

# CSE12 Lab 4: Merging sorted Linked Lists

Due June 7th 11:59 on Gradescope

---

## Minimum Gradescope Submission Requirements

Ensure that your Lab4 submission contains the following files:

- *newNode.asm*
- *print\_list.asm*
- *insertSorted.asm*
- *mergeLinkedLists.asm*

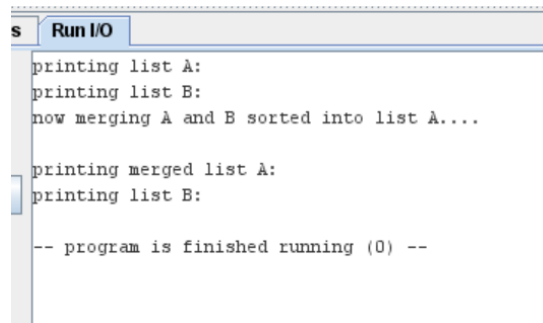
## Objective

By the end of this lab, students will be able to:

1. Implement functions in RISC V assembly adhering to RISC-V register conventions.
2. Understand the architecture and basic operations of the data structure known as linked list in RISC-V assembly.
3. Implement functions for creating new nodes and printing the linked list which contains a list of sorted integers.
4. Write assembly code to insert nodes in a sorted manner.
5. Merge two sorted linked lists into one.

You will need to correctly write code into the files: *newNode.asm*, *print\_list.asm*, *insertSorted.asm*, *mergeLinkedLists.asm* , and then submit them to Gradescope. These files as they appear in the Lab4 folder are currently incomplete.

If you run the testbench file *lab4\_testbench.asm* as it is (ensuring all other provided .asm files are in the same runtime folder), you will get the following output:



```
s Run I/O
printing list A:
printing list B:
now merging A and B sorted into list A...

printing merged list A:
printing list B:

-- program is finished running (0) --
```

If you submit the files exactly as they are provided, you will get grace points for submission as explained later in the Grading Rubric section.

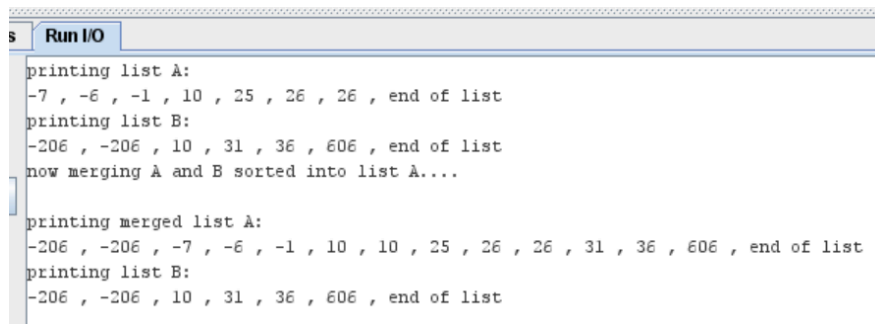
In the testbench file, we create 2 distinct linked lists, A and B. Into list A, we fill in the keys in sequence: -1, 10, -7, -6, 26, 25, 26. However, after we insert the keys correctly in order, the list should appear as: -7, -6, -1, 10, 25, 26, 26. Similarly, into list B, we fill in the keys in sequence: -1, 10, -206, -206, 36, 606. However, after we insert the keys correctly in order, the list should appear as: -206, -206, 10, 31, 36, 606.

We created the above 2 lists with the function *insertSorted* contained in *insertSorted.asm*. This function itself uses the function *newNode* contained in *newNode.asm*.

We then merge the 2 linked lists A and B using the function *mergeLinkedLists* contained in *mergeLinkedLists.asm*. The merged list is now contained by list A.

We print the lists in this assignment using the *print\_list* function contained in *print\_list.asm*.

When we run (the completed) lab4\_testbench.asm in RARS, we should get the output console as shown below.



```

s Run I/O
printing list A:
-7 , -6 , -1 , 10 , 25 , 26 , 26 , end of list
printing list B:
-206 , -206 , 10 , 31 , 36 , 606 , end of list
now merging A and B sorted into list A...

printing merged list A:
-206 , -206 , -7 , -6 , -1 , 10 , 10 , 25 , 26 , 26 , 31 , 36 , 606 , end of list
printing list B:
-206 , -206 , 10 , 31 , 36 , 606 , end of list

```

## Learning Outcomes

We After completing this lab assignment, you should be able to:

- Understand how to write assembly code in RISC V for functions.  
Please go through the following two provided files before reading any further: *add\_function.asm* and *multiply\_function.asm*. These 2 files will give you a basic introduction to how to write functions in general before we move on to the more complicated stuff.
- Understand how a linked list is implemented in the memory of a Von Neumann architecture.

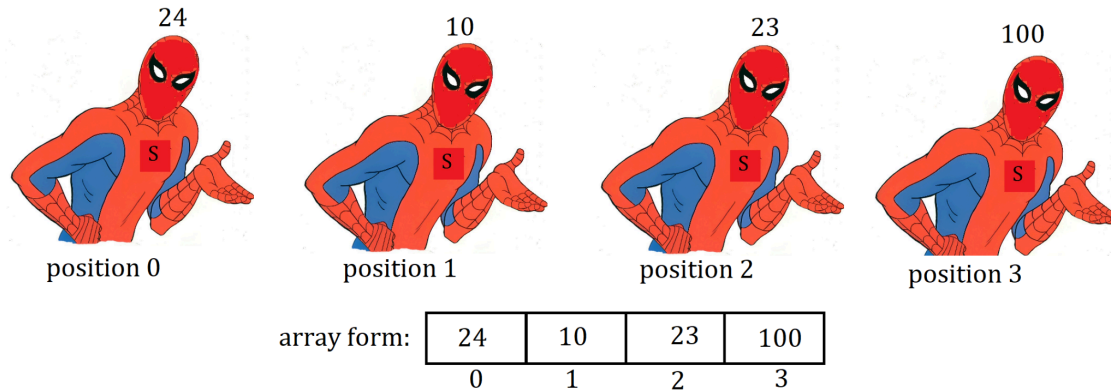
## From Arrays to Linked Lists: The Spoder-men Perspective

Arrays have become familiar terrain for many of you. An array is essentially a **data structure** of contiguous memory locations where each location contains an item/element of the same data type. For example, picture an integer array "arr" of size 5, i.e. containing 5 integers in the sequence: arr={-7, 12, 10, 4, 20}. Think of them as a parking lot where each space is meant for a vehicle of the exact same size. It's ordered, it's neat, and it's predictable. However, as you'll soon discover, the world of data structures offers terrains far more intricate than a simple parking lot. Indeed, data structures will later constitute an important part of your CS curriculum.

To help us navigate this terrain of data structures (and the ones known as **Linked Lists** in particular), we call upon the Spoder-Men, representatives from a multiverse where each one carries a unique identifier – a '**key**' that denotes their specific universe. For instance, the Spoder-man from universe#24 proudly wears the key 24.

## Array Challenges: A Linear Spoder-Parade

Now, let us arrange these Spoder-men in a line (akin to an array). In the following example, the Spoder-men from universes 24, 10, 23, 100 arrived in a *preordered* fashion. We have neatly arranged them in an array of size 4 as shown below:



We can abstract the above arrangement into a more recognisable array scenario. So we know beforehand, it is only 4 elements (4 spoder-men) and we accordingly size our array to 4 and put the array values in the given order .

While this arrangement is convenient for now, we'd bump into a few hurdles:

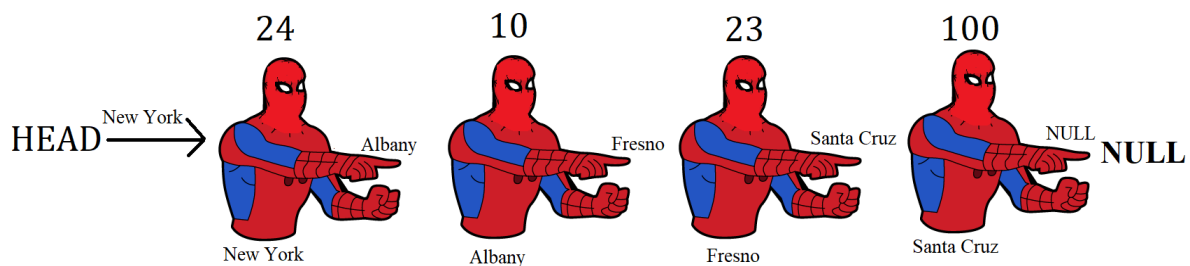
1. *Size Constraints*: The line has a fixed capacity (currently of size 4 in example). Should another Spoder-man appear (because, let's face it, the multiverse is vast), he'd be left standing if all slots are occupied (because the array has *predefined length*).
2. *Insertion and Deletion Woes*: Need a Spoder-man to leave or a new one to slot in between? Everyone after must shuffle around, making the process cumbersome.
3. *Wasted Spaces*: Though spaces can be reserved anticipating new arrivals, they often remain unoccupied, leading to inefficient use of space.

The Spoder-men, in their infinite wisdom, realized the need for something more *dynamic* as opposed to the array-like structure, which is *static*.

## Linked Lists Enter the Scene: A Spoder-chain?

Spoder-men are infamous for pointing at each other. Realizing this, instead of a static line, each Spoder-man points to the next, crafting a flexible chain. Here's how it works. Remember the previous array, while having indexed positions of 0,1,2,.. was itself occupying physical space. We will now conceive a simple linked list of each Spoder-man pointing to the next Spoder-man in the intended sequence. Let us say we still need to maintain the order of placing Spoder-Man based on their universe key value sequence: 24,10,23,100. Assume the Spoder-man with the respective key numbers are in the current locations New York, Albany, Fresno and Santa Cruz respectively.

To organize a data structure that is able to maintain this above sequence, we set up a starting reference, which we call HEAD. HEAD itself is a location that does not contain a Spoder-man. Instead, it merely indicates only the start of the Linked List. It only *points* to the first general item in the linked list. More specifically, HEAD contains the address of the location of Spoder-Man from universe 24, that is New York. Following HEAD's direction we land in New York and find Spoder-man with key 24. This Spoder-man in turn points to the location of the next Spoder-Man in sequence, Spoder-man 10, i.e. Albany. Following Spoder-man 24's direction, we head to Albany to locate Spoder-man 10 who in turn points to Spoder-man 23 in Fresno, who in turn points to Spoder-man 100 in Santa Cruz. For our given current list, no Spoder-man exists after Spoder-man 100, so Spoder-man 100 points to NULL, aka *Nothing*. This pointing game is illustrated below.



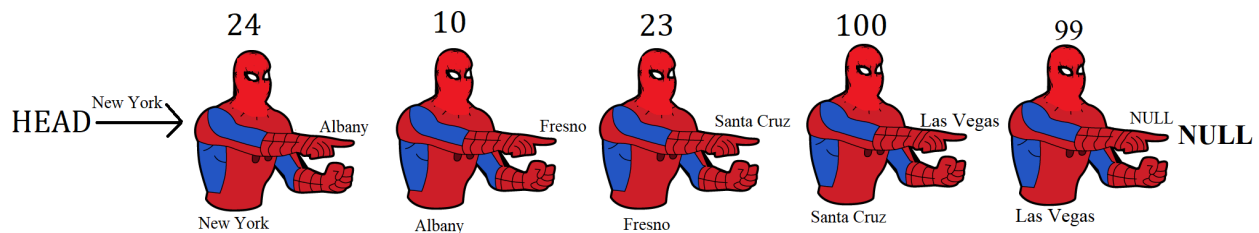
Technically speaking, each Spoder-man constitutes a *Node* in the Linked List. HEAD itself is often called the head node. If the linked list were empty, HEAD would directly point to NULL. The values 24,10,23,100 in the example are technically called *keys*.

The Advantages:

1. *Adaptable Structure*: When a Spoder-man joins the crew, he merely stands next to the last in line, getting pointed to by his predecessor. No need for a fixed slot.

2. *Easy Additions and Removals*: Should a Spoder-man wish to exit, the one before merely adjusts his pointing to the subsequent one. A seamless transition, sans the shuffle.

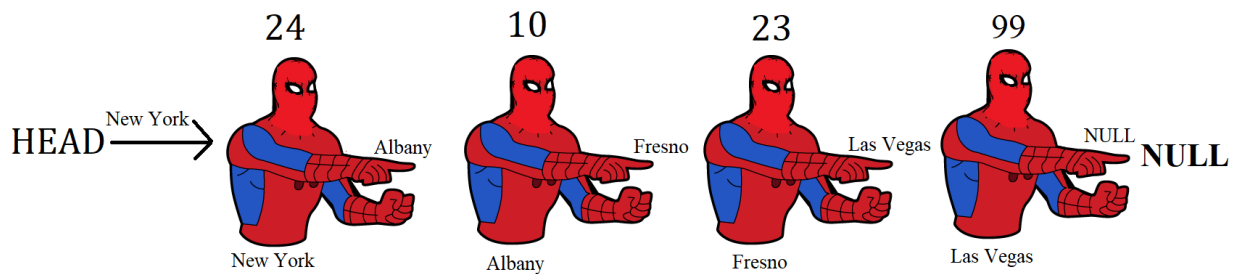
Look at the case below of linking Spoder-man 99, who is in Las Vegas, into the aforementioned sequence.



Here, we would have technically said that the linked list dynamically grew to insert one more node with key=99 at the end.

We could be even more flexible with our node insertion and deletion in this linked list. Now, let's say Spoder-man 100 wishes to exit the linked list and depart for his Universe because the rent prices in Santa

Cruz are becoming atrocious(and Spoder-men are infamous for being late on paying their rent). We can easily readjust the linked list as shown below with the updated pointers.



Similarly, it is possible for us to insert a new node somewhere in between as well

3. *Efficiency*: No more vacant slots. The chain continues till the last Spoder-man, who signifies the end by pointing to none.

In short, arrays serve well when numbers and the sequence pattern are set in stone. But for scenarios demanding flexibility, the dynamic nature of linked lists, as illustrated by our Spoder-men, comes into its own.

## Practical Linked List Implementation in memory

Exploring linked lists within the realm of assembly language offers a unique perspective, especially for those encountering the concept for the first time. Unlike high-level languages such as C/C++, where abstractions like the *struct* variable type and *pointers* (denoted by the \* keyword in C) are used, assembly allows you to directly engage with how keys and nodes interact in the computer's physical memory. Such high-level abstractions in HLL, while useful, can sometimes obscure the intricate relationship between linked lists and the actual memory addresses. Studying this in assembly provides a clearer and more direct understanding of these relationships. Remember that we are basically using the notional machine framework of looking at all possible types of computation, including operating on a linked list, as nothing more than the guided transaction of data between the CPU and memory.



Let us now see how we can build the previously mentioned Spoder-man list into a more practical data structure in memory.

We can define each node as an entity that consists of 2 things:

1. The **key** value (this is an integer value and will take up 4 bytes)
2. The **next node** that the **current node** is pointing to, i.e. the address of the next node (this will take up 4 bytes as well). So we can say that each node in our physical implementation contains a total of 8 bytes.

For example, consider the Spider-man linked list in the previous discussion having the following keys in nodes N1, N2, N3, N4 in sequence: HEAD->N1(24)->N2(10)->N3(23)->N4(100)->NULL

In RISC-V, all linked lists will be created starting in the **Heap Memory, i.e., the memory section starting from address 0x10040000**. So the HEAD is contained at location 0x10040000. HEAD itself has no key and only contains the pointer to node N1. Thus HEAD encapsulates only 4 bytes of space. The other nodes in a given linked list will contain 8 bytes.

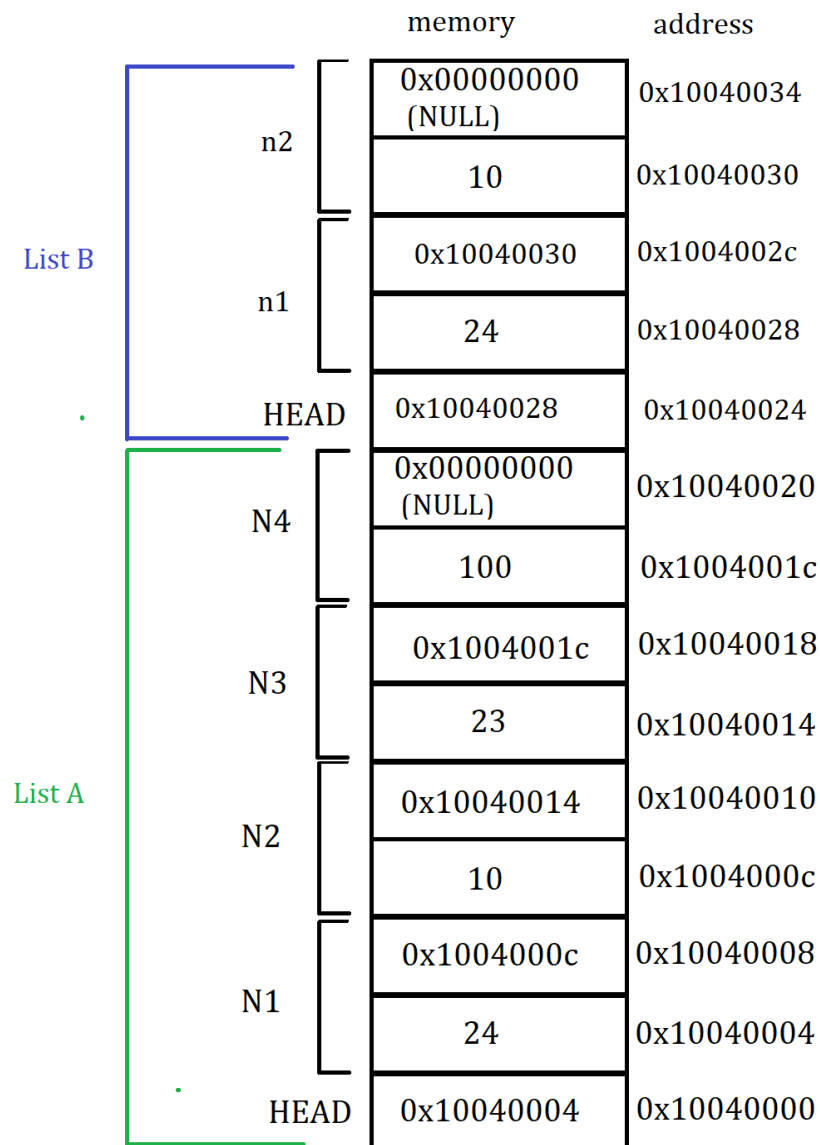
Assume that nodes are located in the following addresses:

N1 in 0x10040004, N2 in 0x1004000c, N3 in 0x10040014, N4 in 0x1004001c. Within each node itself (besides HEAD), the first 4 bytes contain the key and the next 4 contain the next node pointer address.

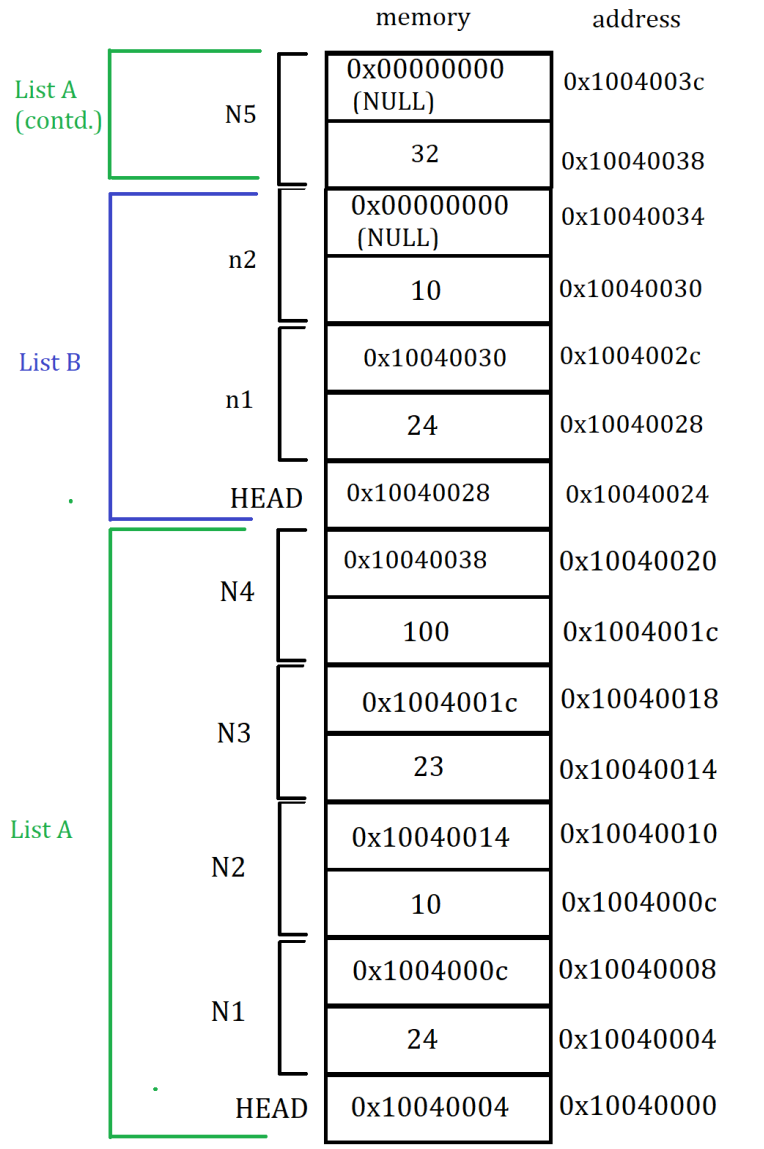
Thus the list HEAD->N1(24)->N2(10)->N3(23)->N4(100)->NULL will look in our Heap memory like the figure below:

	memory	address
N4	0x00000000 (NULL)	0x10040020
	100	0x1004001c
N3	0x1004001c	0x10040018
	23	0x10040014
N2	0x10040014	0x10040010
	10	0x1004000c
N1	0x1004000c	0x10040008
	24	0x10040004
HEAD	0x10040004	0x10040000

Likewise, imagine that we were to form a new linked list on top of the existing one: HEAD\_2->n1(-21)->n2(30)->NULL. Let's call this second list B and the previous one A. Then our updated heap memory section would look like below:



If we want to operate in future on these lists A and B, such as printing the list, finding min/max key, adding new nodes, etc, then we should save for posterity the location of these 2 lists. By default, the location of a linked list refers to the location of that list's HEAD. In our example therefore, A is located at 0x10040000 and B is located at 0x10040024. Note that if we wanted to grow our list A further at this point, then the next available address available for node N4 to point to for the new node N5 would be 0x10040038, i.e.  $\text{mem}[0x10040020] = 0x10040038$ . The updated heap memory diagram after N5 insertion would look like below (assuming key value of 32):



In our implementation, to grow the allocated memory space in Heap memory, we are going to utilize the system break (sbrk) ecall (a7=9, a0=number of bytes allocated). On returning from ecall, a0 will contain the starting address of the allocated bytes. Refer to the example file *test\_sbrk.asm* to understand the workings of sbrk since it will be relevant to implementing the newNode and insertSorted functions in the Lab 4 assignment. Feel free to modify the file *test\_sbrk.asm* by repeatedly calling the sbrk ecall and see how it changes the return value in a0 register after each call.

## Some Helpful Pseudocode

At this point in our discussion, it is worthwhile to introduce some language agnostic pseudocode to help you to decide how to implement the linked lists discussed so far in assembly.



Specifically, we are going to focus on the functions from the assignment *newNode* and *print\_list*. Instead of the function *insertSorted*, we will be looking at the much simpler *insert* function which just adds a node at the end of a given linked list without maintaining any ascending/descending order. Understanding how these functions work will go a long way in helping you to figure out solving for the more complicated functions *insertSorted* and *mergeLinkedLists*. In the pseudocode below, we use the “#” character to indicate a comment.

*Structure Node:*

*Declare key As Integer*

*Declare next As Node*

#-----#

*Function newNode(key As Integer) -> Node:*

#the “->” symbol means what the function is returning

*Declare temp As Node = Allocate a new Node*

*temp.key = key*

*temp.next = Null*

*Return temp*

# the dot operator

*End Function*

#-----#

*Function insert(Reference head As Node, key As Integer):*

*Declare newNode As Node = newNode(key)*

*// If the list is empty, set the new node as the head*

*If head Is Null:*

*head = newNode*

*Return*

*End If*

*Declare temp As Node = head*

*// Traverse to the end of the list*

*While temp.next IsNot Null:*

*temp = temp.next*

*End While*

*// Add the new node to the end of the list*

*temp.next = newNode*

*End Function*

#-----#

*Function print\_list(head As Node):*

*Declare temp As Node = head*

```

While temp IsNot Null:
    Print temp.key and a space
    temp = temp.next
End While
Print a newline
End Function
#-----#

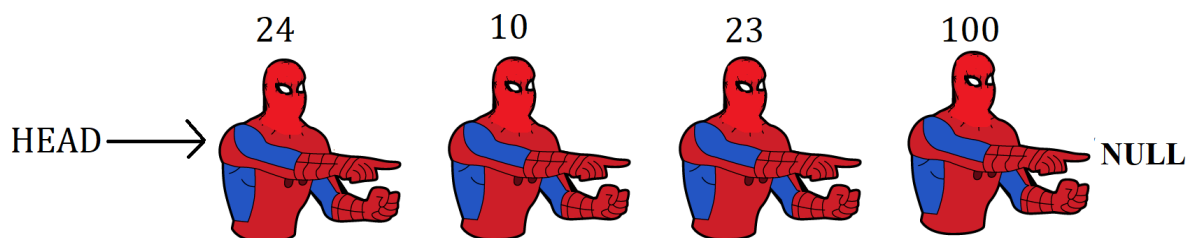
```

**PRO TIP#1:** Make sure you know how to correctly implement functions in assembly, following the correct RISC-V register save convention. Refer to *add\_function.asm* and *multiply\_function.asm* for examples on this. Your implementation later on of the function *mergeLinkedLists* would be expected to itself call on *insertSorted* function as a nested function call so it is helpful to see how the *multiply function* in the provided example calls *add* in a nested manner as well.

**PRO TIP#2:** It is highly recommended that you first correctly understand how to build a generic linked list insertion routine in assembly language before trying out the specialized *insertSorted* function pertinent to Lab4. Also note that in the *newNode* pseudocode, the moment we create a new Node (before even having linked it to the linked list through *insert* function), we are initializing the *next* node of this new node to NULL. What that boils down to in assembly is that within the 8 bytes allocated for the node structure, while the 1st 4 bytes hold the key, the next 4 bytes are explicitly to 0. Looking at how the *createHead* function was implemented in the *lab4\_testbench.asm* file will give you a good starting point to implement the functions listed in the pseudocode section above.

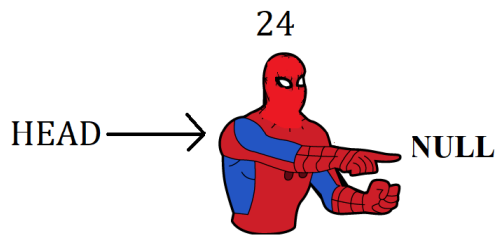
## How *insertSorted* is supposed to work: The Spider-men Perspective

We will now again reuse the previous Spider-man list example where the different Spider-men with unique universe key values arrived in the sequence: 24, 10, 23, 100. This led us to create the Linked list below which we had shown previously as well:

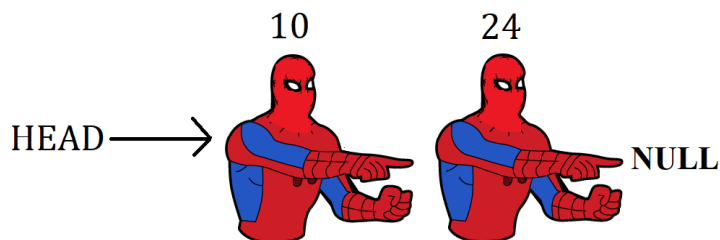


For convenience, and because we are no longer concerned with locations in cities but locations in the computer memory, we have removed the reference to location names in the above image. If you were to have implemented your linked list inserting nodes using the *insert* pseudocode described previously, the above image is what your list would have looked like. However, let us see how *insertSorted*, which in lab4 is meant to insert new nodes in an ascending order, is supposed to look like for the sequence: 24, 10, 23, 100

1. Keys entered: 24

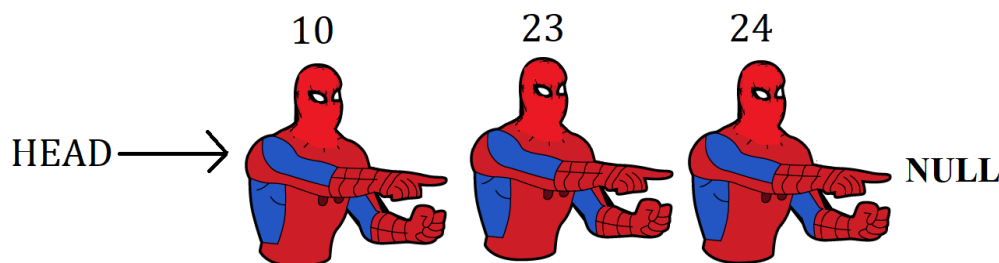


2. Keys entered: 24, 10

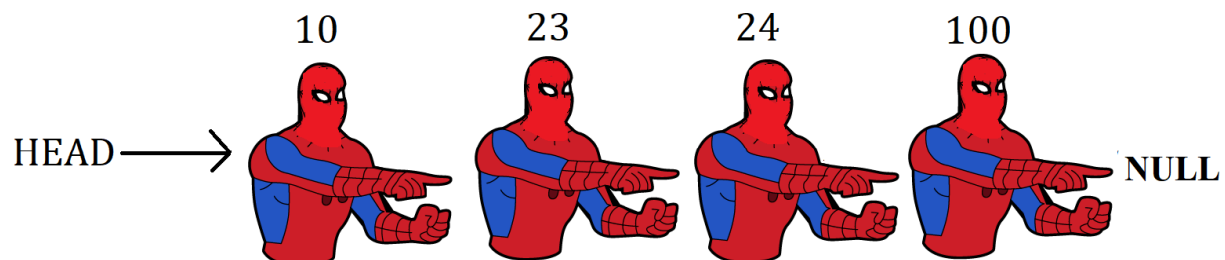


A very important thing to point out at this point is that Spoder-man with key 24 did **NOT** *physically* readjust himself with respect to HEAD location upon the insertion of Spoder-man with key 10. Instead, only the pointer locations within each node were correctly updated to reflect the maintenance of ascending order of key values in the list. This trend will continue with respect to *insertSorted* for the insertion of the remaining nodes/Spoder-men.

3. Keys entered: 24, 10, 23



3. Keys entered: 24, 10, 23, 100



You should by now be able to chart out a rudimentary algorithm for implementing *insertSorted*. When you are inserting a NEW node(with the NEW key) to list, you would first check to see if the list is empty. If it is so, make the NEW node your head and exit. Else, you keep on traversing the list until you hit a node where the key becomes larger than NEW key. It is here you need to break the chain and make the appropriate adjustments to the next node location values.

**PRO TIP#3:** When you think you have narrowed down the pseudocode and then the actual assembly code, make sure you have subjected your code to enough varying test cases! If you look at *lab4\_testbench.asm*, you will see that the testbench code is inserting data from *fill\_listA.asm* and *fill\_listB.asm*. You can try applying your insertSorted solution to these 2 lists and see if you get the intended result. You may have to continuously refine your code for new edge cases.

## How *mergeLinkedLists* is supposed to work

At this point, let us assume you have successfully implemented all the functions: *newNode.asm*, *print\_list.asm*, *insertSorted.asm*. We will now see how *mergeLinkedLists* should execute within the context of *lab4\_testbench.asm*. Assume that you have 2 **already sorted** lists in your heap memory, A and B. The files *fill\_listA.asm* and *fill\_listB.asm* deal with bigger list size but for brevity's sake, we are going to use smaller lists here.

For list A, assume we entered keys in sequence: 3, 10, 5 in *fill\_listA.asm*. The list becomes : 3->5-> 10  
For list B, assume we entered keys in sequence: 20, 16 in *fill\_listB.asm*. The list becomes : 16->20

You can see it is going to be useful to keep track of the location of each list A and B. The location of HEAD for both A and B are kept in the extern memory section(starting from 0x10000000) of memory in the executed testbench code as shown below:

Data Segment		
Address	Value (+0)	Value (+4)
0x10000000	0x10040000	0x1004001c

As we can see, A's HEAD is located in heap memory at 0x10040000 and B's HEAD is located at 0x1004001c.

If we were to inspect the heap memory at those specific locations, you should be able to trace out the location of each node with A and B and how they point to one another respectively:

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10040000	0x10040004	0x00000003	0x10040014	0x0000000a	0x00000000	0x00000005	0x1004000c	0x10040028
0x10040020	0x00000014	0x00000000	0x00000010	0x10040020	0x00000000	0x00000000	0x00000000	0x00000000
0x10040040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

As seen above, HEAD of A points to address 0x10040004. At this first node, key as we see is 3. Similarly, HEAD of B points to address 0x10040028. At this first node, key as we see is 16(0x10). It will be a helpful exercise for you to see how each node connects to its next node for A and B. The above image is provided as a separate file in the lab 4 folder (*heap\_trace1.png*) for better viewability.

Now, the testbench will run *mergeLinkedLists* function on A and B. The lists A and B will be merged, maintaining the overall ascending order of keys and the resulting merged list will become the new A list. So, before merging, we had,

A=3->5-> 10

B=16->20

After merging through *mergeLinkedLists*, we have,  
A=3->5-> 10->16->20  
B=16->20

If we were to inspect the contents of heap memory now, we would get the following:

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10040000	0x10040004	0x00000003	0x10040014	0x0000000a	0x10040030	0x00000005	0x1004000c	0x10040028
0x10040020	0x00000014	0x00000000	0x00000010	0x10040020	0x00000010	0x10040038	0x00000014	0x00000000
0x10040040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10040060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10040080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

It will be a helpful exercise for you to see how each node connects to its next node for A and B. The above image is provided as a separate file in the lab 4 folder (*heap\_trace2.png*) for better viewability.

## Files to be submitted to your Lab4 Gradescope submission

- *newNode.asm*
- *print\_list.asm*
- *insertSorted.asm*
- *mergeLinkedLists.asm*
- *lab4\_testbench.asm* (**DO NOT EDIT/MODIFY THIS FILE**)

## A Note About Academic Integrity

This is the lab assignment where most students continue to get flagged for cheating. Please review the pamphlet on [Academic Dishonesty](#) and look at the examples in the first lecture for acceptable and unacceptable collaboration.

***You should be doing this assignment completely all by yourself!***

## Grading Rubric (100 points total)

The following rubric applies provided you have fulfilled all criteria in **Minimum Submission Requirements**. Failing any criteria listed in that section would result in an automatic grade of zero **which cannot be eligible for applying for a regrade request**.

**20 pt** *lab4\_testbench.asm* **assembles** without errors (*so even if you submit lab4\_testbench.asm with all the other required .asm files COMPLETELY unmodified by you, you would still get 20 pts.!*)

**80 pt** output in file *lab3\_output.txt* matches the specification:

**10 pt** *newNode.asm* works

**10 pt** *print\_list.asm* works

**30 pt** *insertSorted.asm* works

**30 pt** *mergeLinkedLists.asm* works

## Legal Notice

All course materials and relevant files located in the Lab4 folder in the course Google Drive must not be shared by the students outside of the course curriculum on any type of public domain site or for financial gain. Thus, if any of the Lab3 documents is found in any type of publicly available site (e.g., GitHub, stack Exchange), or for monetary gain (e.g., Chegg), then the original poster will be cited for misusing CSE12 course-based content and will be reported to UCSC for academic dishonesty.

In the case of sites such as Chegg.com, we have been able to locate course material shared by a previous quarter student. Chegg cooperated with us by providing the student's contact details, which was sufficient proof of the student's misconduct leading to an automatic failing grade in the course.