MET CS688

WEB ANALYTICS AND MINING

ZLATKO VASILKOSKI

BASIC DATA STRUCTURES

Computation

Thermodynamics and Information Theory

- Work is a **resource** in thermodynamics, allowing us to perform tasks like charging batteries or powering cars.
- Information theory views information as a **resource**, which can be used for storing data and solving computational problems.

The connection between thermodynamics and information theory lies in the concept of **resources**.

  - Both require resources (work or information).

Szilard's Engine - A Connection between Information and Thermodynamics

  - Leo Szilard proposed an engine that uses information to generate work.
  - The engine involves a box with a gas particle, which stores one bit of information (the side it is in).
  - By using this information, the system can perform thermodynamic work.

The process of **erasing information** (e.g., rewriting a calculation) requires thermodynamic work.

  - There is a fundamental limit on the amount of work required to erase information.
  - Physicist Rolf Landauer is credited with pointing out that erasing information requires energy, proposing what is now known as "Landauer's principle" in 1961; which states that any logically irreversible operation, like deleting a bit of information, must dissipate heat, meaning it requires energy to erase data.
  - Charlie Bennett, at IBM, built on the work of IBM's Rolf Landauer to show that ordinary computations (in principle) can be performed at zero-energy cost by a logically and thermodynamically reversible apparatus.

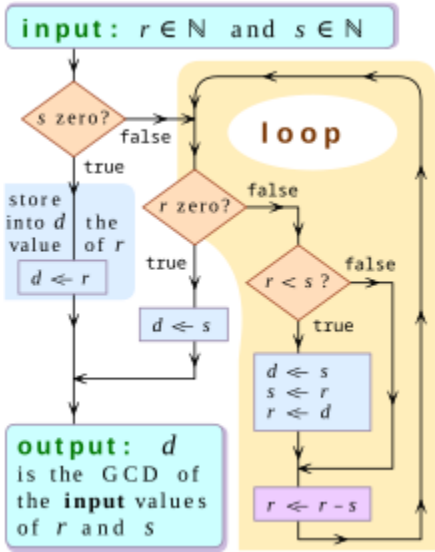
What is an algorithm?

An algorithm is a finite set of step-by-step (written in order) instructions to solve a specific problem or to get the desired output. Named after **Al-Khwarizmi**, 8th century Persian originator of algebra, whose name (romanized variously as Algorithm, Alghoarism, Algorism, etc.) is the etymological origin of algorithm in its modern sense (Source: Wikipedia).

- Algorithms are generally developed independently of the underlying language, meaning you can use an algorithm in more than one programming language.

- Data structures are used to store and organize data. Algorithms are used to solve problems by using that data.

- What makes a good algorithm:
  - The input and output of an algorithm must be defined precisely.
  - In the algorithm, each step must be clear and univocal.
  - An algorithm must be the most efficient method to solve a problem over other ways.
  - An algorithm should be written in such a way that it can be used for other languages.
  - An algorithm is not a complete program or code, it is just the solution or core logic of a problem that can be expressed using a flowchart or as pseudocode (plain language description with steps).



How to Evaluate an Algorithm?

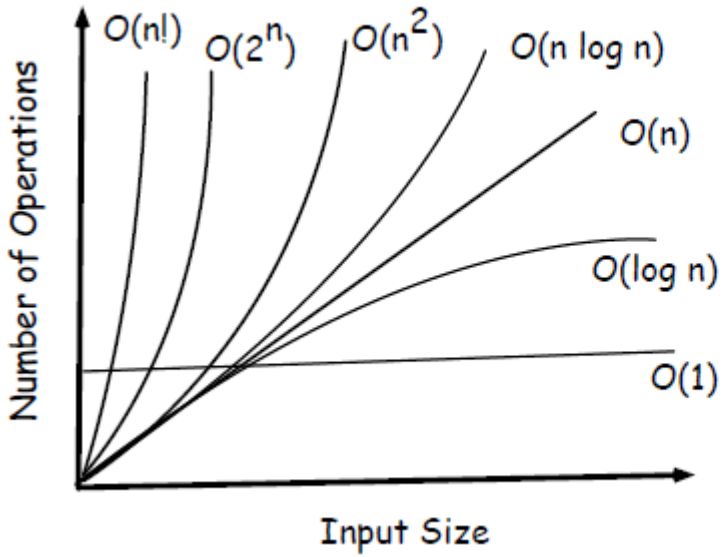
1. Efficiency and Computational Complexity (Space, Time). Along efficiency it includes resource utilization including response time, memory use, … as well

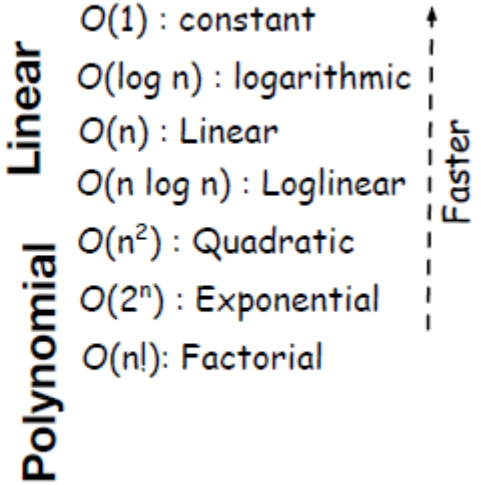
2. Accuracy

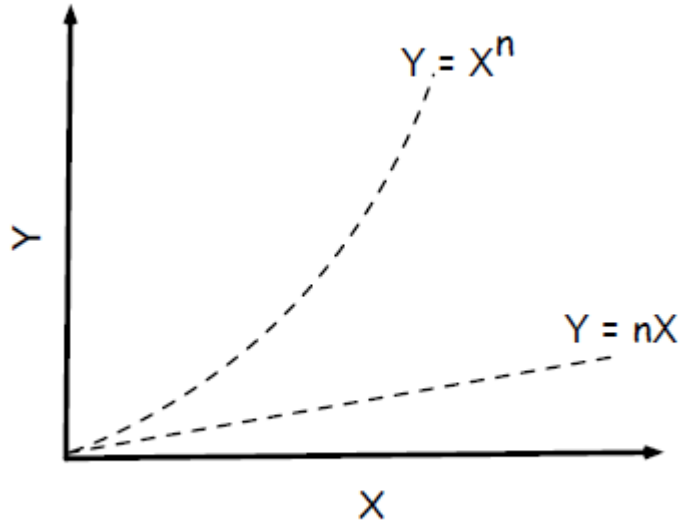
3. Algorithm Efficiency

Big O complexity

- Big O (O()) describes the upper bound of the complexity.
- Omega (Ω()) describes the lower bound of the complexity.
- Theta (Θ()) describes the exact bound of the complexity.
- Little O (o()) describes the upper bound excluding the exact bound.



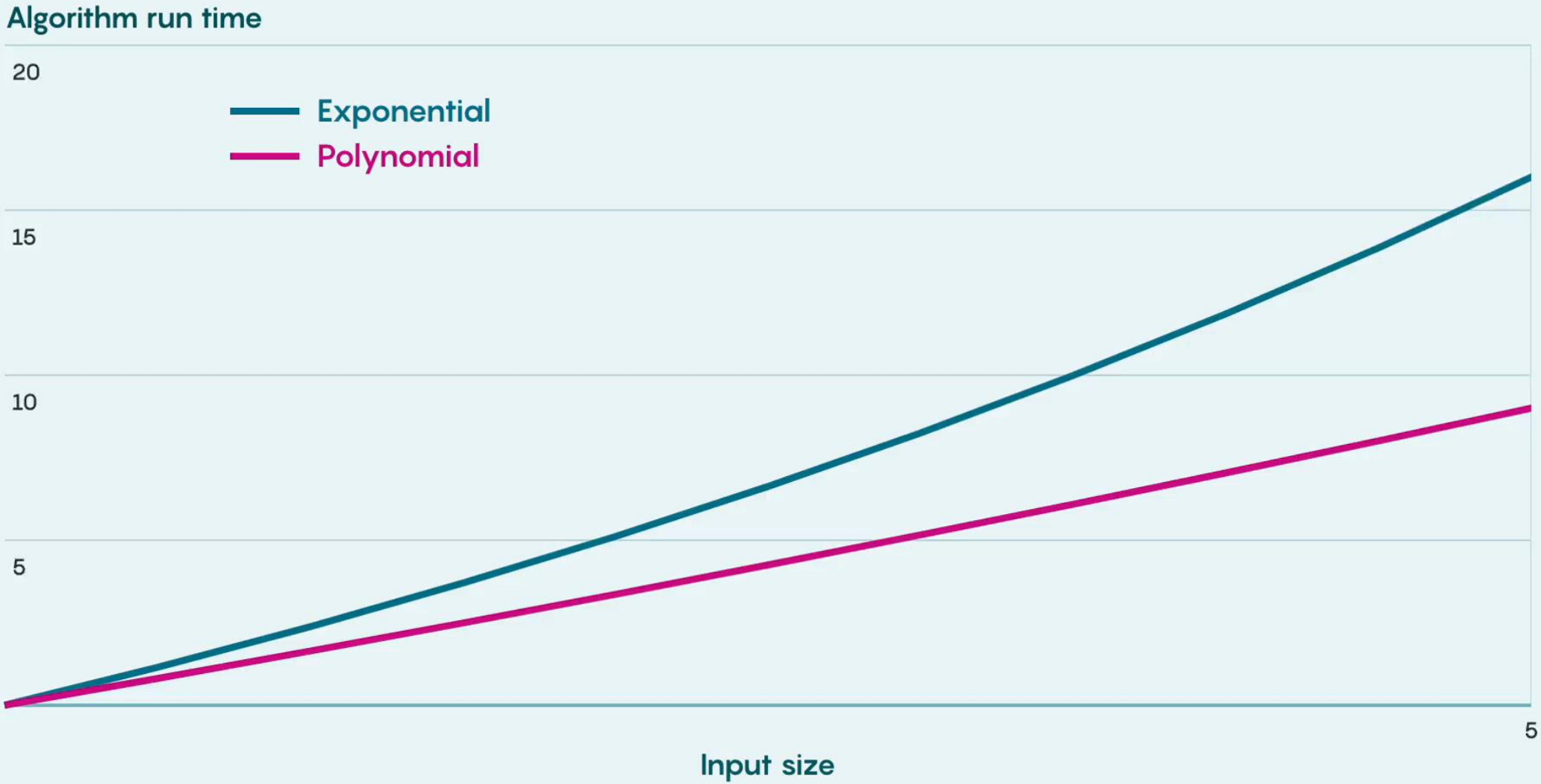




Linear and exponential growth example.

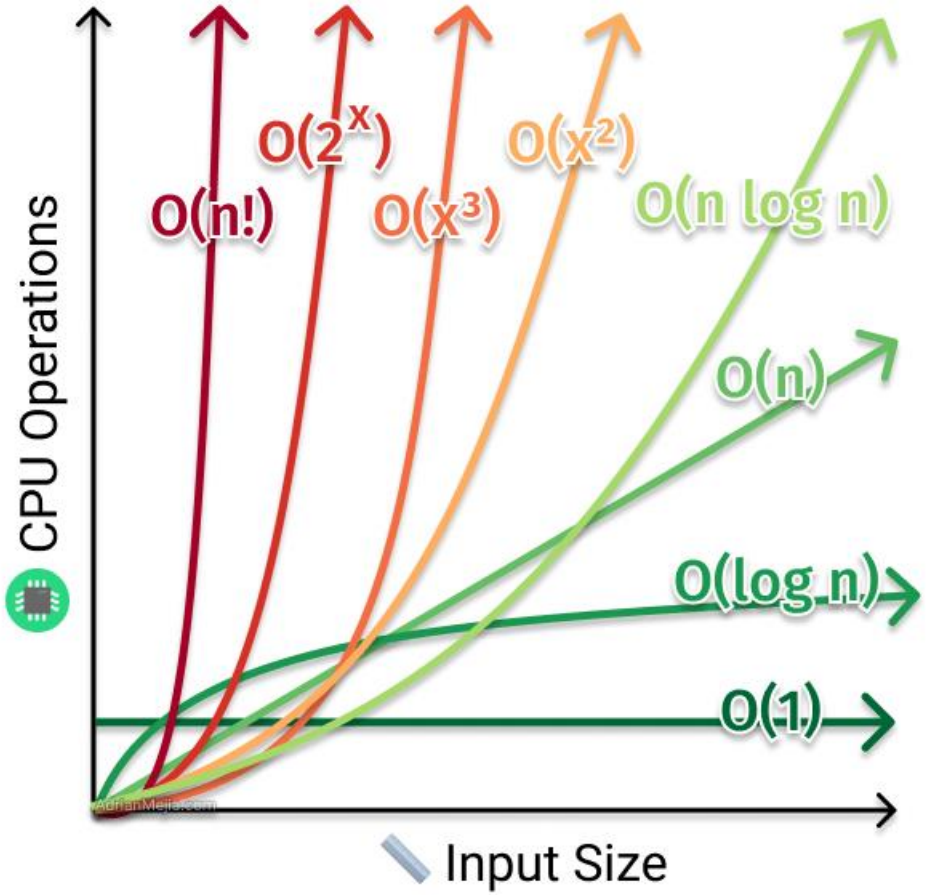
Exponential and Polynomial Algorithms

Every computational problem takes longer to solve as its input grows larger, but not all problems grow harder at the same rate.



How to determine the complexity of an Algorithm?

Count the number of operations that the algorithm requires and compare that to the size of the input.



An easy experimental way to determine the complexity of an algorithm is to run it and time it for input sizes N such as

1. N=1,000,
2. N=10,000,
3. N= 100,000
4. N= 1 million

1. A running time of 3, 4, 5, 6 seconds (or some multiple) indicates time complexity of O(log N).
   - This will correspond to **traversing a tree**, **binary search**, partitioning the space such as in **sorting** algorithms.
2. A running time of 1, 10, 100, 1000 seconds indicates time complexity of O(N).
   - This will correspond to a **single** loop
3. A running time of 3, 40, 500, 6000 seconds (or some multiple) indicates time complexity of O(N log N).
   - Code containing conditional (if/else) Statements
4. A running time of 1, 100, 10,000, million seconds indicates time complexity of $O(N^2)$.
   - This will correspond to **double** loop

How to determine the complexity of an Algorithm?

Source: https://www.bigocheatsheet.com/

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

How to determine the complexity of an Algorithm?

Source: https://www.bigocheatsheet.com/

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

A

A

Object Oriented Programing

1) **Abstraction and Encapsulation** - allow you to create a simple and consistent interface for your data structures and algorithms, while hiding the complexity and implementation details from the users.

   - **Abstraction** is the process of hiding the unnecessary details and exposing only the essential features of an object.

   - **Encapsulation** is the mechanism of bundling the data and the methods that operate on it within the same object.

   - **Example:** Create a Stack class that abstracts the concept of a last-in, first-out (LIFO) data structure, and encapsulates the underlying array or linked list, as well as the methods to push, pop, and peek elements.

Object Oriented Programing

2) Inheritance and Polymorphism - create a hierarchy of classes that share common features and functionality, but also have their own specificities and variations.

- **Inheritance** is the process of creating new classes from existing ones.
- **Polymorphism** is the ability of an object to take different forms depending on the context.
- **Example:** Create a Sort class that defines a generic algorithm for sorting an array, and then create subclasses that implement different sorting strategies, such as BubbleSort, MergeSort, and QuickSort. Then, you can use polymorphism to invoke the appropriate sort method based on the type of the object.

Object Oriented Programing

3) **Composition and Aggregation** - ways of creating complex objects from simpler ones, by establishing a has-a relationship between them.

- **Composition** implies a strong and exclusive ownership of the component objects by the composite object.
- **Aggregation** implies a weak and shared ownership.
- **Example:** Create a Graph class that aggregates a set of Node objects and a set of Edge objects, and then use composition to create a WeightedGraph class that inherits from Graph and adds a weight attribute to each edge.

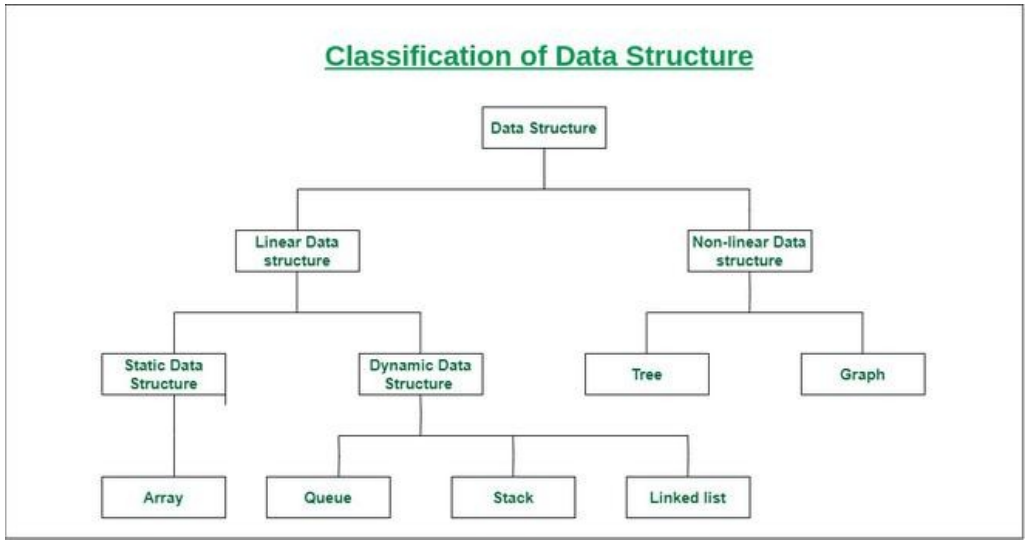
Object Oriented Programing

4) **Design Patterns** - There are various types and categories of design patterns.

- Creational patterns - most useful ones include Factory, Singleton, and Builder.
- Structural patterns - Adapter, Decorator, and Composite.
- Behavioral patterns - Strategy, Observer, and Iterator.

What is a data structure?

- A data structure is a specific method to store and organize data in a computer to be easily used to efficiently perform operations on it.
- It is estimated that 80% of the world's data is unstructured (not suitable for the analysis).
- This difference in the data storage scheme decides which data structure would be more suitable for a given situation.
- Most common types of data structure:



Linear Data Structures

1. Array
2. Linked List
3. Stack
4. Queue

Non-Linear Data Structures

5. Hash tables
6. Trees
7. Heaps

Advanced Data Structures

8. Graph
9. Trie

Data Structure Complexities

Big O complexity

- Big O (O()) describes the upper bound of the complexity.
- Omega (Ω()) describes the lower bound of the complexity.
- Theta (Θ()) describes the exact bound of the complexity.
- Little O (o()) describes the upper bound excluding the exact bound.

M maximum string length, N is number of encoded words (keys).

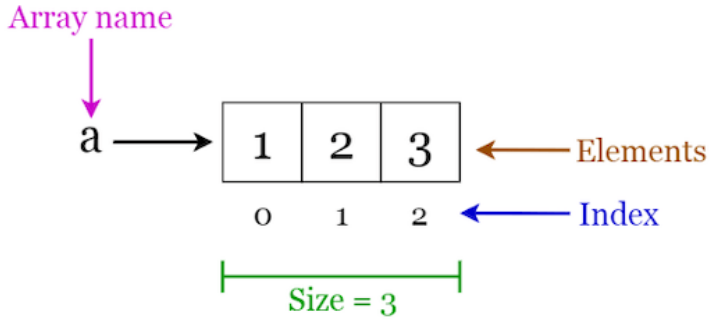
Data Structure	Time Complexity				Space Complexity
	Access	Search	Insert	Delete	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(n)
Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)
Hash table	N/A	Θ(1)	Θ(1)	Θ(1)	O(n)
Tree (text)		$O(Mlog_2(n))$			O(n)
Heap	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)
Graph		O(V+E)			O(V)
Trie		O(M)			O(n)

1) Linear Data Structures – Arrays

Arrays – store elements in contiguous memory locations, resulting in easily calculable addresses for the elements stored and this allows faster access to an element at a specific index. It is a fixed-size structure that stores multiple items of the same kind of data sequentially.

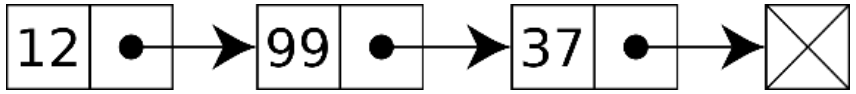
Time Complexity				Space Complexity
Access	Search	Insert	Delete	
Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(n)

- An array contains fixed-size number of elements (also known as variables).
- That's why it is hard to change the array's size.
- To add an element, you need to create a new array with increased size (+1), then copy the existing elements and add a new element to it.
- An array has elements inside each container and lined up small containers in a sequence.
- Used as a structure to build more complicated data structures and for sorting algorithms.



2) Linear Data Structures – Linked Lists

Linked List – Element's order is not given by their physical placement in memory. Each element points to the next. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked (connected) using pointers.

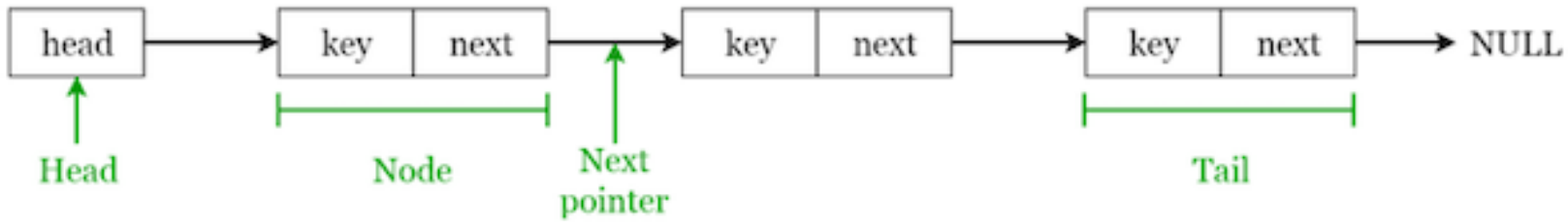


Time Complexity				Space Complexity
Access	Search	Insert	Delete	
Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)

- Linked List is not of fixed-size number of elements.
- The elements can be accessed only in sequential order, not in arbitrary order.

2) Linear Data Structures – Linked Lists

- The first element in the list is known as “Head,” and the last item is known as “Tail.”
- Each element is called a “node,” and it contains a pointer and a key.
  - The Pointer takes you to the next Node known as “next.”
- A single linked list traverses each item from head to tail (in forward direction).
- A double-linked list can traverse in both directions.
- Used in web browsers to keep track of web pages visited and moves in multi-player board games.



2) Linear Data Structures – Linked Lists

To find length of a Linked List you need to iterate recursively

Linked list nodes contain two fields:

1) Key or Data (stores integer, strings or any type of data).

2) Pointer connects one node to the next node. The last node is linked to a terminator used to signify the end of the list.

In C, we can represent a node using structures.

In Python, Java or C#, Linked List can be represented as a class.

2) Linear Data Structures – Linked Lists

Why Using Linked List?

Arrays can be used to store linear data of similar types, but arrays have the following limitations.

1. Memory is linked to the number of elements in arrays while Linked Lists have dynamic size.
   - The size of the arrays is fixed: So, we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
2. Adding/Deleting new element in an array is expensive
   - Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted but in Linked list if we have the head node then we can traverse to any node through it and insert new node at the required position.

Drawbacks of Linked Lists:

- *No random access*. Only access elements sequentially starting from the first node (head node). As a result, the default implementation of the binary search is not efficient.
- *Extra memory space* for a pointer is required with each element of the list.
- Arrays have better *cache locality* compared to linked lists since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

2) Linear Data Structures – Linked Lists

Adding a new node to a Linked List:

- Adding a node at the front is a 4 steps process.
    1. & 2. Allocate the Node & Put in the data
    3. Make new Node as head
    4. Move the head to point to new Node
- Adding a node after a given node is a 5 steps process
    1. Check if the given previous node exists
    2. & 3. Create new node and put in the data
    4. Make next of new Node as next of previous node
    5. make next of previous node as new Node
- Adding a node at the end is a 6 steps process in which we have to traverse the list till the end and then change the next to last node to a new node.

2) Linear Data Structures – Linked Lists

Circular Linked List – Is useful for implementation of queue. Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

Doubly Linked List – Contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list.

Advantages over singly linked list

1) It can be traversed in both forward and backward direction.

2) The delete operation is more efficient if pointer to the node to be deleted is given.

- In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

3) A new node can be quickly inserted before a given node.

Disadvantages over singly linked list

1) Every node requires extra space for a previous pointer.

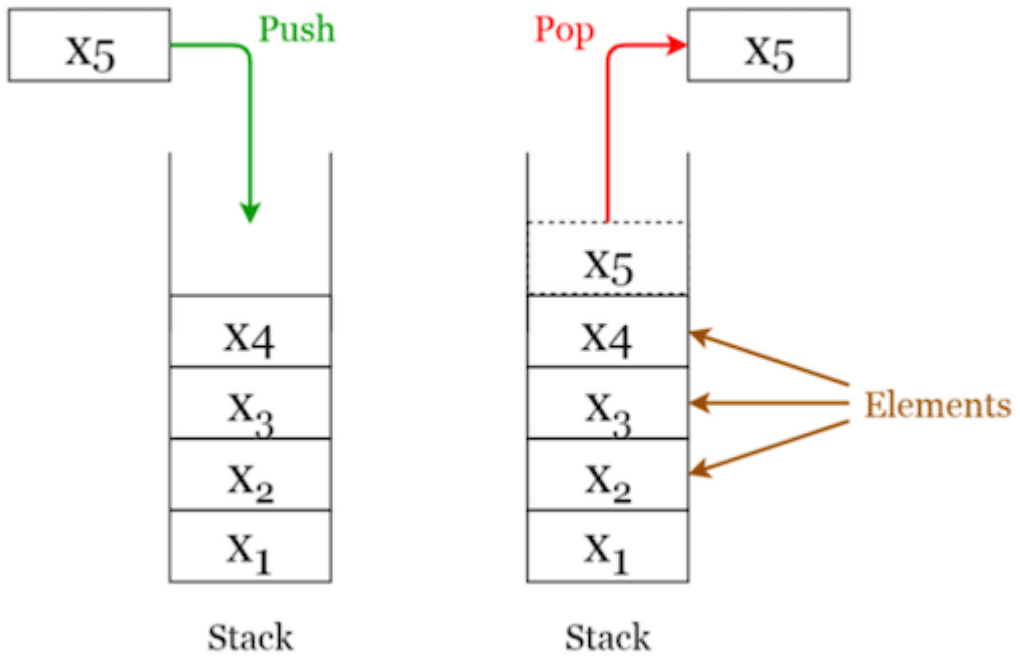
2) All operations require an extra pointer (for previous) to be maintained.

3) Linear Data Structures – Stacks

Stacks – a linear order structure. It works in a LIFO (Last In First Out) order.

Time Complexity				Space Complexity
Access	Search	Insert	Delete	
Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)

- LIFO means the last-placed element can be accessed first.
- You can push (add a new) or pop (delete) an element on/from the top of the elements, like a stack of plates in the real world.
- Stacks are mostly used in recursion programming to implement function calls and mathematical expressions for parsing and evaluations.



3) Linear Data Structures – Stacks

Stacks – is a container of objects that are inserted and removed according to the **last-in first-out** (**LIFO**) or **first-in last-out** (**FILO**) principle. For example, stack of books.

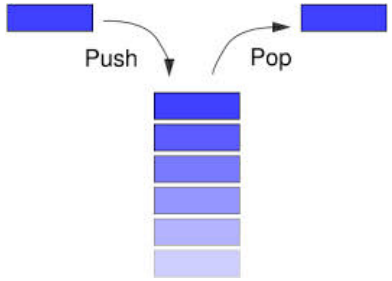
Illustration of a Stack: plates stacked on top of each other.

- The plate which is at the top is the first one to be removed, i.e., the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow the LIFO/FILO order.

Basic operations performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns the top element of the stack.
- **isEmpty:** Returns true if the stack is empty, else false.

Illustration of the operations push and **pop**



3) Linear Data Structures – Stacks

Time Complexities of operations on stack:

- Constant - all take O(1) time (no need to run any loop in any of these operations.).

Implementation: There are two ways to implement a stack

1. Using array
2. Using linked list

Space Complexity: O(N) – where N is the size of array for storing elements.

Applications of a Stack:

- In Memory management, any modern computer uses a stack as the primary management for a running purpose. Each program that is running in a computer system has its own memory allocations.
- Redo-undo features in many applications.
- Forward and backward feature in web browsers.
- Used in many algorithms like Tower of Hanoi, tree traversals, etc.

3) Linear Data Structures – Stacks

Applications of a Stack - Example: Check for Balanced Brackets in an expression (i.e. “{“, “}”, “(“, “)”, “[“, “]”).

- Initialize a character stack S.
- Traverse (loop) the input expression string
  - If the current character is a **starting** bracket (‘**(**‘ or ‘**{**‘ or ‘**[**‘) then push it to stack.
  - If the current character is a **closing** bracket (‘**)**’ or ‘**}**’ or ‘**]**’) then pop from stack.
    - If the top of stack (popped character) is the matching starting bracket, then the brackets are **not balanced**!
  - After complete traversal, if there is some starting bracket left in stack then the input expression of brackets is “**not balanced**”.

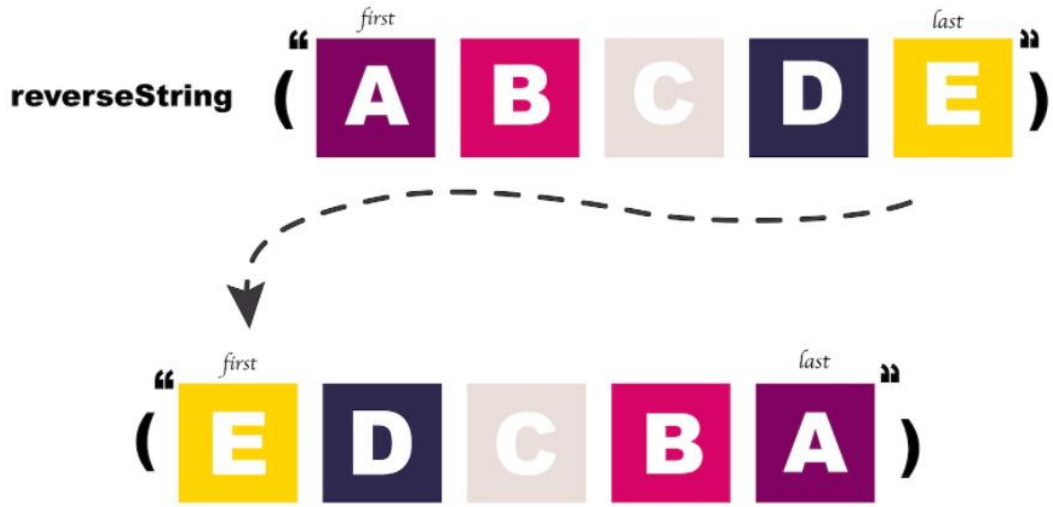
There are 5 different situations when the brackets are **not balanced**.

- Empty input expression
- Input expression starts with closing bracket.
- Any of three brackets are not closed.

3) Linear Data Structures – Stacks

Applications of a Stack - Example: String reversal

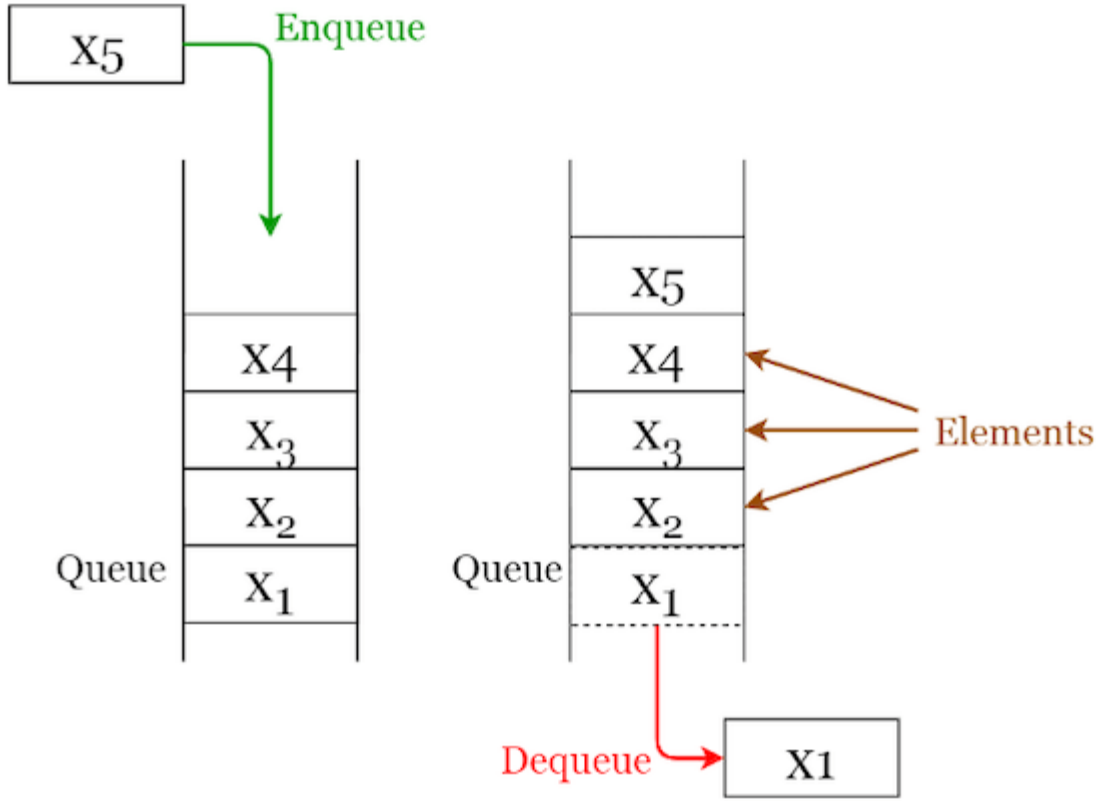
- String reversal is also another application of stack. Here one by one each character gets inserted into the stack. So, the first character of the string is on the bottom of the stack and the last element of a string is on the top of the stack. After Performing the pop operations on the stack, we get a string in reverse order.



4) Linear Data Structures – Queues

Queues – a linear order structure. It works in a FIFO (First In First Out) order.

- FIFO means the first element can be accessed first.
- You can add a new element (**enqueue**) at the end of the structure and **dequeue** (delete) an element from the starting of the structure.
- They are mostly used in multithreading to manage threads as well as to execute priority queuing systems.



4) Linear Data Structures – Queues

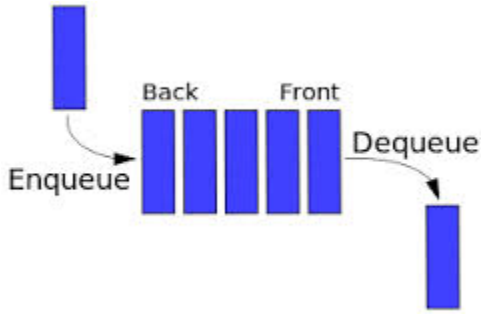
Queue – is a linear container of objects that are inserted and removed according to the **first-in first-out** (**FIFO**) principle. The difference between **stacks** and **queues** is in removing the elements.

Illustration of a queue: A line of students in a cafeteria (first came, first served).

Basic operations performed in the Queue:

- **Enqueue:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
- **Front:** Get front item
- **Rear:** Get Rear item

Illustration of the operations: **Enqueue** and **Dequeue**. We *enqueue* (add) an item at the *back* and *dequeue* (remove) an item from the *front*.



4) Linear Data Structures – Queues

Time Complexities of operations on queue: Constant - all take O(1) time (no need to run any loop in any of these operations.).

Space Complexity: O(N) – where N is the size of array for storing elements.

Implementation: We need to keep track of two indices, front and rear. If we simply increment front and rear indices, then there may be problems, the front may reach the end of the array. The solution to this problem is to increase front and rear in circular manner.

1. Using array
2. Using linked list

Applications of Queue: Queue is used when things don't have to be processed immediately but have to be processed in First-In First-Out order like *Breadth First Search*. This property of Queue makes it also useful in following kind of scenarios.

1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.

2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

4) Linear Data Structures – Queues

Advantages: Easy to implement.

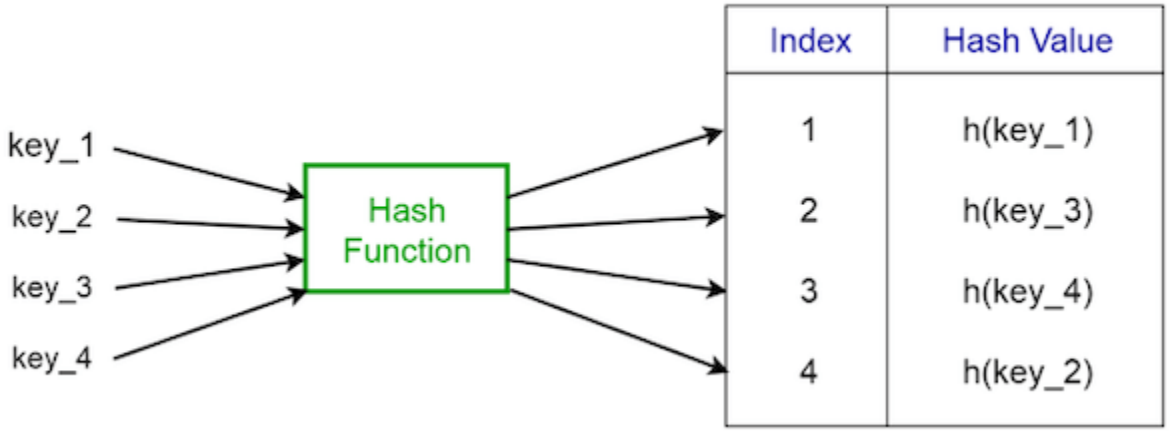
Disadvantages: Static Data Structure of fixed size.

A priority queue is an abstract data type similar to a regular queue or stack data structure in which each element additionally has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority.

5) Non-Linear Data Structures – Hash Tables

Hash table – a data structure that connects and stores each **value** with a **key**. It provides efficiently lookup values by using a key.

- An example, Social Security Number (**key**) assigned to workers (**value**), so all the working history (and social security income) can be related to a particular person.
- To map any size of data set to a fixed size, the hash table uses the **hash function**. The values returned by a hash function are known as hash values.
- They are mostly used to create associate arrays, database indexes, and a "set."



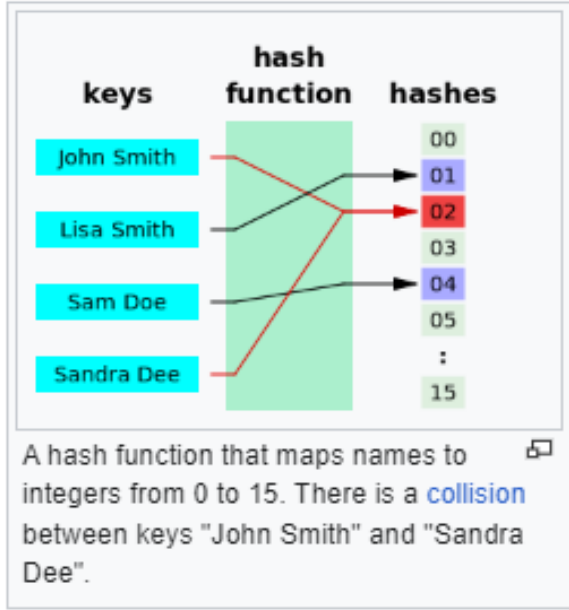
Hashing – Motivation

Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:

1. **Insert** a phone number and corresponding information (access memory many times).
2. **Search** a phone number and fetch the needed information.
3. **Delete** a phone number and related information (access memory many times).

We can think of using the following data structures to maintain information about different phone numbers.

1. **Array** of phone numbers and records.
2. **Linked List** of phone numbers and records.
3. **Balanced binary** search **tree** with phone numbers as keys.
4. Direct Access Table.



A hash function that maps names to integers from 0 to 15. There is a collision between keys "John Smith" and "Sandra Dee".

Source: Wikipedia

For **arrays and linked lists**, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(LogN) time using Binary Search but insert and delete operations become costly as we have to maintain sorted order.

Hash Tables

Hash tables are among the oldest, simplest, fastest and most widely used data structures today.

In a 1953 internal IBM paper, an engineer at IBM named Hans Peter Luhn proposed a new technique for storing and retrieving information - the **hash table**, that is now built into just about all computational systems.

Designed to perform three basic operations:

1. Insertions, which add new items to the database
2. Queries, which access an item or check to see whether it exists
3. Deletions.

Information is stored as a pair such as in English Dictionary:

- key that identifies the information (a word)
- the information itself (word definition)

Any database or browser have multiple built-in hash tables intended to keep track of different kinds of data.

In 1953, Hans Peter Luhn suggested a new way to store and retrieve information called the hash table.

—

Courtesy of IBM Corporation

Image Source: Quanta Magazine

Hash Tables

- **Hash tables** are dynamic. Their advantage is in making it easy to insert new words and delete outdated ones.
- They can find **any** word in a **constant** amount of time. While the search time for other methods can go up as the number of words in the dictionary increases.

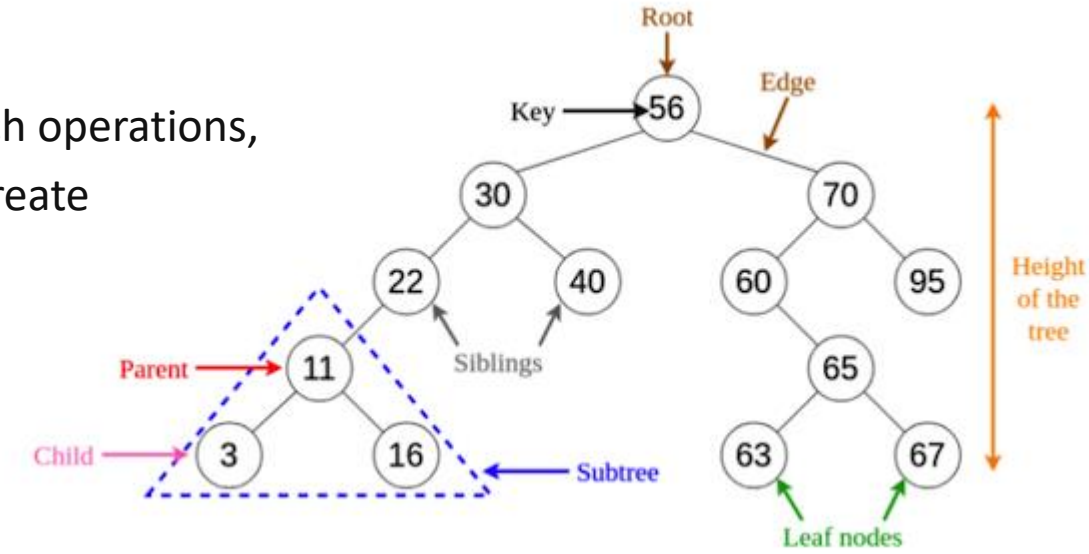
- Because of how they're built, hash tables need more memory than just the bare minimum required to store a given set of items. Thus, hash table technology comes with an unavoidable trade-off.
  - Hash tables need to be **fast**. Operations are quicker when the hash table has more memory, slower with less memory.
  - Hash tables need to be **compact**, using as little memory as possible.
- In 2003, researchers showed that it was theoretically possible to make a major efficiency leap in both time and space simultaneously.

- Two decades later, in 2023 computer scientist at Harvard finally have found the best balance between time and space complexity of hash table data structures. They introduced hash table with 2 data structures:
  - Primary – stores items in a minimally possible way. It stores the preferred storage locations as integer.
  - Secondary – helps a query request find the item it's looking for. Operates only with the integers from primary data structure (finds 1, 2,…).
  - Note: To represent 1 in binary form (1100100) you need more memory, significantly more if you have to store millions of numbers!

6) Non-Linear Data Structures – Trees

Trees – a basic data structure in which the data is linked together as in the linked list but organized hierarchically, as in a family tree.

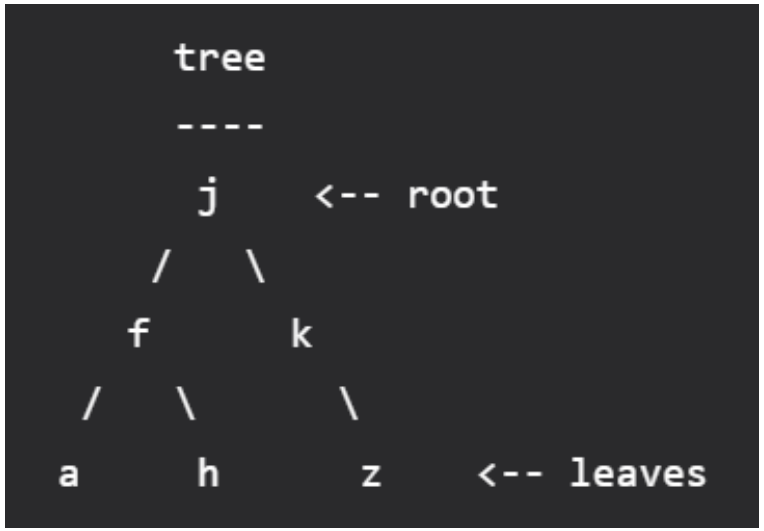
- Various types of trees, each type is suited for certain applications.
- For example, the Binary Search Tree (BST) data structure stores values (data) in sorted order.
- In a BST, every node comprises the following attributes:
  - **Key** is the stored value in the node
  - **Left** is the Pointer to the left child node
  - **Right** is the point to the right child node
  - **P** is the Pointer to the parent node
- BST structure is widely used in different types of search operations, and other types of tree structures are used to create expression solvers and in wireless networking.



6) Non-Linear Data Structures – Trees

Trees – Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

Tree Vocabulary: The topmost node is called **root** of the tree. The elements that are directly under an element are called its **children**. The element directly above something is called its **parent**. For example, ‘a’ is a child of ‘f’, and ‘f’ is the parent of ‘a’. Finally, elements with no children are called **leaves**.



6) Non-Linear Data Structures – Trees

Why Using Trees?

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer.
2. Trees (with some ordering e.g., BST - Binary Search Tree) provide moderate access/search (**quicker** than Linked List and **slower** than arrays).
3. Trees provide moderate insertion/deletion (**quicker** than Arrays and **slower** than Unordered Linked Lists).
4. Like Linked Lists and unlike Arrays, Trees don't have an **upper limit** on number of nodes as nodes are linked using pointers.

Applications of Trees:

1. Manipulate hierarchical data.
2. Make information easy to search (tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms
6. Form of a multi-stage decision-making (see business chess).

6) Non-Linear Data Structures – Trees

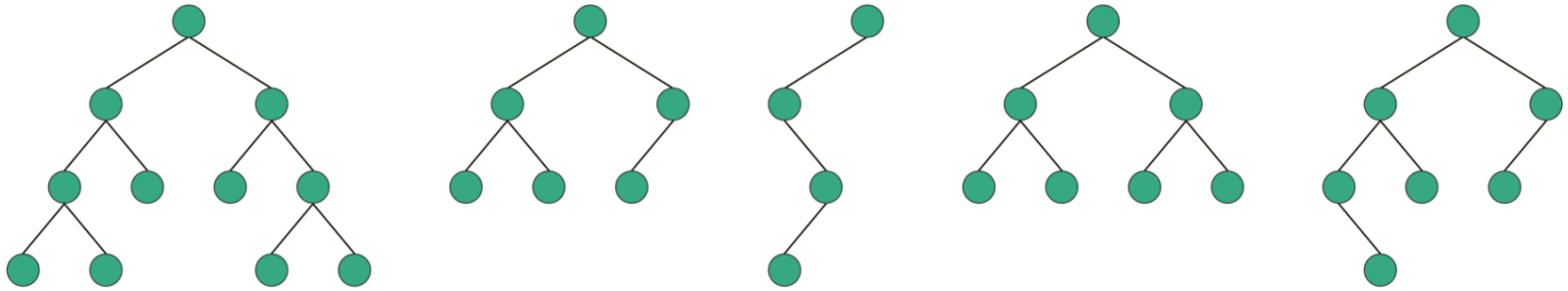
Binary Trees – A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Properties:

- **Level** is the number of nodes on the path from the root to the node (including root and node). Level of the root is 0.
- **Height** of a tree is the *maximum* number of nodes on the root to leaf path. Height of a tree with a single node is considered as 1.

6) Non-Linear Data Structures – Binary Trees

Types of Binary Trees:

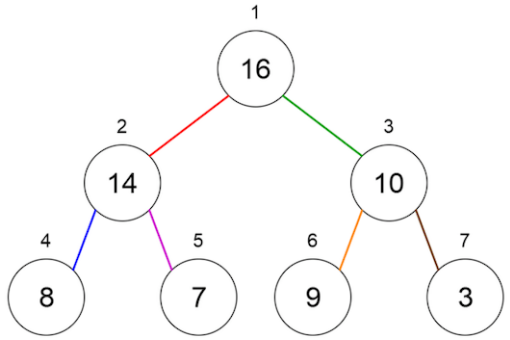


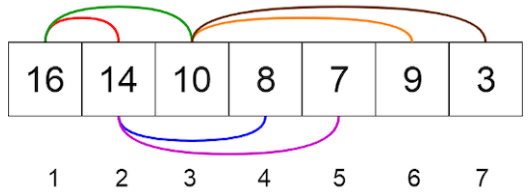
- **Full Binary Tree** – A binary tree in which all nodes except leaf nodes have two children.
- **Complete Binary Tree** – If all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.
- **Degenerate Binary Tree** – A Tree where every internal node has one child. Such trees are performance-wise same as linked list.
- **Perfect Binary Tree** – A Binary Tree in which all the internal nodes have two children, and all leaf nodes are at the same level.
- **Balanced Binary Tree** – If the difference between the heights of the left and right subtrees is at most 1.

7) Non-Linear Data Structures – Heap

Heap – a specific type of binary tree where the parent nodes are compared to their child nodes, and values are arranged in the nodes accordingly.

- A heap can be represented as an array or a binary tree.
- There are two types of heaps:
  1. **Min heap**, where the parent's key is equal or less than the keys of its children.
  2. **Max heap**, where the parent's key is greater than the keys of its children.
- Heaps are widely used to find the largest and smallest values in an array and to create priority queues in algorithms.





7) Non-Linear Data Structures – Heap

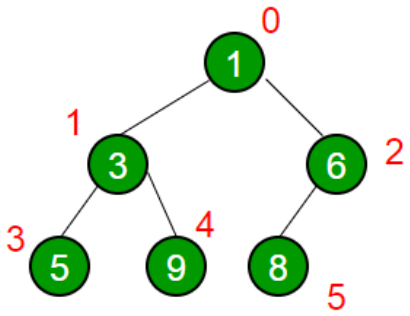
Heap – A heap is an implementation of a data type called a priority queue. It is a specialized tree-based data structure (Parent – Child nodes that have values) that satisfies the *maximum heap property* (the minimum is inverse):

- For any node C, the key (the value) of the parent node P is greater than or equal to the key of C.

Heaps are usually implemented with an array, as follows:

- Each element in the array represents a node of the heap, and
- The parent / child relationship is defined implicitly by the elements' indices in the array.
- Level Order is used as a traversal method achieve Array representation (min heap prop)

7) Non-Linear Data Structures – Heap



1	3	6	5	9	8
0	1	2	3	4	5

Given a node at index i, its children are at indices 2i + 1 and 2i + 2

i=2 (node 6), has children at array indices 2i + 1=5 (node 8) and 2i + 2=6 (out)

i=1 (node 3), has children at array indices 2i + 1=3 (node 5) and 2i + 2=4 (9)

This simple indexing scheme makes it efficient to move "up" or "down" the tree.

A Min heap is typically represented in Python as an array.

The root element will be at **Arr[0]**.

For any i-th node, i.e., Arr[i] we have:

Arr[(i -1) / 2] returns its parent node.

Arr[(2 * i) + 1] returns its left child node.

Arr[(2 * i) + 2] returns its right child node.

7) Non-Linear Data Structures – Binary Heap

Binary Heap – Is a Binary Tree with following properties:

- It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
- A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

Implementation: A binary heap is typically represented as an array.

Applications of Heaps:

1. **Heap Sort**: Heap Sort uses Binary Heap to sort an array in O(nLogn) time.
2. **Priority Queue**: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time. Binomoial Heap and Fibonacci Heap are variations of Binary Heap. These variations perform union also efficiently.
3. **Graph Algorithms**: The priority queues are especially used in Graph Algorithms like Dijkstra's Shortest Path and Prim's Minimum Spanning Tree.
4. Many problems can be efficiently solved using Heaps. See following for example.
    1. Kth Largest Element in an array.
    2. Sort an almost sorted array.
    3. Merge K Sorted Arrays.

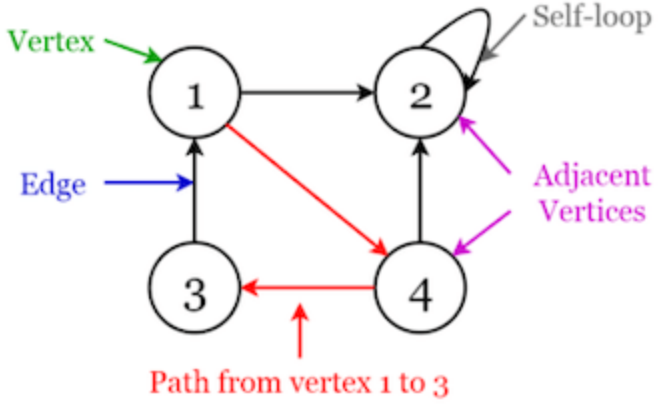
7) Non-Linear Data Structures – Binary Heap

A **Binary Heap** is a **binary tree** (each node has at most 2 children) structure. It is used as a search structure. They can become unbalanced (by insert operations), so that some nodes are deep in the tree, decreasing the search. Keeping it balanced search cost would only be Θ(log(n)). Balancing a tree ensures that the path length from the root to any leaf node is similar and minimized.

8) Advanced Data Structures – Graphs

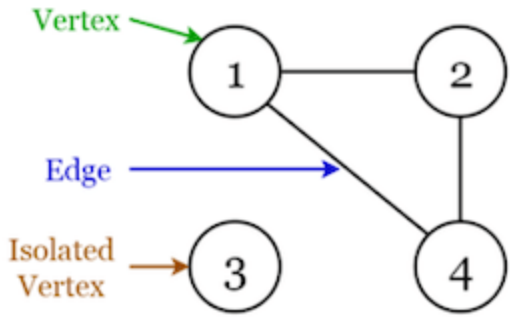
Graphs – Advanced and abstract data structure that consists of a fixed (finite) set of nodes or vertices and is connected by a set of edges. Edges are the arcs or lines that simply connect nodes in the graph.

- Graphs are great for solving real-world problems, as well as representations of digital networks. They're also used for the representations of the networks like circuit networks.



Directed Graph

G = {1, 2, 3, 4}
E = {(1, 2), (1, 4), (2, 2), (3, 1), (4, 3), (4, 2)}



Undirected Graph

G = {1, 2, 3, 4}
E = {(1, 2), (1, 4), (2, 4)}

9) Advanced Data Structures – Trie

Trie – Advanced data structure that.

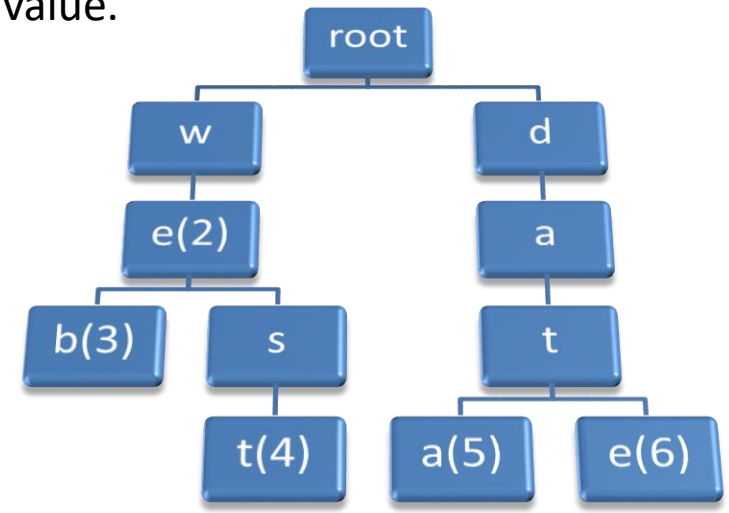
- Trie.

9) Advanced Data Structures – Trie

Trie – Advanced data structure that is used in finding prefix string matches.

- Predictive text or auto complete dictionary, such as is found in a spell check or on a mobile telephone.

- The trie data structure (from the word re***trie***val), or prefix tree, stores strings by decomposing them into characters.
- The complexity $O(n^{\alpha})$ (polynomial) of an algorithm is a function of the size of the input n (in bits).
- Consider: M maximum string length, N is number of encoded words (keys).
  - For tree, the search time is proportional to $M*\log_2(N)$ - logarithmic in time .
  - For trie, the search time is proportional to O(M) - linear in time. However, the trie has larger storage (memory) complexity (requirements).
- Retrieving words is done by traversing the tree structure to the node that represents the prefix being queried.
- A search ends if a node has a non-null value.



A

A

A

A