MET CS688

Web Analytics and Mining

Zlatko Vasilkoski

Basic Data Structures

How to Evaluate ML Algorithm?

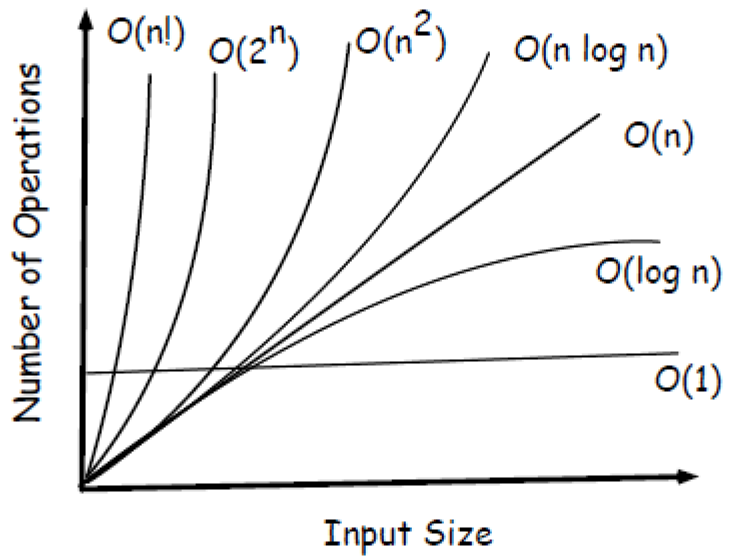
1. Efficiency and Computational Complexity (Space, Time). Along efficiency it includes resource utilization including response time, memory use, … as well

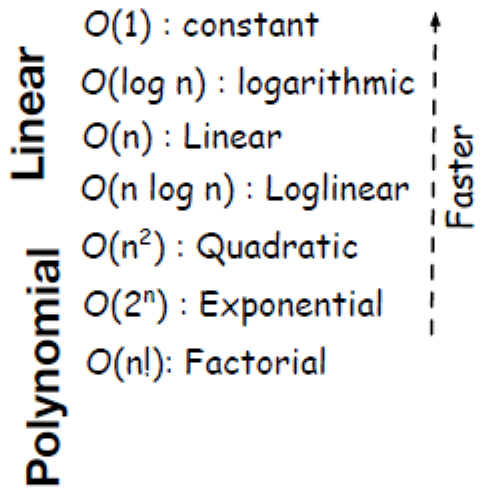
2. Accuracy

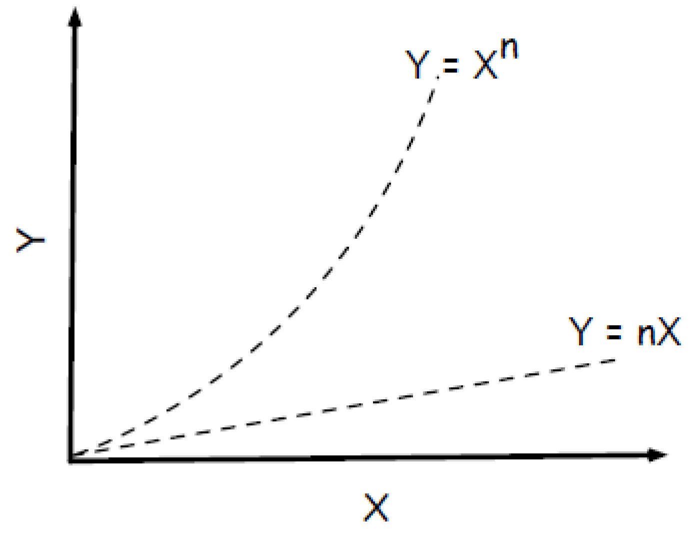
3. Algorithm Efficiency

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.

Data structures

A data structure is a particular way of organizing data in a computer so that it can be used effectively. For example, we can store a list of items having the same data-type using the array data structure.

This difference in the data storage scheme decides which data structure would be more suitable for a given situation.

- Linear Data Structures
  - Array
  - Linked List
  - Stack & Queue
- Non-Linear Data Structures
  - Matrix
  - Trees
  - Heaps
  - Hash
- Advanced Data Structures
  - Graph
  - Trie

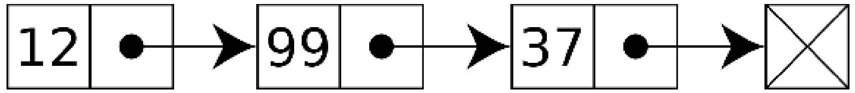
Linear Data Structures

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.

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 using pointers).

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

Linked list nodes contain two fields:



1) Data (we can store 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.

Linear Data Structures – Linked List

Why Using Linked List?

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

1. Memory linked to the number of elements in arrays. 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.

Linear Data Structures – Linked List

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.

Linear Data Structures – Linked List

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.

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.

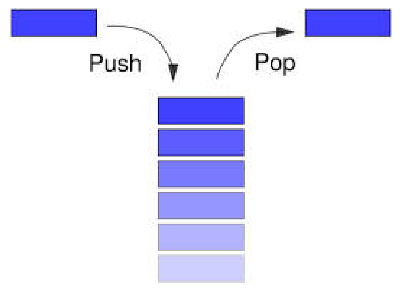
Illustration of a Stack: plates stacked on top of each other.

- 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**



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

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.

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**”.

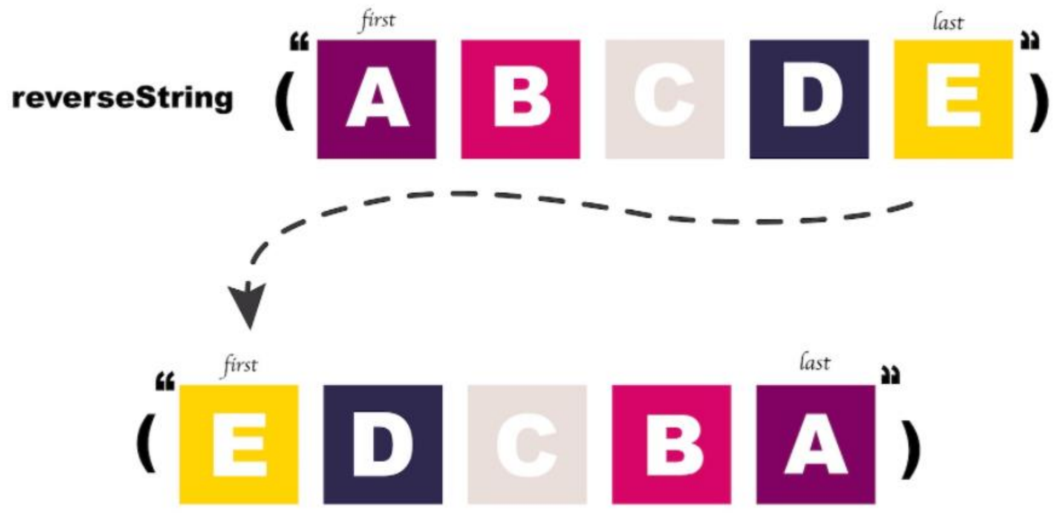
There are 5 different situations when the brackets are **not balanced**.

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

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.



Linear Data Structures – Queue

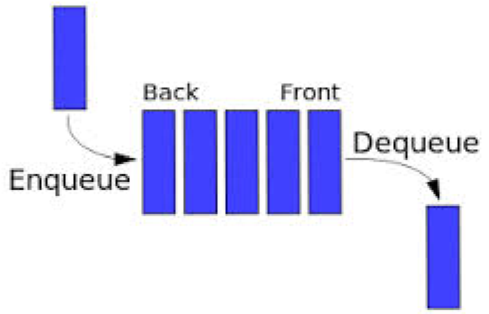
Queue – is a linear container of objects that are inserted and removed according to the **first-in first-out** (**FIFO**) principle. The difference between **stacks** and **queues** is in removing the elements.

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*.



Linear Data Structures – Queue

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.

Linear Data Structures – Queue

Advantages: Easy to implement.

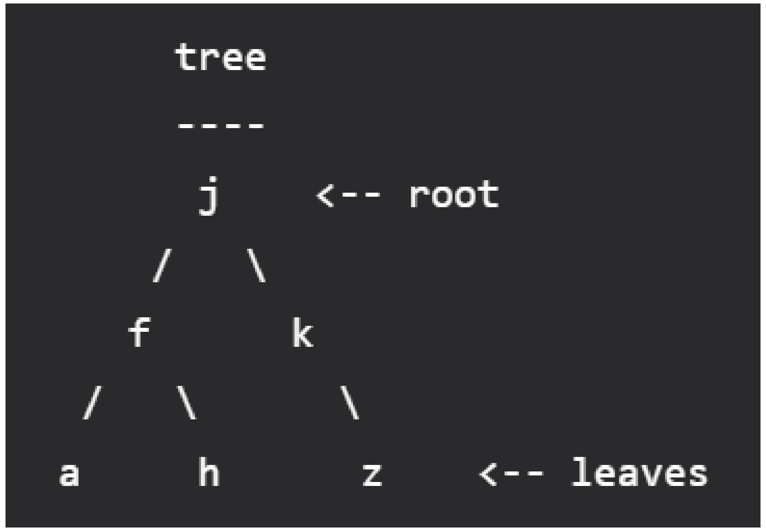
Disadvantages: Static Data Structure of fixed size.

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.

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**.



```
     tree
     ----
      j    <-- root
    /   \
   f      k
 /   \      \
a     h      z    <-- leaves
```

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) 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).

Non-Linear Data Structures – Binary Trees

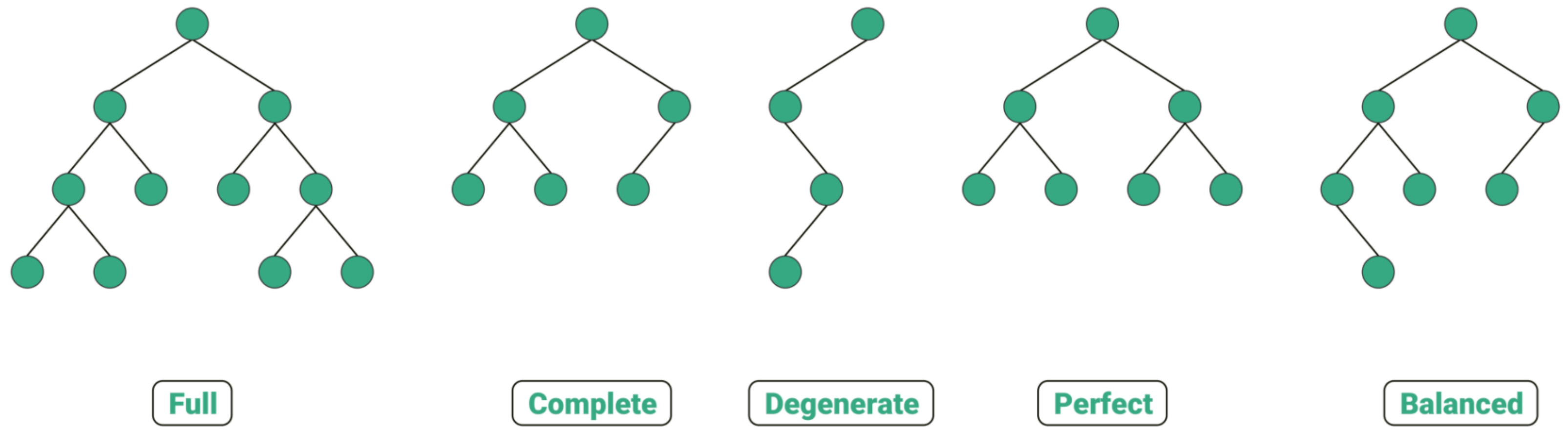
Binary Trees – A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

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.

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.

Non-Linear Data Structures – Heap

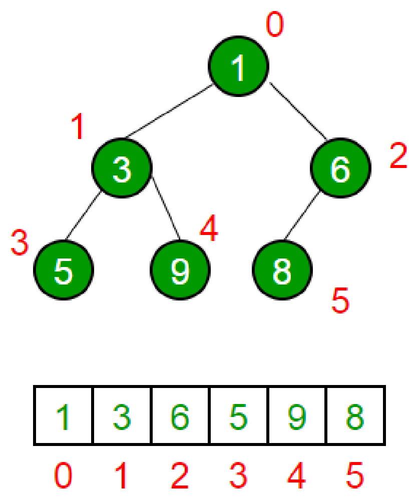
Heap – A heap is an implementation of a data type called a priority queue. It is a specialized tree-based data structure (Parent – Child nodes that have values) that satisfies the *maximum heap property* (the minimum is inverse):

- 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)

Non-Linear Data Structures – Heap



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.

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.

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.

Non-Linear Data Structures – Hashing

Hashing –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.
2. Search a phone number and fetch the information.
3. Delete a phone number and related information.

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.

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.

A

A