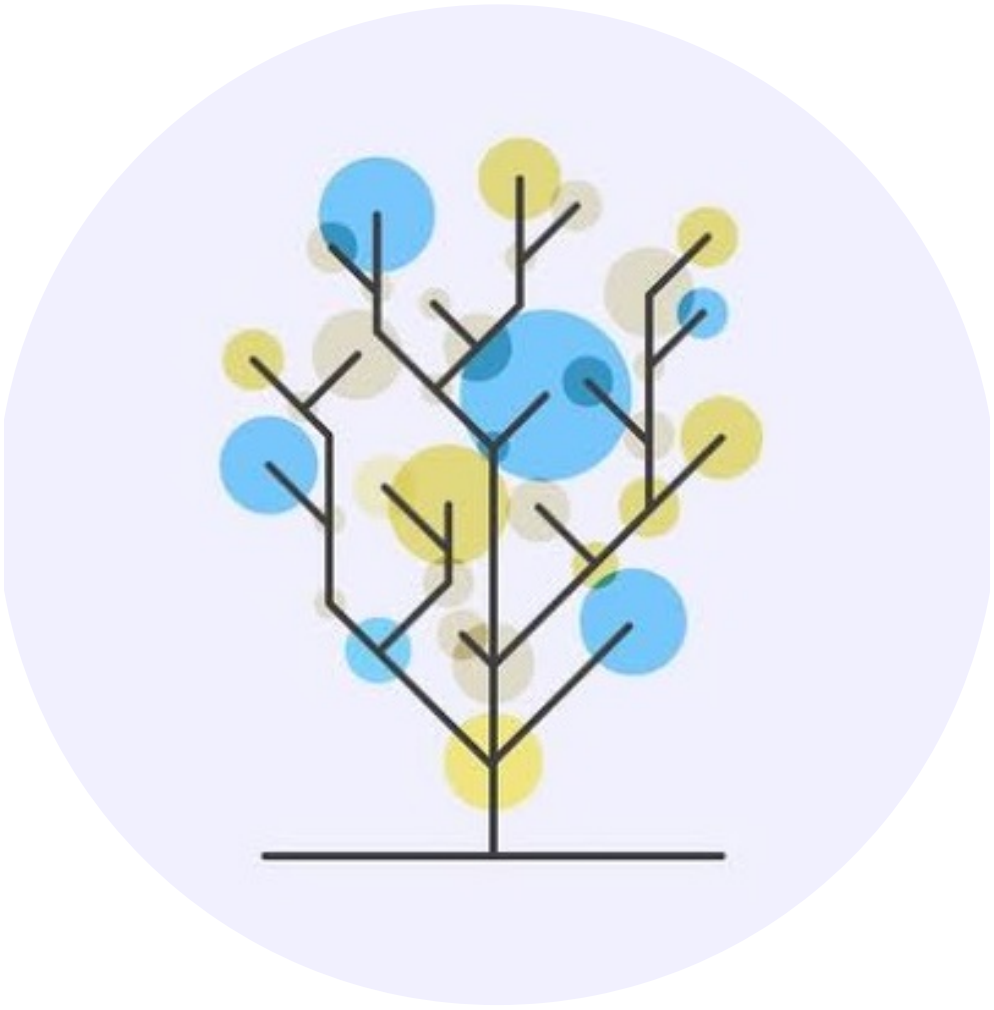
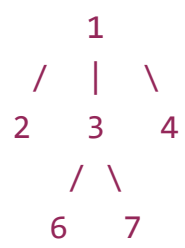


Trees

04.01.2020

Data Structures & Algorithms

Intro

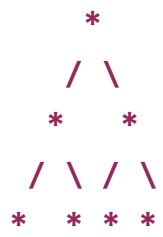
- A **tree** is a nonlinear **data structure**, compared to arrays, linked lists, stacks and queues which are linear **data structures**.
- 

- **Root** : 1
  Parent : 1, 3
  Child : 2,3,4,6,7
  Leaf : 2,4,6,7
  Sibling : 2,3,4 - 6,7

- As opposed to linked lists or arrays which are linears, trees can have zero or more child nodes!
- Every child descents from only one parent.
- Unidirectional parent-child relationship.
- Leaf nodes: very end nodes of a tree.

- We used the same principle as we used in linked lists. We have nodes that can contain any type of data.
- Linked lists are type of tree but with just one single path
- Nodes can only point their children

Binary Trees

- 

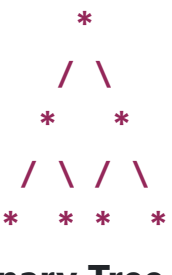
- **RULE 1:** EACH NODE CAN ONLY HAVE EITHER 0 1 OR 2 CHILDREN
- **RULE 2:** EACH CHILD CAN ONLY HAVE ONE PARENT

- Mesela aşağıdaki örnek bir Binary Tree değil! Çünkü root 3 node gösteriyor, binary tree olması için en fazla 2 göstermesi gerekir.
- 

- ```
  function BinaryTreeNode(value){

        this.value = value;
        this.left = null;
        this.right = null;

    }
  ```

- **Perfect Binary Tree**
  - Has everything filled in. That means all the leaf nodes are full. Bottom layer is compeletely full. A node has eihter 0 or 2 children.
  - 
- **Full Binary Tree**
  - A node has eihter 0 or 2 children.
  - 
- **Perfect Binary Trees are efficient:**
  - They have two properties:
    - 1 - Number of nodes at each level doubles as we move down the tree (1 - 2 - 4 - 8 - 2^n) where n = 0. , 1. levels.
    - 2- Son leveldaki node sayısı, kalan node sayısından 1 eksiktir. (1 + 2 = 4 -1)
    - Yani data'yı tree yapısı ile organize edersek datanın yarısı en alt levelda tutulacak.
  ```

- Bu tree yapısı ile datayı bielirli bir düzene göre dizebiliriz böylece daha sonra datayı bulması kolay olur.
- Bu yapı bize yeni bir BigO notation olanağı sağlar: **O(logN)**
- Binary Search Tree göreceğiz --> **Lookup**, **Insert**, **Delete** --> **O(logN)**

O(logN)

- Perfect Binary Trees göz önüne alınırsa:

- **Level 0** --> has --> **2^0** = 1 --> Nodes
  Level 1 --> has --> **2^1** = 2 --> Nodes
  Level 2 --> has --> **2^2** = 4 --> Nodes
  Level 3 --> has --> **2^3** = 8 --> Nodes
  .
  .
  .
  Level m --> has --> **2^m** --> Nodes

- Say, **h = total number of levels**
- Number of total nodes = **2^h -1** where **h = m + 1**
- **Ex :** 3 kat var ise (level 0, 1, 2) --> 2^3-1 = 7 nodes

- **Number of nodes** = **2^h -1** = almost **2^h**
- **log2 (Number of nodes) = h**
- `log2 (n) = h`

- O(logN)'deki log binary log yani log2

- YANİ O(logN) şu demek. Totalde diyelim ki 7 elemanlı bir perfect binary tree data structure'ım var. n = 7. Ben bunda bir işlem yaparken (**search**, **insert**, **delete**) her bir elemana tek tek bakmayacağım, bunun yerine her node için ya sağa ya sola gideceğim bu şekilde TÜM ELEMANLARA (7 tane) BAKMAK YERİNE YALNIZCA 3 ELEMENA (height=3) BAKARIM VE YETERLİ OLUR!
- BUNUN GENELLEŞTİRİLMİŞ FORMÜLÜ --> height = log2 (n) --> Yani işlemler **O(logN) time complexity**'ye sahip deriz.

- TREE'nin avantajını rehber'e bakıyormuşsun gibi düşün. Rehberde bir numara ararken tek tek tüm numaralara bakmam, ilgili başlıklara göre belirli yerlere bakarım bu da benim time complexity yükümü hafifletir.

Binary Search Tree (BST)

- Binary Search Tree is a subset of Binary Trees
- As the name suggest binary search trees are great for searching

- **RULE 1:** Sağa dallanan her child node, parentdan büyük olmalı. Sürekli sağa gidersem tutulan sayı sürekli artmalı. Sola doğru gidersem ise sürekli azalmalı.
- **RULE 2:** Since it is a binary tree, node can have up to two children.
- ```
          101
        /    \
  ```
  ```

```
      33      105
     / \      / \
    9  37   104 144
```

- Hashtable'da bir düzen yok, burada datalar bir düzene göre dizilir.

- **LOOKUP : O(logN)** : Let's say 37'yi arıyorum.
  > 37 101'den büyük mü? Hayır. O zaman sola git.
  > 37 33'den büyük mü? Evet. O zaman sağa git.
  > I find 37. I don't have to iterate through each node.
- **INSERT : O(logN)**
- **DELETE : O(logN)**

- In order to gain a visual understanding about BST check this link out.

Balanced vs Unbalanced Binary Search Trees

- Öyle bir case düşün ki mesela bir para koleysiyonundaki her parayı bir data olarak kaydediyorum diyelim. 5-10 tane küçük para var 1 kuruş, 25 kuruş, 50 kuruş vb. onun dışında hep büyük paralar 5 lira 10 lira 100 lira 200 lira 1000 10000 lira vb gibi. gitsin gerçekçi değil ama anlaşışması için. İlk başta küçük paraları eklersek bir tree oluşacak sonra gelen büyük paralar hep sağa sağa eklenecek. O zaman unbalanced tree olacak.

- UNBALANCED TREE ISTEMEYIZ CUNKU DUSUN n=1000 AMA 990 TANESİ SAĞA DİZİLMİŞ O ZAMAN BU BINARY SEARCH TREE OLMAKTAN ÇIKIP LINKED LIST OLUR. İŞLEMLER O(logN) yerine O(n) olmaya başlar! BU İSTENMEZ!!!

- UNBALANCE TREE'YI BALANCE ETMEK İÇİN ADVANCED ALOGORİTMALAR VAR. BUNLARDAN ALGORITMALAR KISMINDA BAHSEDILECEK.
  - AVL TREE
  - RED BLACK TREE
  - Bu iki tip tree otomatik olarak kendini rebalance eder.

Binary Search Tree Pros and Cons

- +It has a really good performance.
- +All operations are better than O(n) (average case. Worst case (unbalanced bst) için O(n) olur)
- +Ordered
- +Flexible size

- - No O(1) operation.

- **Array Comparison**
  - Search will be faster. O(logN) versus O(n): Iterate through the entire array. (If the array is unsorted).
  - Insert and Deletes are also faster than array. Unless the array is adding to the end. Otherwise array ave to shift all the indexes.

- **Hash Table Comparison**
  - Although hash tables allow us to INSERT and SEARCH at O(1). With binary search trees we have sorted data.

    - We also have this structure of parent child relationship that you won't be able to get too much with hash tables.

- BINARY SEARCH TREES AREN'T FASTEST FOR ANYTHING. ON AVERAGE AN ARRAY OR AND HASH TABLE WILL HAVE FASTER OPERATIONS.
- BALANCED BINARY SEARCH TREES GENEL OLARAK GAYET İYİ PERFORMANSA SAHİPTİR İLERDE DAHA DETAYLI BAHSEDECEĞİZ.

Implement BST

```
        //IMPLEMENT BINARY SEARCH TREES


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

class BinarySearchTree {
  constructor(){
    this.root = null;
  }
  insert(value){
    const newNode = new Node(value);
    if (this.root === null) {
      this.root = newNode;
    } else {
      let currentNode = this.root;
      while(true){
        if(value < currentNode.value){
          //Left
          if(!currentNode.left){
            currentNode.left = newNode;
            return this;
          }
          currentNode = currentNode.left;
        } else {
          //Right
          if(!currentNode.right){
            currentNode.right = newNode;
            return this;
          }
          currentNode = currentNode.right;
        }
      }
    }
  }
  lookup(value){
    if (!this.root) {
      return false;
    }
    let currentNode = this.root;
    while(currentNode){
      if(value < currentNode.value){
        currentNode = currentNode.left;
      } else if(value > currentNode.value){
        currentNode = currentNode.right;
      } else if (currentNode.value === value) {
        return currentNode;
      }
    }
    return null
  }
  remove(value) {
```

```
    if (!this.root) {
      return false;
    }
    let currentNode = this.root;
    let parentNode = null;
    while(currentNode){
      if(value < currentNode.value){
        parentNode = currentNode;
        currentNode = currentNode.left;
      } else if(value > currentNode.value){
        parentNode = currentNode;
        currentNode = currentNode.right;
      } else if (currentNode.value === value) {
        //We have a match, get to work!

        //Option 1: No right child:
        if (currentNode.right === null) {
          if (parentNode === null) {
            this.root = currentNode.left;
          } else {

            //if parent > current value, make current left child a child of parent
            if(currentNode.value < parentNode.value) {
              parentNode.left = currentNode.left;

            //if parent < current value, make left child a right child of parent
            } else if(currentNode.value > parentNode.value) {
              parentNode.right = currentNode.left;
            }
          }

        //Option 2: Right child which doesnt have a left child
        } else if (currentNode.right.left === null) {
          currentNode.right.left = currentNode.left;
          if(parentNode === null) {
            this.root = currentNode.right;
          } else {

            //if parent > current, make right child of the left the parent
            if(currentNode.value < parentNode.value) {
              parentNode.left = currentNode.right;

            //if parent < current, make right child a right child of the parent
            } else if (currentNode.value > parentNode.value) {
              parentNode.right = currentNode.right;
            }
          }

        //Option 3: Right child that has a left child
        } else {

          //find the Right child's left most child
          let leftmost = currentNode.right.left;
          let leftmostParent = currentNode.right;
          while(leftmost.left !== null) {
            leftmostParent = leftmost;
            leftmost = leftmost.left;
          }
```

```
        //Parent's left subtree is now leftmost's right subtree
        leftmostParent.left = leftmost.right;
        leftmost.left = currentNode.left;
        leftmost.right = currentNode.right;

        if(parentNode === null) {
          this.root = leftmost;
        } else {
          if(currentNode.value < parentNode.value) {
            parentNode.left = leftmost;
          } else if(currentNode.value > parentNode.value) {
            parentNode.right = leftmost;
          }
        }
      }
    return true;
    }
  }
}

const tree = new BinarySearchTree();
tree.insert(9)
tree.insert(4)
tree.insert(6)
tree.insert(20)
tree.insert(170)
tree.insert(15)
tree.insert(1)
tree.remove(170)
JSON.stringify(traverse(tree.root))

//     9
//  4     20
//1  6  15  170

function traverse(node) {
  const tree = { value: node.value };
  tree.left = node.left === null ? null : traverse(node.left);
  tree.right = node.right === null ? null : traverse(node.right);
  return tree;
}
```

Binary Heap

- Binary Tree ve bunun subset'i Binary Search Tree gördük.
- Binary Tree dışında Heap ve Trie olarak iki farklı çeşit tree yapısından söz etmek de mümkün.

- **BINARY HEAPS**
  - 101
    / \
    72 33

```
          / \     / \
         2  45   5   1
```

- **Binary Max Heap :** Every child is lower than its parent.
- **Binary Min Heap :** Exact opposite where the root nodes are the smallest.

- BU BST KADAR ORDERED DEGIL O YUZDEN **LOOKUP : O(n)**
- MESELA 1'İ ARIYORUM DİYELİM HEM SAĞA HEM SOLA BAKMAM LAZIM YOK MU, YİNE HEM SAĞA HEM SOLA SONUÇTA HEPSİNE BAKMIŞ OLUYORUM.
- BU TİP TREE'LERİN KENDİ KULLANIM ALANI VAR MESELA SADECE 33 YAŞINDAN BÜYÜK İNSANLARI BULMAK İSTİYORUM BU ŞEKİLDE ÇOK KOLAY BST İLE BAYA ZOR OLURDU.

- **LOOKUP : O(n)**
  INSERT : O(1): O(logN)
  DELETE : O(logN)

- LEFT TO RIGHT INSERTION YAPIYOR. DOLAYISIYLA UNBALANCED BINARY HEAP DIYE BIR SEY YOK. SUREKLI EN ALT KATMANA SOLDAN SAGA YERLESTIRIYOR BU YUZDEN INSERTION O(1) ANCAK TABİ BAZEN DEĞİŞİMLER KAYMALAR GEREKEBİLİYOR.
  - WE CAN IMPLEMENT BINARY HEAPS USING ARRAYS.
  - ONLY GUARANTEE THAT BINARY HEAP GIVES US IS THAT: PARENT IS ALWAYS GREATER THAN THE CHILDREN.

- **PRIORITY QUEUES**
  - BINARY HEAPS ARE VERY USEFULL FOR PRIORITY QUEUES
  - QUEUE İÇİN FIRST IN FIRST OUT DEMISTIK BİR LINE GIBI
  - PRIORITY QUEUE İÇİN FARKLI TİP DATALARIN ÖNCELİĞİ OLABİLİR YANİ SANKİ BİR LİNE VAR AMA VİPLER GELİNCE GEÇ BİLE GELSE ÖNCE ONLAR GEÇİYOR.
  - VEYA UÇAĞA BİNME SIRASI GİBİ DÜŞÜN:
    -
```
YOLCU  --->   KAPTAN --->  KAPTAN   --->  KAPTAN
                /            /  \          /   \
              YOLCU        YOLCU YOLCU   HOSTES  YOLCU
                                  /
                                YOLCU
```

- **REVIEW**
  -
```
     BST                                BINARY HEAPS
-----------                      ------------------
+Better than O(n)                +  Better than O(n)
+Ordered                         +  Priority
+Flexible Size                   +  Flexible Size
-No O(1) operation.              +  Fast insert
                                 -  Slow lookups
```

Trie

- A TRIE IS SPECIALIZED TREE USED IN SEARCHING MOST OFTEN WITH TEXT.
-
```
                NULL
              / / | \ \
             A  B --- Z
           / / \
          Ç  T  K
```

- Her harf altında farklı harfler var ve kelimeler oluşturuyorlar.
- O(length of the word)

More blogs