the checking register for next_ptr. Continue this process until the register that holds the next_ptr value is 0 and then exit the loop.

Part 4.5

To find an element we pass the delete_val and the address of the head_ptr as x12 and x13 to the DELETE_NODE function.

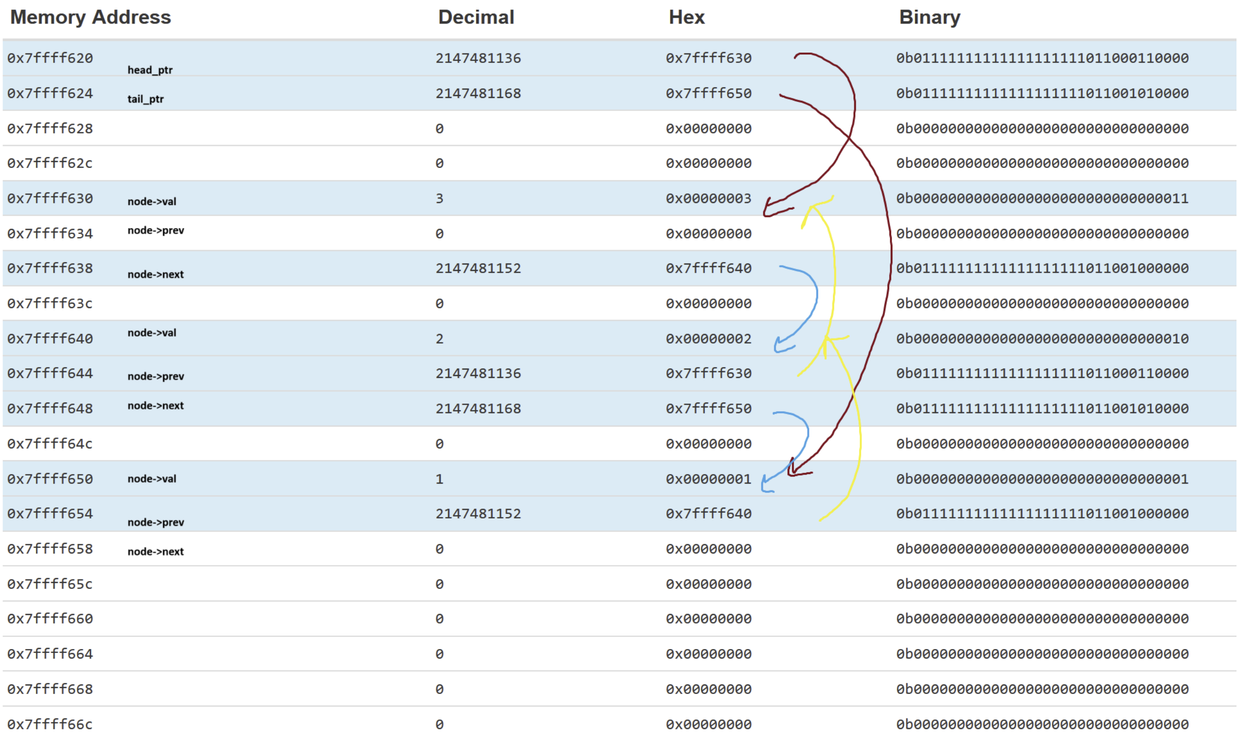
In the function, we first load the address of the first node (which is pointed to by head_ptr). A while loop is used to check if our current node is null or not: if it's NULL we exit the function, if it isn't, then we check if its value is equal to our target value. If it is, we must check the different edge cases before deciding how to restructure the pointers (each case is shown below except for a list with one value, in which case the head and tail pointers point to nothing and the node is freed). If it's not, then we move on to the next node. This cycle repeats until a node is deleted or the current node is NULL at which point we return back to main().

Case 1: head node is deleted (delete_val=3). The image below is the result of running the interpreter and stopping immediately after the node containing 3 was deleted (this is the state of our ddlist_final.S solution). Our strategy is to load the value of the next node into a temporary register and store that result in the head pointer. We then update our register that stores the head pointer (x30). An offset of 4 bytes from that address allows us to store 0 at head_ptr->prev_node.



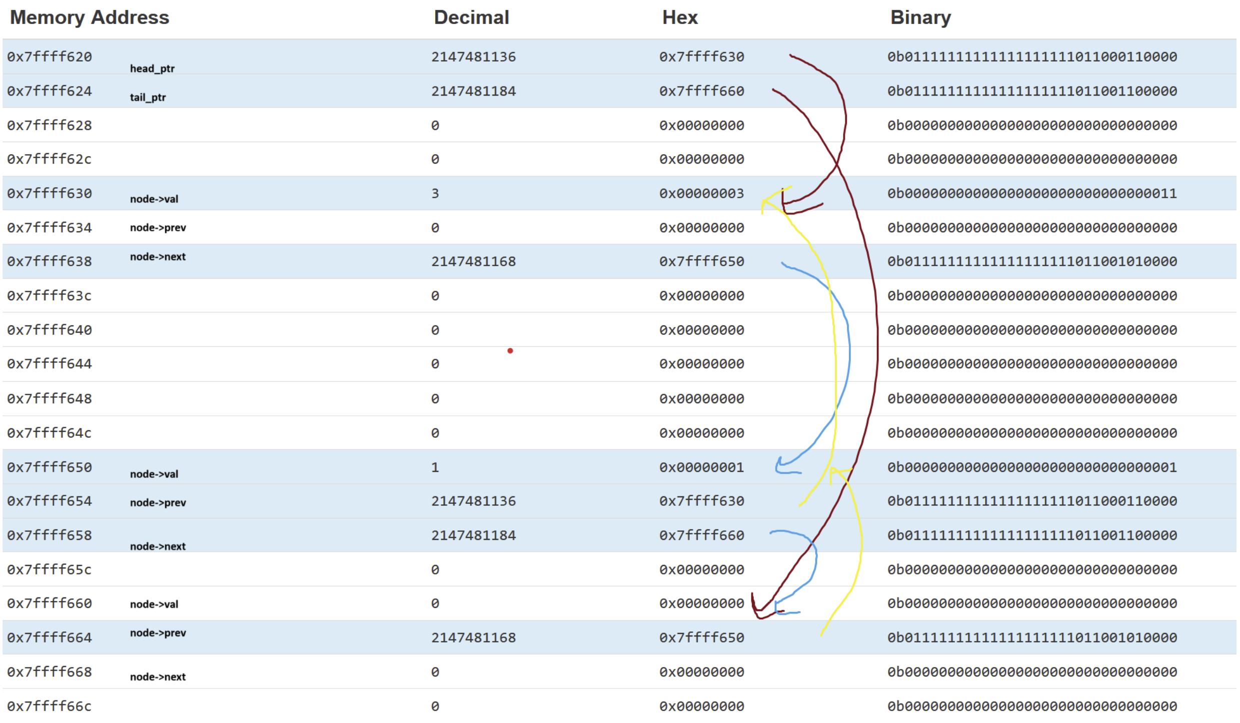
Memory Address		Decimal	Hex	Binary
0x7ffff620	list->head	2147481152	0x7ffff640	0b01111111111111111111011001000000
0x7ffff624	list->tail	2147481184	0x7ffff660	0b01111111111111111111011001100000
0x7ffff628		0	0x00000000	0b00000000000000000000000000000000
0x7ffff62c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff630		0	0x00000000	0b00000000000000000000000000000000
0x7ffff634		0	0x00000000	0b00000000000000000000000000000000
0x7ffff638		0	0x00000000	0b00000000000000000000000000000000
0x7ffff63c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff640	node->valu	2	0x00000002	0b00000000000000000000000000000010
0x7ffff644	node->prev	0	0x00000000	0b00000000000000000000000000000000
0x7ffff648	node->next	2147481168	0x7ffff650	0b01111111111111111111011001010000
0x7ffff64c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff650	node->value	1	0x00000001	0b00000000000000000000000000000001
0x7ffff654	node->prev	2147481152	0x7ffff640	0b01111111111111111111011001000000
0x7ffff658	node->next	2147481184	0x7ffff660	0b01111111111111111111011001100000
0x7ffff65c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff660	node->val	0	0x00000000	0b00000000000000000000000000000000
0x7ffff664	node->prev	2147481168	0x7ffff650	0b01111111111111111111011001010000
0x7ffff668	node->next	0	0x00000000	0b00000000000000000000000000000000
0x7ffff66c		0	0x00000000	0b00000000000000000000000000000000

Case 2: tail node is deleted (target_val=0). The image below is the result of changing the value of **int delete_val to 0** running the interpreter and stopping immediately after the node containing 0 was deleted. (Note, in this run, 3 is *not* deleted). The method here is to load the value of the previous node into a temporary register and store that result in the tail pointer. We then update our register that stores the tail pointer (x31). An offset of 8 bytes from that address allows us to store 0 at tail_ptr->next_node.



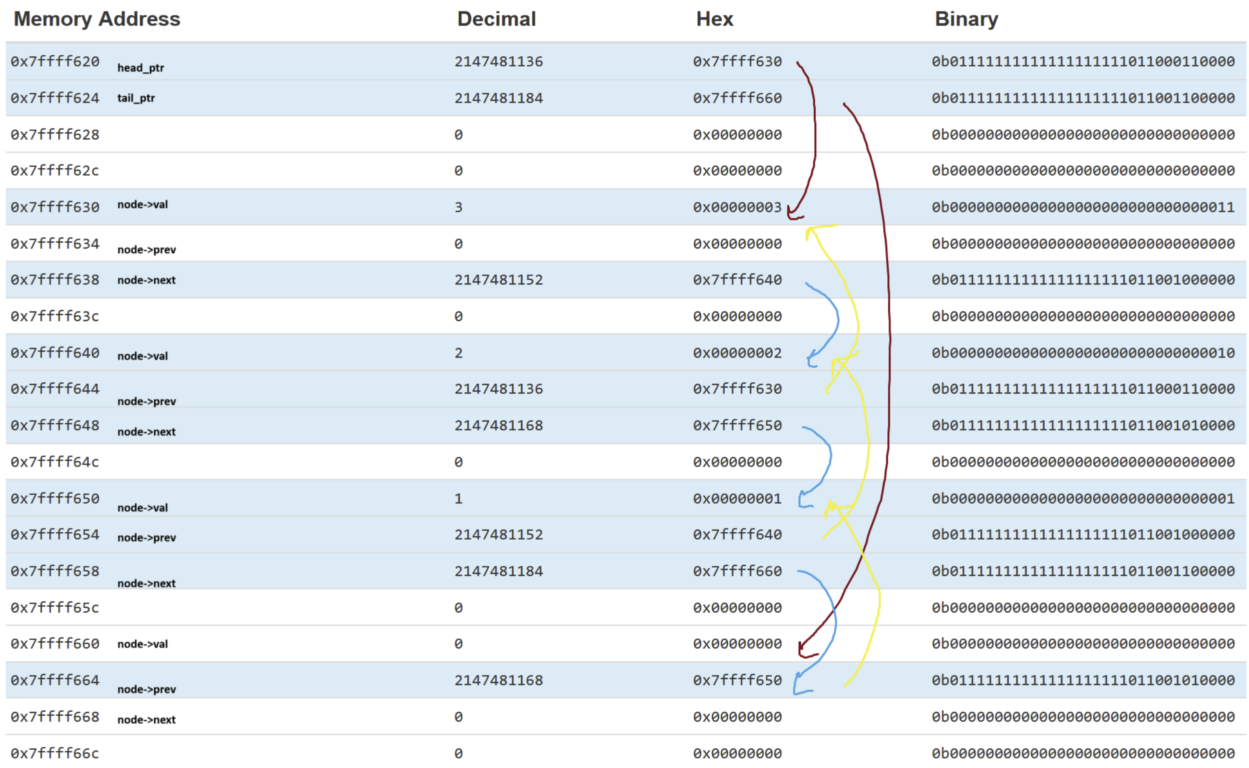
Memory Address		Decimal	Hex	Binary
0x7ffff620	head_ptr	2147481136	0x7ffff630	0b01111111111111111111011000110000
0x7ffff624	tail_ptr	2147481168	0x7ffff650	0b01111111111111111111011001010000
0x7ffff628		0	0x00000000	0b00000000000000000000000000000000
0x7ffff62c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff630	node->val	3	0x00000003	0b00000000000000000000000000000011
0x7ffff634	node->prev	0	0x00000000	0b00000000000000000000000000000000
0x7ffff638	node->next	2147481152	0x7ffff640	0b01111111111111111111011001000000
0x7ffff63c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff640	node->val	2	0x00000002	0b00000000000000000000000000000010
0x7ffff644	node->prev	2147481136	0x7ffff630	0b01111111111111111111011000110000
0x7ffff648	node->next	2147481168	0x7ffff650	0b01111111111111111111011001010000
0x7ffff64c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff650	node->val	1	0x00000001	0b00000000000000000000000000000001
0x7ffff654	node->prev	2147481152	0x7ffff640	0b01111111111111111111011001000000
0x7ffff658	node->next	0	0x00000000	0b00000000000000000000000000000000
0x7ffff65c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff660		0	0x00000000	0b00000000000000000000000000000000
0x7ffff664		0	0x00000000	0b00000000000000000000000000000000
0x7ffff668		0	0x00000000	0b00000000000000000000000000000000
0x7ffff66c		0	0x00000000	0b00000000000000000000000000000000

Case 3: deletion of internal node (delete_val=2). The third image is the result of changing the value of **int delete_val to 2** running the interpreter and stopping immediately after the node containing 2 was deleted (note, in this run, neither 3 nor 0 is *not* deleted). In this case, we load the value of the previous and next node of the current node into temporary registers. An offset of 8 bytes from x28 gives us prev_node->next_node, to which we can store x29 (which is curr->next_node). Similarly, an offset of 4 bytes from x29 gives us next_node->prev_node, to which we store x28 (which is curr->prev_node). We also must clear the temporary registers.



Memory Address		Decimal	Hex	Binary
0x7ffff620	head_ptr	2147481136	0x7ffff630	0b01111111111111111111011000110000
0x7ffff624	tail_ptr	2147481184	0x7ffff660	0b01111111111111111111011001100000
0x7ffff628		0	0x00000000	0b00000000000000000000000000000000
0x7ffff62c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff630	node->val	3	0x00000003	0b00000000000000000000000000000011
0x7ffff634	node->prev	0	0x00000000	0b00000000000000000000000000000000
0x7ffff638	node->next	2147481168	0x7ffff650	0b01111111111111111111011001010000
0x7ffff63c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff640		0	0x00000000	0b00000000000000000000000000000000
0x7ffff644		0	0x00000000	0b00000000000000000000000000000000
0x7ffff648		0	0x00000000	0b00000000000000000000000000000000
0x7ffff64c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff650	node->val	1	0x00000001	0b00000000000000000000000000000001
0x7ffff654	node->prev	2147481136	0x7ffff630	0b01111111111111111111011000110000
0x7ffff658	node->next	2147481184	0x7ffff660	0b01111111111111111111011001100000
0x7ffff65c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff660	node->val	0	0x00000000	0b00000000000000000000000000000000
0x7ffff664	node->prev	2147481168	0x7ffff650	0b01111111111111111111011001010000
0x7ffff668	node->next	0	0x00000000	0b00000000000000000000000000000000
0x7ffff66c		0	0x00000000	0b00000000000000000000000000000000

Now, trying with a value of -1, nothing is deleted. In this circumstance, the while loop just runs, checks if every the_int in each node contains the target, and eventually sees that the curr_ptr in x7 is NULL and then exits.



Memory Address		Decimal	Hex	Binary
0x7ffff620	head_ptr	2147481136	0x7ffff630	0b01111111111111111111011000110000
0x7ffff624	tail_ptr	2147481184	0x7ffff660	0b01111111111111111111011001100000
0x7ffff628		0	0x00000000	0b00000000000000000000000000000000
0x7ffff62c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff630	node->val	3	0x00000003	0b00000000000000000000000000000011
0x7ffff634	node->prev	0	0x00000000	0b00000000000000000000000000000000
0x7ffff638	node->next	2147481152	0x7ffff640	0b01111111111111111111011001000000
0x7ffff63c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff640	node->val	2	0x00000002	0b00000000000000000000000000000010
0x7ffff644	node->prev	2147481136	0x7ffff630	0b01111111111111111111011000110000
0x7ffff648	node->next	2147481168	0x7ffff650	0b01111111111111111111011001010000
0x7ffff64c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff650	node->val	1	0x00000001	0b00000000000000000000000000000001
0x7ffff654	node->prev	2147481152	0x7ffff640	0b01111111111111111111011001000000
0x7ffff658	node->next	2147481184	0x7ffff660	0b01111111111111111111011001100000
0x7ffff65c		0	0x00000000	0b00000000000000000000000000000000
0x7ffff660	node->val	0	0x00000000	0b00000000000000000000000000000000
0x7ffff664	node->prev	2147481168	0x7ffff650	0b01111111111111111111011001010000
0x7ffff668	node->next	0	0x00000000	0b00000000000000000000000000000000
0x7ffff66c		0	0x00000000	0b00000000000000000000000000000000