# Introduction

## What Is Algorithm?

Let us consider the problem of preparing an omelet dish. To do that , we follow the steps given below:

1. Get the frying pan.
2. Get the oil.
   1. Do we have oil?
3. If yes, put it in the pan.
4. If no, do we want to buy oil?
5. If yes, then go out and buy.
6. If no, we can terminate.
7. Turn on the stove, etc...

What we are doing is are providing a step-by-step procedure for solving it.

Formal definition of an algorithm: **An algorithm is the step-by-step unambiguous instructions to solve a given problem.**

## What Is Analysis of Algorithms?

Multiple algorithms are available for solving the same problem (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort, etc.). Algorithm analysis helps us **determine which algorithm is most efficient in terms of time and memory consumed**.

## How to Compare Algorithms?

Do you think following measures are good to compare algorithms?

* ~~Execution times~~? Not a good measure as they are specific to a particular computer.
* ~~Number of statements executed~~? Not a good measure as it varies with the programming language and the style of the individual programmer.

Ideal solution? Let us assume that we express the running time of a given algorithm as a function of the input size n (f(n)) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc.

## What is Rate of Growth?

The rate at which the **running time increases as a function of input** is called *rate of growth*.

Below is the list of growth rates you will come across in the following chapters:

|  |  |  |
| --- | --- | --- |
| **Time Complexity** | **Name** | **Example** |
| 1 | Constant | Adding an element to the front of a linked list |
| logn | Logarithmic | Finding an element in a sorted array |
| n | Linear | Finding an element in a unsorted array |
| nlogn | Linear Logarithmic | Sorting n items by ‘Divide and Conquer’ |
| n2 | Quadratic | Shortest path between 2 nodes in a graph |
| n3 | Cubic | Matrix Multiplication |
| 2n | Exponential | The Towers of Hanoi problem |

## How Many Types of Analysis?

There are three types of analysis:

1. **Worst case**

* Defines the input for which the algorithm takes **the longest time to complete**.

1. **Best case**

* Defines the input for which the algorithm takes **the fastest time to complete**.

1. **Average case**

* Provides a prediction about the running time of the algorithm.
* Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.
* Assumes that the input is random.

Lower Bound <= Average Time <= Upper Bound

## Notation

### Big-O Notation

This notation gives the tight upper bound of the given algorithm and we represent it as f(n) = O(g(n)).

For example, if f(n) = n4 + 100n2 + 10n + 50 is the given algorithm, then g(n) is n4 => O(n4).

**O(1)**

Time complexity of a function (or set of statements) is considered as O(1) if it doesn’t contain loop, recursion and call to any other non-constant time function. For example:

// set of non-recursive and non-loop statements

A loop or recursion that runs a constant number of times is also considered as O(1). For example:

// Here c is a constant

for (int i = 1; i <= c; i++) {

    // some O(1) expressions

}

**O(n)**

Time complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount. For example:

// Here c is a positive integer constant

for (int i = 1; i <= n; i += c) {

    // some O(1) expressions

}

**O(nc)**

Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example – the following sample loops have O(n2) time complexity:

for (int i = 1; i <=n; i += c) {

    for (int j = 1; j <=n; j += c) {

        // some O(1) expressions

    }

}

**O(logn)**

Time complexity of a loop is considered as O(logn) if the loop variables is divided / multiplied by a constant amount. For example:

for (int i = 1; i <=n; i \*= c) {

    // some O(1) expressions

}

**O(loglogn)**

Time complexity of a loop is considered as O(loglogn) if the loop variables is reduced / increased exponentially by a constant amount. For example:

// Here c is a constant greater than 1

for (int i = 2; i <=n; i = pow(i, c)) {

    // some O(1) expressions

}

### Omega-Ω Notation

This notation gives the tight lower bound of the given algorithm and we represent it as f(n) = Ω(g(n)).

For example, if f(n) = 100n2 + 10n + 50 is the given algorithm, then g(n) is n2 => Ω(n2).

### Theta-Θ Notation

This notation decides whether the upper and lower bounds of a given algorithm are the same.

IMPORTANT NOTE:

In the remaining chapters, we generally **focus on the upper bound (O) because knowing the lower bound (Ω) of an algorithm is of no practical importance**, and we use the Θ notation if the upper bound (O) and lower bound (Ω) are the same.

# Recursion and Backtracking

# Data Structures

## Linked List

Great video about how to implement linked list in C:

<https://www.youtube.com/playlist?list=PL9IEJIKnBJjFiudyP6wSXmykrn67Ykqib>

### Definition

### Basic Operations & Time Complexities

### Applications

### Linked List vs Array

Both arrays and linked list can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.





**Drawbacks of arrays:**

1. The size of the arrays is fixed: We must know the upper limit on the number of elements in advance. Also, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.

2. Inserting a new element to an array is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

For example, suppose we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040, ...].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

3. Deletion is also expensive with arrays until unless some special techniques are used.

For example, to delete 1010 in id[], everything after 1010 has to be moved.

**Linked list provides following two advantages over arrays:**

1. Dynamic size

2. Ease of insertion/deletion

**But linked lists have following drawbacks:**

1. Random access is not allowed. We have to access elements sequentially starting from the first node. So, we cannot do binary search with linked lists.

2. Extra memory space for a pointer is required with each element of the list.

3. Arrays have better cache locality that can make a pretty big difference in performance.

### Why double pointers are used in linked list?

It is clear that both methods (single pointer and double pointer) lead to the **same result**. The only difference is **what will be changed afterward**.

Double pointers are used as **arguments** of function when the function modifies and updates the linked list without needing to return the value (address or data) of the list again.

When using single pointers as arguments of function that modifiers and updates the linked list, we must return the value (address or data) of the list. Or else, the effect won’t be noticed.

Briefly, remember the simple C rule: If you want to **modify local variable of one function inside another function**, pass pointer to that variable. It is called "call by pointers". In this case, the pointer is C’s way of implementing "call by reference" when there is no reference variable.

For example, you want to add a new node before the head (first node) of the list, and hence, the pointer pointing to the first node will be then changed. When you exit this function, you want this change to reflect in the calling function and the following code in the main() function (suppose you call this function in the main()). In this case, you have to use a double pointer. One of them is to indicate that you are passing an address and another is to make the changes available to the calling function (to achieve call by reference).

## Stack

### Definition

Stack is a linear data structure that allows adding and removing elements in a specific order. In particular, every time an element is added, it goes on the top of the stack. The only element that can be removed is the one at the top of the stack. In other words, **the first item added to a stack will be the last item removed from it**. As a result, a stack is said to have "last in first out" behavior (or *LIFO*).

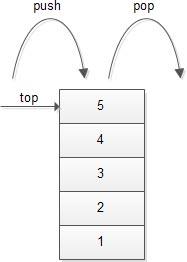
A typical example of using stack is function calling. A function calls another function, which in turn calls a third function; it's important that the third function return back to the second function rather than the first one.

*You might not know!*

The "call stack" is the term used for the list of functions either executing or waiting for other functions to return.

### Basic Operations & Time Complexities

* **push**: Adds an item in a stack. If the stack is full, it is said to be an *Overflow* condition – Time complexity: O(1).
* **pop**: Removes an item from a stack. If the stack is empty, it is said to be an *Underflow* condition – Time complexity: O(1).
* **peek** or **top**: Returns a reference to the top most element of a stack – Time complexity: O(1).
* **size**: Returns the size of a stack – Time complexity: O(1).
* **isEmpty**: Returns true if stack is empty, else false – Time complexity: O(1).
* **isFull**: Returns true if stack is full, else false – Time complexity: O(1).



### Applications

* [Balancing of symbols](https://www.geeksforgeeks.org/check-for-balanced-parentheses-in-an-expression/)
* [Infix to Postfix](http://quiz.geeksforgeeks.org/stack-set-2-infix-to-postfix/) /Prefix conversion
* Redo-undo features at many places like editors, photoshop
* Forward and backward feature in web browsers
* Used in many algorithms like [Tower of Hanoi,](https://www.geeksforgeeks.org/recursive-functions/)[tree traversals](https://www.geeksforgeeks.org/618/), [stock span problem](https://www.geeksforgeeks.org/the-stock-span-problem/), [histogram problem](https://www.geeksforgeeks.org/largest-rectangular-area-in-a-histogram-set-1/).
* Other applications can be Backtracking, [Knight tour problem](https://www.geeksforgeeks.org/backtracking-set-1-the-knights-tour-problem/), [rat in a maze](https://www.geeksforgeeks.org/backttracking-set-2-rat-in-a-maze/), [N queen problem](https://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/) and [sudoku solver](https://www.geeksforgeeks.org/backtracking-set-7-suduku/)
* In Graph Algorithms like [Topological Sorting](https://www.geeksforgeeks.org/topological-sorting/) and [Strongly Connected Components](https://www.geeksforgeeks.org/strongly-connected-components/)

### Implementation

There are two ways to implement a stack:

* Using array
* Using linked list

<https://www.geeksforgeeks.org/stack-data-structure-introduction-program/>

### Stack in C++ STL

<https://www.geeksforgeeks.org/stack-in-cpp-stl/>

## Queue

### Definition

Queue is a linear data structure that allows adding and removing elements in a specific order To understand a queue, think of a cafeteria line: new people are added to the line at the back; the first person in line is served first, and the last person is served last. So, **in a queue the first item added to it will be the first item removed from it**. As a result, a queue is said to have "first in first out" behavior (or *FIFO*). That is opposite to a [stack](#_2et92p0).

*Note:*

Although the concept is simple, programming a queue is not as simple as programming a *stack*.

Let's go back to the example of the cafeteria line. Let's say one person leaves the line. Then what? Everyone in line must step forward, right? Now, imagine if only one person could move at a time. So, the second person steps forward to fill the space left by the first person, and then the third person steps forwards to fill the space left by the second person, and so on. Now imagine that no one can leave or be added to the line until everyone has stepped forward. You can see the line will move very slowly.

It is not difficult to program a queue that works, but it is **quite touch to make a queue that works fast**!

### Basic Operations & Time Complexities

* **enqueue**: Adds an item to a queue. If the queue is full, it is said to be an *Overflow* condition – Time complexity: O(1).
* **dequeue**: Removes an item from a queue. If the queue is empty, it is said to be an *Underflow* condition – Time complexity: O(1).
* **size**: Get the numer of elements in a queue – Time complexity: O(1).
* **isEmpty**: Returns true if queue is empty, else false – Time complexity: O(1).
* **isFull**: Returns true if queue is full, else false – Time complexity: O(1).
* **front**: Get the front item from a queue – Time complexity: O(1).
* **rear** or **back**: Get the last item from a queue – Time complexity: O(1).



### Applications

Queue is used when things don’t have to be processed immediately, but have to be processed in FIFO order like [Breadth First Search](http://en.wikipedia.org/wiki/Breadth-first_search). This property makes queue useful in following scenarios.

* When a resource is shared among multiple consumers. For examples, CPU scheduling or disk scheduling.
* When data is transferred asynchronously (data isn’t necessarily received at same rate as sent) between two processes. For examples, IO buffers, pipes, file IO, sockets, etc.
* Simulation of real-world queues such as lines at a ticket counter or any other first-come first-served scenario.

### Implementation

There are two ways to implement a stack:

1. **Using array**: The first method is to make an array and shift all the elements to accommodate enqueues and dequeues. This is slow, because with many elements, the shifting takes time.

The second method is, instead of shifting the elements, shifting the enqueue and dequeue points. Imagine that cafeteria line again. If the front of the line continually moves backwards as each person leaves the line, then people don't need to step forward or backwards, which saves time.

This method is much more complicated than the first one. Instead of keeping track of just the enqueue point (the "end"), we also need to keep track of the dequeue point (the "front"). This all gets even more complicated when we realize that after a bunch of enqueues and dequeues, the line will need to wrap around the end of the array. Think of the cafeteria line. As people enter and leave the line, the line moves farther and farther backwards, and eventually it will circle the entire cafeteria and end up at its original location.

<https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/>

1. **Using linked list**:

<https://www.geeksforgeeks.org/queue-data-structure/>

### Queue in C++ STL

<https://www.geeksforgeeks.org/queuepush-and-queuepop-in-cpp-stl/>

### Different Types of Queues

#### Priority queue

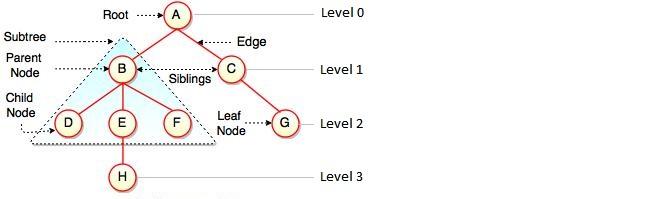
#### Circular queue

## Tree

### General Tree

#### Definition

Tree is a non-linear data structure which represents the nodes connected by edges. It’s used to store the information in the form of **hierarchy style**.



Terms:

* *Path* – Represents the sequence of nodes along the edges of a tree.
* *Visiting* – Represents checking the value of a node when control is on the node.
* *Traversing* – Represents passing through nodes in a specific order.
* *Keys* − Represents a value of a node based on which a search operation is to be carried out for a node.
* *Height of Tree* – Represents the height of its root node.
* *Height of Node* – Represents the number of edges on the longest path between that node and a leaf.
* *Depth of Node* – Represents the number of edges from the tree's root node to the node.
* *Degree of Node* – Represents a number of children of a node.

#### Traversal

In order to process trees, we need a mechanism for traversing them. Each node is processed only once but it may be visited more than once. As we have already seen **in linear data structures (like linked lists, stacks, queues, etc.), the elements are visited in sequential order. But, in tree structures there are many different ways**.

Starting at the root of a BT, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps are: performing an action on the current node (denoted with "D"), traversing to the left child node (denoted with "L"), and traversing to the right child node (denoted with "R"). This process can be easily described through recursion.

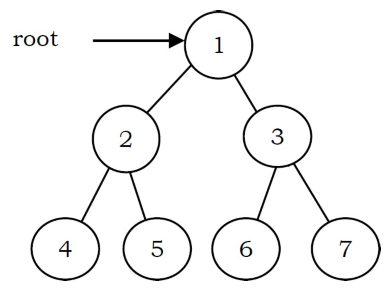
*So, we can classify the traversals into several styles:*

* *Pre-order (DLR) traversal*
* *In-order (LDR) traversal*
* *Post-order (LRD) traversal*

*There is another traversal method which does not depend on the above orders. It is:*

* *Level-order traversal: This method is inspired from Breadth First Traversal (BFS of Graph algorithms).*

Let us use the diagram below for the remaining discussion.



**Pre-Order Traversal**

The nodes of tree would be visited in the order: 1 2 4 5 3 6 7

In pre-order traversal, each node is processed before (pre-) either of its subtrees. It is defined as follows:

1. Visit the root.
2. Traverse the left subtree in pre-order.
3. Traverse the right subtree in pre-order.

*Time Complexity: O(n).*

*Space Complexity: O(n).*

**In-Order Traversal**

The nodes of tree would be visited in the order: 4 2 5 1 6 3 7

In in-order traversal, the root is visited between the subtrees. It is defined as follows:

1. Traverse the left subtree in in-order.
2. Visit the root.
3. Traverse the right subtree in in-order.

*Time Complexity: O(n).*

*Space Complexity: O(n).*

**Post-Order Traversal**

The nodes of the tree would be visited in the order: 4 5 2 6 7 3 1

In post-order traversal, the root is visited after both subtrees. It is defined as follows:

1. Traverse the left subtree in post-order.
2. Traverse the right subtree in post-order.
3. Visit the root.

*Time Complexity: O(n).*

*Space Complexity: O(n).*

**Level-Order Traversal**

The nodes of the tree are visited in the order: 1 2 3 4 5 6 7

Level-order traversal is defined as follows:

1. Visit the root.
2. While traversing level (, keep all the elements at level ( + 1 in queue.
3. Go to the next level and visit all the nodes at that level.
4. Repeat this until all levels are completed.

*Time Complexity: O(n).*

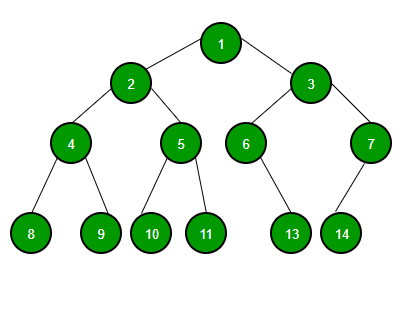
*Space Complexity: O(n). Since, in the worst case, all the nodes on the entire last level could be in the queue simultaneously.*

### Binary Tree

#### Definition

BT is a special tree used for data storage purposes. It has a special condition that **each node can have a maximum of two children**.

A BT has the benefits of both an ordered array and a linked list – **Search in BT is as quick as in a sorted array, and insertion or deletion in BT are as fast as in linked list**.



#### Basic Operations & Time Complexities

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Steps** | **Average** | **Worst** |
| **Search** |  | O(n) | O(n) |
| **Insert** |  | O(n) | O(n) |
| **Delete** |  | O(n) | O(n) |

Where: n is number of nodes

#### Applications

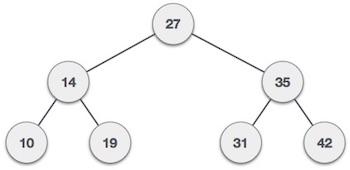
Following is the some of the applications where BT play an important role:

* Expression trees used in compilers.
* Huffman coding trees used in data compression algorithms.
* [Binary Search Tree](#_1fob9te) (BST), which supports search, insertion and deletion on a collection of items in O(logn) (average).
* [Priority Queue](#_gjdgxs) (PQ), which supports search and deletion of minimum (or maximum) on a collection of items in logarithmic time (worst case).

### Binary Search Tree

#### Definition

BST exhibits a special behavior of a binary tree. **A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value**.



**Note:**

Equal node values are not allowed in BST.

#### Basic Operations & Time Complexities

Complexity for all BST operations depends on BST height. It's like:

|  |  |
| --- | --- |
| Worst case | Best case |
|  |  |

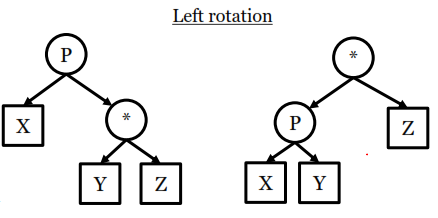
In particular:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Steps** | **Average** | **Worst** |
| **Search** | - Start at root  - Take left and right children as necessary | O(logh) | O(h) |
| **Insert** | - Similar to search, but look for null pointer | O(logh) | O(h) |
| **Delete** | - Search (if necessary)  - Different cases based on # children:  + 0: just delete  + 1: similar to doubly linked list  + 2: swap with max(left) and delete (0 or 1 children)  - Update size and root | O(logh) | O(h) |

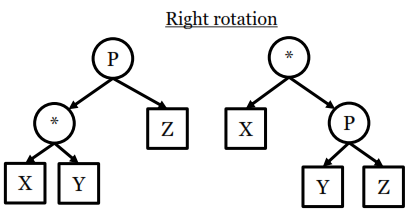
Where: h is height of tree

##### Tree Rotations

* Based around a pivot node
* Two directions: left and right
* Left rotation:
* Parent becomes left child
* Pivot becomes parent
* Old left child becomes parent right child
* Pivot is new "root" (fix parent)



* Right rotation:
  + Parent becomes right child
  + Pivot becomes parent
  + Old right child becomes parent left child
  + Fix pivot's new parent



##### Insertion

##### Deletion

#### Applications

### Balanced Binary Search Tree

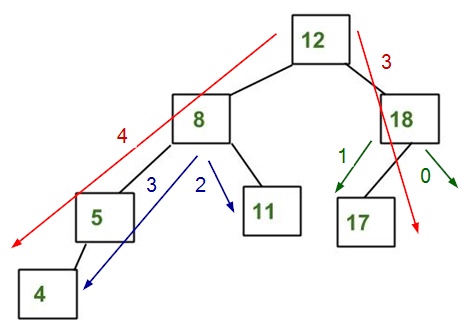
Balanced BST is a BST with self-balancing capability. It is divided into following subtypes:

#### AVL Tree

##### Definition

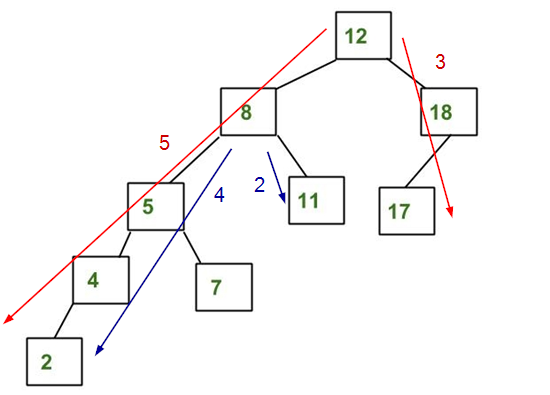
AVL tree (named after inventors Adelson-Velsky and Landis) is a self-balancing BST where the **difference between heights of left and right subtrees cannot be more than one for all nodes**. The balance is maintained using *tree rotations* as nodes inserted and deleted.

Example: AVL Tree



It is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.

Example: NOT AVL Tree



It is not AVL because differences between heights of left and right subtrees for 8 is 4 (greater than 1) and for 12 is 5 (greater than 1).

##### Basic Operations & Time Complexities

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Steps** | **Average** | **Worst** |
| **Search** |  | O(logh) | O(logh) |
| **Insert** |  | O(logh) | O(logh) |
| **Delete** |  | O(logh) | O(logh) |

Where: h is height of tree

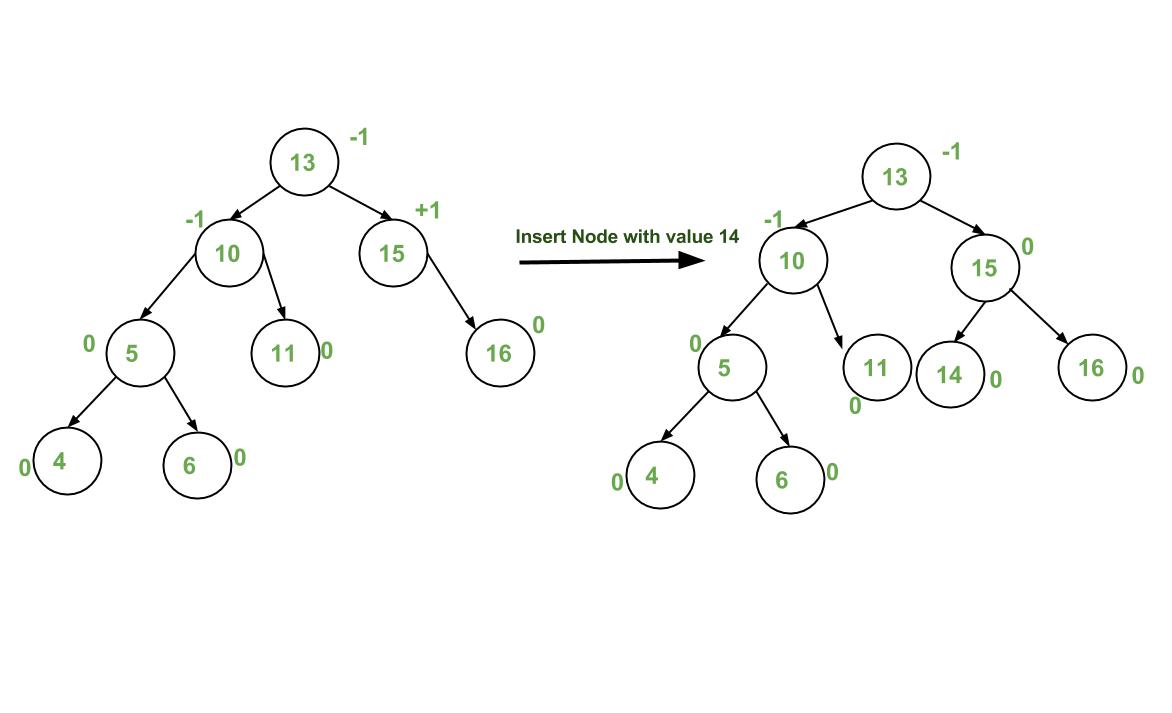
*More details:*

Most of the BST operations take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed binary tree. If we **make sure that height of the tree remains O(logn)** after every insertion and deletion, then we can guarantee an upper bound of O(logn) for all these operations. The height of an AVL tree is always O(logn) where n is the number of nodes in the tree.

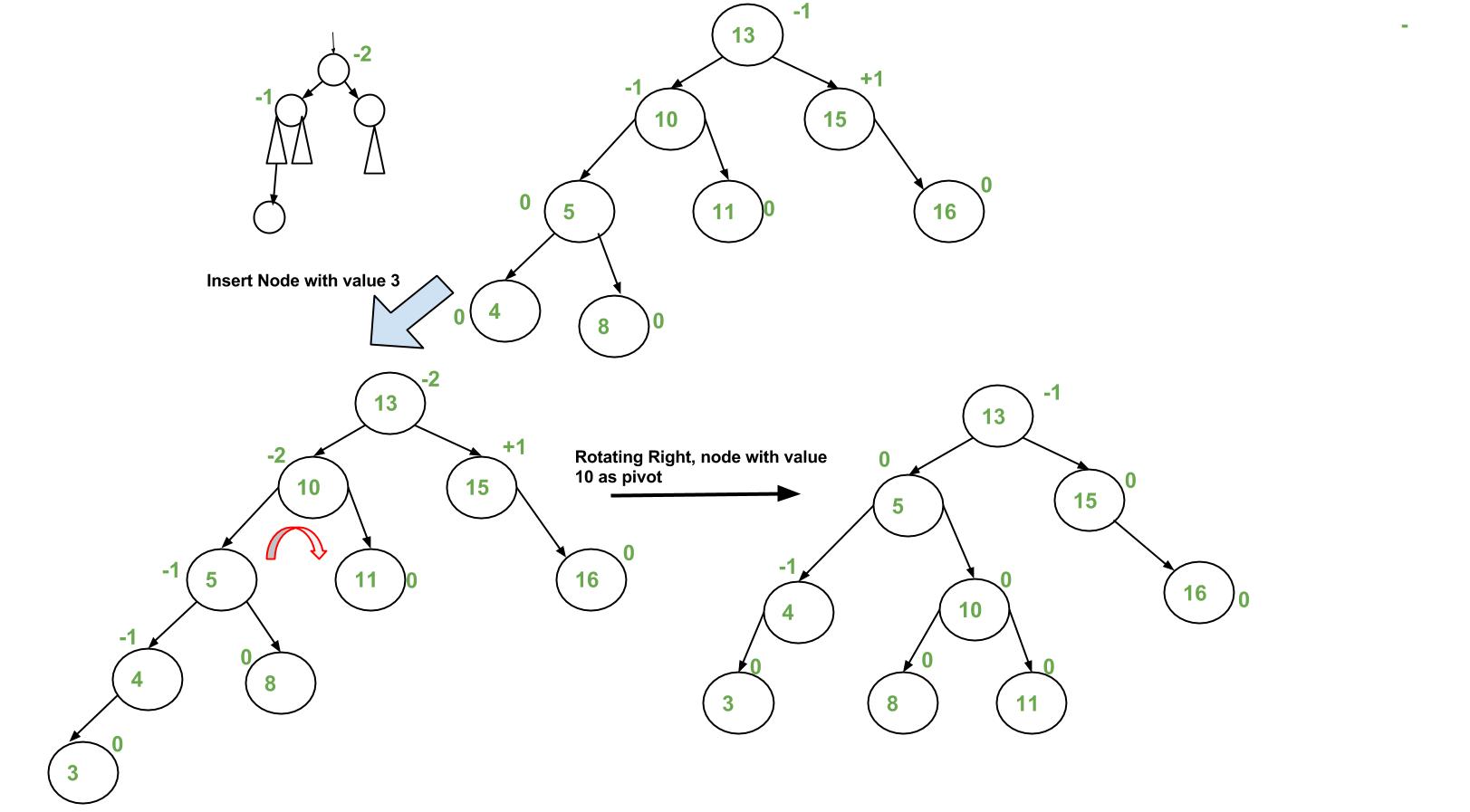
###### Insertion

Every time a new node is inserted, the tree must be re-balanced (if needed). One of following five cases can happen:

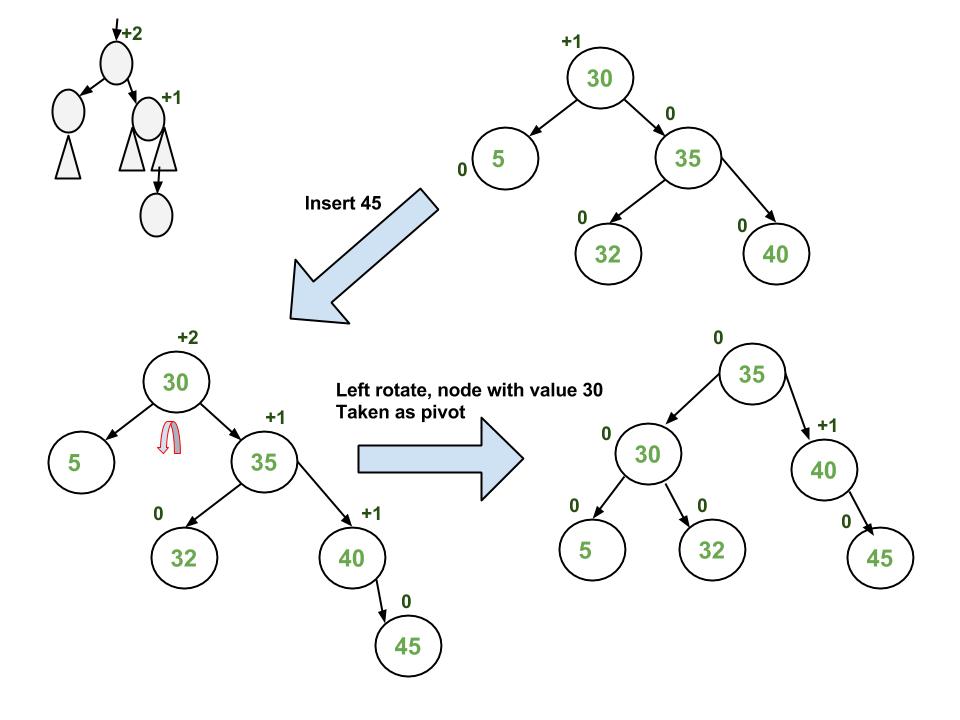
Case 1: No rotation needed



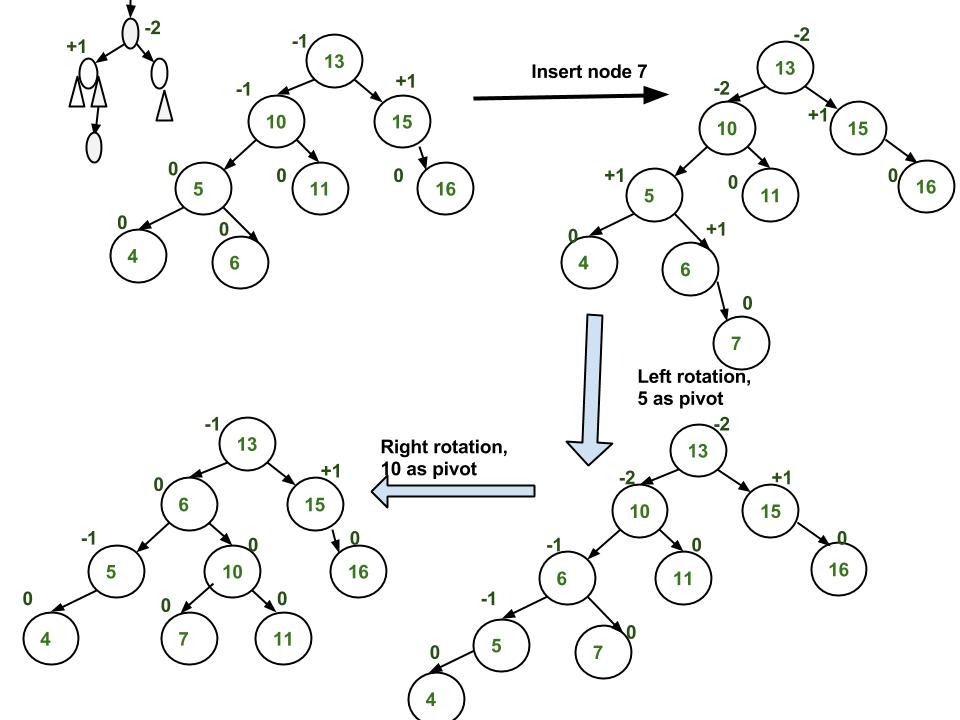
 Case 2: Right rotation



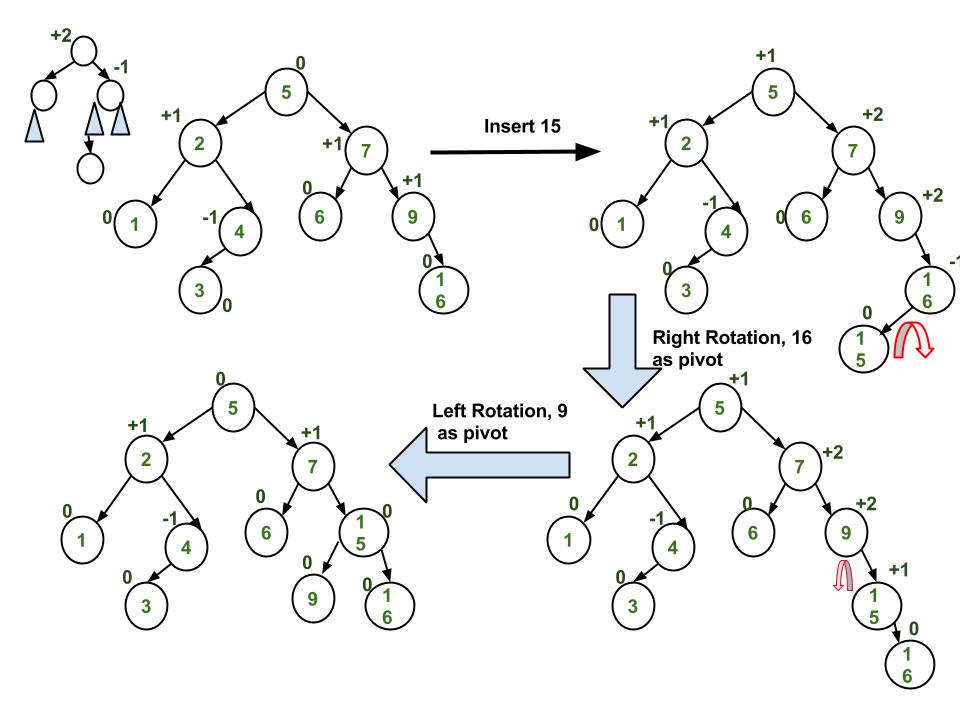
Case 3: Left rotation



Case 4: Left rotation, then right rotation



Case 5: Right rotation, then left rotation



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

###### Deletion

Every time a node is deleted, the tree must be re-balanced (if needed). And there are five cases can happen, just similar to insertion.

Implementation: <https://www.geeksforgeeks.org/avl-tree-set-2-deletion/>

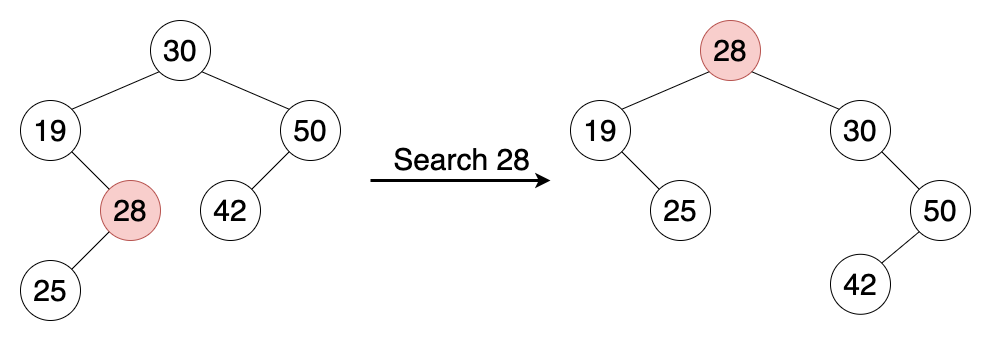
##### Applications

* Used in situations where frequent insertions are involved.
* Used in Memory management subsystem of the Linux kernel to search memory regions of processes during preemption.

#### Splay Tree

##### Definition

A splay tree is a self-balancing BST. After performing a search, insertion or deletion, splay trees perform an action called *splaying* where the tree is rearranged (using rotations) so that the **particular element is placed at the root of the tree**.



##### Basic Operations & Time Complexities

##### Applications

* Used to implement caches
* Used in garbage collectors.
* Used in data compression

#### Red-Black Tree

##### Definition

##### Basic Operations & Time Complexities

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Steps** | **Average** | **Worst** |
| **Search** |  | O(logh) | O(logh) |
| **Insert** |  | O(logh) | O(logh) |
| **Delete** |  | O(logh) | O(logh) |

Where: h is height of tree

##### Applications

* As a base for data structures used in computational geometry.
* Used in the Completely Fair Scheduler used in current Linux kernels.
* Used in the epoll system call implementation of Linux kernel.

#### Comparions

**AVL Tree vs. Red Black Tree**

The AVL tree and other self-balancing search trees like Red Black are useful to get all basic operations done in O(log n) time. The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So, if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is the more frequent operation, then AVL tree should be preferred over Red Black Tree.

**AVL Tree vs. Splay Tree**

Splay trees are simpler compared to AVL and Red-Black Trees as no extra field is required in every tree node. However, unlike AVL tree, a splay tree can change even with read-only operations like search.

### Balanced Search Tree

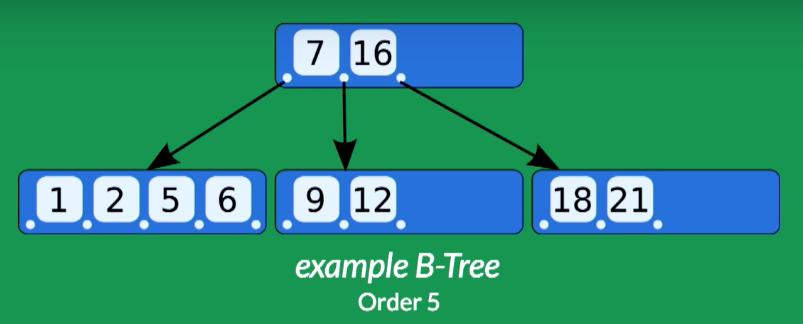
#### B-Tree

##### Definition

Should watch it first: <https://www.youtube.com/watch?v=C_q5ccN84C8&ab_channel=FullstackAcademy>

B-Tree is a self-balancing search tree, which contains **multiple nodes storing data in sorted order**. Each node can have **multiple children** and consists of **multiple keys**.

Example:



**Root node**

**Child node**

**Key**

The above B-Tree has:

* 1 root containing 2 keys: 7 and 16
* 3 child nodes. The left most node containing keys whose values are < 7. The middle node containing keys whose values are > 7 and < 16. The right most node containing keys whose values are > 16.
* Order of 5 (because its node – the left most one – has at most 5 children)

Properties:

A B-Tree of order **m** has:

* All leaves appear in the same level.
* Every node has at most **m** children.
* A non-leaf node with **k** children contains **k-1** keys.
* The root has at least two children if it is not a left node.
* Every non-left node (except the root) has at least a **ceiling of m/2** children.

##### Basic Operations & Time Complexities

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Steps** | **Average** | **Worst** |
| **Search** |  | O(logh) | O(logh) |
| **Insert** |  | O(logh) | O(logh) |
| **Delete** |  | O(logh) | O(logh) |

Where: h is height of tree

**Insertion: 7:30**

##### Applications

* Used in database indexing to speed up the search.
* Used in file systems to implement directories.

*You might not know!*

To understand the use of B-Trees, we must think of the huge amount of data which cannot fit in main memory. So, we have to put the data in disk. But accessing and reading data from disk takes much more time than from main memory.

In this case, we can take advantage of B-Trees to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, etc.) require O(h) disk accesses where h is the height of the tree. But the height of B-Trees is kept low by putting maximum possible keys in a B-Tree node, so total disk accesses for most of the operations are reduced significantly compared to balanced BST (like AVL Tree, Red-Black Tree, etc.).

To conclude, **whenever you deal with some kind of external memory and the time to access the data of a node greatly exceeds the time spent processing that data (such as big databases)**, consider using B-Trees.

## Heap

## Hash Table

## Graph

# Algorithms

## Linked List

## Stack

### Checking Balancing of Brackets

#### Problem-1: Discuss how stacks can be used for checking balancing of brackets.

**Solution**

Stacks can be used to check whether the given expression has balanced brackets. This algorithm is very useful in compilers. Each time the parser reads one character at a time. If the character is an opening delimiter such as (, {, or [ - then it is written to the stack. When a closing delimiter is encountered like ), }, or ] - the stack is popped.

The opening and closing delimiters are then compared. If they match, the parsing of the string continues. If they do not match, the parser indicates that there is an error on the line.

A linear-time O(n) algorithm based on stack can be given as:

1. Create a stack.
2. while (end of input is not reached) {
3. If the character read is not a bracket to be balanced, ignore it.
4. Else if the character is an opening bracket like (, [, {, push it onto the stack.
5. Else if it is a closing symbol like ), ], }
6. If the stack is empty, report an error. Otherwise pop the stack.
   * 1. If the bracket popped is not the corresponding opening bracket, report an error.

}

1. At end of input, if the stack is not empty report an error.

Time Complexity: O(n). Space Complexity: O(n) [for stack]. Since we are scanning the input only once (using one loop).

## Searching

### Linear Search – O(n)

**Problem**

Given an array arr[] of n elements, write a function to search a given element x in arr[].

**Examples**

Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}; x = 110;

Output: 6 -> Element x is at index 6

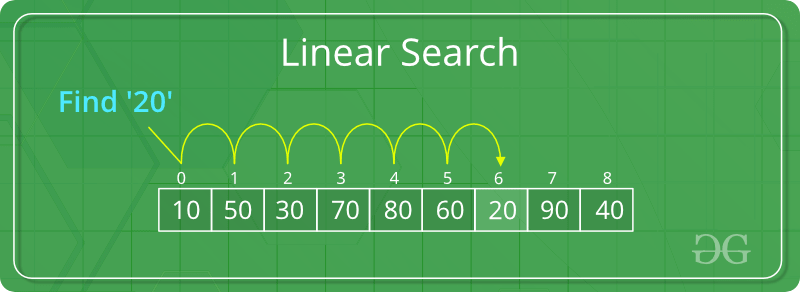
Input: arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}; x = 175;

Output: -1 -> Element x is not in arr[].

**Algorithm**

Start from the leftmost element of arr[] and one by one compare x with each element of arr[]:

1. If x matches with an element, return the index.
2. If x doesn’t match with any of elements, return -1.



**Time Complexity**:O(n)

**Code**

<https://www.geeksforgeeks.org/linear-search/>

### Binary Search – O(1) or O(logn)

**Problem**

Given a sorted array arr[] of n elements, write a function to search a given element x in arr[].

**Examples**

Input: arr[] = {10, 20, 30, 40, 50, 60}; x = 20;

Output: 1 -> Element x is at index 1

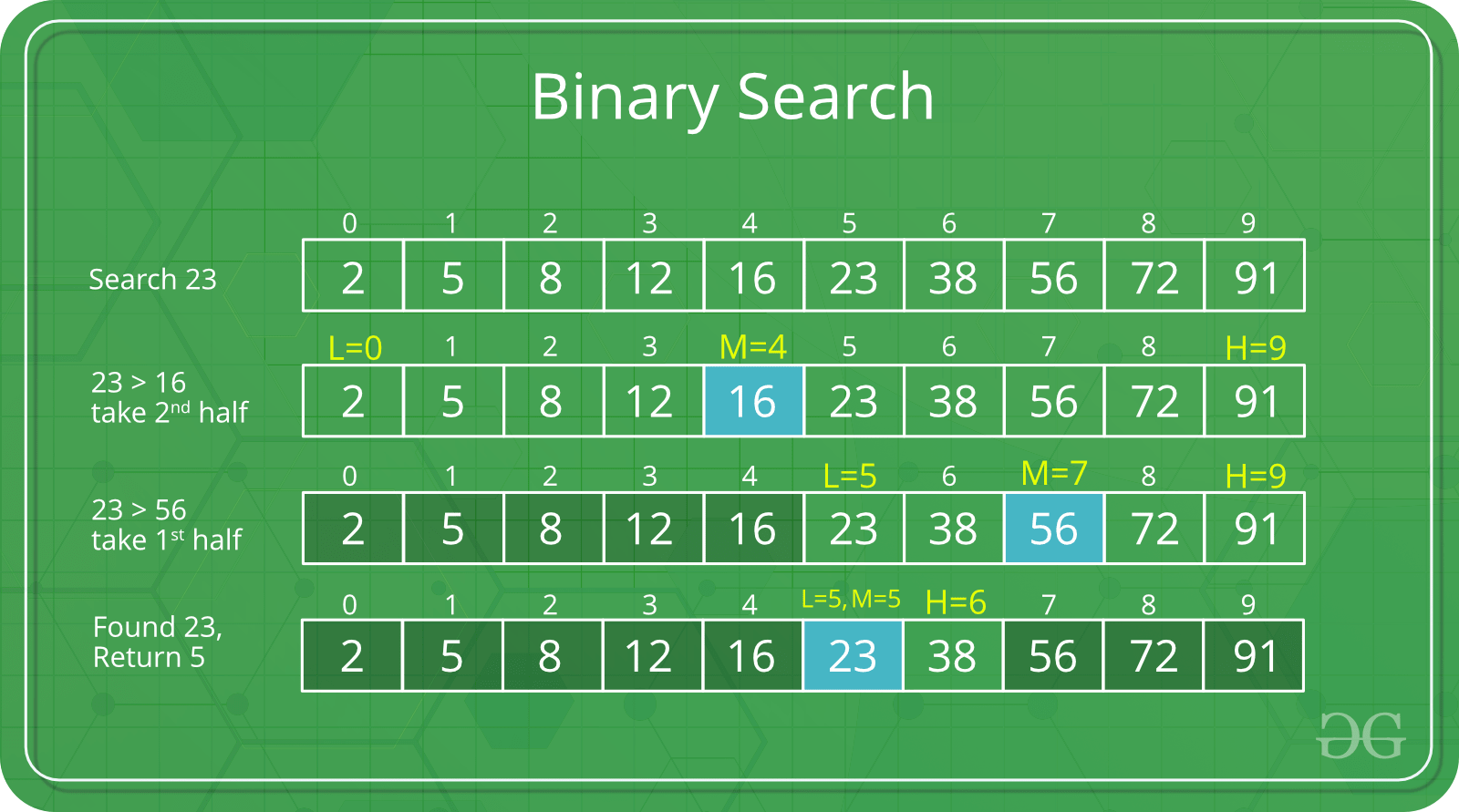
Input: arr[] = {10, 20, 30, 40, 50, 60}; x = 70;

Output: -1 -> Element x is not in arr[]

**Algorithm**

Compare x with the middle element in the array:

1. If x matches with middle element, return the mid index.
2. Else:
   1. If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
   2. Else (x is smaller) recur for the left half.



**Time Complexity**: O(1) if using iteration. Or O(logn) if using recursion.

**Code**

<https://www.geeksforgeeks.org/binary-search/>

## Sorting

### Selection Sort – O(n2)

**Problem**

Given an array arr[] of n elements, write a function to sort this array in an ascending order.

**Algorithm**

Repeatedly find the minimum element (considering ascending order) from unsorted part. Each time, the found element from the unsorted array is picked and moved to the sorted subarray (one by one, after each other).

The algorithm maintains two subarrays in a given array.

1. The subarray which is already sorted.
2. The remaining subarray which is unsorted.

Example: arr[] = { 64 25 12 11 }

// Find the minimum element in { 64 25 12 11 } and place it at the beginning of { 64 25 12 11 }:

**11** 25 12 64

// Find the minimum element in { 25 12 64 } and place it at the beginning of { 25 12 64 }:

11 **12** 25 64

// Find the minimum element in { 25 64 } place it at the beginning of { 25 64 }:

11 12 **25** 64

**Time Complexity**: O(n2) as there are two nested loops.

**Code**

<https://www.geeksforgeeks.org/selection-sort/>

### Bubble Sort – O(n2)

**Problem**

Given an array arr[] of n elements, write a function to sort this array in an ascending order.

**Algorithm**

Repeatedly swap adjacent elements in the array if they are in wrong order.

**Example**:

First pass:

{ **5 1** 4 2 8 } –> { **1 5** 4 2 8 }, compares the first two elements and swaps because 5 > 1.

{ 1 **5 4** 2 8 } –> { 1 **4 5** 2 8 }, swaps because 5 > 4

{ 1 4 **5 2** 8 } –> { 1 4 **2 5** 8 }, swaps because 5 > 2

{ 1 4 2 **5 8** } –> { 1 4 2 **5 8** }, because these elements are already in order, does not swap them.

Second pass:

{ **1 4** 2 5 8 } –> { **1 4** 2 5 8 }

{ 1 **4 2** 5 8 } –> { 1 **2 4** 5 8 }, swap because 4 > 2

{ 1 2 **4 5** 8 } –> { 1 2 **4 5** 8 }

{ 1 2 4 **5 8** } –> { 1 2 4 **5 8** }

Now, the array is already sorted, but our algorithm does not know if it is completed. It needs one more pass without any swap to know it is sorted.

Third pass:

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

{ 1 2 4 5 8 } –> { 1 2 4 5 8 }

**Time Complexity**: O(n2)

**Code**

<https://www.geeksforgeeks.org/bubble-sort/>

### Quick Sort

**Problem**

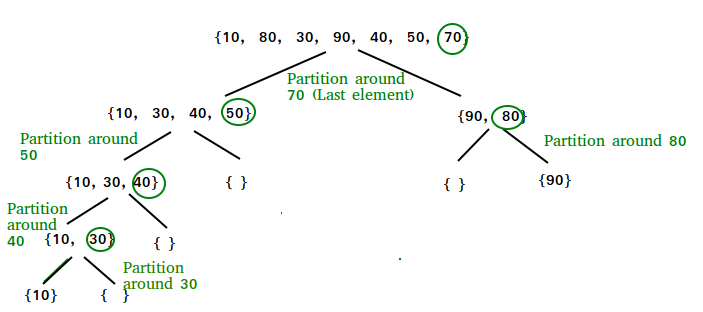
Given an array arr[] of n elements, write a function to sort this array in an ascending order.

**Algorithm**

Picks an element as pivot and partitions the array around that pivot. The pivot can be picked in different ways:

* Always pick first element as pivot.
* Always pick last element as pivot (illustration below).
* Pick a random element as pivot.
* Pick median as pivot.

The key process in quick sort is partition: put all smaller elements (than the pivot) before the pivot (if ascending order), and put all greater elements (than the pivot) after the pivot. All this should be done in linear time. Then place the pivot at its correct place.



For example:

arr[] = {10, 80, 40, 90, 30, 50, 70}

Index: 0 1 2 3 4 5 6

start = 0, end = 6, pivot = arr[end] = 70

Index of smaller element: i = -1

Traverse elements from j = start to end-1

j = 0 : Because arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 0, arr[] = {10, 80, 40, 90, 30, 50, 70} // No change as i and j are same

j = 1 : Because arr[j] > pivot, do nothing

j = 2 : Because arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 1, arr[] = {10, **40**, **80**, 90, 30, 50, 70} // We swapped 80 and 40

j = 3 : Because arr[j] > pivot, do nothing

j = 4 : Because arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2, arr[] = {10, 40, **30**, 90, **80**, 50, 70} // We swapped 80 and 30

j = 5 : Because arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

i = 3, arr[] = {10, 40, 30, **50**, 80, **90**, 70} // We swapped 90 and 50

Loop ends because j = end-1.

We place pivot at correct position by swapping arr[i+1] and arr[end] (or pivot)

arr[] = {10, 40, 30, 50, **70**, 90, **80**} // We swapped 80 and 70

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

But we have not done sorting. For each sub-array {10, 40, 30, 50} and {90, 80}, we partition again (and might again and again …) until start > end.

The final result will be arr[] = {10, 30, 40, 50, 70, 80, 90}.

**Time Complexity**

Worst case: Θ(n2)

Best case: Θ(nLogn)

**Code**

<https://www.geeksforgeeks.org/quick-sort/>

### Merge Sort

**Problem**

Given an array arr[] of n elements, write a function to sort this array in an ascending order.

**Algorithm**

**Time Complexity**

Θ(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.

**Code**

<https://www.geeksforgeeks.org/merge-sort/>