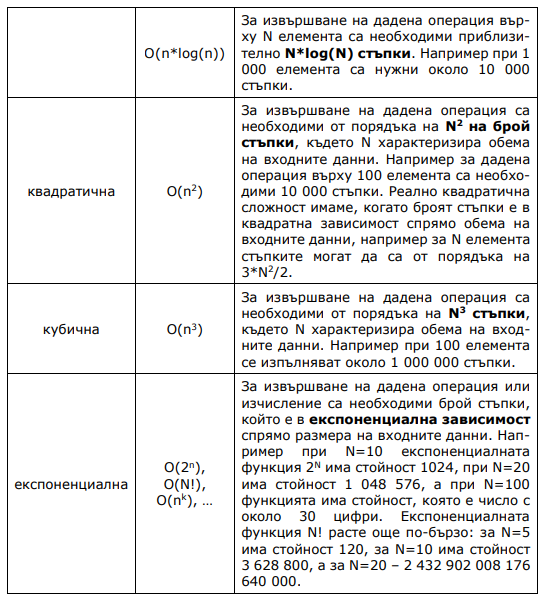
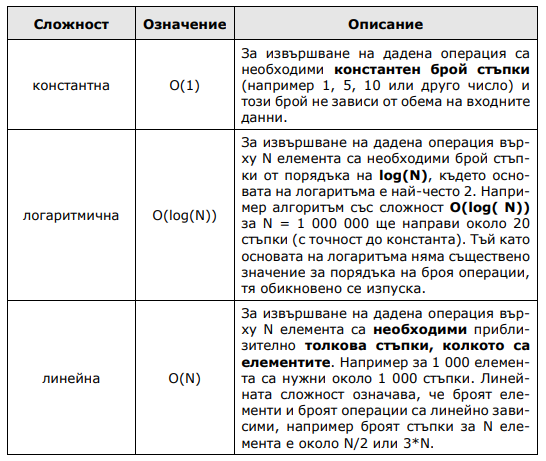
**Data Structures & Algorithms**

**Big O**

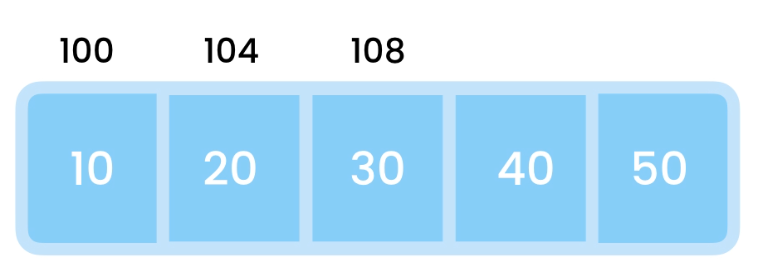
Сложност на алгоритъм се използва, за да опишем бързината на изпълнение на даден алгоритъм



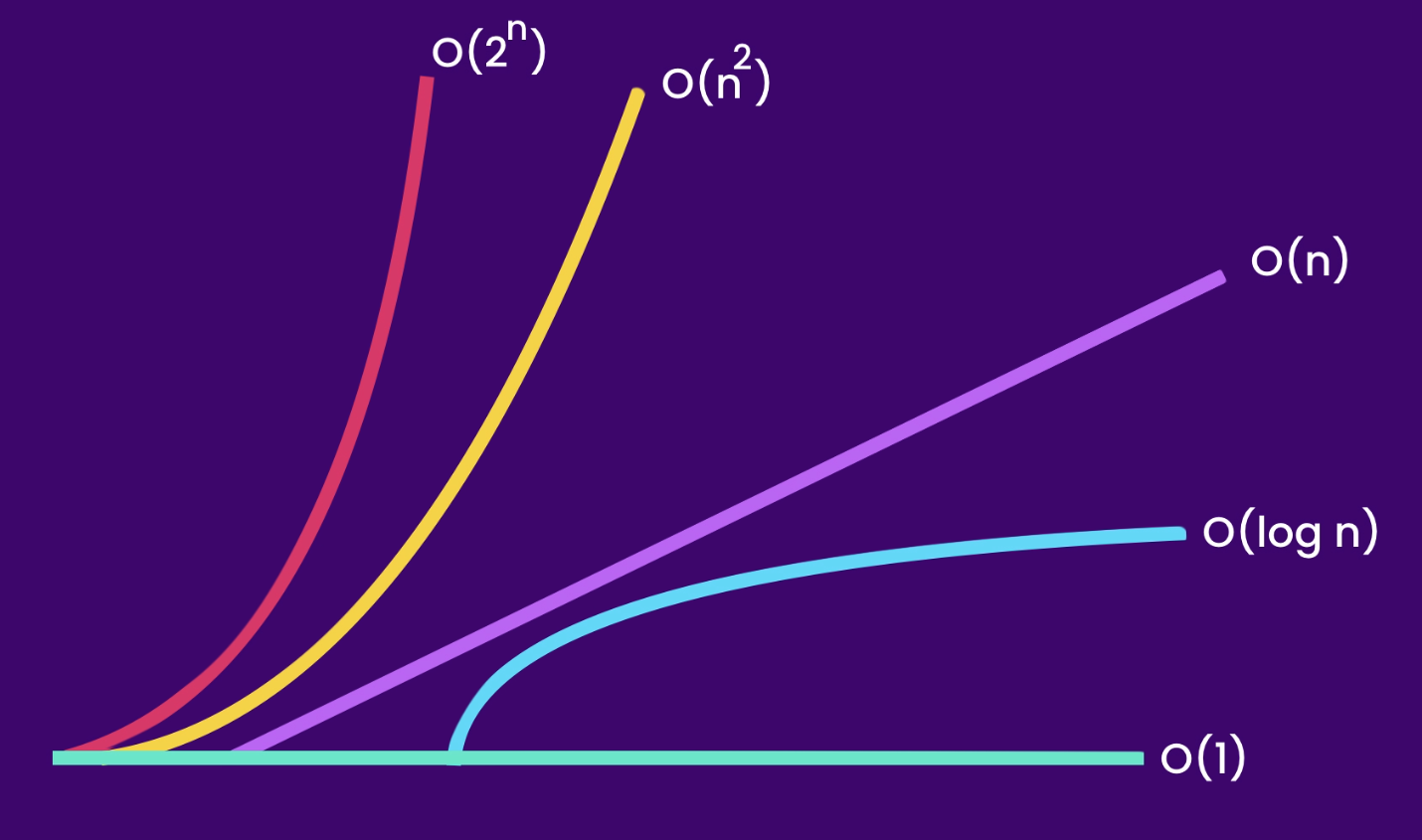
Квадратична използваме при вложени цикъла. Когато имаме два вложени цикъла (общо три) – тогава имаме кубична

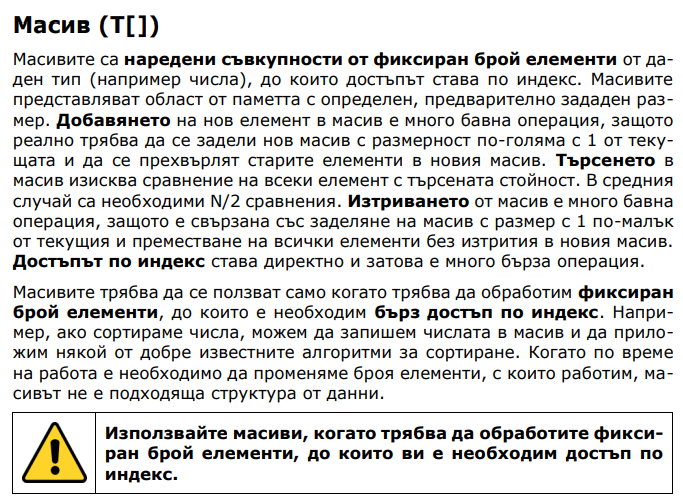
0(logn) се използва при Binary Search. Чрез този алгоритъм, ако имаме 1милион сортирани числа в масива, са ни нужни само 19 сравнения, за да намерим дадено число.

Търсенето по индекс в Array е с константа сложност, защото когато знае адреса на първата стойност и знае, че всяко число заема по 4 байта памет, може еднакво бързо да изчисли в кой индекс всяко едно число се намира



В JS Array is dynamic – като List in C#





**ArrayList**

който представлява динамично-разширяем масив. Той е реализиран по сходен начин със [статичната реализация на списък](https://introprogramming.info/intro-csharp-book/read-online/glava16-lineini-strukturi-ot-danni/#_%D0%A1%D1%82%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D0%BD_%D1%81%D0%BF%D0%B8%D1%81%D1%8A%D0%BA_(%D1%80%D0%B5%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F). **ArrayList** дава възможност да добавяме, премахваме и търсим елементи в него. един от основните проблеми при тази реализация е прео­размеряването на вътрешния масив при добавянето и премахването на елементи. В класа **ArrayList** проблемът е решен чрез предварително създаване на по-голям масив, който ни предоставя възможност да доба­вяме елементи, без да преоразмеряваме масива при всяко добавяне или премахване на елементи. **List<T>** е шаблонният вариант на **ArrayList**.

Класът **List** се представя в паметта като масив, от който една част съхранява елементите му, а останалите са свободни и се пазят като резервни. Благодарение на резервните празни елементи в масива опера­цията добавяне почти винаги успява да добави новия елемент без да разширява (преоразмерява) масива. Понякога, разбира се, се налага преоразмеряване, но понеже всяко преоразмеряване удвоява размера на масива, това се случва толкова рядко, че може да се пренебрегне на фона на броя добавяния. Можем да си представим един **List** като масив, който има някакъв капацитет и запълненост до определено ниво.

Благодарение на предварително заделеното пространство в масива, съхраняващ елементите на класа **List<Т>**, той е изключително ефективна структура от данни, когато е необходимо бързо добавяне на елементи, извличане на всички елементи и пряк достъп до даден елемент по индекс.

Може да се каже, че **List<Т>** съчетава добрите страни на списъците и масивите – бързо добавяне на края, променлив размер и директен достъп по индекс.

класът **List<T>** използва вътрешно масив за съхранение на елементите, който удвоява размера си, когато се препълни. Тази негова специфика води до следните особености:

-     Търсенето по индекс става много бързо – можем да достъпваме с еднаква скорост всеки един от елементите независимо от общия им брой.

-     Търсенето по стойност на елемент работи с толкова сравнения, колкото са елементите (в най-лошия случай), т.е. не е бързо.

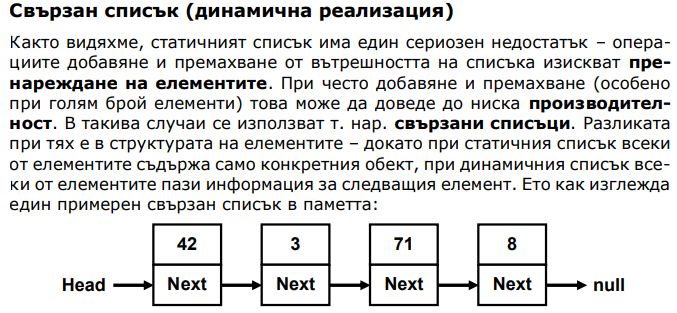
-     Добавянето и премахването на елементи е бавна операция – когато добавяме или премахваме елементи, особено, ако те не се намират в края на списъка, се налага да разместваме всички останали еле­менти, а това е много бавна операция.

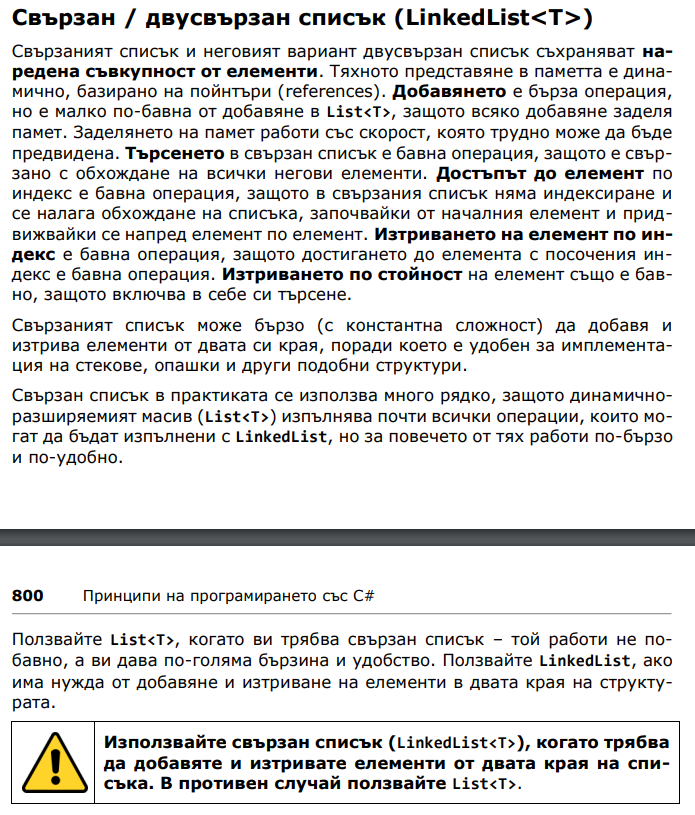
-     При добавяне понякога се налага и увеличаване на капацитета на масива, което само по себе си е бавна операция, но се случва много рядко и средната скорост на добавяне на елемент към **List** не зависи от броя елементи, т.е. работи много бързо.

**Използвайте List<T>, когато не очаквате често вмъкване и премахване на елементи, но очаквате да добавяте нови елементи в края или ползвате елементите по индекс.**

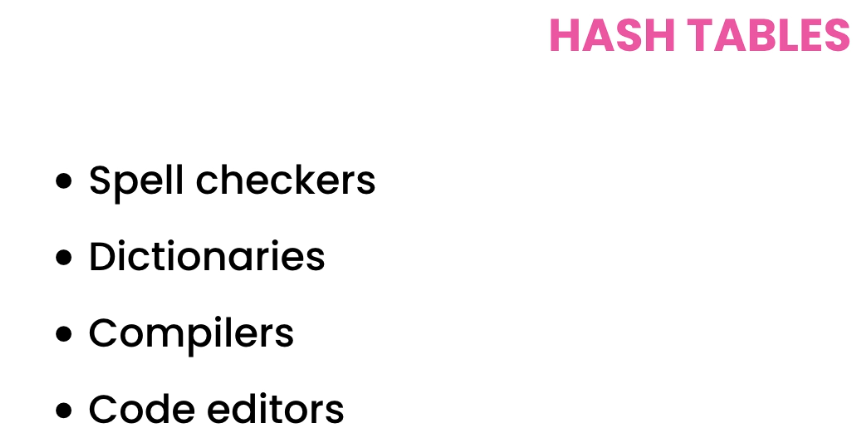
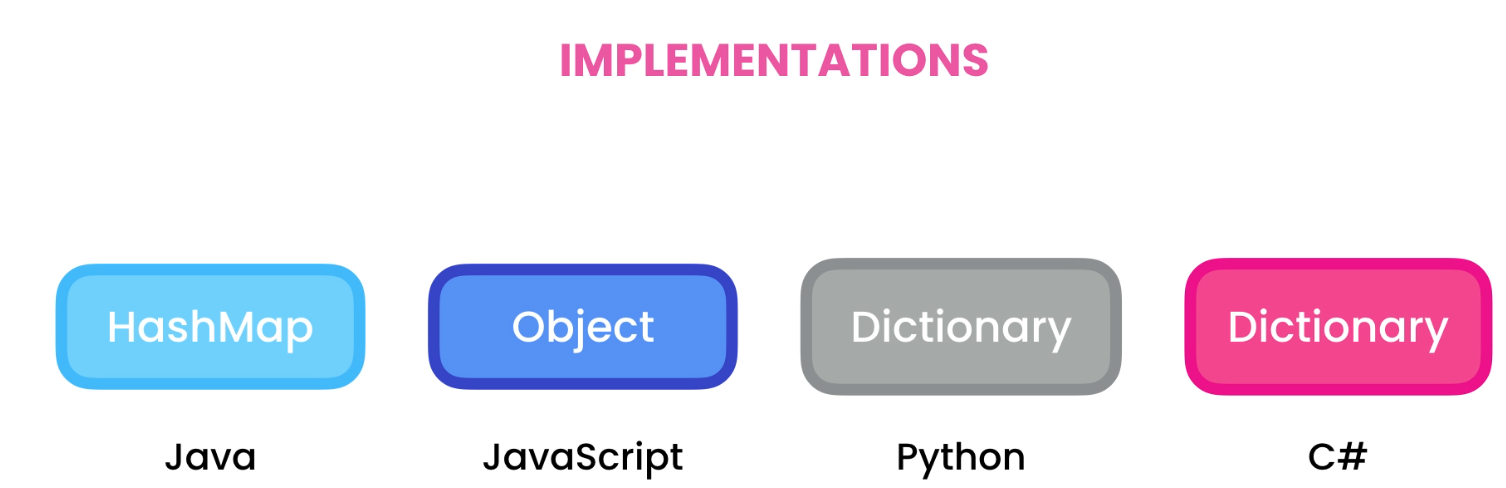
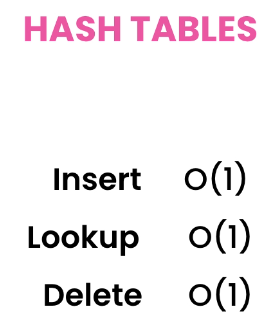
Списъците позволяват добавяне на нови елементи, докато масивите имат фиксиран размер. Въпреки това е възможна реализация на списък чрез масив, който автоматично увеличава размера си при нужда (по подобие на класа **StringBuilder**). Такъв списък се нарича **статичен.** Статичният списък има един сериозен недостатък – опера­циите добавяне и премахване от вътрешността на списъка изискват пренареждане на елементите. При често добавяне и премахване (особено при голям брой елементи) това може да доведе до ниска производителност.

**Linked List**





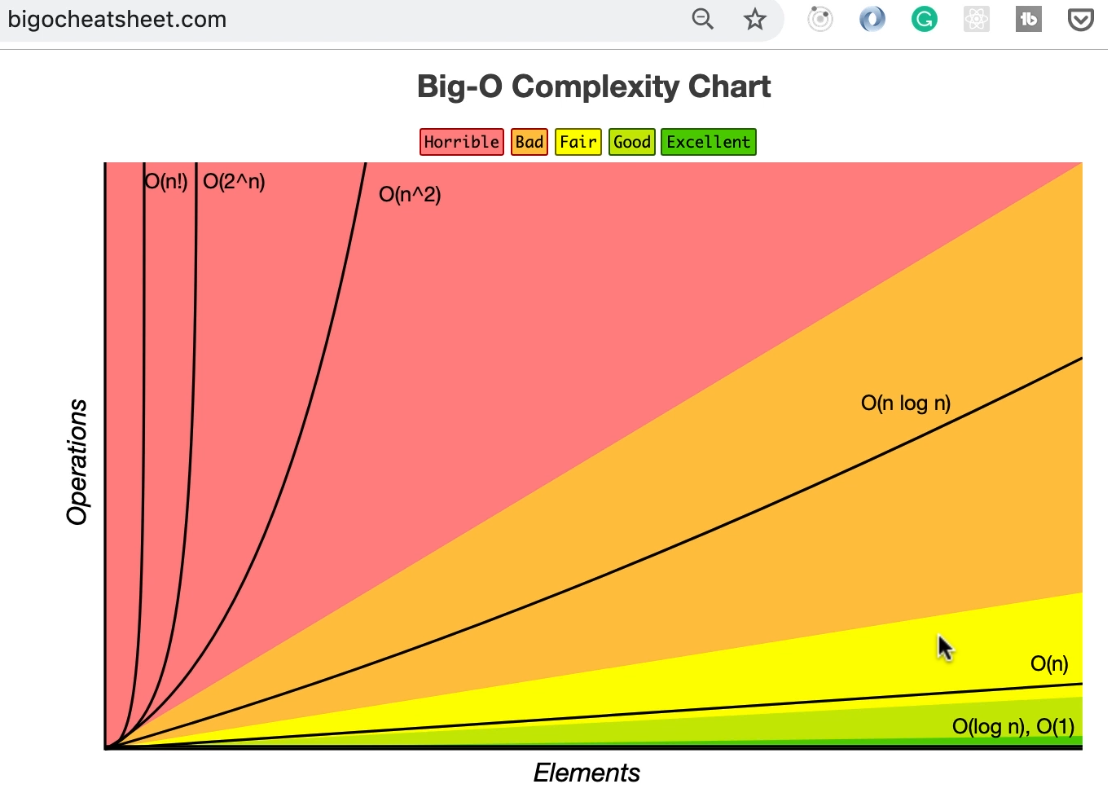
**Hash tables**

Our Hash Table takes employeeNumber, then passes it to a hash function and the hash function will tell where the employee object should be stored in memory. Our hash table will then store this employee object at that location. When we lookup and employee by their number, our Hash Table once again passes the employeeNumber to this hash function and it will figure out where this employee object is stored so it will return it to us. Internally Hash Table uses an array to store our objects.

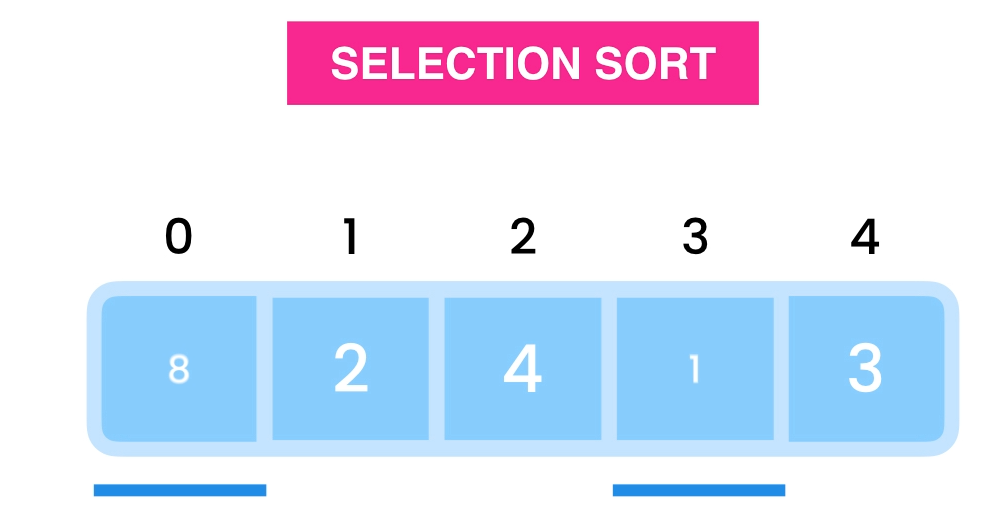
**BubbleSort**

Best case 0(n), worst case 0(n2) – квадратична сложност. Сортира елементите от началото по двойки, съответно при първата итерация най-големият елемент отива накрая, при втората – вторият най-голям елемент отива до първия накрая и т.н. – т.е. след всяка итерация най-големият елемент „изплува“ и затова се нарича Bubble Sort. Не е особено ефективен, особено когато обема на информация се увеличава:



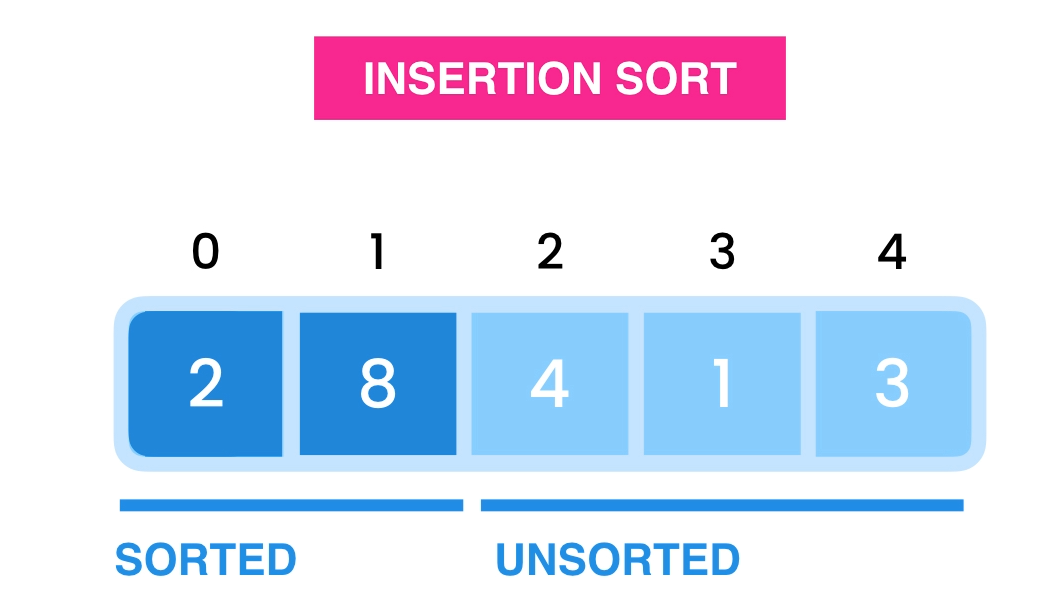
**Selection Sort**

Бавен – квадратична сложност. При него намираме най-малката стойност и я слагаме най-отпред, заменяйки я със стойността, която е там.



**Insertion Sort**

Бавен – квадратична сложност. При него взимаме втората стойност, сравняваме с първата и ако е по-малка, ги сменяме. След това взимаме третата стойност, сравняваме я с първите две и я слагаме на правилното място. Така всички стойности наляво да сортирани – имаме сортирана и несортирана част на масива



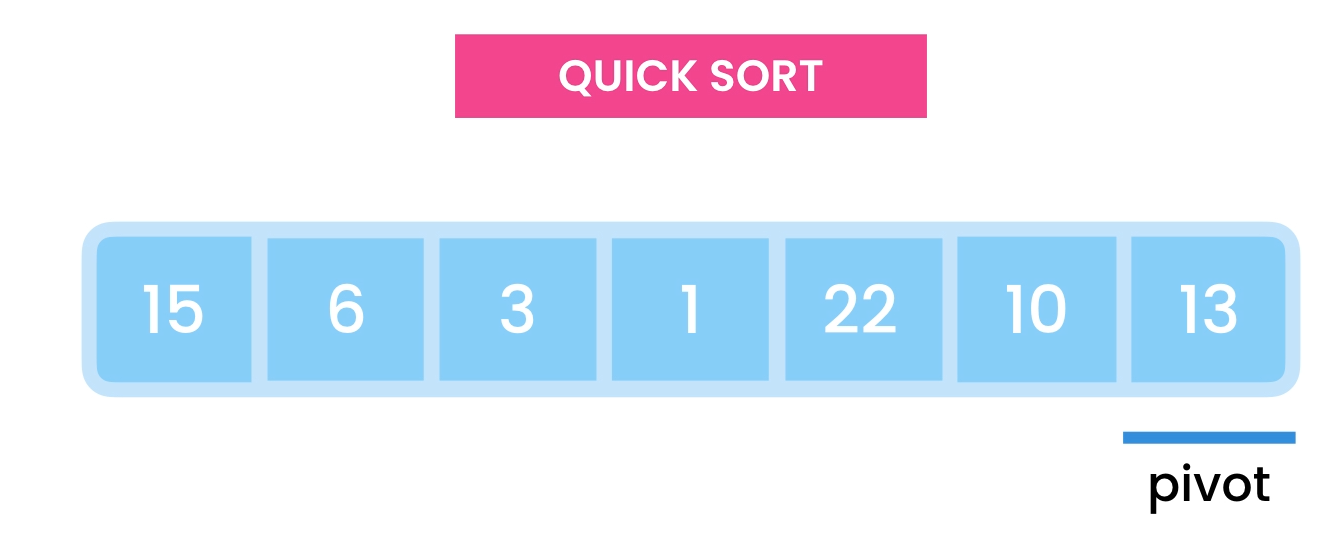
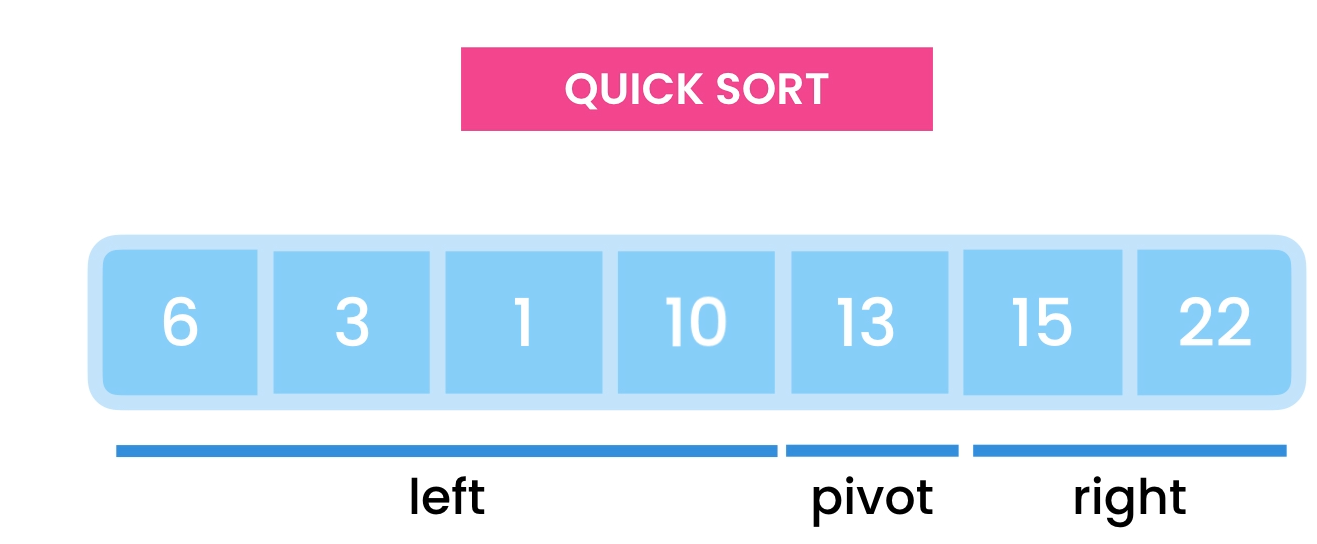
Тези три алгоритъма са неефективни, защото имат квадратична сложност. Следващите са по-ефективни, защото са O(n log n) (при 1 000 000 елемента са ни нужни 10 000 стъпки)

**Merge Sort**

Идеята при него е листа да се раздели рекурсивно на по-малки подмасиви, да се сортират и след това обратно да се обединят.

**Quick Sort**

Един от най-използваните алгоритми е алг. Който е вграден в повечето езици и фреймуъркс. Относително ефикасен е и за разлика от Merge Sort, изисква допълнително място. При него избираме елемент(пивот) и пренареждаме ел. така, че по-малките от него да са отляво, а по-големите – отдясно. Това се нарича partition. Така разделяме масива на две, разделени от пивота. Обикновено избираме пивота да е последният елемент, но има и изключения. Когато изберем пивот и го позиционираме на правилната позиция, той повече не се мести. Има лог. Сложност, защото при всяка итерация, намаляваме масива почти двойно

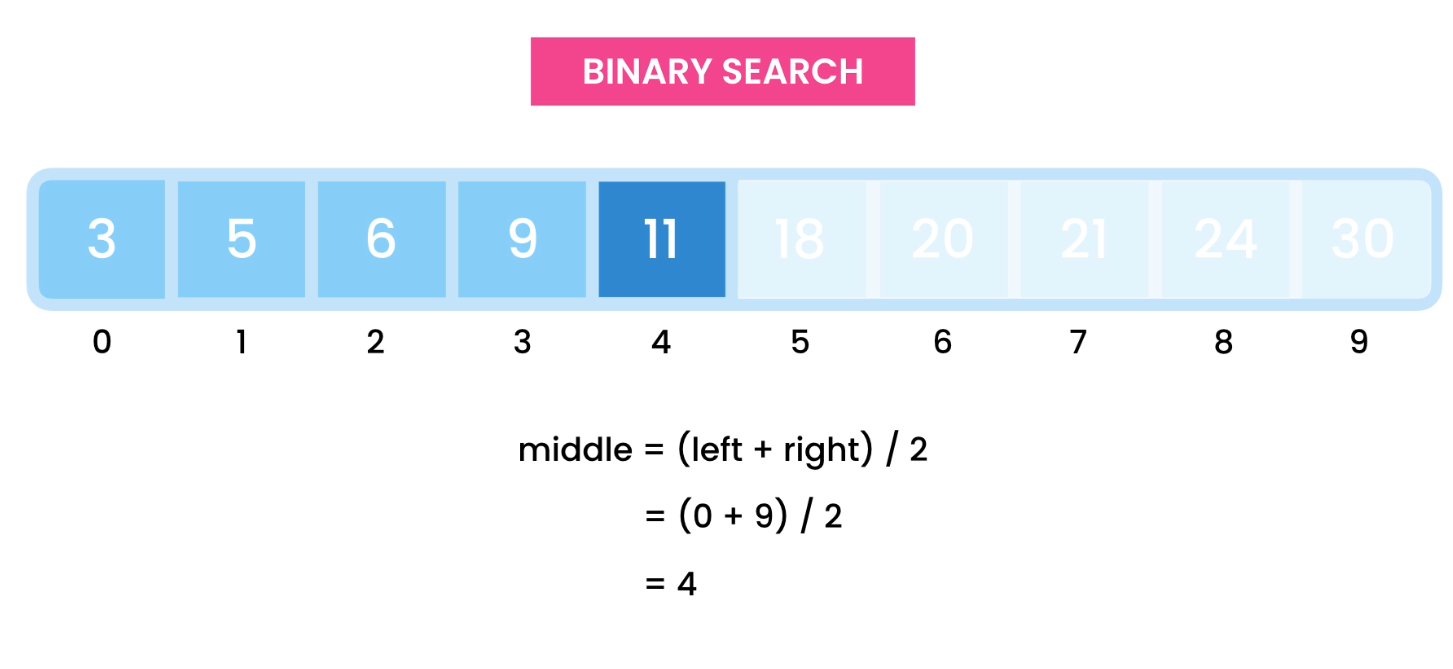
 

**Linear Search**

Най-простия алг. За тъсене. При него обхождаме всеки елемент и ако намерим търсения го връщаме. Сложност линейна. Относително бавен, в сравнение с други, но работи добре при малки масиви. С увеличаване на масива, този алг. Става бавен и е по-добре да използваме друг алг. напр. Binary Search

**Binary Search**

По-бърз е от Linear Search, но работи само на соритрани масиви. Ако нашият масив не е сортиран, трябва или първо да го сортираме и после да използваме Binary Search или да използваме Linear Search. Ако трябва да тъсим няколко стойности в масива, обикновено си струва да го сортираме и да използваме Бин. Сърч



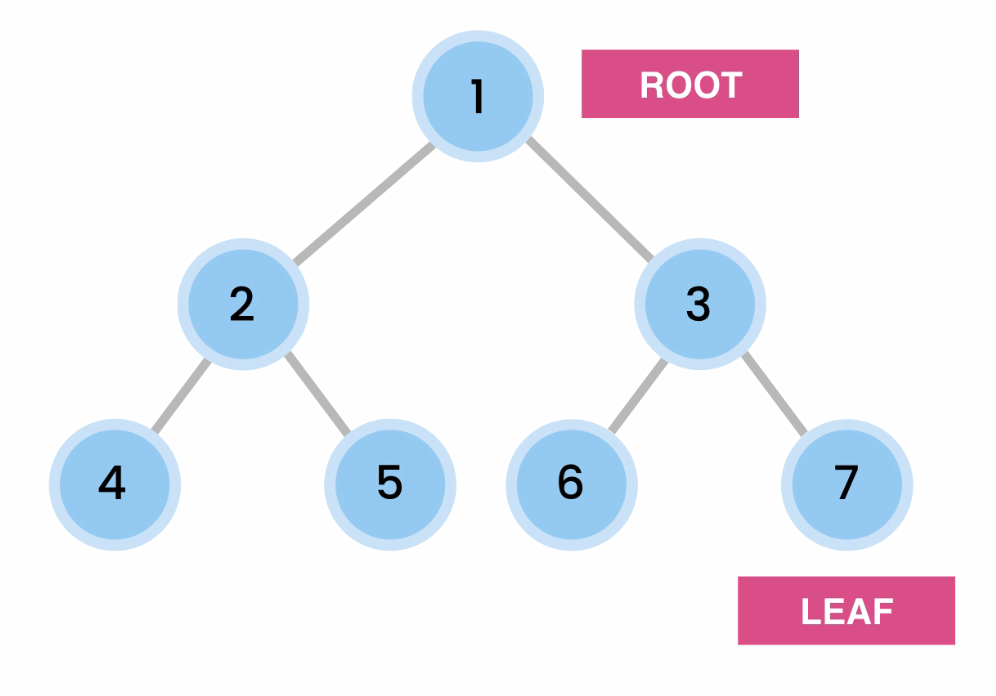
При него намираме средния индекс, след това ако търсеното число е по-малко от него, то тогава можем да елиминираме половината (дясната) част на масива. Затова и е толкова бърз – сложността му е 0(logn) (за 1 000 000 елемента са ни нужни само 20 стъпки)

**Trees**

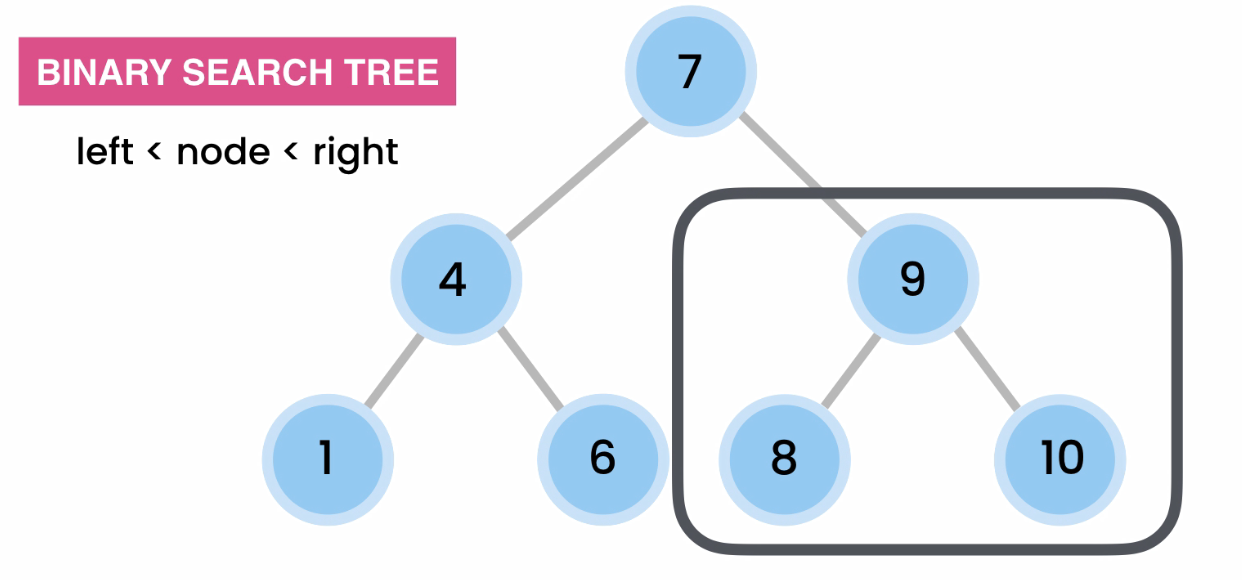
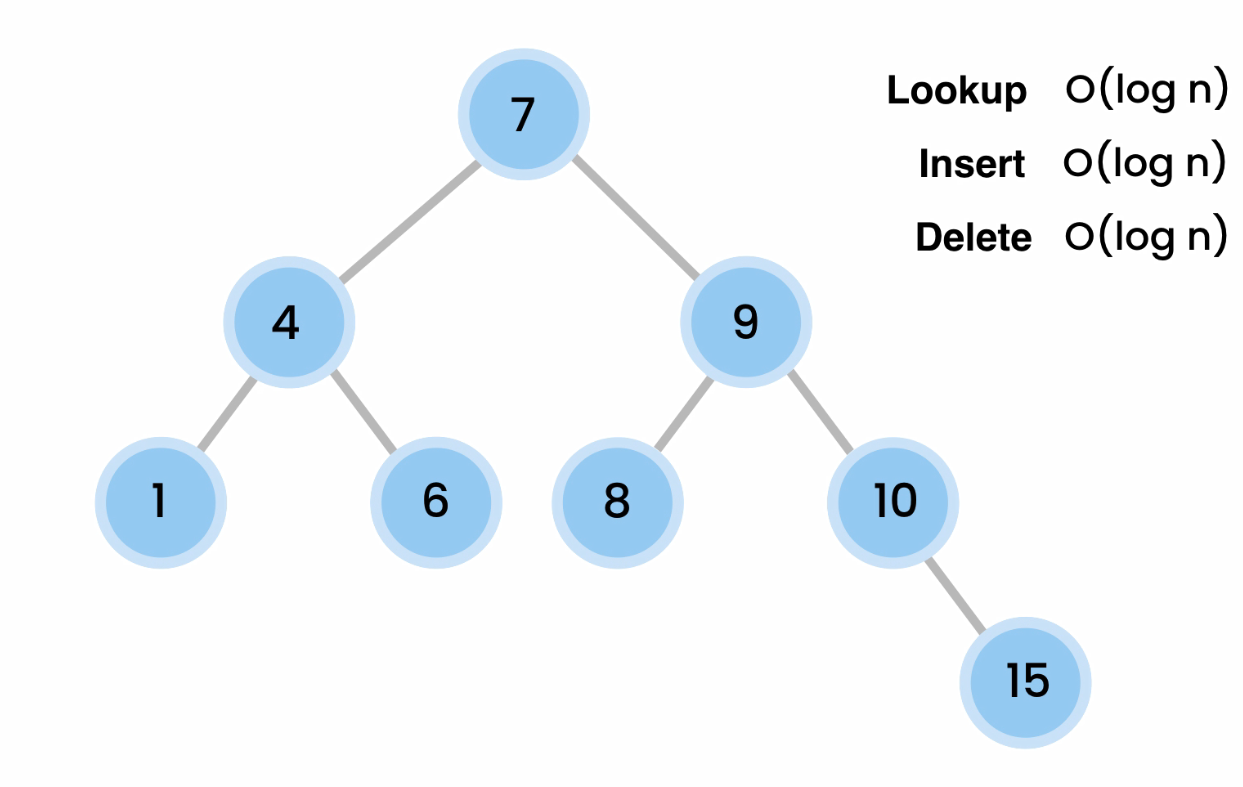
Нелинейни структури от данни. Използват се в бази данни, сайтове, graphical user interfaces, където трябва информацията трябва да се представи йерархично. Структура от данни, която съхранява елементите(nodes) в йерархия и линиите, който ги свързват са edges. Each node contains value or data (i.e. person objects, part of organization or family, files and folders on our disk).

В базаданните се използват за индексиране, за да можем бързо да намираме данни. V autocompletion се използва напр. в Хром, който пази всичките ни минали търсения в дърво. И когато напишем нещо, се опитва да го match-ne с някое от запазените в дървото query-ta. They are also used in compilers and copression algorithms(jpeg, mp3)

Има различни видове дървета. Фундаментално всички са еднакви, разликата в колко деца може да има всеки node и как стойностите в тези nodes мога да се нареждат. Binary tree – each node has a maximum of two children

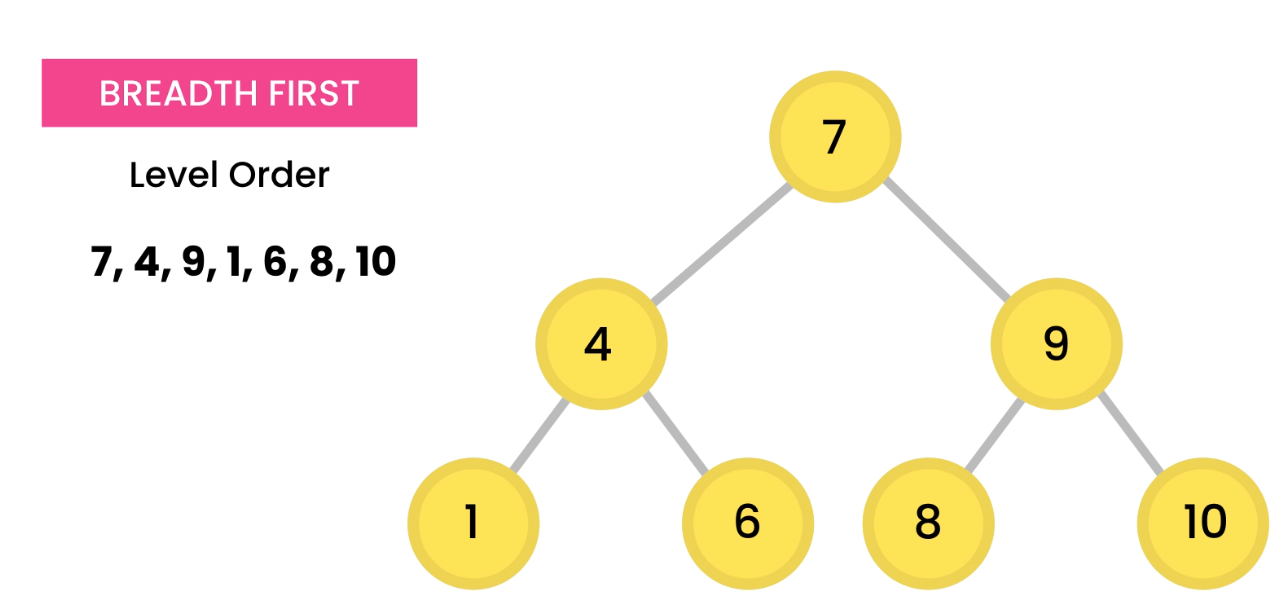


**Binary Search Tree** - Позволява ни бъро търсене на данни. Left < node < right. These nodes represent a sub-tree with 4 and 9 being the root of the subtree. Като си организираме данните по този начин, можем да бързо да търсим стойност. За разлика от array and linked list, не е нужно да обикаляме цялото дърво, за да намерим дадена стойност. Времето за намиране е логаритмично **О(log n)**, защото въпреки, че дървото има 7 nodes, ако искаме да намерим 1, ни трябват само три сравнения. Защото на всяка стъпка отхвърляме половината nodes. Добавяне на елемент също е лог., защото първо трябва да намерим node отнема логар., а добавянето на нов елемент – const. Същото важи и за изтриване на node. **Lookup O(log n) Insert O(log n)**

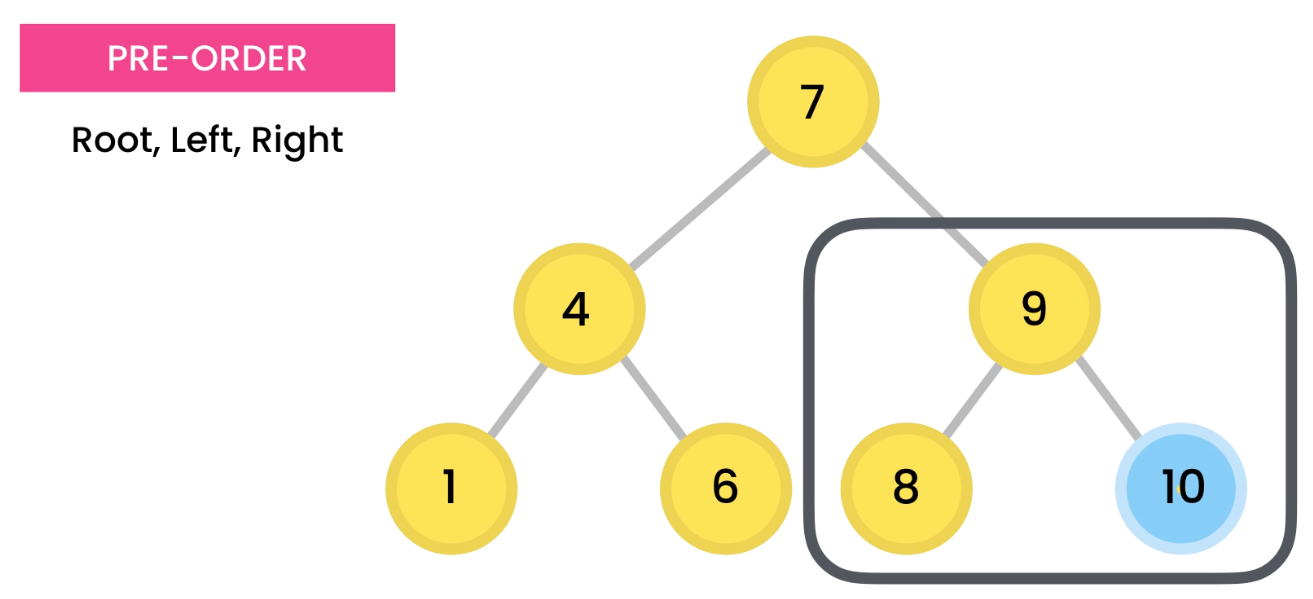
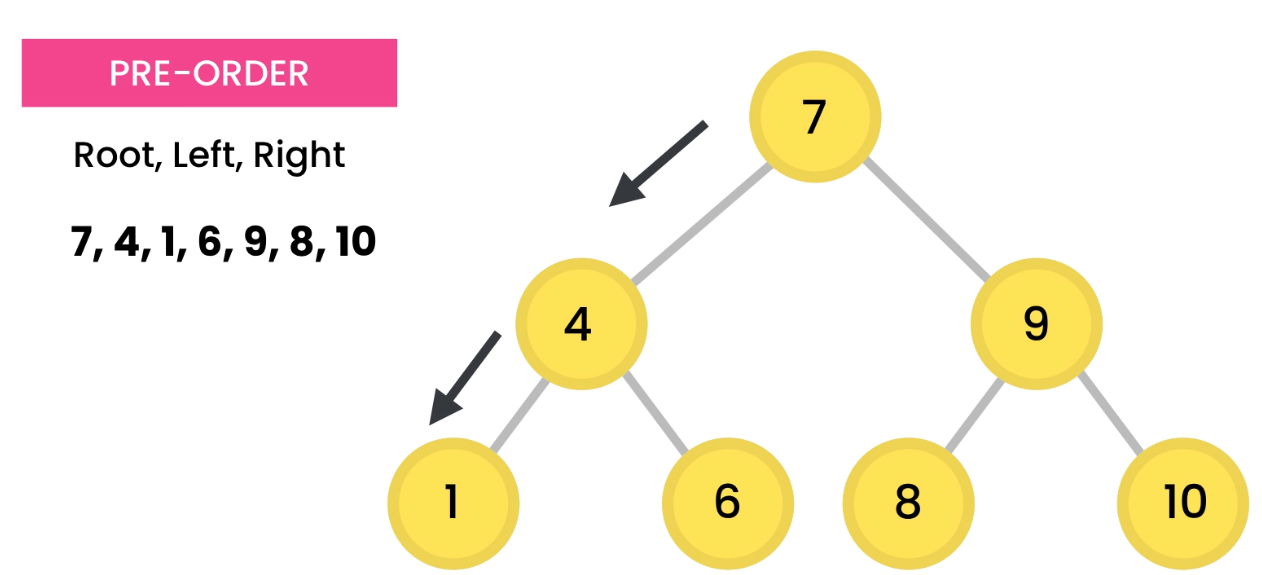
 

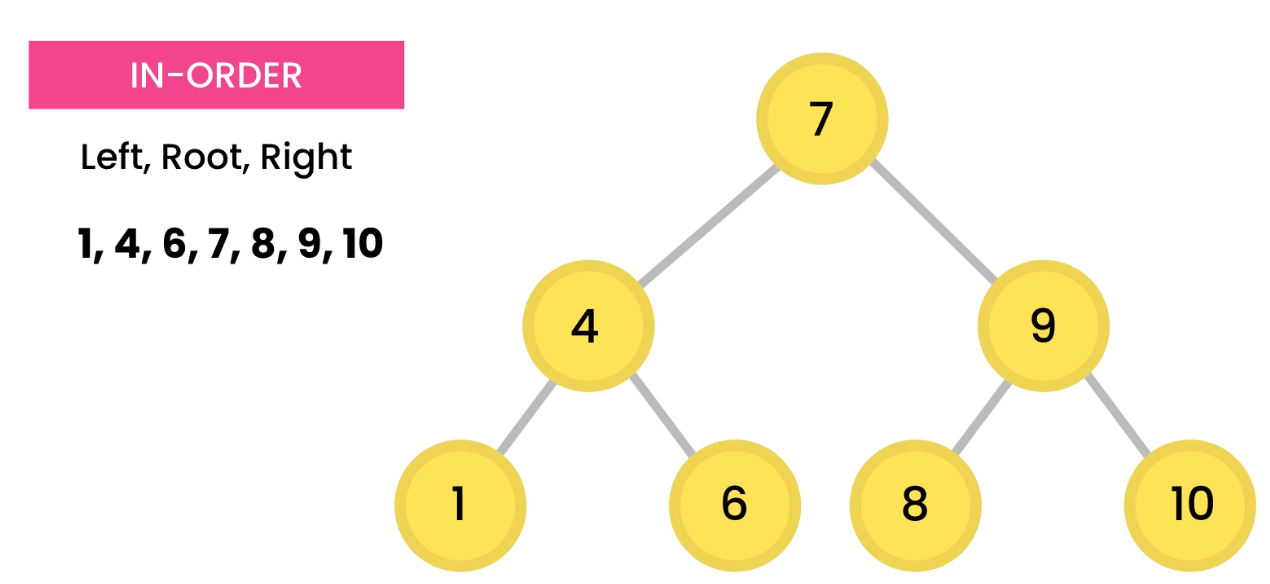
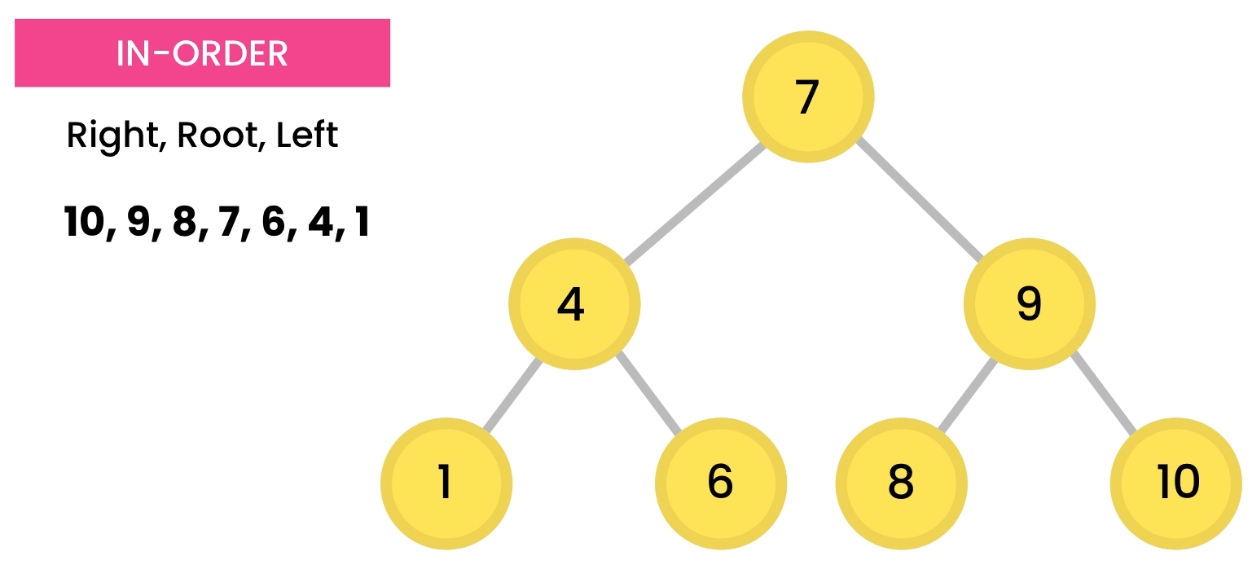
**Breadth First**

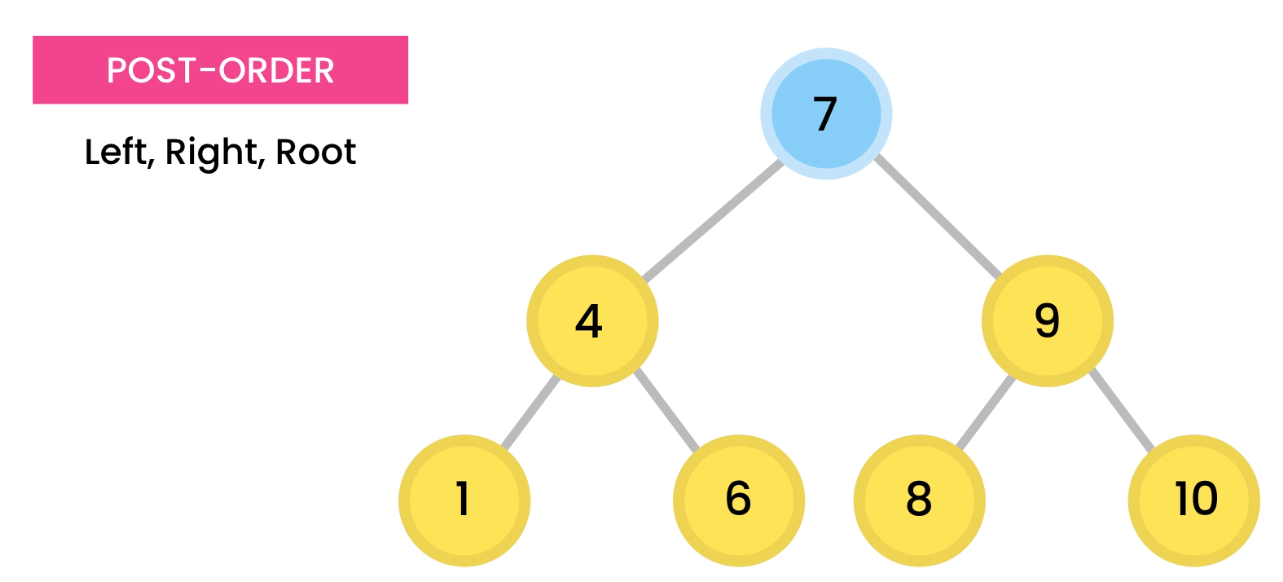
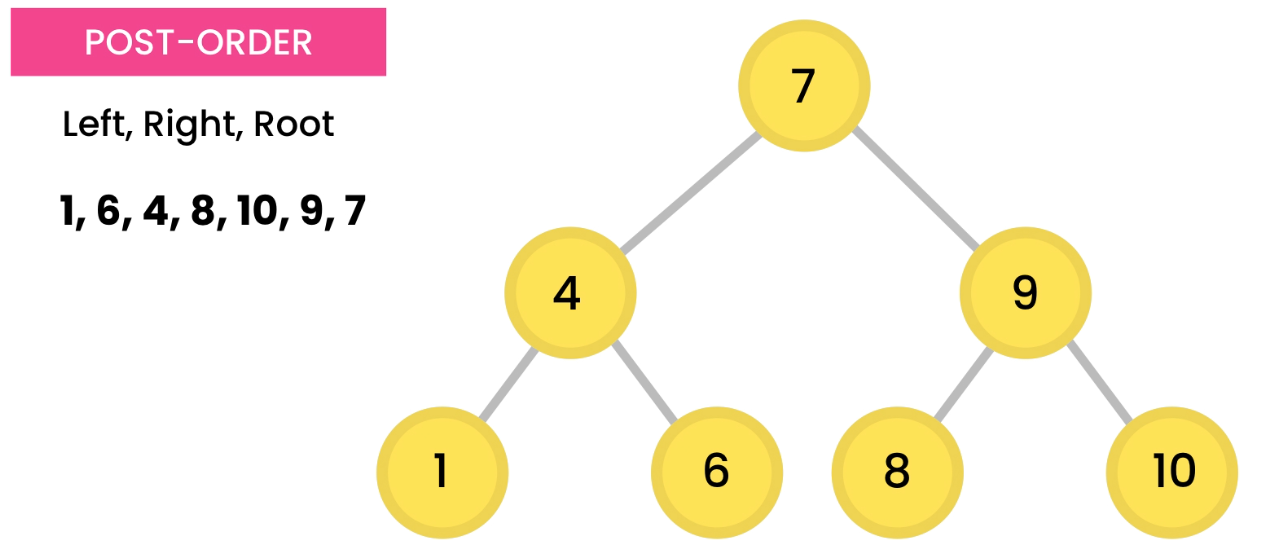
Also called Level Order traversal. We visit the nodes one level at a time



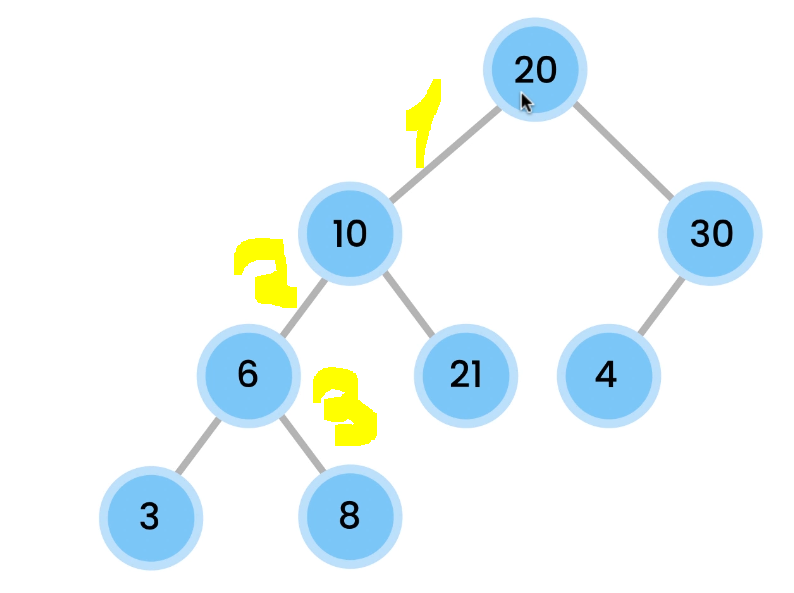
**Depth First**: We have three options – Pre-order (Root, Left, Right) , in-order (Left, Root, Right) , post-order (Left, Right, Root). In in-order numbers come out sorted in ascending. Съответно ако искаме да ги искараме в низходящ ред, обхождаме right, root, left

Depth can be calculated by how many edges do we have from the root to that node: The depth of 20 is zero, the depth of 8 is 3



Height is the opposite of depth. All the leaf’s nodes’ height is zero.The height of 6 is one, the hight of 30 also one. To calculate the height of 10, we must calculate the longest path from a leaf to it, so it’s height is 2

