Data Structure and Algorithms Notes

Abdallah Abou-Chahine

2023-02-27

1 LINKED LISTS

A **linked list** is a **linear** data structure in which elements are stored in **nodes** and each node **points** to the **next node** in the list. The main advantage of linked lists over arrays is that elements can be **easily inserted** or **removed** without reallocating memory for the entire list. There are two main types of linked lists: **single linked** lists and **double linked lists**.

Each list typically has the following functions:

1. **Insertion:** Adding a new node to the linked list, either at the beginning, end, or a specific position.

2. **Deletion:** Removing a node from the linked list, either based on its value or its position in the list.

3. **Search:** Finding a specific node in the linked list based on its value.

4. **Traversal:** Iterating through all the nodes in the linked list to access their values.

5. **Length:** Finding the number of nodes in the linked list.

6. **Reverse:** Reversing the order of the nodes in the linked list.

7. **Sort:** Sorting the nodes in the linked list based on their values.

1.1 Single Linked List

In a single linked list, each node has a single reference to the next node in the list. This makes it easy to traverse the list in one direction, but difficult to traverse the list in reverse or to efficiently find the previous node.

We start by creating a **Node** class with a constructor:

```
class Node {

    constructor(data){
        this.data = data;
        this.next = null;
    }

}
```

What's effectively happening here is that when the constructor is called it will initialize a newly created object of type Node with the following properties set:

```
this.data = data;
this.next = null
```

Once thats done, we start by creating the singleLinkedList class:

```
    class singleLinkedList{

        constructor(){
            this.head = null;
            this.tail = null
        }


    }
```

The singleLinkedList class contains a constructor that initializes the value of the head and tail of the list to null. Now we will create all the mentioned methods to the class - insertion, deletion, search, traversal, reversal, length and sort.

1.1.1 append()

The purpose of this method is to add a Node to the end of the list. We can create the method with the following code within the *singleLinkedList* class:

```
{
    ...

    append(data){

        const newNode = new Node(data)
        // if the list is empty set the value of the head and tail to the newNode
        if(this.head==null){

            this.head = newNode;
            this.tail = newNode;

        }else{

            this.tail.next = newNode;
            this.tail = newNode;
        }

    this.length++;
    return this;
    }

    ...
}
```

The first thing we need to do is to check if the **head** of the list is null i.e. empty. If the list is empty (which it would be in the beginning) then we will assign the value of the head to the newNode. In the case where the head is not empty, (which will be the case now) we want to set the value of the tail to the newNode. This can be accomplished by the else statement below:

```
...
else{
    this.tail.next = newNode;
    this.tail = newNode;
    }
 ...
```

The first line sets the value of the current tail's **next property** to the newNode. Then the second line sets the value of the tail to the newNode - appending the Node to the end of the list.

Whats interesting about the code above is that we are manipulating and changing the value of *this.tail.next* first (which is a property of *this.tail*) first

before changing the value of *this.tail* to the new Node right after. One may be under the impression that the value of *this.tail.next* might be overwritten by the assignment *this.tail = newNode*. But this is in fact not the case, since we are not dealing with **primitive data types**, but rather with **object references** in memory. The assignment *this.tail = newNode* simply updates the reference of *this.tail* to point to a different object in memory, while the property *next* of the original object remains unchanged. In other words, we are updating the value of the *next* property in the *this.tail* node to reference the *newNode*. Then we are updating the value of the *this.tail* reference to be equal to the *newNode* instead of the previous value that it was pointing to. This allows us to keep adding new nodes to the end of the linked list as the tail reference will keep pointing to the latest node that was added to the list.

1.1.2 prepend()

In this method we would like to add the newly created Node in the beginning of the list. This can be achieved in the following code:

```
...
prepend(data){

    const newNode = new Node(data);
    newNode.next = this.head;
    this.head = newNode;
    this.length++;
    return this;

}
...
```

The *prepend()* method logic is fairly intuitive and simple. We start off by creating a new Node (*new Node(data)*) with the variable name **newNode**. Then we make *this.head* point to the same place in memory that the *newNode* variable is pointing to - making the **newNode** the new head of the list. Then we make the next property of the newNode variable point to where *this.head* is pointing to in memory - **connecting** or **linking** the nodes together.

1.1.3 remove()

The *remove()* method requires a little bit of extra effort to fully understand but is nonetheless still not that hard. Please find the method definition below:

```
...

remove(data){

let current  = this.head;

if(this.head===null){
    console.log("There's nothing to remove!");
}

if(current.data===data){
    this.head = current.next;
    this.length--;
    return this;
}

while(current.next){
    if(current.next.data==data){
        break;
    }
    current = current.next;
}

if(current.next==null){
    console.log("Node not found");
}

current.next = current.next.next;
...
}
```

As it was said earlier this function deals with a lot. The code starts with the variable *current* being assigned to the value of *this.head*, so *current* and *this.head* now reference the same object in memory. So, whenever *current* is modified, the value of *this.head* is also modified. We then check the value of *this.head* to see if its null or not - if the list is empty. Once that is out of the way, we check the value of the *data* property of the *current* node to see if it matches with the data that we are looking for. If true, the value of *this.head* will point to the same node object in memory that the *next* property of the *current* node is pointing to. Thus, skipping over the head and "removing" it.

So the *this.head* node should be equal to the node that was in the *next* property of the *current* node.

If the data of the current node is not equal to the data that we want to remove, the code enters a while loop and iterates over the linked list until it finds a node whose next property has the data that we want to remove. The while loop checks if there are any items left in the linked list by checking if the *next* property of the *current* node is not null. Within the while loop, the *data* property of the *current* node is compared to the desired value. If a match is found, the program exits the while loop using the break statement and sets the next property of the *current* node to the *next* property of the *next* node - skipping over the node that matches the data. If there are no matches found, the *current* node will point to the next property of the *current* node - iterating over the list.

1.2 Double Linked List

In a double linked list, each node has references to both the next and previous nodes in the list. This makes it easy to traverse the list in both directions and to efficiently find both the next and previous nodes. However, double linked lists require more memory than single linked lists because each node needs to store two references.

2 Equations

In LaTeX, equations can be written using the `\[` and `\]` commands:

$$e^{i\pi} + 1 = 0$$

3 Images

Images can be included in a LaTeX document using the `\includegraphics` command: