Dylan Chen

Sep 10, 2021 · 6 min read · Member-only · 3:58

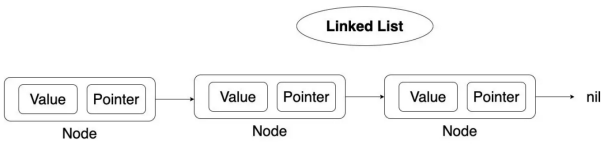
Swift 5: Linked Lists for Beginners

Photo by Becca Tapert on Unsplash

Hi iOS developers,

Data Structures are containers that are used to organize and store data in the computer so that we can efficiently perform operations. They are the most fundamental components of programming. The best known and most used data structures are Array, Stacks, and Queues, among others. Today, I'm going to talk about another useful Data Structure — Linked List. Unfortunately, Swift doesn't provide built-in Linked List structures, so we need to build our own.

What are Singly Linked Lists?



A Linked List is a list of linked nodes. A node is a single element that contains a generic value and a reference/pointer to the next node. There are a few types of Linked Lists, such as:

- **Singly Linked Lists** — Each node only has a reference/pointer to the next node. The operations can only travel in one direction.
- **Doubly Linked Lists** — Each node has two references/pointers, one to the next node and one to the previous node. Operations can travel in both forward and backward directions.
- **Circular Linked Lists** — The next reference/pointer of the last node points to the first node, and/or the previous reference/pointer of the first node points to the last node.

Today we are going to focus on **Singly Linked Lists.**

. . .

Node

`Node` must be defined as a class. It needs to be a reference type, one that won't work in a struct. The `Node` class has two properties:

- value: a generic data type that stores the actual value of the node.
- next: a reference/pointer that points to the next node.

```
class Node<T> {

    var value: T
    var next: Node<T>?

    init(value: T, next: Node<T>? = nil) {
        self.value = value
        self.next = next
    }
}
```

Node.swift hosted with ❤ by GitHub view raw

Linked List

Unlike `Node` , a Linked List is a value type. It's defined as a `struct` . By default, a Linked List has three basic properties:

- `head` : the first node of the list.
- `tail` : the last node of the list.
- `isEmpty` : whether or not the list is empty.

Of course, you can add additional properties based on your needs, such as:

- `count` : the number of nodes in the list.
- `description` : text that describes the list.

```
struct LinkedList<T> {

  var head: Node<T>?
  var tail: Node<T>?

  var isEmpty: Bool { head == nil }

  init() {}
}
```

Linked List Properties.swift hosted with ❤ by GitHub view raw

Unlike `Stacks` and `Queues` , a Linked List doesn't contain a collection of data that you can call and use directly. All nodes in the list must be linked to the next available node if one exists. If we mess up a node in the middle of the list, the list is going to be broken. So, read carefully.

. . .

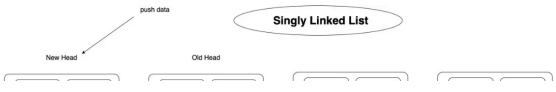
`Push`

By pushing data to a list, we are adding data in the front of the list. This means the current head will be replaced with the new node, and the new node will become the head of the list.

```
struct LinkedList<T> {
  ...
  mutating func push(_ value: T) {
      head = Node(value: value, next: head)

      if tail == nil {
          tail = head
      }
  }
}
```

Linked List Push.swift hosted with ❤ by GitHub view raw

By calling `push(_ value:T)` , we create a new node with the `value` , and have the new node pointed to the old head. We then replace the head with the new node, so the old head becomes the second node of the list.



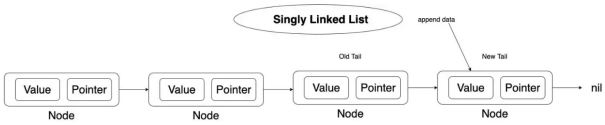
Append

Similar to `push` , by appending data to a list, we add data to the end of the list. This means the current tail will be replaced with the new node, the new node will become the new tail.

```
struct LinkedList<T> {
  ...
  mutating func append(_ value: T) {
    let node = Node(value: value)

    tail?.next = node
    tail = node
  }
}
```

Linked List Append.swift hosted with ❤ by GitHub view raw

By calling `append(_ value:T)` , we create a new node and have the old tail pointed to the new node. Lastly, we replace the old tail with the new node, so the old tail becomes the second last node of the list. The new node will become the new tail.



Node At

Indices don't subscribe to Linked Lists, so we are not able to read data the collection in the same way as arrays: `array[0]` x.

```
struct LinkedList<T> {
  ...
  func node(at index: Int) -> Node<T>? {
      var currentIndex = 0
      var currentNode = head

      while currentNode != nil && currentIndex < index {
          currentNode = currentNode?.next
          currentIndex += 1
      }

      return currentNode
  }
}
```

Linked List nodeAt.swift hosted with ❤ by GitHub view raw

To retrieve data from an index, we need to use a loop.

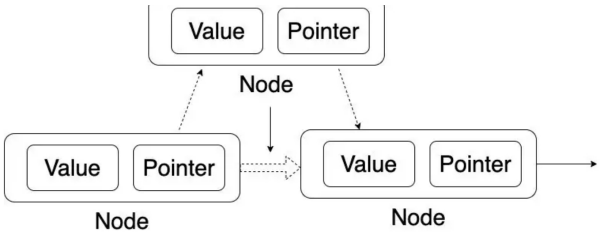
Insert

As I said previously, indices don't subscribe to Linked Lists. They have no idea about positioning. All they know is which node is linked to which. In order to tell the Linked List to insert a node at a specific position, we need to find the node that's linked to the position. We need to use the **`node(at index: Int) -> Node<T>?`** function above.


```
struct LinkedList<T> {
  ...
  func insert(_ value: T, after index: Int) {
    guard let node = node(at: index) else { return }

    node.next = Node(value: value, next: node.next)
  }
}
```

Linked List insert.swift hosted with ❤ by GitHub view raw

First, we must find the node that's at the given position. We have `next` pointed to the new node, and the new node pointed to the original next node.



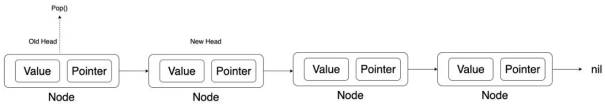
Pop

By popping data, we remove the head from the list, so the second node will then become the head.

```
struct LinkedList<T> {
  ...
  mutating func pop() -> T? {
    defer {
      head = head?.next

      if isEmpty {
        tail = nil
      }
    }

    return head?.value
  }
}
```

Linked List pop.swift hosted with ❤ by GitHub view raw

We return the value of the old head and replace the old head with the next node. So the second node becomes the head.



Remove Last

Similar to `pop()`, this removes the tail of the list, so the second last node will become the tail.

```
struct LinkedList<T> {
  ...
  mutating func removeLast() -> T? {
    guard let head = head else { return nil }
    guard let head.next != nil else { return pop() }

    var previousNode = head
    var currentNode = head

    while let next = currentNode.next {
      previousNode = currentNode
      currentNode = next
    }

    previousNode.next = nil
    tail = previousNode

    return currentNode.value
  }
}
```

Linked List removeLast.swift hosted with ❤ by GitHub view raw

But unlike `pop()`, removing the tail is a little bit more complicated since the tail doesn't know which is the previous node. We need to iterate through the list to find the node before the tail and make it the tail.

- If head is `nil`, meaning that the list is empty, we have nothing to remove, then return `nil`.
- If `head.next` is `nil`, meaning that there is only one node in the list, then remove the head.
- Iterate through the list, find the node before the tail, and make it the tail.

Remove After

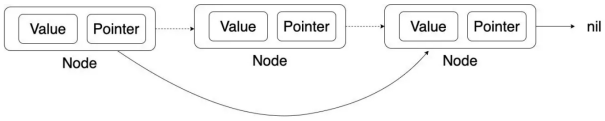
Similar to `insert(_ value: T, after index: Int)`, we need to find the linked node since the list has no idea about positioning

node since the list has no idea about positioning.

```swift
struct LinkedList<T> {
  ...
  mutating func remove(after index: Int) -> T? {
    guard let node = node(at: index) else { return nil }

    defer {
      if node.next === tail {
        tail = node
      }

      node.next = node.next?.next
    }

    return node.next?.value
  }
}
```

Linked List Remove After.swift hosted with ❤ by GitHub — view raw

It's kind of tricky to remove a node at a given index, so we basically just skip one node, and point to the node after that one.



These are the basic properties and functions that a Singly Linked List normally has. Of course, you can add add-ons on top of these.

. . .

To use Singly Linked Lists

```swift
var list = LinkedList<Int>()

list.push(10)
list.push(22)

print(list) // 22 -> 10

list.append(5)
list.append(2)

print(list) // 22 -> 10 -> 5 -> 2

list.insert(15, after: 1)

print(list) // 22 -> 10 -> 15 -> 5 -> 2

list.pop()

print(list) // 10 -> 15 -> 5 -> 2

list.removeLast()

print(list) // 10 -> 15 -> 5

list.remove(after: 0)

print(list) // 10 -> 5
```

Linked List Usecase.swift hosted with ❤ by GitHub — view raw

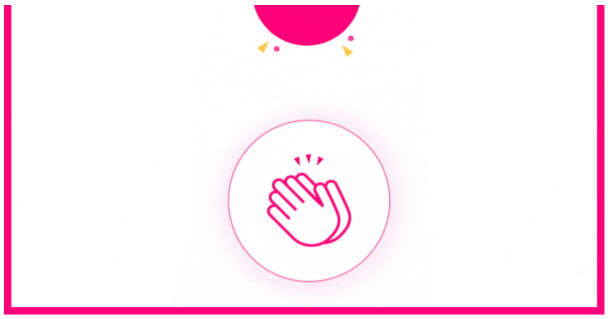
FAQ:

Q: Why do we always insert and remove nodes after a given index? Why can we do `remove(at index: Int)` and `insert(at index: Int)` as arrays do?

A: Nodes in a Singly Linked List must be linked to another in one single direction. To insert or remove nodes, we must tell the previous node to skip the next node and point to the node after that one. That's why we must always find the previous nodes, and remove or insert nodes after them.

Please clap if you enjoyed this story. Follow me. I'll see you in subsequent stories :)

1:05 5:04 1.0×
Get Speechify Chrome Extension
1:05 5:04 1.0×
Get Speechify Chrome Extension



Medium Claps — Made in Flinto by Thuy Gia Nguyen on Dribbble

Become a Medium member, read more great stories and grow your career.

YOU MAY ALSO BE INTERESTED IN:

Week 5: Stack and Queue in Swift

The role for Stacks is LIFO(Last in, first out), and the role for Queues is FIFO(First in, first out).

medium.com

Week 7: Handy Xcode Features to Save Us Time

Building apps is very time-consuming.

medium.com

100 1

Enjoy the read? Reward the writer.[Beta]

Your tip will go to Dylan Chen through a third-party platform of their choice, letting them know you appreciate their story.

Give a tip

Sign up for Geek Culture Hits

By Geek Culture

Subscribe to receive top 10 most read stories of Geek Culture — delivered straight into your inbox, once a week. Take a look.

Emails will be sent to ahmad.atef.ali.ahmad@gmail.com. Not you?

Get this newsletter

More from Geek Culture

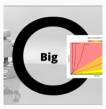
Follow

A new tech publication by Start it up (https://medium.com/swlh).

Samuel Martins · Sep 10, 2021

Big O Notation Explained for Beginners

Learn Data Structures and Algorithms by Understanding the Fundamentals — One of the most dreaded topics in computer science is the Big O notation, especially when you are first learning data...



Technology 6 min read

Share your ideas with millions of readers. Write on Medium

Apache ShardingSphere · Sep 10, 2021

AutoTable: Your Butler-Like Sharding Configuration Tool

Summary In the previous article “An Introduction to DistSQL” written by Haoran Meng, the Apache ShardingSphere Committer shared the...

Open Source 5 min read

Yurii K · Sep 10, 2021

Authentication flow in Angular

OpenId connect — Every website has closed resources. In order to get those resources, users must have the rights to do it. That means our website must have an authentication flow. In this article, we impleme…

Angular 3 min read

Christianlauer · Sep 10, 2021 Member-only

How to Determine the ROI of Cloud Computing

Using useful Metrics for calculating the Return of Investment — Every IT manager is concerned with how to make the case for the cloud and justify the costs. But also whether the operation is really more cost-…

It Management 3 min read

Dr Stuart Woolley · Sep 10, 2021 Member-only

How To Not Answer Software Engineering Interview Questions

Your working out is way more important than the actual answer. Rule #1 If there's one thing I remember[1] from 'O' level maths at secondary…

Software Engineering 6 min read

Read more from Geek Culture