Data Structures are the programmatic way of storing data so that data can be used efficiently..

## Why to Learn Data Structure and Algorithms?

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

**Data Search** − Consider an inventory of 1 million(106) items of a store. If the application is to search an item, it has to search an item in 1 million(106) items every time slowing down the search. As data grows, search will become slower.

**Processor speed** − Processor speed although being very high, falls limited if the data grows to billion records.

**Multiple requests** − As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

To solve the above-mentioned problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

## Applications of Data Structure and Algorithms

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms −

**Search** − Algorithm to search an item in a data structure.

**Sort** − Algorithm to sort items in a certain order.

**Insert** − Algorithm to insert item in a data structure.

**Update** − Algorithm to update an existing item in a data structure.

**Delete** − Algorithm to delete an existing item from a data structure.

The following computer problems can be solved using Data Structures −

* Fibonacci number series
* Knapsack problem
* Tower of Hanoi
* All pair shortest path by Floyd-Warshall
* Shortest path by Dijkstra
* Project scheduling

**Java-algorithm-complexity**

**Big O Notation :-**

We often hear the performance of an algorithm described using Big O Notation.

The study of the performance of algorithms – or algorithmic complexity – falls into the field of algorithm analysis. Algorithm analysis answers the question of how many resources, such as disk space or time, an algorithm consumes.

We'll be looking at time as a resource. Typically, the less time an algorithm takes to complete, the better.

**Constant Time Algorithms – O(1) :-**

How does this input size of an algorithm affect its running time? Key to understanding Big O is understanding the rates at which things can grow. The rate in question here is time taken per input size.

Consider this simple piece of code:

int n = 1000;

System.out.println("Hey - your input is: " + n);

System.out.println("Hmm.. I'm doing more stuff with: " + n);

Clearly, it doesn't matter what n is, above. This piece of code takes a constant amount of time to run. It's not dependent on the size of n.

The above example is also constant time. Even if it takes 3 times as long to run, it doesn't depend on the size of the input, n. We denote **constant time algorithms** as follows: O(1). Note that O(2), O(3) or even O(1000) would mean the same thing.

We don't care about exactly how long it takes to run, only that it takes constant time.

**Logarithmic Time Algorithms – O(log n) :-**

Constant time algorithms are (asymptotically) the quickest.

Logarithmic time is the next quickest. Unfortunately, they're a bit trickier to imagine.

One common example of a logarithmic time algorithm is the binary search algorithm.

What is important here is that the running time grows in proportion to the logarithm of the input (in this case, log to the base 2):

for (int i = 1; i < n; i = i \* 2){

System.out.println("Hey - I'm busy looking at: " + i);

}

If n is 8, the output will be the following:

Hey - I'm busy looking at: 1

Hey - I'm busy looking at: 2

Hey - I'm busy looking at: 4

Our simple algorithm ran log(8) = 3 times.

what logarithms mean and how to use them when applied to the calculation of the time complexity of algorithms.

Relationship between the number of operations and the size of the input as it grows.

**Logarithms** are the mathematical inverse of exponentials.

Let’s look at an example of an exponential:

[ 2^3=8 ]

This can be described as what is the number that we get when multiplying 2 by itself 3 times:

[ 2^3 => 2\*2\*2=8 ]

Now let’s look at its inverse operation, the logarithm:

[ log\_2(8)=3 ]

This can be imagined as how many times we divide the number 8 by 2 to get to 1:

\[ \log\_2(8) = 3 => (((8 / 2) / 2) / 2) \]

Why Base-2?

Which brings us to the question: Why base-2? Really, it’s just that it comes up so frequently.

For example, they often come up when designing algorithms:

When we need to repeatedly divide an array in half – this is an operation used, for instance, in some sorting, like Merge Sort, or searching algorithms, like Binary Search; in this scenario, the number of times we can divide an array of size n in half is log2(n)

When doing bit operations – for instance, writing a number in binary uses about log2(n) bits

Because of this, if we classify something as O(log n) we’ll typically mean O(log2 n). Using Big-O notation, though, this simplification doesn’t cause a problem since all logarithms are asymptotically equivalent.

When analyzing the time complexity of an algorithm, the question we have to ask is what’s the relationship between its number of operations and the size of the input as it grows.

Very commonly, we’ll use Big-O notation to compare the time complexity of different algorithms.

**Linear Time Algorithms – O(n) :-**

After logarithmic time algorithms, we get the next fastest class: linear time algorithms.

If we say something grows linearly, we mean that it grows directly proportional to the size of its inputs.

Think of a simple for loop:

for (int i = 0; i < n; i++) {

System.out.println("Hey - I'm busy looking at: " + i);

}

How many times does this for loop run? n times, We don't know exactly how long it will take for this to run – and we don't worry about that.

What we do know is that the simple algorithm presented above will grow linearly with the size of its input.

Again, if the algorithm was changed to the following:

for (int i = 0; i < n; i++) {

System.out.println("Hey - I'm busy looking at: " + i);

System.out.println("Hmm.. Let's have another look at: " + i);

System.out.println("And another: " + i);}

The runtime would still be linear in the size of its input, n. We denote linear algorithms as follows: O(n).

As with the constant time algorithms, we don't care about the specifics of the runtime. O(2n+1) is the same as O(n), as Big O Notation concerns itself with growth for input sizes.

**N Log N Time Algorithms – O(n log n) : -**

n log n is the next class of algorithms. The running time grows in proportion to n log n of the input:

for (int i = 1; i <= n; i++){

for(int j = 1; j < n; j = j \* 2) {

System.out.println("Hey - I'm busy looking at: " + i + " and " + j);

}

}

For example, if the n is 8, then this algorithm will run 8 \* log(8) = 8 \* 3 = 24 times. Whether we have strict inequality or not in the for loop is irrelevant for the sake of a Big O Notation.

**Examples :** **Practice Questions on Time Complexity Analysis**

**What is the time, space complexity of following code:**

int a = 0, b = 0;

for (i = 0; i < N; i++) {

a = a + rand();

}

for (j = 0; j < M; j++) {

b = b + rand();

}

Ans : **O(N + M) time, O(1) space**

**Explanation:** The first loop is O(N) and the second loop is O(M). Since we don’t know which is bigger, we say this is O(N + M). This can also be written as O(max(N, M)).  
Since there is no additional space being utilized,

**The space complexity is constant / O(1)**

**2. What is the time complexity of following code:**

int a = 0;

for (i = 0; i < N; i++) {

for (j = N; j > i; j--) {

a = a + i + j;

}

}

O(N\*N)

**3. What is the time complexity of following code:**

int i, j, k = 0;

for (i = n / 2; i <= n; i++) {

for (j = 2; j <= n; j = j \* 2) {

k = k + n / 2;

}

}

O(nLogn)

**Explanation:**If you notice, j keeps doubling till it is less than or equal to n. Number of times, we can double a number till it is less than n would be log(n).  
Let’s take the examples here.  
for n = 16, j = 2, 4, 8, 16  
for n = 32, j = 2, 4, 8, 16, 32  
So, j would run for O(log n) steps.  
i runs for n/2 steps.  
So, total steps = O(n/ 2 \* log (n)) = **O(n\*logn)**

**4. What does it mean when we say that an algorithm X is asymptotically more efficient than Y?**

X will always be a better choice for large inputs

**Explanation:** In asymptotic analysis we consider growth of algorithm in terms of input size. An algorithm X is said to be asymptotically better than Y if X takes smaller time than y for all input sizes n larger than a value n0 where n0 > 0.

**5. What is the time complexity of following code:**

int a = 0, i = N;

while (i > 0) {

a += i;

i /= 2;

}

Binary search algorithm as a example

**Explanation:** We have to find the smallest x such that N / 2^x N  
x = log(N)

**Linked List :**

A linked list is a common data structure that is made of a chain of nodes. Each node contains a value and a pointer to the next node in the chain.

The head pointer points to the first node, and the last element of the list points to null. When the list is empty, the head pointer points to null.

Linked lists can dynamically increase in size. It is easy to insert and delete from a linked list because unlike arrays, as we only need to change the pointers of the previous element and the next element to insert or delete an element.

Some important applications of Linked Lists include:

* Implementing [HashMaps](https://www.educative.io/blog/data-strucutres-hash-table-javascript" \t "https://www.educative.io/blog/_blank), File System and Adjacency Lists
* Dynamic memory allocation: use linked lists of free blocks
* Performing arithmetic operations on long integers
* Maintaining a directory of names

**Types of linked lists**

Since a linked list is a linear data structure, meaning that the elements are not stored at contiguous locations, it’s necessary to have different types of linked lists to access and modify our elements accordingly.

There are a three different types of linked lists

#### **Singly linked list (Uni-directional) :** A singly linked list is unidirectional, meaning that it can be traversed in only one direction from head to the last node (tail).

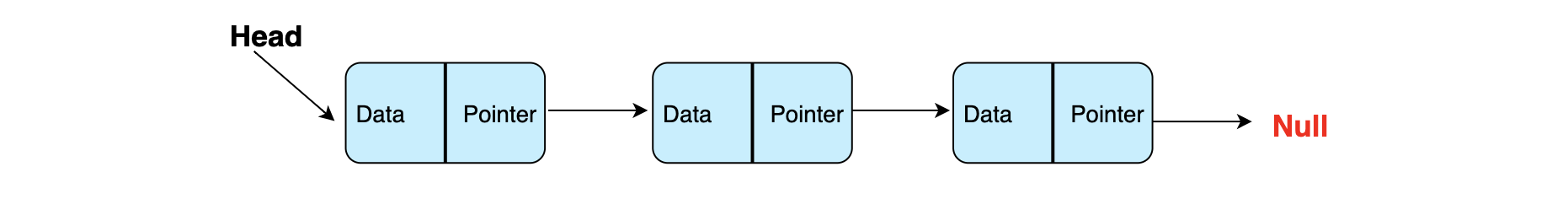
**Doubly linked list (Bi-directional) :** Doubly linked lists (DLLs) are an extension of basic linked lists, but they contain a pointer to the next node as well as the previous node. This ensures that the list can be traversed in both directions. A DLL node has three fundamental members:

* The data
* Pointer to the next node
* Pointer to the previous node

**Circular linked list :** Circular linked lists function circularly: the first element points to the last element, and the last element points to the first. A single linked list and double linked list can be made into a circular linked list. The most important operations for a circular linked list are:

* insert − insert elements at the start of the list
* display − display the list
* delete − delete elements from the start of the list

**Structure of a Singly Linked List**



In Java, the **linked list** class is an ordered collection that contains many objects of the same type. Data in a Linked List is stored in a sequence of containers. The list holds a reference to the first container and each container has a link to the next one in the sequence.

**Linked lists** in Java implement the abstract list interface and inherit various constructors and methods from it. This sequential data structure can be used as a list, [stack](https://www.educative.io/blog/data-structures-stack-queue-java-tutorial" \l "stack" \t "https://www.educative.io/blog/_blank) or [queue](https://www.educative.io/blog/data-structures-stack-queue-java-tutorial" \l "queue" \t "https://www.educative.io/blog/_blank).

To implement a linked list, we need the following two classes:

**Class Node :** The Node class stores data in a single node. It can store primitive data such as integers and string as well as complex objects having multiple attributes.

Along with data, it also stores a pointer to the next element in the list, which helps in linking the nodes together like a chain.

public class Node<T> {

  public T data; //Data to store (could be int, string, Object etc)

  public Node nextNode; //Pointer to next node in list

}

**Class Linked list :** The Singly Linked List contains a head node: a pointer pointing to the first element of the list. Whenever we want to traverse the list, we can do so by using this head node.

### **Algorithm**

* Create a class Node which has two attributes: data and next. Next is a pointer to the next node.
* Create another class which has two attributes: head and tail.
* addNode() will add a new node to the list:
* Create a new node.
* It first checks, whether the head is equal to null which means the list is empty.
* If the list is empty, both head and tail will point to the newly added node.
* If the list is not empty, the new node will be added to end of the list such that tail's next will point to the newly added node. This new node will become the new tail of the list.