

# Data Structure Basics

## Array

### Definition:

- Stores data elements based on an sequential, most commonly 0 based, index.
- Based on [tuples \(http://en.wikipedia.org/wiki/Tuple\)](http://en.wikipedia.org/wiki/Tuple) from set theory.
- They are one of the oldest, most commonly used data structures.

### What you need to know:

- Optimal for indexing; bad at searching, inserting, and deleting (except at the end).
- **Linear arrays**, or one dimensional arrays, are the most basic.
  - Are static in size, meaning that they are declared with a fixed size.
- **Dynamic arrays** are like one dimensional arrays, but have reserved space for additional elements.
  - If a dynamic array is full, it copies it's contents to a larger array.
- **Two dimensional arrays** have x and y indices like a grid or nested arrays.

### Big O efficiency:

- Indexing: Linear array:  $O(1)$ , Dynamic array:  $O(1)$
  - Search: Linear array:  $O(n)$ , Dynamic array:  $O(n)$
  - Optimized Search: Linear array:  $O(\log n)$ , Dynamic array:  $O(\log n)$
  - Insertion: Linear array: n/a Dynamic array:  $O(n)$
- 

## Linked List

### Definition:

- Stores data with **nodes** that point to other nodes.
  - Nodes, at its most basic it has one datum and one reference (another node).
  - A linked list *chains* nodes together by pointing one node's reference towards another node.

### What you need to know:

- Designed to optimize insertion and deletion, slow at indexing and searching.
- **Doubly linked list** has nodes that reference the previous node.
- **Circularly linked list** is simple linked list whose **tail**, the last node, references the **head**, the first node.
- **Stack**, commonly implemented with linked lists but can be made from arrays too.

- Stacks are **last in, first out** (LIFO) data structures.
- Made with a linked list by having the head be the only place for insertion and removal.
- **Queues**, too can be implemented with a linked list or an array.
  - Queues are a **first in, first out** (FIFO) data structure.
  - Made with a doubly linked list that only removes from head and adds to tail.

#### Big O efficiency:

- Indexing: Linked Lists:  $O(n)$
  - Search: Linked Lists:  $O(n)$
  - Optimized Search: Linked Lists:  $O(n)$
  - Insertion: Linked Lists:  $O(1)$
- 

## Hash Table or Hash Map

#### Definition:

- Stores data with key value pairs.
- **Hash functions** accept a key and return an output unique only to that specific key.
  - This is known as **hashing**, which is the concept that an input and an output have a one-to-one correspondence to map information.
  - Hash functions return a unique address in memory for that data.

#### What you need to know:

- Designed to optimize searching, insertion, and deletion.
- **Hash collisions** are when a hash function returns the same output for two distinct inputs.
  - All hash functions have this problem.
  - This is often accommodated for by having the hash tables be very large.
- Hashes are important for associative arrays and database indexing.

#### Big O efficiency:

- Indexing: Hash Tables:  $O(1)$
  - Search: Hash Tables:  $O(1)$
  - Insertion: Hash Tables:  $O(1)$
- 

## Heap

#### Definition

- A specialized tree-based data structure that satisfies the heap property.

#### What you need to know:

- if P is a parent node of C, then the value of P is  $\geq$  value of C (max heap), or  $\leq$  value of C (min heap).
- efficient implementation of priority queue

## Binary Tree

### Definition:

- Is a tree like data structure where every node has at most two children.
  - There is one left and right child node.

### What you need to know:

- Designed to optimize searching and sorting.
- A **degenerate tree** is an unbalanced tree, which if entirely one-sided is essentially a linked list.
- They are comparably simple to implement than other data structures.
- Used to make **binary search trees**.
  - A binary tree that uses comparable keys to assign which direction a child is.
  - Left child has a key smaller than it's parent node.
  - Right child has a key greater than it's parent node.
  - There can be no duplicate node.
  - Because of the above it is more likely to be used as a data structure than a binary tree.

### Big O efficiency:

- Space: Binary Search Tree: Average:  $O(n)$  Worst:  $O(n)$
- Indexing: Binary Search Tree: Average:  $O(\log n)$  Worst:  $O(n)$
- Search: Binary Search Tree: Average:  $O(\log n)$  Worst:  $O(n)$
- Insertion: Binary Search Tree: Average:  $O(\log n)$  Worst:  $O(n)$

### Search

```
def search(root, key):
    if root is None or root.val == key:
        return root
    elif root.val < key:
        return search(root.right, key)
    else:
        return search(root.left, key)
```

### Insert

```
def insert(root, node):
    if root is None:
        root = node
    elif root.val < node.val:
        if root.right is None:
            root.right = node
        else:
            insert(root.right, node)
    else:
        if root.left is None:
            root.left = node
        else:
            insert(root.left, node)
```

---

## Red-Black Tree

### Definition

- Is a self-balancing BST where every node
  - has a color either red or black
  - root of tree is always black
  - there are no two adjacent red nodes
  - every path from root to a None node has the same number of black nodes

### What you need to know:

- Binary tree has worst case of  $O(n)$  for a skewed tree.
- Red-black tree make sure the height of the tree remains  $O(\log n)$  after every insertion and deletion, then it is guaranteed to have an upper bound of  $O(\log n)$  for all these operations.
- The height of the tree is always  $O(\log n)$
- **Black height** is number of black nodes on a path from a node to a leaf. Leaf nodes are also counted black nodes. A node of height  $h$  has black height  $\geq h / 2$ .
- Color of a None node is considered as black

### Big O efficiency:

- Space: Red Black Tree: Average:  $O(n)$  Worst:  $O(n)$
- Indexing: Red Black Tree: Average:  $O(\log n)$  Worst:  $O(\log n)$
- Search: Red Black Tree: Average:  $O(\log n)$  Worst:  $O(\log n)$
- Insertion: Red Black Tree: Average:  $O(\log n)$  Worst:  $O(\log n)$

### Implementation

- Use two tools to do balancing:
  1. recoloring
  2. rotation
- Recolor first, if does not work, then rotate

## Insertion

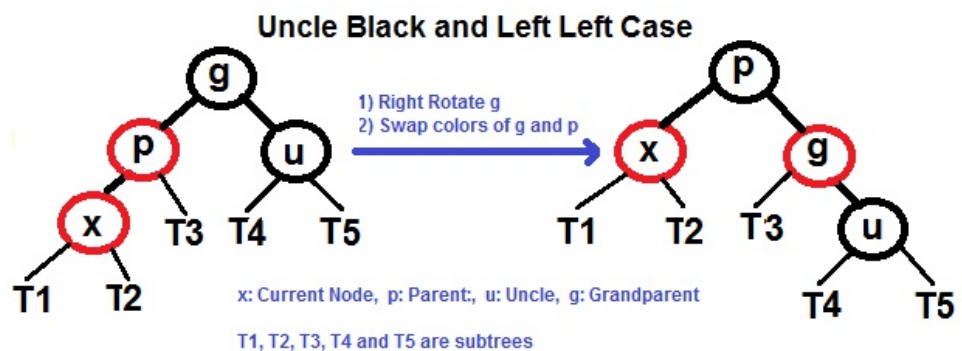
1. perform standard BST insertion and make the color of newly inserted nodes **x** as red
2. if x is root, change color of x as black
3. if color of x's parent is not black or x is not root

1. if x's uncle is red

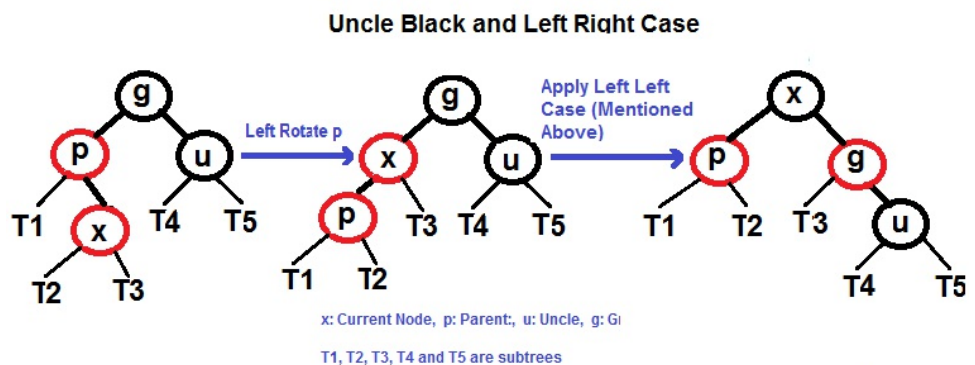
1. change color of parent and uncle as black
2. color of grandparent as red
3. change x = x's grandparent, repeat steps 2 and 3 for new x

2. if x's uncle is black, then there can be four configurations for x

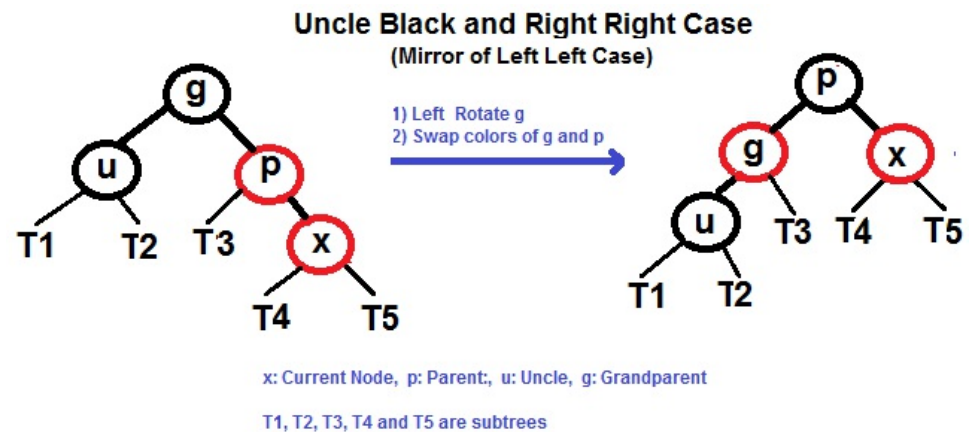
1. left left case (p is left child of g and x is left child of p)



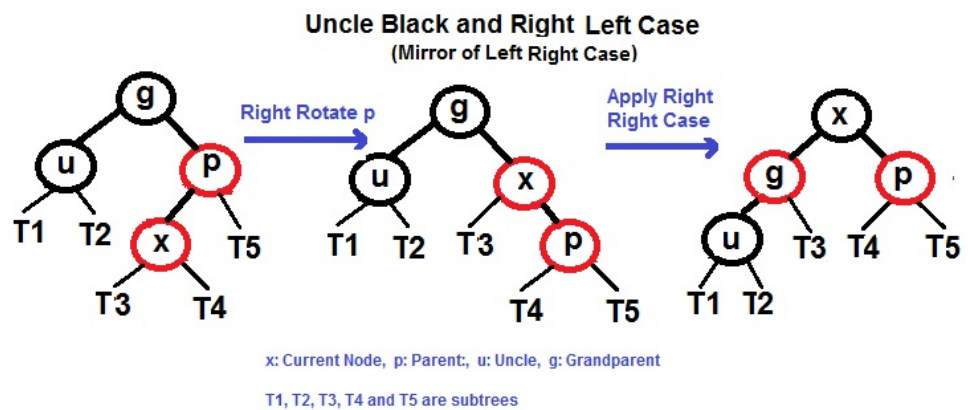
2. left right case (p is left child of g and x is right child of p)



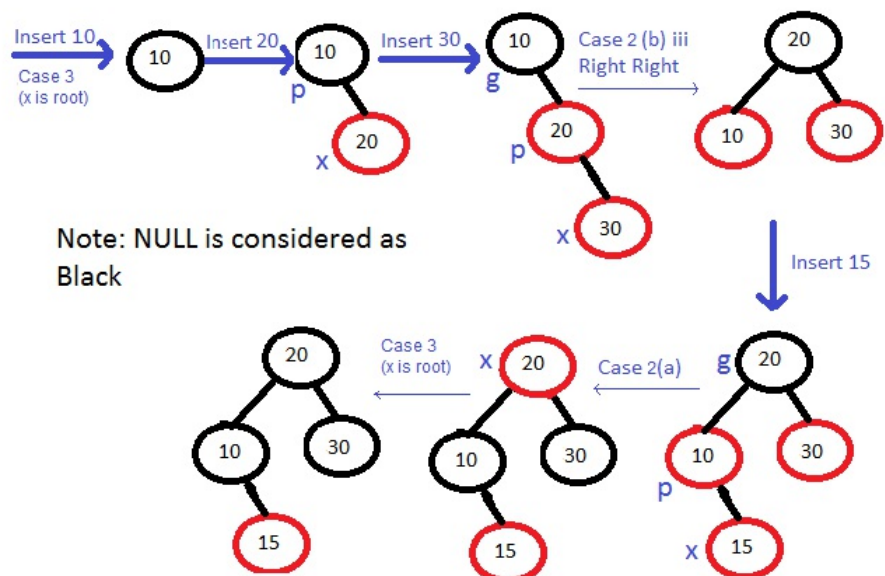
3. right right case (mirror of case a)



4. right left case (mirror of case c)



Insert 10, 20, 30 and 15 in an empty tree



## Deletion

1. Perform standard BST delete. Let **v** be the node to be deleted and **u** be the child that replaces **v**.
2. if either **u** or **v** is red: mark the replaced child as black
3. if both **u** and **v** are black:
  1. color **u** as double black. Then we have to reduce to convert this double black to single black
  2. do while current node **u** is double black and is not root
    1. if sibling **s** is black and at least one of sibling's children is red perform rotation(s)
    2. if sibling is black and its both children are black, perform recolouring and recur for the parent if parent is black. If parent was red then we don't need to recur for parent, simply make parent black.
    3. if sibling is red, perform a rotation to move old sibling up, recolour the old sibling and parent. The new sibling is always black.
  3. if **u** is root, make it single black and return.

# AVL Tree

## Definition

- A self-balancing BST where the difference between heights of left and right subtrees cannot be more than one for all nodes.

## What you need to know:

- The height of an AVL tree is always  $O(\log n)$ .

## Big O efficiency:

- Space: AVL Tree: Average:  $O(n)$  Worst:  $O(n)$
- Indexing: AVL Tree: Average:  $O(\log n)$  Worst:  $O(\log n)$
- Search: AVL Tree: Average:  $O(\log n)$  Worst:  $O(\log n)$
- Insertion: AVL Tree: Average:  $O(\log n)$  Worst:  $O(\log n)$

## Implementation

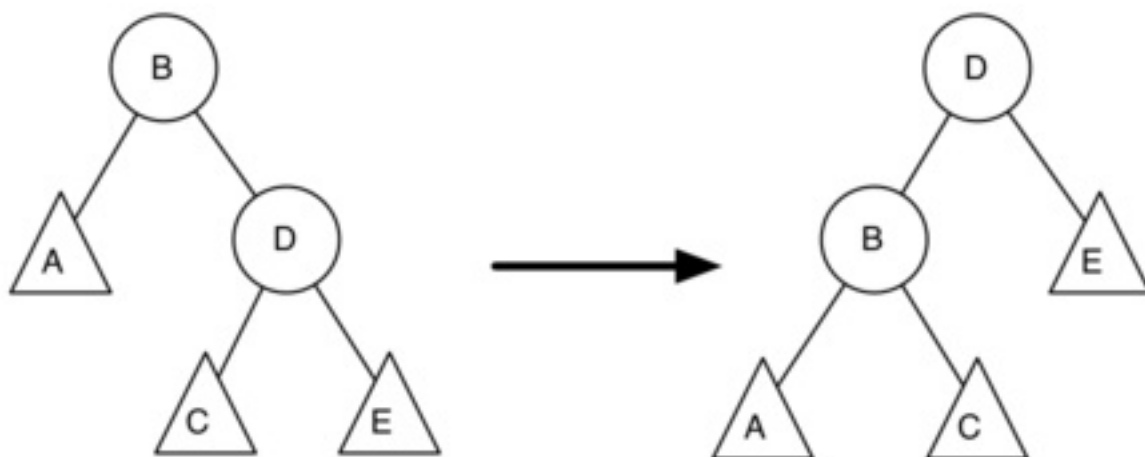
```
def insert(key, value, node):
    if key < node.key:
        if node.left:
            insert(key, value, node.left)
        else:
            node.left = TreeNode(key, value, parent=node)
            update_balance(node.left)
    else:
        if node.right:
            insert(key, val, node.right)
        else:
            node.right = TreeNode(key, val, parent=node)
            update_balance(node.right)
```

```
def update_balance(node):
    if node.balance_actor > 1 or node.balance_actor < -1:
        rebalance(node)
        return
    if not node.parent:
        if node.is_left_child():
            node.parent.balanceFactor += 1
        elif node.is_right_child():
            node.parent.balanceFactor -= 1

    if node.parent.balanceFactor != 0:
        update_balance(node.parent)
```

```
def rebalance(node):
    if node.balanceFactor < 0:
        if node.right.balanceFactor > 0:
            rotate_right(node.right)
            rotate_left(node)
        else:
            rotate_left(node)
    elif node.balanceFactor > 0:
        if node.left.balanceFactor < 0:
            self.rotate_left(node.left)
            self.rotate_right(node)
        else:
            self.rotate_right(node)
```

```
def rotate_left(node):
    new = node.right
    node.right = new.left
    if new.left != None:
        new.left.parent = node
    new.parent = node.parent
    if not node.parent: # node is root
        self.root = new
    else:
        if node.isLeftChild():
            node.parent.left = new
        else:
            node.parent.right = new
    new.left = node
    node.parent = new
    node.balanceFactor = node.balanceFactor + 1 - min(new.balanceFactor, 0)
    new.balanceFactor = new.balanceFactor + 1 + max(node.balanceFactor, 0)
```



### Red-Black Tree vs AVL Tree

- AVL tree are more balanced compare to red-black trees, but may causes more rotations during insertion and deletion.
- AVL tree is better in search, red-black tree is better in insertion and



deletion.


---

## Trie-Tree

### Definition:

- An search tree that has ordered data structure that is used to store a dynamic set of associative array where the keys are usually strings

### What you need to know:

- Unlike a BST, no node in the tree stores the key associated with that node
  - Its position in the tree defines the key with which it is associated
- 
- Can use to replace hash table, has faster worst case look up time  $O(m)$ , where  $m$  is the length of a search string, compare to  $O(n)$  for an imperfect hash table
  - Common application is storing a predictive text or autocomplete dictionary, such as found on mobile. It takes advantage of a trie's ability to quickly search, insert and delete entries.
  - FSM (deterministic acyclic finite state automaton) is better if only storing words.
- 

## K-ary-Tree

### Definition:

- A rooted tree in which each node has no more than  $k$  children.

### What you need to know:

- Binary tree is a special case of K-ary tree with  $k = 2$ .
  - For a  $k$ -ary tree with height  $h$ , it has maximum of  $k^h$  number of leaves.
  - The height  $h$  of a  $k$ -ary tree does not include the root node.
- 

## Search Basics

### Breadth First Search

#### Definition:

- An algorithm that searches a tree (or graph) by searching levels of the tree first, starting at the root.
  - It finds every node on the same level, most often moving left to right.
  - While doing this it tracks the children nodes of the nodes on the current level.
  - When finished examining a level it moves to the left most node on the next level.

- The bottom-right most node is evaluated last (the node that is deepest and is farthest right of it's level).

#### What you need to know:

- Optimal for searching a tree that is wider than it is deep.
- Uses a queue to store information about the tree while it traverses a tree.
  - Because it uses a queue it is more memory intensive than **depth first search**.
  - The queue uses more memory because it needs to stores pointers

#### Big O efficiency:

- Search: Breadth First Search:  $O(|E| + |V|)$
  - E is number of edges
  - V is number of vertices
- 

## Depth First Search

#### Definition:

- An algorithm that searches a tree (or graph) by searching depth of the tree first, starting at the root.
  - It traverses left down a tree until it cannot go further.
  - Once it reaches the end of a branch it traverses back up trying the right child of nodes on that branch, and if possible left from the right children.
  - When finished examining a branch it moves to the node right of the root then tries to go left on all it's children until it reaches the bottom.
  - The right most node is evaluated last (the node that is right of all it's ancestors).

#### What you need to know:

- Optimal for searching a tree that is deeper than it is wide.
- Uses a stack to push nodes onto.
  - Because a stack is LIFO it does not need to keep track of the nodes pointers and is therefore less memory intensive than breadth first search.
  - Once it cannot go further left it begins evaluating the stack.

#### Big O efficiency:

- Search: Depth First Search:  $O(|E| + |V|)$
- E is number of edges
- V is number of vertices

## Breadth First Search Vs. Depth First Search

- The simple answer to this question is that it depends on the size and shape of the tree.
  - For wide, shallow trees use Breadth First Search

- For deep, narrow trees use Depth First Search

#### **Nuances:**

- Because BFS uses queues to store information about the nodes and its children, it could use more memory than is available on your computer. (But you probably won't have to worry about this.)
  - If using a DFS on a tree that is very deep you might go unnecessarily deep in the search. See [xkcd \(http://xkcd.com/761/\)](http://xkcd.com/761/) for more information.
  - Breadth First Search tends to be a looping algorithm.
  - Depth First Search tends to be a recursive algorithm.
- 

## **Efficient Sorting Basics**

### **Merge Sort**

#### **Definition:**

- A comparison based sorting algorithm
  - Divides entire dataset into groups of at most two.
  - Compares each number one at a time, moving the smallest number to left of the pair.
  - Once all pairs sorted it then compares left most elements of the two leftmost pairs creating a sorted group of four with the smallest numbers on the left and the largest ones on the right.
  - This process is repeated until there is only one set.

#### **What you need to know:**

- This is one of the most basic sorting algorithms.
- Know that it divides all the data into as small possible sets then compares them.

#### **Big O efficiency:**

- Best Case Sort: Merge Sort:  $O(n)$
  - Average Case Sort: Merge Sort:  $O(n \log n)$
  - Worst Case Sort: Merge Sort:  $O(n \log n)$
  - Number of comparisons: Merge Sort:  $O(n \log n)$
- 

### **Quicksort**

#### **Definition:**

- A comparison based sorting algorithm
  - Divides entire dataset in half by selecting the average element and putting all smaller elements to the left of the average.
  - It repeats this process on the left side until it is comparing only two

- elements at which point the left side is sorted.
  - When the left side is finished sorting it performs the same operation on the right side.
- Computer architecture favors the quicksort process.

### What you need to know:

- While it has the same Big O as (or worse in some cases) many other sorting algorithms it is often faster in practice than many other sorting algorithms, such as merge sort.
- Know that it halves the data set by the average continuously until all the information is sorted.

### Big O efficiency:

- Best Case Sort: Merge Sort:  $O(n)$
- Average Case Sort: Merge Sort:  $O(n \log n)$
- Worst Case Sort: Merge Sort:  $O(n^2)$

### Merge Sort vs. Quicksort

- Merge Sort divides the set into the smallest possible groups immediately then reconstructs the incrementally as it sorts the groupings.
- Quicksort continually divides the set by the average, until the set is recursively sorted.
- Quicksort is significantly faster in practice than other  $O(n \log n)$  algorithms, because its inner loop can be efficiently implemented on most architectures, and in most real-world data, it is possible to make design choices that minimize the probability of requiring quadratic time. Additionally, quicksort tends to make excellent usage of the memory hierarchy, taking perfect advantage of virtual memory and available caches. Although quicksort is not an in-place sort and uses auxiliary memory, it is very well suited to modern computer architectures.
- Sorting a LinkedList is likely to be faster with merge sort since it makes fewer total comparisons and is not affected by a poor pivot choice.
- Merge sort **worst case** use about 39% less comparisons than quicksort's **average case**.
- Merge sort is well suited for data that can only be accessed sequentially (e.g. LinkedList).

## Bubble Sort

### Definition:

- A comparison based sorting algorithm
  - It iterates left to right comparing every couplet, moving the smaller element to the left.
  - It repeats this process until it no longer moves and element to the left.

### What you need to know:

- While it is very simple to implement, it is the least efficient of these three sorting methods.
- Know that it moves one space to the right comparing two elements at a time and moving the smaller one to left.

### Big O efficiency:

- Best Case Sort: Merge Sort:  $O(n)$
- Average Case Sort: Merge Sort:  $O(n^2)$
- Worst Case Sort: Merge Sort:  $O(n^2)$

## Basic Types of Algorithms

### Recursive Algorithms

#### Definition:

- An algorithm that calls itself in its definition.
  - **Recursive case** a conditional statement that is used to trigger the recursion.
  - **Base case** a conditional statement that is used to break the recursion.

#### What you need to know:

- **Stack level too deep** and **stack overflow**.
  - If you've seen either of these from a recursive algorithm, you messed up.
  - It means that your base case was never triggered because it was faulty or the problem was so massive you ran out of RAM before reaching it.
  - Knowing whether or not you will reach a base case is integral to correctly using recursion.
  - Often used in Depth First Search

### Iterative Algorithms

#### Definition:

- An algorithm that is called repeatedly but for a finite number of times, each time being a single iteration.
  - Often used to move incrementally through a data set.

#### What you need to know:

- Generally you will see iteration as loops, for, while, and until statements.
- Think of iteration as moving one at a time through a set.
- Often used to move through an array.

### Recursion Vs. Iteration

- The differences between recursion and iteration can be confusing to

distinguish since both can be used to implement the other. But know that,

- Recursion is, usually, more expressive and easier to implement.
- Iteration uses less memory.
- **Functional languages** tend to use recursion. (i.e. Haskell)
- **Imperative languages** tend to use iteration. (i.e. Ruby)
- Recursive functions have to keep the function records in memory and jump from one memory address to another to be invoked to pass parameters and return values. That makes them very bad performance wise.

### Pseudo Code of Moving Through an Array (this is why iteration is used for this)

Recursion	Iteration
-----	-----
recursive method (array, n)	iterative method (array)
if array[n] is not nil	for n from 0 to size of array
print array[n]	print(array[n])
recursive method(array, n+1)	
else	
exit loop	

## Greedy Algorithm

### Definition:

- An algorithm that, while executing, selects only the information that meets a certain criteria.
- The general five components:
  - A candidate set, from which a solution is created.
  - A selection function, which chooses the best candidate to be added to the solution.
  - A feasibility function, that is used to determine if a candidate can be used to contribute to a solution.
  - An objective function, which assigns a value to a solution, or a partial solution.
  - A solution function, which will indicate when we have discovered a complete solution.

### What you need to know:

- Used to find the optimal solution for a given problem.
- Generally used on sets of data where only a small proportion of the information evaluated meets the desired result.
- Often a greedy algorithm can help reduce the Big O of an algorithm.

### Pseudo Code of a Greedy Algorithm to Find Largest Difference of any Two

Numbers in an Array.

```
greedy algorithm (array)
  var largest difference = 0
  var new difference = find next difference (array[n], array[n+1])
  largest difference = new difference if new difference is > largest difference
  repeat above two steps until all differences have been found
  return largest difference
```

This algorithm never needed to compare all the differences to one another, saving it an entire iteration.

---

## Dijkstra algorithm

### Definition:

- An algorithm for finding the shortest paths between nodes in a graph.

### What you need to know:

- Known as uniform cost search, formulated as an instance of BFS.

### Algorithm:

- See [YouTube \(https://www.youtube.com/watch?v=\\_IHSawdgXpl\)](https://www.youtube.com/watch?v=_IHSawdgXpl)
1. Mark all nodes unvisited and create a set of unvisited nodes
  2. Assign every node a tentative distance value. Set zero to the initial node and infinite for all other nodes
  3. For the current node, consider all of its unvisited neighbours and calculate their tentative distance through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.
  4. After considering all of the neighbours, mark the current node as visited and remove it from the unvisited set. A visited node will never be visited again.
  5. Move to the next unvisited node with the smallest tentative distance and repeat steps 2 to 5.
  6. Stop the algorithm when the target node has been visited or all nodes have been visited.

### Big O efficiency:

- Time complexity  $O(|E| + |V| \log |V|)$  with heap
- 

## A\*

### Definition:

- A widely used in path-finding and graph traversal, the process of plotting an efficiently directly path between multiple points.

### What you need to know:

- choose the path with the minimum heuristic value  $f(n) = g(n) + h(n)$

- $g(n)$  = path cost to node  $n$
  - $h(n)$  = estimated distance to goal from node  $n$
- condition on  $h(n)$ 
  - admissible:  $h(n) \leq h^*(n)$  the cost of an optimal path, never over-estimates
  - monotonic:  $h(n_1) \leq c(n_1, a, n_2) + h(n_2)$

**Big O efficiency:**

- Time complexity  $O(|E| + |V| \log |V|)$  with heap
-