CS Notes

Huu Duc Nguyen M.Sc.

20 January 2023

Contents

Abbreviations

a.k.a.	also known as
vs.	versus
CS	conditioned stimuli
CS	Computer Science
DS	Data Structure
DSA	Data Structure & Algorithms
ADT	Abstract Data Type
BST	Binary Search Tree
BBST	Balanced Binary Search Tree
DFS	Depth First Search
BFS	Breadth First Search
LIFO	Last In First Out
FIFO	First In First Out
TDD	Test Driven Development
CI	Continuous Integration

1 Introduction

This is my notes for Data Structure & Algorithms (DSA) in Computer Science (CS).

1.1 Definitions

- A Data Structure (DS) is a way of organizing data so that it can be used in an efficient way.
  - Essential for creating fast and powerful algorithms
  - Help to manage and organize data
  - Make code cleaner and easier to understand
- An Abstract Data Type (ADT) is an abstraction of a DS, which provides only the interface. It doesn't give specific details about how it should be implemented or in what programming language. E.g.:
  - List: Dynamic array, linked list
  - Queue: linked list based queue, array based queue, stack based queue
  - Map: tree map, hash map / hash table
  - Vehicle: bike, car, truck

1.2 Computational Complexity Analysis

Big-O notation $\mathcal{O}$ is used to give an upper bound of the complexity in the worst case. Sorted computational complexity in increasing order:

Constant time	$\mathcal{O}(1)$
Logarithmic time	$\mathcal{O}(\log N)$
Linear time	$\mathcal{O}(N)$
Linearithmic time	$\mathcal{O}(N \log N)$
Quadric time	$\mathcal{O}(N^2)$
Cubic time	$\mathcal{O}(N^3)$
Exponential time	$\mathcal{O}(a^N)$, $a > 1$
Factorial time	$\mathcal{O}(N!)$

Properties:

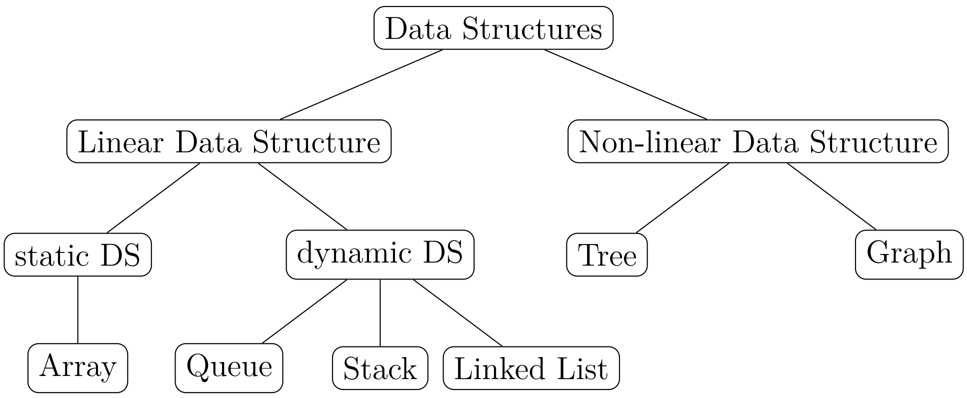
- Only the most dominant term matters
- $\mathcal{O}(N + c) = \mathcal{O}(N)$
- $\mathcal{O}(cN) = \mathcal{O}(N)$, $c > 0$

1.3 References

- The Missing Semester of Your CS Education
- Refactoring Guru
- Google | Tech Dev Guide | Data Structures & Algorithms
- GeeksForGeeks - Data structures
- TutorialsPoint - Data structures and Algorithms
- Data structures by William Fiset

2 Data Structures

2.1 Introduction



2.2 Array

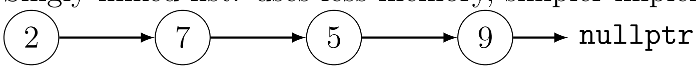
- Includes: static and dynamic arrays
- Usages:
  - Storing and accessing sequential data
  - Lookup tables and inverse lookup tables
  - Used in dynamic programming to cache answers to sub-problems
- Complexity

	Static array	Dynamic array
Access	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Search	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Insertion	N/A	$\mathcal{O}(N)$
Appending	N/A	$\mathcal{O}(1)$
Deletion	N/A	$\mathcal{O}(N)$

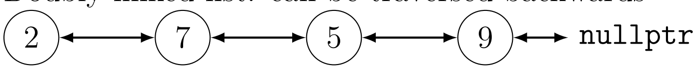
2.3 Linked List

- A linked list is a sequential list of nodes that hold data which point to other nodes
- Comparing with static array
  - Pros:
    * a linked list is a dynamic array (you don't have to specify the upper capacity in the beginning)

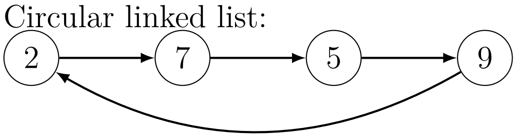
    * insertion and deletion of elements are easier.
  - Cons:
    * No random or direct access
    * Extra memory
- Terminology:
  - Head: the first node in a linked list
  - Tail: the last node
  - Pointer: reference to another node
  - Node: an object containing data and pointer(s)
- In implementation, ***always maintain*** pointers to the head/tail for quick addition/removal
- Classification
  - Singly linked list: uses less memory, simpler implementation



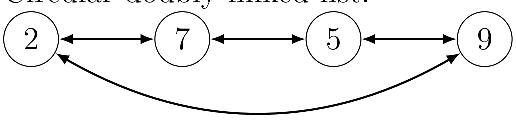
  - Doubly linked list: can be traversed backwards



  - Circular linked list:



  - Circular doubly linked list:



- Basic operations' complexity:

	Singly Linked List	Doubly Linked List
Search	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Insert at head	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Insert at tail	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Delete at head	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Delete at tail	$\mathcal{O}(N)$	$\mathcal{O}(1)$
Delete at middle	$\mathcal{O}(N)$	$\mathcal{O}(N)$

- Implementation in `C++`: `std::list` and `std::forward_list`

A (too) simple draft in `C++` for doubly linked list:

```
template<typename T>
struct Node
{
    T val_;
```

```
    Node<T> *prev_, *next_;
};

template<typename T>
struct DoublyLinkedList
{
    Node<T> head_, tail_;
    int size_;

    void clear();
    void insert(const T &val, const T &prev_val);
    void remove(const T &val);
}
```

Less common operations:

- Swap nodes without swapping data: simply search for the two nodes.
  Can be optimized a bit by running 1 single traverse instead of 2, time complexity $\mathcal{O}(N)$
- Rotate a Linked List: by placing the first element at the end.
- Reverse a Linked List: time complexity $\mathcal{O}(N)$
  - Divide the list in two parts – first node and rest of the linked list.
  - Call reverse for the rest of the linked list.
  - Link the rest linked list to first.
  - Fix head pointer to `nullptr`
- Merge two sorted linked lists: time complexity $\mathcal{O}(M + N)$ (sizes of two lists)
  - Create a dummy node for the merged list and two pointers for traversing two lists
  - Traverse the lists until reaching the end `nullptr`
  - While traversing, compare the node values of the pointers
- Merge Sort for linked lists: time complexity $\mathcal{O}(N)$, memory complexity $\mathcal{O}(N)$ or $\mathcal{O}(\log N)$
  - If the `head_ptr` is `nullptr` or there is only 1 element, then `return`
  - Else divide the list into two halves
  - Sort the two halves
  - Merge the two sorted linked lists (check above)
- Reverse a Linked List in groups of given size??
- Detect and Remove Loop in a Linked List

2.4 Stack

- ADT for Last In First Out (LIFO)
- Basic operations
  - Push: append new elements
  - Pop: remove the top element
  - Peek: view the top element without popping it from the stack
- Complexity

Pushing	$\mathcal{O}(1)$
Popping	$\mathcal{O}(1)$
Peeking	$\mathcal{O}(1)$
Searching	$\mathcal{O}(N)$
Size	$\mathcal{O}(1)$

- Implementation: `std` library, using linked lists, etc.
- Usage
  - Used behind the scene to support recursion
  - Find the next greater element `(geeksforgeeks.org)`
  - Can be used to do Depth First Search
  - Balanced parenthesis
  - Backtracking
  - Reverse a word, vector, etc.

2.5 Queue

- Is a linear data structure which models real world queues for First In First Out (FIFO)
- Two primary operations, i.e., enqueue and dequeue.
- Complexity

Enqueue	$\mathcal{O}(1)$
Dequeue	$\mathcal{O}(1)$
Peeking	$\mathcal{O}(1)$
Contains	$\mathcal{O}(N)$
Removal	$\mathcal{O}(N)$
Is empty	$\mathcal{O}(1)$

- Usage:
  - Breadth first search
  - Task scheduling: CPU, disk management, data transfer.
  - Find the starting node for a circular traverse over a graph `(geeksforgeeks.org)`

    - etc.
- Implementation:
    - Array
    - (Singly / Doubly) Linked list: enqueueing is adding elements to the tail, dequeueing is delete elements at the head

2.6 Priority Queue

- Priority Queue is an ADT, similar to normal queue, except that each element has a priority score.
- Usage:
    - Dijkstra's algorithm
    - Best First Search algorithm: e.g., A* algorithm
    - Minimum Spanning Tree
- Complexity

Binary Heap construction	$\mathcal{O}(N)$
Polling	$\mathcal{O}(\log N)$
Peeking	$\mathcal{O}(1)$
Adding	$\mathcal{O}(\log N)$
Naive removing	$\mathcal{O}(N)$
Removing with hash table	$\mathcal{O}(\log N)$
Naive contains	$\mathcal{O}(N)$
Contains with hash table	$\mathcal{O}(1)$

- Turning Min PQ to Max PQ: simply by subtracting the maximum score

2.7 Heap

- Is one common implementation for Priority Queue
- Is a DS that satisfies the ***heap invariant***
- Types of heaps:
    - Binary heap: a binary tree that supports the ***heap invariant***
    - Fibonacci heap
    - Binomial heap
    - Pairing heap
- Adding element to gradually form a complete binary tree: by simply bubbling up if it violates the ***heap invariant***
- Polling / Removing the top element:

  - Swap it with the last element in the array
  - Delete it (at the last position)
  - The element at top violates the heap invariant, bubbling it down, swap with the smallest child nodes
  - If they are tie, go for the left one
- Removing specific element:
  - Find that element in linear time
  - Swap it with the last element and remove it
  - Bubbling up or down to satisfy the heap invariant

2.8 Union

Union Find, or Disjoint Set:

- Is a DS that keeps track of elements which are split into one or more disjoint sets
- Primary operations: find and union
  - Find: which component/set a particular element belongs to
  - Union: unify two elements, by make the root of one point to the other root
- Usage
  - Kruskal's minimum spanning tree algorithm
  - Grid percolation
  - Network connectivity
  - Least common ancestor in trees
  - Image processing
- Complexity:

Construction	$\mathcal{O}(N)$
Union	$\alpha(N)$
Find	$\alpha(N)$
Get component size	$\alpha(N)$
Check if connected	$\alpha(N)$
Count components	$\mathcal{O}(1)$

- Construct a bijection (mapping) between your objects and the integers in the range $[0, n)$ (not necessary, but recommended)
  - In the array, each position is for an object
  - The value in each position is the root node of that object. Initially, the root of an object is itself
  - As the objects are grouped together, the root of each object will changed.

  - Also keep an array of the size of each group so we can break the tie if necessary, or return the size of the group
  - Path compression: when taking union, not only point the old root to the new root, but also point all elements of the old/smaller set to the new root.

2.9 Binary Search Tree

- Basic definitions:
  - Tree is an undirected graph / acyclic connected graph, has N nodes and N-1 edges
  - Root node, child, parent node, leaf node, subtree
  - Binary tree is a tree for which every node has at most two child nodes
  - Binary Search Tree (BST) is a binary tree that satisfies BST invariant: left subtree has smaller elements, and right subtree has larger elements (possibly when comparing with the root node)
- Usage: Implementation of some map and set ADTs, red black trees, AVL trees, splay trees, binary heaps, syntax trees, etc.
- Complexity:

Operation	Average	Worst
Insert	$\mathcal{O}(\log N)$	$\mathcal{O}(N)$
Delete	$\mathcal{O}(\log N)$	$\mathcal{O}(N)$
Remove	$\mathcal{O}(\log N)$	$\mathcal{O}(N)$
Search	$\mathcal{O}(\log N)$	$\mathcal{O}(N)$

  Balanced binary search trees were invented due to bad linear behavior in the worst case.

Operation in BST: Elements must be ***comparable***

- Insertion: compare the element to current node, recurse down accordingly
- Removal: find the element (if exists), replace the node with its child node to maintain the BST invariant
  - If leaf node, remove it
  - If has just one left or right subtree, replace the node with the subtree
  - If has both subtree, either replace with largest node in the left subtree or smallest node in the right subtree
- Traversal:

```
void preorder(node) {
  if (node == nullptr) return;
  print(node.value);
  preorder(node.left);
```

```
    preorder(node.right);
}

void inorder(node) {
    if (node == nullptr) return;
    inorder(node.left);
    print(node.value);
    inorder(node.right);
}

void postorder(node) {
    if (node == nullptr) return;
    postorder(node.left);
    postorder(node.right);
    print(node.value);
}
```

- level-order traversal: Breadth First Search (BFS) with queue

2.10 Hash Table

- A Hash table is a data structure mapping keys to values using hash functions.
- Keys must be unique, but values can be repeated
- Usage
  - Track item frequencies
- Properties of hash functions:
  - If $H(x) = H(y)$, then objects x and y might be equal
  - If $H(x) \neq H(y)$, then objects x and y are certainly not equal
  - A Hash function must be deterministic
  - We try very hard to make uniform hash functions to minimize the number of hash collisions
- The hash function is used as a way to index into the array, which is the hash table.
- Dealing with hash collision
  - Separate chaining: maintaining a data structure (usually a linked list) to hold all the different values which hashed to a particular value
  - Open addressing: finding another place within the hash table for the object to go by offsetting it from the position to which it hashed to
- Complexity:

Operation	Average	Worst
Insertion	$\mathcal{O}(1)$*	$\mathcal{O}(N)$
Removal	$\mathcal{O}(1)$*	$\mathcal{O}(N)$
Search	$\mathcal{O}(1)$*	$\mathcal{O}(N)$

2.11 Fenwick Tree

Fenwick Tree, also known as (a.k.a.), Binary Indexed Tree [Fen94]

- A DS that supports sum range queries as well as setting values in a static array
- Complexity:

Construction	$\mathcal{O}(N)$
Point Update	$\mathcal{O}(\log N)$
Range Sum	$\mathcal{O}(\log N)$
Range Update	$\mathcal{O}(\log N)$
Adding Index	N/A
Removing Index	N/A

- The tree and array is base 1, not 0
- Each element/node in the tree is responsible for a specific range

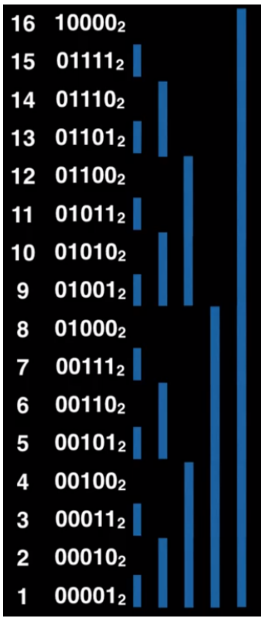


Figure 2.1: Fenwick tree: each node is responsible for a specific (blue) range

2.12 Balanced Binary Search Tree

- Balanced Binary Search Tree (BBST) is a self-balanced BST
- BBST adjusts itself in order to maintain a low (logarithmic) height allowing for faster operations such as insertions and deletions
- Complexity: unlike BST, BBST's time complexity for all tasks (insert, delete, remove, search) is $\mathcal{O}(\log N)$
- Tree invariant is property/rule that the tree must satisfy after every operation. If not, rotations are applied
- Tree rotations

```
void rightRotate(Node* node) {

   Node* P = A->parent;
   Node* B = A->left;
   A->left = B->right;
   B->right = A;
   // Also change pointer of A's parent
}
```

- AVL tree is one of many types of BBST, e.g., 2-3 tree, the AA tree, the scapegoat tree, red-black tree

2.13 Segment Tree

- Read cp-algorithms
- A DS that stores information about array intervals as a tree
- It allows range queries and point update over an array efficiently

Point Update	$\mathcal{O}(\log N)$
Range Query	$\mathcal{O}(\log N)$

  As the height of the tree is $\mathcal{O}(\log N)$
- The standard Segment tree requires $4N$ vertices for working on an array of size N
- Just as in a binary tree, with the `ID_P` of the parent node, the `ID` of child nodes can be easily identified: `(ID_P * 2 + 1)` and `(ID_P * 2 + 2)`
- Check the source code for implementation details

2.14 Sparse Table

- Check: WilliamFiset

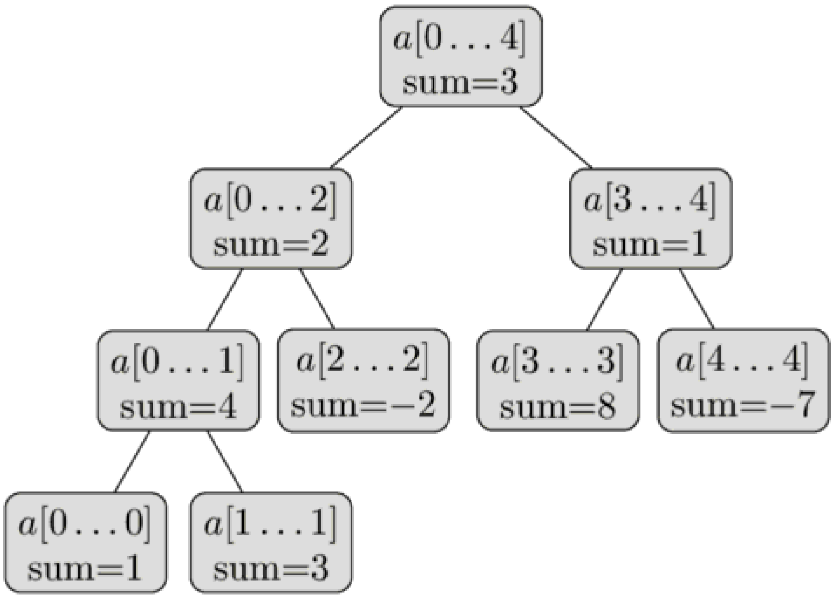


Figure 2.2: Segment tree for sum queries

- Sparse Table is all about doing efficient ***range queries*** on a static / ***immutable array***
- The high level idea in sparse table is simply to pre-computed the answer for all possible range queries of size 2^x (Fig. 2.3)

	0	1	2	3	4	5	6	7	8	9	10	11	12
0, 2^0	4	2	3	7	1	5	3	3	9	6	7	-1	4
1, 2^1			3		1		3		6				
2, 2^2			1				3						
3, 2^3			1										

Figure 2.3: Example of sparse table with min query. Each cell is responsible to answer a range of size 2^x

- Common type queries: min, max, sum, gcd, etc.
- The table can be filled using dynamic programming

$$\begin{aligned} dp[i][j] &= f(dp[i-1][j], dp[i-1][j+2^{i-1}]) \\ &= \min(dp[i-1][j], dp[i-1][j+2^{i-1}]) \end{aligned}$$
$$dp[3][2] = \min(dp[2][2], dp[2][2+2^{3-1}]) = \min(dp[2][2], dp[2][6])$$

- If the query is based on an associative functions, sparse table can answer in $\mathcal{O}(\log_2 N)$ $f(a, f(b, c)) = f(f(a, b), c) \quad \forall a, b, c$. E.g.: sum, product, etc.
- If the query is based on an *"overlap friendly"* functions, we will reach constant time $\mathcal{O}(1)$ $f(f(a, b), f(b, c)) = f(f(a, b), c) = f(a, f(b, c))$. E.g.: min, max, gcd, etc.
  Thus: $f(l, r) = f(f(l, k), f(k + 1, r))$

E.g.: for min[2, 7], calculate $p = floor(\log_2(7-2+1)) = 2$; $k = 2^p = 4$
$\Rightarrow min[2,7] = \min(t[2][2], t[2][4])$

- If the function is not overlap agnostic, we have to break down the range quite similar to in a segment tree in $\mathcal{O}(\log_2 N)$
  E.g.: with product query: $prod[0,6] = t[2][0] * t[1][4] * t[0][6]$

2.15 Suffix Array

- References: codeforces
- The suffix array is an array containing all the sorted suffixes of a string
- It is actually the array of sorted indices of the suffixes
- The suffix array and ***suffix tree*** are different, but both are compressed version of a ***trie***
- Usage:
  - The Longest Common Prefix (LCP) array

2.16 Graph

Formally, a graph is defined as a tuples: $G = \{V, E\}$, in which V is a collection of vertices and E is a collection of edges

2.16.1 Terminology

- A *self-loop* edge is in the form (u, u)
- An edge (u, v) is *incident* to both vertex u and vertex v
- The *degree of a vertex* is the number of edges which are incident to it
- Vertex u is *adjacent* to vertex v if there is some edge to which both are incident
- Directed graph versus (vs.) undirected graph
- A *path* from vertex u to vertex x is a sequence of vertices $(v_0, v_1, \ldots, v_k)$ such that $v_0 = u, v_k = x$
- A path is *simple* if it contains no vertex more than once
- A path is *cycle* if it is a path from some vertex to that same vertex

2.16.2 Graph Representation

- Edge list: $E = \{(v_i, v_j) | v_i, v_j \in V\}$
  - Easy to code, debug
  - Space efficient
  - Determining the edges incident to a given vertex is expensive

  - Adding an edge is quick, but deleting one is difficult
- Adjacency matrix: a $N \times N$ array. The $a^{i,j}$ contains a "1" if the edge (j, j) is in the graph, otherwise contains a "0"
  - Easy to code
  - Costly to store large, sparse graph
  - Finding all the edges incident to a vertex is expensive
  - Checking if 2 vertices is adjacent is very quick
  - Adding, removing edges is cheap
- Adjacency list: keep track of all the edges incident to a give vertex. The i^{th} entry of the array is a list of the edges incident to the i^{th} vertex
  - More difficult to code
  - Memory efficient
  - Finding vertices adjacent to each node is very cheap
  - Adding an edge is easy, deleting one is more difficult
- Implicit representation: there are underlying information about the edges and vertices. E.g., in chess, assuming position of the knight as vertex, it's easy to determined the fixed position it can move to

2.16.3 Connectedness

- An undirected graph is said to be *connected* if there is a path from every vertex to every vertex
- A *component* of a graph is a maximal subset of the vertices that are reachable from other vertex in the component

2.16.4 Special Graphs

- A tree is an undirected graph if it contains no cycles and is connected
- An undirected graph which contains no cycles is called a *forest*.
- A *complete* graph is a graph with edges between every pair of vertices

2.17 Matrix

2.18 Advanced Data Structure

3 CS Algorithms

3.1 Sorting

References:

- BATTLE OF THE SORTS: which sorting algorithm is the fastest? (visualization)
- stackoverflow
- cprogramming.com

Sorting algorithms:

- Selection sort: finds the next smallest element
- Bubble sort: swaps adjacent out-of-order elements
- Insertion Sort: inserts next element into its place
- Heap Sort: Selection sort with a heap
- Merge Sort: Divide and conquer method

```
void merge(
int a[], const int left, const int mid, const int right);

void mergeSort(int a[], const int begin, const int end)
{
   if (begin >= end)
   return;

   int mid = begin + (end - begin) / 2;
   mergeSort(a, begin, mid);
   mergeSort(a, mid + 1, end);
   merge(array, begin, mid, end);
}
```

- Timsort: A hybrid version of Insertion and Merge sort.
- Quick Sort: swaps elements around a pivot
- IntroSort: A hybrid version of Insertion, Quick and heap sort

3.2 Complete Search

- Keep it simple, stupid
- Brute force within the time allowed
- Can search in either the input space or the output space

3.3 Greedy Search

- Constructs a solution by always making a choice that looks the best at the moment
- Is FAST, generally linear to quadratic
- Requires little extra memory
- Is usually not correct. But when it works, it's easy and fast to implement

3.4 More Search Techniques

- Depth First Search (DFS)
- BFS
- Depth First with Iterative Deepening

3.5 Flood Fill

- Given a graph, find the all connected sub-graphs / components
- Can be performed via 3 basic ways: depth-first, breadth-first, breadth-first scanning
  - Depth-first: if current node is not visited yet, visit it, assign to a new component, recurses over all of its unvisited neighbors. However, it is bad with implicit graph
  - Breath-first: instead of recursing on the newly assigned nodes, add them to a queue.
  - Both depth-first and breadth-first run in $\mathcal{O}(N + M)$
  - Breath-first scanning, despite being a little tricky, requires no extra space. However, it is slower, $\mathcal{O}(N * N + M)$

3.6 Backtracking

- Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.
- Backtracking uses recursive calling to find the solution by building a solution step by step increasing values with time.
- It removes the solutions that doesn't give rise to the final solution of the problem based on the given constraints.

3.7 Dynamic Programming

- Dynamic Programming is an optimization over plain recursion.
- It is the approach of breaking down a problem into simpler and repeated sub-problems.
- The solutions to each sub-problem are stored so that each of them is only solved once.

Steps:

- Visualize examples
- Find suitable sub-problem
- Find relationships among sub-problems
- Generalize the relationship

4 DevOps

DevOps is a set of practices to build, test and release your code / software in small increments (source freeCodeCamp.org)

- The name comes from the name of two teams: Development team and Operations team.
  - Development team creates the plan, designs and builds the software
  - Operation team tests and feedback on the implementation
- DevOps' approach aim to improve efficiency, deliver and deploy more quickly
- Steps of DevOps:
  - Plan: work with your team to decide on the specifications for some features
  - Code
  - Build: make releases for different OSs
  - Test: automatic (/acaka continuous integration), manual testing (a.k.a. quality assurance)
  - Release: deliver your software in a way that users won't know if there are any problems. Perhaps show to a subset of users to check for issues
  - Deploy
  - Operate: scaling, ensure enough servers, etc.
  - Monitor

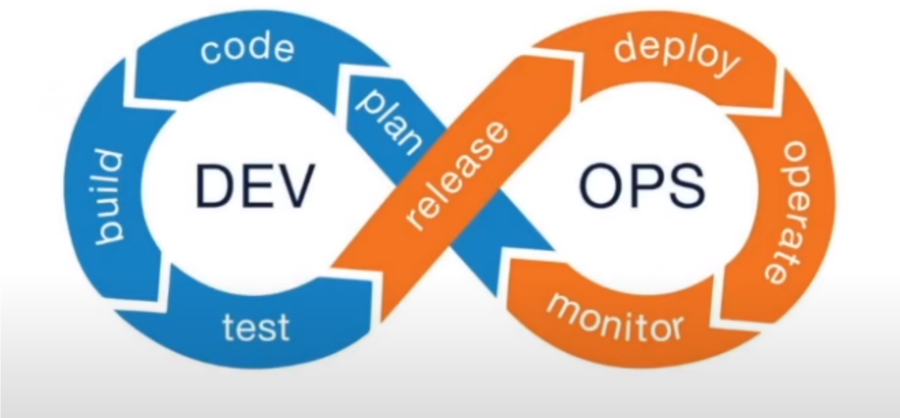


Figure 4.1: Steps of DevOps freeCodeCamp.org

4.1 DevOps Engineering Pillars

- DevOps Engineering is about being able to automate build, test, release and monitor applications

- Pull request automation: help developers build thing faster
  - Share code changes using `git` tools
  - Dealing with pull request, merge request
  - Feedback on pull request: code style, architecture, scaling
- Deployment automation: help you deploying your code in a way that users don't complain
- Application performance management: automation around making sure everything is healthy, detecting downtime, rolling back if there is problem.

You can automate:

- Test run (Continuous Integration (CI)) for each proposed changes
- Security scanning
- Notifications to reviewers, the right people to review at the same time
- Help developer to change proposals, get review and merged within 23 hours

4.1.1 Application Performance Management

- Metrics: numeric measurements of KPIs
- Logging: text descriptions of what is happening during processing
- Monitoring: take metrics and logs to convert them into health metrics
- Alerting: of monitoring detects of problem, it notifies developers

4.2 Testing and Test Driven Development

- Test Driven Development (TDD) is a coding methodology where the tests are written before the code is written
- Test levels derived from factory production:
  - Unit test
  - Integration test: ensure a few components work together
  - System / end to end tests: ensure the whole product works
  - Acceptance test: after being launched, are the customers' needs are satisfied
- Testing is about knowing when something breaks and where

4.3 Continuous Integration

- CI is the practice for the developers to commit the changes to shared repositories, trigger a workflow on a CI server

5 Review Questions

5.1 General

- What is the purpose of data structures?
- What is an ADT? Give example.

6 Technical Tools

Bibliography

[Fen94] P. M. Fenwick. “A new data structure for cumulative frequency tables”. In: *Software: Practice and experience* 24.3 (1994), pp. 327–336.