Tutorials References Exercises Sign In

SS JAVASCRIPT SQL PYTHON JAVA PHP HOW TO W3.CSS C C

DSA Linked Lists Operations

❮ Previous Next ❯

Linked List Operations

Basic things we can do with linked lists are:

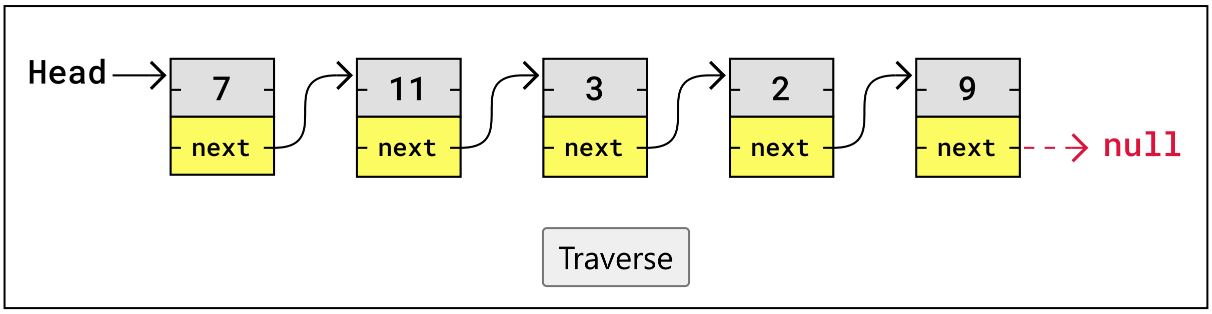
1. Traversal
2. Remove a node
3. Insert a node
4. Sort

For simplicity, singly linked lists will be used to explain these operations below.

Traversal of a Linked List

Traversing a linked list means to go through the linked list by following the links from one node to the next.

Traversal of linked lists is typically done to search for a specific node, and read or modify the node's content, remove the node, or insert a node right before or after that node.



The code below prints out the node values as it traverses along the linked list, in the same way as the animation above.

Example

Traversal of a singly linked list in Python:

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def traverseAndPrint(head):
    currentNode = head
    while currentNode:
        print(currentNode.data, end=" -> ")
        currentNode = currentNode.next
    print("null")

node1 = Node(7)
node2 = Node(11)
node3 = Node(3)
node4 = Node(2)
node5 = Node(9)

node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node5
```

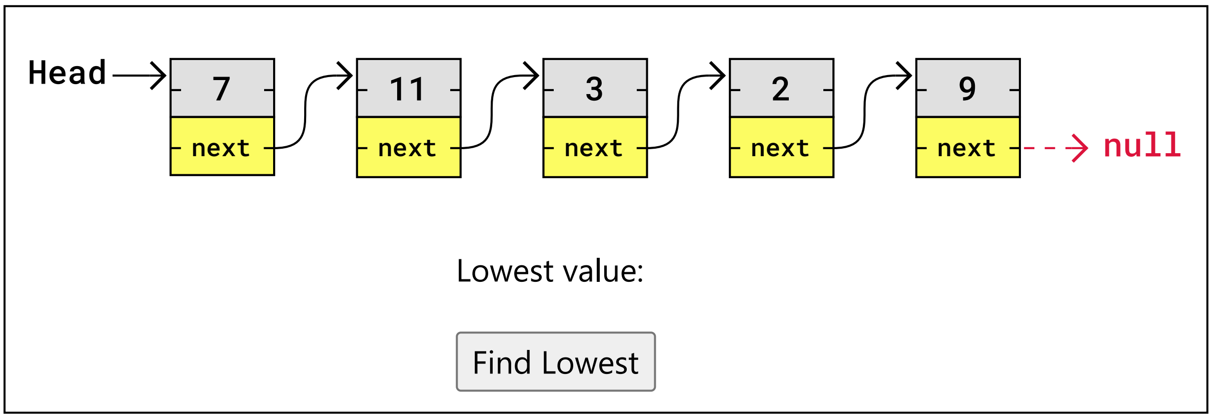
Try it Yourself »

Find The Lowest Value in a Linked List

Let's find the lowest value in a singly linked list by traversing it and checking each value.

Finding the lowest value in a linked list is very similar to how we found the lowest value in an array, except that we need to follow the next link to get to the next node.

This is how finding the lowest value in a linked list works in principle:



To find the lowest value we need to traverse the list like in the previous code. But in addition to traversing the list, we must also update the current lowest value when we find a node with a lower value.

In the code below, the algorithm to find the lowest value is moved into a function called `findLowestValue`.

Example

Finding the lowest value in a singly linked list in Python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        currentNode = currentNode.next
    return minValue

node1 = Node(7)
node2 = Node(11)
node3 = Node(3)
node4 = Node(2)
node5 = Node(9)

node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node5

print("The lowest value in the linked list is:", findLowestValue(node1))
```

The marked lines above is the core of the algorithm. The initial lowest value is set to be the value of the first node. Then, if a lower value is found, the lowest value variable is udated.

Try it Yourself »

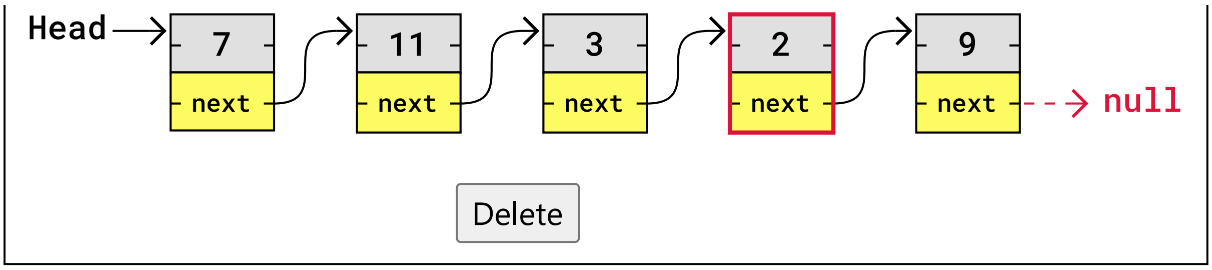
Delete a Node in a Linked List

In this case we have the link (or pointer or address) to a node that we want to delete.

It is important to connect the nodes on each side of the node before deleting it, so that the linked list is not broken.

So before deleting the node, we need to get the next pointer from the previous node, and connect the previous node to the new next node before deleting the node in between.

In a singly linked list, like we have here, to get the next pointer from the previous node we actually need traverse the list from the start, because there is no way to go backwards from the node we want to delete.



Also, it is a good idea to first connect next pointer to the node after the node we want to delete, before we delete it. This is to avoid a 'dangling' pointer, a pointer that points to nothing, even if it is just for a brief moment.

In the code below, the algorithm to delete a node is moved into a function called `deleteSpecificNode`.

Example

Deleting a specific node in a singly linked list in Python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def traverseAndPrint(head):
    currentNode = head
    while currentNode:
        print(currentNode.data, end=" -> ")
        currentNode = currentNode.next
    print("null")
    return head

node1 = Node(7)
node2 = Node(11)
node3 = Node(3)
node4 = Node(2)
node5 = Node(9)

node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node5

print("Before deletion:")
traverseAndPrint(node1)

# Delete node4
node1 = deleteSpecificNode(node1, node4)

print("\nAfter deletion:")
traverseAndPrint(node1)
```

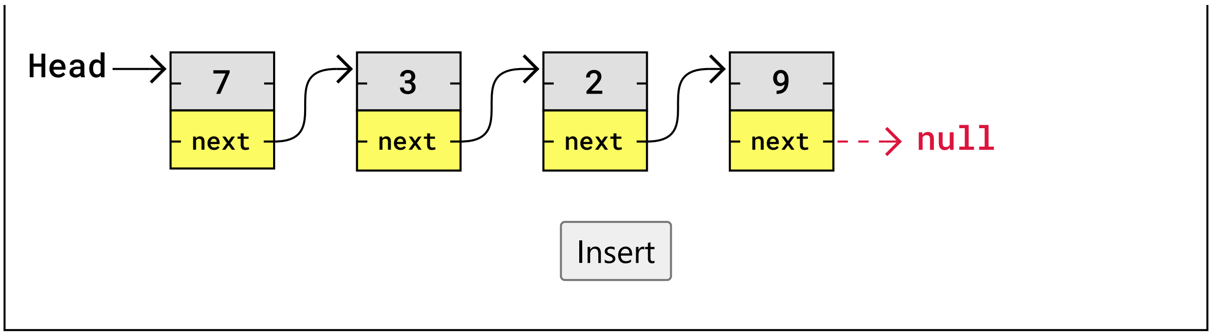
Try it Yourself »

In the `deleteSpecificNode` function above, the return value is the new head of the linked list. So for example, if the node to be deleted is the first node, the new head returned will be the next node.

Insert a Node in a Linked List

Inserting a node into a linked list is very similar to deleting a node, because in both cases we need to take care of the next pointers to make sure we do not break the linked list.

To insert a node in a linked list we first need to create the node, and then at the position where we insert it, we need to adjust the pointers so that the previous node points to the



1. New node is created
2. Node 1 is linked to new node
3. New node is linked to next node

Example

Inserting a node in a singly linked list in Python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def traverseAndPrint(head):
    currentNode = head
    while currentNode:
        print(currentNode.data, end=" -> ")
        currentNode = currentNode.next
    print("null")
```

```
node1 = Node(7)
node2 = Node(3)
node3 = Node(2)
node4 = Node(9)

node1.next = node2
node2.next = node3
node3.next = node4

print("Original list:")
traverseAndPrint(node1)

# Insert a new node with value 97 at position 2
newNode = Node(97)
node1 = insertNodeAtPosition(node1, newNode, 2)

print("\nAfter insertion:")
traverseAndPrint(node1)
```

Try it Yourself »

In the `insertNodeAtPosition` function above, the return value is the new head of the linked list. So for example, if the node is inserted at the start of the linked list, the new head returned will be the new node.

Other Linked Lists Operations

We have only covered three basic linked list operations above: traversal (or search), node deletion, and node insertion.

There are a lot of other operations that could be done with linked lists, like sorting for example.

Previously in the tutorial we have covered many sorting algorithms, and we could do many of these sorting algorithms on linked lists as well. Let's take selection sort for example. In selection sort we find the lowest value, remove it, and insert it at the beginning. We could do

Note: We cannot sort linked lists with sorting algorithms like Counting Sort, Radix Sort or Quicksort because they use indexes to modify array elements directly based on their position.

Linked Lists vs Arrays

These are some key linked list properties, compared to arrays:

- Linked lists are not allocated to a fixed size in memory like arrays are, so linked lists do not require to move the whole list into a larger memory space when the fixed memory space fills up, like arrays must.
- Linked list nodes are not laid out one right after the other in memory (contiguously), so linked list nodes do not have to be shifted up or down in memory when nodes are inserted or deleted.
- Linked list nodes require more memory to store one or more links to other nodes. Array elements do not require that much memory, because array elements do not contain links to other elements.
- Linked list operations are usually harder to program and require more lines than similar array operations, because programming languages have better built in support for arrays.
- We must traverse a linked list to find a node at a specific position, but with arrays we can access an element directly by writing `myArray[5]`.

Note: When using arrays in programming languages like Java or Python, even though we do not need to write code to handle when an array fills up its memory space, and we do not have to shift elements up or down in memory when an element is removed or inserted, these things still happen in the background and can cause problems in time critical applications.

Time Complexity of Linked Lists Operations

operations needed by the algorithm based on a large set of data n, and does not tell us the exact time a specific implementation of an algorithm takes.

This means that even though linear search is said to have the same time complexity for arrays as for linked list: $O(n)$, it does not mean they take the same amount of time. The exact time it takes for an algorithm to run depends on programming language, computer hardware, differences in time needed for operations on arrays vs linked lists, and many other things as well.

Linear search for linked lists works the same as for arrays. A list of unsorted values are traversed from the head node until the node with the specific value is found. Time complexity is $O(n)$.

Binary search is not possible for linked lists because the algorithm is based on jumping directly to different array elements, and that is not possible with linked lists.

Sorting algorithms have the same time complexities as for arrays, and these are explained earlier in this tutorial. But remember, sorting algorithms that are based on directly accessing an array element based on an index, do not work on linked lists.

DSA Exercises

Test Yourself With Exercises

Exercise:

Complete the code for the Linked List traversal function.

```
def traverseAndPrint(head):
    currentNode = 
    while currentNode:
        print(currentNode.data, end=" -> ")
```

Submit Answer »

Start the Exercise

❮ Previous Sign in to track progress Next ❯

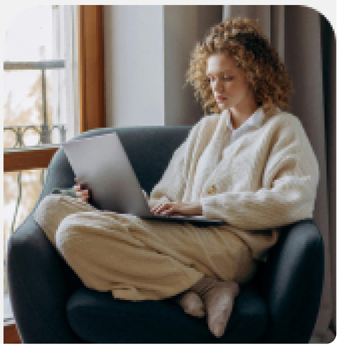
Get Certified!

Document your skills with all of W3Schools Certificates

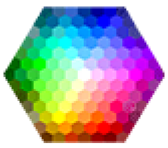
~~$1,995~~

$499

Save 75%



COLOR PICKER



PLUS SPACES

GET CERTIFIED FOR TEACHERS

FOR BUSINESS CONTACT US

Top Tutorials

HTML Tutorial
CSS Tutorial
JavaScript Tutorial
How To Tutorial
SQL Tutorial
Python Tutorial
W3.CSS Tutorial
Bootstrap Tutorial
PHP Tutorial
Java Tutorial
C++ Tutorial
jQuery Tutorial

Top References

HTML Reference
CSS Reference
JavaScript Reference
SQL Reference
Python Reference
W3.CSS Reference
Bootstrap Reference
PHP Reference
HTML Colors
Java Reference
AngularJS Reference
jQuery Reference

Top Examples

HTML Examples
CSS Examples
JavaScript Examples
How To Examples
SQL Examples
Python Examples
W3.CSS Examples
Bootstrap Examples
PHP Examples

Get Certified

HTML Certificate
CSS Certificate
JavaScript Certificate
Front End Certificate
SQL Certificate
Python Certificate
PHP Certificate
jQuery Certificate
Java Certificate

Tutorials ▾ References ▾ Exercises ▾ Sign In

SS JAVASCRIPT SQL PYTHON JAVA PHP HOW TO W3.CSS C C

FORUM ABOUT ACADEMY

W3Schools is optimized for learning and training. Examples might be simplified to improve reading and learning.
Tutorials, references, and examples are constantly reviewed to avoid errors, but we cannot warrant full correctness
of all content. While using W3Schools, you agree to have read and accepted our terms of use, cookies and privacy policy.

Copyright 1999-2026 by Refsnes Data. All Rights Reserved. W3Schools is Powered by W3.CSS.