|  |  |
| --- | --- |
| Untitled-1.jpg | Coding Dojo Academy  **Algorithms III** (Pilot)  Singly & Doubly Linked List Data Structures  Mar 5 - Mar 16 M/W/F @ 9:00 am PST  Syllabus |

*Algorithms III is Coding Dojo Academy’s live lecture course covering Algorithms (Singly and Doubly linked list data structures) This course will use the JavaScript language. We use JavaScript because it’s easy to setup and get started, and it’s one of the most utilized programming language on the web. The below links are essential tools utilized during this course, all class communication is handled through the Slack channel, be sure to join your classmates in chat, where you may ask and answer questions.*

***Coding Dojo Academy Rules***

*Be Humble, and respectful*

*Be Present, on time, attentive and responsive*

*Strength through Struggle*

*Be Supportive to your peers, help each other*

|  |  |
| --- | --- |
| ***Instructor:*** *Speros Misirlakis*  *Head of Curriculum and Dojo Academy*  ***Slack:*** [*@Speros - Coding Dojo*](https://algorithms-iii-201803.slack.com/messages/D9JJ1KU7M)  ***Email:***[*speros@codingdojo.com*](mailto:speros@codingdojo.com)  [*LinkedIn*](https://www.linkedin.com/in/speros-misirlakis-38474328/) | [*Github Link*](https://github.com/keephopealive) | ***Moderator****: Jason Franz*  *Online Instructor*  ***Slack****: @Jason - Coding Dojo*  *Email:* [*jfranz@codingdojo.com*](mailto:jfranz@codingdojo.com)  [*Github Link*](https://github.com/jfrazx) |

* [**Algorithm App** *(click this link to visit our platform)*](http://algorithm.codingdojo.com/)**:***Algorithm App is Coding Dojo’s algorithm platform designed to help new developers learn the very fundamentals of algorithms using JavaScript.*
* [***Zoom*** *(click this link to download Zoom)*](https://support.zoom.us/hc/en-us/articles/201362233-Where-Do-I-Download-The-Latest-Version-): *Zoom is the software required for Coding Dojo Academy’s live lectures.*
* [**Webinar** *(click this link to join our live lecture)*](https://codingdojo.zoom.us/j/283344688?pwd=mhv9A0IpinY)*: Use this link to join live lectures at the scheduled date/times. Password: ‘****academy’*** *(without quotes).*
* [**Slack** *(click this link to join our chatroom)*](https://join.slack.com/t/algorithms-iii-201803/shared_invite/enQtMzI0MTYwMjg4NDAxLTEzMjE5MzJmYzQ2MzNlOTQzNGU5ZThiM2EyN2M3N2E1YmY4MmVlNzFmZTQ1MjFlZGI1MzQyNmFiYWU1MGEyZjI)***:*** *Slack is the chatroom service used for Coding Dojo Academy’s courses and the primary method of communication.*
* [**JS Bin** (click this link to practice Algorithms with JavaScript)](https://jsbin.com/?js,console): *JS Bin is a JavaScript Playground tool which helps us experiment with javascript code on the browser, without the need of creating our own local file, and launching that file on our browser.*
* [**Homework Submission** *(click this link to view our class G-Drive)*](https://drive.google.com/drive/u/2/folders/1gtbCgbXm8zvC9NdlV7g5tjn-kO4WCCTP): *We will be using this Google Drive to upload homework, to receive an invite to the G-Drive please join the Slack channel and private message* [*@Speros*](https://algorithms-iii-201803.slack.com/messages/D9JJ1KU7M) *your gmail address. Once you have access, please create a folder with your full first name and initial last name:*

*e.g. Tom Jones = Tom J || Diana Mills = Diana M*

*Please be respectful and keep the folder pattern as shown above. Do not use usernames, please use your real first name and real last name initial. Use your own folder to upload the homework files into.*

*Warning: Any malicious use of the google drive will get you ejected from the Academy programs and blacklisted from all Coding Dojo services, indefinitely.*

Schedule

|  |  |  |
| --- | --- | --- |
| **Week 1 Monday**  *Recorded Lecture:* [*https://vimeo.com/258667800/a5108bab9c*](https://vimeo.com/258667800/a5108bab9c)   * Introductions to Singly Linked Lists * Nodes, Val, Next * Node Object Constructor * [SList Reading Material](#4ljhjtb6uy8y) * [Assignments W1 Monday](#w0nd93r4lmqg)   (due Sunday 11th 11:59pm local)   * + - **addFront**(Node) => Node     - **removeFront**(Node) => Node     - **contains**(Node, Number) => Boolean     - **front**(Node) => Node | **Week 1 Wednesday**  Recorded Lecture: <https://vimeo.com/259046276/dcb96f1fc5>   * Review Monday’s algorithms and open Q&A * List Object, Head, other * List Object Constructor * [Assignments W1 Wednesday](#ui6squiphc4m)   (due Sunday 11th 11:59pm local)   * **length**(list) => Number * **average**(list) => Number * **min**(list) => Number * **max**(list) => Number * **display**(list) => None * BONUS:   + prependVal(list, val, afterVal)   + removeVal(list, value)   + appendVal(list, value, afterVal) | **Week 1 Friday**  Recorded Lecture: <https://vimeo.com/259397919/4c77c2a546>   * Review and open Q&A * [**OPTIONAL** Assignments W1 Friday](#wwlfkr802b2u)   + **splitOnVal**()   + **partition**()   + **deleteGivenNode**() |
| **Week 2 Monday**  *Recorded Lecture:* [*https://vimeo.com/259748569/1fc66785a8*](https://vimeo.com/259748569/1fc66785a8)   * Introductions to Doubly Linked Lists * [DList Reading Material](#jxdm0wgaar7v) * [Assignments W2 Monday](#kix.fgtqzqfzh72) * (due Sunday 18th 11:59pm local) * **Prepend Value** * **Kth To Last Value** * **Is Valid dList** * **Palindrome** * **Loop Start** | **Week 2 Wednesday**  *Recorded Lecture:* [*https://vimeo.com/260108852/1211b96259*](https://vimeo.com/260108852/1211b96259)  *California follows daylight savings time which caused a time change on March 11th. For the next two lecture, we will start at 9am PST (you may be affected by this), however we will run the lecture for 1.5 hours to accommodate anyone that may not be able to make the adjusted time and offer them time for Q&A.*   * Review Monday’s algorithms and open Q&A * [Assignments W2 Wednesday](#oh3xsbusekl5) | **Week 2 Friday**   * Review Object Constructors (Class) * Review and open Q&A * Prep for exam, green highlighted algorithms will likely be similar to exam questions. * (no new algorithms) |
| **Week 2 Saturday**   * Singly and Doubly Linked List Algorithm Exam   + 10:00 am PST (exam #1)   + 7:00 pm PST (exam #2) * Email exam **.txt** to * [Academy@codingdojo.com](mailto:Academy@codingdojo.com) |  |  |

### 

# **Linked Lists**

This chapter we explore linked lists, a data structure used widely in lower layers such as backends, frameworks, runtimes or operating systems. You should be familiar with object oriented ideas, including the *reference* concept: not a local copy of a value, but a pointer to the value in shared memory.

How does your operating system keeps track of the files in a directory? Modern systems do not do this with an array. They use a data structure called a *linked list*. Linked lists are easily reordered and well-suited for large data collections because (unlike arrays) they store data in small pieces of memory that “fit in the holes” between variables, rather than requiring a large chunk of contiguous memory. Linked lists are the first data structure we discuss as an *object* and introduce us to the concept of a *reference*.

A **class definition** is like a blueprint of a complex machine, from which many copies can be made. Actually constructing a machine is a separate step. Likewise, *declaring* a class merely informs us of that blueprint; actual objects must be individually constructed. In JavaScript, class declarations take the form of functions called *object constructors* – when called, they create an **object** for the caller. An object is an instance of the class, brought to life, just like a physical copy of the ideas in the blueprint.

Not all machines are complex, and not all objects are complicated. However, code can add and remove attributes of objects on-the-fly, so this makes them different than a boolean or number which always occupies the same amount of memory space. Why does this matter? Well, if you have debugged your JavaScript code in the browser, you may understand the idea of a call stack. This is the series of function calls that led the computer to where it is right now. Whenever the currently running function returns, the JavaScript runtime will look to the call stack to help it “remember” which function it came from, as well as the state of all its local variables at the time when it called into another function. The runtime stores local variables in the call stack while changing the execution to another function. Setting aside call stack space for booleans and numbers is easy -- regardless of value, numbers occupy a 64-bit chunk of memory. However, objects are tricky: the JavaScript runtime cannot determine *a priori* how much space to set aside for your objects. So how can it quickly construct a call stack?

The answer is that objects are created using a common chunk of memory set aside for variable-sized allocations. This memory is called the *heap*, and it is used for any unpredictable memory needs. When the system looks at your ‘blueprint’ and constructs a ‘machine’ corresponding to those plans, it goes to the heap and sets aside space for all that object’s attributes and functions. If the object needs more space, it expands into adjacent heap memory. During normal operation, the heap is wide-open for large and small allocations. The call stack is apartment space in a high-rise tower; the heap is Montana.

When you create an object and store it in a local **var**, the system doesn’t put the object in that memory slot the way it does for a number or a boolean. It puts a *reference* to that heap location into your local **var**. References (called pointers) are fixed-size, so this enables the runtime do its stack magic. A pointer represents an object’s location in memory, but you can think of it as an object’s contact info: its email address. True to its name, a pointer *points* to where the object is found. If you have information to retrieve from (or store to) an object, you “go there” by dereferencing that pointer, followed by the attribute you want within the object. This could look like **myProject.name** or **myQuizzes[3]** or even **getAverage(myArr)**. Yes, arrays, strings and even functions are objects – dereferenced by **.** or **[** or **(** .

[Back to schedule.](#843uwrr9jjda)

**Week 1 Monday**

Over the chapter’s course, we’ll coalesce a considerable collection of concepts to contemplate. Some or all of these will be used in this chapter’s challenges.

*classes* and *objects object constructors local vars vs. heap allocations pointers*

*reference* vs. *value private* vs. *public ===* vs. *== push( ) & pop( )*

|  |  |
| --- | --- |
| To the left is a definition of a **node** object. A node object simply holds a value, as well as a *pointer* that links it to the next node in the sequence, if there is one. A sequence of node objects is called a *linked list*. | **function listNode(value)**  **{**  **this.val = value;**  **this.next = null;**  **}** |

|  |  |
| --- | --- |
| **addFront** Given a pointer to the **first node** in a list, and a **value**, create a new node, connect it to the head of the list, and return a pointer to the list’s new head node.  **addFront(Node, Number) => Node** **contains** Given a pointer to a **listNode** and a **value**, return whether value is found in any node within the list. | **removeFront** Given a pointer to the **first node** in a list, remove the head node and return the new list. If list is empty, return null.  **removeFront(Node) => Node** **front** Return the *value* (not the node) at the **head** of the list. If list is empty, return null.  **front(Node) => Number** |

**contants(Node, Number) => Boolean**

[Back to schedule.](#843uwrr9jjda)

### 

### **Week 1 Wednesday**

### **length**

Create a function that accepts a pointer to first list node, and returns number of nodes in sList.

### **average**

Create a standalone function **average(node)** that returns (…wait for it … ) the *average* of all values contained in that list.

### 

### **min, max**

Create function**min(node)** and **max(node)** to returning smallest and largest values in the list.

### **display**

Create **display(node)** for debugging that returns a string containing all list values. Build what you wish **console.log(myList)** did!

**BONUS:**

### **prependVal**

Create **prependVal(list,value,before)** that inserts a new node with given **value** immediately before the node with **before** (or at end). Return the new list.

### **removeVal**

Create **removeVal(list,value)** that removes from our list the node with the given **value**. Return the new list.

### **appendVal**

Create **appendVal(list,value,after)** that inserts a new listNode with given **value** immediately after the node containing **after** (or at end). Return the new list.

[Back to schedule.](#843uwrr9jjda)

### **Week 1 Friday**

### **splitOnVal**

Create **splitOnVal(list,num)** that, given **number**, splits a list in two. The latter half of the list should be returned, starting with node containing **num**. E.g.: **splitOnVal(5)** for the list (1 >3>5>2>4) will change list to (1>3) and return value will be (5>2>4).

Given: (1)->(2)->(3)->(4), **3**

Return: **(3)**->(4)

### **partition**

Create **partition(list,value)** that locates the first node with that value, and moves all nodes with values *less than* that value to be earlier, and all nodes with values *greater than* that value to be later. Otherwise, original order need not be perfectly preserved.

Given: **(3)**->(1)->(2)->(4)->(5)

Return: (1)->(2)->**(3)**->(4)->(5)

Given: **(10)**->(15)->(11)->(4)->(9)

Return: (4)->(9)->**(10)**->(15)->(11)

### **deleteGivenNode**

Create *listNode method* **removeSelf()** to disconnect (remove) itself from linked lists that include it. Note: the node might be the first in a list, and you do NOT have a pointer to the previous node. Also, don’t lose any subsequent nodes pointed to by **.next**.

*Hint:*

function sList() {

this.head = null;

this.\_\_\_\_ = function(){ .... }

}

function sNode(val) {

this.val = val;

this.next = null;

this.removeSelf = function() { ... }

}

[Back to schedule.](#843uwrr9jjda)

# **Doubly Linked List**

There is certainly no reason why a linked list node must refer to only one other node. For the best flexibility when traversing a list, we would want to be connected in both directions: forward and backward. Whereas singly linked lists feature nodes that only know about their forward neighbor (unable to look backward), *doubly linked lists* are more like lines of preschoolers holding hands as they walk down the street together, on a field trip to the fire station.

For the Doubly Linked List, all the concepts and techniques of Singly Linked Lists apply (see below). This extra flexibility comes with a cost, however. Maintaining both sets of pointers can be tedious.

*classes* and *objects private* vs. *public prototype ===* vs. *== reference* vs. *value*

### **DList Class**

Given the above reference implementations for doubly linked node and doubly linked list, can you construct the rest of a basic dList class? This would include dList methods push(),pop(), front(), back(), contains(), and size().

**Second:** implement these so that they are available as *standalone functions* as well as methods on both dlNode *and* dList classes.

|  |  |
| --- | --- |
| **function dlNode(value)**  **{**  **if (!(this instanceof dlNode))**  **{**  **return new dlNode(value);**  **}**  **this.val = value;**  **this.prev = null;**  **this.next = null;**  **}** | **function dList()**  **{**  **if (!(this instanceof dList))**  **{**  **return new dList();**  **}**  **this.head = null;**  **this.tail = null;**  **}** |

**Ninjas:** as shown on the next page, implement functions we previously built for singly linked lists: **isValid()**, **prependValue()**, **kthToLastElement()**, **deleteMiddleNode()**, **reverse()**, **isPalindrome()** and **partition()**….

[Back to schedule.](#843uwrr9jjda)

### **Week 2 Monday**

### **Prepend Value**

Given **dList**, **new value**, and **existing value**, insert new val into dList immediately *before* existing val.

**prependValue(list, 9, 4)**

**Given:** (1) <--> (2) <--> (3) <--> (4)

**Return:** (1) <--> (2) <--> (3) <-**->(9) <-**-> (4)

### **Kth To Last Value**

Given k, return the value ‘k’ from a dList’s end. *(K from last meaning # of nodes away from the last node.)*

**kthToLast(list, k)**

**Given:** (1) <--> (2) <--> (3) <--> (4) , 2

*(last node being (4), return the node 2 positions before the last node, in this case the node (2))*

**Return:** (2)

### **Is Valid dList**

Determine whether given dList is well-formed and valid: whether next and prev pointers match, etc.

**Given:** (1) <--> (2) <--> (3) <--> (4)

**Return**: true

Given: (1) <--> (2) <--> (3)  **x-**-> (4) *(Node 4 is missing it’s prev pointer)*

**Return**: false

### **Palindrome**

Determine whether a dList is a palindrome

**Given:** (1) <--> (2) <--> (3) <--> (2) <--> (1)

**Return**: true

**Given:** (1) <--> (2) <--> (2) <--> (1)

**Return**: true

**Given:** (1) <--> (2) <--> (3) <--> (1)

**Return**: false

### **Loop Start**

Given a dList that may contain a loop, return a pointer to the node where the loop begins (or null if no loop).

**Given:** (1) <--> (2) <--> (3) <--> *(node (3) points back to the first node)*

**Return**: (3) *(return the node, which starts the loop, in this case, node 3 is starting the loop because it points back to node1)*

[Back to schedule.](#843uwrr9jjda)

### **Week 2 Wednesday**

### **Repair**

Combine previous work with a function that fixes errors found by **isValid** and breaks loops.

### **Append Value**

Given 1) dList, 2) new value, and 3) existing value, insert new val into dList immediately *after* existing val.

### **Delete Middle Node**

Given a node in the middle of a dList, remove it.

### **Reverse**

Create function to reverse nodes in a dList.

[Back to schedule.](#843uwrr9jjda)

### **Partition**

Given dList and partition value, perform a simple partition (no need to return the pivot index).

### **Break Loop**

Given dList that may contain a loop, break the loop while retaining original node order.

[Back to schedule.](#843uwrr9jjda)

### 