Теория - СДА

1. Оценка на сложност за алгоритми.

Алгоритъм- Последователност стъпки за решаване на проблем.

Структура от данни- Начин за организиране на данни във формат удобен за ползване.

Видове сложности- по време, по памет

Амортизирана- При многократно изпълнение

Big O - За големи стойности на входните данни каква функция може да се използва за да се ограничат отгоре броя на необходимите операции (пренебрегвайки константи с които може да се умножи функцията)

Подредба: O(1), O(logN), O(N), O(N^2), O(2^N), O(N!)

Kак да измерим сложност- с оценка на сложност или емпирично(увеличаваме линейно големината на входните данни)

long mysteryFunc( int n) {

    if(n == 0) {

    return 0;

    }

    if(n == 1) {

    return 1;

    }

    return mysteryFunc(n-1) + mysteryFunc(n-2);

}

Отговор: О(2^n)

int fact(n) {

if (n <= 1) {

return 1;

}

return n \* fact(n - 1);

}

Отговор: О(n)

<https://github.com/mchechev/data-structures-and-algorithms/blob/master/Exercises/01-ComplexityOfAlgorithms.md>

1. Алгоритми за сортиране (Bubble,Selection,Insertion, Merge, Quick, Counting)

Бавни алгоритми- Bubble,Selection,Insertion О(n^2)

Insertion- най-ефикасен от бавните алгоритми, добър за почти сортиран масив

Бързи алгоритми- Merge, Quick O(n\*logn)

Merge Sort: O(n\*logn) – изчислителна сложност в най-лошия случай, O(n)- сложност по памет

Quick Sort: В средния случай е О(n\*logn), в най-лошия случай е O(n^2)- когато масива е сортиран наобратно, всички елементи са еднакви или се взима за пивот прекалено голям или прекалено малък елемент

Специфични алгоритми- Counting О(n+k)

Merge sort изисква O(n) допълнителна памет, докато quicksort не изисква допълнителна памет.

Quicksort - по-бърз от mergesort за малки масиви, но mergesort е по-добър за големи масиви.

Основна идея:

Метод на мехурчето- гледаме 2 елемента и сравняваме, като като лявото е по-голямо от дясното разменяме. Така правим толкова обхождания колкото е броя на елементите и винаги поставяме най-голямото в края.

Сортиране с пряка селекция- Намираме най-малкия елемент и го местим в началото

Сортиране с вмъкване- Сортиране постепенно на все по-голяма част от масива, като обхождайки несортираната част всеки един елемент го поставяме в сортираната част с намиране на правилното за него място на което сортираният масив остава сортиран.

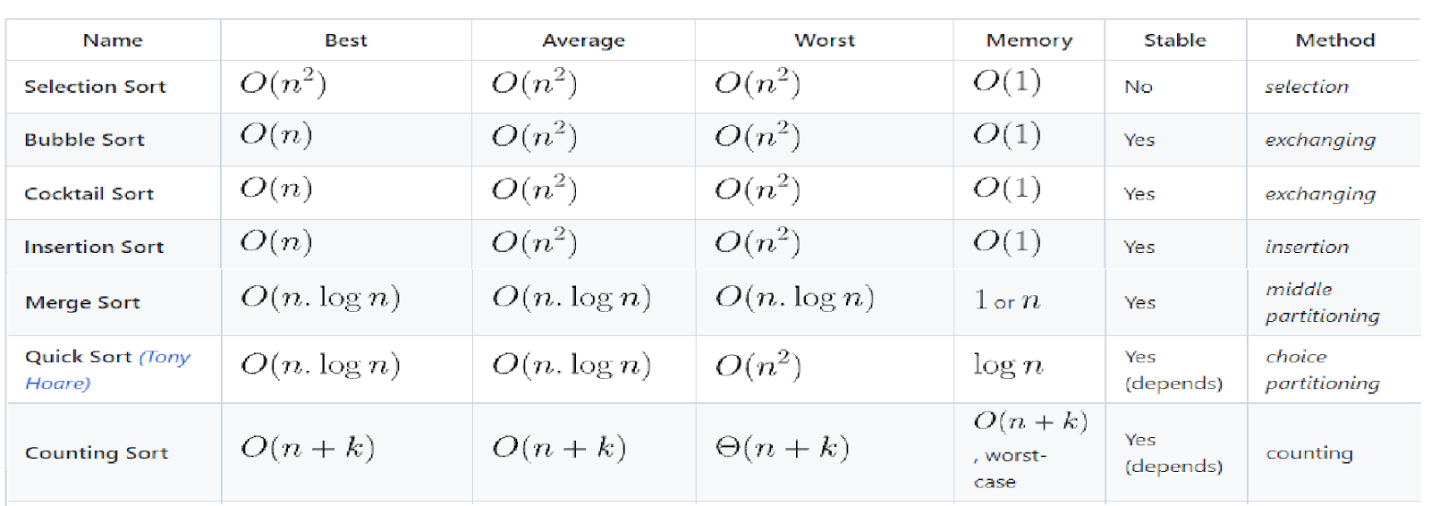
Сортиране чрез сливане- Ако имаме два сортирани масива то със линейна сложност може да ги влеем в един масив. Тогава ако разделим масива който искаме да сортираме на по-малки масиви и на всяка стъпка сливаме два по-малки масива в един голям то за log(N) стъпки ще слеем всички масиви до един масив, като всяка от стъпките е била линейна.

Бързо сортиране- Ако вземем едно произволно число от масива, то с линейна сложност можем да прехвърлим всички по-малки числа от масива да са в ляво на числото, а всички по-големи в дясно.

Рандомизирано бързо сортиране- Справя се с проблема, че точно определена редица прави сложността О(n^2), като използва произволно избиране на елемент за разделяне.

Сортиране с броене- Понеже имаме ограничен брой различни стойности в масива то може да преброим по колко пъти се среща всяка една от тези стойности(с едно обхождане на масива) и след това със второ обхождане да наредим стойностите по техният ред.

Radix Sort- подобно на Сортиране с броене, но за големи числа/дроби



[https://docs.google.com/presentation/d/1BtrGdLTMMHF7uLsdgdIfaXVTtVwY5nht9t865XKKVfU/edit - slide=id.gf717549e4d\_0\_79](https://docs.google.com/presentation/d/1BtrGdLTMMHF7uLsdgdIfaXVTtVwY5nht9t865XKKVfU/edit#slide=id.gf717549e4d_0_79) +In\_place / Stable

1. Алгоритми за търсене(Linear, Binary, Ternary Search)

Видове търсене- Линейно, Двоично, Тристранно, Търсене със скоци, Минимум/Максимум и средна точка, К-тия най-голям елемент

Линейно- При несортиран масив няма как да търсим със сложност по-малка от О(n)!.В случай, че ще извършваме много търсения върху един масив то за да ускорим общото време за търсене първоначално следва да сортираме масива ( за О(nlog(n)) ) и после да търсим със по-ниска сложност O(log(n)).

Двоично - O(log(n)).

Търсене със скоци - (√ n)

К-тия най-голям- Сортираме масива О(nlog(n)) и взимаме съответния елемент;

Как да се справим по-бързо?- Не е нужно да сортираме, трябва само да знаем кои са елементите по-големи и по-малки от к-тия

<https://docs.google.com/presentation/d/1MsrcbJ0688tUyNfBNQaiqVoNHHpUU2A5YRI90ttBGY8/edit#slide=id.p>

1. Свързан списък. Специфични особености. Реализации.

Масив:

Характеристики масив “+”

1. Константна сложност на достъп до всеки един елемент по-индекс
2. Добавяме в края на масива(ако има място) с константна сложност

Характеристики масив “-”

1. Фиксираме размера на масива при създаването му, като ако не е запълнен хабим памет без да я използваме, ако се запълни трябва да създадем нов масив и да копираме.
2. За да премахнем елемент по-средата трябва да преместим всички елементи след него с едно напред.
3. За да добавим елемент трябва да преместим всички елементи след него.

Списък:

Характеристики “+”:

* Добавянето на елемент в началото и в края е с константна сложност
* Размера е динамичен и за това се ползва памет пропорционална на елементите в него
* Може лесно да добавим или премахнем елемент.

“-”:

* Линеен достъп до елемент по индекс
* Използва допълнителна памет(в сравнение с масив) за съхранение на адресите на следващите клетки.



<https://docs.google.com/presentation/d/169xHHqSR8R7Ib8sWpx4fyBsZqa6EndYIbiPLaSZhkuE/edit#slide=id.gf670897c24_0_52>

5.Стек и опашка. Shunting-yard algorithm.

Стек- реализация със свързан списък или с масив

* Create()
* Push()
* Pop()
* Peek()
* IsEmpty()

Опашка- реализация със свързан списък или цикличен масив

* create()
* enqueue()
* dequeue()
* peek()
* isEmpty()

enqueue = InsertAtEnd за свързан списък

dequeue = DeleteFromFront за свързан списък

<https://docs.google.com/presentation/d/1qsrsa_QOl7JT0H4q-U530cGIyVPVgR9VKjjJ6EMbypI/edit#slide=id.gfb71afe97e_0_105>

6.Дървета.

Дърво- Използва се за организиране на данни в йерархия. Основна търсена характеристика: бързо добавяне и търсене на елемент.

-Дървото е ацикличен граф, НЯМА ЦИКЛИ

-Броят на върховете = брой ребра +1

-За всеки връх има само един път от корена до него, защото няма цикли!

* Двоично дърво- дърво в което всеки възел има максимум два наследника

Двоично наредено дърво: двоично дърво, в което всеки възел и наследниците му са в симетрична подредба( например: Всички леви наследници са по-малки от бащата, всички десни по-големи)

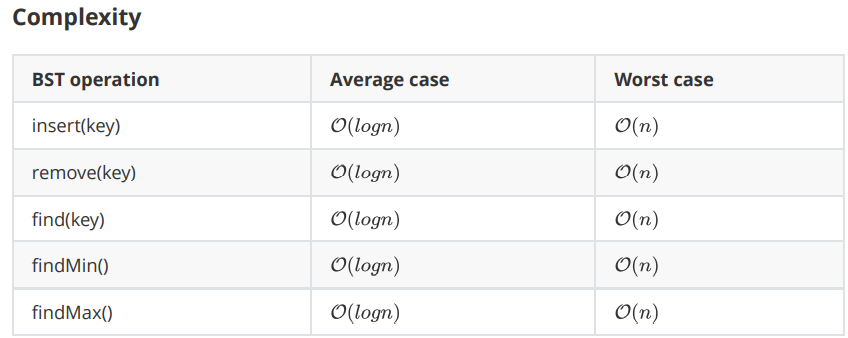
search(X) - търси елемент в дървото

insert(X) - добавя елемент в дървото

remove(X) - изтрива елемент от дървото

print() - принтира дървото в специфичен ред

В най-лошия случай е с О(n).

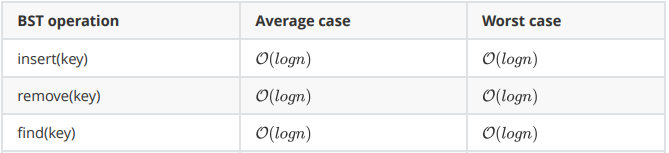


* Балансирани дървета. 2-3-4,Red-Black, АVL, Treap

Балансирано дърво- Дърво, в което за всеки възел имаме свойството, че височината на лявото му поддърво се различава от височината на дясното поддърво с максимум единица.(Ако ще търсим, но и добавяме/изваждаме елементи)

- Дървото преди и след всяка операция задължително се намира в състояние, че за всеки възел X имаме b(X)∈(-1,0,1). Ако стане -2/2 правим ротация.

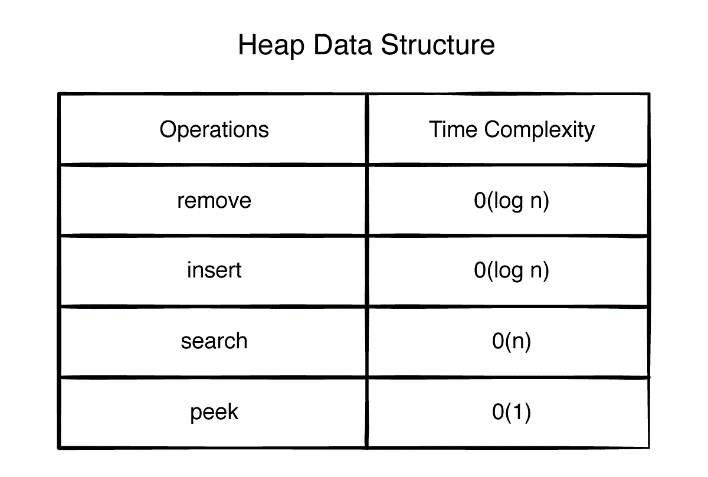
-Минимално число за нива - Log2(N+1)



* A Set stores the elements in sorted order.
* Set stores unique elements.
* Elements can only be inserted and deleted but cannot be modified within the set.
* Sets are implemented using a Red-Black Tree.
* Sets are traversed using iterators.
* Insert, erase and find for O(Log(N))

Heap -Пирамида е почти пълно двоично дърво, като само най-дълбокото ниво не е пълно и то последователно. За пирамидата имаме свойството, че за всеки възел бащата е по-голям и от двете си деца (при максимална пирамида).

* insert(X)
* getMax(X)
* extractMax(X)



if you have an array of size n and you want to build a heap from all items at once, Floyd's algorithm can do it with O(n) complexity.

Priority Queue- Опашка, в която се вкарват последователно елементи, а при изкарване елементите ги получаваме наредени по приоритет (например големина).

Реализации на пирамида: Свързано представяне стандартно за двоично дърво или Представяне с масив (space efficient!)

* Удобно поради последователното запълване на пирамидата
* Корена е на позиция 0, а наследниците му са на позиция 1 и 2
* За всяка позиция от масива i наследниците са 2\*i+1 и 2\*i+2 (ако съществуват).

Пирамидално сортиране- Преобразува масива в пирамида, след което вади един по един елементите от върха на пирамидата и ги поставя сортирани последователно в края на масива. Сортиране със сложност O(n^log(n)) и константна допълнителна памет!

<https://docs.google.com/presentation/d/1duDSuW5Fyj4Lz1naG9cYJk1KWK-N9-BV9vBrW6Fv9pI/edit#slide=id.p> + PostOrder/InOrder/PreOrder

7. Хеширане

* Какво е хеш фунцкия. Свойства на хеш функциите. Какво е колизия. Видове хеш функции. Как да се справяме с колизии. Приложения на хеширането.
* Структури от данни използващи хеш функции: Хеш Таблици и Хеш Мапове.

Хеш функция- Функция, която преобразува обект в число в определени граници

Хеш множество- Ненаредена структура от данни, която може да запомня обекти, като позволява търсенето за това дали обект е вече добавен да става с сложност О(1) в средният случай.

Хеш таблица-Структура от данни, която съдържа двойки (ключ, стойност), която позволява добавяне и изваждане на нови двойки и търсене по ключ със сложности 0(1) в средният случай.

Колизии - два обекта имат една и съща хеш стойност h(k1)=h(k2)

Birthday Problem- 50%

Oсновни стратегии- Запазване на няколко стойности в една клетка(търсене със списък) , Използване на следваща празна клетка, Използване на няколко хеширащи функции

Linear Probing- взимане не следваща празна клетка (Паркиране на Кнут)

За да е оптимално търсенето - трябва да се поддържа N/M < ½

За това следваме следната процедура:

* Ако N/M > ½ удвояваме големината на таблицата
* Ако N/M < ⅛ намаляме на половина големината на таблизата

При намаляване или увеличаване на големината на таблицата за всички елементи до момента се преизчислява хеш функцията.

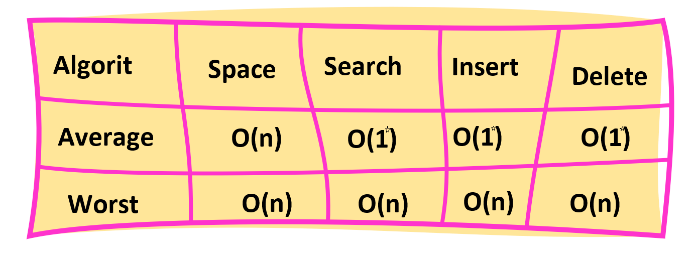
Изтриване в хеш таблица- Вместо изтриване отбелязваме за изтрито.

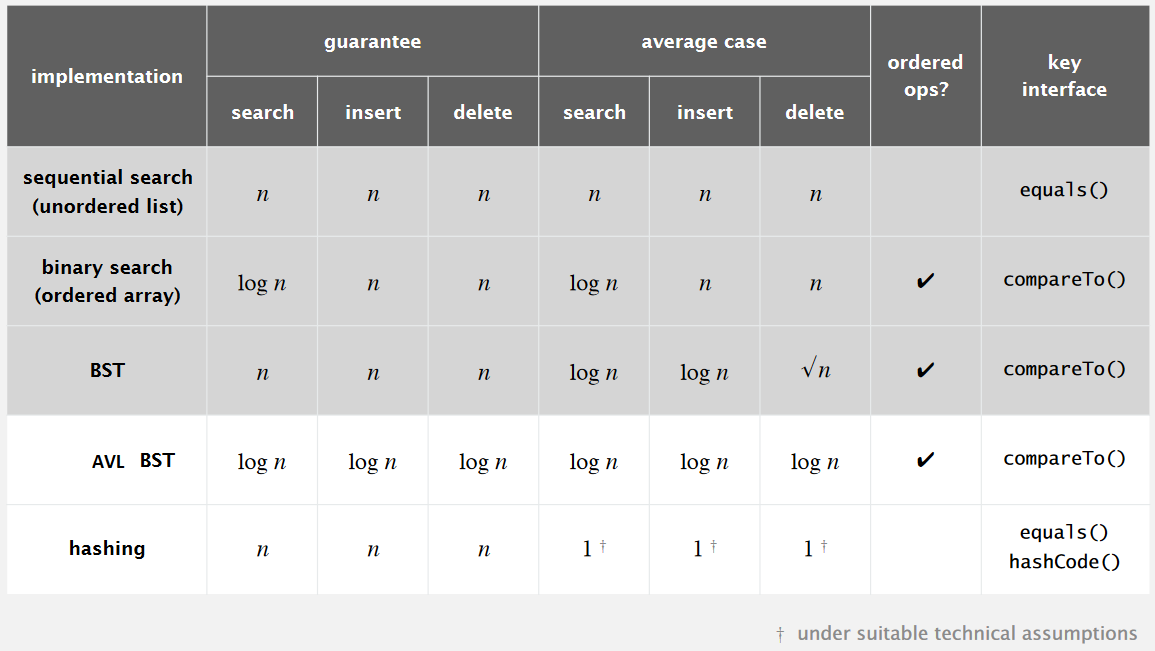
Двойно хеширане-Имаме две(или повече) хеш функции и когато се получи колизия със първата се изчислява хеш от втората хеш функция и се търси да се добави стойността на място хеш1+хеш2, ако и то е заето се търси място на хеш1+2\*хеш2 и т.н. Докато се намери място.

Универсално хеширане: (Sha, MD и т.н са прекалено бавни)

h(x) = ((a\*x+b) mod p) mod n

Където n e големината на таблицата. p e просто число по-голямо от n, а и b са произволни числа по-малки от p





Rolling Hash (Rabin Karp) is a hash function in which the input is hashed in a window that moves through the input.

Хеш таблиците (unordered\_map) и балансираните дървета (map) са доста оптимални структури за съхранение и търсене.

При балансираните дървета имаме гарантирана сложност в най-лошият случай от O(log(n)) за основните операции(добавяне, изтриване, търсене) докато при хеш таблицата нямаме такава гаранция за сложност в най-лошият случай, но практически работи в доста реални случаи работи по-добре и по-бързо от балансираните дървета(в средният случай).

<https://docs.google.com/presentation/d/1hC6o6OHeZQxBnSwoSgyzYypjzMygQKjZoEoAu4w76Fs/edit#slide=id.g102b3710b55_0_41>

8.Граф

* Какво е граф и основни представяния и имплементации
* Алгоритми за търсене в дълбочина и широчина в граф
* Топологично сортиране
* Цикъл в граф. Ойлеров и Хамилтонов цикъл в граф.
* Алгоритми за Минимално покриващо дърво ( Prim, Kruskal )
* Търсене на най-кратък път в граф. Наивен алгоритъм. Алгоритъм на Dijkstra.

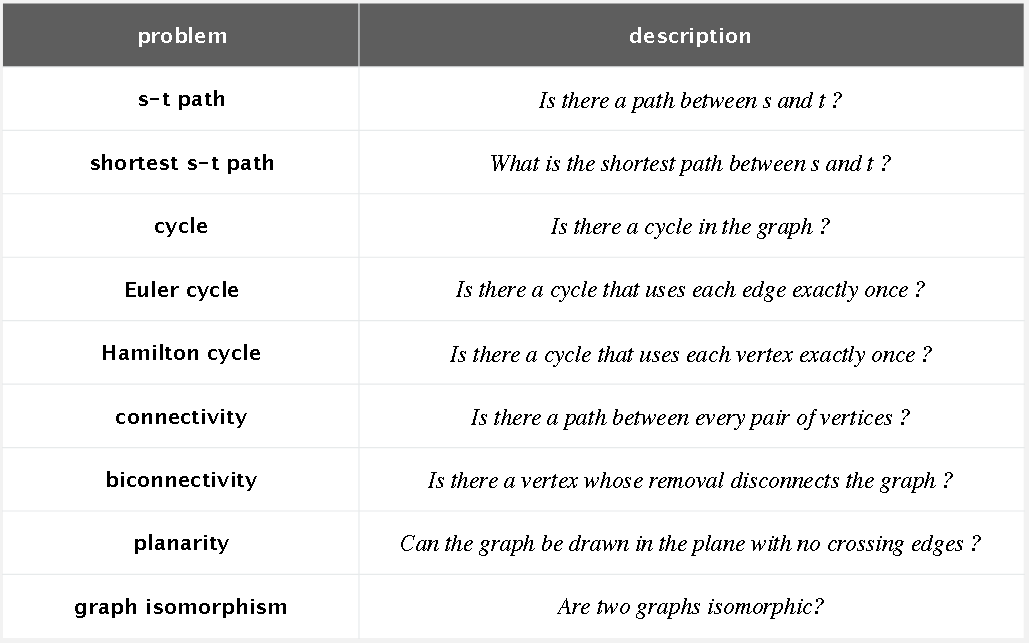
Граф- Нелинейна структура от данни, съдържаща обекти-точки, свързани помежду си с връзки - ребра. (Множество от върхове, свързани с ребра)

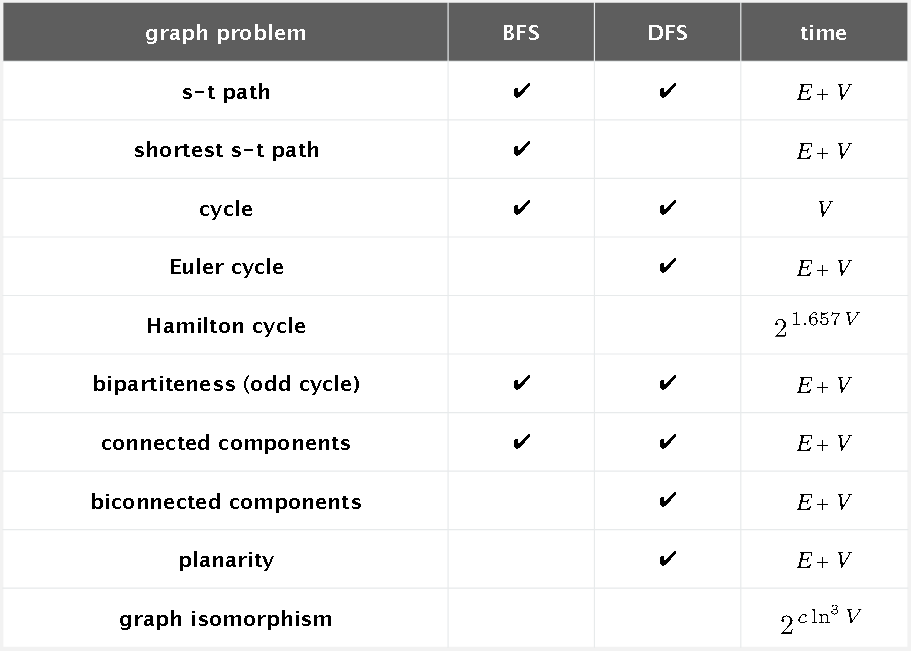
Път в граф- Последователност от върхове в граф, свързани с ребро, без да се повтаря ребро.

Свързаност- Два върха са свързани ако съществува път между тях

Цикъл - път с дължина повече от 1, който започва и свършва с един и същи възел

Граф, в който има повече от едно ребро между два върха, се нарича мултиграф ("multigraph")



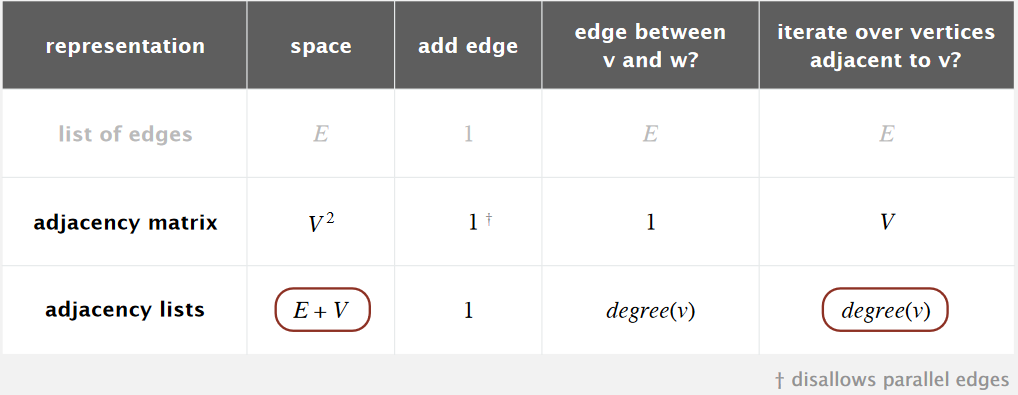


Complete Graphs – A simple graph of vertices having exactly one edge between each pair of vertices is called a complete graph

The maximum number of edges in an undirected graph is n(n-1)/2

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.



Топологична наредба(насочен ацикличен граф DAG)- Да се наредят възлите на графа така, така че всяко насочено ребро да започва от възел по-напред в редицата и да отива във възел, който е по-назад в редицата.

Time Complexity : O(V + E)

Видове графи

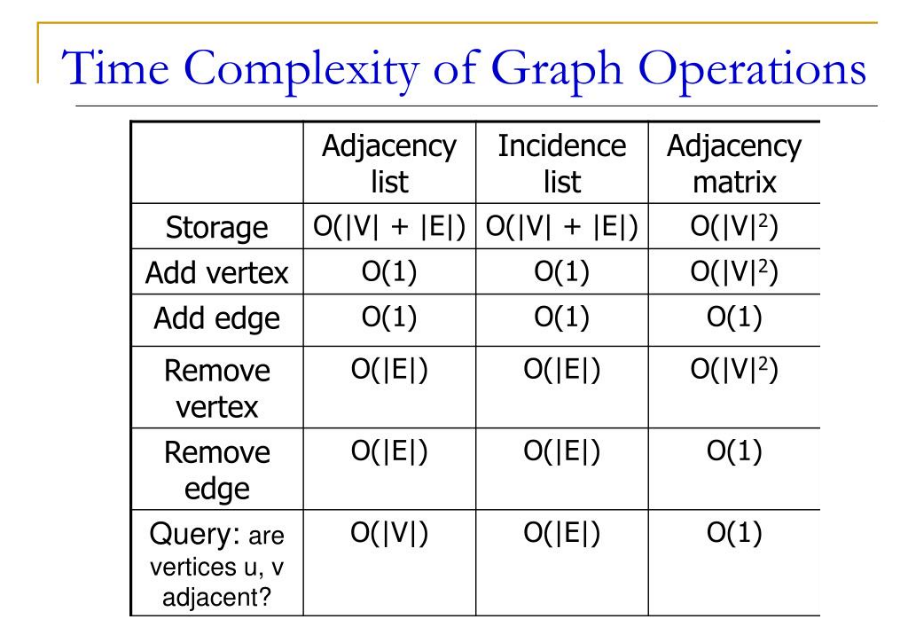
* насочен/ненасочен
* с тегла по ребрата/без тегла по ребрата

Представяне на графи

* с матрица на съседство
* със списък на съседи

Обхождане на графи

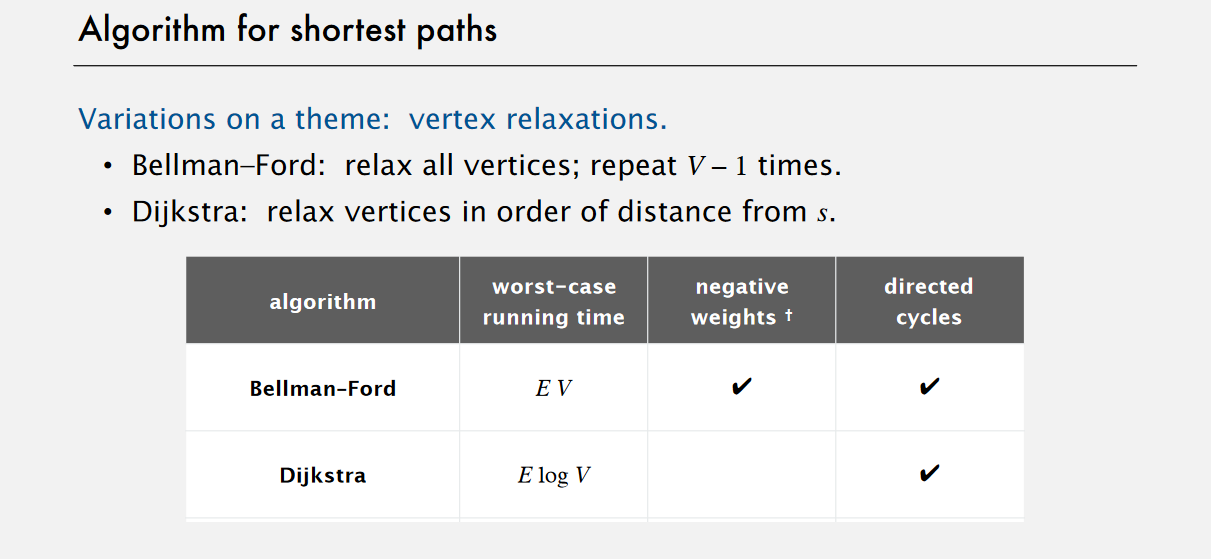
* dfs
* bfs (Минимален път в непретеглен граф)



Алгоритъм на Дийкстра – Намира най-къс път от един връх в графа до всички останали(работи за насочен граф с неотрицателни тегла).

Основна идея: Подобен на търсене в ширина, но вместо с опашка с приоритетна опашка.

Time Complexity : O(V + E\*logV)



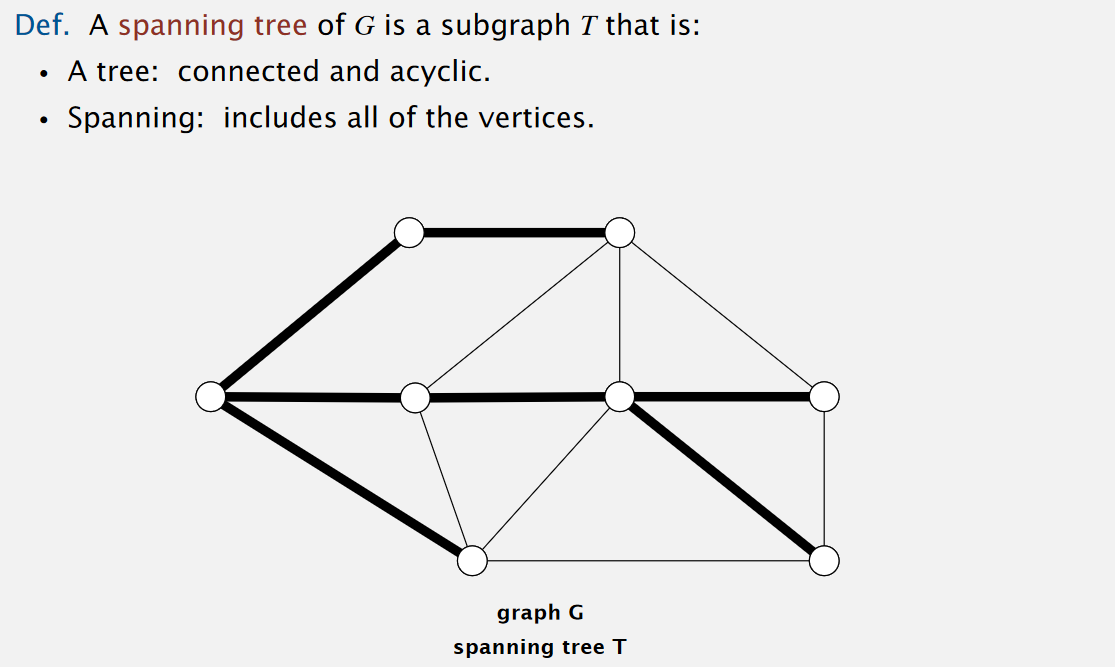
Белман-Форд- to check if there is a negative cycle in the graph:

Time Complexity : O(V\*E)

Time Complexity : O(V^3) , E=V^2

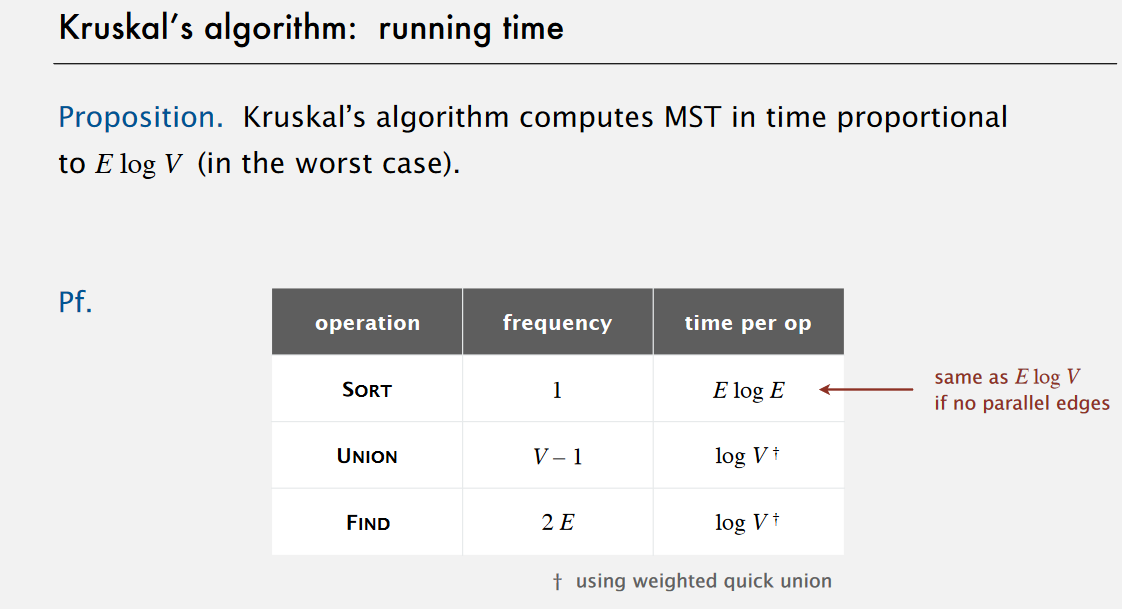
Floyd Warshall- алгоритъм за намиране на най-късите пътища в претеглена графика с положителни или отрицателни тегла на ръбовете (но без отрицателни цикли). При едно изпълнение на алгоритъма ще се намерят дължините (общите тегла) на най-късите пътища между всички двойки върхове.

Минимално покриващо дърво

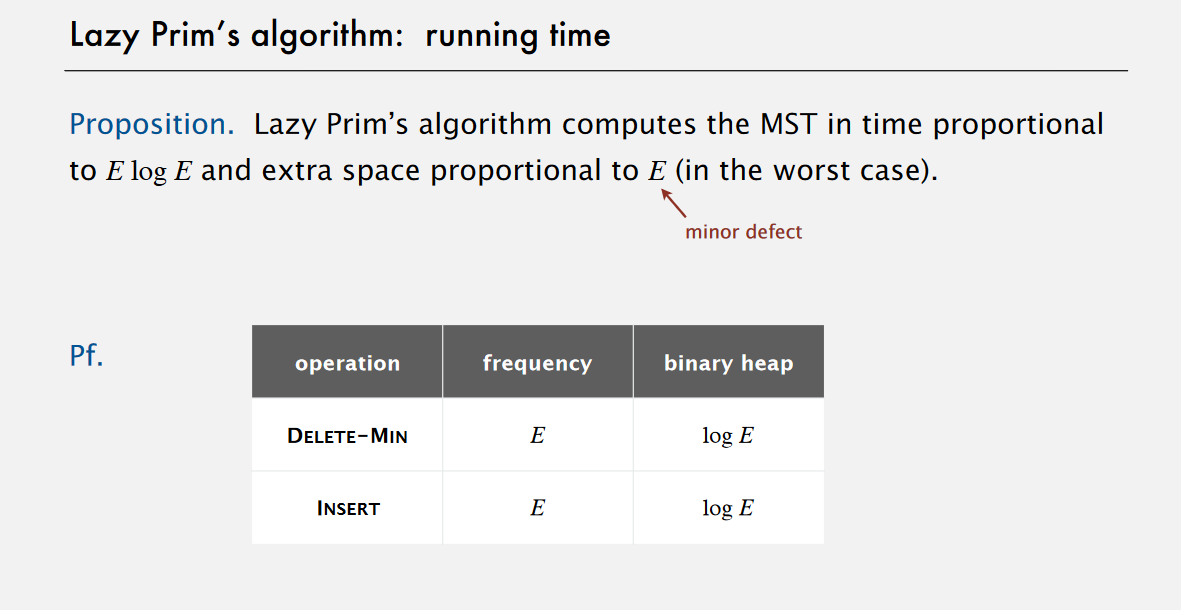


Алгоритъм на Крускал

* Сортираме всички ребра в нарастващ ред.
* Докато покриващото дърво няма V-1 ребра
  + Избираме следващо по-големина ребро от графа
  + Ако то не създава цикъл със вече избраните ребра за покриващото дърво, включваме реброто като част от покриващото дърво

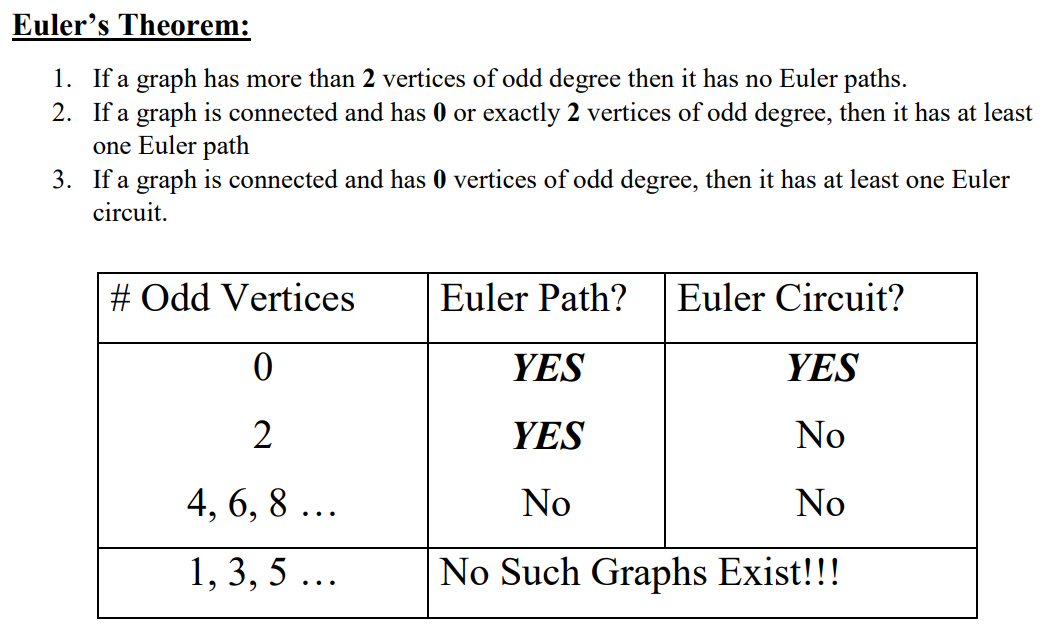


Алгоритъм на Прим



Ойлеров път(Алгоритъм на Флеъри- сложност Е^2 - Започваме от един от двата нечетни върха и обхождаме графа премахвайки последователно ребра като винаги ако е възможно се избира ребро което няма с изтриването си да прекъсне връзката в графа )

Ойлеров път(Алгоритъм на Хелхолзер – сложност Е - Обхождаме графа с dfs,като винаги като минем по път го изтриваме. Обхождането започва от единият връх с нечетен брой ребра и ще свърши в другият, като при самото обхождане вероятно част от ребрата няма да бъдат включени. Връщаме се назад и за всеки възел който все още има ребра правим обхождане като вмъкваме намереният цикъл в намереният път до сега.)



Хамилтонов път (V! – Backtracking- рекурсивно от всеки връх пробваме всеки един възможен случай)

Проблема за Хамилтонов път в граф е NP Complete проблем.

NP проблем: Проблем за който може да проверим дали дадено решение е вярно за полиномиално време.

NP Complete задача:Задача за която ако се намери решение за полиномиално време, то това ще доведе до автоматично намиране на решение за за всички NP проблеми.

NP Hard проблем- Проблеми, който са поне толкова трудни колкото най-тежките проблеми в NP (т.е. дори и да се намери полиномиално решение за NP complete проблем, то за тях не означава автоматично, че ще има полиномиално решение за този проблем).

Пример за Хард: Задача за пътуващият търговец (Traveling Salesman Problem)- Да се намери път, който обхожда всички върхове точно 1 път и е с минимална дължина.

<https://docs.google.com/presentation/d/1PDmZg0a7GfO3lO0FPY3UcxP8xDH8jJTxxrYoXulb3N4/edit#slide=id.g1053066e81b_0_43>

<https://docs.google.com/presentation/d/1ncBHS6i7ID_a5scs2ubJg0-G8cLHdrsYyN1GOTNSFlE/edit#slide=id.g10407e22651_0_26>

<https://docs.google.com/presentation/d/1AfTQ7l6qW0stRSjIudtvyX8ZGEjt_W2rn--LAzfqURI/edit#slide=id.g10b80440ed3_0_6>