**Data Structures and Algorithms Fundamentals**

**Table of contents**

1) Data Structures Introduction (What and Why)

2) Asymptotic Analysis (O, θ, Ω)

3) Array

4) Linked List

5) Stack

6) Queue

7) Tree

8) Heap

9) Graph

10) Hash Table

11) Searching (Linear search vs binary search)

12) Recursion

13) Sorting

**1 Data Structures Introduction (What and Why)**

<https://www.youtube.com/watch?v=N-vU8VZ5ukA&list=PLPt2dINI2MIZX2EtY81WI-lDkvhKziLKM&index=2>

<https://www.youtube.com/watch?v=owCqVRbZlbg&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=2>

<https://www.youtube.com/watch?v=l0jMqpRwamM&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=2>

<https://www.youtube.com/watch?v=jGP19W5IObA&list=PL1DUmTEdeA6JlommmGP5wicYLxX5PVCQt&index=1&t=16s>

<https://www.youtube.com/watch?v=Ndc67sUKMsg>

<https://www.youtube.com/watch?v=bum_19loj9A&list=PLBZBJbE_rGRV8D7XZ08LK6z-4zPoWzu5H&index=2&t=3s>

<https://www.programiz.com/dsa/algorithm>

# **2 Asymptotic Analysis** (O, θ, Ω)

<https://www.youtube.com/watch?v=sHhVsGQz9MI&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=2&pp=iAQB>

<https://www.youtube.com/watch?v=8QDFAK8Y5Ts&list=PLwCMLs3sjOY6KH-8c9F-lMWn-r02hyoV_&index=2&pp=iAQB>

<https://www.youtube.com/watch?v=Day3_mw1F-Y&list=PLwCMLs3sjOY6KH-8c9F-lMWn-r02hyoV_&index=3&pp=iAQB>

<https://www.youtube.com/watch?v=w7gYfHx8Z1Q&list=PLwCMLs3sjOY6KH-8c9F-lMWn-r02hyoV_&index=4&pp=iAQB>

<https://www.youtube.com/watch?v=oV6YzMr1PBY&list=PLwCMLs3sjOY6KH-8c9F-lMWn-r02hyoV_&index=5&pp=iAQB>

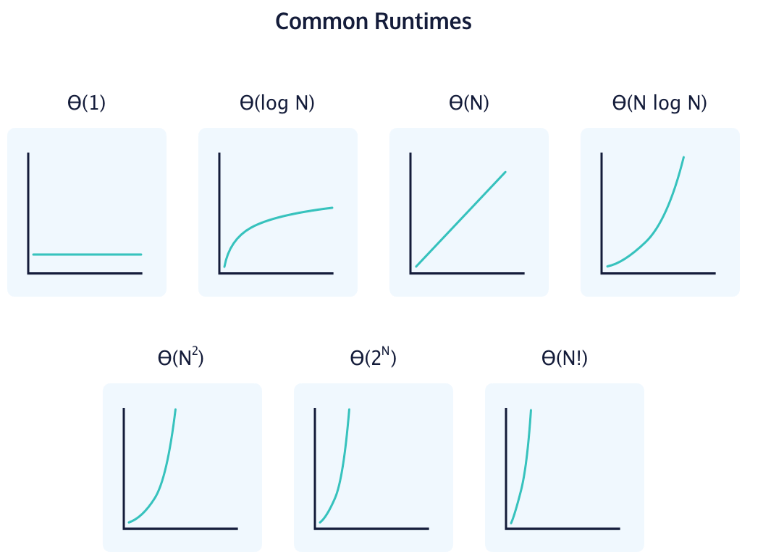
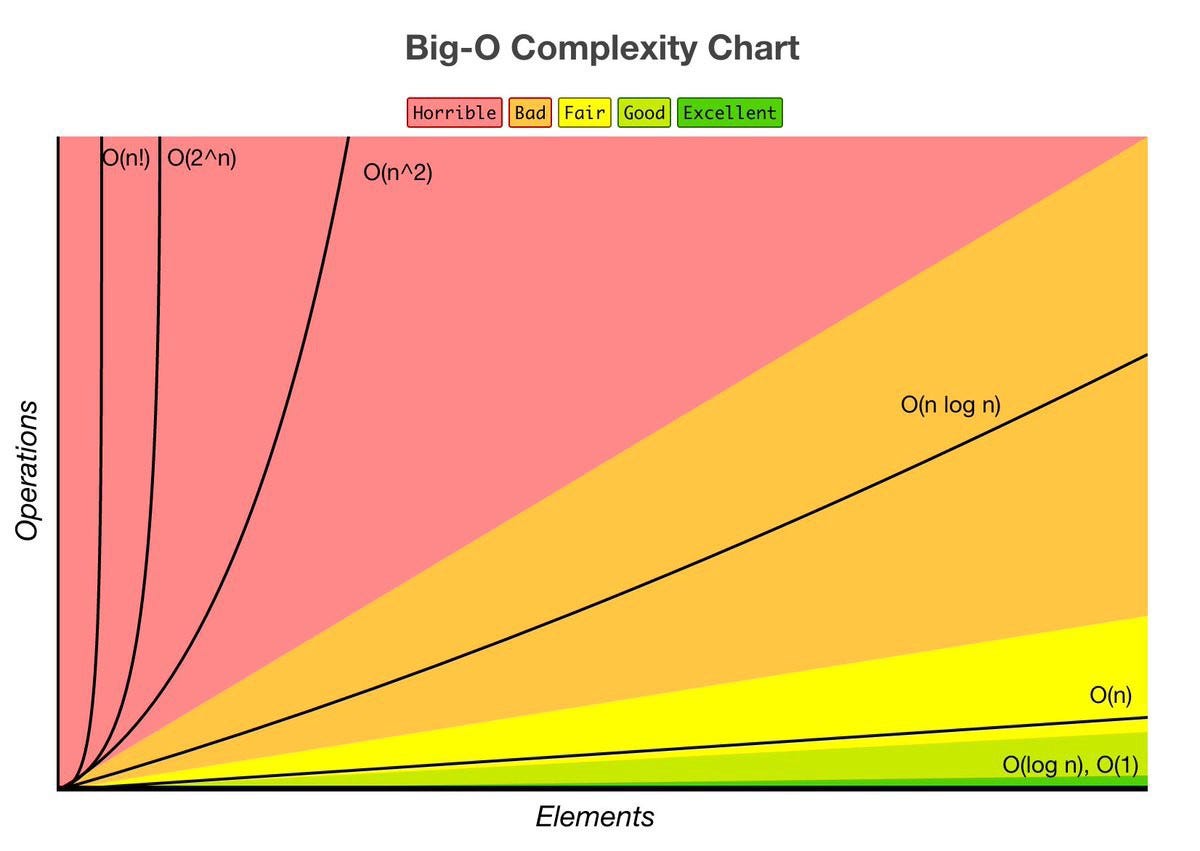
<https://www.youtube.com/watch?v=nrSVRPYy76c&list=PLwCMLs3sjOY6KH-8c9F-lMWn-r02hyoV_&index=6&pp=iAQB>

<https://www.youtube.com/watch?v=yG1l1NiJhhc&list=PLwCMLs3sjOY6KH-8c9F-lMWn-r02hyoV_&index=7&pp=iAQB>

<https://www.youtube.com/watch?v=EQzmtn4PzYQ&list=PLPt2dINI2MIYOPOhyU_5_bjhpC7J-nNDI&index=1&t=8s>

<https://www.youtube.com/watch?v=ZNYQrKpR42g&list=PLPt2dINI2MIYOPOhyU_5_bjhpC7J-nNDI&index=2>

<https://stackoverflow.com/questions/107165/big-o-for-eight-year-olds/3461832#3461832>



**3 Array**

**3.1 Array definition**

**Definition and Characteristics:**

An array is a collection of elements stored in contiguous memory locations.

Elements in an array are accessed by their index,

which is an integer value representing their position in the array.

Arrays have a fixed size, meaning the number of elements they can hold is determined

at the time of declaration.

**Array Operations:**

Accessing Elements: Elements in an array are accessed using their index.

The index starts from 0 for the first element and increases sequentially.

Insertion and Deletion: Inserting or deleting elements in an array usually involves

shifting other elements to create space or close gaps,

which can be inefficient for large arrays.

Searching: Linear search is commonly used to find an element in an array by iterating

through each element until a match is found.

Binary search can be used on sorted arrays for faster searching.

Updating Elements: Elements in an array updated by assigning a new value to a specific index.

Sorting: Arrays can be sorted in various ways, such as using algorithms like bubble sort,

insertion sort, merge sort, quicksort, or by using built-in sorting functions.

Merging and Splitting: Arrays can be merged by concatenating two arrays, and they can be split

by dividing them into smaller arrays.

**Multidimensional Arrays:**

Arrays can have multiple dimensions, such as 2D (matrix), 3D, or even higher dimensions.

Multidimensional arrays are useful for representing grids, matrices, and other structured data.

**Time Complexity:**

Arrays provide constant-time access to elements by index: O(1).

Insertion or deletion at the beginning or middle of an array requires shifting elements,

resulting in a time complexity of O(n).

Searching in an unsorted array using linear search has a time complexity of O(n),

while binary search on a sorted array has a time complexity of O(log n).

Sorting an array typically has a time complexity of O(n ^ log n) or O(n^2),

depending on the sorting algorithm used.

**Common Array Applications:**

Arrays are widely used in various programming tasks, including data structures like stacks,

queues, heaps, and hash tables.

They are used to store and process collections of data, such as lists, matrices,

graphs, and trees.

Arrays are often used to implement algorithms and solve problems across different domains.

* 1. **Array in C#**

Like other programming languages, array in C# is a group of similar types of elements

that have contiguous memory location.

In C#, array is an object of base type System.Array.

In C#, array index starts from 0.

We can store only fixed set of elements in C# array.

Advantages of C# Array:

* Code Optimization (less code)
* Random Access
* Easy to traverse data
* Easy to manipulate data
* Easy to sort data etc.

Disadvantages of C# Array:

* Fixed size

There are 3 types of arrays in C# programming:

* Single Dimensional Array
* Multidimensional Array
* Jagged Array

**// Single Dimensional Array**

static void Main(string[] args)

{

int[] arrEx1 = new int[2]; // initialize - size of arrEx1 = 2 \* 4 = 8 byte in memory

arrEx1[0] = 1; // assignment

// array taked fixed size in memory for just 2 elements

int[] arrEx2 = new int[5] { 1, 2, 3, 4, 5 }; // initialize – size = 5 \* 4 = 20 byte

// arrEx2[5] = 6; Error

int[] arrEx3 = new int[] { 1, 2, 3, 4, 5 };

int[] arrEx5 = { 1, 2, 3, 4, 5 };

}

**// Multidimensional Array**

// The multidimensional array is also known as rectangular arrays in C#.

// It can be two dimensional or three dimensional.

// The data is stored in tabular form (row \* column) which is also known as matrix.

static void Main(string[] args)

{

// 2D Array

int[,] matrix2D = new int[3, 3] {

{ 1, 2, 3 },

{ 4, 5, 6 },

{ 7, 8, 9 }

};

for (int i = 0; i < matrix2D.GetLength(0); i++)

{

for (int j = 0; j < matrix2D.GetLength(1); j++)

{

Console.Write(matrix2D[i, j] + " ");

}

}

// 1 2 3 4 5 6 7 8 9

Console.WriteLine(matrix2D[0, 0]); // 1

// There are also 3D, 4D, etc …

}

**// Jagged Array**

// In C#, jagged array is also known as "array of arrays" because its elements are arrays.

// The element size of jagged array can be different.

static void Main(string[] args)

{

int[][] jaggedArray = new int[3][];

jaggedArray[0] = new int[] { 1, 2, 3 };

jaggedArray[1] = new int[] { 4, 5 };

jaggedArray[2] = new int[] { 6 };

Console.WriteLine(jaggedArray[0][0]); // 1

for (int i = 0; i < jaggedArray.Length; i++)

{

for (int j = 0; j < jaggedArray[i].Length; j++)

{

Console.Write(jaggedArray[i][j] + " ");

}

}

// 1 2 3 4 5 6

foreach (var item in jaggedArray) Console.Write(item + " ");

// System.Int32[] System.Int32[] System.Int32[]

}

**// Params keyword**

class Program

{

static void Main(string[] args)

{

string names = (string)concate(new string[] { "shiref", "ahmed", "ali"});

Console.WriteLine(names); // shiref || ahmed || ali

string mix = (string)concate(1, "Hello", true);

Console.WriteLine(mix); // 1 || Hello || True

}

static object concate(params object[] objects)

{

return objects.Aggregate((acc, item) => $"{acc} || {item}");

}

}

**1.4 Resources**

**Google**

**Free code camp article**

<https://www.freecodecamp.org/news/data-structures-101-arrays-a-visual-introduction-for-beginners-7f013bcc355a/>

**YouTube**

**Elzero web school**

<https://www.youtube.com/watch?v=NWP1cTeDFVY>

<https://www.youtube.com/watch?v=MLVJhya1CV0&list=PLDoPjvoNmBAx3kiplQR_oeDqLDBUDYwVv&index=40>

**CS Dojo**

<https://www.youtube.com/watch?v=pmN9ExDf3yQ&t=211s>

**the roadmap**

<https://www.youtube.com/watch?v=QJNwK2uJyGs>

**JS/TS Array Implementation Path:** code\ src\ Array.ts

**4 Linked List**

**4.1 Linked List definition**

In computer science, a linked list is a data structure used to store and organize a collection of elements. It consists of a sequence of nodes, where each node contains two components: data and a reference (or a link) to the next node in the sequence.

**Here's a step-by-step explanation of how a linked list works:**

* Node Structure: Each node in a linked list contains two parts:

Data: This is the actual value or information that the node holds.

It can be any type of data, such as an integer, a character,

or even a complex object.

Next Reference: This is a reference or a pointer that points to the next

node in the sequence. It holds the memory address of the next node.

Head Node: The linked list is typically represented by a reference to

the first node in the sequence, called the "head" node.

* Creating Nodes: To create a linked list, you create individual nodes and connect them

together. Each node is allocated dynamically using memory allocation

techniques like "malloc" or "new" in programming languages like C/C++

or Java.

* Linking Nodes: Once you have created multiple nodes, you establish the links between

them. The "next" reference of a node points to the memory location of

the next node in the sequence. The last node's "next" reference is set

to null or a special value to indicate the end of the list.

* Traversing the List: To access or traverse the linked list, you start from the head

node and follow the next references until you reach the end of the

list (i.e., a node with a null reference). This allows you to

iterate over each element in the list sequentially.

* Insertion and Deletion: One of the advantages of linked lists is the ease of insertion

and deletion operations. To insert a new node, you update the

next reference of a previous node to point to the new node and

update the next reference of the new node to point to the next

node. To delete a node, you update the next reference of the

previous node to skip the node you want to delete.

**Types of Linked Lists: There are different types of linked lists, including:**

* Singly Linked List: Each node contains a reference to the next node.
* Doubly Linked List: Each node contains references to both the next and previous nodes,

allowing traversal in both directions.

* Circular Linked List: The last node's next reference points back to the head node,

forming a cycle.

**Linked lists have several advantages and disadvantages:**

* They are efficient for insertion and deletion operations.
* but accessing elements by index is slower than in arrays O(n) vs O(1).
* They also have dynamic size, meaning they can grow or shrink as needed.

Linked lists are fundamental data structures and are widely used in various algorithms and applications. Understanding linked lists is essential for building a strong foundation in computer science and programming.

* 1. **Linked List Types and Its Implementation**

**4.2.1 Single Linked List**



**Implementation Path:** code\ src\ LinkedList.SingleLinkedList.ts

**4.2.2 Doubly Linked List**



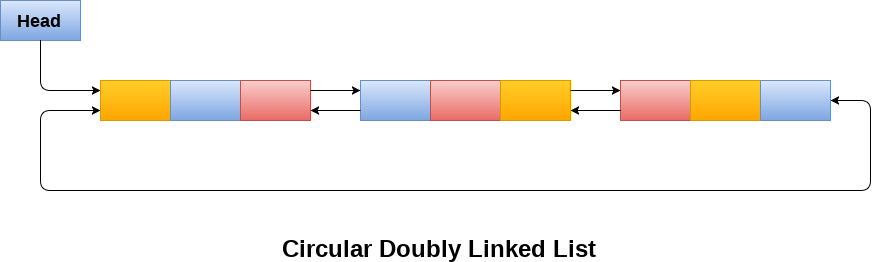
**Implementation Path:** \_

**4.2.3 Circular Single Linked List**



**Implementation Path:** code\ src\ LinkedList.CircularSingleLinkedList.ts

**4.2.4 Circular Doubly Linked List**



**Implementation Path: \_**

**4.3 Resources**

**Google**

**Programiz**

<https://www.programiz.com/dsa/linked-list>

**JavaTPoint**

<https://www.javatpoint.com/ds-linked-list>

**YouTube**

**Adel Nasim**

<https://www.youtube.com/watch?v=t7WnaypBoNw&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=14&pp=iAQB>

<https://www.youtube.com/watch?v=t7iviFyL3K4&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=15&t=4s&pp=iAQB>

<https://www.youtube.com/watch?v=_v8lL7zMlFc&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=16&pp=iAQB>

<https://www.youtube.com/watch?v=8kvQ-O10h4k&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=17&pp=iAQB>

<https://www.youtube.com/watch?v=9xtRgr2V2mY&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=18&pp=iAQB>

**Arabic Competitive Programming**

<https://www.youtube.com/watch?v=e6zJaMxzciA&list=PLPt2dINI2MIZX2EtY81WI-lDkvhKziLKM&index=2&pp=iAQB>

**CS Dojo**

<https://www.youtube.com/watch?v=WwfhLC16bis>

**the roadmap**

<https://www.youtube.com/watch?v=odW9FU8jPRQ>

**Search For ...**

**Skip List Data Structure from JavaTPoint**

<https://www.javatpoint.com/skip-list-in-data-structure>

1. **Stack**
   1. **Stack definition**

A stack is a fundamental data structure that follows the Last-In-First-Out (LIFO) principle.

It is an abstract data type that represents a collection of elements,

where elements are added or removed from the top of the stack.

The topmost element is the one that was most recently added,

and it's the only element accessible for manipulation.

**The stack has two primary operations:**

* Push: This operation adds an element to the top of the stack.

The new element becomes the top of the stack, and any existing elements are pushed down.

* Pop: This operation removes the topmost element from the stack.

The element that was below the topmost element becomes the new top of the stack.

**In addition to these core operations, stacks often provide other auxiliary operations,**

**such as:**

* Peek: This operation retrieves the value of the topmost element without removing

it from the stack.

* IsEmpty: This operation checks whether the stack is empty or not.
* Size: This operation returns the number of elements in the stack.

**Stacks can be implemented using various underlying data structures,**

**with the most common options being arrays and linked lists:**

* When implementing a stack with an array, you allocate a fixed-size array and keep track of the index of the topmost element.

Each push operation increments the top index and places the new element at that index.

Similarly, each pop operation removes the element at the top index and decrements

the index.

* Alternatively, a stack can be implemented using a linked list.

In this case, each element in the stack is represented by a node containing

the element's value and a reference to the next node.

The top of the stack is indicated by the head of the linked list.

When pushing a new element, a new node is created and linked as the new head,

while popping removes the head node and updates the reference to the next node.

**Stacks find applications in various areas of computer science. Some common use cases include:**

* Function call stack: Stacks are used to store information about function calls

in programming languages. When a function is called, its state

(such as local variables and return address) is pushed onto the stack,

and when the function completes, its state is popped from the stack to

resume execution at the calling point.

* Expression evaluation: Stacks are used to evaluate arithmetic expressions by converting

them from infix notation to postfix or prefix notation and then

calculating the result using the stack.

* Undo/Redo functionality: Stacks can be used to implement undo and redo operations in

applications, where each action is pushed onto the stack,

and undoing a step pops the last action from the stack.

* Backtracking algorithms: Stacks are often used in backtracking algorithms,

such as depth-first search, to keep track of the visited nodes

or the current state of exploration.

Understanding stacks is crucial for computer science students as they form the basis

for more complex data structures like queues, trees, and graphs.

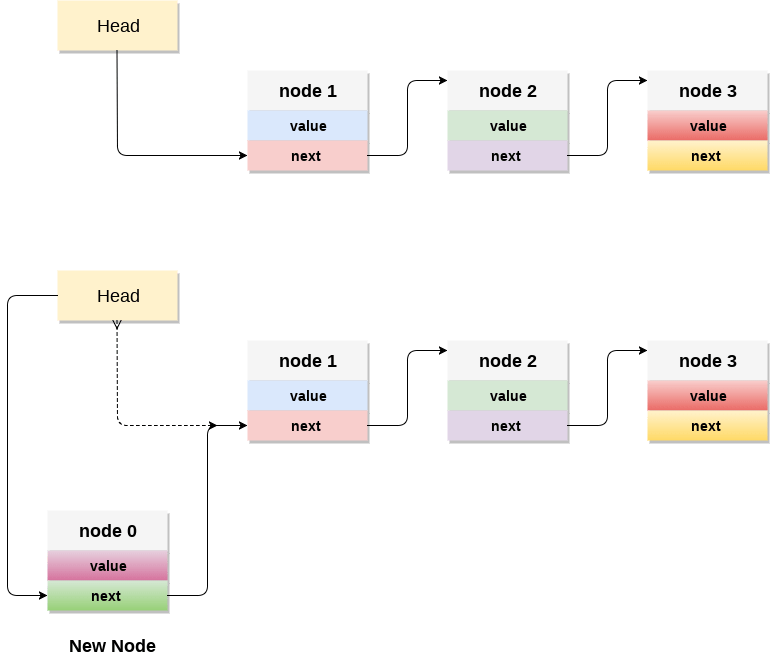
The LIFO property of stacks provides a simple and efficient way to manage data

in various scenarios.

* 1. **Stack Implementation**

**Implementation Path:** code\ src\ Stack.StackByArray.ts

**Implementation Path:** code\ src\ Stack.StackByLinkedList.ts



**One of stack apps**

**Check if Infix Ex is correct, converting from infix to postfix, calculating postfix Ex**

**Implementation Path:** code\ src\ Stack.Calculator.ts

**5.3 Resources**

**Google**

**Programiz**

<https://www.programiz.com/dsa/stack>

**JavaTPoint**

<https://www.javatpoint.com/data-structure-stack>

**YouTube**

**Adel Nasim**

<https://www.youtube.com/watch?v=vdPNQzgBu_I&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=3&pp=iAQB>

<https://www.youtube.com/watch?v=N4o69Gqarhc&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=4&pp=iAQB>

<https://www.youtube.com/watch?v=9BZlgbZFR_c&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=5&pp=iAQB>

<https://www.youtube.com/watch?v=XSm6MivqBrM&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=6&pp=iAQB>

<https://www.youtube.com/watch?v=PLvD3pHaWHQ&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=7&pp=iAQB>

<https://www.youtube.com/watch?v=Q4X7pZ5pyA4&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=8&t=4s&pp=iAQB>

<https://www.youtube.com/watch?v=xhcChs9jijM&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=9&pp=iAQB>

**Hard-code**

<https://www.youtube.com/watch?v=Rrc1D4E7mTQ&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=2&pp=iAQB>

<https://www.youtube.com/watch?v=dIpf5TKcr88&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=3&pp=iAQB>

<https://www.youtube.com/watch?v=71V60ZpH-8M&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=4&pp=iAQB>

<https://www.youtube.com/watch?v=g1Zwz2kSKEU&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=5&pp=iAQB>

<https://www.youtube.com/watch?v=b8RHZ78aMws&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=6&pp=iAQB>

<https://www.youtube.com/watch?v=GKDtCqsM5uY&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=7&pp=iAQB>

**Arabic Competitive Programming**

<https://www.youtube.com/playlist?list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS>

**The roadmap**

<https://www.youtube.com/watch?v=odW9FU8jPRQ>

1. **Queue**
   1. **Queue definition**

**Queue Introduction:**

In computer science, a queue is an abstract data type (ADT) that represents a collection

of elements in a specific order.

It follows the First-In-First-Out (FIFO) principle, which means

that the element that has been in the queue the longest is the first one to be removed.

Queues are commonly used in various applications and algorithms,

such as job scheduling, handling requests, breadth-first search, and more.

They provide an efficient way to manage a sequence of elements and process them

in a specific order.

**Key Characteristics of a Queue:**

* FIFO: The element that has been in queue the longest is the first one to be removed.
* Enqueue: Adding an element to the end of the queue.
* Dequeue: Removing the element from the front of the queue.
* Peek/Top: Accessing the element at the front of the queue without removing it.
* Size: The number of elements currently present in the queue.
* Empty: Checking if the queue is empty.

**Operations on a Queue:**

* Enqueue: Adds an element to the end of the queue.
* Dequeue: Removes the element from the front of the queue.
* Peek/Top: Retrieves the element at the front of the queue without removing it.
* Size: Returns the number of elements present in the queue.
* Empty: Checks if the queue is empty and returns a boolean value.

**Queue Implementation:**

Queues can be implemented using various data structures, such as arrays and linked lists.

The choice of implementation depends on the specific requirements of your application.

**Queue Applications:**

Queues are useful in scenarios where you need to maintain the order of elements

and process them in a specific sequence. Some common use cases include:

* Task scheduling: Processing a set of tasks in the order they were received.
* Print spooling: Managing print jobs in the order they were submitted.
* Event handling: Handling events in the order they occurred.
* Breadth-first search: Exploring nodes in a graph level by level.

**Implementation Path:** code\ src\ Queue.QueueByArray.ts

**Implementation Path:** code\ src\ Queue.QueueByLinkedList.ts

**Note:** There is special type of queue called priority queue which its implementation by min/max-heap you will see it in heap section**.**

**6.2 Resources**

**Google**

**Programiz**

<https://www.programiz.com/dsa/queue>

**JavaTPoint**

<https://www.javatpoint.com/data-structure-queue>

**YouTube**

**Adel Nasim**

<https://www.youtube.com/watch?v=8t_tzT52br8&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=10&pp=iAQB>

<https://www.youtube.com/watch?v=bsk-mNsHhzA&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=11&t=938s&pp=iAQB>

<https://www.youtube.com/watch?v=VhRKjM759W8&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=12&pp=iAQB>

**Hard-code**

<https://www.youtube.com/watch?v=cxJhNUo2dpQ&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=8&pp=iAQB>

<https://www.youtube.com/watch?v=3yKxwHVzisE&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=9&pp=iAQB>

<https://www.youtube.com/watch?v=JNrev_FBqF4&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=10&pp=iAQB>

**Arabic Competitive Programming**

<https://www.youtube.com/watch?v=PJHcOVmkRbw&list=PLPt2dINI2MIZX2EtY81WI-lDkvhKziLKM&index=3&pp=iAQB>

**The roadmap**

<https://www.youtube.com/watch?v=mDCi1lXd9hc&list=PLkZYeFmDuaN2-KUIv-mvbjfKszIGJ4FaY&index=5&pp=iAQB>

**Search For ...**

**Deque Data Structure from JavaTPoint**

<https://www.javatpoint.com/ds-deque>

1. **Tree**

**7.1 Introduction**

****

**Some basic terms used in Tree data structure.**

Root - Child node - Parent node – Siblings - Leaf Node - Internal nodes - Ancestor node – Descendant node.

**Properties of Tree data structure**

* **Recursive data structure:** The tree is also known as a **recursive data structure**. Recursion means reducing something in a self-similar manner.
* **Number of edges:** If there are n nodes, then there would n-1 edges. Each node, except the root node, will have at least one incoming link known as an edge.
* **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. The root node has 0 depth.
* **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

**Based on the properties of the Tree data structure, trees are classified into various categories.**

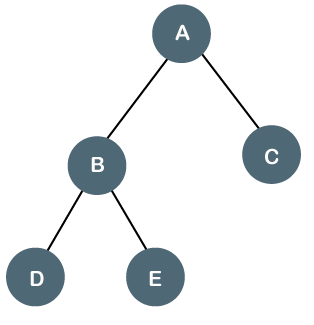
General tree - [Binary tree](https://www.javatpoint.com/binary-tree) - [Binary Search tree](https://www.javatpoint.com/binary-search-tree) - [AVL tree](https://www.javatpoint.com/avl-tree) - [Red-Black Tree](https://www.javatpoint.com/red-black-tree) - Splay tree – Treap - [B-tree](https://www.javatpoint.com/b-tree)

**7.2 Binary Tree**

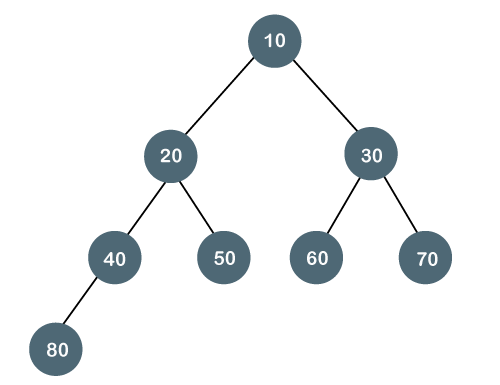
A binary tree is a hierarchical data structure in which each node can have at most two children, known as the left child and the right child

**Types of Binary Trees:**

**Full Binary Tree:** A binary tree is considered full if every node has either zero or two children.



**Complete Binary Tree:** A binary tree is called complete if all levels, except possibly the last one, are completely filled, and all nodes are as left as possible.

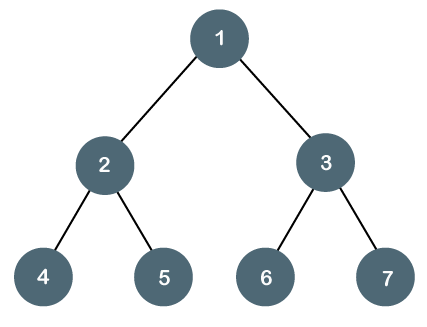


**Perfect Binary Tree:** A binary tree is perfect if all levels are completely filled with nodes, meaning it is both full and complete.

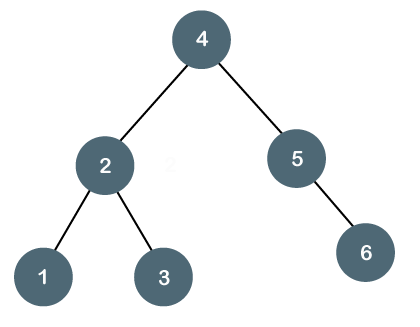
Nodes of level = 2 ^ level.

Nodes of Tree = 2 ^ (height of tree + 1) – 1.

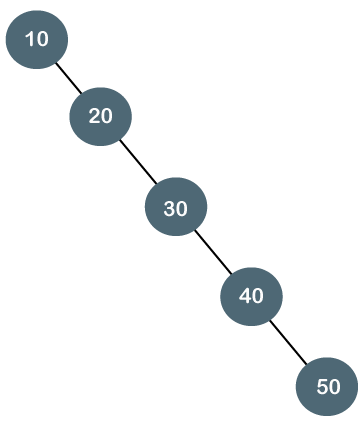
Height of tree = log2(Nodes of Tree + 1) – 1.



**Balanced Binary Tree:** A binary tree is considered balanced if the height difference between the left and right subtrees of every node is at most one. Examples of balanced binary trees include AVL trees and red-black trees.



**Degenerate (or Pathological) Binary Tree:** A degenerate binary tree is a tree in which each parent node has only one child. It essentially becomes a linked list.



**Binary Tree Traversal:**

Traversal refers to the process of visiting each node in a tree data structure.

There are three common methods for traversing binary trees:

**Inorder Traversal:** left – root – right.

**Preorder Traversal:** root – left – right.

**Postorder Traversal:** left – right – root.

**Binary Tree Operations:**

**Insertion:** To insert a new node into a binary tree, you typically start from the root and traverse the tree until you find an appropriate spot to insert the new node as a leaf node.

**Deletion:** Deleting a node from a binary tree involves finding the node to be deleted, handling different cases depending on the node's children, and rearranging the tree accordingly.

**Searching:** To search for a specific value in a binary tree, you can start from the root and recursively traverse the tree by comparing the target value with each node until you find a match or reach a leaf node.

**Applications of Binary Trees:**

**Binary Search Trees (BSTs):** Binary search trees are binary trees that follow a specific ordering property, making them efficient for searching, insertion, and deletion operations.

**Expression Trees:** Binary trees can be used to represent mathematical expressions, with operators as internal nodes and operands as leaf nodes.

**Huffman Trees:** Huffman trees are binary trees used in data compression algorithms, where characters or symbols with higher frequencies have shorter codes.

**Decision Trees:** Binary trees can be used to represent decision-making processes, where each node represents a decision or test condition.

Understanding binary trees is crucial as they serve as a foundation for various advanced tree-based data structures and algorithms. Mastering the concepts of binary trees can enhance your understanding of more complex tree structures and their applications in computer science.

**7.3 Binary Search Tree BST**

**Definition:**

A binary search tree is a binary tree in which the value of every node is greater than

all values in its left subtree and less than all values in its right subtree.

The property of a BST allows for efficient searching, insertion, and deletion operations.

**Operations:**

Searching: To search for a specific value in a BST, you compare the target value with

the value of the current node and traverse left or right accordingly until

you find a match or reach a leaf node.

The time complexity for searching in a balanced BST is O(log n) on average.

Insertion: To insert a new value into a BST, compare it with the value of each node

and navigate left or right until reaching an empty spot.

Maintain the BST property by adjusting the tree structure if necessary.

The time complexity for insertion in a balanced BST is also O(log n) on average.

Deletion: Deleting a node from a BST involves finding the node to be deleted,

handling different cases depending on the node's children, and rearranging

the tree accordingly.

The time complexity for deletion in a balanced BST is O(log n) on average.

**Balanced Binary Search Trees:**

Balanced BSTs, such as AVL trees and red-black trees, are self-adjusting binary search trees

that ensure the tree remains balanced to guarantee efficient operations.

AVL trees maintain a balance factor for each node, and if the balance factor

becomes greater than 1 or less than -1, rotations are performed to restore balance.

Red-black trees use color annotations on nodes and apply specific rules to maintain

balance during insertion and deletion operations.

**Operations for Sorted Data:**

BSTs are particularly suited for handling sorted data efficiently.

In-order Traversal: In a BST, in-order traversal visits nodes in ascending order,

making it useful for retrieving the elements in sorted order.

Finding Successor/Predecessor: BSTs allow finding the (smallest value greater than a given

value) or predecessor (largest value smaller than a given

value) efficiently.

**Time Complexity:**

The time complexity of operations in a balanced BST is O(log n) on average.

In the worst-case scenario, when the BST is highly unbalanced (degenerate),

the time complexity can degrade to O(n), similar to a linked list.

**Applications:**

BSTs are commonly used in various applications, such as dictionary implementations,

symbol tables, database indexing, and range searching.

Understanding the concepts and operations of binary search trees is crucial

for designing efficient algorithms and data structures.

It's also important to be aware of balanced BSTs and their role in maintaining

optimal performance.

Additionally, considering the limitations of BSTs in the worst-case scenario

can help you decide when to choose alternative data structures.

**Implementation Path:** code\ src\ Tree.BinarySearchTree.ts

**Adel Nasim Data Structure Course**

<https://www.youtube.com/watch?v=br-1squJfkA&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=23&pp=iAQB>

**Abdul bari Algorithm Course**

<https://www.youtube.com/watch?v=jDM6_TnYIqE>

AVL Tree comes to solve unbalanced binary search tree and avoid Binary Search Tree to become degenerated tress like linked list, so AVL tree makes sure balance factor happen in tree witch -1, 0, 1.

**Time Complexity:** In All cases for inserting, searching and delete O(log(n)).

**Space Complexity**: O(n).

AVL Tree may make more rotation in insert and delete node so red-black tree comes

To solve such that issue by make rotation only in specific conditions like:

Different between left and right subtrees 1:2 - 3:6 - 50:100.

**Implementation Path:** code\ src\ Tree.AVLTree.ts

**7.5 Resources**

**Google**

**Programiz**

<https://www.programiz.com/dsa/trees>

**YouTube**

**Adel Nasim Data Structure Course**

<https://www.youtube.com/watch?v=XDDZNL-yG2U&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=19&t=17s&pp=iAQB>

<https://www.youtube.com/watch?v=abaxG_-P9Ug&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=20&pp=iAQB>

<https://www.youtube.com/watch?v=tFsfKTTJD6I&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=21&t=517s&pp=iAQB>

<https://www.youtube.com/watch?v=MikbKCPVsBY&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=22&pp=iAQB>

<https://www.youtube.com/watch?v=br-1squJfkA&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=23&pp=iAQB>

**Hard-code Data Structure Course**

<https://www.youtube.com/watch?v=7Gg8bbshTIw&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=22&pp=iAQB>

<https://www.youtube.com/watch?v=oNJfm5Gnb_I&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=23&pp=iAQB>

<https://www.youtube.com/watch?v=1BfVu4hmSmE&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=24&pp=iAQB>

<https://www.youtube.com/watch?v=kCFPcWdqxmQ&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=25&pp=iAQB>

<https://www.youtube.com/watch?v=9uF63PUeZmM&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=26&t=9s&pp=iAQB>

<https://www.youtube.com/watch?v=IAVj3OThMnY&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=27&pp=iAQB>

<https://www.youtube.com/watch?v=fJiPtIb01Hw&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=28&pp=iAQB>

<https://www.youtube.com/watch?v=sdmOFkllNEw&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=29&pp=iAQB>

<https://www.youtube.com/watch?v=O8pI0TeXUR4&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=30&pp=iAQB>

<https://www.youtube.com/watch?v=vzkSM0G657k&list=PLwCMLs3sjOY4UQq4vXgGPwGLVX1Y5faaS&index=31&t=1108s&pp=iAQB>

**Abdul bari Algorithm Course**

<https://www.youtube.com/watch?v=aZjYr87r1b8&t=84s>

**Search For ...**

**B, B+, Red-black Trees Programiz and Abdul bari Algorithm Course**

<https://www.programiz.com/dsa/b-tree>

1. **Heap**

**––**

**8.1 Definition**

**Heap Data Structure:**

A heap is a complete binary tree-based data structure that satisfies the heap property.

The heap property states that for every node in the heap, the value of the node is

either greater than or equal to (in a max-heap) or less than or equal to (in a min-heap)

the values of its children.

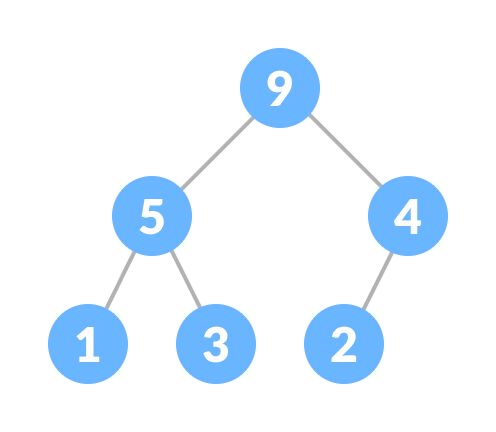
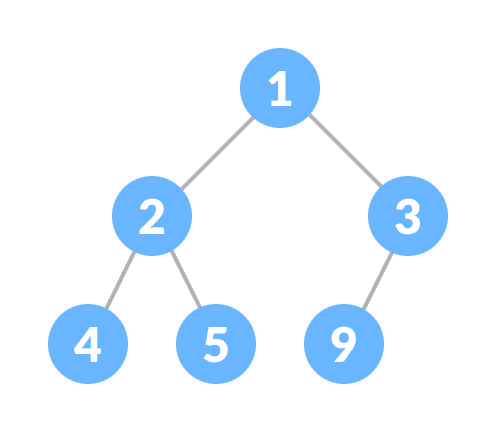
In other words, the root node of a max-heap contains the maximum value in the heap,

while the root node of a min-heap contains the minimum value.

Heaps are commonly used to implement priority queues, where elements with higher priority

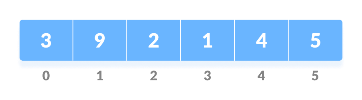
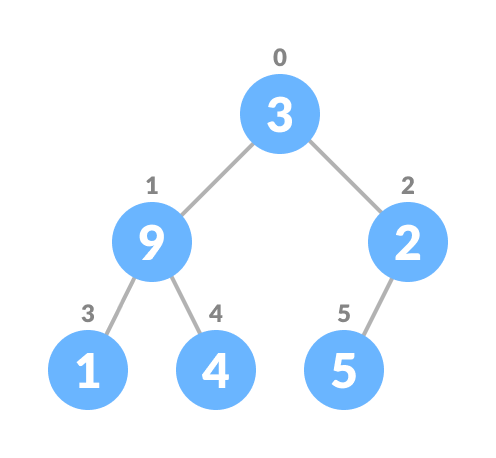
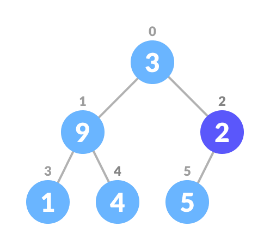
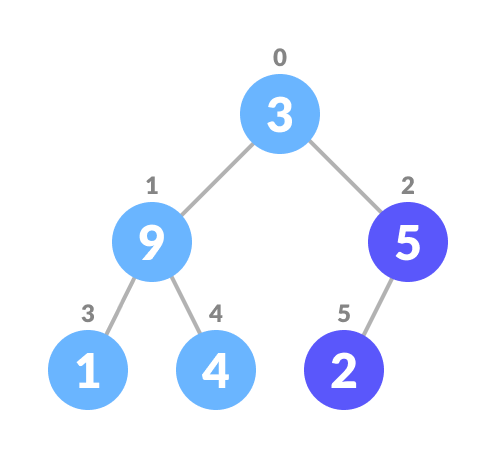
are dequeued before elements with lower priority.

**Max-heap vs Min-heap**:

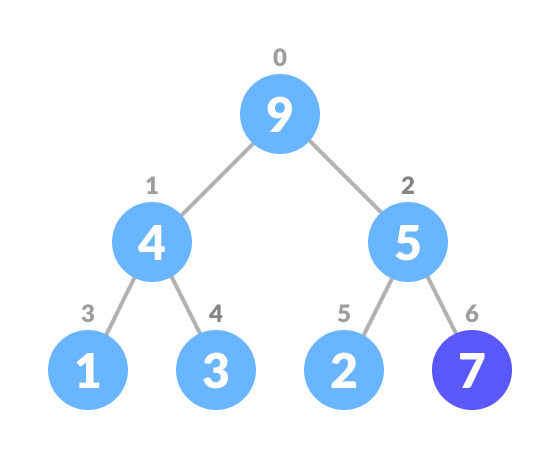
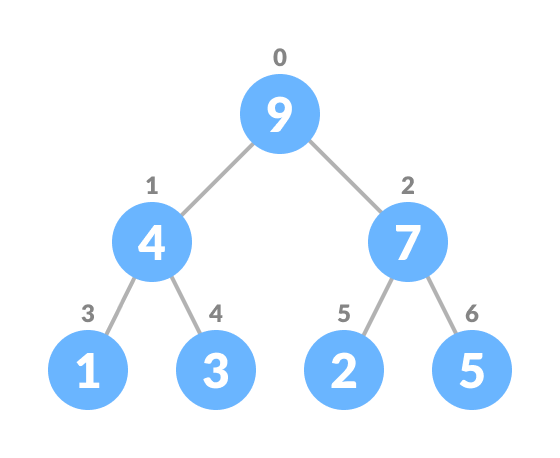
 

**Heap Operations:**

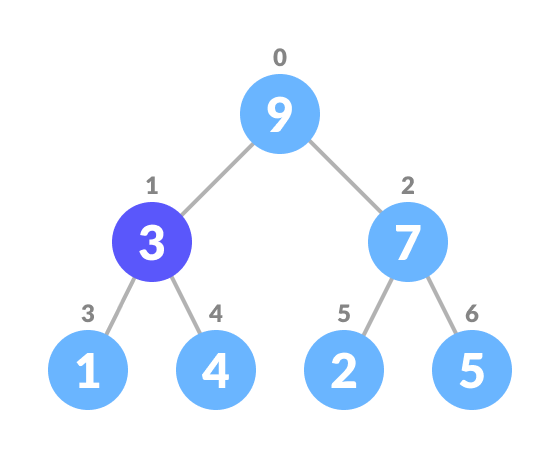
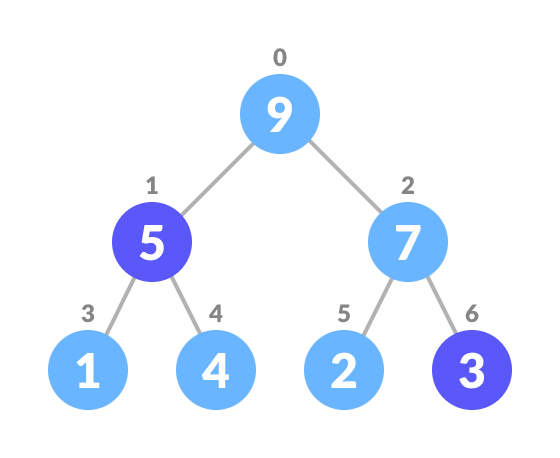
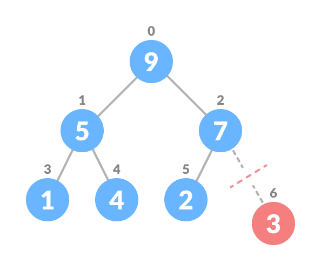
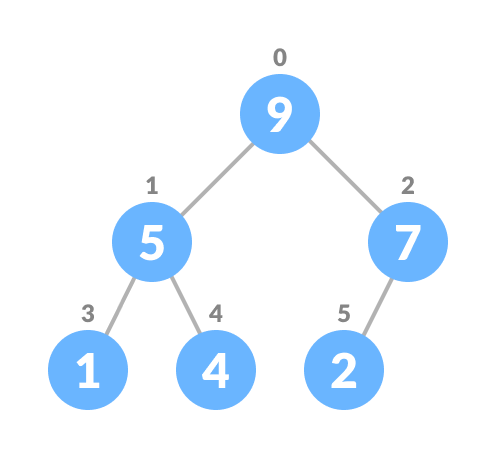
* Heapify

* Insertion:

* Deleting:

* Get min or max: return root node.

**8.2 Complexity**

**Time Complexity:**

**Insertion**: The time complexity of inserting an item into the heap is O(log n),

where n is the number of elements in the heap.

This is because the item is inserted at the bottom of the heap and then moved up the tree

by potentially swapping with its parent until the heap property is satisfied.

**Removal**: Both removing the maximum item (RemoveMax) and removing an item at a specific index

(RemoveAt) have a time complexity of O(log n).

This is because the item being removed is replaced with the last item in the heap,

and then heapification is performed by potentially swapping the item with its children

until the heap property is satisfied.

**Peek**: The time complexity of peeking at the maximum item (PeekMax) is O(1) as it simply

returns the value at the root node of the heap.

**Space Complexity:**

The space complexity of the heap data structure is O(n),

where n is the number of elements in the heap.

This space is required to store the elements in the heap array.

It's worth noting that the time and space complexity analysis assumes a balanced heap.

In the worst-case scenario, where the heap is completely unbalanced,

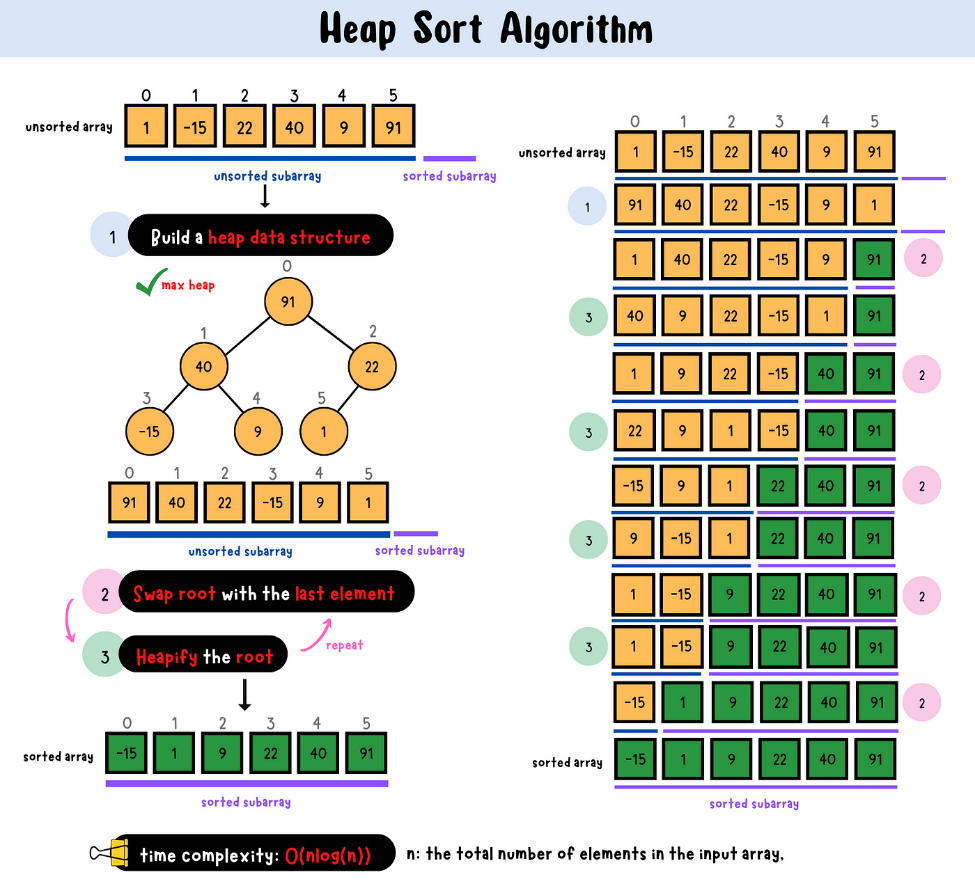
the time complexity can be O(n) for certain operations like building the heap.

However, in practice, heaps tend to be balanced, and the average-case time complexity

remains O(log n) for most operations.

**Implementation Path:** code\ src\ Heap.Heap.ts

**8.3. Heap Sort**



**Implementation Path:** code\ src\ Sorting.ts

**8.4. Resources**

**Google**

**Programiz**

<https://www.programiz.com/dsa/heap-data-structure>

<https://www.programiz.com/dsa/heap-sort>

**YouTube**

**Adel Nasim**

<https://www.youtube.com/watch?v=REOsj0nYWKE&pp=ygUPaGVhcCBhZGVsIG5hc2lt>

**Arabic Competitive programming**

<https://www.youtube.com/watch?v=8GqjRhlmccg&list=PLPt2dINI2MIZX2EtY81WI-lDkvhKziLKM&index=6&pp=iAQB>

**The Road Map**

<https://www.youtube.com/watch?v=F_r0sJ1RqWk>

**Abdul Bari**

<https://www.youtube.com/watch?v=HqPJF2L5h9U&t=248s&pp=ygUOaGFyZC1jb2RlIGhlYXA%3D>

1. **Graph**

**9.1 Graph Introduction**

<https://www.programiz.com/dsa/graph>

**9.2 Spanning Tree and Minimum Spanning Tree**

<https://www.programiz.com/dsa/spanning-tree-and-minimum-spanning-tree>

**9.3 Strongly Connected Components**

<https://www.programiz.com/dsa/strongly-connected-components>

**9.4 Adjacency Matrix**

<https://www.programiz.com/dsa/graph-adjacency-matrix>

**9.5 Adjacency List**

<https://www.programiz.com/dsa/graph-adjacency-list>

**9.6 DFS**

<https://www.programiz.com/dsa/graph-dfs>

**9.7 BFS**

<https://www.programiz.com/dsa/graph-bfs>

**9.8 Graph Implementation Adjacency Matrix and Adjacency List**

**Implementation Path:** code\ src\ Graph.ts

**9.9 Resources**

**Google**

**Programiz**

<https://www.programiz.com/dsa/graph>

**YouTube**

**Adel Nasim Data Structure Course**

<https://www.youtube.com/watch?v=R74DnYySxv0&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=34&pp=iAQB>

<https://www.youtube.com/watch?v=pNWCCHFw7og&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=35&pp=iAQB>

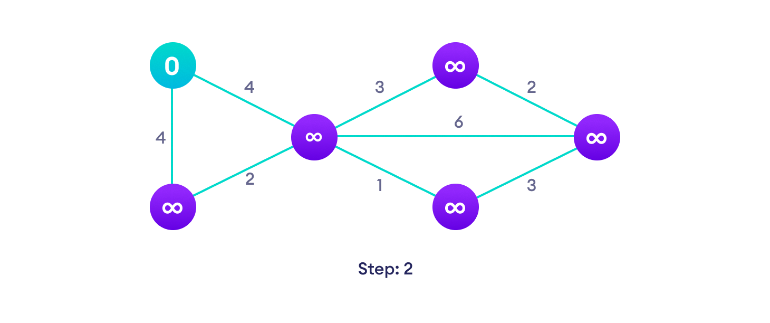
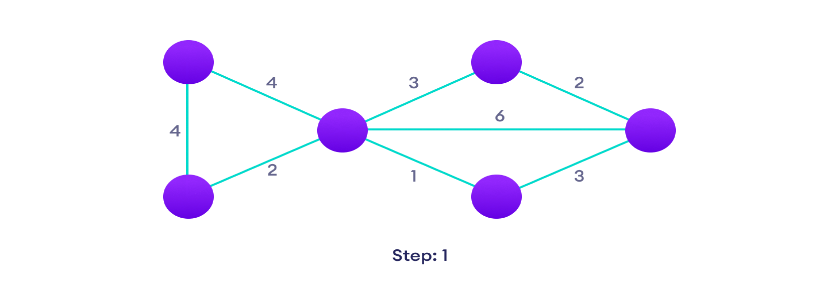
<https://www.youtube.com/watch?v=vAvPkrLbczg&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=36&pp=iAQB>

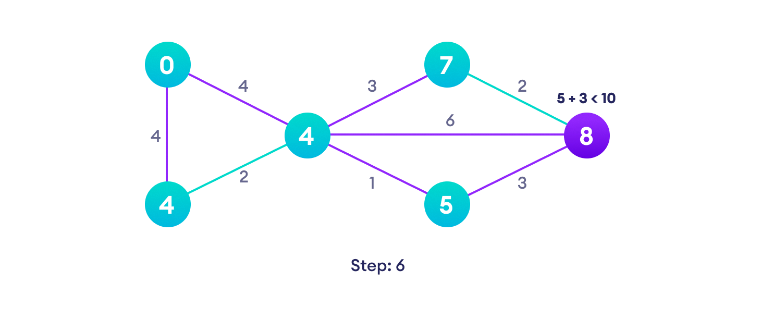
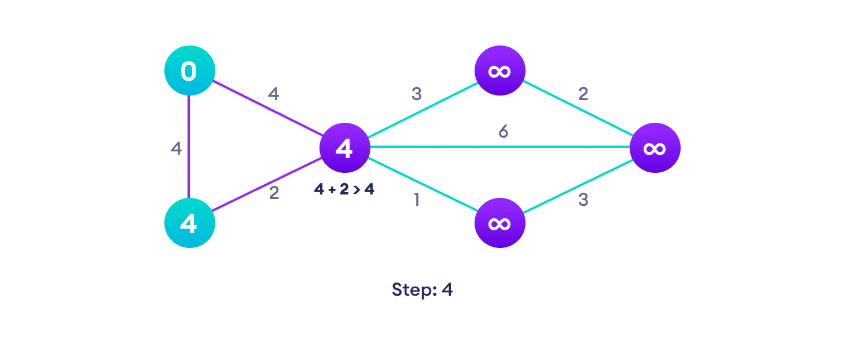
**Arabic Competitive programming**

<https://www.youtube.com/watch?v=jzfcfQVBtKA&list=PLPt2dINI2MIZX2EtY81WI-lDkvhKziLKM&index=9&pp=iAQB>

**9.10 Some Graph Algorithms**

**Dijkstra’s Algorithm**

Dijkstra's algorithm is a popular algorithm for finding the shortest path between nodes in a weighted graph. It works for graphs with non-negative edge weight.



**Dijkstra's Algorithm Complexity:**

Time Complexity: O(E Log V)

where, E is the number of edges and V is the number of vertices.

Space Complexity: O(V)

**Resources:**

**YouTube**

**Adel Nasim Data Structure Course**

<https://www.youtube.com/watch?v=rUkj_Wiurdc&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=37&pp=iAQB>

**Arabic Competitive Programming**

<https://www.youtube.com/watch?v=6GzxGabB5MI&t=644s>

**Abdul Bari Algorithms Course**

<https://www.youtube.com/watch?v=XB4MIexjvY0>

**Spanning Tree Channel**

<https://www.youtube.com/watch?v=EFg3u_E6eHU&t=135s>

**Revan Academy**

<https://www.youtube.com/watch?v=M1B-7FChkxs>

**Barngrader**

<https://www.youtube.com/watch?v=5GT5hYzjNoo>

**Google**

**Programiz**

<https://www.programiz.com/dsa/dijkstra-algorithm>

**Implementation Path:** \_

1. **Hash Table**

**10.1 Hash Table Introduction**

**Hash Table:**

The Hash table data structure stores elements in key-value pairs where

- Key- unique integer that is used for indexing the values

- Value - data that are associated with keys.



**Hashing (Hash Function):**

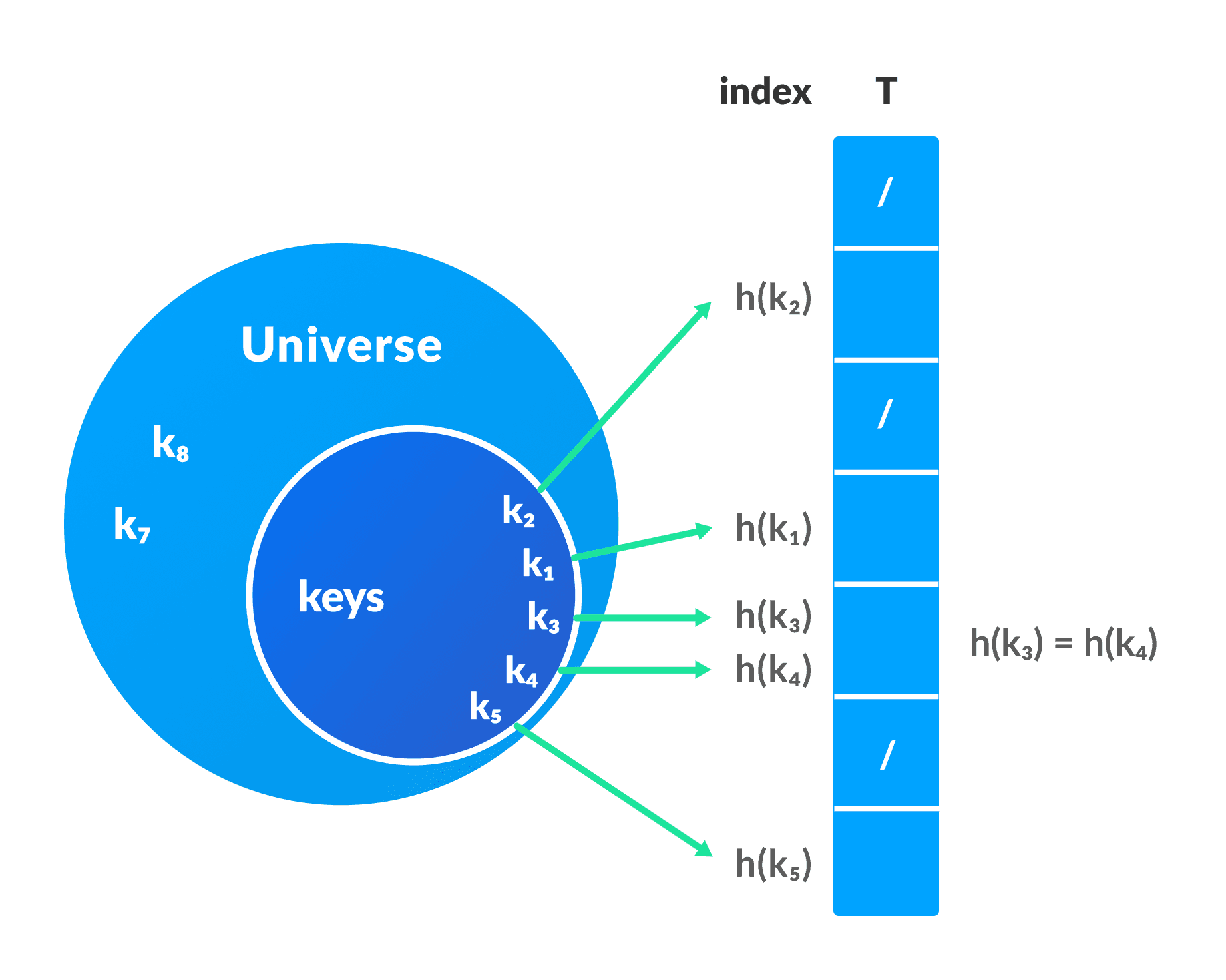
In a hash table, a new index is processed using the keys.

And, the element corresponding to that key is stored in the index.

This process is called hashing.

Let k be a key and h(x) be a hash function.

Here, h(k) will give us a new index to store the element linked with k.



**Hash Collision:**

When the hash function generates the same index for multiple keys,

there will be a conflict (what value to be stored in that index).

This is called a hash collision.

We can resolve the hash collision using one of the following techniques.

- Collision resolution by chaining

- Open Addressing: Linear/Quadratic Probing and Double Hashing

**1. Collision resolution by chaining:**

In chaining, if a hash function produces the same index for multiple elements,

these elements are stored in the same index by using a doubly-linked list.

If j is the slot for multiple elements,

it contains a pointer to the head of the list of elements.

If no element is present, j contains Null.

**2. Open Addressing:**

Unlike chaining, open addressing doesn't store multiple elements into the same slot.

Here, each slot is either filled with a single key or left Null.

Example in open addressing is Linear Probing.

In linear probing, collision is resolved by checking the next slot.

The problem with linear probing is that a cluster of adjacent slots is filled.

When inserting a new element, the entire cluster must be traversed.

This adds to the time required to perform operations on the hash table.

**Good Hash Functions:**

A good hash function may not prevent the collisions completely however it can reduce the number of collisions like h(k) = k % size.

**Applications of Hash Table:**

- constant time lookup and insertion are required.

- cryptographic applications.

- indexing data is required.

**Complexity of Hash Table Chaining and Linear Probing:**

Time Complexity:

* Average Case: Insertion, Deletion, Access: O(1).
* Worst Case: Insertion, Deletion, Access: O(n).

Space Complexity: O(n), where n is the number of elements in the hash table.

**Implementation Path:** code\ src\ HashTable.ts

**10.2 Resources**

**Google**

**Programiz**

<https://www.programiz.com/dsa/hash-table>

**YouTube**

**Adel Nasim**

<https://www.youtube.com/watch?v=mZFuZHOESEY&list=PLCInYL3l2AajqOUW_2SwjWeMwf4vL4RSp&index=33&t=614s&pp=iAQB>

**Arabic Competitive Programming**

<https://www.youtube.com/watch?v=Rp3KxUdV09Y&list=PLPt2dINI2MIZX2EtY81WI-lDkvhKziLKM&index=7&pp=iAQB>

**Abdul Bari**

<https://www.youtube.com/watch?v=mFY0J5W8Udk&list=PLDN4rrl48XKpZkf03iYFl-O29szjTrs_O&index=80&t=9s>

**11 Searching (linear search vs binary search)**

**Complexity:** Linear search has a linear time complexity of O(n) space complexity O(1),

while binary search has a logarithmic time complexity of O(log n) space O(1).

Binary search is more efficient for large lists.

**Sorted Requirement:** Linear search works on both sorted and unsorted lists,

while binary search requires a sorted list.

**Search Efficiency:** Binary search is more efficient as it significantly reduces

the search space with each comparison, while linear search

examines each element sequentially.

**Use Cases:** Linear search is suitable for small lists or unsorted data,

while binary search is effective for large sorted lists.

In summary, linear search is simple and flexible but less efficient,

while binary search is more efficient but requires a sorted list.

The choice between them depends on the specific requirements of the problem

and the characteristics of the data being searched.

**Implementation Path:** code\ src\ Searching.ts

**Resources**

**Google**

**Programiz**

<https://www.programiz.com/dsa/linear-search>

<https://www.programiz.com/dsa/binary-search>

**YouTube**

**Adel Nasim**

<https://www.youtube.com/watch?v=U1wy9kZzOGI&pp=ygUqYWRlbCBuYXNpbSBsaW5lYXIgc2VhcmNoIGFuZCBiaW5hcnkgc2VhcmNo>

<https://www.youtube.com/watch?v=qxhfkoDGGZc&t=24s&pp=ygUqYWRlbCBuYXNpbSBsaW5lYXIgc2VhcmNoIGFuZCBiaW5hcnkgc2VhcmNo>

**Hard-code**

<https://www.youtube.com/watch?v=kleyhFIsU5E>

**Arabic Competitive Programming** \* Advanced

<https://www.youtube.com/watch?v=2G7RzlxTNPo&list=PLPt2dINI2MIZcJ3kADyFAOKOwzuvT-g7P&index=1&pp=iAQB>

<https://www.youtube.com/watch?v=Z-1Z-1utYuI&list=PLPt2dINI2MIZcJ3kADyFAOKOwzuvT-g7P&index=2&pp=iAQB>

<https://www.youtube.com/watch?v=hLXVhRzqq18&list=PLPt2dINI2MIZcJ3kADyFAOKOwzuvT-g7P&index=3&pp=iAQB>

<https://www.youtube.com/watch?v=bXbm6UUzqB8&list=PLPt2dINI2MIZcJ3kADyFAOKOwzuvT-g7P&index=4&pp=iAQB>

<https://www.youtube.com/watch?v=62oWdABsCRc&list=PLPt2dINI2MIZcJ3kADyFAOKOwzuvT-g7P&index=5&pp=iAQB>

**12 Recursion**

Recursion is a programming concept where a function calls itself during its execution. It's a powerful and elegant technique that allows a function to solve a problem by breaking it down into smaller instances of the same problem.

**Key Concepts:**

Base Case: Every recursive function should have a base case, which defines the simplest scenario where the function can return a result without making a recursive call. In the factorial example, the base case is when n is 0 or 1.

Recursive Case: The part of the function where it calls itself with a smaller instance of the problem. In the factorial example, it's the line return n \* factorial(n - 1);.

Recursive Call: The act of a function calling itself.

Unwinding the Recursion: As the function calls itself, a stack of function calls is created. When the base case is reached, the recursion starts to unwind, and each function returns a result until the original call is resolved.

**When to Use Recursion:**

Recursion is often used when a problem can be broken down into smaller, similar sub-problems. It can lead to concise and elegant solutions but should be used with care to avoid stack overflow errors (infinite recursion).

**Implementation Path:** code\ src\ Recursion.ts

**Resources**

**YouTube**

**Hard-code**

<https://www.youtube.com/watch?v=Gqn6hmm9HEw&pp=ygURcmVjdXJzaW9uICDYtNix2K0%3D>

<https://www.youtube.com/watch?v=bb1__43twpE&t=1s&pp=ygURcmVjdXJzaW9uICDYtNix2K0%3D>

**Arabic Competitive Programming** \* Advanced

<https://www.youtube.com/playlist?list=PLPt2dINI2MIYmHYBSEdkdKMf_3nzFMveo>

**13 Sorting**

**13.1 Selection Sort**



**Selection Sort Complexity:**

* Time complexity: O(n^2)
* Space complexity: O(1)

**Implementation Path:** code\ src\ Sorting.ts

**Resources**

**YouTube**

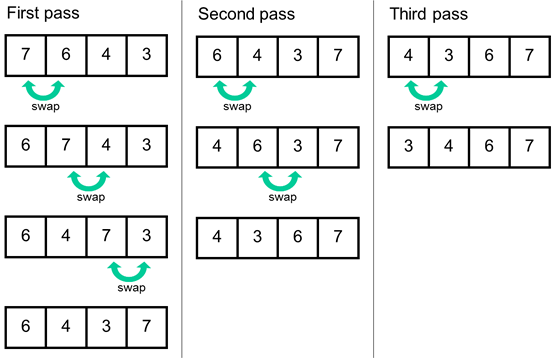
**Hard-code**

<https://www.youtube.com/watch?v=hSAGzbUKz04&pp=ygUYc2VsZWN0aW9uIHNvcnQgSGFyZC1jb2Rl>

**Adel Nasim**

<https://www.youtube.com/watch?v=EnodMqJuQEo&t=11s&pp=ygUuc2VsZWN0aW9uIHNvcnQgYXJhYmljIGNvbXBldGl0aXZlIHByb2dyYW1taW5nIA%3D%3D>

**13.2 Bubble Sort**



**Bubble Sort Complexity:**

* Time complexity: O(n^2)
* Space complexity: O(1)

**Implementation Path:** code\ src\ Sorting.ts

**Resources**

**YouTube**

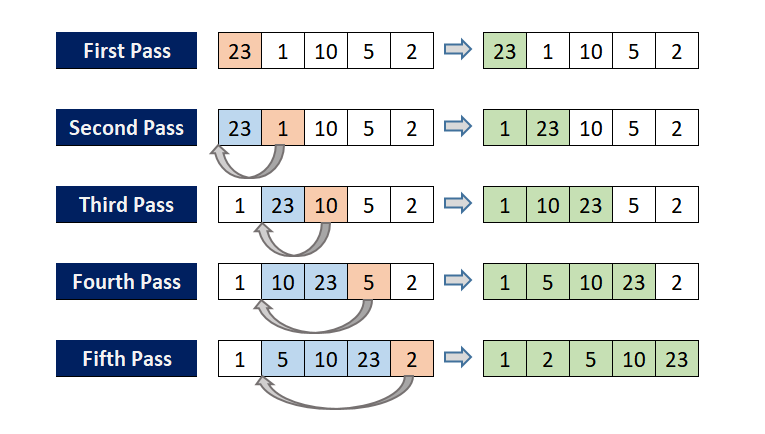
**Hard-code**

<https://www.youtube.com/watch?v=LyWZmJ-d8Os&pp=ygUVQnViYmxlIHNvcnQgSGFyZC1jb2Rl>

**Adel Nasim**

<https://www.youtube.com/watch?v=pIEGHDZHOCk&t=41s&pp=ygUVQnViYmxlIHNvcnQgSGFyZC1jb2Rl>

**13.3 Insertion Sort**



**Insertion Sort Complexity:**

* Time complexity: O(n^2)
* Space complexity: O(1)

**Implementation Path:** code\ src\ Sorting.ts

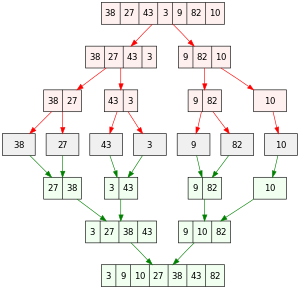
**Resources**

**YouTube**

**Adel Nasim**

<https://www.youtube.com/watch?v=JecAk1FAOck&pp=ygUYSW5zZXJ0aW9uIHNvcnQgSGFyZC1jb2Rl>

**13.4 Merge Sort**



**Merge Sort Complexity:**

* Time complexity: O(n^log(n))
* Space complexity: O(n)

**Implementation Path:** code\ src\ Sorting.ts

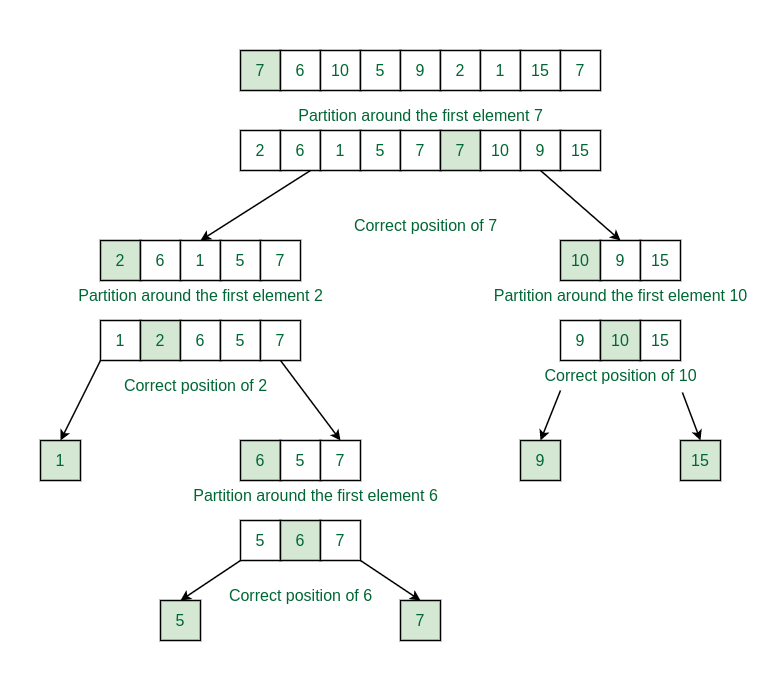
**Resources**

**YouTube**

**Adel Nasim**

<https://www.youtube.com/watch?v=3P43wofKYOM&pp=ygUbbWVyZ2Ugc29ydCBhbGdvcml0aG0g2LTYsdit>

**13.5 Quick Sort**



**Quick Sort Complexity:**

* Time complexity: O(n^log(n))
* Space complexity: O(1)

**Implementation Path:** code\ src\ Sorting.ts

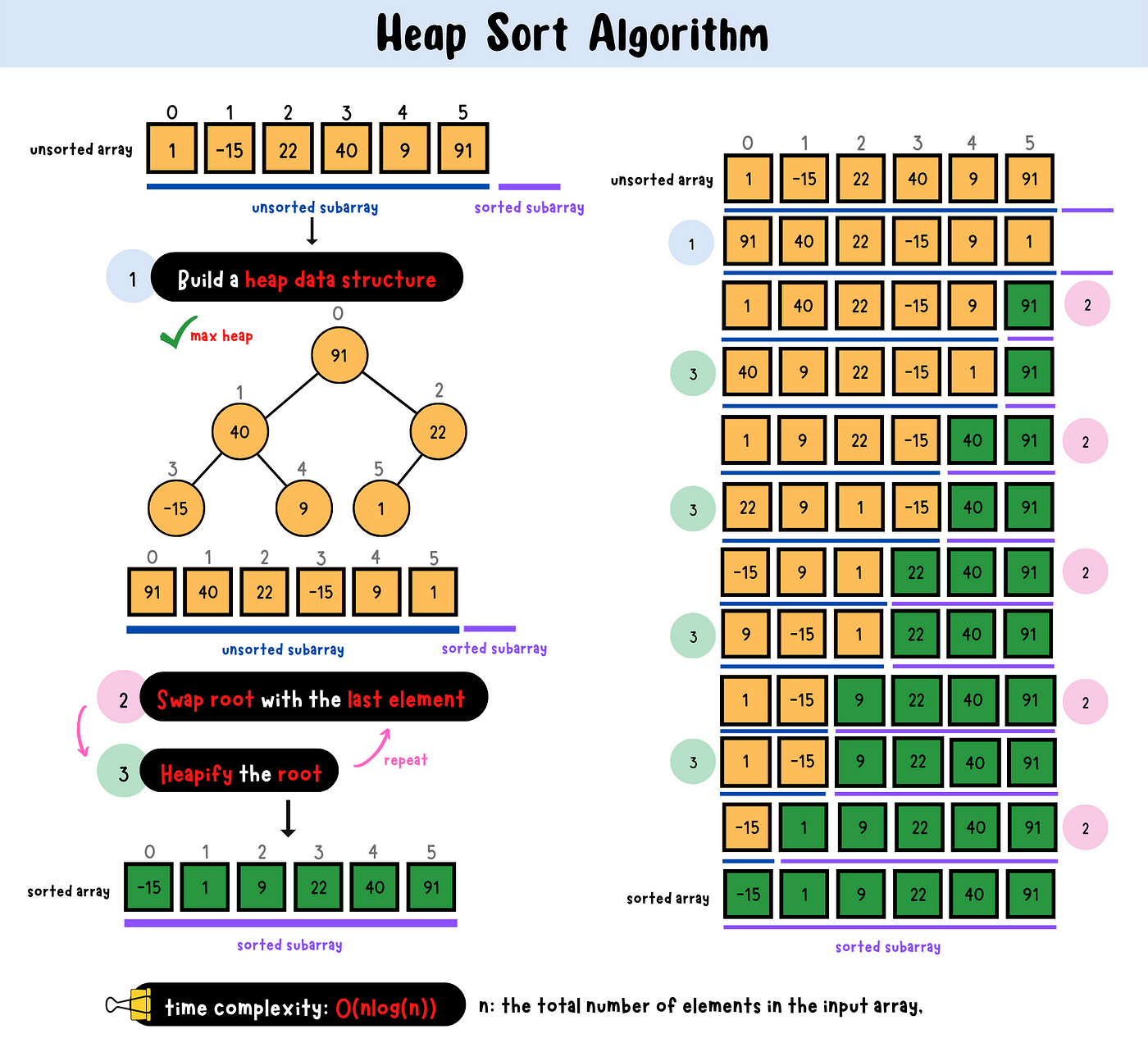
**Resources**

**YouTube**

**Adel Nasim**

<https://www.youtube.com/watch?v=lOB2TIwBiX8&t=105s&pp=ygUbUXVpY2sgc29ydCBhbGdvcml0aG0g2LTYsdit>

**13.6 Heap Sort**



**Heap Sort Complexity:**

* Time complexity: O(n^log(n))
* Space complexity: O(1)

**Implementation Path:** code\ src\ Sorting.ts

**Resources**

**YouTube**

**Adel Nasim**

<https://www.youtube.com/watch?v=REOsj0nYWKE&pp=ygUTaGVhcCBzb3J0IEhhcmQtY29kZQ%3D%3D>