URBAN EDGE

Data Structures & Algorithms by CodeWithHarry

This course will get you prepared for placements and will teach you how to create efficient and fast algorithms.

Data structures and algorithms are two different things.

Data Structures: Arrangement of data so that they can be used efficiently in memory (data items)

Algorithms: Sequence of steps on data using efficient data structures to solve a given problem.

Other Terminology

Database - Collection of information in permanent storage for faster retrieval and updation.

Data warehousing - Management of huge amount of legacy data for better analysis.

Big data - Analysis of too large or complex data which cannot be dealt with traditional data processing application.

Data Structures and Algorithms are nothing new. If you have done programming in any language like C you must have used Arrays → A data structure and some sequence of processing steps to solve a problem → Algorithm ☺

URBAN EDGE

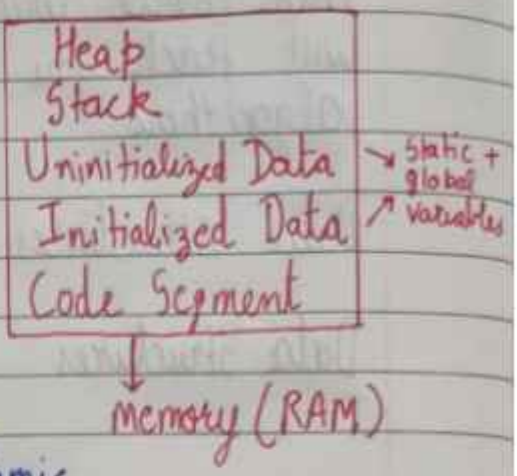
Memory layout of C programs

When the program starts, its Code is copied to the main memory.

Stack holds the memory occupied by the functions.

Heap contains the data which is requested by the program as dynamic memory.

Initialized and uninitialized data segments hold initialized and uninitialized global variables respectively



URBAN EDGE

Time Complexity & Big O notation

This morning I wanted to eat some pizzas; So I asked my brother to get me some from Dominos (3 km far)
He got me the pizza and I was happy only to realize it was too less for 29 friends who came to my house for a surprize visit!

My brother can get 2 pizzas for me on his bike but pizza for 29 friends is too huge of an input for him which he cannot handle.

2 pizzas →  okay! not a big deal!

68 pizzas →  Not possible! in short time

What is Time Complexity?
Time Complexity is the study of efficiency of algorithms.

(i) Time Complexity = How time taken to execute an algorithm grows with the size of the input!

Consider two developers who created an algorithm to sort n numbers. Shubham and Rohan did this independently.

When ran for input size n, following results were recorded:

no of elements (n)	Shubham's Algo	Rohan's Algo
10 elements	90 ms	122 ms
70 elements	110 ms	124 ms
110 elements	180 ms	131 ms
1000 elements	2 s	800 ms

We can see that initially Shubham's algorithm was shining for smaller input but as the number of elements increases Rohan's algorithm looks good!

Quick Quiz: Who's Algorithm is better?

Time Complexity: Sending GTA V to a friend

Let us say you have a friend living 5 kms away from your place. You want to send him a game.

Final exams are over and you want him to get this 60 GB file from you. How will you send it to him?

Note that both of you are using (JIO) 4G with 1 GB/day data limit.

The best way to send him the game is by delivering it to his house.
Copy the game to a Hard disk and send it!

Will yo do the same thing for sending a game like minesweeper which is in kbs of size?
No because you cand send it via internet.

As the file size grows, time taken by online sending increases linearly → $O(n^1)$
As the file size grows, time taken by physical sending remains constant. $O(n^0)$ or $O(1)$

Calculating order in terms of Input size
In order to calculate the order, most impactful term containing n is taken into account.
↳ Size of input

Let us assume that formula of an algorithm in terms of input size n looks like this:

Algo 1 → $\underbrace{k_1 n^2}_{\text{Highest order term}} + \underbrace{k_2 n + 36}_{\text{can ignore lower order terms}} \Rightarrow O(n^2)$

Algo 2 → $k_1 k_2^2 + k_3 k_2 + 8$

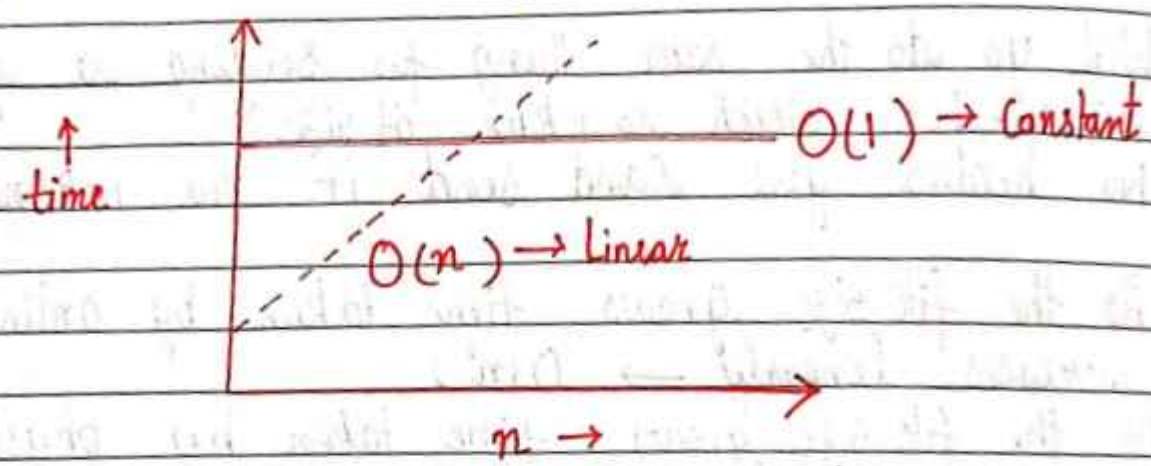
$\Downarrow$

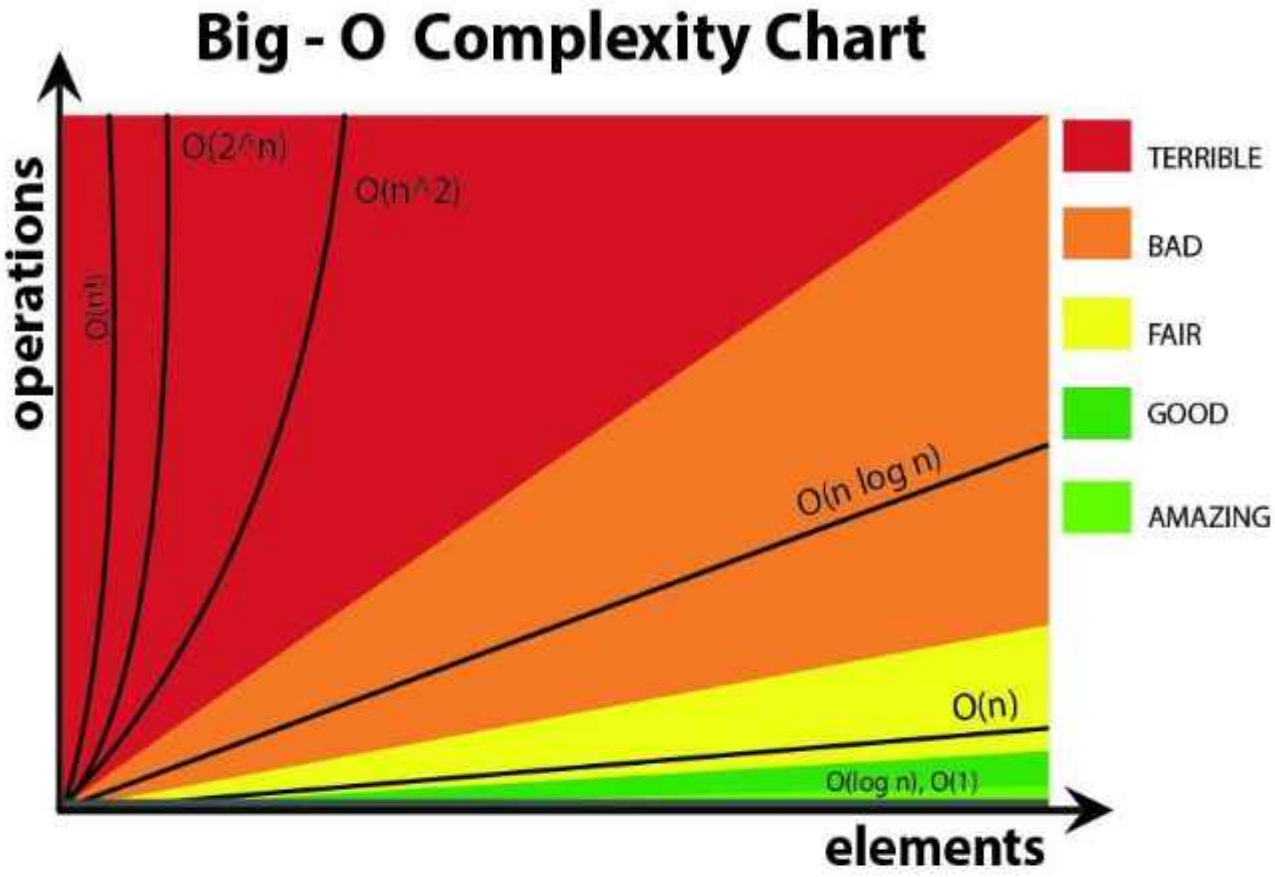
$k_1 k_2^2 n^0 + k_3 k_2 + 8 \Rightarrow O(n^0)$ or $O(1)$

Note that these are the formulas for time taken by them.

Visualising Big O

If we were to plot O(1) and O(n) on a graph, they will look something like this :





Source: https://stackoverflow.com/questions/3255/big-o-how-do-you-calculate-approximate-it

Asymptotic Notations

Asymptotic notations give us an idea about how good a given algorithm is compared to some other algorithm
Let us see the mathematical definition of "order of" now.

Primarily there are three types of widely used asymptotic notations.

1) Big oh notation (O)
2) Big omega notation (Ω)
3) Big theta notation (Θ) ⟶ Widely used one!

Big oh notation

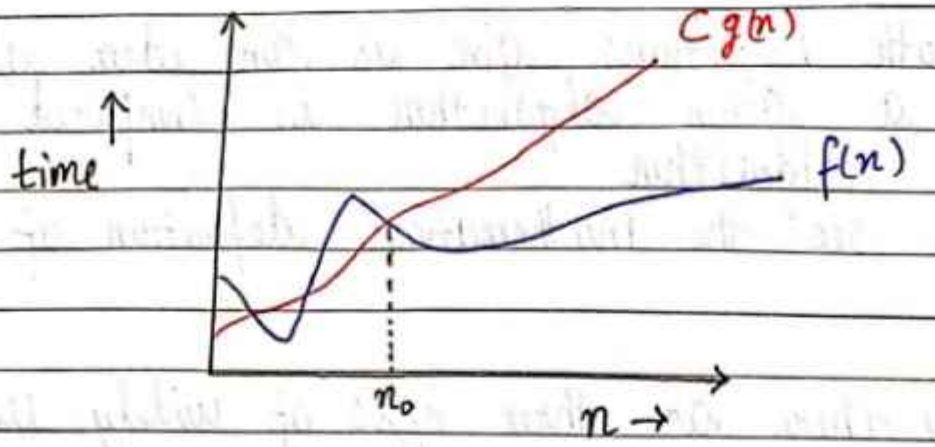
Big oh notation is used to describe asymptotic upper bound.
Mathematically, if $f(n)$ describes running time of an algorithm; $f(n)$ is $O(g(n))$ iff there exist positive constants C and n_0 such that

$$0 \leq f(n) \leq C\,g(n) \quad \text{for all } n \geq n_0$$

⇓
used to give upper bound on a function.

if a function is $O(n)$, it is automatically $O(n^2)$ as well!

Graphic example for Big oh (O)



Big Omega notation

Just like O notation provides an asymptotic upper bound, Ω notation provides asymptotic lower bound. Let $f(n)$ define running time of an algorithm;

$f(n)$ is said to be $\Omega(gn)$ if there exists positive constants c and n_0 such that

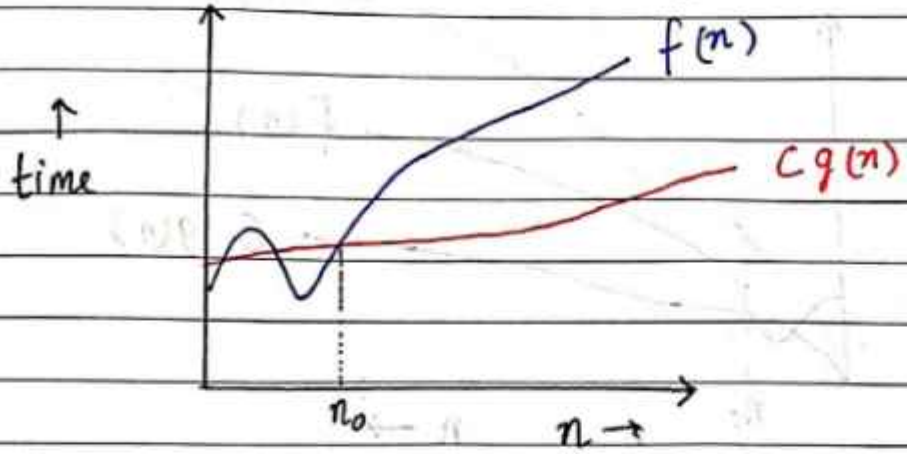
$$0 \leq c\,g(n) \leq f(n) \quad \text{for all } n \geq n_0$$

⇓ used to give lower bound on a function

if a function is $O(n^2)$ it is automatically $O(n)$ as well

URBAN EDGE

Graphic example for Big omega (Ω)



Big theta notation

Let $f(n)$ define running time of an algorithm

$f(n)$ is said to be $\Theta(gn)$ iff $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$

Mathematically,

$$0 \leq f(n) \leq C_1 g(n) \quad \forall\ n \geq n_0$$ → Sufficiently large value of n

$$0 \leq C_2 g(n) \leq f(n) \quad \forall\ n \geq n_0$$

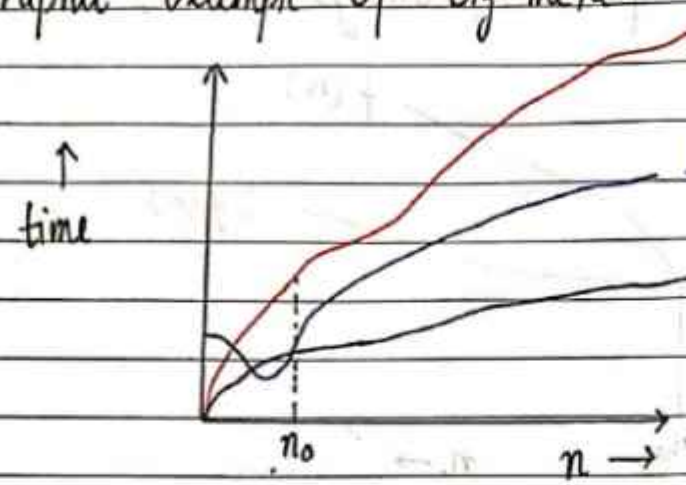
Merging both the equations, we get:

$$0 \leq C_2 g(n) \leq f(n) \leq C_1 g(n) \quad \forall\ n \geq n_0$$

The equation simply means there exist positive constants C_1 and C_2 such that $f(n)$ is sandwiched between $C_2 g(n)$ and $C_1 g(n)$

URBAN EDGE

Graphic example of Big theta



Which one of these to use?
Since Big theta gives a better picture of runtime for a given algorithm, most of the interviewers expect you to provide an answer in terms of Big theta when they say "order of".

Quick Quiz :- Prove that n^2+n+1 is $O(n^3)$, $\Omega(n^2)$ and $\Theta(n^2)$ using respective definitions.

Increasing order of common runtimes

$$1 < \log n < n < n\log n < n^2 < n^3 < 2^n < n^n$$

Better ⇓ Worse

Common runtimes from better to worse →

URBAN EDGE

Best, worst and Expected Case

Sometimes we get lucky in life. Exams cancelled when you were not prepared, surprise test when you were prepared etc. ⟹ Best case

Some times we get unlucky. Questions you never prepared asked in exams, rain during Sports period etc. ⟹ Worst case

But overall the life remains balance with the mixture of lucky and unlucky times. ⟹ Expected case.

Analysis of a search algorithm

Consider an array which is sorted in increasing order

1	7	18	28	50	180

We have to search a given number in this array and report whether it's present in the array or not.

Algo 1 → Start from first element until an element greater than or equal to the number to be searched is found.

Algo 2 → Check whether the first or the last element is equal to the number. If not find the number between these two elements (center of the array). If the center element is greater than the number to be searched, repeat the process for first half else repeat for second half until the number is found.

Analyzing Algo 1

If we really get lucky, the first element of the array might turn out to be the element we are searching for. Hence we made just one comparison.

Best case Complexity = $O(1)$

If we are really unlucky, the element we are searching for might be the last one.

Worst case Complexity = $O(n)$

For calculating Average case time, we sum the list of all the possible case's runtime and divide it with the total number of cases.

⇓

Sometimes calculation of average case time gets very complicated

Analyzing Algo 2

If we get really lucky, the first element will be the only one which gets compared

Best Case Complexity = $O(1)$

If we get unlucky we will have to keep dividing the array into halves until we get a single element (the array gets finished)

URBAN EDGE

Worst case Complexity = $O(\log n)$

What log(n)? What is that

log(n) → Number of times you need to half the array of size n before it gets exhausted

$\log 8 = 3 \Rightarrow \frac{8}{2} \rightarrow \frac{4}{2} \rightarrow \frac{2}{2}$ → Can't break anymore.

$1 + 1 + 1$

$\log 4 = 2 \Rightarrow \frac{4}{2} \rightarrow \frac{2}{2}$ → Can't break anymore.

$1 + 1$

Log n simply means how many time I need to divide n units such that we cannot divide them (into halves) anymore.

Space Complexity

Time is not the only thing we worry about while analyzing algorithms. Space is equally important.

Creating an array of size n → O(n) space
↳ Size of input

If a function calls itself recursively n times its space complexity is O(n).

URBAN EDGE

Quick Quiz → Calculate Space Complexity of a function which calculates factorial of a given number n.

Why cant we calculate Complexity in seconds?

→ Not everyone's Computer is equally powerful

→ Asymptotic analysis is the measure of how time (runtime) grows with input

URBAN EDGE

Techniques to Calculate Time Complexity

Once we are able to write the runtime in terms of size of the input (n), we can find the time complexity.

For example $T(n) = n^2 \Rightarrow O(n^2)$

$T(n) = \log n \Rightarrow O(\log n)$

Some tricks to calculate complexity

1. Drop the constants : Any thing you might think is $O(3n)$ is $O(n)$
   ↳ Better representation
2. Drop the non dominant terms : Any thing you represent as $O(n^2+n)$ can be written as $O(n^2)$
3. Consider all variables which are provided as input : $O(mn)$ & $O(mnq)$ might exist for some cases!

In most of the cases, we try to represent the runtime in terms of the input which can be more than one in number. For example :

Painting a park of dimension $m \times n \Rightarrow O(mn)$

Time Complexity – Competitive Practice Sheet

1. Fine the time complexity of the func1 function in the program show in program1.c as follows:

```c
#include <stdio.h>

void func1(int array[], int length)
{
    int sum = 0;
    int product = 1;
    for (int i = 0; i < length; i++)
    {
        sum += array[i];
    }

    for (int i = 0; i < length; i++)
    {
        product *= array[i];
    }
}

int main()
{
    int arr[] = {3, 5, 66};
    func1(arr, 3);
    return 0;
}
```

2. Fine the time complexity of the func function in the program from program2.c as follows:

```c
void func(int n)
{
    int sum = 0;
    int product = 1;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            printf("%d , %d\n", i, j);
        }
    }

}
```

3. Consider the recursive algorithm above, where the random(int n) spends one unit of time to return a random integer which is evenly distributed within the range [0,n][0,n]. If the average processing time is T(n), what is the value of T(6)?

```
int function(int n)
{
    int i;

    if (n <= 0)
    {
        return 0;
    }
    else
    {
        i = random(n - 1);
        printf("this\n");
        return function(i) + function(n - 1 - i);
    }
}
```

4. Which of the following are equivalent to O(N)? Why?
    a) O(N + P), where P < N/9
    b) 0(9N-k)
    c) O(N + 8log N)
    d) $O(N + M^2)$

5. The following simple code sums the values of all the nodes in a balanced binary search tree. What is its runtime?

```
int sum(Node node)
{
    if (node == NULL)
    {
        return 0;
    }
    return sum(node.left) + node.value + sum(node.right);
}
```

6. Find the complexity of the following code which tests whether a give number is prime or not?

```
int isPrime(int n){
    if (n == 1){
        return 0;
    }

    for (int i = 2; i * i < n; i++) {
        if (n % i == 0)
            return 0;
    }
```

```
    return 1;
}
```

7. What is the time complexity of the following snippet of code?

```
int isPrime(int n){

    for (int i = 2; i * i < 10000; i++) {
        if (n % i == 0)
            return 0;
    }

    return 1;
}
isPrime();
```

URBAN EDGE

Operations on an Array

Following operations are supported by an array.

Traversal
Insertion
Deletion
Search

⇒ There can be many other operations one can perform on arrays as well:
eg: Sorting asc., Sorting desc.

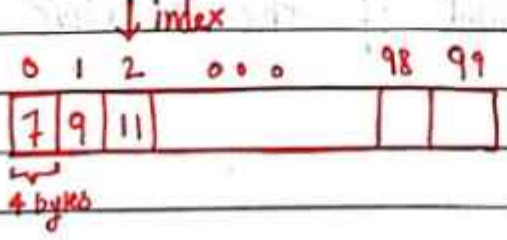
Traversal

Visiting every element of an array once → Traversal

Why traversal? → For use cases like:
- → Storing all elements → using scanf
- → Printing all elements → using printf

An important note about arrays
If we create an array of length 100 using a[100] in C language, we need not use all the elements. It is possible for a program to use just 60 elements out of these 100.
↳ But we cannot go beyond 100 elements.

An array can easily be traversed using a for loop in C language

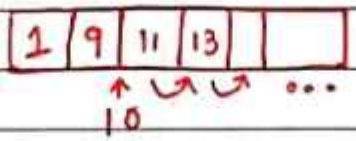


URBAN EDGE

Insertion

An element can be inserted in an array at a specified position.

In order for this operation to be successful, the array should have enough capacity

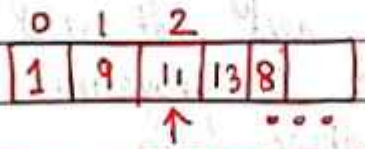


⇒ Elements need to be shifted to maintain relative order.

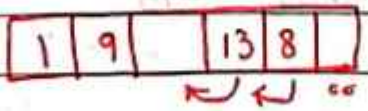
When no position is specified its best to insert the element at the end.

Deletion

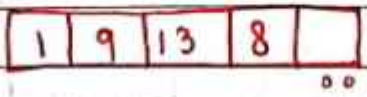
An element at specified position can be deleted creating a void which needs to be fixed by shifting all the elements to the left as follows:



Delete 11 at ind 2



Shift the elements



Deletion done!

We can also bring the last element of the array to fill the void if the relative ordering is not important

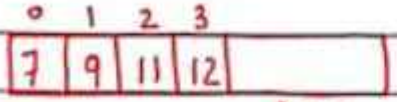
URBAN EDGE

Searching

Searching can be done by traversing the array until the element to be searched is found

⇓

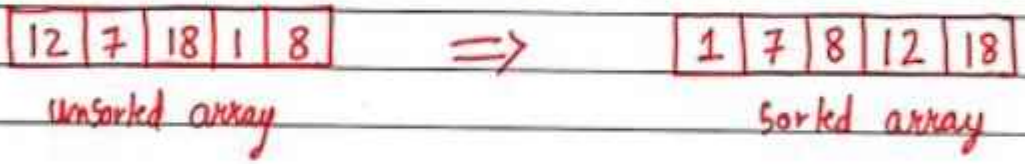
for sorted array time taken to search is much less than unsorted array!!



Sorting

Sorting means arranging an array in order (asc or desc)

We will see various sorting techniques later in the course.

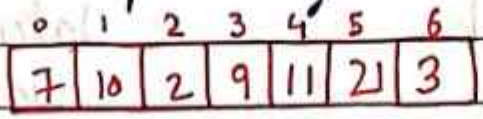


URBAN EDGE

Linear Vs Binary Search

Linear Search

Searches for an element by visiting all the elements sequentially until the element is found.

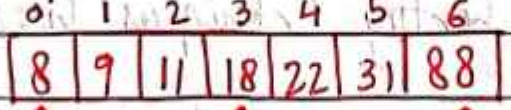


⇒ can be sorted or unsorted

Search '2' → Element found WC Complexity : O(n)

Binary Search

Searches for an element by breaking the search space into half in a sorted array.



WC Complexity O(log n)

Search 18

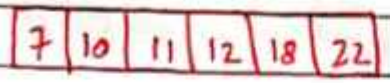
The search continues towards either side of mid based on whether the element to be searched is lesser or greater than mid.

	Linear Search	Binary Search
1.	Works on both sorted and unsorted arrays	Works only on sorted arrays
2.	Equality operations	inequality operations
3.	O(n) WC Complexity	O(log n) WC Complexity

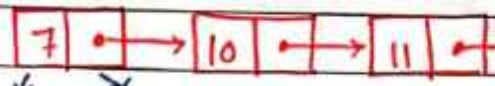
URBAN EDGE

Introduction to Linked Lists

Linked lists are similar to arrays (Linear data structures)



⇒ In Arrays elements are stored in contiguous memory locations



⇒ In linked lists, elements are stored in non contiguous memory locations

Why Linked Lists?

Memory and the capacity of an array remains fixed.
In case of linked lists, we can keep adding and removing elements without any capacity constraints

Drawbacks of Linked lists

→ Extra memory space for pointers is required (for every node 1 pointer is needed)
→ Random access not allowed as elements are not stored in contiguous memory locations.

Implementation

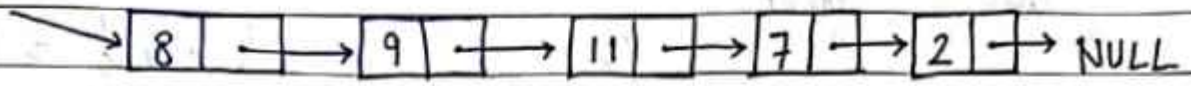
Linked list can be implemented using a structure in C language

```
Struct Node {
    int data;
    Struct Node * next;
};
```

⇒ Self referencing structure

Deletion in a Linked List

Consider the following Linked List

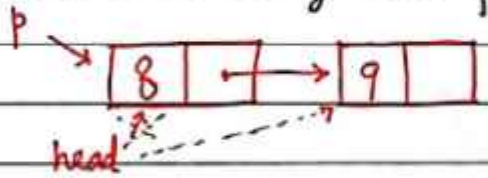


Deletion can be done for the following cases:

1) Deleting the first Node
2) Deleting the node at an index
3) Deleting the last Node
4) Deleting the first node with a given value.

The deletion just like insertion is done by rewiring the pointer connections, the only caveat being: we need to free the memory of the deleted node using free().

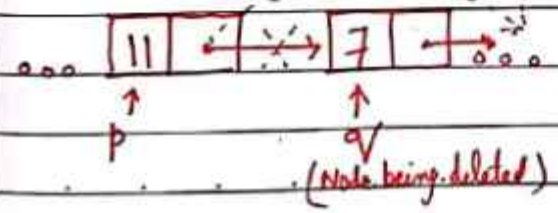
Case 1: Deleting the first node




```
Struct Node * p = head
head = head → next;
free(p);
```

Case 2: Deleting the node at an index

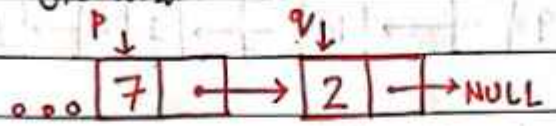
for deleting a given node, we first bring a temporary pointer p before element to be deleted and q on the element being deleted




```
p → next = q → next;
free(q);
```

URBAN EDGE

Case 3: Deleting the last Node
Last node can be deleted just like Case 2 by bringing p on second last element and q on last element.



p→next = NULL
free(q) → To free the memory!

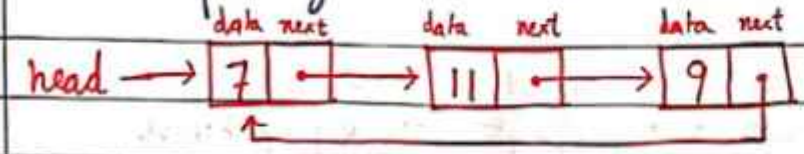
Case 4: Delete the first node with a given value
This can be done exactly like Case 2 by bringing pointers p & q to appropriate positions

p→next = q→next;
free(q);

URBAN EDGE

Circular Linked List

A circular linked list is a linked list where the last element points to the first element (head) hence forming a circular chain



Operations on a Circular Linked List

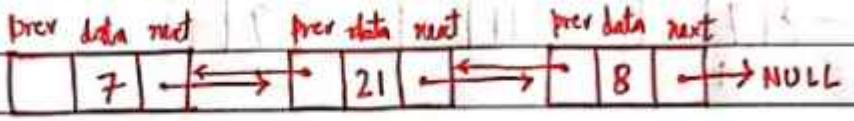
operations on a circular linked lists can be performed exactly like a Singly linked list.

Visit www.codewithharry.com for practice sets / code / more

URBAN EDGE

Doubly Linked List

In a doubly linked list, each node contains a data part along with the two addresses, one for the previous node and the other one for the next node



Implementation

A doubly linked list can be implemented in C language as follows:

```
struct Node {
    int data;
    struct Node * next;
    struct Node * prev;
};
```

Operations on a Doubly Linked List

The insertion and deletion on a Doubly linked list can be performed by rewiring pointer connections just like we saw in a Singly linked list.

The difference here lies in the fact that we need to adjust two pointers (prev & next) instead of one (next) in the case of a Doubly linked list.

URBAN EDGE

Introduction to Stack Data Structure

Stack is a linear data structure. Operations on stack are performed in LIFO (last in first out) order.

⇒ Item 2 which entered the basket last will be the first one to come out

LIFO (Last in First out)

Applications of Stack

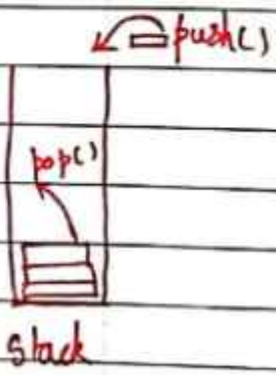
1. Used in function calls
2. Infix to postfix conversion (and other similar conversions)
3. Parenthesis matching & more...

Stack ADT

In order to create a stack we need a pointer to the topmost element along with other elements which are stored inside the stack.

Some of the operations of Stack ADT are:

1. push() → push an element into the Stack
2. pop() → remove the topmost element from the Stack
3. peek(index) → Value at a given position is returned
4. isEmpty()/isFull() → Determine whether the stack is empty or full.



URBAN EDGE

Implementation

A Stack is a collection of elements with certain operations following LIFO (Last in First out) diciplins.

A Stack can be implemented using an array or a linked list