[**Time complexity**](#_el0xfz3u32yv) **2**

[Big O](#_b07tao3y8tq8) 2

[Big Omega Ω](#_45fj1sgx90nl) 2

[Big Theta](#_v2pssk4trev) 2

[f(n) < c0\*g(n) for all n > N0](#_b73ph5tl8df8) 2

[**Simple data structures (sorted and unsorted)**](#_c9c3amps9h6a) **3**

[**Binary tree**](#_qnt96xjntw0) **4**

[Complexity](#_uuokbqtusu70) 4

[Traversing](#_1yae0phkbhfr) 4

[Deleting](#_af21511ytorf) 5

[Balancing](#_1osfufed0sli) 7

[**Hashing**](#_p9d4xo3n7tq8) **12**

[Chaining](#_plqfkmuxxjw) 12

[Linear Probing](#_bklmz0fka7h8) 12

[Double hashing](#_mlhgjjmbg433) 12

[**Sorting**](#_tff9xidahtmf) **12**

[Selection sort](#_to99l9x2m0bi) 12

[Insertion sort](#_9t2bgm3qxqyb) 12

[Count sort](#_s6pbv32mluuu) 13

[Quick sort](#_nlk8vj2whbqk) 13

[Merge sort](#_yecyc4frv8fb) 16

[Master theorem](#_yqw8huz2hrwt) 16

[**Heaps**](#_pvuimaq0gzky) **18**

[Deletemax function](#_qriym7viukp1) 18

[Downheap](#_sxtr4kys3cwf) 19

[Upheap](#_u7zm9kmj68sf) 20

[Heap sort](#_d29o6tkj0uau) 21

[**Graphs**](#_ilenukygecw4) **22**

[Depth First Search (DFS)](#_uw71xf2rq8sb) 23

[Breadth First Search (BFS)](#_bslc6em3xo97) 23

[Adjacency list, Matrix, Sparse graph, dense graph](#_d5to16j6lh21) 25

[Single shortest path - Dijkstra](#_2grnp7trl1p) 26

[All pairs shortest path](#_8sm7v97wki50) 28

[Warshall](#_6wu0jstux8gg) 28

[Floyd-Warshall](#_r80ehofjutge) 29

[Minimum spanning tree](#_8fxg2gwa6f05) 32

[Prims](#_zh9fmdxp9bfv) 33

[Kruskal](#_s5kkop7rfow1) 33

[Toposort](#_mjuckhstjhqu) 34

[**Heuristic (or informed)**](#_ymordafdj7ib) **35**

# **Time complexity**

***Time complexity order***

1 - slowest

logn

n

nlogn

n^2

2^n

n! - fastest

<https://www.bigocheatsheet.com/>

## Big O

Big O is (closest) upper bound

If a function is O(n2), it’s also O(n3), O(n4), O(n!), etc.

## Big Omega Ω

Big Omega Ω is lower bound

T(n) = n^2 is in Ω(n)

## Big Theta

Big Theta Θ is tight bound. Where it is in both Big O and Big Omega

* 2n^2+4 is in Θ(n^2) because it is in both Ω(n^2) and Θ(n^2)
* Θ(n^2) always grows at n^2
* When we refer to O(n) in CS terms, we are usually referring to Θ(n)

## f(n) < c0\*g(n) for all n > N0

For two functions f(n) and g(n), we say that f(n) is in O(g(n)) if:

There are constants c0 and N0, such that

f(n) <= c0\*g(n) for all n > N0

How to do?

sub value of No into N.

f(n) <=c0\*g(O(f(n))) holds true?

Is n^2+33 in O(n^2)?

Yes, for c0 = 2, N0 = sqrt(33): n^2 + 33 < 2n^2

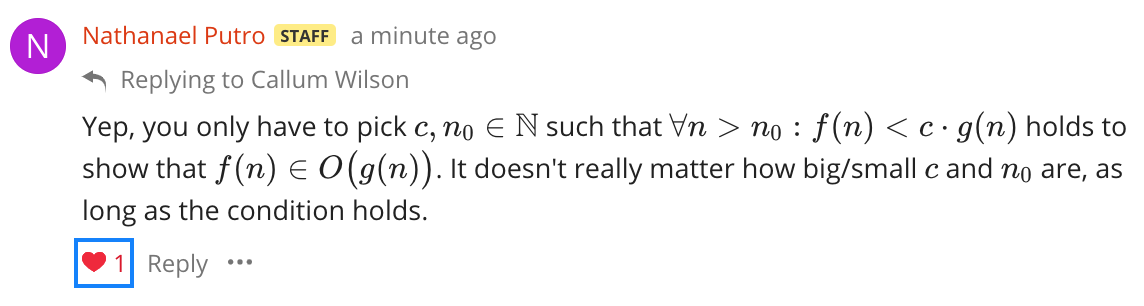
(2)^2 + 33 < 2(sqrt(33))^2

Is n^2 + 33n + 17 is in O(n^2)?

Yes, for c0 = 2, N0 = 34 n^2 + 33n + 17 < 2n^2

Is 15n^2 + 33n + 17 is in O(n^2)?

Yes, for c0 = 15, N0 = 34 15n^2 + 33n + 17 < 16n^2



if f(n) = 2n and g(n) = n^2,

make c any number. Let’s say c = 10.

then

2n < 10\*n^2 which can be simplified to: n<⅕.

so N\_o has to be greater than ⅕ for f(n) to be an element of O(g(n))

write in this format: 

# 

# 

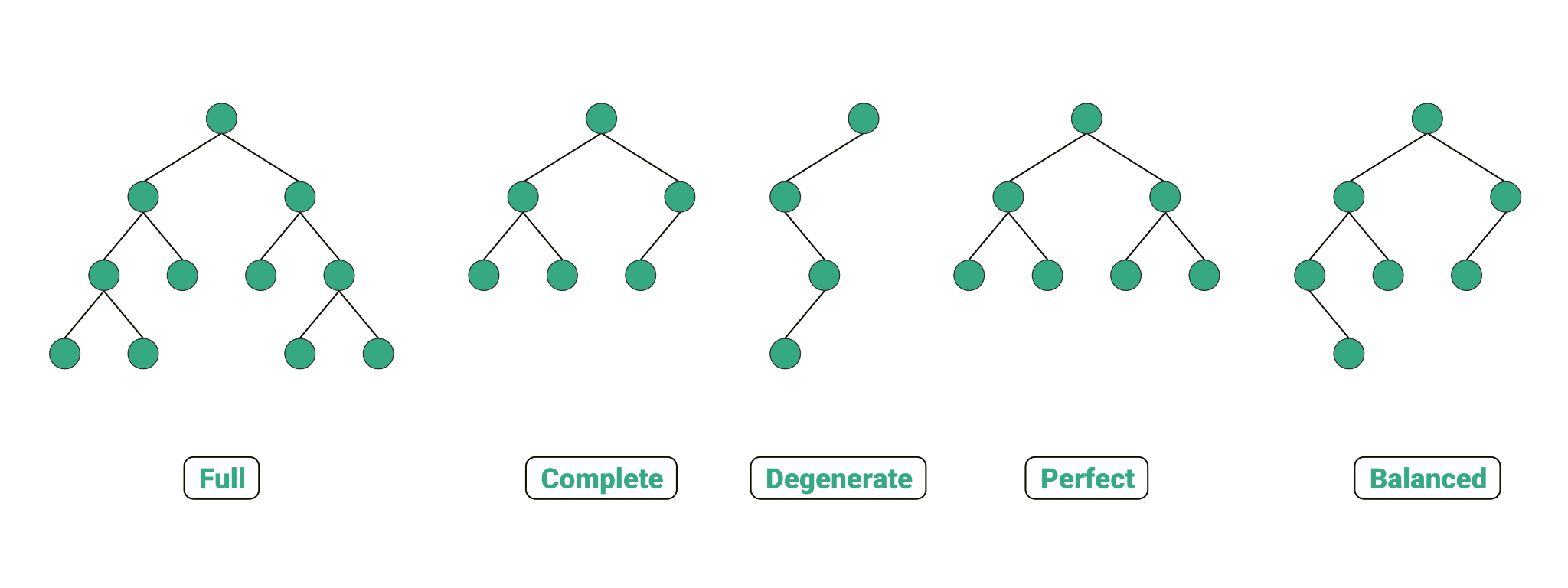
# 

# **Simple data structures (sorted and unsorted)**

Arrays do support random access whereas List do not. You have to traverse a list.

|  | Unsorted Array | Sorted Array | Unsorted List | Sorted List |
| --- | --- | --- | --- | --- |
| Build | O(n) | O(n^2) | O(n) | O(n^2) |
| Search | O(n) | O(log(n)) - binary | O(n) | O(n) |
| Insert | Static: O(1) Dynamic: O(n) - realloc | O(n) -Move array down | O(1) | O(n) - need to traverse |

# **Binary tree**

****

Full: All nodes either 2 child or no child

Complete: binary search tree, each level (except possibly the last) is completely filled, and all nodes are as far left as possible.

Degenerate: all nodes only have 1 child

Balanced: at any node, the number of nodes to its left and the number of nodes to its right (downwards) don't differ by more than 1

## Complexity

Search/delete:

* Best/ average case = O(logn)
* Worst = O(n) (stick)

## Traversing

In order traversal

void traverse (struct node \*root) {

if (root) {

traverse(root->left);

print(root); //do smthing

traverse(root->right);

}

}

* Good for sorting/deleting

Pre order traversal

void traverse (struct node \*root) {

if (root) {

print(root); //do smthing

traverse(root->left);

traverse(root->right);

}

}

* Good for copying trees because it starts from the root first, and then copies as it traverses down

Post order traversal

void traverse (struct node \*root) {

if (root) {

traverse(root->left);

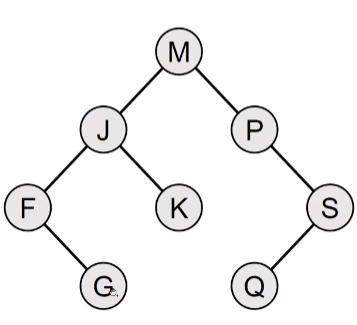
traverse(root->right);

print(root); //do smthing

}

}

* Good for freeing



Given that the visit function prints the value at the node, in what order does this tree get printed for in order traversal?

Ans) F,G,J,K,M,P,Q,S (Alphabetical order)

Post order: GFKJQSPM

Pre order: MJFGKPSQ

Successor for M is P, predecessor for M is K for in order traversal

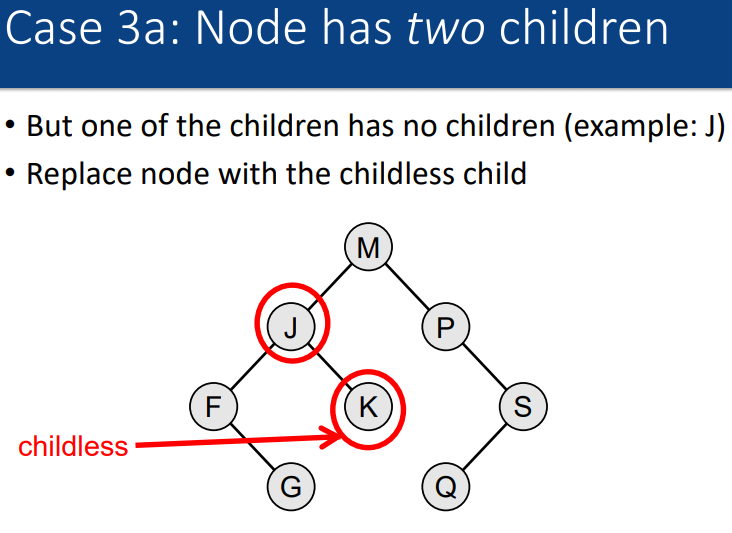
## Deleting

Case 1: Node is a leaf, just delete it

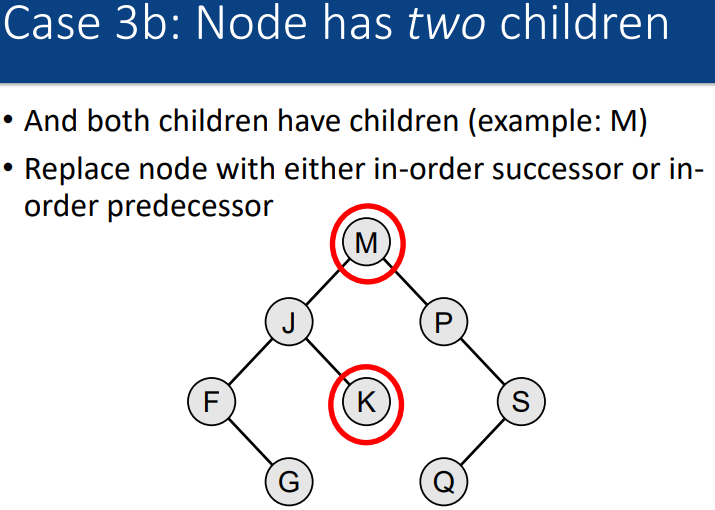
Case 2: Node has one child, replace node with child



Case 3a: Node has two children, But one of the children has no children



Case 3b: Node has two children and both children have children



## Balancing

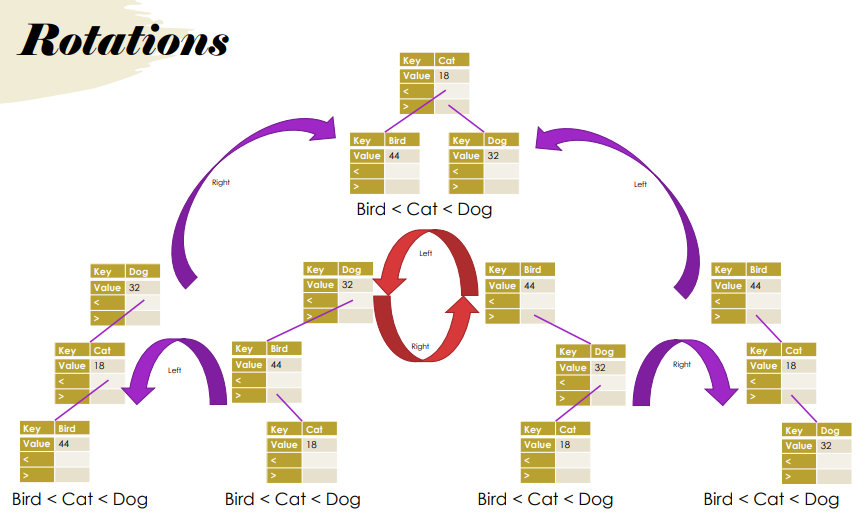
<https://www.geeksforgeeks.org/avl-tree-set-1-insertion/>

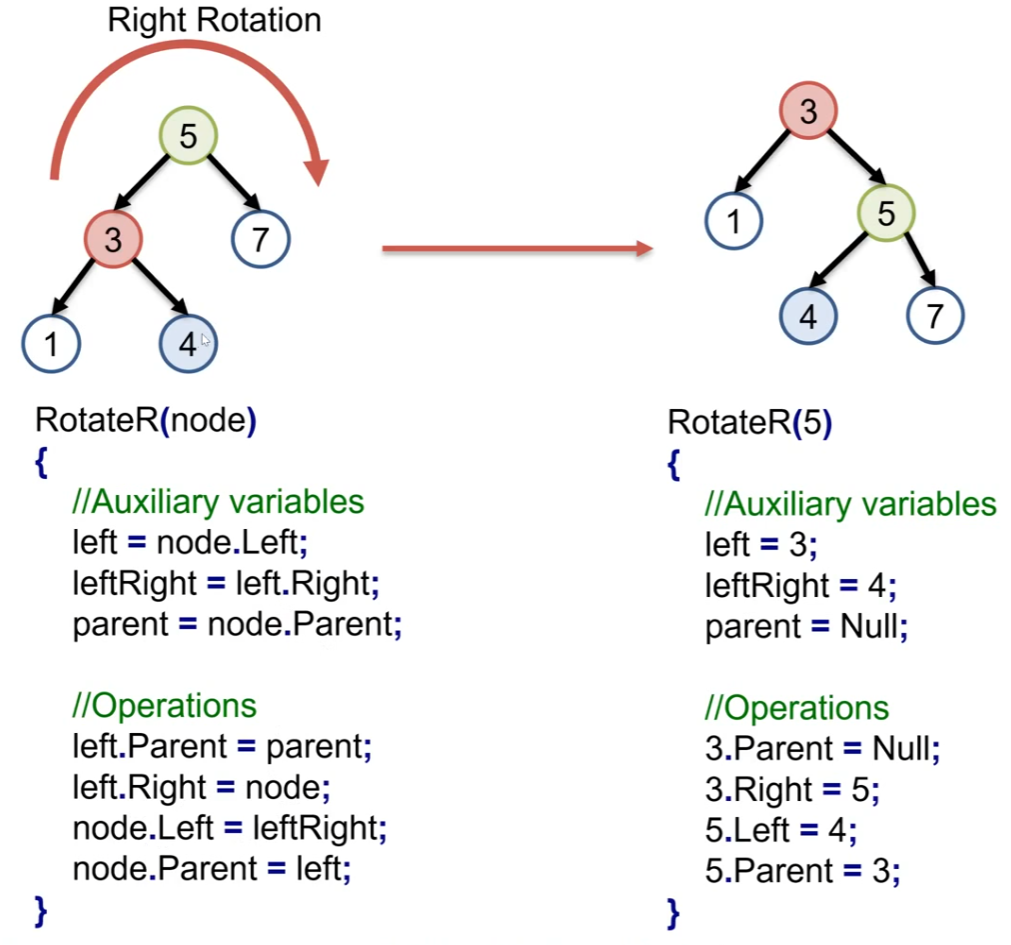
The steps that you should take when trying to balance an AVL tree should be:

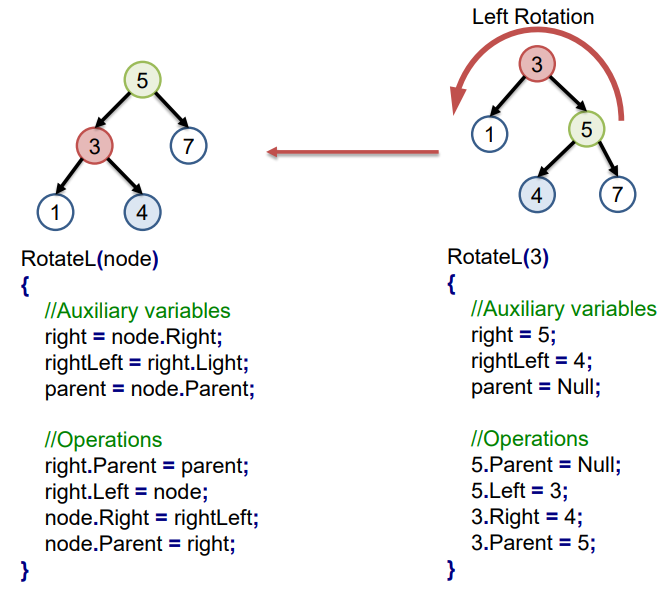
1. Calculate the counters of *all* nodes
2. Find an imbalanced node via a *bottom-up* inspection of the counters of each node
3. Determine the type of imbalance the node is in (hint: check the counters of the unbalanced node and one of its children; refer to slide 10 of the [AVL lecture](https://edstem.org/au/courses/6413/lessons/14203/slides/102738) for all the possible imbalances)
4. Determine and execute the rotations you need to fix that imbalance (hint: refer to slides 11, 19–20 of the [AVL lecture](https://edstem.org/au/courses/6413/lessons/14203/slides/102738) for this)
5. Recalculate *all* the node counters
6. Repeat until there are no more unbalanced nodes in the entire tree

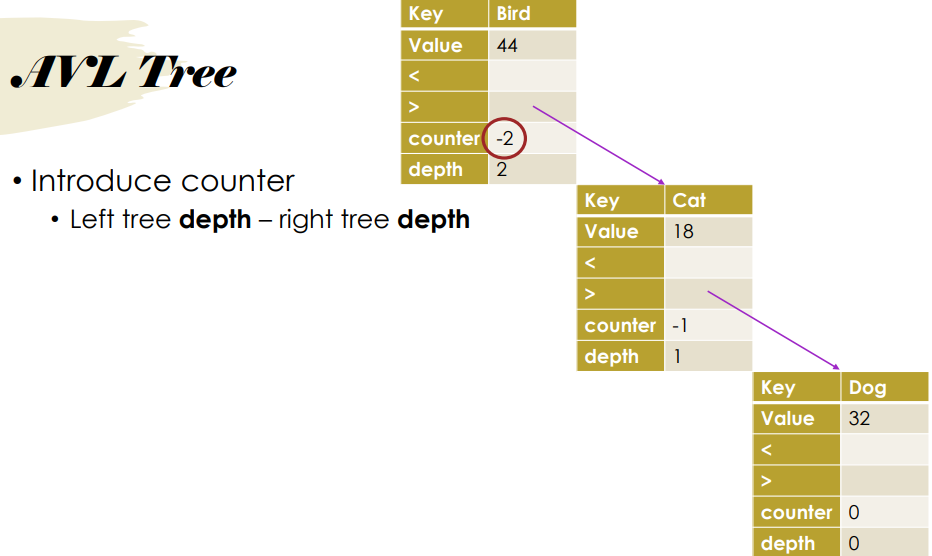
**Complexity: O(logn) to insert, O(1) to balance**

The swap begins at the node that is unbalanced









Depending on the counter number, we do rotation.

Negative negative = left rotation

Positive positive = right rotation

Positive negative = left right rotation

Negative positive = right left rotation

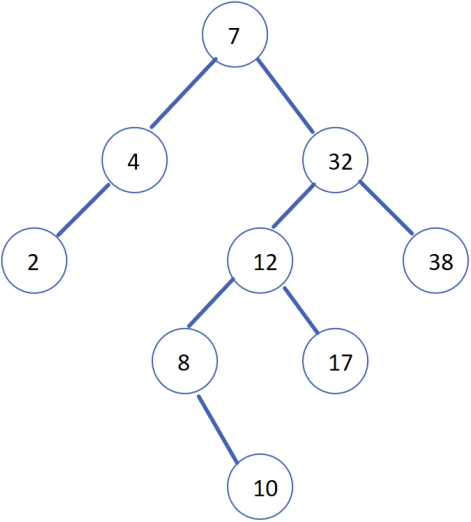
Left rotation

// Swap "right" for "left" if the rotation occurs on the other side of the tree.

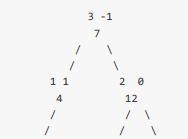
// Shift child up. grandparent->right = child;

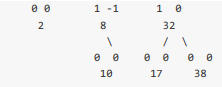
// Swap middle item. parent->right = child->left;

// Shift old parent down. child->left = parent;



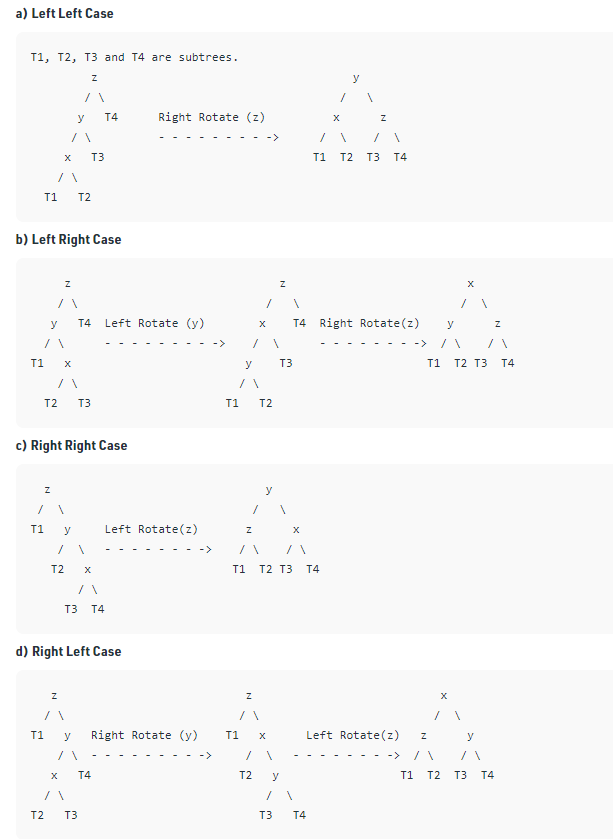
Solution:





Keep track of depth (left) and counter.

Start from the bottom then work up



# **Hashing**

## Chaining

Instead of storing values, they store pointers to linked lists. When a collision happens, it gets added to the corresponding linked list/tree/dynamic array. Disadvantage: If given a bad set of numbers: array size 7, numbers are 7,14,21,28 then the linked list would be long. Ie; occupy a lot of space. Does not require the t > n rule that the other two require.

Worst time complexity: O(m) where m is the size of the linked list.

We can insert at the head or the tail. Best to specify.

## Linear Probing

If there is a collision, then would just slot in the next available slot. Disadvantage: worst case would be O(n), deletion is problematic.

## Double hashing

If there is a collision for a hash of %5 (example), then for the second item, there will be another hash of %7 (example), and the item will be placed at the location of the first hash + second hash.

# **Sorting**

Any sort that uses a swap is not stable!

Stable sorting algorithms maintain relative order of records with equal key values.

## Selection sort

Stability: Not stable

Worst case complexity: O(n^2)

Best case complexity: O(n^2)

Average case complexity: O(n^2)

Pros: Useful when moving items in memory is expensive

Find the largest element in the array, bring it to the last position.

Decrease ‘size’ of array by 1.

Find the largest element within this new size. Bring it to the new last position.

Repeat.

Can do partition at back or partition at front

## Insertion sort

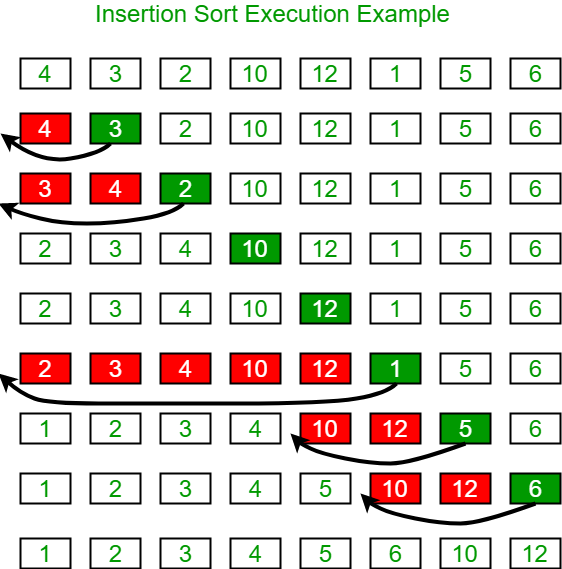
Stability: Is stable

Worst case complexity: O(n^2)

Best case complexity: O(n)

Average case complexity: O(n^2)

Pros: Few steps; generally outperforms O(n log n) algorithms when n is small



A = [6,2,7,4,3,1,9,5,0,8];

Swap 2 and 6, swap 4 and 7, swap 4 and 6

## Count sort

<https://www.youtube.com/watch?v=OKd534EWcdk&ab_channel=CSDojo>

Distribution counting is an unusual approach to sorting

• Does not require key comparisons

• Requirement: Key values within a certain range

• Stable sort

Steps:

Start with array of: • Records, or • Keys + pointers to records

Count number of records associated with each key value (lowest to highest)

Redistribute array elements

Result: • Sorted array • Stable sort

Cons:

Take extra space

Generally less flexible than comparison-based

MSD radix sort can be fiddly if keys are not the same length

Complexity:

Time: O(n) + O(range)

Space: O(n) + O(range) • Specifically, 2\*range + n

Assuming range < n, we can consider this as O(n) • If range > n? O(range)

## Quick sort

Stability: Not stable

Quicksort is an O(n^2) algorithm

Mergesort will always be Θ(nlogn) therefore, for quicksort to also have the same worst-case complexity of Θ(nlogn) with a sorted array, it must choose a pivot where it halves the input at each step. For a sorted array, it is the middle element.

Worst case complexity: O(n^2) - if partition is bad

Best case complexity: O(n log n)

Average case complexity: O(n log n)

Pros: In-place sort, no extra space required

Cons:

• Worst-case complexity: O(n^2)

• Requires random access

<https://people.eng.unimelb.edu.au/ammoffat/ppsaa/c/quicksort.c>

<https://www.youtube.com/watch?v=MZaf_9IZCrc&ab_channel=KCAngKCAng>

<https://www.youtube.com/watch?v=kUon6854joI&ab_channel=Udacity>

Requires a pivot and 2 counters.

Find pivot. (can be anything. Random, last element, middle element)

//this sudo code makes sure the items on the left of the pivot are less than the pivot and the //right is more than the pivot. Repeating this will make sure all elements are sorted.

int i=-1;

int j=0;

while(j<=arraysize-1){

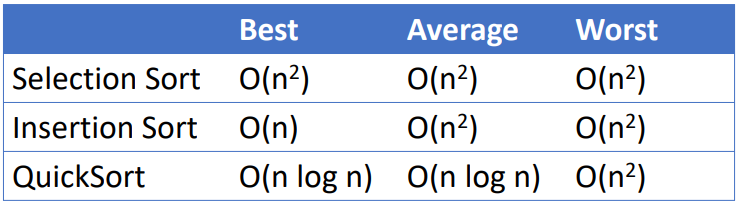
If (array[j] > pivot), increment j.

If (array[j] < pivot), increment i, swap array[i] <> array[j], increment j

}

Move pivot to array[i+1] //now everything on the left of the pivot is smaller. Right is bigger





## Merge sort

Break up the array into singletons. Then slowly merge them one by one.

Stability: Is stable

Worst case complexity: O(nlogn)

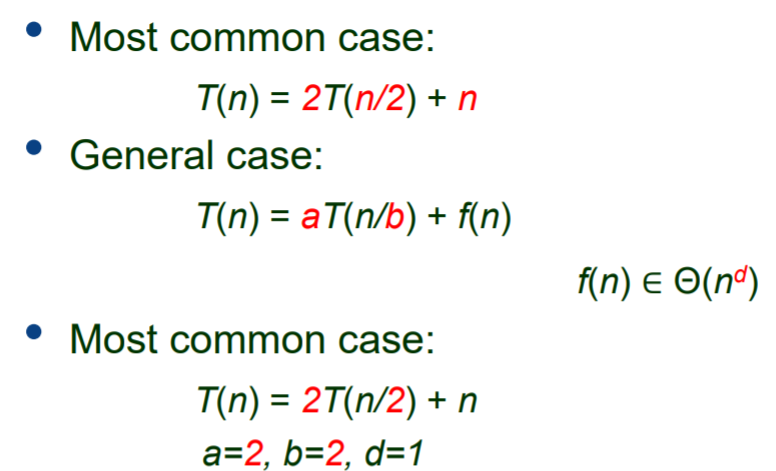
Best case complexity: O(nlogn)

Average case complexity: O(nlogn)

Pros: Can sort huge files on disk , usable for both arrays and lists

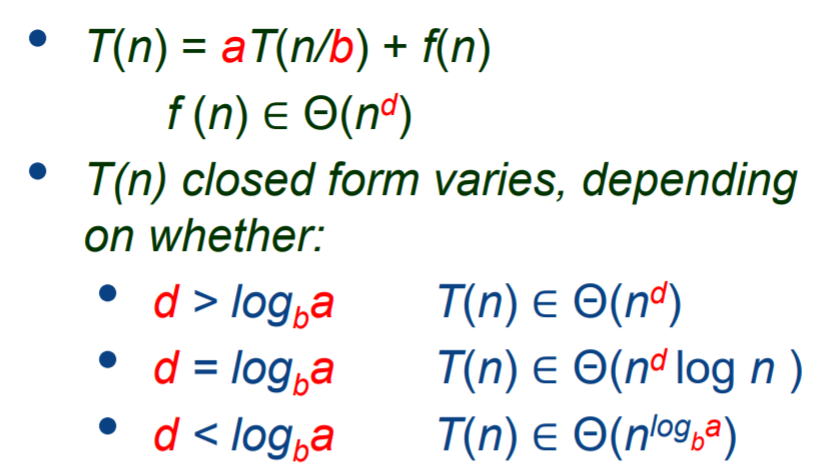
Cons: Requires O(n) extra space, slower than quicksort

## Master theorem



where a is the number of subproblems, each subproblem is size 1/b, for every item there is n^d operations. where a >= 1, b>1.

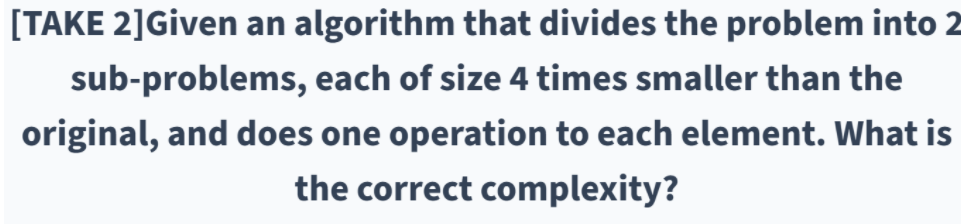
f(n)=O(1), then d = 0



The constant 'a' represents the number of subproblems that we solve in our recursion. In the case of binary search, we divide our array into two halves, but we apply binary search recursively to only one of those halves. Consequently, a = 1 for binary search. The term 'n/b' represents the size of the subproblems that we are solving. In this case, b = 2. The last component, the f(n), represents how much "work" we need to do to divide the problem into subproblems and/or combine the results. For binary search, f(n)=O(1), and d = 0

The recurrence relation for merge sort is: T(n) = 2T(n/2) + O(n)  
  
Quicksort best case: *T*(*n*)=2*T*(*n*/2)+*O*(*n*). The associated complexity is *O*(*nlog*2*n*).

Quicksort worst case: 

Cannot use master theorem for this example: T(n) = T(n-1) + O(1) "do n ×O(1) operations" = O(n)Answer) O(n) because d = 1 and ln(2)/ln(4) = 0.5<1 so case 1 b = 2 a = 4

*T*(*n*)=2*T*(*n*/2)+*O*(*n*)

a = b = 2, d = 1 D is the power of o(n)

so case 2 is called and complexity is O(nlogn)

*T*(*n*)=2*T*(*n*/2)+*O*(1)

a = b = 2, d = 0, 0<1, so case 3 is called and O(nlog-base2(2)) = O(n)

**Priority Queue**

|  | Unsorted Array | Sorted Array | Unsorted List | Sorted List |
| --- | --- | --- | --- | --- |
| Construct | O(n) | O(n^2) | O(n) | O(n^2) |
| Get highest priority | O(n) | O(1) | O(n) | O(1) |

# Heaps

Finding child from parent: 2n+1, 2n+2 considering it starts from 1

It is a complete tree.

parent->key >= child->key, for all children

Does not have to be binary.

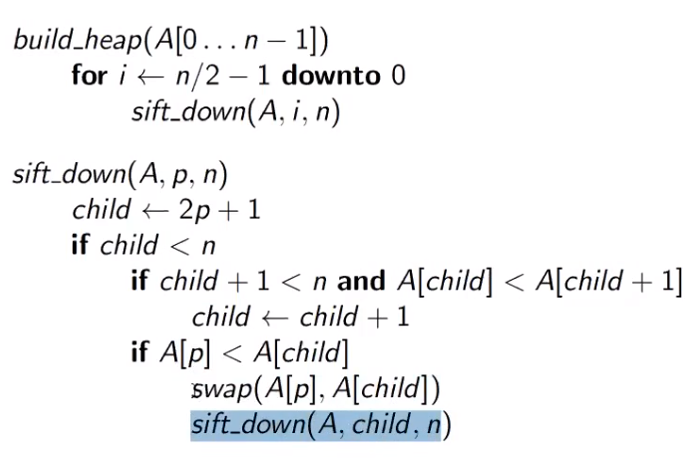
Don’t care if for children, bigger value is on right. Only has to follow the rule that children are smaller than parents

children positions = parent position\*2 and parent position\*2 + 1.

Array starts at 1 not 0.

To make an array a heap array:

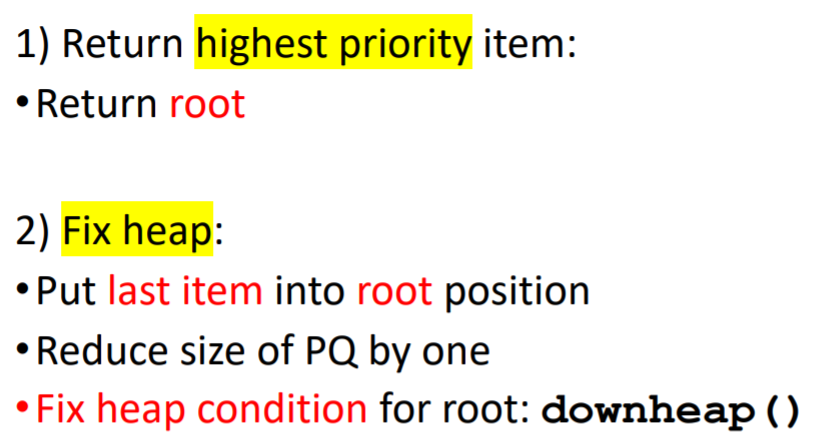
1. Find the last parent node. Which is A[(size/2)-1] Swap it with the larger of two child nodes.
2. I-- (move up the tree) until i=0. But now the new child node may still be bigger than the child’s child node.
3. That is why there is a recursive call.



## Deletemax function

Deletemax function has to: (max refers to highest priority item in queue)

* return max item (root)
* Fix the heap by taking the last item in the array and placing to root
* Now call downheap

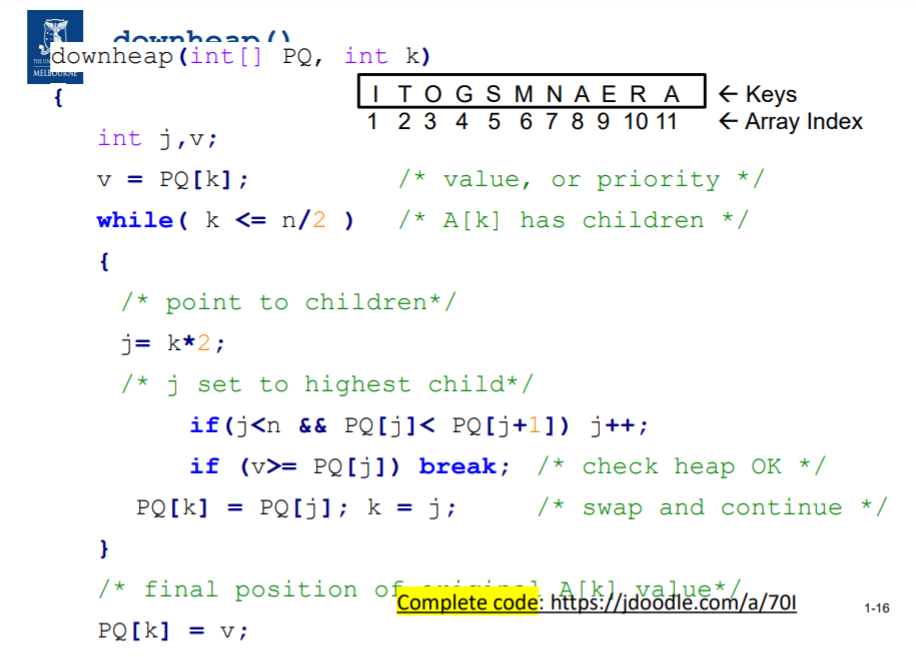


Complexity: O(logn)

## Downheap

Purpose of downheap: Fix the heap

swaps parent with child if child is bigger than parent.

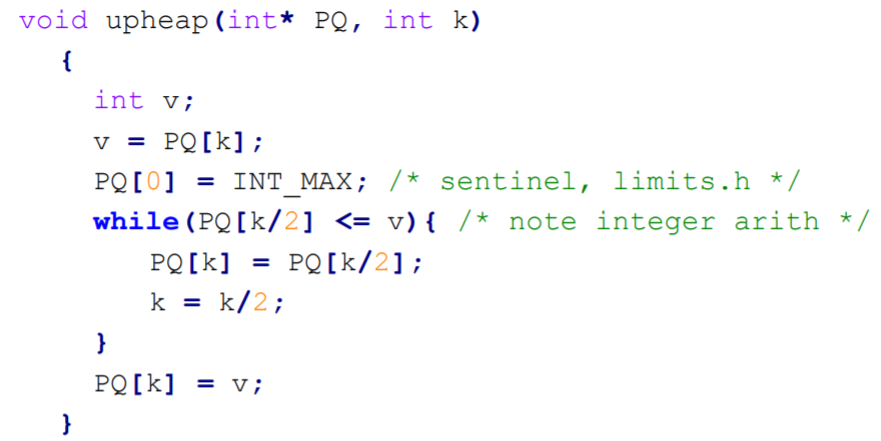


## Upheap

Insert new item to already formed heap: use upheap

how it works:

* Inserts the item at the last slot of the heap: K
* Swaps with parent until it is at the correct position





## 

## Heap sort

Purpose: Make array from a heap form to a sorted form

1. Find the largest number in the heap. (At the top)
2. Swap it with the smallest/last position in the heap. Do not touch the largest number anymore
3. Reorganise the heap so that every parent is bigger than the child (downheap/ upheap)
4. Repeat steps 1-3 for the next highest number. And next smallest/last position in the heap.

Complexity:O(n log(n)) for best/worst/average

Stability: Not stable Naturally

Building max heap complexity:

* Top down: Insert items O(log(n)) one by one and upheap O(log(n)): Total = O( nlog(n) )
* Bottom up: Unordered array (O(n)), downheap O(log(n)) is called from A[n/2] to A[1] to sort. Total = O(n)

There are two alternate methods for building a max-heap from an array of keys. In the top-down approach, we start with an empty heap and insert each element into the heap. At each insertion, we shift the inserted node up to its correct position in the heap. Insert n items into heap of size n:

•Each insertion: O(log n)

•How many insertions? O(n)

•Overall: O( n log n )

In the bottom-up approach, the array of items forms the initial heap structure. We then fix the heap by looking at each subheap whose root is a parent to at least one child, calling downheap() on the root to move it to its correct position in the heap. What is the worst case time complexity of constructing a max-heap using the top-down and bottom-up methods?

Top down: O(nlogn)

Bottom up: O(n)

Pros:

Many applications

Bandwidth Management:

• VoIP, IPTV

Shortest Path Algorithms:

• Pathfinding, navigation, games

Job Scheduling:

• OS, Clusters

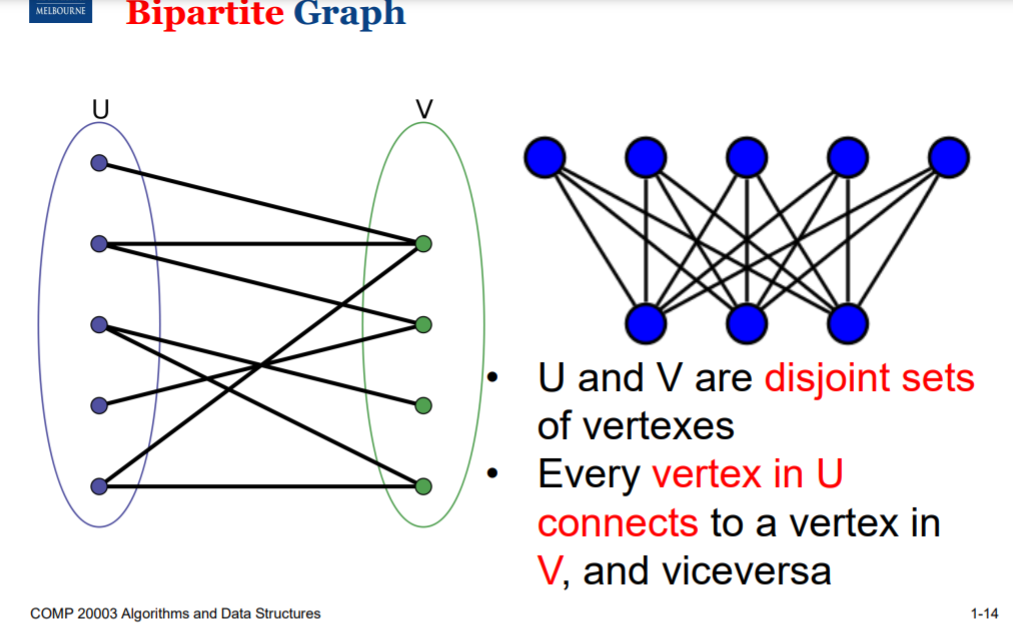
Minimum Spanning Tree algorithm:

• network design

Huffman Code:

• Entropy encoding, compression jpeg, mp3

# **Graphs**



Every vertex in U connects to a vertex in V and vice versa. But there will never be a U-U or V-V connection.

Example: Dating app, kidney donor matching.



Not every vertex can reach every vertex but every vertex in an individual group is strongly connected.

Can have each group have their own selected properties.

**Complete Graph:**

Every vertex is connected to every other vertex DIRECTLY. (Does not connect to itself)

number of edges: V(V-1)/2 (divide by two because it is a directed graph)

**Tree**

Tree, undirected graph that is:

* Connected
* Acyclic
* Any two vertices are connected by exactly one simple path
* All vertices are connected

## Depth First Search (DFS)

Depth-first tree search can be done as:

⚫ In-order

⚫ Pre-order

⚫ Post-order

## Breadth First Search (BFS)

Breadth-first tree search goes layer by layer starting at the root. from left to right.

Doesn't consider path weights.

DFS:

Fill in the visited[] array: V

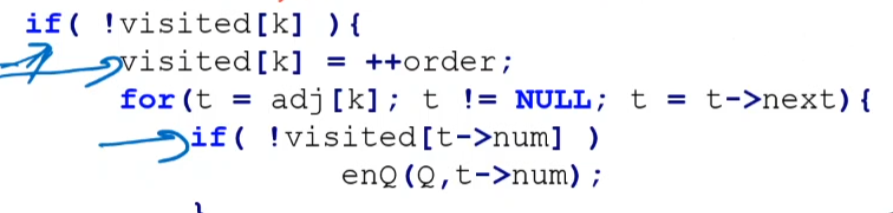
Examine (at most) each edge twice: E

Total: V+E

Depth first search vs Breadth first search:

DFS: Keep visiting the next edge until you cannot, then backtrack. (Below example: 1>2>3>4)

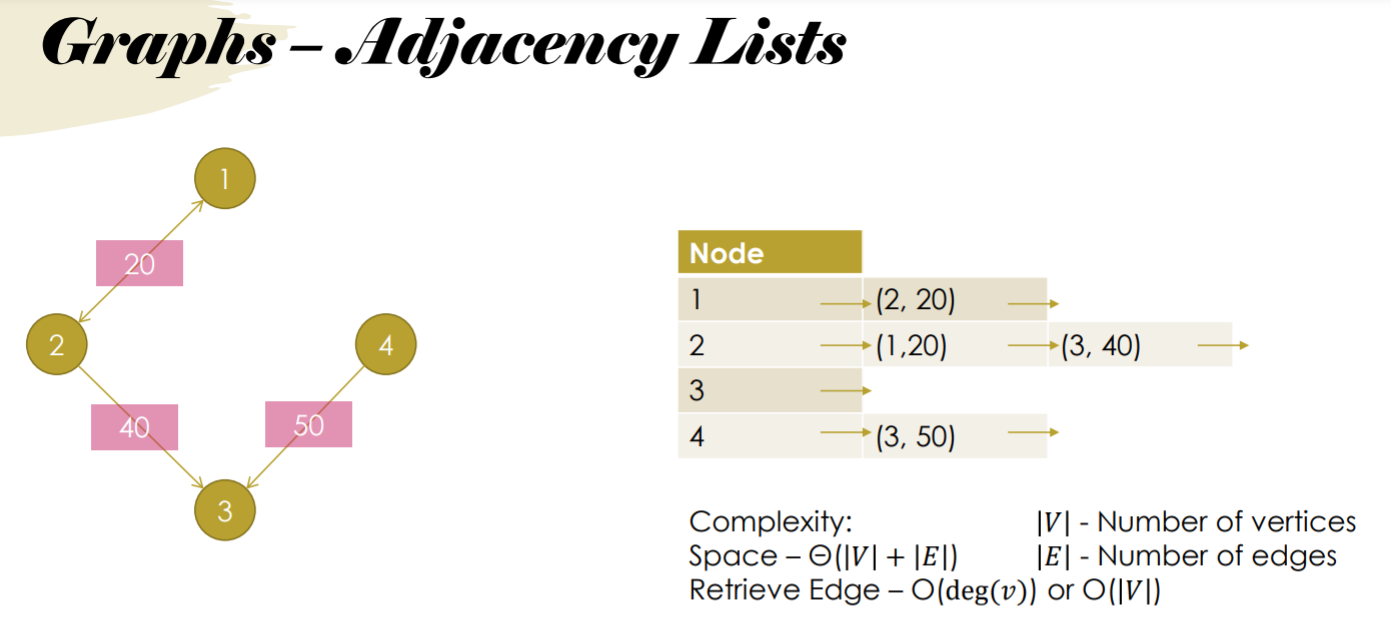
BFS: Visit all possible paths at the same time (Below example: 1>2, 1>3 (same time), then 3>4

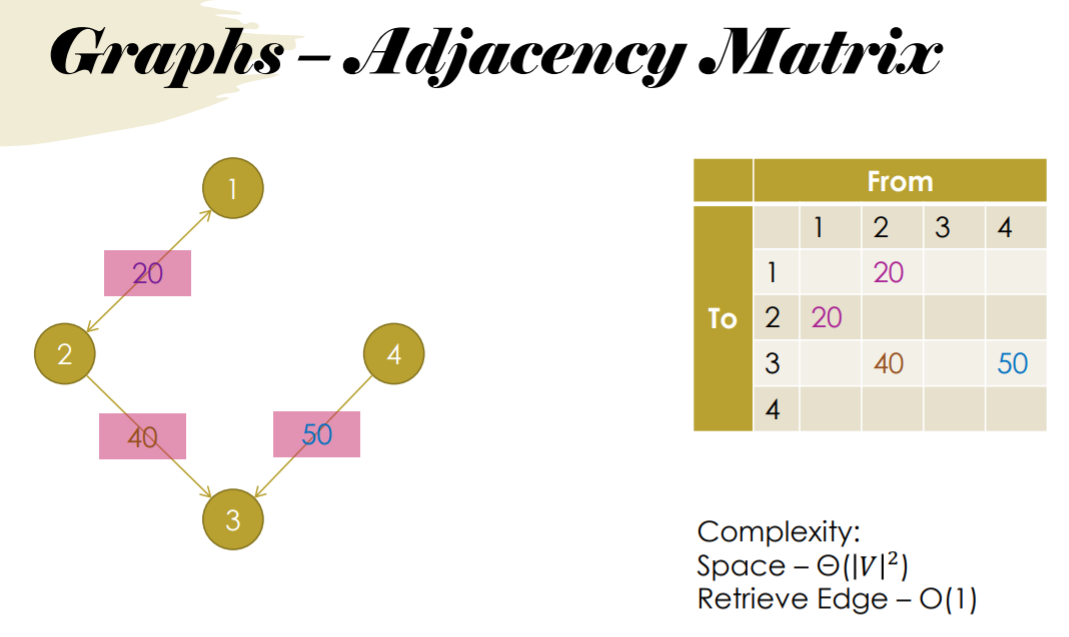


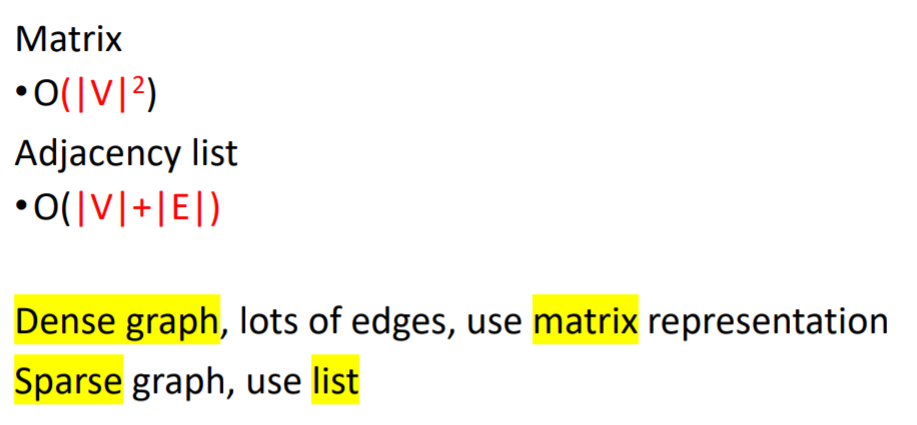
these 3 lines changed

****

## Adjacency list, Matrix, Sparse graph, dense graph







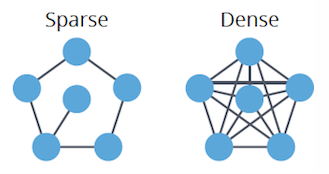
Where |E| number of edges, |V| number of vertices

Dense graph is a graph in which the number of edges is close to the maximal number of edges. Sparse graph is a graph in which the number of edges is close to the minimal number of edges.

Dense graph: complete graph where E = V(V-1)

Sparse graph: E = V

Strongly connected: If every vertex is reachable from every other vertex



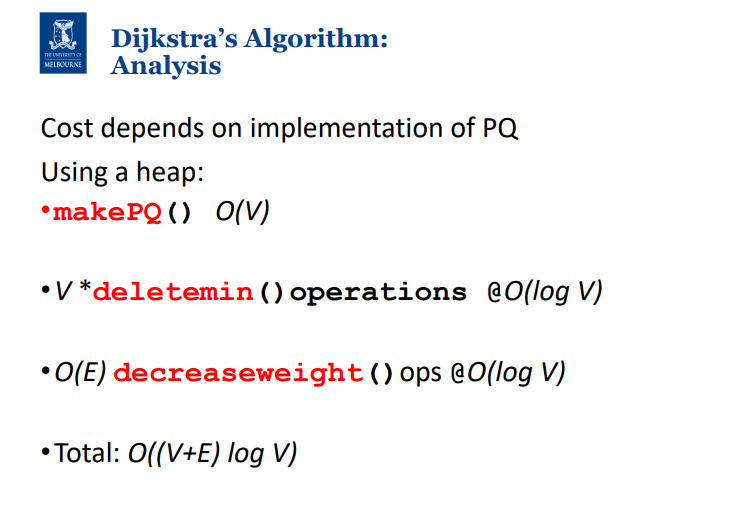
## Single shortest path - Dijkstra

Dijkstra’s Shortest Path algorithm.

Dijkstra's algorithm cannot be applied to graphs with negative edge weights.

<https://www.youtube.com/watch?v=_lHSawdgXpI&ab_channel=MichaelSambol>

Take the start node and give it 0. Look at all neighbours and visit the neighbour with the least weight (smallest value). Mark node as visited in array and update values of all reachable nodes from that visited node . Repeat until all shortest paths are shown.



PQ with linked list:  
- making a PQ will take O(V^2)

- while loop occurs O(V) times, inside while loop:

- deletemin - O(1): which consists of O(1) delete top.

- for loop operates out\_degree(V) times, and inside:

- update operation: O(V), consists of finding the right place to insert O(V) and swapping O(1)

Total = O(V^2) + O(V) \* ( O(1) + Out\_deg(V)\*O(V) )

as E = V \* Out\_deg(V)

Total = O(V^2 + E\*V) -> O(E \* V), if sparse graph then O(V^2), if dense graph then O(E \* V) or O(V^3)

~~In essence, Dijkstra's algorithm on an unweighted graph can be simplified to a breadth-first search. Since the worst case scenario for a BFS is exploring every vertex and edge once, the time complexity ends up being~~

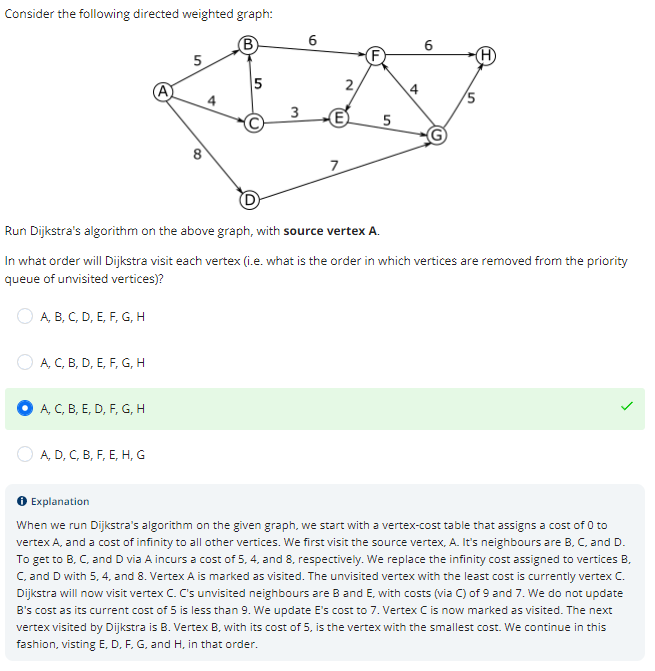
*~~O~~*~~(∣~~*~~E~~*~~∣+∣~~*~~V~~*~~∣).~~

If Dijkstra is applied once for every vertex: 

–O(V3 log V) for dense graphs.

Dijkstra's algorithm cannot be applied to graphs with negative edge weights.

We might use Dijkstra's algorithm to compute all-pairs shortest paths *over* the Floyd-Warshall algorithm if we have a sparse graph with positive edge weights



Give an example of a graph such that running Dijkstra on it would give incorrect distances

negative loop - in this example, dijkstra might find a,c,b,d as the shortest

path, while an infinite path traversing acba would be shorter. This is because

Dijkstra's algorithm assumes the path is not made shorter by following later

negative cycles

## All pairs shortest path

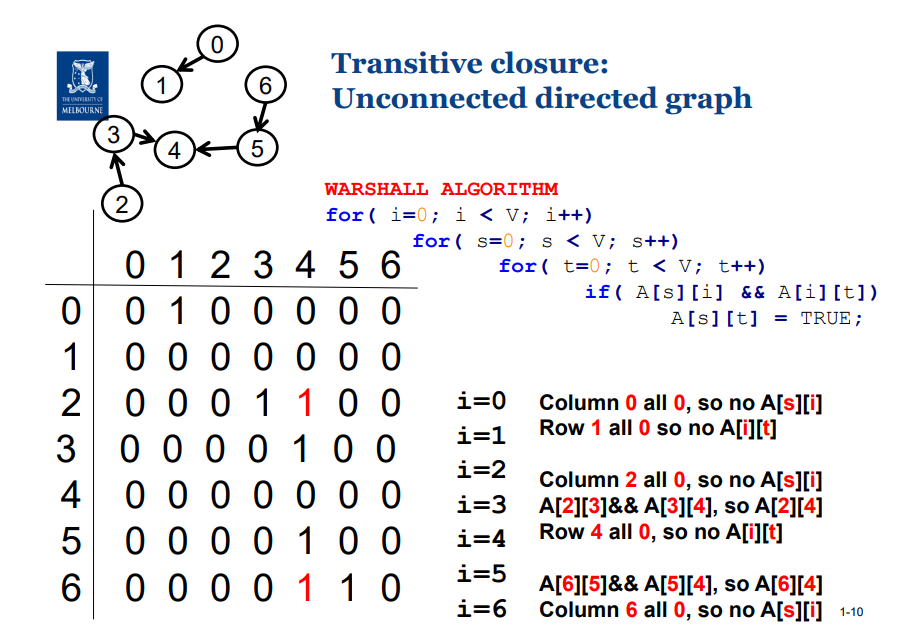
Shortest path from every vertex to every other vertex

The number of edges in the shortest path between any two nodes x and y in the graph cannot be greater than |V|-1

### Warshall

Converts an adjacency matrix to a Transitive closure matrix which shows indirect connectivity.

Row header represents “From” Column header represents “To”



i = intermediate vertex / current vertex (not source or end nodes)

s = source

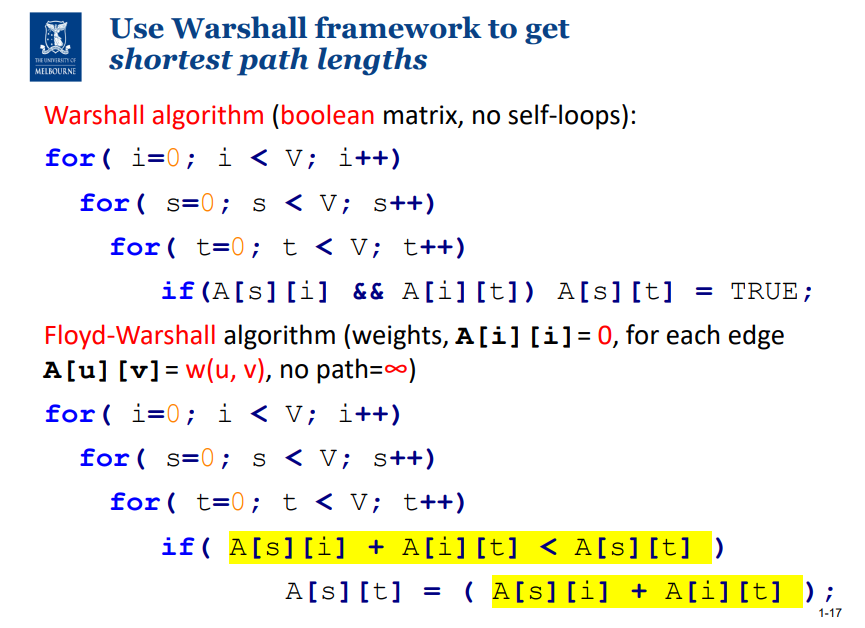
t = to

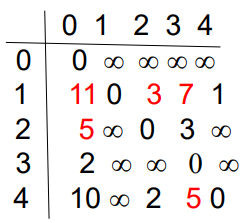
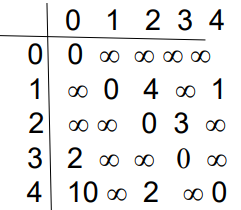
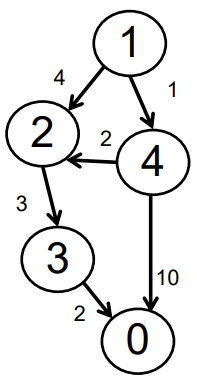
We ignore all source and end nodes. So we first look at node 3 where i = 3, s = 2, t = 4. if we look at the matrix, we are looking at A[2][4] and set that to true because A[2][3] && A[3][4] == TRUE. If there are any more sources/ to for vertex 2, update. Then, move on to next intermediate node (not source or end nodes).

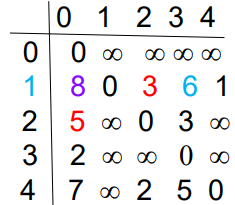
### Floyd-Warshall

<https://www.youtube.com/watch?v=4OQeCuLYj-4&ab_channel=MichaelSambol>

Includes weight to edge. No negative weight allowed







Complexity: Θ(V^3)

~~In terms of the number of edges E in a dense graph G: θ(E^(3/2))~~

~~In terms of the number of edges E in a sparse graph G: θ(E^3)~~

No shortest path has length (number of segments, not

distance) greater than V ‐ 1

Floyd‐Warshall gives

•Distance of shortest path, for all a->x

•But does not establish the actual paths!

Assumed graph representation is matrix For sparse graphs, adjacency list representation, use Johnson’s algorithm

## Minimum spanning tree

Goal of MST vs Shortest paths:

Shortest paths- Here the goal is to reach from start to end. You are concerned about only these 2 points, and optimize your path accordingly.

MST - Here you can start from any point and have to visit all other points in the graph. So, you may not always choose the shortest path for any two points. Instead the focus is to choose the path that will lead you to a shorter path for all the other points.

MST: Minimum sum of edge weights

To find minimum possible weight of a path from u to v in G, construct mst, run Dijkstra's

A tree (no cycles)

Contains every vertex (spans)

minimum cut: cuts the least number of edges

maximum cut: cuts the most number of edges

cross: the edge that gets cut

light edge: lightest edge crossing the cut

respect: the edges that do not get affected during the cut

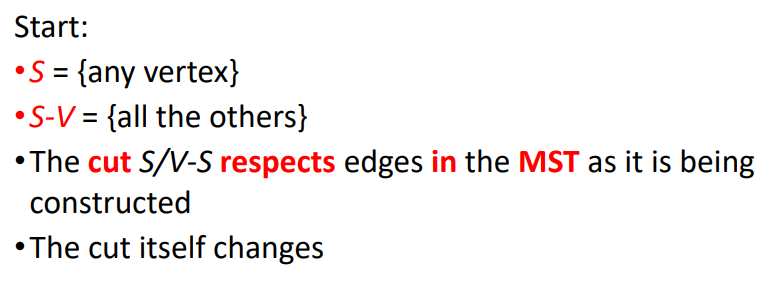
s = alr in MST

v-s = everything else

For a weighted graph with unique edge weights (all edges have a different weight), only one minimum spanning tree exists. Where multiple edges have the same weight, there may or may not be more than one minimum spanning tree for the graph.

Prims and kruskals cannot work on undirected graph?

### Prims

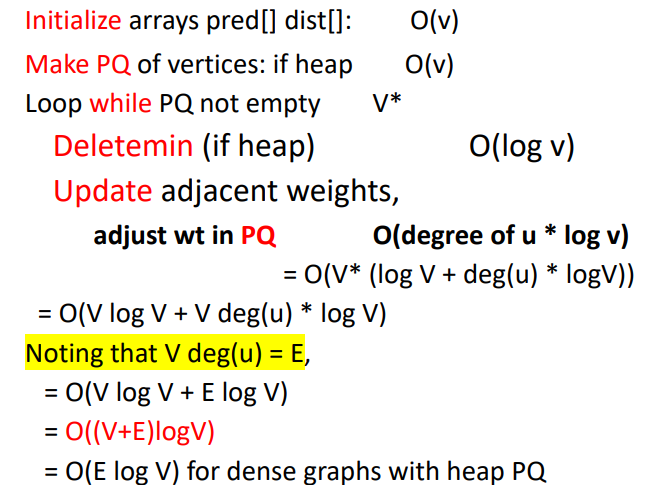


Prims example:

<https://www.youtube.com/watch?v=cplfcGZmX7I&ab_channel=MichaelSambol>

Complexity:

(V+E)(logV)



### Kruskal

Kruskal:

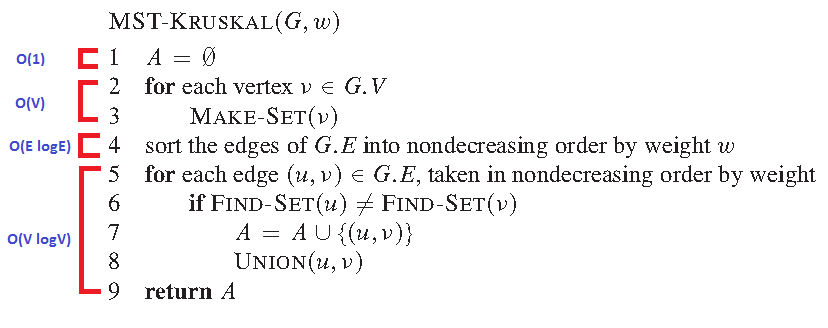
Prim’s algorithm adds the next closest vertex.

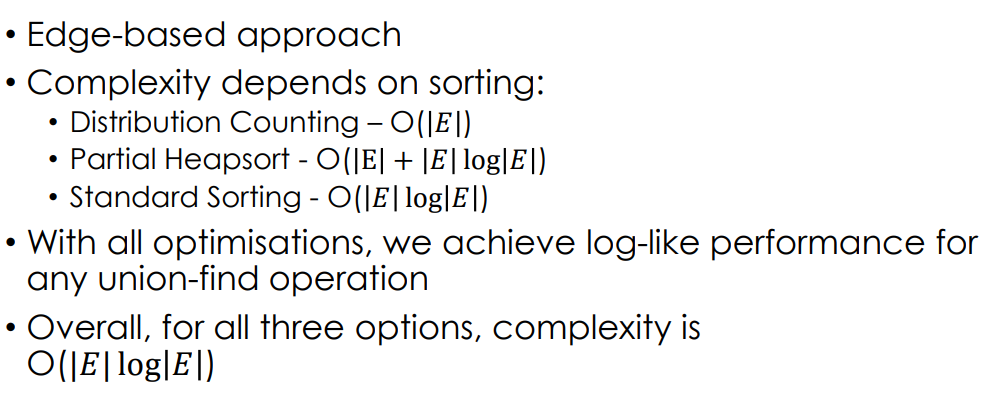
Kruskal’s algorithm adds the next lowest weight edge that doesn’t form a cycle

<https://www.youtube.com/watch?v=71UQH7Pr9kU&ab_channel=MichaelSambol>

Kruskal Complexity:

ElogE





Naive Union: Array based approach merge 4 and 1: Make array[4] = value of 1 merge 4 and 6: Make array[6] = value of 4 O(EV)

Speed union:

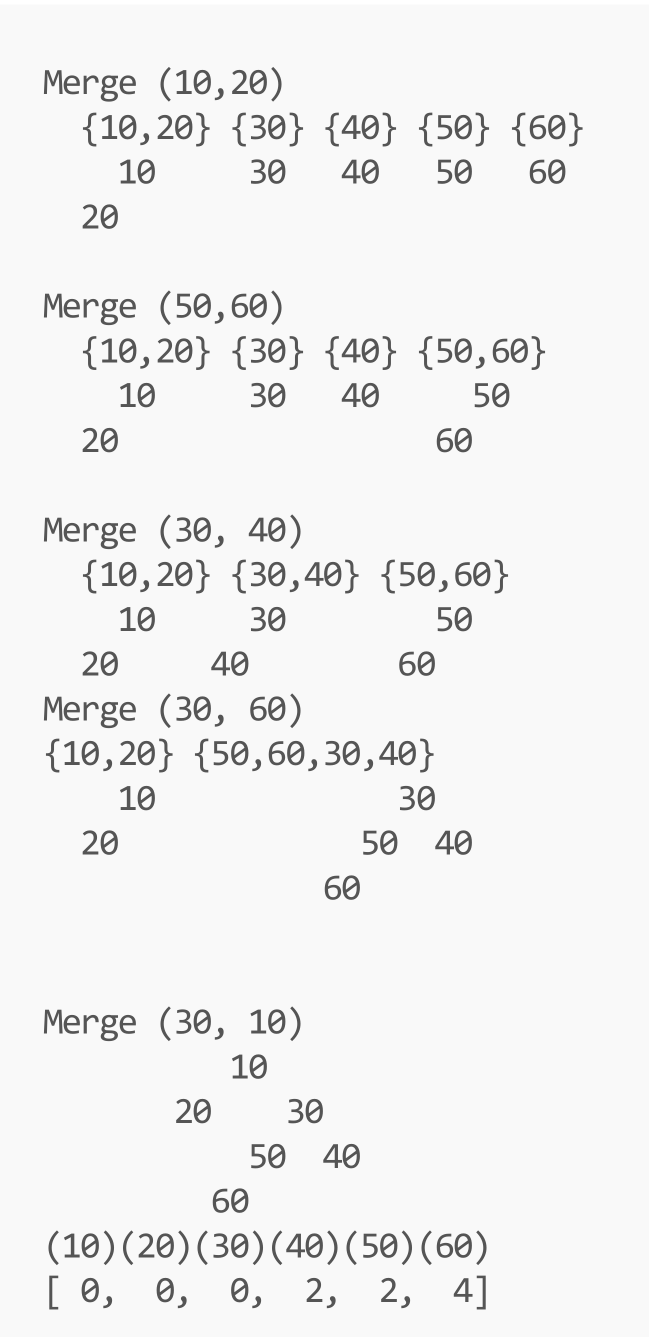
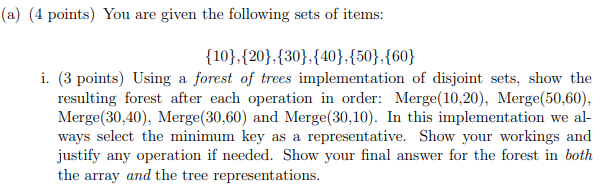
Tree based approach like 6>1 and 4>1

Union by rank & path compression: union find O(logV) in worst case, O(1) on average

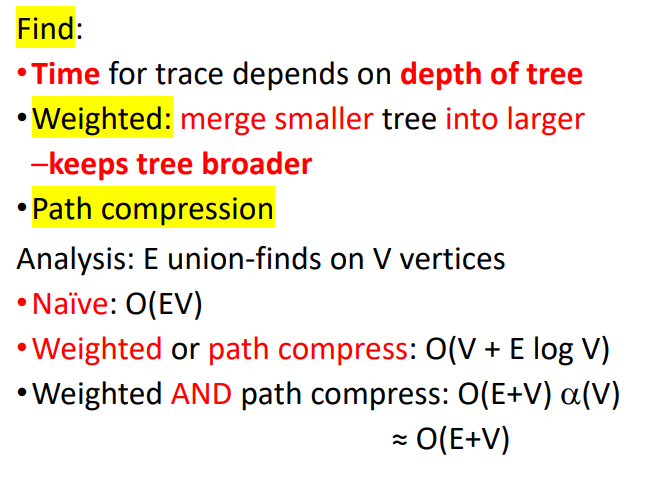
Neither applied = O(V)

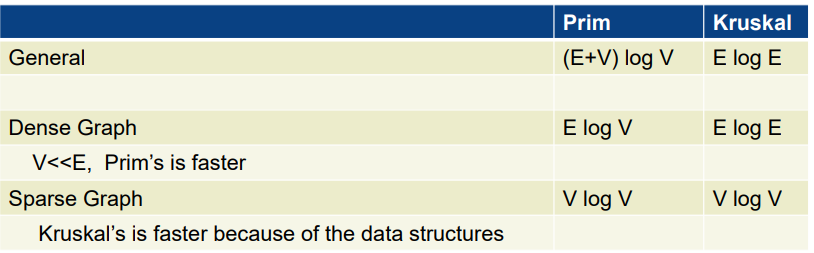
Path compression only: O(∣V∣), though the average case will end up better than O(log|V|)

With union-by-rank only, it's simple to show we have a worst case of O(log|V|)



Array representation: Put index of parent node





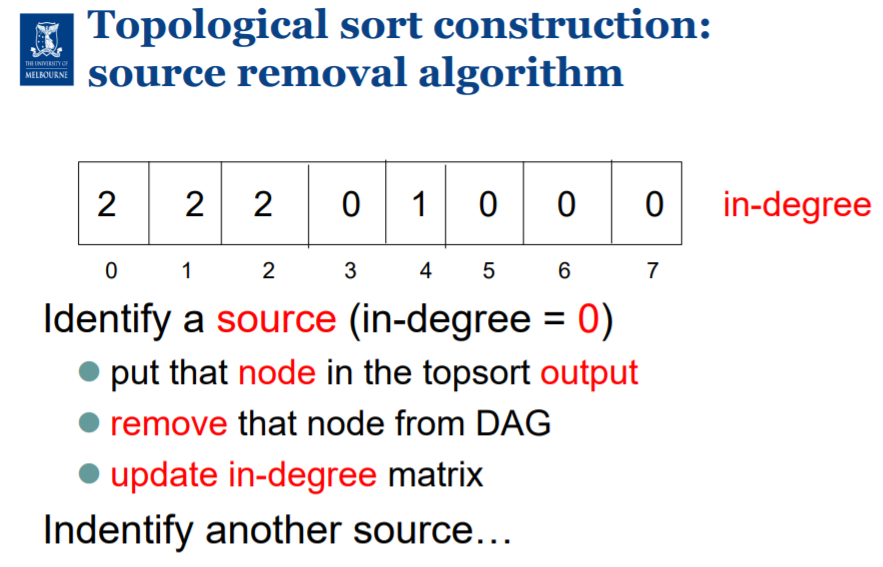
## Toposort

DFS or BFS works

Topological sort: a partial ordering that fulfils certain constraints

vertices are arranged in a single row

In degree: number of vertices that are pointing towards the current vertice.



Multiple possible outcomes.

Topological sorting needs the graph to have at least one source and one sink.

A cyclic directed graph does not have these properties.

Toposort complexity: O(E)

Hamiltonian path: Path whereby each node is visited only once.

If a hamiltonian path exists in the directed acyclic graph, then it’s toposort is unique. How to check: Compute a topological sort, if two consecutive vertices are not connected, it is not unique and a hamiltonian path does not exist. Can be done in linear time.

A Hamiltonian path is a traceable path in a directed or undirected graph that visits each vertex only once. If a Hamiltonian path exists in a DAG, there is one unique topological ordering of vertices.

# **Heuristic (or informed)**

Systematic search algorithms: Consider a large number of search nodes

simultaneously.

Local search algorithms: Work with one (or a few) candidate solutions (search

nodes) at a time.

Blind search does not require any input beyond the graph.

→ Pros and Cons? Pro: No additional work for the programmer. Con: It’s not

called “blind” for nothing . . . same expansion order regardless what the problem

actually is.

Informed search requires as additional input a heuristic function h that maps

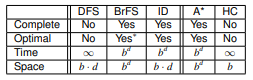
nodes to estimates of their distance.

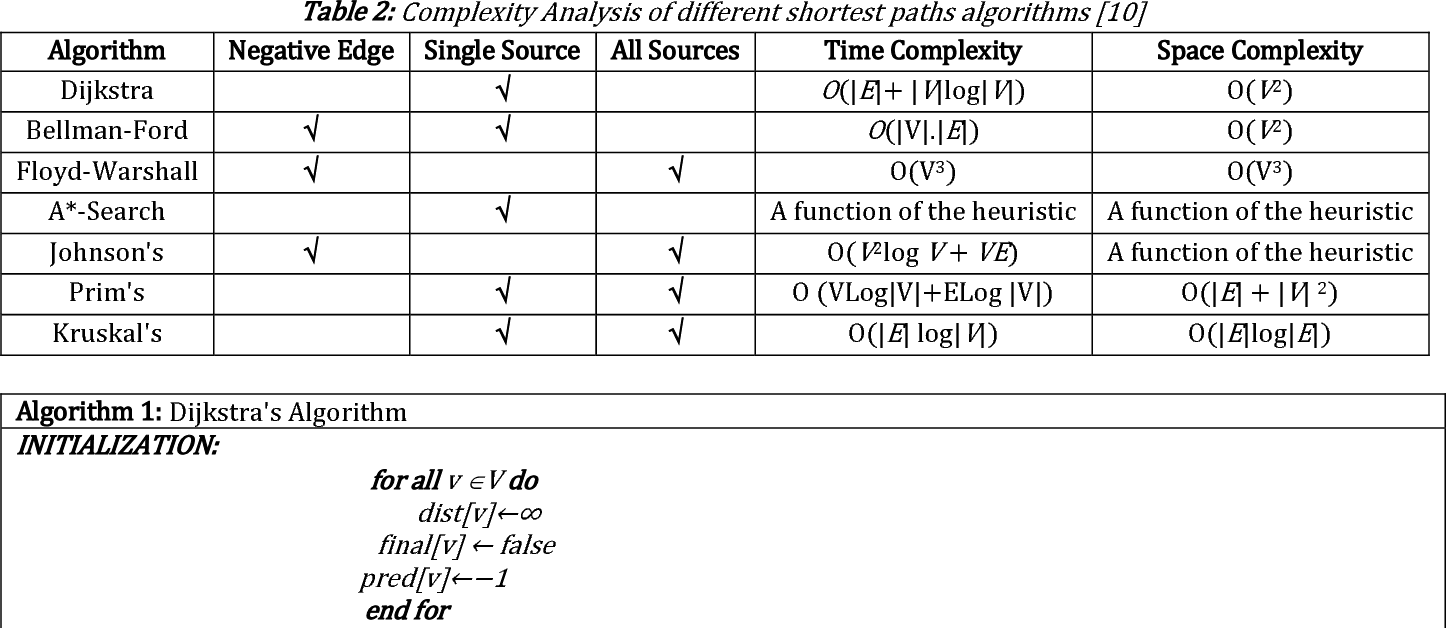
→ Pros and Cons? Pro: Typically more effective in practice. Con: Somebody’s gotta come up with/implement h.

-A\* and best first search algorithm

**Path finding**

A complete search algorithm will always find a solution to a given search problem if one exists. Consider an algorithm for finding a path from an initial state to a goal state. If this algorithm is complete, it is guaranteed to find a path between the initial state and the goal if such a path exists. If this algorithm is optimal, it will find the best possible path between the initial state and the goal. 'Best possible' may mean the shortest or least cost path. Other objectives may be considered when defining the quality of a solution to a search problem.





**N=NP**

P is the class of problems solvable in polynomial time by a deterministic Turing machine.

NP is the class of problems solvable in polynomial time by a non-deterministic Turing machine.

A is polynomial-time reducible to B, A ≤p B iff there is some polynomial-time computable function f , such that for all w, w ∈ A if f (w) ∈ B.

B is NP-hard if every A ∈ NP is reducible to B in polynomial time.

B is NP-complete if B ∈ NP, and B is NP-hard.

Before exam to do:

Quicksort

Open code editor and set up - memory allocation

Open pdf lecture notes

AVL rotation link

P=NP