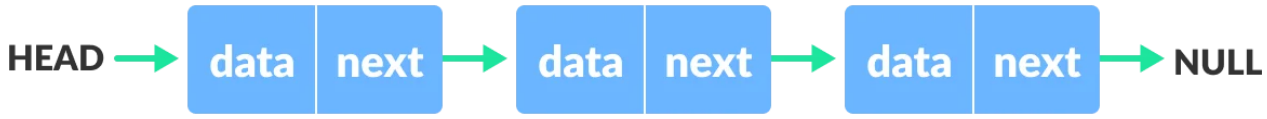
A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the data and the address of the next node. For example,



You have to start somewhere, so we give the address of the first node a special name called HEAD. Also, the last node in the linked list can be identified because its next portion points to NULL.

Linked lists can be of multiple types: singly, doubly, and circular linked list. In this lesson, we will focus on the singly linked list.

```
Note: You might have played the game Treasure Hunt, where each clue
includes the information about the next clue. That is how the linked list
operates.
```

Representation of Linked List

Let's see how each node of the linked list is represented. Each node consists:

A data item

An address of another node

We wrap both the data item and the next node reference in a struct as:

```
struct node
{
  int data;
  struct node *next;
};
```

Understanding the structure of a linked list node is the key to having a grasp on it.

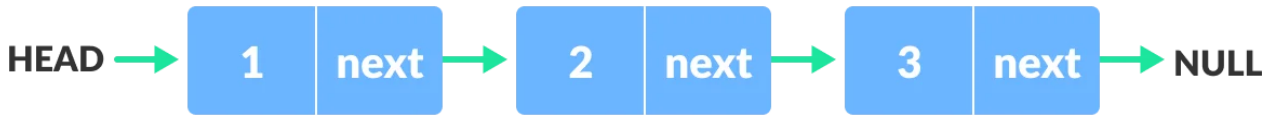
Each struct node has a data item and a pointer to another struct node. Let us create a simple Linked List with three items to understand how this works.

```
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;
```

```
/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data=3;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
```

If you didn't understand any of the lines above, all you need is a refresher on pointers and structs.

In just a few steps, we have created a simple linked list with three nodes.



The power of a linked list comes from the ability to break the chain and rejoin it. E.g. if you wanted to put an element 4 between 1 and 2, the steps would be:

```
Create a new struct node and allocate memory to it.

Add its data value as 4.

Point its next pointer to the struct node containing 2 as the data value.

Change the next pointer of "1" to the node we just created.
```

Doing something similar in an array would have required shifting the positions of all the subsequent elements.

Linked List Utility

Lists are one of the most popular and efficient data structures, with implementation in every programming language like C, C++, Python, Java, and C#.

Apart from that, linked lists are a great way to learn how pointers work. By practicing how to manipulate linked lists, you can prepare yourself to learn more advanced data structures like graphs and trees.

Linked List Implementations in Python, Java, C, and C++ Examples

```
// Linked list implementation in C++
```

```
#include <bits/stdc++.h>
#include <iostream>
using namespace std;

// Creating a node
class Node {
   public:
  int value;
  Node* next;
};

int main() {
  Node* head;
  Node* one = NULL;
  Node* two = NULL;
  Node* three = NULL;

  // allocate 3 nodes in the heap
  one = new Node();
  two = new Node();
  three = new Node();

  // Assign value values
  one->value = 1;
  two->value = 2;
  three->value = 3;

  // Connect nodes
  one->next = two;
  two->next = three;
  three->next = NULL;

  // print the linked list value
  head = one;
  while (head != NULL) {
    cout << head->value;
    head = head->next;
  }
}
```

```
// Linked list implementation in Java
```

```
class LinkedList {
  // Creating a node
  Node head;

  static class Node {
    int value;
    Node next;

    Node(int d) {
      value = d;
      next = null;
    }
  }

  public static void main(String[] args) {
    LinkedList linkedList = new LinkedList();

    // Assign value values
    linkedList.head = new Node(1);
    Node second = new Node(2);
    Node third = new Node(3);

    // Connect nodess
    linkedList.head.next = second;
    second.next = third;

    // printing node-value
    while (linkedList.head != null) {
      System.out.print(linkedList.head.value + " ");
      linkedList.head = linkedList.head.next;
    }
  }
}
```

Linked List Complexity

Time Complexity

```
                Worst case  Average Case
Search              O(n)       O(n)
Insert              O(1)       O(1)
Deletion            O(1)       O(1)

Space Complexity: O(n)
```

Linked List Applications

```
Dynamic memory allocation

Implemented in stack and queue

In undo functionality of softwares

Hash tables, Graphs
```