# DS and Algorithm Master Reference

[DS and Algorithm Master Reference 1](#_Toc8336)

[Commonly used words 2](#_Toc31513)

[Power set ??? 3](#_Toc13454)

[Problem solving approaches 3](#_Toc25202)

[Array 3](#_Toc27051)

[Recursion 3](#_Toc582)

[Multiple Recursion Calls 3](#_Toc19156)

[Recursion tree 3](#_Toc8827)

[Subsequences 4](#_Toc3649)

[Primitive Data Structures 5](#_Toc7855)

[Integer 5](#_Toc11858)

[Float 5](#_Toc18453)

[Character 5](#_Toc15994)

[Boolean 5](#_Toc30516)

[Linear Data Structures 5](#_Toc1146)

[Arrays 5](#_Toc21689)

[Linked Lists 5](#_Toc24696)

[Singly Linked Lists 5](#_Toc22969)

[Doubly Linked Lists 5](#_Toc8879)

[Circular Linked Lists 5](#_Toc19217)

[Stacks 5](#_Toc9169)

[Queues 5](#_Toc20684)

[Priority Queue 5](#_Toc23895)

[Circular Queue 6](#_Toc4535)

[Deque (Double-Ended Queue) 6](#_Toc26947)

[Non-Linear Data Structures 6](#_Toc15495)

[Trees 6](#_Toc19761)

[Binary Trees 6](#_Toc29504)

[Binary Search Trees 7](#_Toc21155)

[AVL Trees 7](#_Toc29368)

[Red-Black Trees 7](#_Toc32488)

[Heap (Min-Heap/Max-Heap) 7](#_Toc24441)

[B-Trees 7](#_Toc3863)

[Tries 7](#_Toc10816)

[Graphs 7](#_Toc16617)

[Directed Graphs 8](#_Toc20976)

[Undirected Graphs 8](#_Toc14213)

[Weighted Graphs 8](#_Toc13753)

[Unweighted Graphs 8](#_Toc882)

[File Structures: 8](#_Toc438)

[Sequential Files 8](#_Toc20299)

[Indexed Files 8](#_Toc20828)

[Direct Files 8](#_Toc32492)

[Hashing: 8](#_Toc15972)

[Hash Function 9](#_Toc15665)

[Hash Table 9](#_Toc2081)

[Hash Map 9](#_Toc5243)

[Handling collision 9](#_Toc7772)

[Resizing 11](#_Toc20635)

[Advanced Data Structures: 12](#_Toc29735)

[Segment Trees 12](#_Toc7379)

[Fenwick Trees (Binary Indexed Trees) 12](#_Toc25169)

[Suffix Trees 12](#_Toc9988)

[K-D Trees 12](#_Toc24151)

[Sparse Tables 12](#_Toc6477)

[Disjoint-Set (Union-Find) 12](#_Toc12810)

[Specialized Data Structures: 12](#_Toc1447)

[Bloom Filters 12](#_Toc325)

[LRU Cache 12](#_Toc31546)

[Abstract Data Types (ADTs): 12](#_Toc5730)

[List ADT 12](#_Toc31615)

[Stack ADT 12](#_Toc29526)

[Queue ADT 12](#_Toc30071)

[Map ADT 13](#_Toc21652)

[Set ADT 13](#_Toc24434)

[Dynamic Programming 13](#_Toc31784)

[Algorithm 13](#_Toc3138)

[DFS (Depth First Search) 13](#_Toc3765)

[Backtracking 13](#_Toc32458)

[Misc and Questions 13](#_Toc22830)

[1. Difference between graph and tree 13](#_Toc22347)

#### References

Coursera: <https://www.coursera.org/learn/algorithms-part1?action=enroll>

Github: https://github.com/BalajiBaskaran24/DotnetDev/tree/main/DSAndAlgoSample/DSAndAlgoReference

## Commonly used words

Trivial - Less significant or little value

## Power set ???

## Problem solving approaches

### Array

1. Check whether sorting is required or not
2. If its related to

## Recursion

Function calling itself. Base for dynamic programming. Parameterized recursion, functional recursion are some kind of recursion.

Sample in github (RecursionMethods class in DSAndAlgoReference solution)

### Multiple Recursion Calls

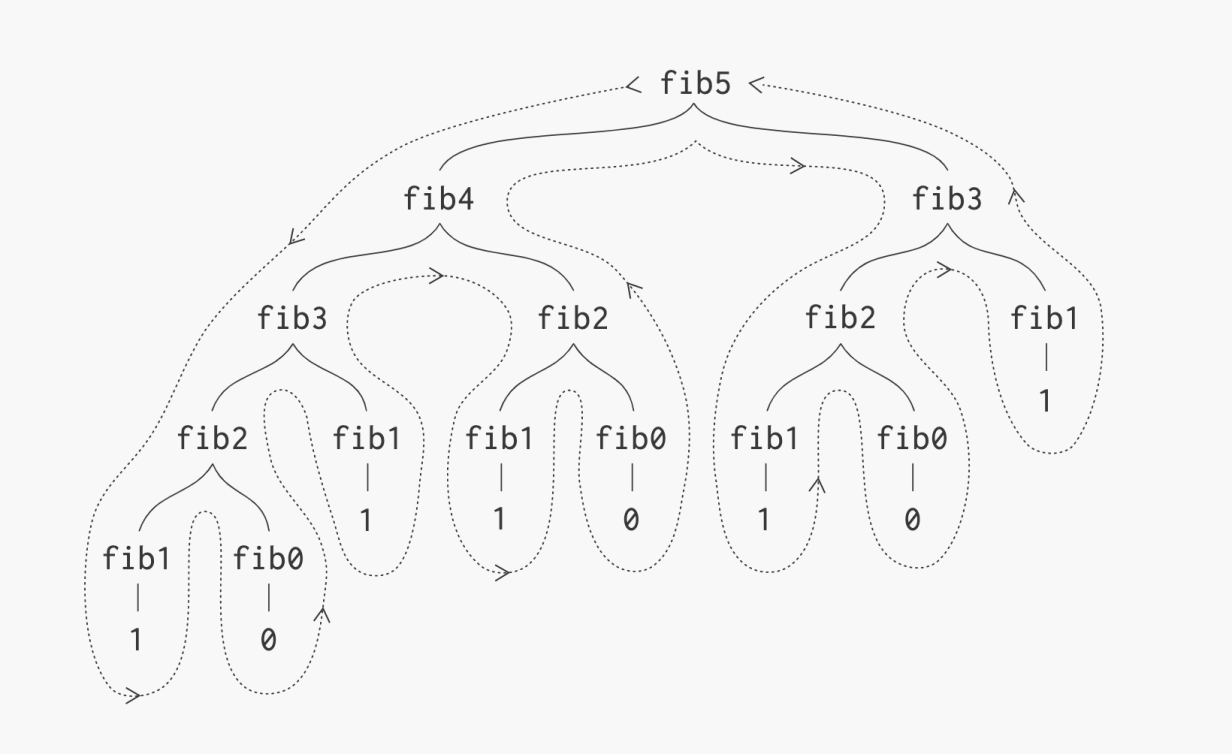
Function called multiple times

F()

{F();F();}

### Recursion tree

For fibonacci series,



### Subsequences

Contiguous or non-contiguous sequences. Follows order. Take or not take a value at certain index is done to form sub-sequences.

## Primitive Data Structures

### Integer

### Float

### Character

### Boolean

## Linear Data Structures

### Arrays

### Linked Lists

### Singly Linked Lists

### Doubly Linked Lists

### Circular Linked Lists

### Stacks

### Queues

### Priority Queue

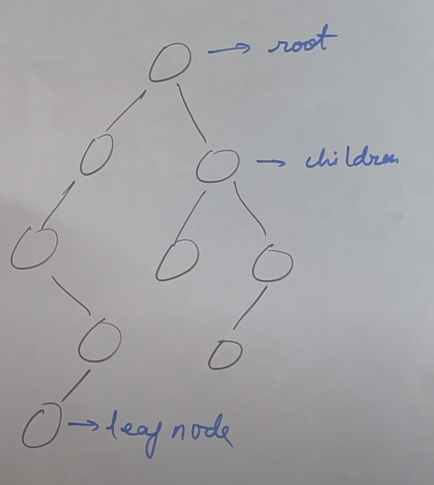
### Circular Queue

### Deque (Double-Ended Queue)

## Non-Linear Data Structures

### Trees

### Binary Trees



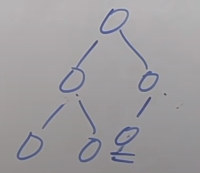
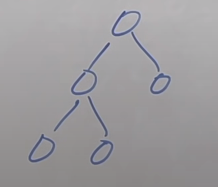
Sub tree - For any particular node, all connected nodes below that node is mentioned as subtree

Ancestor - all nodes above the current nodes are said to be ancestor

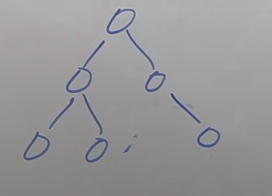
#### Types

**Full binary tree** - Each node will have 0 or 2 children

**Complete binary tree** - All levels are completely filled except the last level, last level has all nodes in left as possible



Below one is not a complete BT. Last level should be in left, if its incomplete.

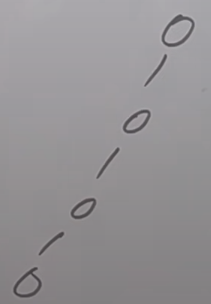


**Perfect binary tree:** All leaf nodes are at same level

**Balanced binary tree:** Height of the three can be at max of log(n)

Ex: If n=8, log 8 = 3

**Degenerate tree:** Skewed to one side



#### DFS traversal

**In**-order: Left, **root**, right

**Pre**-order: **Root**, left, right

**Post**-order: Left, right, **root**

**Position of root will be based on order of traversal**

#### BFS traversal

### Binary Search Trees

For each node,

All nodes in its left subtree have values **less than** the node's value

All nodes in its right subtree have values **greater than** the node's value.

### AVL Trees

### Red-Black Trees

### Heap (Min-Heap/Max-Heap)

### B-Trees

### Tries

### Graphs

**Node:** Each element is represented as node/vertex

**Edge:** Connecting lines between nodes

Graph can be open structure as well

**Cyclic/acyclic:** Starting at one node and reaching the same node in directed and un-directed graph.

**Path:** Contains lot of nodes and each of them are reachable. Node cannot appear twice in a path. There should be edge between consecutive nodes in a path.

**Degree of node:** Number of edges attached to node in case of undirected graph. In directed it will be in-degree(number of edges inward) and out-degree(number of edges outward)

**Edge weight:** Represents weight of edge. 1 if the weight is not assigned.

**Property:**

Total degree of graph = 2 \* Number of edges

Way to store edge,

1. Matrix
   1. Uses N\*N space
   2. How to store:
      1. In a 2D array mark the edge co-ordinates with 1. For example graph contains 5 elements. 1,2,3,4,5, If it contains edge between 1,2 then mark 1st row \* 2nd column as 1 and 2nd row \* 1st column as 1
2. List

### Directed Graphs

All edges are directed. The directed edge between nodes can be multiple.

### Undirected Graphs

Edge will be in both the directions.

### Weighted Graphs

### Unweighted Graphs

## File Structures:

### Sequential Files

### Indexed Files

### Direct Files

## Hashing:

Hashing is the process of converting a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index to an array (called a hash table). The values are used to index a fixed-size table called a hash table. In C# **Dictionary** is implemented using hashing technique

**C# classes**

**Hash Table:** Hashtable class. - Recommeded to use **Dictionary**

**Hash Map:** Dictionary<TKey, TValue> class.

**Hash Set:** HashSet<T> class. - Can be thought of as a Dictionary<TKey,TValue> collection without values. Set is a collection without duplicate elements

### Hash Function

A hash function takes a key and returns an integer, known as the hash code. A good hash function distributes keys uniformly across the hash table to avoid collisions as much as possible.

### Hash Table

A hash table is an array of fixed size, and it stores values. The index where a value is stored is determined by the hash code of its key, often modulo the size of the array.

### Hash Map

A hash map is a data structure that provides a way to store key-value pairs and allows you to efficiently insert, delete, and retrieve values based on the key. It's also known as a hash table or dictionary in various programming languages.

### Handling collision

A collision in hashing is when two or more keys are hashed to the same index in the hash table. This situation needs to be handled, or else the new key would overwrite the value that was already at that index in the hash table. Collisions can occur due to various reasons, such as limited size of the table, a poor hash function, etc.

#### Separate Chaining:

##### Linked List Chaining:

Each cell in the hash table contains a linked list of all the keys that hash to that index.

Code in github

##### Binary Search Tree Chaining:

A variation of chaining that uses a balanced binary search tree instead of a linked list.

#### Open Addressing:

Open addressing stores all entries directly in the array itself. When a collision occurs, the algorithm searches through the array to find the next open spot.

##### Linear Probing:

After a collision, check the next cell in the array, then the next, etc., until an empty cell is found.

##### Quadratic Probing:

Similar to linear probing but looks at cells that are a quadratic number of cells away from the original hash.

##### Double Hashing:

Uses a second hash function to decide how many cells to skip before checking again.

#### Rehashing:

When the table becomes too full, a common strategy is to create a new table of a larger size and rehash all existing keys.

#### Cuckoo Hashing:

Uses two or more hash functions. If a key is hashed to a place that's already occupied, the key that's already there is rehashed with its second hash function, making room for the new key.

#### Hopscotch Hashing:

Allows a new key to displace a key at an existing hash, as long as it doesn't move it too far from its original hash.

#### Robin Hood Hashing:

Works with open addressing and tries to ensure that each key is as close to its original hash as possible. It "steals" from rich cells (those with keys that are close to their original hashes) and gives to poor cells (those with keys far from their original hashes).

#### 2-Choice Hashing (or d-Choice Hashing):

Picks d random hash functions and places the new key in the bucket that is the least full among the d choices.

#### Coalesced Hashing:

Combines open addressing with a linked list, effectively merging cells together as they fill up.

#### Dynamic Perfect Hashing:

A two-level technique where the first level divides the keys into buckets, and the second level uses a perfect hash function for the keys in each bucket.

#### Bloom Filters (Probabilistic Hashing):

A probabilistic data structure that can tell you if a key is definitely not in the set or may be in the set. It's not a traditional collision resolution strategy but is related to hashing.

### Resizing

If the hash table becomes too full, it may need to be resized to maintain efficiency. This often involves creating a new, larger array and rehashing all existing keys.

## Advanced Data Structures:

### Segment Trees

### Fenwick Trees (Binary Indexed Trees)

### Suffix Trees

### K-D Trees

### Sparse Tables

### Disjoint-Set (Union-Find)

## Specialized Data Structures:

### Bloom Filters

### LRU Cache

## Abstract Data Types (ADTs):

### List ADT

### Stack ADT

### Queue ADT

### Map ADT

### Set ADT

## Dynamic Programming

## Algorithm

### DFS (Depth First Search)

Algorithm for traversing or searching, tree or graph data structure. The algorithm starts at the root node and explores as far as possible along each branch before backtracking. In other words, it goes deep into each path before exploring its siblings.

### Backtracking

Trial-and-error-based approach where you attempt to build a solution incrementally. If a partial solution is found not to satisfy the problem's constraints, you "backtrack" to a previous step and try a different path.

## Misc and Questions

### Difference between graph and tree

|  |  |  |
| --- | --- | --- |
|  | Graph | Tree |
| Definition | Collection of vertices (nodes) and edges that connect pairs of vertices. It represents relationships between elements. | A tree is a special type of graph that has no cycles and is fully connected. It's a hierarchical structure with parent-child relationships. |
| Cycles | May contain cycles, i.e., a path where the start and end vertices are the same | Trees do not contain cycles by definition. |
| Edges | Can have directed or undirected edges. | Edges are directed, always from parent to child. |
| Root | Graphs do not necessarily have a root node. | Trees have a specific root node from which all other nodes descend. |
| Paths | There may be more than one path between two vertices. | There is exactly one unique path between any two nodes. |
| Types | Graphs can be undirected, directed, weighted, unweighted, cyclic, acyclic, etc. | Trees can be binary, binary search, AVL, red-black, etc. |
| Complexity | Generally more complex than trees. | Generally simpler than graphs |
| Use Cases | Modeling networks, social networks, transportation systems, etc. | Representing hierarchical data like file systems, parsing expressions, managing sorted data, etc. |

**Cycles:** Graphs may contain cycles, while trees cannot.

**Root:** Trees have a specific root node, whereas graphs may not.

**Paths:** In a tree, there is exactly one path between any two nodes, while in a graph, there can be zero or more paths.

**Complexity:** Trees are a specific type of graph with constraints that often make them simpler to work with.