Vijini Mallawaarachchi Follow

Feb 28, 2020 · 11 min read ★ · Listen



# 8 Common Data Structures every Programmer must know

## A quick introduction to 8 commonly used data structures

**Data Structures** are a specialized means of organizing and storing data in computers in such a way that we can perform operations on the stored data more efficiently. Data structures have a wide and diverse scope of usage across the fields of Computer Science and Software Engineering.

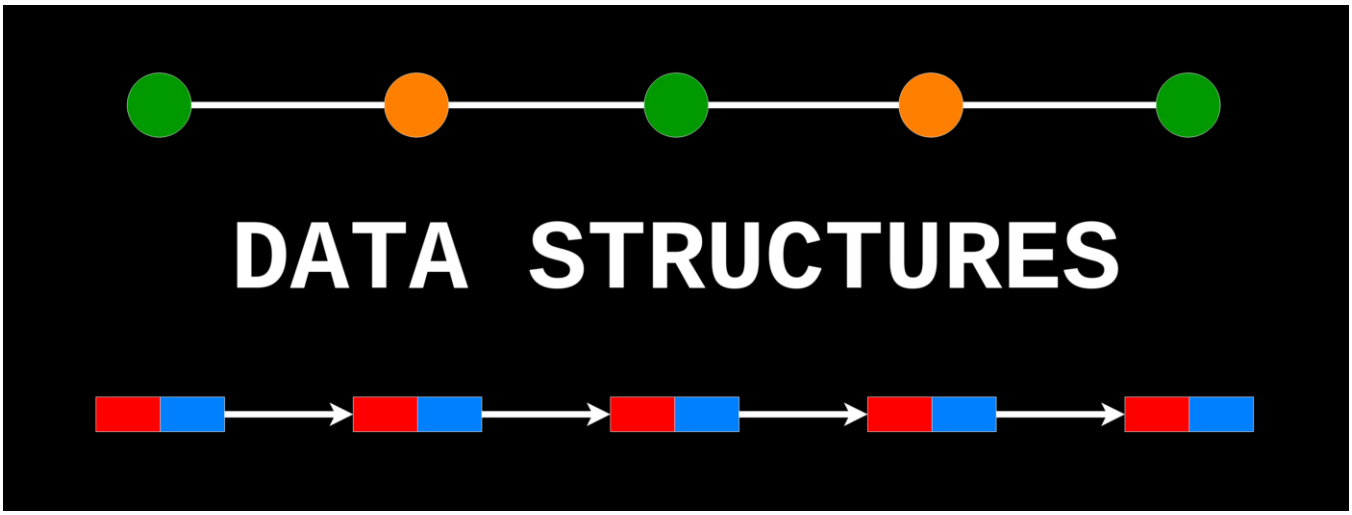



Image by author

Data structures are being used in almost every program or software system that has been developed. Moreover, data structures come under the fundamentals of Computer

In this article, I will be briefly explaining 8 commonly used data structures every programmer must know.

## 1. Arrays

An **array** is a structure of fixed-size, which can hold items of the same data type. It can be an array of integers, an array of floating-point numbers, an array of strings or even an array of arrays (such as *2-dimensional arrays*). Arrays are indexed, meaning that random access is possible.

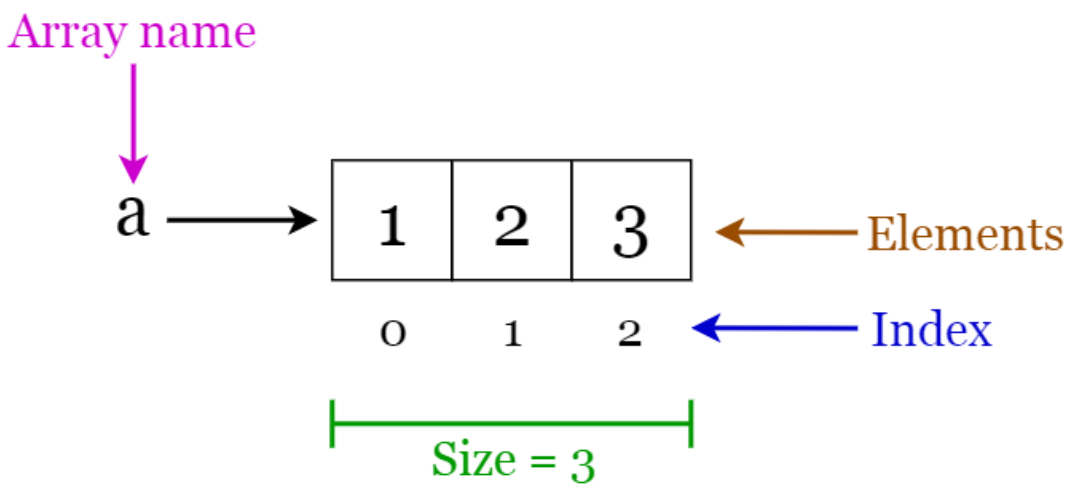



Fig 1. Visualization of basic Terminology of Arrays (Image by author)

#### Array operations

- **Traverse**: Go through the elements and print them.
- **Search**: Search for an element in the array. You can search the element by its value or its index
- **Update**: Update the value of an existing element at a given index

**Inserting** elements to an array and **deleting** elements from an array cannot be done straight away as arrays are fixed in size. If you want to insert an element to an array, first you will have to create a new array with increased size (current size + 1), copy the existing elements and add the new element. The same goes for the deletion with a new array of reduced size.

- Used for different sorting algorithms such as insertion sort, quick sort, bubble sort and merge sort.

## 2. Linked Lists

A **linked list** is a sequential structure that consists of a sequence of items in linear order which are linked to each other. Hence, you have to access data sequentially and random access is not possible. Linked lists provide a simple and flexible representation of dynamic sets.

Let's consider the following terms regarding linked lists. You can get a clear idea by referring to Figure 2.

- Elements in a linked list are known as **nodes**.
- Each node contains a **key** and a pointer to its successor node, known as **next**.
- The attribute named **head** points to the first element of the linked list.
- The last element of the linked list is known as the **tail**.

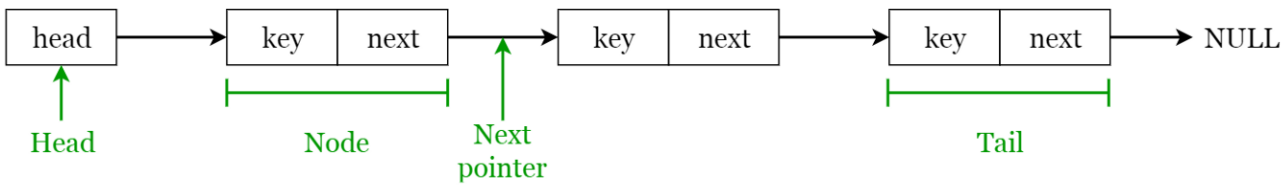



Fig 2. Visualization of basic Terminology of Linked Lists (Image by author)

Following are the various types of linked lists available.

- **Singly linked list** — Traversal of items can be done in the forward direction only.
- **Doubly linked list** — Traversal of items can be done in both forward and backward directions. Nodes consist of an additional pointer known as **prev**, pointing to the previous node.
- **Circular linked lists** — Linked lists where the prev pointer of the head points to the tail and the next pointer of the tail points to the head.

- **Insert**: Insert a key to the linked list. An insertion can be done in 3 different ways; insert at the beginning of the list, insert at the end of the list and insert in the middle of the list.
- **Delete**: Removes an element **x** from a given linked list. You cannot delete a node by a single step. A deletion can be done in 3 different ways; delete from the beginning of the list, delete from the end of the list and delete from the middle of the list.

#### Applications of linked lists

- Used for *symbol table management* in compiler design.
- Used in switching between programs using Alt + Tab (implemented using Circular Linked List).

## 3. Stacks

A **stack** is a **LIFO** (Last In First Out — the element placed at last can be accessed at first) structure which can be commonly found in many programming languages. This structure is named as “stack” because it resembles a real-world stack — a stack of plates.

### Stack operations

Given below are the 2 basic operations that can be performed on a stack. Please refer to Figure 3 to get a better understanding of the stack operations.

- **Push**: Insert an element on to the top of the stack.
- **Pop**: Delete the topmost element and return it.

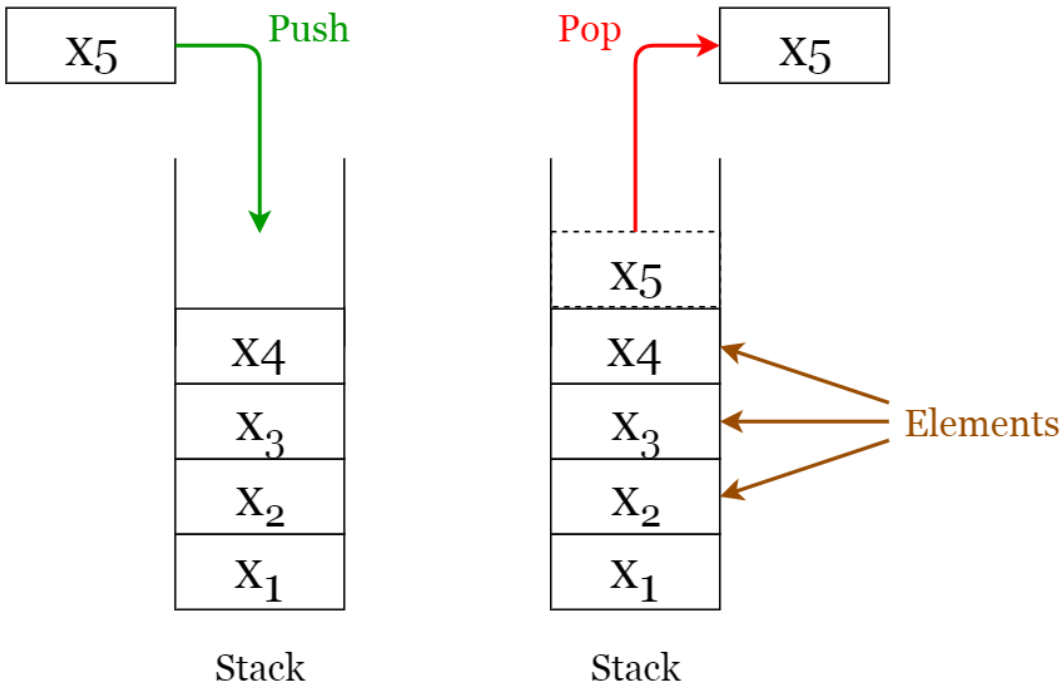



Fig 3. Visualization of basic Operations of Stacks (Image by author)

Furthermore, the following additional functions are provided for a stack in order to check its status.

- **Peek**: Return the top element of the stack without deleting it.
- **isEmpty**: Check if the stack is empty.
- **isFull**: Check if the stack is full.

### Applications of stacks

- Used for expression evaluation (e.g.: *shunting-yard algorithm* for parsing and evaluating mathematical expressions).




A **queue** is a **FIFO** (First In First Out — the element placed at first can be accessed at first) structure which can be commonly found in many programming languages. This structure is named as “queue” because it resembles a real-world queue — people waiting in a queue.

Image by Sabine Felidae from Pixabay

#### Queue operations

Given below are the 2 basic operations that can be performed on a queue. Please refer to Figure 4 to get a better understanding of the queue operations.

- **Enqueue**: Insert an element to the end of the queue.
- **Dequeue**: Delete the element from the beginning of the queue.

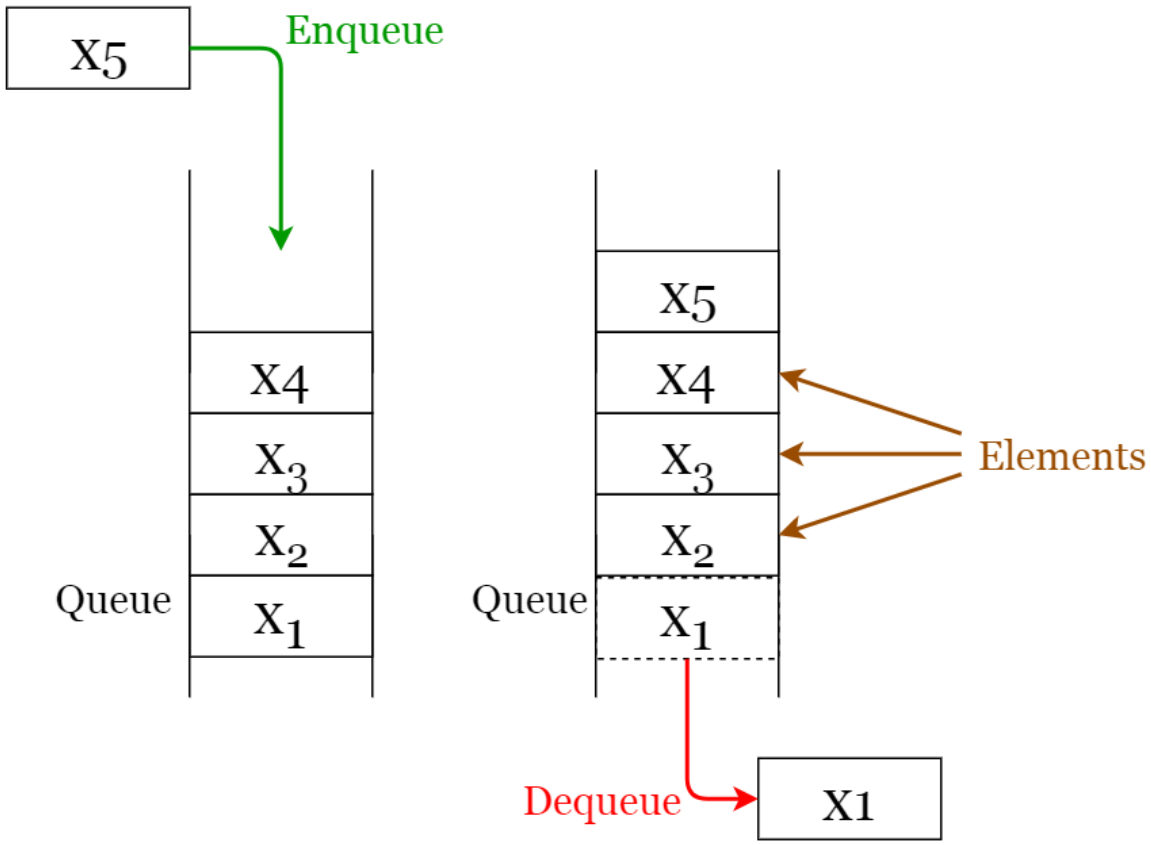



Fig 4. Visualization of Basic Operations of Queues (Image by author)

**Applications of queues**

- Used to manage threads in multithreading.
- Used to implement queuing systems (e.g.: priority queues).

## 5. Hash Tables

A **Hash Table** is a data structure that stores values which have keys associated with each of them. Furthermore, it supports lookup efficiently if we know the key associated with the value. Hence it is very efficient in inserting and searching, irrespective of the size of the data.

**Direct Addressing** uses the one-to-one mapping between the values and keys when storing in a table. However, there is a problem with this approach when there is a large number of key-value pairs. The table will be huge with so many records and may be impractical or even impossible to be stored, given the memory available on a typical computer. To avoid this issue we use **hash tables**.




A special function named as the **hash function** (**h**) is used to overcome the aforementioned problem in direct addressing.

In direct accessing, a value with key **k** is stored in the slot **k**. Using the hash function, we calculate the index of the table (slot) to which each value goes. The value calculated using the hash function for a given key is called the **hash value** which indicates the index of the table to which the value is mapped.

$$h(k) = k \% m$$

- **h:** Hash function
- **k:** Key of which the hash value should be determined
- **m:** Size of the hash table (number of slots available). A prime value that is not close to an exact power of 2 is a good choice for **m**.

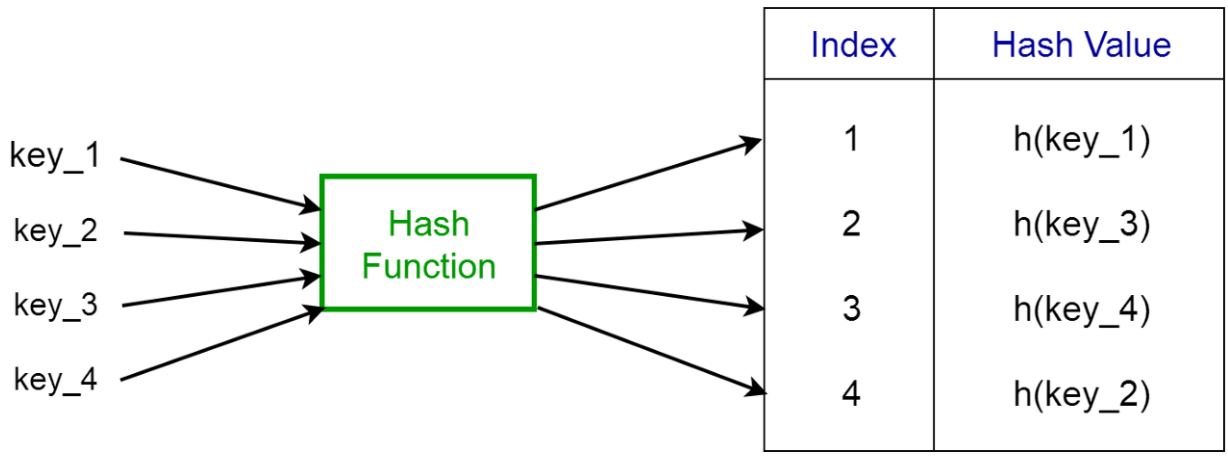



Fig 5. Representation of a Hash Function (Image by author)

Consider the hash function **h(k) = k % 20**, where the size of the hash table is 20. Given a set of keys, we want to calculate the hash value of each to determine the index where it should go in the hash table. Consider we have the following keys, the hash and the hash table index.

- 1 → 1%20 → 1

- 63 → 63%20 → 3

From the last two examples given above, we can see that **collision** can arise when the hash function generates the same index for more than one key. We can resolve collisions by selecting a suitable hash function h and use techniques such as **chaining** and **open addressing**.

#### Applications of hash tables

- Used to implement database indexes.
- Used to implement associative arrays.
- Used to implement the “set” data structure.

## 6. Trees

A **tree** is a hierarchical structure where data is organized hierarchically and are linked together. This structure is different from a linked list whereas, in a linked list, items are linked in a linear order.

Various types of trees have been developed throughout the past decades, in order to suit certain applications and meet certain constraints. Some examples are binary search tree, B tree, treap, red-black tree, splay tree, AVL tree and n-ary tree.

#### Binary Search Trees

A **binary search tree (BST)**, as the name suggests, is a binary tree where data is organized in a hierarchical structure. This data structure stores values in sorted order.

Every node in a binary search tree comprises the following attributes.

1. **key**: The value stored in the node.
2. **left**: The pointer to the left child.
3. **right**: The pointer to the right child.
4. **p**: The pointer to the parent node.




- If **y** is a node in the **left** subtree of x, then **y.key ≤ x.key**
- If **y** is a node in the **right** subtree of x, then **y.key ≥ x.key**

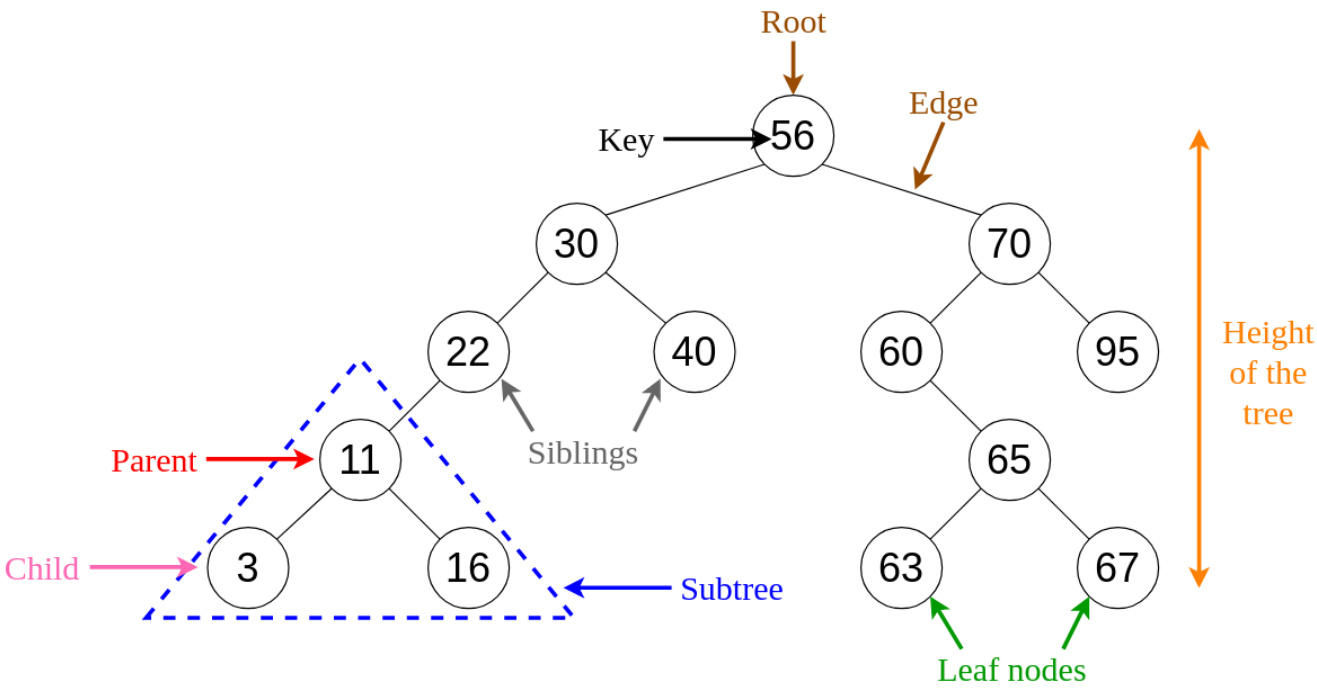



Fig 6. Visualization of Basic Terminology of Trees (Image by author)

**Applications of trees**

- **Binary Trees**: Used to implement expression parsers and expression solvers.
- **Binary Search Tree**: used in many search applications where data are constantly entering and leaving.
- **Heaps**: used by JVM (Java Virtual Machine) to store Java objects.
- **Treaps**: used in wireless networking.

Check my articles below on 8 useful tree data structures and self-balancing binary search trees.

**8 Useful Tree Data Structures Worth Knowing**
An overview of 8 different tree data structures
towardsdatascience.com

**Self-Balancing Binary Search Trees 101**

Introduction to Self-Balancing Binary Search Trees

towardsdatascience.com

## 7. Heaps

A **Heap** is a special case of a binary tree where the parent nodes are compared to their children with their values and are arranged accordingly.

Let us see how we can represent heaps. Heaps can be represented using trees as well as arrays. Figures 7 and 8 show how we can represent a binary heap using a binary tree and an array.

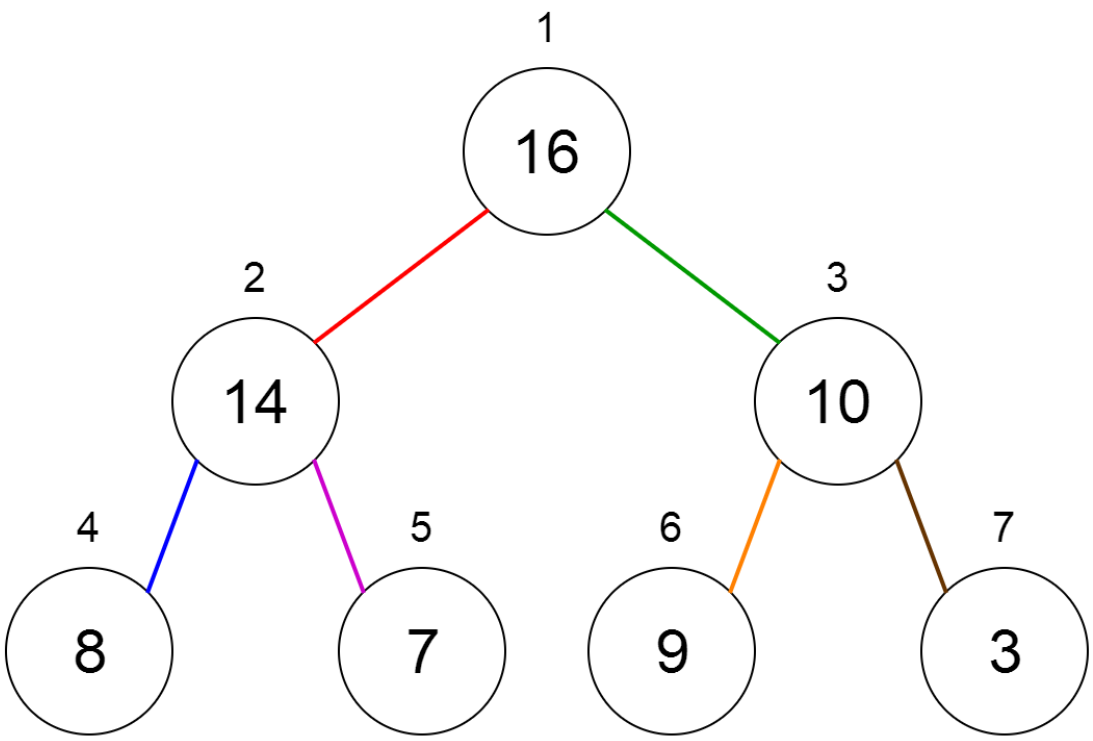



Fig 7. Binary Tree Representation of a Heap (Image by author)

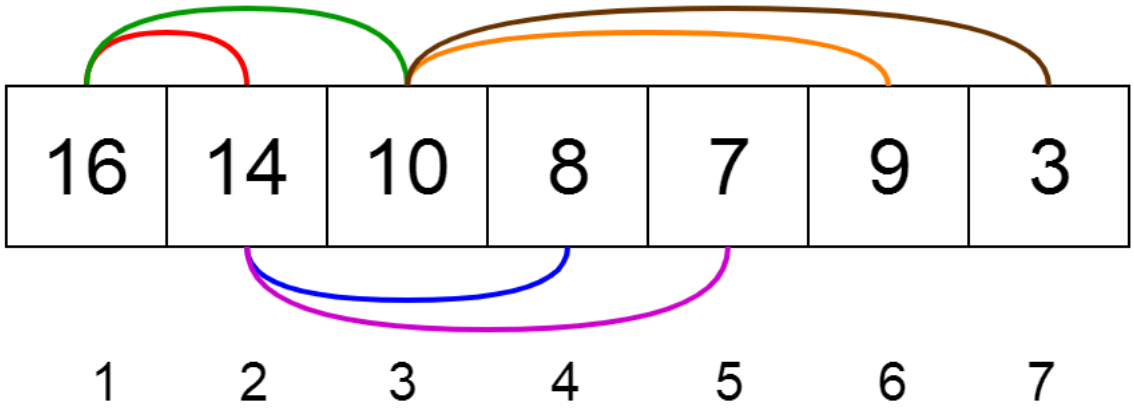



Fig 8. Array Representation of a Heap (Image by author)

Heaps can be of 2 types.

1. **Min Heap** — the key of the parent is less than or equal to those of its children. This is called the **min-heap property**. The root will contain the minimum value of the heap.
2. **Max Heap** — the key of the parent is greater than or equal to those of its children. This is called the **max-heap property**. The root will contain the maximum value of the heap.

#### Applications of heaps

- Used in **heapsort algorithm**.
- Used to implement priority queues as the priority values can be ordered according to the heap property where the heap can be implemented using an array.
- Queue functions can be implemented using heaps within **O(log n)** time.
- Used to find the $k^{th}$ smallest (or largest) value in a given array.

Check my article below on implementing a heap using the python heapq module.

**Introduction to Python Heapq Module**

## 8. Graphs

A **graph** consists of a finite set of **vertices** or nodes and a set of **edges** connecting these vertices.

The **order** of a graph is the number of vertices in the graph. The **size** of a graph is the number of edges in the graph.

Two nodes are said to be **adjacent** if they are connected to each other by the same edge.

### Directed Graphs

A graph **G** is said to be a **directed graph** if all its edges have a direction indicating what is the start vertex and what is the end vertex.

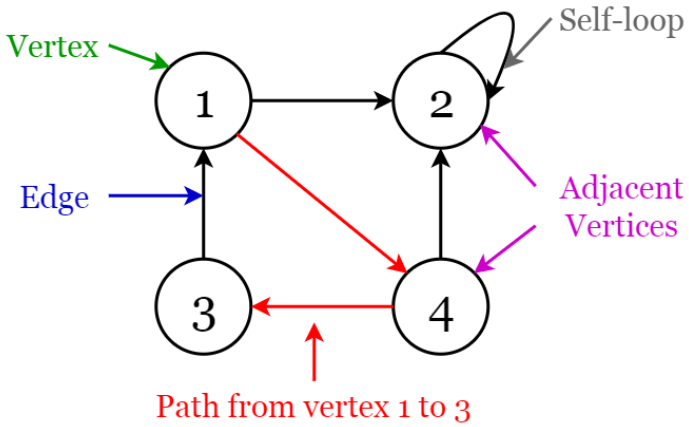
We say that **(u, v)** is **incident from** or **leaves** vertex **u** and is **incident to** or **enters** vertex **v**.

**Self-loops**: Edges from a vertex to itself.

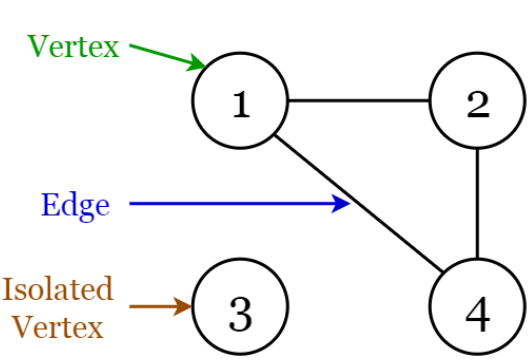
### Undirected Graphs

A graph **G** is said to be an **undirected graph** if all its edges have no direction. It can go in both ways between the two vertices.

If a vertex is not connected to any other node in the graph, it is said to be **isolated**.




**Directed Graph**




**Undirected Graph**

You can read more about graph algorithms from my article 10 Graph Algorithms Visually Explained.

**10 Graph Algorithms Visually Explained**

A quick introduction to 10 basic graph algorithms with examples and visualisations

medium.com

**Applications of graphs**

- Used to represent social media networks. Each user is a vertex, and when users connect they create an edge.
- Used to represent web pages and links by search engines. Web pages on the internet are linked to each other by hyperlinks. Each page is a vertex and the hyperlink between two pages is an edge. Used for Page Ranking in Google.
- Used to represent locations and routes in GPS. Locations are vertices and the routes connecting locations are edges. Used to calculate the shortest route between two locations.

## Final Thoughts

A cheat sheet for the time complexities of the data structure operations can be found in this link. Moreover, check out my article below where I have implemented a few common data structures from scratch using C++.

**Data Structures in C++ — Part 1**

Implementing common data structures in C++

towardsdatascience.com

Finally, I would like to thank Mr. A Alkaff Ahamed for providing valuable feedback and




Thanks a lot for reading. 😊

Cheers! 😃

## References

[1] Introduction to Algorithms, Third Edition By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein.

[2] List of data structures from Wikipedia (https://en.wikipedia.org/wiki/List_of_data_structures)