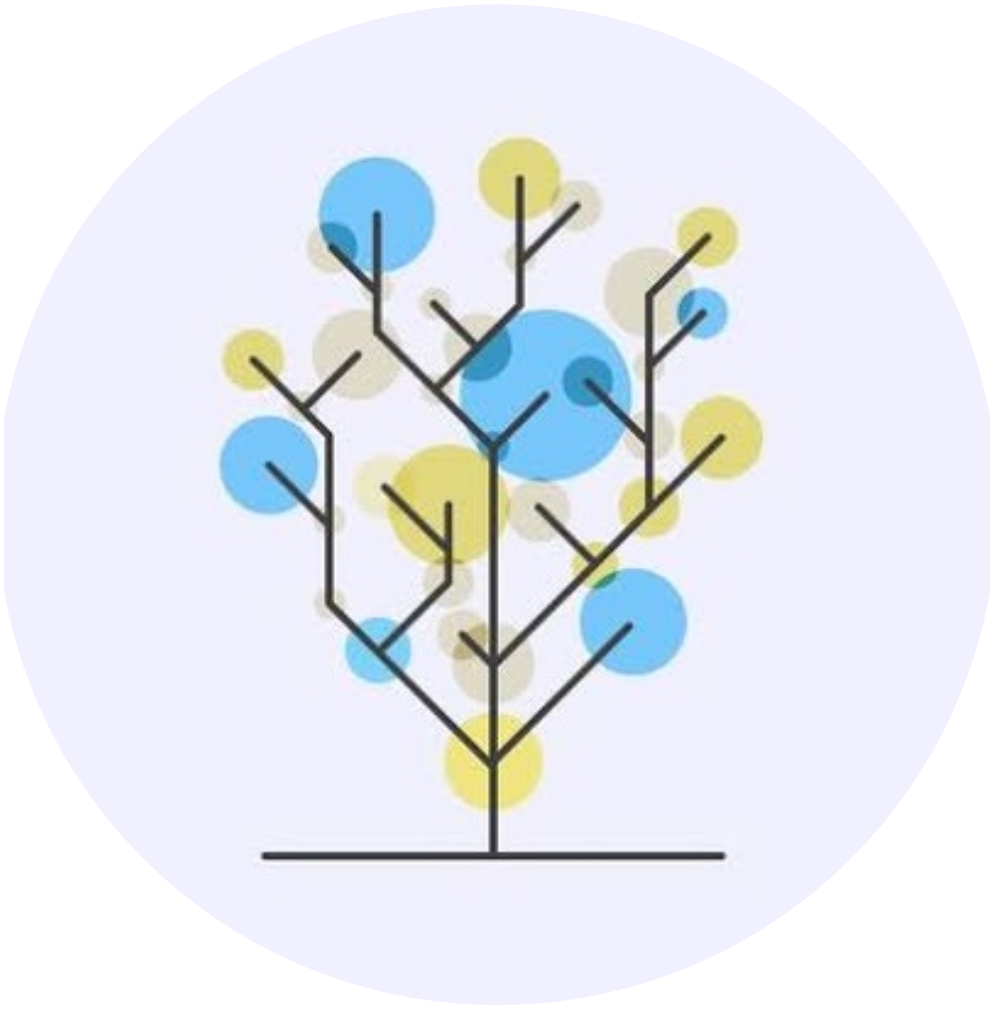


Linked Lists

04.01.2020

Data Structures & Algorithms

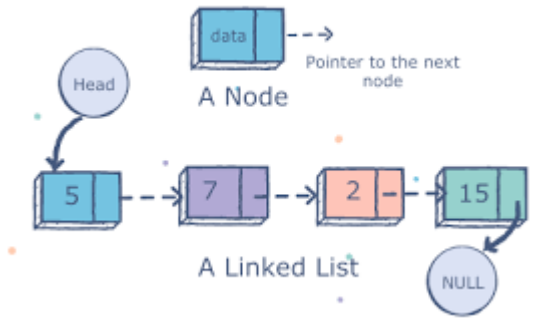
Intro

- **What problem do we encounter with arrays?**
  - We only had a certain amount of memory that can be allocated next to each other in memory.
  - Dynamic arrays can increase their memory, once it hits a certain limit, it double up the current memory in another location. But it costs us O(n).
  - Also arrays have bad performance for any sort of operations like insert and delete that had to shift indexes over.
- **Then came hashtables**,
  - They were great. We were pretty much store things wherever we wanted and memory and hashtables would just take care of it for us.
  - It knows where to place it in memory.
  - But unfortunately they weren't really ordered.
  - **So how can we solve this problem?**
    - By linked lists!
    - Ama bu demek değil ki sürekli linked list kullanalım. Data structures arası trade-off var!

What are Linked Lists

- It is basically a list that is linked.
- Check this link out, to gain a better visual understanding.
- Singly ve Doubly olarak 2 tipi var.

- Singly lists den bahsedersek:
  - They contains a set of nodes.
  - Nodes iki farklı bloktan oluşuyor gibi düşün. Biri **VALUE** tutuyor. Diğeri de **POINTER** to the next node.
  - First node is **HEAD** and the last node is **TAIL**
  - Listenin son node'unu nasıl anlarız? Pointer'ı null gösterir! Yani tail pointer'ı null gösterir.
- Yani temelde linked lists bahsedersek, her biri elemanın sıradaki elemana bağlandığı bir listeden bahsediyoruz.
- Nodes can contain any type of data! Yani VALUE herhangi bir datatype olabilir. HashTables'da da aynı şekilde idi. Hashtable içinde ayrı bir array veya obje yerleştirebiliriz.
-



Pseudo Code

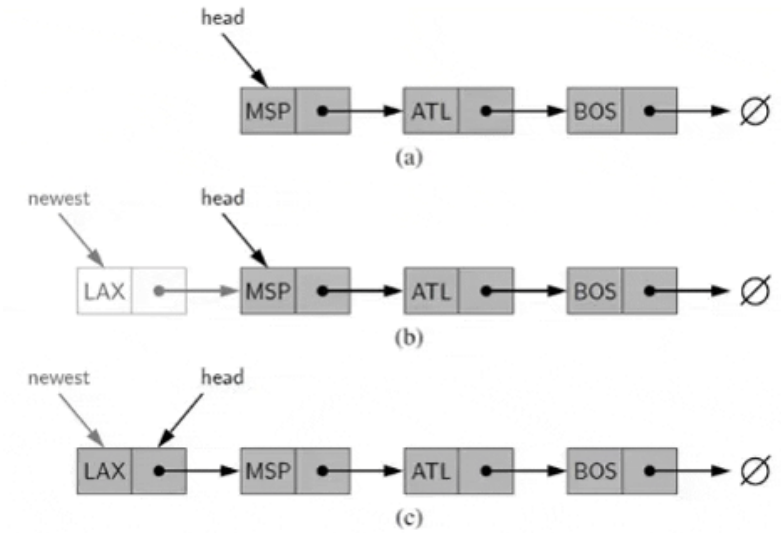
- Aşağıda bir array basket tanımlayıp dolduralım:
  - `const basket = ['apples', 'grapes', 'pears'];`
- Şimdi ise bir Linked List basketimiz olsun. Pseudo Code şeklinde:
  - `linked list : apples --> grapes --> pears`
- Our linked list contains apples that points grapes that points pears. Arrows are pointers. Data detaylı bir diagram şöyle olabilir:

```
apples
3223 --> grapes
          4299 --> pears
                    372 --> null
```

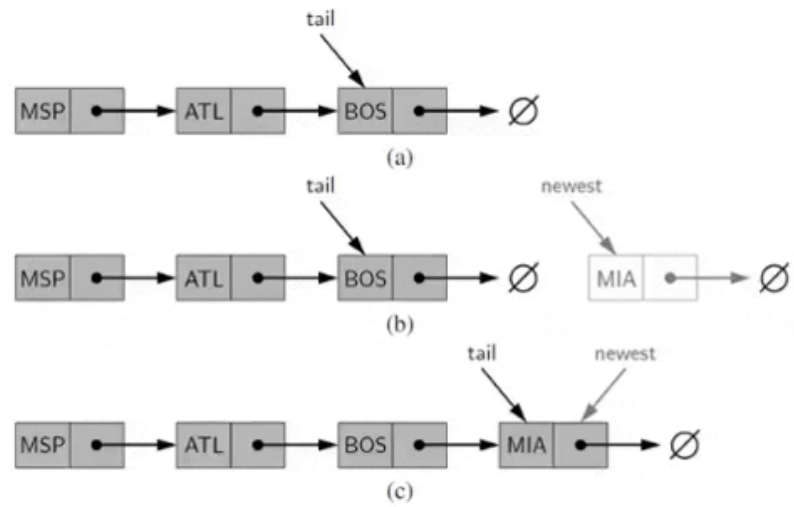
- Apples elamanı 3223. memory locationda, ilk node 4299. memory space'i point ediyor burada grapes var ve 2. node ise 372. memory space'i point ediyor where pears exist. Son olarak son node ise null point ediyor.
- Javascript veya Python built-in linked liste sahip değil.

More about Singly Linked Lists

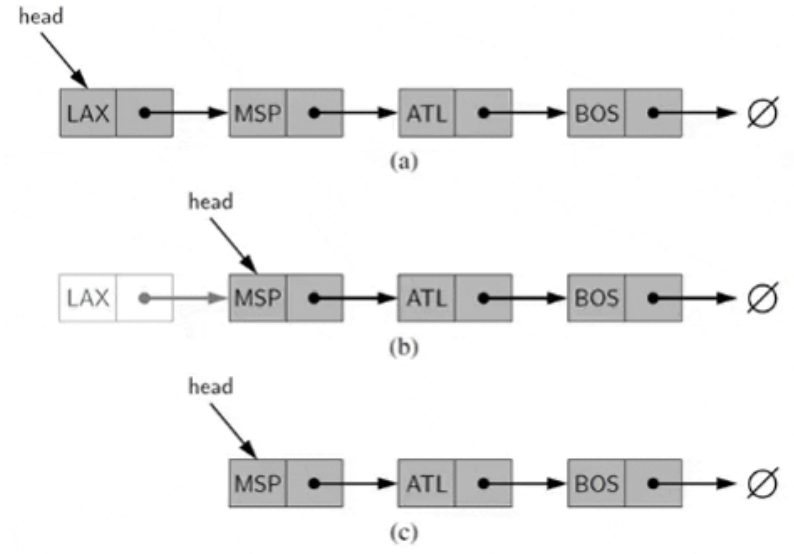
- Process of going throughout the nodes is commonly known as **traversing**.
- **To insert a new element at the head of the list:**
  - Create a node
  - Set its element to the new element
  - Set its next link to refer the current head
  - Set list's head property to point to the new node.

- 

- **To insert a new element at the tail of the the list**
    - Create a node
    - Assign its reference to None
    - Set the next reference of the tail to point to this new node
    - Then update the tail reference itself to this new node

- 

- **Removal of Head:**

- 

- **Removal of Tail:**
    - We can not easily do it.
    - Tail'e doğrudan ulabiliriz ama ondan bir önceki elemana tail üzerinden ulaşamayız. Head'den başlayıp traverse gerekir bu da işlemi verimsizleştirir.
    - Bu işlemi verimli yapabilmek için doubly linked list kullanılır.

Why Linked Lists?

- Array'de doğrudan 5. indexe ulaşabilirim ama, linked list için önce head'den başlarım ve traverse the list until you get the 5. item. Bu durum da **O(n)** demek, **access** için en baştan başlıyorum tüm elemanları tek tek geziyorum. Pointerları takip etmeden buna ulaşamam.
- Array items are always located right next to each other. Nodes of linked lists are scattered all over memory, kinda like hashtables.

- SO ITERATING OR TRAVERSING THROUGH A LINKED LIST IS USUALLY QUITE A BIT SLOWER THAN ITERATING THROUGH ITEMS LIKE AN ARRAY EVEN THOUGH THEY ARE TECHNICALLY O(n).
- HOWEVER INSERTING IN THE MIDDLE OF A LINKED LIST IS A LOT BETTER THAN ARRAY

- WHAT ABOUT HASHTABLES? JUST LIKE HASHTABLES WHEN WE INSERT SOMETHING INTO A LINKED LIST WE JUST SCATTER IT ALL OVER THE MEMORY AND WE CAN JUST KEEP ADDING IT. ARRAYS İLE YAPILAN SHIFTING OLAYI BURADA DA YOKTU. AYRICA YINE SHIFTING OLMADAN DELETE DE YAPABİLİYORDUK.

- LINKED LIST'IN HASHTABLE'E GÖRE BİR **AVANTAJ**I ŞU: THERE IS SOME SORT OF **ORDER TO LINKED LIST**. EACH NODE POINTS TO THE NEXT NODE SO YOU CAN HAVE SORTED DATA UNLIKE A HASHTABLE
- LINKED LIST İÇİN ELEMANLARIN BİR SIRASI VARDIR KİMİSİ DİĞERİNDEN ÖNCE KİMİSİ SONRA GELİR, AMA HASHTABLES ILE BU BELIRSIZ!

Actions and BigOs

- **PREPEND: O(1)**
- **APPEND : O(1)**
- **LOOKUP : O(n)**

- **INSERT : O(n)**
- **DELETE : O(n)**

- Insert ve Delete Array ile aynı görünüyor ama bu worst case! Bunun üzerinde daha fazla konuşacağız. Demekki arrayden daha avantajlı.

Pointer

- Pointer is basically a reference to another place in memory or another object or another node.
- Let's say I have an object:
  - const obj1 = {a: true};
  - const obj2 = obj1;
- We have just created an object. A reference to an object. So the pointer is simply a reference. Yukarıdaki örnekte objeyi kopyalamadım. Yani a:true property'si olan ve hafızada ayrıca bir yer kaplayan bir başka obje yaratmadım. RAM içinde sadece 1 tane obje var fakat obj1 ve obj2 aynı memory block'u gösteriyor.
- Objelerin bir tanesinde yapılacak bir değişiklik diğerini de etkileyecektir.
- Obj1'ı silersem sadece o referans silinmiş olur obje hala hafızada durur. obj2 ile ulaşabiliriz ancak obj1 ile ulaşamayız. Eğer Obj1 i sildikten sonra Obj2="asd" deyip başka bir noktayı point ettirirsek. Artık {a:true} garbage collection ile hafızadan silinir.
- Low level languages'da bu garbage collection'ı biz yapmalıyız point edilmeyen objeyi kendimiz silmeliyiz. Bu sıkıntılar çıkarabilir ama çok daha hızlı programlar yazabiliriz.

Implement a Singly Linked List

```
// Önce basit bir linked list yaratalım:
// 10 --> 5 --> 16 olsun.

let myLinkedList = {
  head: {//Head node'unu içinde 2 adet bilgi tutan bir bucket(hash tables bucket'ı key ve value tutuyordu) gibi düşünebiliriz. İlki value, ikincisi pointer.
    value: 10,
    next: { //Pointer da yine bir obje! Başka bir node!
        value: 5,
        next: {
           value:16,
           next: null
        }
    }

  }
}

//Yukarıdaki örneği baz alarak linkedList class'ı oluşturacağız!



class LinkedList{
  constructor(value){
    this.head = {
      value: value,
      next : null
    };
    this.tail = this.head;
    this.length = 1;
  }

append(value) {
    const newNode = {
      value: value,
      next: null
    } //Yeni bir node yaratılıyor!
    this.tail.next = newNode; // Biraz evvel tail olan node son eklenen node'u gösteriyor.
    this.tail = newNode; //Yeni node tail oluyor.
    this.length++;
    return this;
  }

  prepend(value) {
    const newNode = {
      value: value,
      next: this.head
    }
    this.head = newNode
    this.length++;
    return this;
  }
}


const myLinkedList = new LinkedList(10); // Bunu dediğimde class'ı instantiate (somutlaştırmak) etmiş olacağım ve constructor çalışacak ve obje yaratmış olacağım.
```

```
//İlk başta girilen değer ile bir linkedList yaratılacak tek bir node olacak. Daha sonra array'e eleman ekler gibi eleman ekleyeceğiz!

//LinkedList'in başlangıçta 3 temel property'si var 1.si head node'u
//2.si tail node'u 3.sü de length'i. Başlangıçta head ve tail nodu aynı olacak girilen ilk değer ve pointerları da null işaret edecek!


//2. adım olarak LinkedList class'ımıza bir append metodu ekleyeceğiz böylece linked list'in sonuna eleman ekleyebilmeliyiz.

//Önce kendim myAppendMethod(value){} olarak deneyeceğim. Sonra kursa bakarak yapacağım.

myLinkedList.append(5);
myLinkedList.append(16);
myLinkedList.prepend(11);

//ÖNEMLİ BİR NOKTA ŞU HER METHOD İÇİN AYRI AYRI NEW NODE OBJESİ OLUŞTURDUK, PROGRAMLAMADA BU PEK MANTIKLI DEĞİL SÜREKLİ YENİ OBJE OLUŞTURMAK YERİNE BİR KERE CLASS OLUŞTURUP BUNU INSTANTIATE ETMEK DAHA MANTIKLI OLUR. BUNU YAPALIM:

class Node {
  constructor(value){
    this.value = value;
    this.next = null;
  }
}

//Yenilenmiş LinkedList (Sadece new nodes yenilendi.)
class LinkedList{
  constructor(value){
    this.head = {
      value: value,
      next : null
    };
    this.tail = this.head;
    this.length = 1;
  }

append(value) {
    const newNode = new Node(value); //Yeni bir node yaratılıyor!
    this.tail.next = newNode; // Biraz evvel tail olan node son eklenen node'u gösteriyor.
    this.tail = newNode; //Yeni node tail oluyor.
    this.length++;
    return this;
  }

  prepend(value) {
    const newNode = new Node(value);
    newNode.next = this.head;
    this.head = newNode
    this.length++;
    return this;
  }

  // Linked List'i görselleştirmek için bir printList() yazalım:
```

```
  printList(){
    const array = [];
    let currentNode = this.head;
    while(currentNode != null) {
      array.push(currentNode.value);
      currentNode = currentNode.next;
    }
    return array;
  }

  //ŞİMDİ ZOR OLAN METHODLARI UGULAYACAĞIZ BUNLAR O(n) OLACAK YANİ BİR LOOPİNG OLACAK:

  insert(index,value){ //İstediğim indexe eleman eklesin.Index saymaya 0 dan başlar array gibi.
Yani index=2'ye "Erdo" value'sunu yerleştir dersek.

  // 1 --> 10 --> 16 iken 1 --> 10 --> "Erdo" --> 16 olur!

    //Check for proper parameters;
    if(index >= this.length) {
      return this.append(value);
    }

    if(index < 1) {
      return this.prepend(value);
    }

    const newNode = {
      value: value,
      next: null
    }

    const leader = this.traverseToIndex(index-1);
    const holdingPointer = leader.next;
    leader.next = newNode;
    newNode.next = holdingPointer;
    this.length++;
    return this.printList();
  }


  remove(index) {
    // Check Parameters
    const leader = this.traverseToIndex(index-1);
    const unwantedNode = leader.next;
    leader.next = unwantedNode.next;
    this.length--;
    return this.printList();
  }



  traverseToIndex(index) {
    //Check parameters
    let counter = 0;
    let currentNode = this.head;
    while(counter !== index){
      currentNode = currentNode.next;
      counter++;
```

```
    }
    return currentNode;
  }

}
```

Doubly Linked List

- Doubly linked lists singly ile çok benzer. Sadece node'lar kendinen önceki node'ları da point ediyor.
-
```
        -->  -->    -->
      1     5    10    null
        <--  <--
```
- DOUBLY LINKED LISTS ALLOW US TO TRAVERSE OUR LIST BACKWARDS. ŞİMDİYE KADAR HEP SAĞA DOĞRU TEK YÖNDE İLERLİYORDUM ÇÜNKÜ POİNTER'LAR TEK YÖNÜ GÖSTERİYORDU. DOUBLY LINKED LIST İLE ARTIK TERSİ YÖNDE DE İLERLEYEBİLİRİM ÇÜNKÜ HEAD DIŞINDA HER NODE HEM KENDİNDEN SONRAKİNİ HEM DE ÖNCEKİNİ POİNT EDER.
- YANİ SINGLY LINKED LIST İLE TEK FARKI PREVIOUS POINTER OLMASI!

- LOOKUP O(n/2) oldu yani O(n)'den biraz daha hızlı.

- **SINGLY LINKED LISTS VS DOUBLY LINKED LISTS**
  - **Singly + :** Fairly simple implementation. Requires less memory. Less memory olduğu için DELETE INSERT gibi methodlar daha kısa sürer. Çünkü sadece tek pointer ile uğraşıyoruz.
  - **Singly - :** Sadece tek yönde traverse yapılabilir.
    - Singly linkled lists are appropriate to use when you have less memory or memory is really expensive. And your main goal should be fast insertion and deletion.

  - **Doubly + :** It can be traversed both from front and the back. Previous node silme işlemi yapmak için headden başlamaya gerek yok işi hızlandırır.
  - **Doubly - :** Fairly complex. Requires more memory. Insert ve Delete biraz daha uzun sürer çünkü her node için 2 pointer var.
    - Doubly linked lists are appropriate to use when you don't have that much limitation on memory.
    - When you want good operation for searching for elements. Such as searching backwards.
    - Aranacak değerin nerede olduğunu biliyorsak doubly linked list ile daha hızlı bulabiliriz mesela sondan başlarız geri gideriz böylece sonlardaki bir node'u çok daha hızlı bulabiliriz.

Implement a Doubly Linked List

```
class DoublyLinkedList {
  constructor(value) {
    this.head = {
      value: value,
      next: null,
      prev: null
    };
    this.tail = this.head;
    this.length = 1;
  }
  append(value) {
    const newNode = {
      value: value,
      next: null,
      prev: null
    }
    console.log(newNode)
    newNode.prev = this.tail
    this.tail.next = newNode;
    this.tail = newNode;
    this.length++;
    return this;
  }
  prepend(value) {
    const newNode = {
      value: value,
      next: null,
      prev: null
    }
    newNode.next = this.head;
    this.head.prev = newNode
    this.head = newNode;
    this.length++;
    return this;
  }
  printList() {
    const array = [];
    let currentNode = this.head;
    while(currentNode !== null){
      array.push(currentNode.value)
      currentNode = currentNode.next
    }
    return array;
  }
  insert(index, value){
    //Check for proper parameters;
    if(index >= this.length) {
      return this.append(value);
    }

    const newNode = {
      value: value,
      next: null,
      prev: null
    }
    const leader = this.traverseToIndex(index-1);
    const follower = leader.next;
    leader.next = newNode;
```

```
    newNode.prev = leader;
    newNode.next = follower;
    follower.prev = newNode;
    this.length++;
    console.log(this)
    return this.printList();
  }
  traverseToIndex(index) {
    //Check parameters
    let counter = 0;
    let currentNode = this.head;
    while(counter !== index){
      currentNode = currentNode.next;
      counter++;
    }
    return currentNode;
  }
}

let myLinkedList = new DoublyLinkedList(10);
myLinkedList.append(5)
myLinkedList.append(16)
myLinkedList.prepend(1)
myLinkedList.insert(2, 99)
```

Reverse Example

```
// SADECE BIR REVERSE METHODU EKLEYECEM VE TAIL HEAD OLACAK SONDAN BASA DOGRU LINKED OLACAKLAR.

class Node {
  constructor(value){
    this.value = value;
    this.next = null;
  }
}

//Yenilenmiş LinkedList (Sadece new nodes yenilendi.)
class LinkedList{
  constructor(value){
    this.head = {
      value: value,
      next : null
    };
    this.tail = this.head;
    this.length = 1;
  }

append(value) {
    const newNode = new Node(value); //Yeni bir node yaratılıyor!
    this.tail.next = newNode; // Biraz evvel tail olan node son eklenen node'u gösteriyor.
    this.tail = newNode; //Yeni node tail oluyor.
    this.length++;
    return this;
  }

  prepend(value) {
    const newNode = new Node(value);
    newNode.next = this.head;
    this.head = newNode
    this.length++;
    return this;
  }

  // Linked List'i görselleştirmek için bir printList() yazalım:

  printList(){
    const array = [];
    let currentNode = this.head;
    while(currentNode != null) {
      array.push(currentNode.value);
      currentNode = currentNode.next;
    }
    return array;
  }

  //ŞİMDİ ZOR OLAN METHODLARI UGULAYACAĞIZ BUNLAR O(n) OLACAK YANİ BİR LOOPİNG OLACAK:

  insert(index,value){ //İstediğim indexe eleman eklesin.Index saymaya 0 dan başlar array gibi.
Yani index=2'ye "Erdo" value'sunu yerleştir dersek.

  // 1 --> 10 --> 16 iken 1 --> 10 --> "Erdo" --> 16 olur!

    //Check for proper parameters;
    if(index >= this.length) {
      return this.append(value);
```

```
    }

    if(index < 1) {
      return this.prepend(value);
    }

    const newNode = {
      value: value,
      next: null
    }

    const leader = this.traverseToIndex(index-1);
    const holdingPointer = leader.next;
    leader.next = newNode;
    newNode.next = holdingPointer;
    this.length++;
    return this.printList();
  }


  remove(index) {
    // Check Parameters
    const leader = this.traverseToIndex(index-1);
    const unwantedNode = leader.next;
    leader.next = unwantedNode.next;
    this.length--;
    return this.printList();
  }



  traverseToIndex(index) {
    //Check parameters
    let counter = 0;
    let currentNode = this.head;
    while(counter !== index){
      currentNode = currentNode.next;
      counter++;
    }
    return currentNode;
  }

    reverse() {
      if (!this.head.next) { // tek eleman var ise
        return this.head;
      }

      let first = this.head;
      this.tail = this.head;
      let second = first.next;

      while(second) {
        const temp = second.next;
        second.next = first;
        first = second;
        second = temp;
      }
```

```
        this.head.next = null;
        this.head = first;
        return this.printList();
    }

}
```

Review

- LINKED LISTS ARE LOW LEVEL DATA STRUCTURES. BU YÜZDEN BAZI HIGH LEVEL DILLERDE YOK!
- HASH TABLES GİBİ DİĞER DATA STRUCTURES İÇERİSİNDE KULLANILIRLAR. MESELA COLLISION DURUMUNDA AYNI BUCKET'DA BİRDEN FAZLA KEY-VALUE PAİR TUTULAMAYACAĞI İÇİN LİNKED LISTS'DEN YARARLANILMIŞTI.
- ILERIDE GORECEGIMIZ STACKS AND QUEUES ICINDE DE KULLANILACAK.

- THERE IS NO RANDOM ACCESS, YANİ ARRAYDEKİ GİBİ DİREKT 3. ELEMANA ULAŞAYIM VEYA HASHTABLES DAKİ GİBİ ŞU KEY'E ULAŞAYIM DİYEMİYORSUN.
- BUNUN YERİNE HEAD'DEN BAŞLAYIP TRAVERSE İLE TEK TEK ELEMANLARI GEZMEN GEREK.
- YANİ **SLOW LOOKUPS**

- AMA **UNLIKE HASHTABLES THEY ARE ORDERED!**

- BUYUK BIR ARRAYIMIZ VAR ISE VE SUREKLI ELEMAN EKLIYORSAK, SONUÇTA BİR ZAMAN AYRILAN HAFIZA DOLACAK VE ARRAY KOPYALANACAK BOYUTU İKİYE KATLANMIŞ ŞEKİLDE BAŞKA BİR HAFIZA BÖLÜMÜNE TAŞINACAK.
- LINKED LIST ILE **ARRAY'E GÖRE FAST INSERTION** VE **FAST DELETION** YAPABİLİRİZ.
- HELE HELE LISTENIN BAŞINA VEYA SONUNA EKLEME YAPACAKSAK LINKED LIST ARRAY'E GORE ÇOK DAHA HIZLI
- ARRAY YERİNE LINKED LIST KULLANMAMIZIN SEBEBİ : ITS SIMPLICITY AND ABILITY TO GROW AND SHRINK AS NEEDED. YANİ FLEXIBLE SIZE!

More blogs