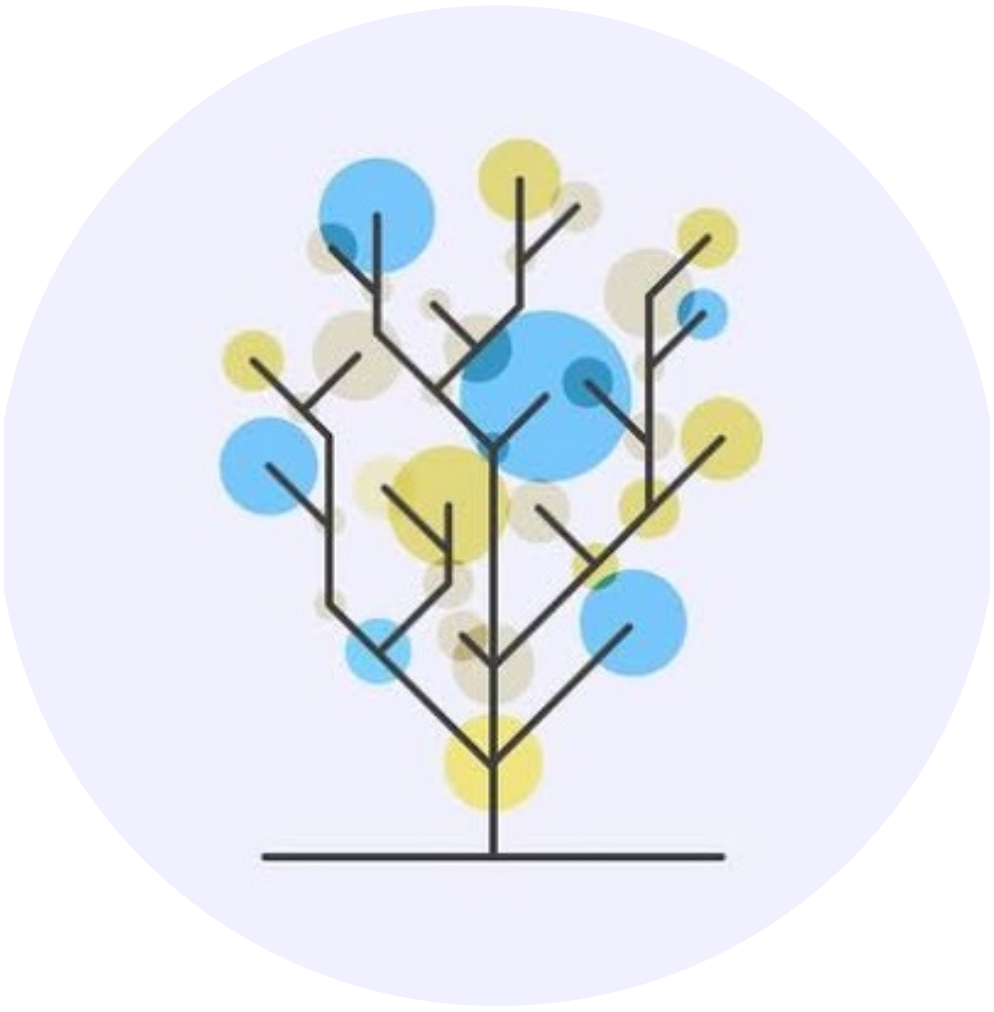

Hash Tables

03.01.2020

Data Structures & Algorithms

Intro

- Different languages have different names for hash tables.
- In javascript objects are a type of hash table.
- Python da dictionaries denir. Java da maps denir. Ruby de hash denir.

- Peki nerede kullanılır. Mesela bir basket objem var.
- Basket.grapes = 1000; dediğim zaman. Basket objesinin grape property'sini 1000 olarak atamış oldum.
- Python için ise, `basket = { "grapes": 1000, "apples" : 50}` şeklinde dictionary tanımlanabilir.

- **Array** de bu tip bir saklama yapmak daha zor olurdu. Array de **INDEX** ve **VALUE** var.
- **Hash Tables** için ise **KEY** (grapes) var ve **VALUE** (1000) var. **KEY-VALUE pair**.

- KEY is used as an index of where to find the value in memory.

Hash Function

- **Hash Function:** içine KEY'i alır yani grapes'i, daha sonra bir index döndürür.
- Bu index bir memory shelf'i gösterir. Bu memory shelf'de aslında hem grapes ismi yani key saklanır hem de bunun value'su saklanır.

- Bu hash function ilgili key'e göre data'yı RAM içinde nereye yerleştireceğine karar veren fonksiyondur.

- **Hash Function:** A function that generates a value of fixed length for each input that it gets.
- Basitçe hash function'ın yaptığı şey şu, bir input alır ve bize rastgele bir pattern oluşturur mesela "Grapes" verdik sonucunda "aXXIJDJ29DNDjjdnasjğuıfjn" gibi bir şey döndürdü.

- **Important Points:**
- Hash Function tek yönlüdür. Birine "aXXIJDJ29DNDjjdnasjğuıfjn" kodunu versem bunu "Grapes" ile bağlamalarının mümkünatı yok.
- Ayrıca ben tekrar "Grapes" yazsam yine birebir aynı kodu üretir. Her seferinde başka kod üretmiyor yani. İnputa göre üretiyor.
- Sanırım her bir key için unique bir kod üretiyor. ancak MEMORY UNIQUE OLMUYOR KI COLLISION OLUYOR.

- By using Hash Function we get fast data access.
- Yani biz Basket.grapes dediğimizde bir hash function bize rastgele bir kod üretiyor sonra bu kod bir adres'e dönüştürülüyor ve artık grapes key'i bir adresi temsil etmiş oluyor.

- Arraylerde data ardarda shelf'lerde tutuluyordu.Hash table için öyle değil.
- Burda key'i verince doğrudan hafızadaki adresine ulaşmış oluyoruz.

- Every time we want to add a property and a value (grapes and 1000) to a hashtable, we have to run it through the hash function so we can decide where to put it in memory.
- EVERY TIME WE ADD A PROPERTY OR RETRIEVE A PROPERTY FROM MEMORY, THE KEY IS SENT TO THE HASH FUNCTION TO FIND THE ADDRESS.
- TIME COMPLEXITY OF HASHFUNCTION IS O(1).

Actions and Big Os

- **INSERT O(1):** We hash the key and place it into the address.
- **LOOKUP O(1):** Exact same. We access the property, property is getting hashed and direct us exactly to the address to find the values.
- **DELETE O(1):** Same thing. By using key we know where to delete the item from. Ordered falan olmadığı için bir şey silinince diğer elemanları shift etmek vb gibi bir şey yok!
- **SEARCH O(1):** Obje içinde bir şey bulmak istiyorsak mesela basket içinde apple var mı? Yine aynı şekilde hash function ile ulaşabilirim.

Example

```
//Let's define an object

let user = {
  age:54,
  name:'Kylie',
  magic:true,
  scream: function(){
    console.log('ahhhh!');
  }
}

//Kodu çalıştırınca bu user objesi içindeki properties farklı adreslerde memory içinde kaydedilecek.

//Bu properties'e hızlı bir şekilde ulaşabilirim. Tek yapmam gereken > user.properyname

//access to a property
user.age; // O(1)

//add a new property
user.spell = 'abra kadabra!'; // O(1)

//access to a function
user.scream; // O(1)
```

Hash Collisions

- Remember our computer has limited memory. When we create an object or hashtable the computer decides how much space to allocate. It is going to allocate.
- For the key and value, the hash function randomly assigns a space in memory.

- THERE IS NOTHING TELLING TO HASH FUNCTION TO EVENLY DISTRIBUTE UNTIL EVERY SPACE IS FULL.
- YANİ BİR MEMORY SPACE'E BİRDEN FAZLA ELEMAN ATMA İHTİMALİ VAR BUNUN KONTROLÜ YAPILMIYOR ÇÜNKÜ RASTGELE ADRES ATIYOR.
- BU KEY VE VALUE'LARI AYNI MEMORY SPACE'E YERLEŞTİRME İŞLEMİNE COLLISION DENİYOR.
- BU COLLISION OLUNCA AYNI ADDRESS SPACE TE BIRDEN FAZLA KEY-VALUE PAIR'INI TUTMAMIZ GEREKİR. BU PROPLEMİN FARKLI ÇÖZÜMLERİ VAR. MESELA LINKED LIST DATA STRUCTURE İLE ÇÖZÜLEBİLİR.

- WITH HASHTABLES WE CAN'T AVOID THESE COLLISIONS. WITH ENOUGH DATA AND WITH LIMITED MEMORY WE'RE ALWAYS GONNA HAVE THIS COLLISION.

- BU AYNI ADRESE KAYDEDILEN BIRDEN FAZLA KEY-VALUE PAIR DAHA SONRA ULASILACAKKEN ISLEMI YAVASLATIR.
- WHEN YOU HAVE A COLLISION, IT SLOWS DOWN READING AND WRITING WITH A HASH TABLE WITH O(n/k) where k is the size of the hash table.
- SONUÇTA O(n/k) > O(n) operation olur.

- SONUÇTA HASHTABLES İÇİN ARA SIRA LOOKUP VEYA DIGERLERI O(1) yerine O(n) alabilir?

Different Languages

- Hash tables implemented differently in different languages.
- But most of the time:
  - THE KEY AND THE VALUE CAN BE ANY TYPE OF DATASTRUCTURE.
  - EXAMPLE DA GÖRDÜK MESELA SCREAM KEY'I İÇİN VALUE BİR FUNCTION İDİ VEYA VALUE BAŞKA BIR OBJECT YA DA ARRAY DE OLABİLİRDİ.
  - ALSO SOMETIMES KEYS CAN BE SOMETHING ELSE THAN STRINGS: FUNCTION KEYS VB OLABİLİRDİ

- Javascript yeni versiyon için objenin içine key olarak hep string veriliyor ancak baika özellikler var:

  - MAPS VE SETS JAVA SCRIPT ICIN HASH TABLE TIPLERI:
  - DIFFERENCE B/W MAP AND OBJECT IS: MAPS ALLOWS YOU TO SAVE ANY DATA TYPE AS KEYS.
  - ANOTHER BENEFIT OF MAP IS THAT IT MAINTAINS INSERTION ORDER. Arrayde elemanlar sıralı olarak hafızada tutuluyordu objede ise rastgele tutuluyordu ilk geleni 123'e atıyor 2. yi 70'e 3. yü 550'ye tamamen rastgele. Ama MAPS için bu ordered.

  - JUST LIKE MAPS. THE ONLY DIFFERENCE FOR SETS: IT ONLY STORES THE KEYS NO VALUES.

Building Hash Table

```
class HashTable {
  constructor(size){
    this.data = new Array(size); //data adında array tanımladı.
  }

  //Basit bir hash function:
  _hash(key) {
    let hash = 0;
    for (let i =0; i < key.length; i++){
        hash = (hash + key.charCodeAt(i) * i) % this.data.length
    }
    return hash;
  } //O(1)

  //Yukarıda data.length ile kalanı bulduğumuz için sonuç her zaman data.length'ten küçük olacak.
  //Yani ben size'ı 50 olarak belirttiysem, adres hep 0 ile 50 arası verecek.

  //Let's create the set method. KEY ve VALUE alsın hafızada bir yere kaydetsin. Kaydedeceği yeri hash function ile belirleyecek.

  set(key,value){
    let address = this._hash(key); // Alınan key'e göre address belirlendi.
      if (!this.data[address]){ //Eğer hashfunction ile verilen adres boş ise. Datayı buraya kaydedelim
        this.data[address] = []; //data arrayinin address indexine bir array yerleştirildi.
    }
    this.data[address].push([key,value]); //yerleştirilen array'e  bir array eleman eklendi. Array içinde array içinde array.
  }//O(1)


  //Let's create a get method. Buraya KEY girilecek ve sonucunda VALUE return edilecek.

  get(key){
    let address = this._hash(key); //Ulaşılacak address elde edildi.
      const currentBucket = this.data[address]; // data arrayinin address indexindeki 2.katman arrayine ulaşıldı. (if exists)
    if(currentBucket){ //Eğer bu currentBucket da yani Hem KEY hem VALUE tutan bu hafıza kısmında bir şey var ise, boş değil ise:
        //Bu currentBucket içinde collision olduysa birden fazla eleman olabilir onun için loop kullanalım.
      for(let i=0; i < currentBucket.length; i++){
        if(currentBucket[i][0] == key){
          return currentBucket[i][1];
        }
      }
    }
    return undefined // Ulaşılan adres boş ise undefined dönüyor. Null gibi.
  } //Eğer collision yoksa veya azsa O(1) genelde de böyledir.
// AMA DÜŞÜN Kİ 2 MEMORY SPACE AYRILMIŞ HER GELEN DATA COLLISION'A UĞRUYOR O ZAMAN O(n/2) den O(n) KABUL EDİLEBİLİR.


  keys(){
    const keysArray = [];
    console.log(this.data.length);
    for (let i = 0; i < this.data.length; i++){
```

```
      if(this.data[i]){
        keysArray.push(this.data[i][0][0])
      }
    }
    return keysArray;
  }  //keyleri döndürüyor ama dikkat et keyleri bulması için tüm hashtable hafızasını (şuan 50 elementlik) turlaması gerekiyor.
  //Array için 2 eleman eklediğimizde sadece bu iki elemanı döneriz yine O(n) ama Hash Table için hafıza 50 iken 2 eleman var ise
  //elemanların nerede olduğu da belli olmadığı için tüm hashtable'ı loop ile dönmemiz gerekiyor. O(n) den büyük.


}

const myHashTable = new HashTable(50);
myHashTable.set('grapes', 10000)
myHashTable.get('grapes')
myHashTable.set('apples', 9)
myHashTable.get('apples')
myHashTable.keys()
```

Arrays vs Hash Tables

- Hash tables are great when you want quick access to certain values.
- **ARRAYS** **HASH TABLES**

  Search: **O(n)** **O(1)**

Arrayde belirli bir string'i veya herhangi bir elemanı ararken, her elemana tek tek bakmamız ve check etmemiz gerekiyor.
Hash table için ise bu hızlı direkt olarak key'in adresine gidilip bakılıyor var mı yok mu.
Bu yüzden Hash tables databases için kullanışlı. Database'de bir şey arayacağımızda hemen sonuç verir.

Insert: **O(n)** **O(1)**

Arrayde ortaya eleman eklenince, diğer elemanlar shift edilir ve sonuçta loop işin işine girer. Hash tables için ise böyle bir şey yoktur anında eleman yerleştirilir. Bazen collision olabilir o zaman O(n) olur ama bunu düşünmesek de olur. O(1) kabul ederiz.

Lookup: **O(1)** **O(1)**

Bu aynı. Array'in KEY'i index'i oluyor.

Delete: **O(n)** **O(1)**

Yine shifting farkı var.

- Ayrıca dynamic array için PUSH metodu var. Statik Array'e eleman ekleyemiyoruz zaten eklemek istersek kopyalayıp yeni bir array yaratmamız lazım direkt O(n) olur. Push metodu dynamic array için normalde O(1) ama bazen, datayı alıp başka yere kopyalayabiliyor o zaman O(n) oluyor!!!

- **HASH TABLE PROBLEMİ ŞU:** NO CONCEPT OF ORDER! ARRAY İÇİN HER ELEMAN ARDIŞIK SHELFS İÇİNDE TUTULUYOR. HASH TABLE ELEMANLARI İSE ALAKASIZ YERLERDE TUTULUYOR. GİREYİM TÜM ELEMANLARI TEK TEK TOPLAYAYIM DESEN HASH TABLE İÇİN DAHA UZUN SÜRÜYOR, BUNU IMPLEMENTATIONDA ANLYABİLİRİZ.

Example

First Recurring Character Problem

```
//Google Question
//Given an array = [2,5,1,2,3,5,1,2,4]:
//It should return 2

//Given an array = [2,1,1,2,3,5,1,2,4]:
//It should return 1

//Given an array = [2,3,4,5]:
//It should return undefined


function firstRecurringCharacter(input) {
  for (let i = 0; i < input.length; i++) {
    for (let j = i + 1; j < input.length; j++) {
      if(input[i] === input[j]) {
        return input[i];
      }
    }
  }
  return undefined
}
//Yukarıdaki kod ile time complexity O(n^2) space complexity O(1)

function firstRecurringCharacter2(input) {
  let map = {};
  for (let i = 0; i < input.length; i++) {
    if (map[input[i]] !== undefined) {
      return input[i]
    } else {
      map[input[i]] = i;
    }
  }
  return undefined
}
//Bu kod ile ise map[input[i]] dediğimiz zaman input[i] key oluyor bu sadece key tutmuş oluyor o
yüzden map dedik sanırım.
//map.input[i] demek ile aynı.
//sonuçta tek operation ile bu eleman hash table içinde var mı yokmu anlayabiliyorsun. Arrayde bunu
yapamayız.
//Eleman hash table de var mı yok mu check et. Var ise elemanı dödnür. Yok ise elemanı hash table'a
ekle. Sıradaki elemana geç!



firstRecurringCharacter2([1,5,5,1,3,4,6])
//Üstteki algoritma 1 döndürür çünkü 1'i seçip diğerlerini tek tek döner. Bu algoritma
ileridekilere bakar.
//Alttaki algoritma ise 5 döndürür. Çünkü bu algoritma geridekilere bakar.


//Bonus... What if we had this:
// [2,5,5,2,3,5,1,2,4]
// return 5 because the pairs are before 2,2
```

Pros and Cons

- **+ FAST LOOKUPS (if there is not any collision)**
- **+ FAST INSERTS**
- **+ ITERATION**
- **+ FLEXIBLE KEYS**

- **- UNORDERED**
- **- SLOW KEY ITERATION**

- GENELDE HASH TABLES TIME COMPLEXITY PROBLEMİNİ ÇÖZER BUNUN İÇİN SPACE COMPLEXITY ARTAR!
- BİR ÖNCEKİ ÖRNEKTE OLDUĞU GİBİ NESTED LOOPLARI - TEK LOOP'A DÜŞÜRDÜK. KARŞILAŞTIRMA İŞİNDE O(n) yerine O(1) OLMASI ÇOK İŞLEVSEL. BU HEP KARŞIMIZA ÇIKAR. O(n^2) yi O(n) e ÇEKEBİLİRİZ. AMA SPACE COMPLEXITY O(1) YERINE O(n) OLUR!

- HASHTABLES ARE UNORDERED, HARD TO GO THROUGH EVERYTHING IN ORDER.
- IT HAS SLOW KEY ITERATION. IF I WANNA GRAB ALL THE KEYS FROM A HASHTABLE I'LL HAVE TO GO THROUGH THE ENTIRE MEMORY SPACE. BU OLAYI KENDI HASHTABLE'IMIZDA GÖRDÜK KEYS() METODU.

More blogs