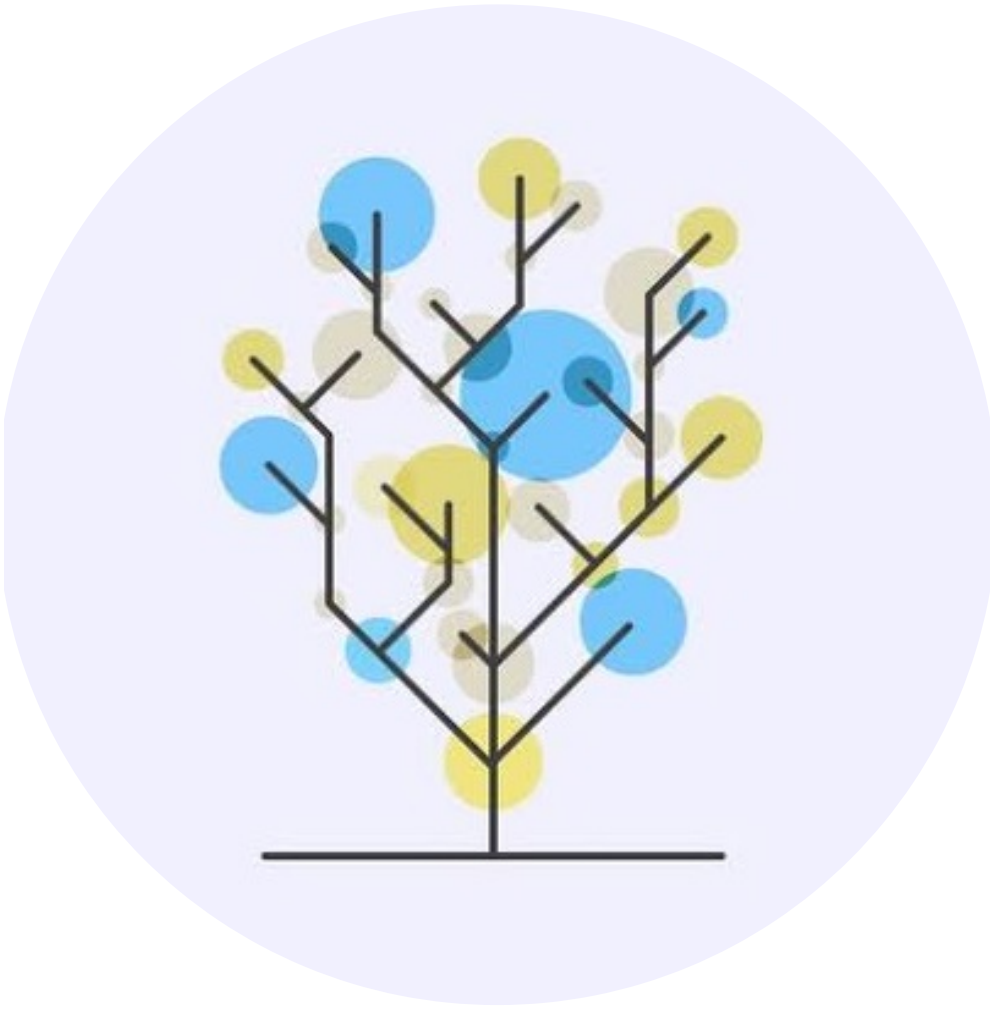


Searching

07.01.2020

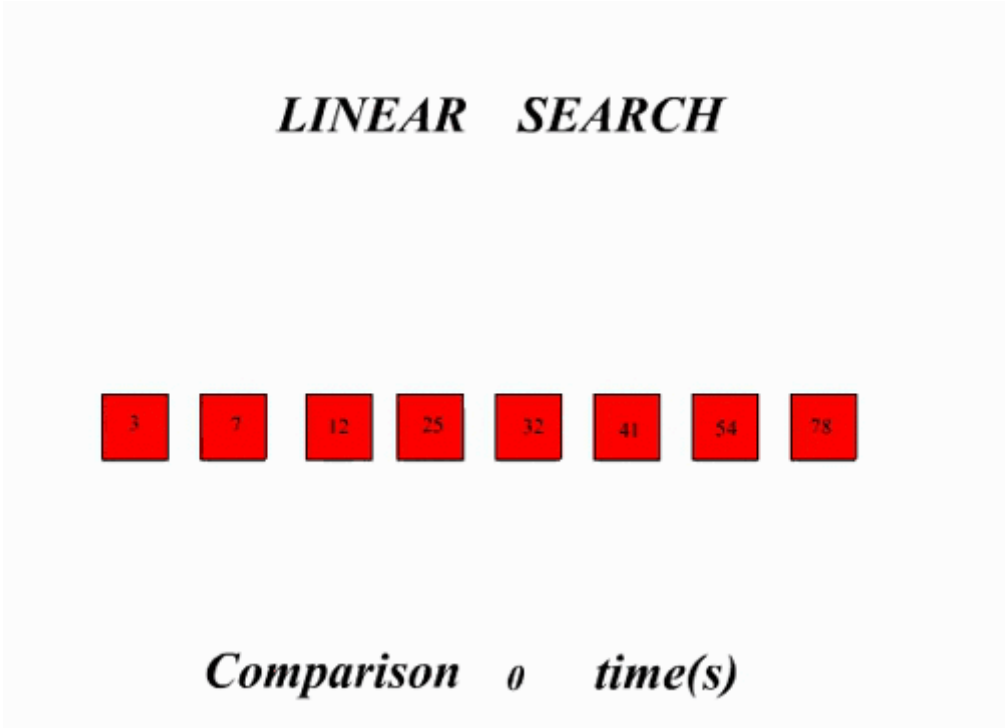
Data Structures & Algorithms

Intro

- Searching algoritmalarını her gün kullanıyoruz. Google, Youtube, bilgisayar içi aramalar buna örnek olarak verilebilr.
- Bu sectionda birkaç farklı searching algoritmasından bahsederek bir temel oluşturacağız.

Linear Search

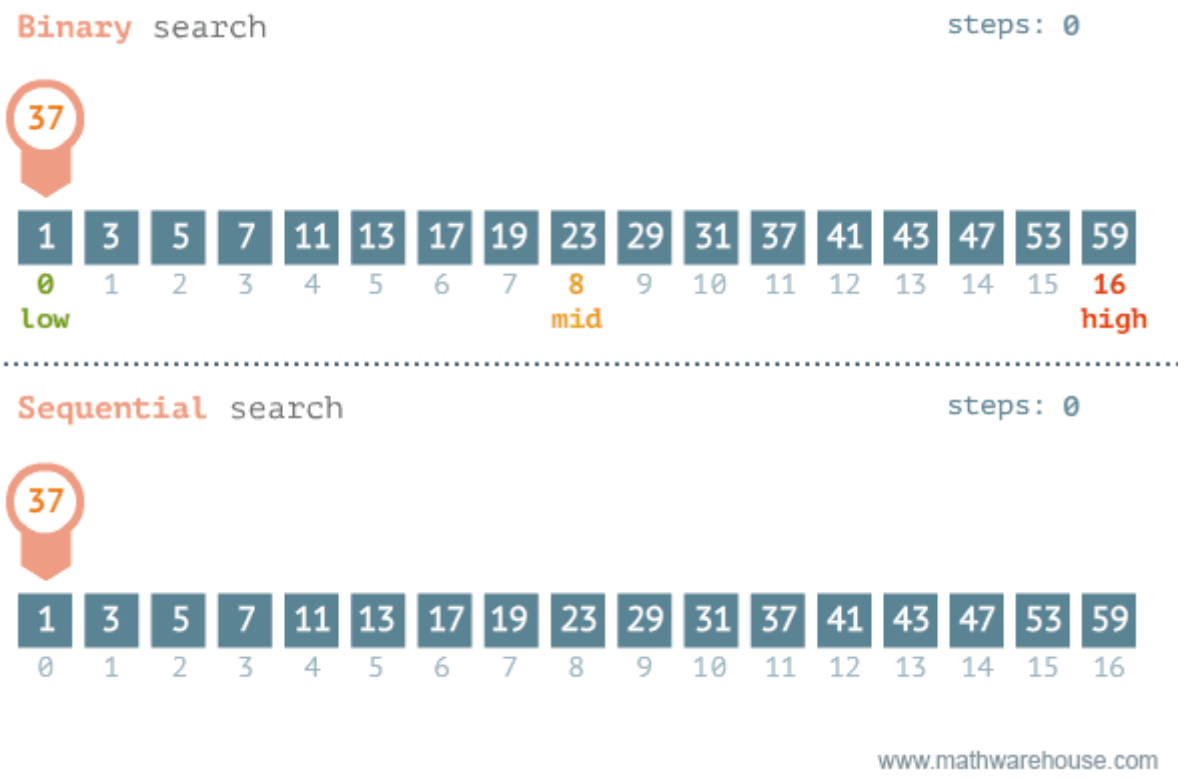
- Linear/Sequential Search is a method of finding a target value within the list.
- Yani temelde linear search için liste elemanları tek tek aranır.

- 

- Bu yüzden eğer aranan elman en başta ise **O(1)** , en sonda ise **O(n)** olur. Tabi bizim için worst case önemli olduğu için **O(n)** deriz.
- Javascript üzerindeki liste içi built-in arama metodları (`list.indexOf(element)` gibi) linear search kullanır.
- Google veya Facebook gibi büyük database işlemlerinde linear time(O(n)) bizi kurtarmaz, daha iyisi lazım.
- Peki daha iyi bir yol var mı? Görelim...

Binary Search

- Eğer listemiz sorted ise, tek tek tüm elemanları gezmeden arama yapabiliriz.
- Linear Search için her seferinde bir eleman kontrol edip eğer aranan eleman o değilse eliyorduk, burada ise tek bir kontrol ile elemanların yarısını eleyebiliriz bu yüzden daha iyi performansa sahip.
- Mesela aşağıdaki listede 37 aranıyor diyelim, önce listenin ortasındaki eleman kontrol edilir 23<37 olduğuna göre sol tarafı tamamen eleriz sonra kalan liste için aynısını yaparız. Bu şekilde 11 step yerine 3 step ile aramayı tamamladık.

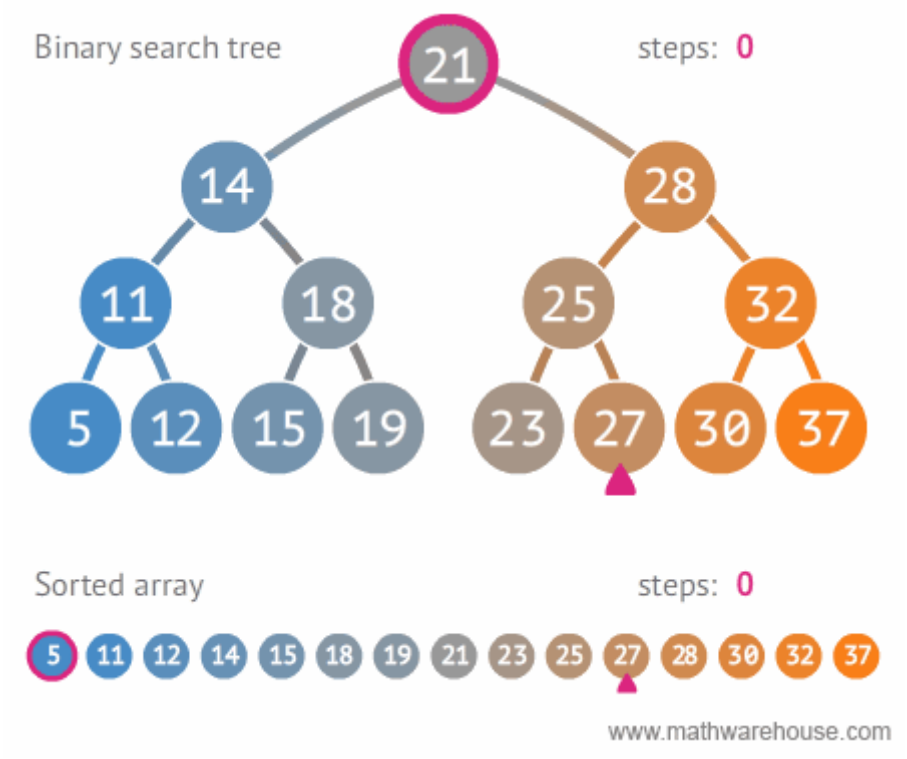
- 

- Zaten aynı mantıktan Binary Search Trees konusunda bahsetmiştik.
- Aslında bu aramada yapılan da aynı şey, listemizden bir binary search tree oluşturmuşuz gibi oluyor.

- Ama burada şöyle bir sıkıntı var, iyi de bu listeyi sıralamaya harcanak complexity de hesaba katılınca linear search daha iyi olmuyor mu?
- Şunu hesaba katmalıyız, tree konusunda bahsettiğimiz gibi veriyi array gibi bir linear datastructure yerine tree içinde tutmak ve veriyi kaydederken sıralı şekilde kaydetmek daha verimli.

- Yani verileri tree'ye kaydederken sıralarsak, önce unsorted list yaratıp sonra arama yapmaktan daha mantıklı olacaktır, daha iyi performans elde ederiz.

- Bu durumda Divide & Conquer dan bahsedebiliriz ve search algortihm **O(logN)** olur ki **O(n)**'den daha iyidir.

- 

Graph + Tree Traversals

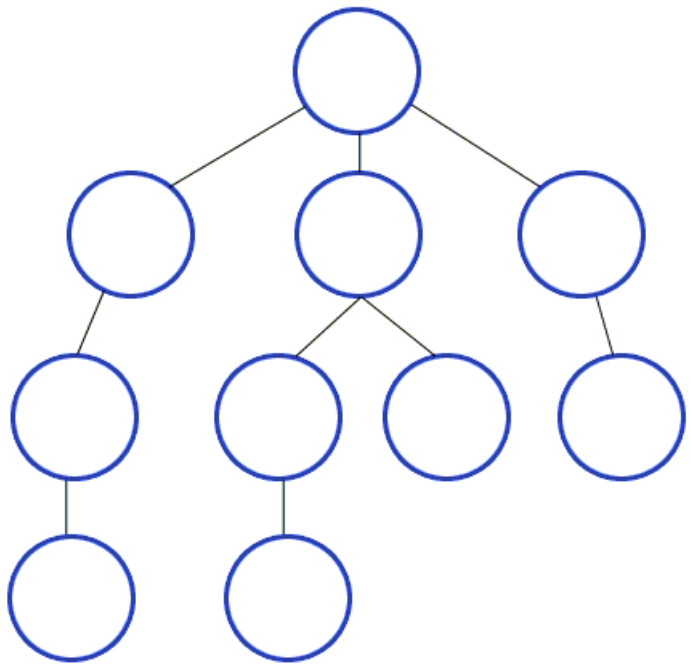
- There are times when we want to do traversal.
- Sometimes traversal and search means the same thing, but sometimes not.
- Mesela tüm node'lara yeni bir property eklemek isteyebilirim veya her birinin değerini 2 ile çarpmak isteyebilirim, veya belki tree unsorted şekilde, ya da sorted treemiz vardır ama kontrol etmek istiyoruzdur.
- Bu sebeplerden dolayı tüm node'ları gezmemiz gerekebilir, ki bu **O(n)**

- Tree veya Graph için böyle bir traversal'ı yapmamızın iki yolu vardır: **Breadth First Search** - **Depth First Search** bazen search yerine traversal da kullanılır.
- Bunların Big O su **O(n)** olur, çünkü dediğimiz gibi her node'u ziyaret etmeliyiz.

- Neden herşeyi liste içinde tutmuyoruz da Graph ve Tree ile uğraşıyoruz?
  - Çünkü böyle yapınca, array search **O(n) yerine O(logN)** search elde ediyoruz.
  - Ayrıca trees, **insert** veya **delete** için array'e göre **daha iyi** performans gösteriyor.
  - Hash tables'a göre avantajı da **order**! Hash tables için key'ler arası bir order söz konusu değildi.

Breadth First Search/Traversal

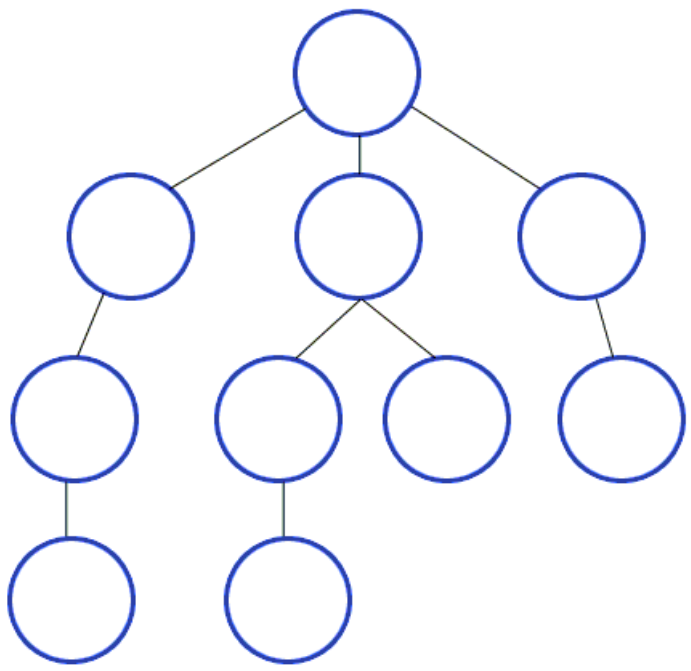
- Bunun olayı şu, tree içinde root'tan başlıyor, sonra bir alt level'a iniyor ve soldan sağa doğru gidiyor, bir alt level'a inerek devam ediyor, sonuçta tüm elemanları gezmiş oluyor.

- 

- Note that, **BFS uses additional memory** because it is necessary to track the child nodes of all the nodes on a given level while searching that level.
- This means we need to track every node and its children in order. Tabi ki bunun artısı ve eksisi var kodlarken bahsedilecek.

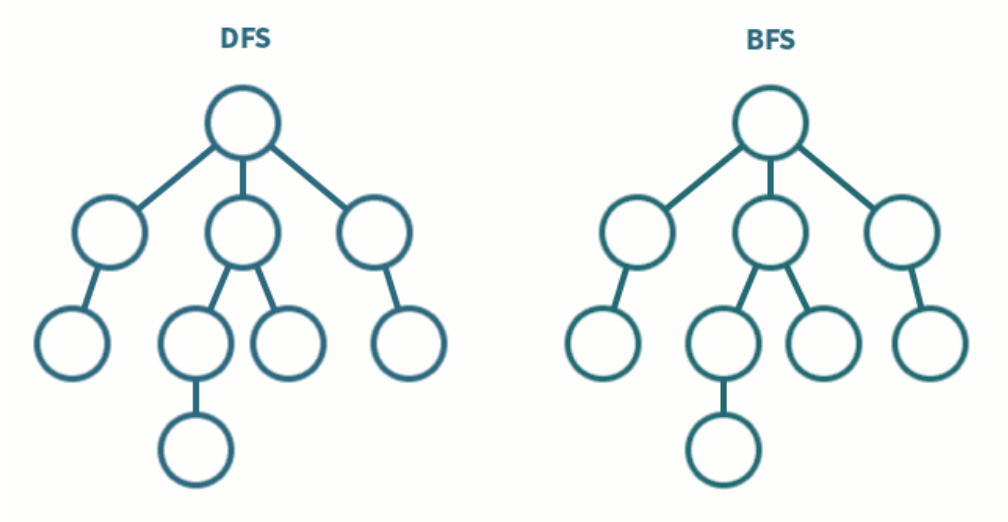
Depth First Search/Traversal

- Burada ise root'tan başlanır en soldaki branch seçilir ve dibe kadar gidilir, daha sonra bir sağdaki branch'e geçilir.

- 

- Lower memory requirement than BFS. Because it is not necessary to store all the child pointers at each level.

Pros and Cons of BFS and DFS

- 

- BFS ile yukarıdan aşağıya doğru yatak kademelerle ilerlenirken, DFS için soldan sağa doğru yatak kademelerle ilerleniyor.

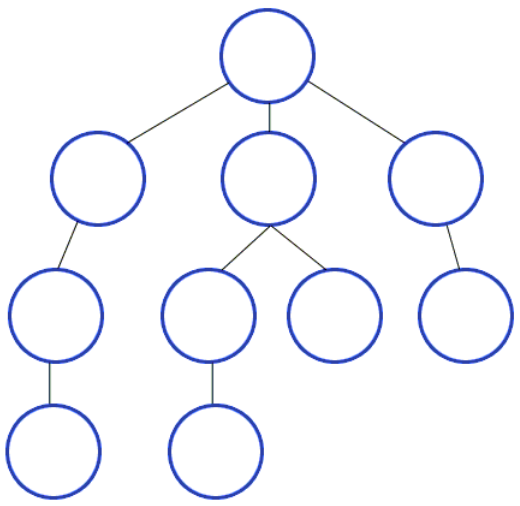
- Time Complexity for both BFS and DFS is **O(n)**

- **BFS:**
  - Is very good for finding the shortest path between a starting point and any other reachable node.
  - Because we always start with the root node and search for the closest nodes, and then we go further and further.
  - Graph traversal'dan bahsedince bu konu daha açık olacak.

- **DFS:**
  - Requires less memory than BFS.
  - It is good when asking the question "Does path exists?".
  - Downside is, it can get slow especially when tree or graph is really deep.
  - It is not very good at finding the shortest path.

- Bu açıklamalar'ı bir başlangıç olarak düşün tam anlaşılmaması normal. İlerleyen kısımlarda daha anlaşılır olacaktır. Daha sonra tekrar dönüp bakmak üzere bir kaç soruyu cevaplayalım:
  - **If you know a solution is not far from the root of the tree:**
    - BFS
  - **If the tree is very deep and solutions are rare:**
    - BFS (DFS will take long çünkü branch branch gidecek her branch çok derin)
  - **If the tree is very wide:**
    - DFS (BFS will need too much memory)
  - **If solutions are frequent but located deep in the tree:**
    - DFS
  - **Determining whether a path exists between two nodes:**
    - DFS
  - **Finding the shortest path:**
    - BFS

BFS Implementation

- 

- 2. level'a geçtiğinde ziyaret edilen her node'un children node'ları kaydedilmeli ki 2. level soldan sağa bitirilince başa dönülüp 3. node soldan sağa tamamlanmalı.
- Bu hafızada tutma işlemi için bir Queue kullanacağız.

- Olay şu, root node'un 2, 3 ve 4 ile doğrudan bağlantısı var bunu biliyoruz diyelim ki 1'den 2'ye gitti ama 2'den 3'e geçmesi için 1'in bağlantılarının hafızada tutulması lazım yoksa 2 ile 3 arasında bir bağlantı yok.
- Aynı şekilde 2,3, 4 gezilirken hepsinin child node'ları bir queue içinde tutulacak ki 4. node'u bitirince 5'e atlayabileyim yoksa 4 ile 5 arasında doğrudan bir bağlantı yok!
- BFS nin space complexity problemi buradan kaynaklanır.

- Kodu aşağıda oradan daha detaylı bu işlemi görebiliriz.
- Kısaca onu da açıklamak gerekirse, zaten Tree'yi daha önce yapmıştık buna BFS methodu ekliyoruz.
- currentNode = root olarak atanıyor, traversal boyunca geçilen elemanların tutulduğu bir liste tanımlanlıyor, bunun yanında child nodes'un tutulduğu queue tanımlanıyor.
- queue içine ilk olarak currentNode yani root veriliyor.
- Daha sonra bir while loop ile queue dolu olduğu sürece:
  - currentNode = queue'nun başındaki eleman olarak atanıyor
  - Listeye current node'un value'su ekleniyor.
  - CurrentNode'un soldan dağa child node'ları sırası ile queue içine ekleniyor. Yani ilk etapta 2 3 ve 4 eklenir.
  - Daha sonra başa dönüyor, böylece currentNode 1. elemanken 2. eleman oluyor ve bu sırada 2'nin childları queue'ya ekleniyor ...

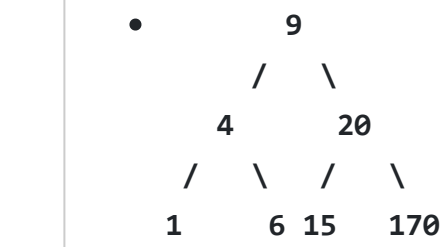
- Genelde BFS yukarıdaki şekilde gerçeklenir ama Recursive olarak da gerçekleştirmek mümkün.

```
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) {
        if(parentNode === null) {
          this.root = currentNode.left;
        } else {
          currentNode.right.left = currentNode.left;

          //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;
      }
    }
}
BreadthFirstSearch(){
  let currentNode = this.root;
  let list = [];
  let queue = [];
  queue.push(currentNode);

  while(queue.length > 0){
    currentNode = queue.shift();
    list.push(currentNode.value);
    if(currentNode.left) {
      queue.push(currentNode.left);
    }
    if(currentNode.right) {
      queue.push(currentNode.right);
    }
  }
  return list;
 }
}

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)

console.log('BFS', tree.BreadthFirstSearch());

//     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;
}
```

DFS Implementation

- DFS implementation için 3 farklı yol mevcut. Diyelim ki aşağıdaki gibi bir tree olsun:
- 

- Bu listeyi DFS ile traverse edersek traverse sonucunda oluşacak liste aşağıdaki 3 şekilden biri olabilir:
  - **Inorder:** 1, 4, 6, 9, 15, 20, 170 --> Küçükten büyüğe, yani inorder.
  - **Preorder:** 9, 4, 1, 6, 20, 15, 170 --> Parent'tan başlarız aşağı doğru ineriz soldan sağa.
  - **Postorder:** 1, 6, 4, 15, 170, 20, 9 --> Children comes before the parent, önce left hand side sonra right hand side.

- Koda bakınca bunlar daha rahat anlaşılabilir.
- DFS genelde Recursion ile implement edilir.

```
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) {
          if(parentNode === null) {
            this.root = currentNode.left;
          } else {
            currentNode.right.left = currentNode.left;

            //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;
      }
    }
}
BreadthFirstSearch(){
  let currentNode = this.root;
  let list = [];
  let queue = [];
  queue.push(currentNode);

  while(queue.length > 0){
    currentNode = queue.shift();
    list.push(currentNode.value);
    if(currentNode.left) {
      queue.push(currentNode.left);
    }
    if(currentNode.right) {
      queue.push(currentNode.right);
    }
  }
  return list;
}
BreadthFirstSearchR(queue, list) {
  if (!queue.length) {
    return list;
  }
  const currentNode = queue.shift();
  list.push(currentNode.value);

  if (currentNode.left) {
    queue.push(currentNode.left);
  }
  if (currentNode.right) {
    queue.push(currentNode.right);
  }

  return this.BreadthFirstSearchR(queue, list);
}
DFTPreOrder(currentNode, list) {
  return traversePreOrder(this.root, []);
}
DFTPostOrder(){
  return traversePostOrder(this.root, []);
}
DFTInOrder(){
  return traverseInOrder(this.root, []);
```

```
  }
}

function traversePreOrder(node, list){
  list.push(node.value);
  if(node.left) {
    traversePreOrder(node.left, list);
  }
  if(node.right) {
    traversePreOrder(node.right, list);
  }
  return list;
}

function traverseInOrder(node, list){
  if(node.left) {
    traverseInOrder(node.left, list);
  }
  list.push(node.value);
  if(node.right) {
    traverseInOrder(node.right, list);
  }
  return list;
}

function traversePostOrder(node, list){
  if(node.left) {
    traversePostOrder(node.left, list);
  }
  if(node.right) {
    traversePostOrder(node.right, list);
  }
  list.push(node.value);
  return list;
}


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))

console.log('BFS', tree.BreadthFirstSearch());
console.log('BFS', tree.BreadthFirstSearchR([tree.root], []))
console.log('DFSpre', tree.DFTPreOrder());
console.log('DFSin', tree.DFTInOrder());
console.log('DFSpost', tree.DFTPostOrder());

//     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;
}
```

Graph Traversals

- Tree traversal'da gördüğümüz şeyler burada da geçerli olacaktır.
- Sadece left ve right child node'lara bakmak yerine tüm children node'lara bakarız, onun dışında Tree ile Graph için search algorithm mantığı aynı kalır.
- Because trees are just simple a type of graph.
- Here is a link to better visualise what is going on.

- **BFS in Graphs**
  - Uses more memory.
  - Yukarıdaki link'e bakıp, orada biraz oynamalar yaparsak bunlar daha anlamlı olur.
  - Bu yüzden iki node arasında shortest path bulmakta iyidir, daha önce de bahsetmiştik.
  - Yakın node'lar ile başlar yavaşça derinleşir.

- **DFS in Graphs**
  - Maze solving için DFS kullanılır.
  - Fikir şu, bir yol seçip gidebildiğin kadar gidiyorsun, yol bittiği zaman geri dönüyorsun başka bir yoldan gidiyorsun.
  - Bu sebeple, shortest path bulmamıza yardımcı olmasa da "Does the path exists?" sorusuna cevap vermekte iyidir.
  - Less memory usage.
  - Ama derin graph yapılarında yavaşlayabilir. Deeper the graph, more recursion, more space complexity çünkü stack içine function calls dolduruluyor.

Dijkstra + Bellman-Ford Algorithms

- BFS shortest path bulmakta kullanılır demiştik ancak BFS her path'in weight'ini aynı kabul eder.
- Daha önce Graph'lerde gördüğümüz gibi weighted veya unweighted graph'lerde söz edebiliriz. Unweighted graph için BFS iyi çalışır.
- Ancak Weighted Graphs için başka algoritmalar kullanılır.
- Örneğin Google Maps düşünelim, bazı yollar diğerlerinden daha hızlı değil mi? Belki biri daha uzun, belki birinde trafik var, belki biri kapalı. O zaman map, weighted graph ile ifade edilecektir bu durumda BFS yerine diğer algoritmalar önem kazanır.

- Dijkstra algorithm shortest path bulmak için daha verimlidir ancak negative weight ile çalışmaz.

More blogs