Arrays

array is a set of elements put next to each other in RAM. Each element has an address and we can go to any element in the array in O(1) time if we know the index or address of element. Thanks to that each element is next to each other in memory we can easily go to the next element by adding size of the element to current's element address.

Stacks

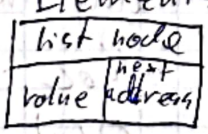
pop - remove last added element to stack
push - add element to stack at the end
top - return last added element on the stack

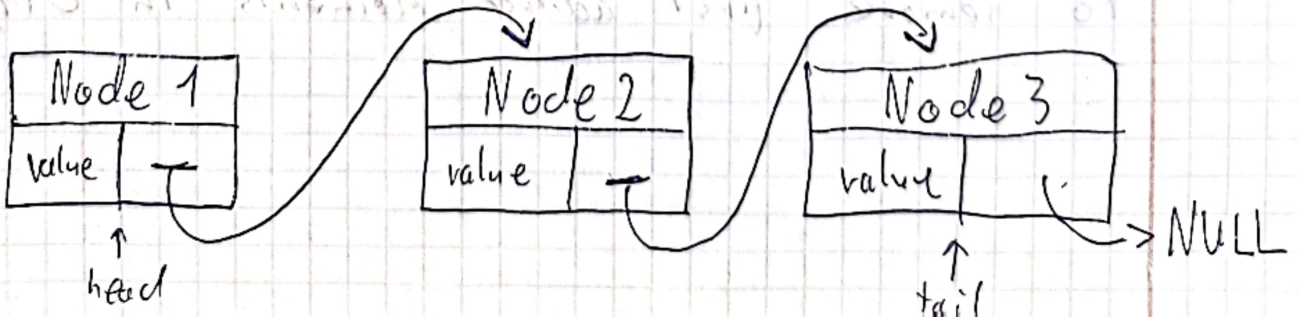
stack is LIFO - Last in, First out data structure

Linked Lists

Linked lists consists of list nodes, each node has a value and address to the next element in the list. Last element in the list points to a NULL. First element is called a head and last is called a tail.

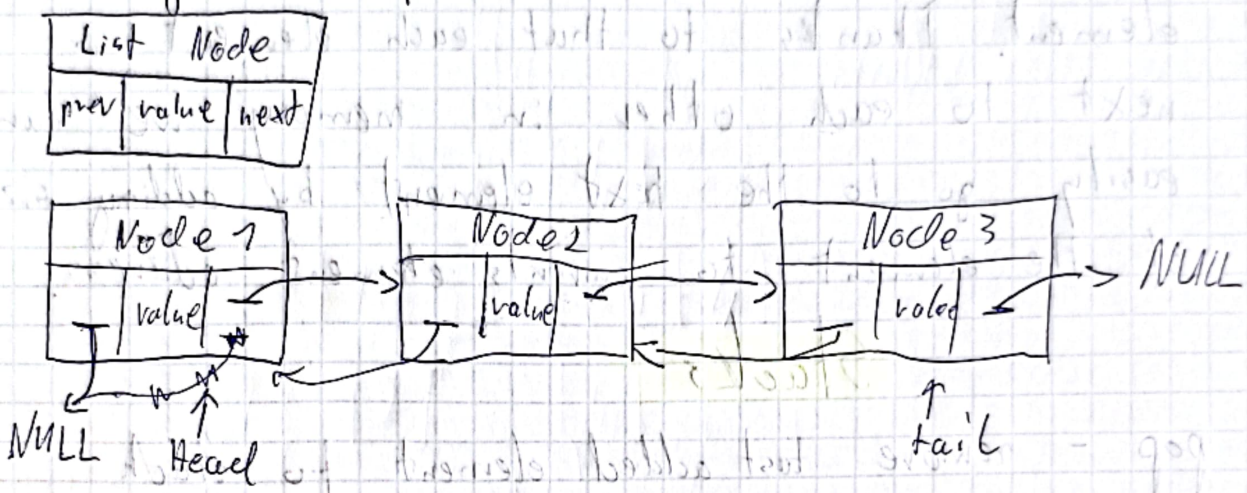
Elements of linked lists doesn't have to be next to each other in memory.





Doubly linked lists

Doubly linked lists are very similar to singly linked lists, but here each node has 2 pointers, pointing to next and previous elements. Thanks to that accesing elements is easier and traveling both forwards and backwards is possible.



Array Vs Linked List

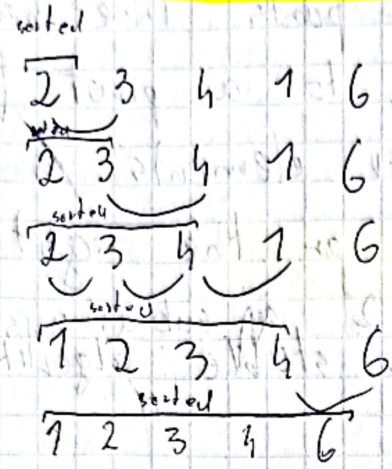
Operation	Big-O time Array	Big-O time Linked list
Access i-th element	O(1)	O(n)
Insert/remove end	O(1)	O(1)
Insert/remove middle	O(n)	O(1)

Queues

Queues follow FIFO - First In, First Out.
Queues have 2 operations enqueue - add element, and dequeue - remove element. They are usually implemented using linked lists because they allow to remove first added elements in O(1) time

Sorting

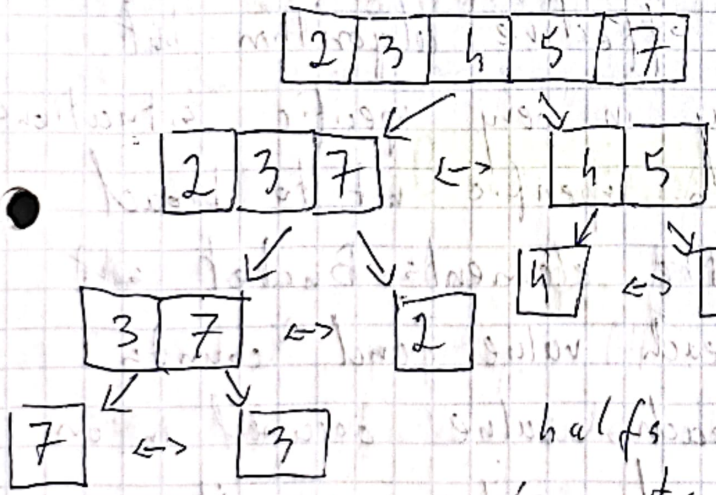
Insertion Sort $O(n^2)$



Insertion sort starts with sorted 1 element array and go through the rest of elements puting them in right places in sorted part, by comparing from left.

If there is a "tie" while sorting algorithm which preserve starting order of equal elements is called stable and that one which doesn't guarantee that is called unstable

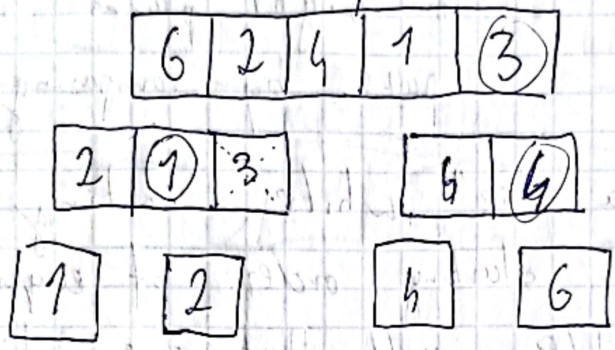
MergeSort $O(n \log n)$



MergeSort uses ~~techni~~ technic called Divide and Conquiere. We divide our array to approximatly halfs until we have single elements, then we compae them and merge. Merge sort is a recursive algorithm.

Quick Sort $O(n^2)$, $\Theta(n \log n)$

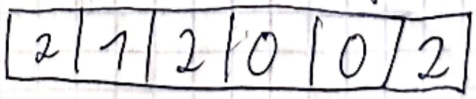
Quick sort is another resursive algorithm, it also splits the array to 2 parts like mergesort but it does it according to a pivot (usually last element). It puts smaller elements on the left of pivot and bigger on the right and then calls itself on the 2 sub arrays.
Quick sort is usually not a stable algorithm.

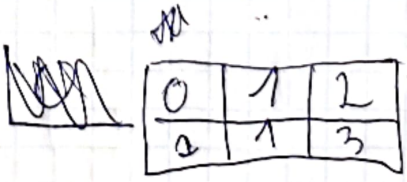


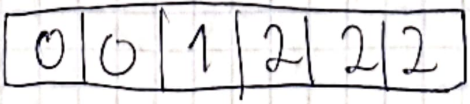
After division pivot is always in a right place so there is no need to sort it further.

Bucket Sort $O(n)$

Bucket sort is a very effective algorithm but it can be used only in very specific situations where we have given specific not to broad range of values of the elements. Bucket sort creates „buckets" for each value and counts how many times each value occured in an array and at the end simply overwrites the array with values from the bucket.
It is not a stable algorithm



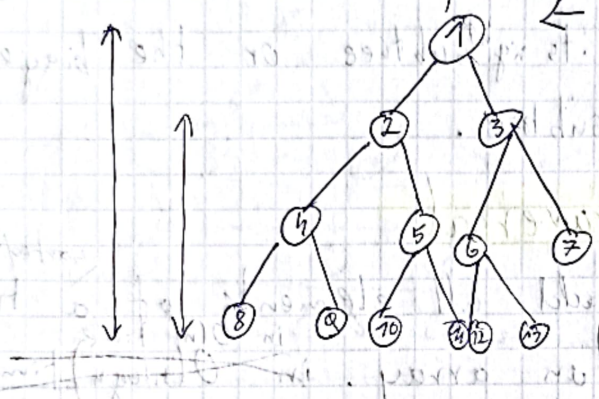




Binary Search O(log n)

It is a search algorithm which with each iteration narrows our search by half. To run it the array must be already sorted.

Binary Tree



← Root Node

ex. 3 is a parent of 7 and 6

8, 9, 4 are desendants of 2

5, 2, 1 are ancestors of 10 and

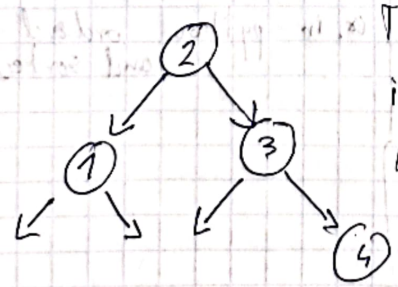
8, 9, 10, 11, 12, 13 are leafes nodes

height is numbers of nodes from a node to a leves
ex. height of 2 is 3, ~~height of 8 is 4~~

Depth of a node is height calculated from it to a root node.

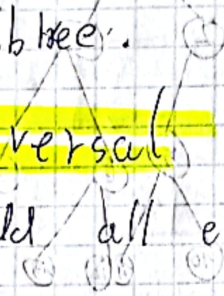
Binary Search Trees (BST)

They are similar to binary trees but have a sorted propety. All smaller elements are on the left of a node and bigger on the right.



Thanks to that searching time is like in sorted array O(log n) which is actualy a height of the tree. The advantage against the array is the fact that adding and removing elements from BST takes also O(log n) time compare

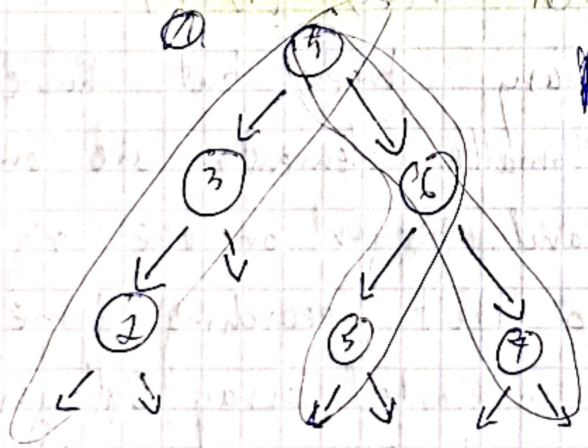
We can add elements to BST as a leaves or in the middle. Adding as leaves is easier and more preferable but it usually result in less balanced tree. While removing an element we have 2 cases: easy one where removed element doesn't have child nodes or have only 1. The hard one where it has 2 child nodes, here we have to substitute this element with the smallest element in its right subtree or the biggest element in its left subtree.



BST - Traversal

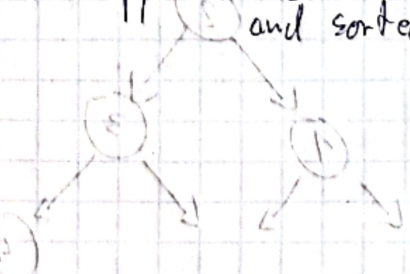
We can (for instance (inorder traversal)) add all elements of a tree in sorted order to an array. in O(n) time ~~in O(n log n) time~~ (sorted)

To do it we have to visit all left decendents of the node before visiting its right node. This is an example of Depth-First Search algorithm where we firstly go as deep as we can in each node.



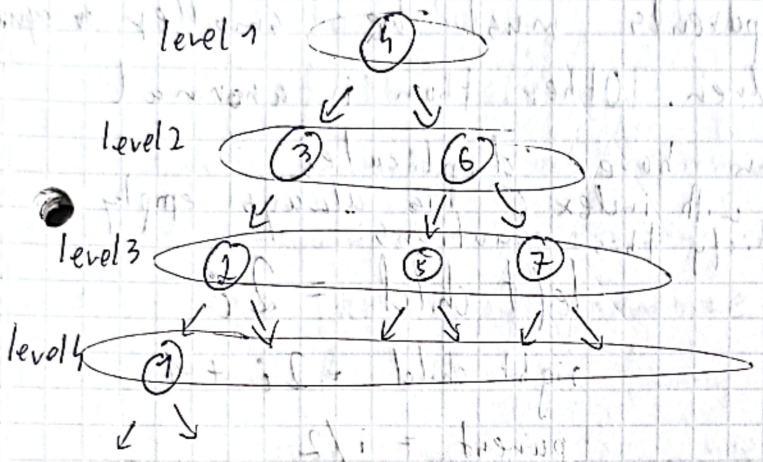
DFS

Binary trees are one of the ways of implementing other DSs like sets and maps ex. in python ordered-set and sorted-dict



Breadth-first Search BFS

In BFS in contrary to DFS we firstly visit each level of the tree. BFS uses queues to append each child nodes of a currently process node and go to the next node in queue. because queues are First-in First-out data structure.



Backtracking

Backtracking is an algorithmic technique which uses recursion to find a solution to a problem. It removes a solution iff at any time it finds a part that doesn't comply with a rules, and go back to last moment where it has a choice ~~which~~ which hadn't been tryied yet.

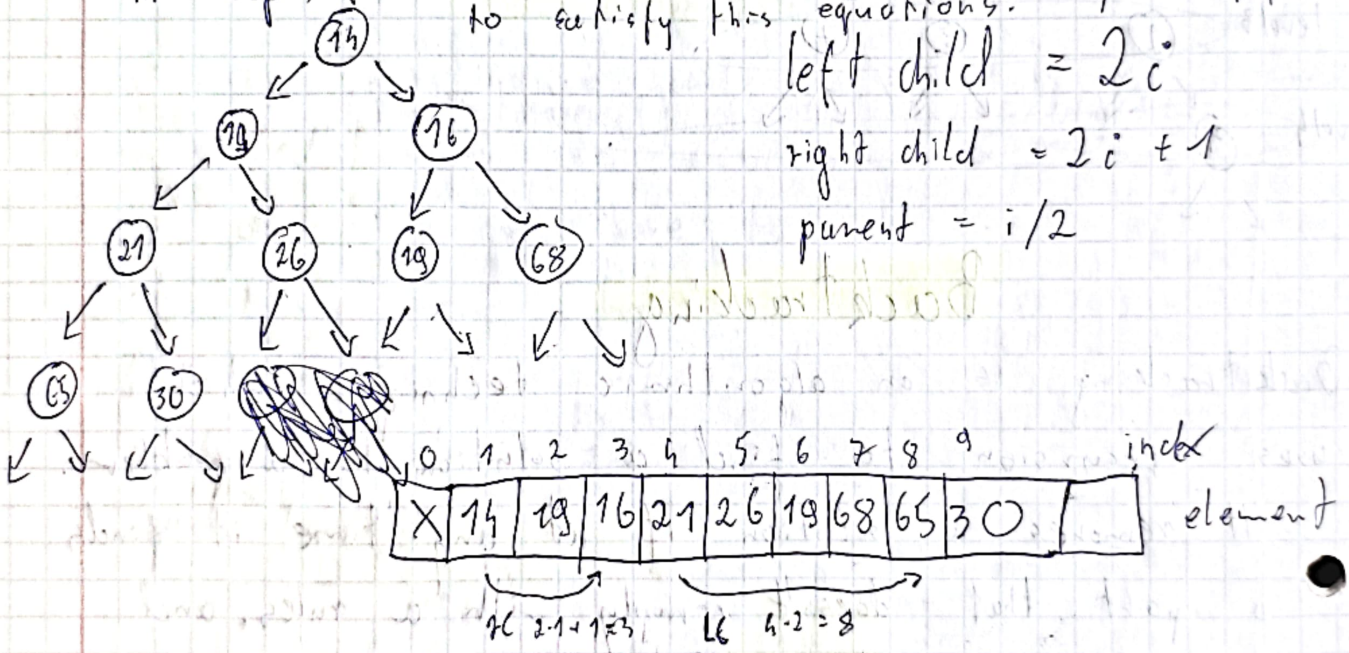
Heap / Priority Queue

Heap is a data structure which is implemented using compleate binary tree (doesn't have missing nodes except a leaves) which in really is an array underneith. This is called a structure propety, there is also an order propety. It can be a max or min heap which are analogus so for min heap all parents must be smaller or equal than their children. Other than in normal binary tree we can have dyplicates.
In heap, first element with index 0 is always empty to satisfy this equations:

left child = $2i$

right child = $2i + 1$

parent = $i/2$



To add an element to heap we add it at the end and swap with parents until we have an order propoty satisfied.
To remove element we remove root and put last element in its place and then swaps it with its child nodes until order property is satisfied. Time complexity of adding/removing is equal to height of a tree which is $O(\log n)$

Heapify allows us to create a heap from unsorted array in O(n) time

Hash sets/maps

Hash maps are a very useful datastructures which allow us to insert, delete and search values in a O(1) time. It is so fast thanks to hashing which calculate the index of the element in the list using some mathematical formula for ex. the remainder of devising sum of ASCII values of the key by size of the array. To avoid collisions (many elements are asigned the same index) the size of array is increased if it is half-full. The collisions will still happen and then we can use arrays in arrays or use open addresing where we assign other index to a key if that index is occupied for eg. the next empty, the spiral of it ect.

Graphs

Graphs are (very similar to linked lists and trees, but graphs allow (cycles) loops. There can be different implementations of graphs:

Connections of nodes are called edges

Matrix: graph is represented as 2D array. I uses 0 to represent a free space and 1 a blocked one. Free spaces (0) represent nodes and edges are implicit, ~~they~~ they are represented by posibility of moving in given direction

1	0	0	1	1
0	1	0	1	0
0	1	0	0	0
0	0	0	1	1
1	1	0	0	1

Adjacency Matrix - usualy represended as ~~2D~~ square matrix. Here nodes are represended by vertices Here each element represents the wage or posibility of moving from node ie. 3 to node 4

matrix[3][4]

	1	2	3	4
1	0	0	0	0
2	1	1	0	0
3	0	0	0	1
4	0	1	0	0

Adjacency List - here we store information about neighbours in a list (of Node type). It is the most popular way to represent a graph because it uses the least amount of space (it don't have representation of nodes that doesn't exist)

```
class Node:
    def __init__(self, val):
        self.val = val
        self.neighbors = []
```

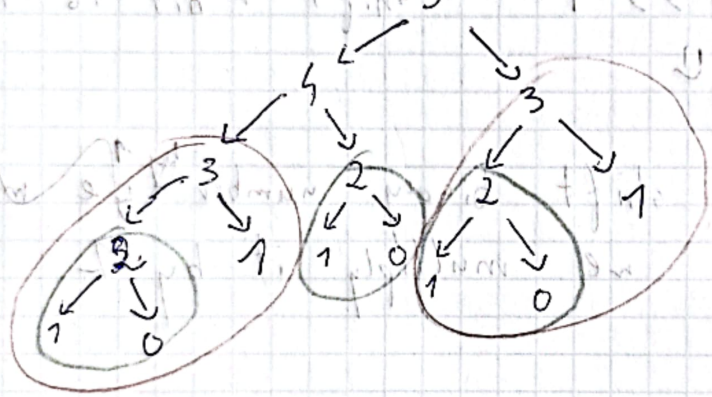
Matrix-DFS

Dynamic Programming

Dynamic programming is about solving complex problems by solving some simpler problems first and then using them to solve starting problem

Memoization (Top Down DP) we store in cache (usually hash map) all currently solved problems so we don't have to solve them again.

i.e. Calculating Fibonacci sequence



Bottom up DP (True dynamic programming) here we start with the base case and work our way up. i.e. Fibonacci sequence

0	1	2	3	4	5
0	1	1	2	3	5

But here we really need last 2 elements to calculate the next element so we can save space

0 1 1 2	3 5

Bit Manipulation

AND	
0 & 0	0
0 & 1	0
1 & 0	0
1 & 1	1

OR	
0 | 0	0
0 | 1	1
1 | 0	1
1 | 1	1

XOR	
0 ^ 0	0
0 ^ 1	1
1 ^ 0	1
1 ^ 1	0

Bit shifting

001 << 1 - shift 1 bit to left

⇓

010

010 >> 1 - shift 1 bit to right

⇓

001

Basicly when we shift binary number by 1 to left we multiply it by 2