Content

|  |  |
| --- | --- |
| **Chapter** | **Page no.** |
| Introduction |  |
| Time and space complexity |  |
| Arrays |  |
| Searching and Sorting |  |
| 2D array |  |
| Strings |  |
| Recursion |  |
| Backtracking |  |
| Bit manipulation |  |
| LinkedList |  |
| Stack |  |
| Queues |  |
| Greedy algoritham |  |
| Binary Trees |  |
| Binary search tree |  |
| Heaps |  |
| Hashing |  |
| Tree |  |
| Tries |  |
| Graph |  |
| Dynamic Programming |  |
| Segment trees |  |

*https://chatgpt.com/share/66e527ea-62f4-800f-99d0-2f03da3c3a3a*

|  |  |
| --- | --- |
| Chapters | Page.No. |
| Inroduction |  |
| Time and space complexity |  |
| Arrays-  >2D array  >Arraylist |  |
| string |  |
| Linkedlist |  |
| Stack |  |
| Queues |  |
| Shorting algorithams |  |
| Divide and Conqure |  |
| Greedy Algorithms |  |
| Recursion |  |
| Backtracking |  |
| Trees-  >Binary Trees  >Binary Search Trees |  |
| Heaps |  |
| Hashing |  |
| Graph |  |
| Dynamic Programming |  |
| Tries |  |
| Segment trees and Mathematic algorithams |  |

introduction

Data Structures and Algorithms

Data Structures and Algorithms (DSA) is a fundamental part of Computer Science that teaches you how to think and solve complex problems systematically.

Using the right data structure and algorithm makes your program run faster, especially when working with lots of data.

[What is Data Structure?](https://www.geeksforgeeks.org/learn-data-structures-and-algorithms-dsa-tutorial/)

A data structure is defined as a particular way of storing and organizing data in our devices to use the data efficiently and effectively. The main idea behind using data structures is to minimize the time and space complexities. An efficient data structure takes minimum memory space and requires minimum time to execute the data.

[What is Algorithm?](https://www.geeksforgeeks.org/learn-data-structures-and-algorithms-dsa-tutorial/)

Algorithm is defined as a process or set of well-defined instructions that are typically used to solve a particular group of problems or perform a specific type of calculation. To explain in simpler terms, it is a set of operations performed in a step-by-step manner to execute a task.

Where is Data Structures and Algorithms Needed?

Data Structures and Algorithms (DSA) are used in virtually every software system, from operating systems to web applications:

* For managing large amounts of data, such as in a social network or a search engine.
* For scheduling tasks, to decide which task a computer should do first.
* For planning routes, like in a GPS system to find the shortest path from A to B.
* For optimizing processes, such as arranging tasks so they can be completed as quickly as possible.
* For solving complex problems: From finding the best way to pack a truck to making a computer 'learn' from data.
* Operating Systems, Database Systems, Web Applications, Machine Learning, Video Games, Cryptographic Systems, Data Analysis, Search Engines and many other fields.

[Time and Space Complexities](https://www.geeksforgeeks.org/learn-data-structures-and-algorithms-dsa-tutorial/?ref=lbp)

**How can we measure which algorithm is best:**

Assume Algorithm X and Algorithm Y both perform the same task, but Algorithm X executes in less time than Algorithm Y. Therefore, Algorithm X is better in terms of time complexity. However, an algorithm may run faster on better hardware and slower on older hardware, so we cannot determine the complexity of an algorithm based on the hardware. Thus, we need a standard way to measure the complexity of an algorithm, which we will discuss later.

**Why efficiency of an algorithm is important:**

Assume that there is a food ordering application that takes almost 10 seconds to open and occupies 16MB of space on my device. On the other hand, another food ordering application is available which takes only 2 seconds to open and occupies only 2MB of space on my device. In my opinion, I would prefer to use the second application. Similarly, users also want to choose better software that works perfectly with minimum resources. On a large scale, it is very bad for any company, for example, the Swiggy app is laggy and provides a bad user experience, causing customers to switch to Zomato. That’s why we need better algorithms.

The primary motive to use DSA is to solve a problem effectively and efficiently. How can you decide if a program written by you is efficient or not? This is measured by complexities. Complexity is of two types:

[**Time Complexity**](https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/)**:** Time complexity is used to measure the amount of time required to execute the code.

[**Space Complexity**](https://www.geeksforgeeks.org/g-fact-86/)**:** Space complexity means the amount of space required to execute successfully the functionalities of the code.

Both of the above complexities are measured with respect to the input parameters. But here arises a problem. The time required for executing a code depends on several factors, such as:

* The number of operations performed in the program,
* The speed of the device, and also
* The speed of data transfer if being executed on an online platform.

So how can we determine which one is efficient? The answer is the use of asymptotic notation.

[**Asymptotic notation**](https://www.geeksforgeeks.org/analysis-of-algorithms-set-3asymptotic-notations/)**:-** It is a mathematical tool that calculates the required time in terms of input size and does not require the execution of the code*.* It neglects the system-dependent constants and is related to only the number of modular operations being performed in the whole program. The following 3 asymptotic notations are mostly used to represent the time complexity of algorithms:

* **Big-O Notation (Ο)** – Big-O notation specifically describes the worst-case scenario.
* **Omega Notation (Ω)** – Omega(Ω) notation specifically describes the best-case scenario.
* **Theta Notation (θ)** – This notation represents the average complexity of an algorithm.

**Rate of Growth of Algorithms**

Rate of growth is defined as the rate at which the running time of the algorithm is increased when the input size is increased.

**Algorithm1:**

Input size:- 1 2 3 4 5

Groth of rate:- 100 110 120 130 140

**Algorithm2:**

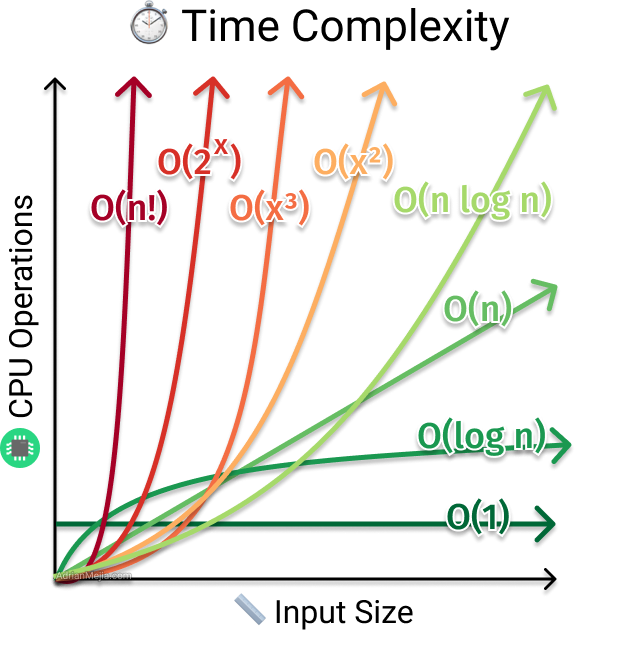
Input size:- 1 2 3 4 5

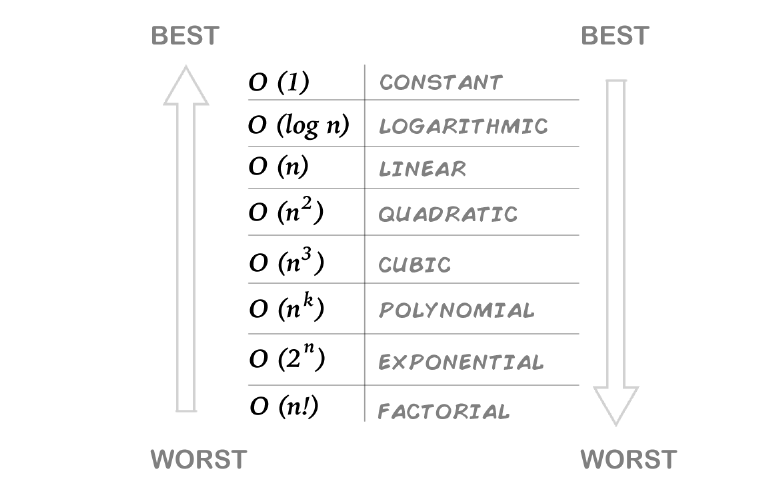
Groth of rate:- 20 60 100 140 180

We can see that the rate of growth of the complexity of Algorithm 1 increases by 10 units, while Algorithm 2 increases by 40 units. Therefore, we can say that Algorithm 2 is very costly for large inputs.

The most used notation in the analysis of a code is the [Big O Notation](https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/) which gives an upper bound of the running time of the code (or the amount of memory used in terms of input size).

*https://www.desmos.com/calculator*





class T and C {

    public static void main(String[] args) {

        System.out.print("Hello World");

    }

}

**Time Complexity:**In the above code “Hello World” is printed only once on the screen.   
So, the time complexity is **constant: O(1)** i.e. every time a constant amount of time is required to execute code.

class T and C {

    public static void main(String[] args) {

        int i, n = 8;

        for (i = 1; i <= n; i++) {

            System.out.printf("Hello World !!!\n");

        }

    }

}

**Time Complexity:**In the above code “Hello World !!!” is printed only **n times** on the screen, as the value of n can change. So, the time complexity is **linear: O(n)**

class T and C {

    public static void main(String[] args) {

        int i, n = 8;

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

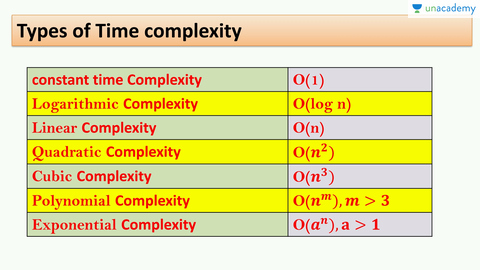
            System.out.printf("Hello World !!!\n");

        }

    }

}

**Time Complexity:** O(log2(n))



Matrix multiplication use cubic time complexity.

All the basic arithmetic operations (addition, subtraction, multiplication, division, and comparison) can be done in **polynomial time.**

The recursive Fibonacci sequence is a good example of **exponential time complexity.**

 for (int i = 0; i < n; i = i + c) {

            // If n=10, c=2

            // Output: 0 2 4 6 8

            // Time complexity = O(n)

        }

for (int i = n; i > 0; i = i - c) {

            // If n =10, c=2

            // Output: 10 8 6 4 2

            // Time coplexity = O(n)

        }

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

            // If n=33, c= 2

            // Output: 1 2 4 6 8 16 32 , 1,c,c^2, c^3-------c^k-1

            // c^k-1<n

            // k-1= log n

            // k = log n+1

            // Time complexity = O(log n)

        }

for (int i = n; i > 1; i = i / c) {

            // If n = 32, c=2

            // Output: 32 16 8 4 2

            // Time complexity = O(log n)

        }

for (int i = 2; i < n; i = pow(i, c)) {

            // n=32, c=2

            // Output: 2, 2^c 2c^2-----2c^k-1

            // 2^c^k-1<n

            // c^k-1 < log2 n

            // k-1 < logc.log2 n

            // k < logc.log2 n+1

            // Time complexity = O(log.logn)

        }

 void fun(int n) {

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

            // Some Work:

        }

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

            // Some Work:

        }

        for (int i = 1; i < 100; i++) {// O(1)

            // Some Work:

        }

        // Time complexity= O(n)+O(logn)+O(1)

        // Consider leading term(Higest term) = O(n)

    }

  void fun(int n) {

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

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

                // Some Work:

            }

        }

    }

    // Time complexity = O(n)\*O(logn) = O(n.logn)

void fun(int n) {

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

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

            // Some Work:

        }

    }

    for (int i = 0; i < n; i++) { // O(n^2)

        for (int j = 1; i < n; j++) {

            // Some Work:

        }

    }

    // Time complexity = O(nlogn)+O(n^2)

    // Consider higest term = O(n^2)

}

**#How to write a recursive relation of a function:-**

 static void fun(int n) {

        if (n <= 0)// O(1)

            return;

        System.out.println("Hi");

        fun(n / 2);// T(n/2)

        fun(n / 2);// T(n/2)

    }

**Explanation:-** T(n) = T(n/2)+T(n/2)+O(1) when{n>0}

2T(n/2)+O(1)

T(n)=O(1) when{n<=0}

static void fun(int n) {

        if (n <= 0)// O(1)

            return;

        for (int i = 0; i < n; i++) {

            System.out.println("Hi");

            fun(n / 2);// T(n/2)

            fun(n / 3);// T(n/3)

        }

    }

**Explanation:-** T(n) = T(n/2)+T(n/3)+O(1) when{n>0}

T(n)=O(1) when{n<=0}

static void fun(int n) {

        if (n <= 1)// O(1)

            return;

        System.out.println("Hi");

        fun(n - 1);// T(n-1)

    }

**Explanation:-** T(n) = T(n-1)+O(1) when{n>0}

T(n)=O(1) when{n<=0}

**#How to solve recursive relations using recursive tree method:-**

* 1. We consider the recursive tree and compute total work done.
  2. We write non-recursive (or function) part as root of the tree and write the recursive part as children.
  3. W keep expending until we see a pattern.

**T(n) = 2T(n/2) + n**

Recursive part Non-recursive part

**(1)T(n) = T(n/2)+T(n/2)+n2**

n2

(n/2)2 (n/2)2 n2/4+n2/4 = 2n2/4 = n2/2

(n/4)2  (n/4)2  (n/4)2  (n/4)2 4\*n2/16 = n2/4

(n/8)2  (n/8)2  (n/8)2  (n/8)2  (n/8)2  (n/8)2  (n/8)2  (n/8)2

8\*n2/64 = n2/8

**Time\_Complexity of this recursive function:**

= n2+n2/2+n2/4+n2/8+n2/16…..

= n2[ 1 + ½ +1/4 + 1/8 + 1/16…….G.P.

= n2[1/1-1/2] – constant

= T.C = O(n)

**Height of Tree is:**

Level-1: (n2/20)2

Level-2: (n2/21)2

Level-3: (n2/22)2

Level-4: (n2/23)2……………(n2/2k)2

n/2k = 1

2k = n

**K = logn**

**(2)T(n) = 2T(n-1)+c**  c c

(n-1) (n-1) 2C

(n-2) (n-2) (n-2) (n-2) 4C

**Time complexity of this recursive function:**

= c+2c+4C…………

=c+(1+2+4……….

=c [ (1\*(2n-1)/2-1]

=O(2n)

**#Upper bound using recursion tree method:**

When one side height of tree is shorter and other side height of tree is longer in that case we use upper bound concept and find maximum height of tree.

When every level of recursive tree work done or cost is same in that case for finding the time complexity of that tree find max height of tree and multiply with cost of tree.

**(3)T(n) = T(n/3)+T(2n/3)+cn**

n cn

n/3 2n/3 cn

n/9 2n/9 2n/9 4n/9 cn

n/27 2n/27 2n/27 4n/27 2n/27 4n/27 4n/29 8n/27 cn

**Height of this recursive tree:-**

(n/3k) (n/3/2)k = 1

**K= log3n k = log3/2 n**

**Time complexity of this recursive tree:-**

Total cost of tree \* Max height of tree

**cn \* log3/2 n = O(n logn)**

**Space Complexity**

Space complexity and Auxiliary space are two of the most often confused and interchangeably used terms when talking about the space complexity of a certain algorithm:

* **Auxiliary Space:**The extra space that is taken by an algorithm temporarily to finish its work
* **Space Complexity:**Space complexity is the total space taken by the algorithm with respect to the input size plus the auxiliary space that the algorithm uses. Order of growth of memory (or Ram) space in term of input size.

**@How to calculate Space Complexity for the code?**

**Input space + auxiliary space**

 int getSum(int n) {

        return n \* (n + 1) / 2;

    }

   int getSum2(int n){

    int sum = 0;

    for(int i=1; i<=n; i++){

        sum = sum +i;

        return sum;

    }

   }

When we use only some variable that consider as constant and when we use array, map or list that can be depend on input value.

Here both above program use constant space so that it’s required only O(1) time complexity.

   int arrSum(int arr[], int n){

    int sum = 0;

    for(int i=0; i<n; i++){

        sum= sum+arr[i];

        return sum;

    }

   }

Here if n (input) is 10 so required only 10 space but it input array size is 1000 or more in that case we need 1000 space means it’s depends on input value so total time complexity of above program is O(n).

**Auxiliary space:-** Order of growth of extra space or temporary space in terms of input size.

 int arrSum(int[], int n){

        int sum= 0;

        for(int i=0; i<n; i++){

            sum = sum+arr[i];

            return sum;

        }

    }

Now , the only variable contributing to auxiliary space complexity which required constand space. Therefore the auxiliary space complexity of the code is O(1) and total space complexity is O(n).

int fun(int n){

    if(n<=0)

        return 0;

        return n+fun(n-1);

   }

This code is use stack as extra space for function call if input value is large so more space required means the space complexity and auxiliary space complexity both is O(n) of this code.

 int fib(int n){

    if(n==0||n==1)

        return n;

    return fib(n-1)+fib(n-2);

   }

This code is also required O(n) space complexity because it’s use stack call as extra space.

 int fib(int n){

    if(n==0|| n==1)

        return n;

        int a = 0; b= 1;

        for (int i = 2; i<= n; i++){

            c = a+b;

            a = b;

            b = c;

            return c;

        }

  }

Here all the variable have constant space means space complexity is O(1)

**Note:** It’s necessary to mention that space complexity depends on a variety of things such as the programming language, the compiler, or even the machine running the algorithm.

arrays

An array is a container object that holds a fixed number of values of a single type.We do not need to create different variables to store many values, instead we store them in different indices of same objects and refer them by these indices whenever we need to call them.

There are three main ways to create an array in java.

int [] marks = new int[5];

int [] marks;

        marks = new int[5];

        marks[1] = 60;

        marks[2] = 70;

        marks[3] = 90;

        marks[4] = 86;

int [] marks = {98, 45, 79, 99, 80};

**Taking array as function argument:**

public class ArrayDemo {

    public static void update(int marks[]) {

        for (int i = 0; i < marks.length; i++) {

            marks[i] = marks[i] + 1;

        }

    }

    public static void main(String[] args) {

        int marks[] = { 45, 23, 49 };

        update(marks);

        // print our marks

        for (int i = 0; i < marks.length; i++) {

            System.out.println(marks[i] + " ");

        }

        System.out.println();

    }

}

**Operations on Arrays:**

**Example1:**

public class OperationsOnArray {

    static int insert(int arr[], int n, int x, int capacity, int position) {

        // Check if array is full

        if (n == capacity)

            return n;

        // Shift elements to make space for insertion

        int idx = position - 1;

        for (int i = n - 1; i >= idx; i--) {

            arr[i + 1] = arr[i];

        }

        // Insert element at specified position

        arr[idx] = x;

        // Increment the size of the array

        return n + 1;

    }

    public static void main(String[] args) {

        // Initialize array and variables

        int arr[] = new int[5];

        int capacity = 5;

        int n = 3; // Current size of array

        // Populate initial elements

        arr[0] = 5;

        arr[1] = 10;

        arr[2] = 20;

        // Print array before insertion

        System.out.println("Before Insertion");

        for (int i = 0; i < n; i++) {

            System.out.print(arr[i] + " ");

        }

        System.out.println();

        // Insert new element

        int x = 7;

        int position = 2;

        n = insert(arr, n, x, capacity, position);

        // Print array after insertion

        System.out.println("After Insertion");

        for (int i = 0; i < n; i++) {

            System.out.print(arr[i] + " ");

        }

    }

}

**Example2:**

class OperationsOnArray {

    static int deleteElement(int[] arr, int n, int x) {

        int i = 0;

        // Find the index of the element to delete

        for (i = 0; i < n; i++) {

            if (arr[i] == x)

                break;

        }

        // If element not found, return current size

        if (i == n)

            return n;

        // Shift elements to the left to fill the gap

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

            arr[j] = arr[j + 1];

        }

        // Decrement the size of the array

        return n - 1;

    }

    public static void main(String args[]) {

        int arr[] = { 3, 8, 12, 5, 6 };

        int x = 12;

        int n = 5;

        System.out.println("Before Deletion");

        for (int i = 0; i < n; i++) {

            System.out.print(arr[i] + " ");

        }

        System.out.println();

        n = deleteElement(arr, n, x);

        System.out.println("After Deletion");

        for (int i = 0; i < n; i++) {

            System.out.print(arr[i] + " ");

        }

    }

}

**Example3:**

class OperationsOnArray {

    static int search(int arr[], int n, int x) {

        for (int i = 0; i < n; i++) {

            if (arr[i] == x)

                return i;

        }

        return -1;

    }

    public static void main(String args[]) {

        int arr[] = { 20, 5, 7, 25 }, x = 5;

        System.out.println(search(arr, arr.length, x));

    }

}

**What is the time complexity for performing basic operations in an array?**

The Time Complexity of different operations in an array is: For analyzing the real-time complexity you also have to consider the time in bringing the block of memory from an external device to RAM which takes O(√N) time.

| ARRAY OPERATION | REAL TIME COMPLEXITY | ASSUMED TIME COMPLEXITY |
| --- | --- | --- |
| Accessing the i-th element. | O(√N) | O(1) |
| Traversing all elements. | O(N + √N) | O(N) |
| Override element at i-th index. | O(√N) | O(1) |
| Insert an element. | O(N + √N) | O(N) |
| Delete an element. | O(N + √N) | O(N) |

**Applying Questions:**

public class ArraysDemo {

    public static int getLargest(int numbers[]) {

        int largest = Integer.MIN\_VALUE;

        int smallest = Integer.MAX\_VALUE;

        for (int i = 0; i < numbers.length; i++) {

            if (largest < numbers[i]) {

                largest = numbers[i];

            }

            if (smallest > numbers[i]) {

                smallest = numbers[i];

            }

        }

        System.out.println("The smallest value is:" + smallest);

        return largest;

    }

    public static void main(String[] args) {

        int numbers[] = { 1, 2, 3, 4, 5, 6 };

        System.out.println("The largest value is:" + getLargest(numbers));

    }

}

Output:

The smallest value is:1

The largest value is:6

class Basic{

    public static int findMaxIndex(int arr[]) {

        int result = 0; // Start with the first element as the maximum

        for (int i = 1; i < arr.length; i++) {

            if (arr[i] > arr[result]) {

                result = i; // Update res if the current element is greater

            }

        }

        return result; // Return the index of the maximum value

    }

    public static int findMinIndex(int arr[]){

        int res = 0;

        for(int j = 1; j<arr.length; j++){

            if(arr[j]<arr[res]){

                res = j;

            }

        }

        return res;

    }

    public static void main(String[] args) {

        int[] arr = { 3, 2, 7, 6, 5 };

        int result = findMaxIndex(arr);

        System.out.println("Index of maximum value: " + result+ " and the value is "+ arr[result]);

        int res = findMinIndex(arr);

        System.out.println(res+" "+ arr[res]);

    }

}

Output:

Index of maximum value: 2 and the value is 7

1 2

class ArraysDemo {

    static void reverse(int arr[]) {

        int low = 0, high = arr.length - 1;

        while (low < high) {

            int temp = arr[low];

            arr[low] = arr[high];

            arr[high] = temp;

            low++;

            high--;

        }

    }

    public static void main(String args[]) {

        int arr[] = { 10, 5, 7, 30 };

        reverse(arr);

        System.out.println("After Reverse");

        for (int i = 0; i < arr.length; i++) {

            System.out.print(arr[i] + " ");

        }

    }

}Output:

After Reverse

30 7 5 10

**Types of Array:**

1. Fixed sized arrays. { as we saw above}
2. Dynamic sized arrays. { Can implement using ArrayList}

**ArrayList:**

Java ArrayList class uses a dynamic array for storing the elements. It is like an array, but there is no size limit. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the **java.util package.** It is like the **Vector** in C++.

In Java, we use the ArrayList class to implement the functionality of **resizable-arrays**. It implements the [List](https://www.programiz.com/java-programming/list) interface of the [collections framework](https://www.programiz.com/java-programming/collections).

**ArrayList Vs Array**

In Java, we need to declare the size of an [array](https://www.programiz.com/java-programming/arrays) before we can use it. Once the size of an array is declared, it's hard to change it.

To handle this issue, we can use the ArrayList class. It allows us to create resizable arrays.

Unlike arrays, arraylists can automatically adjust their capacity when we add or remove elements from them. Hence, arraylists are also known as **dynamic arrays**.

**Methods of arraylist:**

**add(object):-**To add a single element to the arraylist, we use the add() method of the ArrayList class.

**add(index, object):-** To add element on particular index.

**get(index):-**To access an element from the arraylist, we use the get() method of the ArrayList class.

**set(index , object):-**To change elements of the arraylist, we use the set() method of the ArrayList class.

**remove(object):-**To remove an element from the arraylist, we can use the remove() method of the ArrayList class.

**remove(index):-** To remove element of particular index.

**contains(object):-** To check particular element is available in arraylist or not.

**indexOf(object):-** To check index number of particular object.

**lastIndexOf(object):-** To check index number of last element of the arraylist.

**Clear():-** To remove or clear all the element of arraylist and make it empty.

**isEmpty():-** To check is arraylist empty or not.

**Time Complexity of Arraylist methods:**

|  |  |
| --- | --- |
| **Methods** | **Time complexity** |
|  |  |
| add() | O(1) |
| size() | O(1) |
| isEmpty() | O(1) |
| get() | O(1) |
| set() | O(1) |
| contains() | O(n) |
| indexOf() | O(n) |
| lastIndexOf() | O(n) |
| remove(index) | O(n) |
| remove(object) | O(n) |
| add(index, object) | O(n) |
|  |  |

**Example:**

import java.util.ArrayList;

class ArrayListDemo {

    public static void main(String[] args) {

        ArrayList<String> al = new ArrayList<String>();

        al.add("My");

        al.add("Name");

        al.add("is");

        System.out.println(al);

        al.add(1, "practice");

        System.out.println(al);

        System.out.println(al.contains("Name"));

        al.remove(1);

        System.out.println(al.contains("Name"));

        al.remove("is");

        System.out.println(al.contains("is xyz"));

}

}

Output:

[My, Name, is]

[My, practice, Name, is]

true

true

false

import java.util.ArrayList;

class ArrayListDemo {

    public static void main(String[] args) {

        ArrayList<Integer> In = new ArrayList<>();

        In.add(10);

        In.add(20);

        In.add(10);

        In.add(30);

        System.out.println(In.isEmpty());

        System.out.println(In.isEmpty());

        System.out.println(In.get(1));

        In.set(1, 40);

        System.out.println(In.get(1));

        System.out.println(In.indexOf(10));

        System.out.println(In.lastIndexOf(10));

        System.out.println(In.indexOf(50));

        In.clear();

    }

}

Output:

false

false

20

40

0

2

-1

**Traversal on ArrayList:**

package ArrayList;

import java.util.ArrayList;

public class arraylistDemo {

    public static void main(String[] args) {

        ArrayList<Integer> list = new ArrayList<>();

        list.add(2);

        list.add(3);

        list.add(4);

        list.add(5);

        list.add(6);

        list.add(7);

        list.add(8);

        System.out.println(list.size());

        System.out.println("---------------");

        // using for each loop:

        for (Integer x : list)

            System.out.println(x);

        System.out.println("-----------------");

        // size() method is use for find the size of arraylist:

        // using size() method

        for (int i = 0; i < list.size(); i++) { // printing values:

            System.out.print(list.get(i) + " ");

        }

        System.out.println();

        // Reverse of an arraylist:

        for (int i = list.size() - 1; i >= 0; i--) {

            System.out.print(list.get(i) + " ");

        }

        System.out.println();

    }

}

Output:

7

---------------

2

3

4

5

6

7

8

-----------------

2 3 4 5 6 7 8

8 7 6 5 4 3 2

package ArrayList;

import java.util.ArrayList;

import java.util.Collections;

public class arraylistPractice {

    public static void swap(ArrayList<Integer> list, int idx1, int idx2) {

        int temp = list.get(idx1);

        list.set(idx1, list.get(idx2));

        list.set(idx2, temp);

    }

    public static void main(String[] args) {

        ArrayList<Integer> list = new ArrayList<>();

        list.add(1);

        list.add(2);

        list.add(3);

        list.add(32);

        list.add(31);

        // SWAP TO ELEMENTS IN ARRAYLIST:

        int idx1 = 1, idx2 = 3;

        System.out.println("Befor swaping:");

        System.out.println(list);

        swap(list, idx1, idx2);

        System.out.println("After swaping:");

        System.out.println(list);

        // SORTING IN ARRAYLIST: Accending order.

        System.out.println("Sorting in accending order:");

        Collections.sort(list);

        System.out.println(list);

        // DESCENDING ORDER.

        System.out.println("Sorting in descending order:");

        Collections.sort(list, Collections.reverseOrder());

        System.out.println(list);

        // MAXIMAM OF AN ARRALIST:

        int maximum = 0;

        for (int i = 0; i < list.size(); i++) {

            if (maximum < list.get(i)) {

                maximum = list.get(i);

            }

            // WE CAN ALSO USE THIS:

            // maximum = Math.max(maximum, list.get(i));

        }

        System.out.println("The maximum element is:" + maximum);

    }

}

Output:

Befor swaping:

[1, 2, 3, 32, 31]

After swaping:

[1, 32, 3, 2, 31]

Sorting in accending order:

[1, 2, 3, 31, 32]

Sorting in descending order:

[32, 31, 3, 2, 1]

The maximum element is:32

Searching and sortings

**Array.sort():-** The sort() method sorts an array in ascending order. This method sorts arrays of strings alphabetically, and arrays of integers numerically.

**Syntax**

*Arrays.sort(array)*

*Arrays.sort(array, start, end)*

**Parameter Values**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| *array* | Required. The array to be sorted |
| *start* | Optional. The index position of the first element (inclusive) to be sorted |
| *end* | Optional. The index position of the last element (exclusive) to be sorted |

**Technical Details**

***Return value: NA***

***Time Complexity:****O(N log N)****Auxiliary Space:****O(1)*

import java.util.Arrays;

public class BinarySearch {

    public static void main(String[] args) {

        int[] arr1 = { 5, 20, 12, 30 };

        char[] arr2 = { 'B', 'B', 'A', 'C', 'A' };

        int[] myNum = { 50, 10, 25, 1, 17, 99, 33 };

        Arrays.sort(arr1);

        System.out.println(Arrays.toString(arr1));

        Arrays.sort(arr2);

        System.out.println(Arrays.toString(arr2));

        // It's sort only index 1 to 4

        Arrays.sort(myNum, 1, 4);

        System.out.println(Arrays.toString(myNum));

    }

}

Output:

[5, 12, 20, 30]

[A, A, B, B, C]

[50, 1, 10, 25, 17, 99, 33]

**Collections.sort:-** Collections allow sorting which is implements by list interface.It is work for only non-primitive data types**.** When we sorting non primitive data type, should use stable sorting algorithms

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class BinarySearch {

    public static void main(String[] args) {

        List<Integer> list = new ArrayList<Integer>();

        list.add(10);

        list.add(5);

        list.add(20);

        Collections.sort(list);

        System.out.println(list);

        // Sorting the list in reverse order

        Collections.sort(list, Collections.reverseOrder());

        System.out.println(list);

    }

}

Output:

[5, 10, 20]

[20, 10, 5]

**Learn more about stable and unstable sorting algorithm:** *https://www.educative.io/answers/stable-and-unstable-sorting-algorithms*

2d array

**Multidimensional Arrays**

A multidimensional array is an array of arrays. Multidimensional arrays are useful when you want to store data as a tabular form, like a table with rows and columns. To create a two-dimensional array, add each array within its own set of **curly braces**:

**Different way for creating a 2D array**

// Different way for creating 2d array

class str {

   public static void main(String[] args) {

int arr[][] = new int [3][4];

int arr1[][] = {{1,2,3,4},{2,4,6,8},{3,5,7,9}};

int arr2[][];

arr2  = new int[3][4];

}

}

**Some Basic Operations on matrix:**

import java.util.Scanner;

public class basicsMatrix {

    // Searchig in matrix:

    public static void Search(int matrix[][], int key) {

        boolean found = false;

        for (int i = 0; i < matrix.length; i++) { // Use < instead of <=

            for (int j = 0; j < matrix[0].length; j++) { // Use < instead of <=

                if (matrix[i][j] == key) {

                    System.out.println("Element found at cell (" + i + "," + j + ")");

                    // found = true;

                }

            }

        }

        if (!found) {

            System.out.println("Key is not found");

        }

    }

    public static void main(String args[]) {

        // Create a matrix

        int matrix[][] = new int[3][3];

        int n = matrix.length, m = matrix[0].length;

        // User input in 2D array

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter elements of the matrix:");

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < m; j++) {

                matrix[i][j] = sc.nextInt();

            }

        }

        // Printing of a matrix

        System.out.println("The matrix is:");

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < m; j++) {

                System.out.print(matrix[i][j] + "  ");

            }

            System.out.println();

        }

        System.out.println("Enter the key to search:");

        int key = sc.nextInt();

        Search(matrix, key);

        sc.close(); // Close the scanner to avoid resource leak

    }

}

Output:

Enter elements of the matrix:

1 2 3 4 5 6 7 8 9

The matrix is:

1 2 3

4 5 6

7 8 9

Enter the key to search:

5

Element found at cell (1,1)

Strings

Strings in java is a sequence of characters that Is enclosed in double quotes. Whenever java comes across a String literal in the code, it creates a string literal with the value of string.

**Ex:-**

import java.util.Scanner;

public class string {

    public static void main(String[] args) {

        System.out.println("Enter your string here: ");

        Scanner sc = new Scanner(System.in);

        String name = sc.nextLine();         // By taking input from user

        String Name = "sumit";               // inisilize on the spot

        System.out.println(name + "       " + Name);     // print both value by

                                                        // taking user and insially

                                                         // inisilized value.

    }

}

**>>Different way for creating a string in java.**

// Different way for creating a string in java.

public class Str {

    public static void main(String[] args) {

        String sc = new String();

        String str = "Java Program"; // created in pool

        System.out.println(str);

        String str2 = new String("JAVA"); // createdd in heap

        System.out.println(str2);

        char c[] = { 'H', 'e', 'l', 'l', 'o' };

        String str3 = new String(c);

        System.out.println(str3);

        byte b[] = { 65, 66, 67, 68 };

        String str4 = new String(b);

        System.out.println(str4);

    }

}

Output:

Java Program

JAVA

Hello

ABCD

**String Methods**

String methods operate on java strings. They can be used to find length of the string, convert to lowercase, etc.

Some of the commonly used string methods are:

1. **int.length()** – This method is returns the total length of a string. How many spaces used by string.

class Str {

    public static void main(String[] args) {

        String str = "  java  ";

        int a = str.length();

        System.out.println(a);

Outpur:

8

1. **string.toLowerCase()** – Returns a new string which has all the lowercase characters from the string name.

class Str {

    public static void main(String[] args) {

        String str1 = "JAVA";

        String b = str1.toLowerCase();

        System.out.println(b);

Output:

java

1. **string.toUpperCase()** – Returns a new string which has all the upeercase charaters from the string name.

class Str {

    public static void main(String[] args) {

        String str = "  java  ";

        String c = str.toUpperCase();

        System.out.println(c);

Output:

JAVA

1. **string.trim()** – Returns a new string after removing all the leading and trailing spaces from the original string.

class Str {

    public static void main(String[] args) {

        String str = "  java  ";

        String d = str.trim();

        System.out.println(d);

Output:

java

1. **string.substring(int start)** – Returns a substring which start a specific index and go throw the last index

class Str {

    public static void main(String[] args) {

        String str1 = " Java\_Programming ”;

      String e = str.substring(3);

        System.out.println(e);

Output:

va\_Programming

1. **string.substring(int start, int end)** – Returns a substring from start index to the end index start index is included and end is excluded. Here start with index 3 and end on index 5.

class Str {

    public static void main(String[] args) {

        String str = "java\_Programming";

        String f = str.substring(3, 5);

        System.out.println(f);

1. **string.replace(char ‘s’ , ‘k’ )** – Returns a new string after replacing s with k. It’s replace all specific char with new char.

class Str {

    public static void main(String[] args) {

        String str = " java Programming ";

        String g = str.replace('a', 'c');

        System.out.println(g);

Output:

Jcvc Programming

1. **boolean.startsWith( string “su”)** – Returns true if name starts with “su” otherwise false. Here check is string start with a or not.

class Str {

    public static void main(String[] args) {

        String str = "  java  ";

        boolean p = str.startsWith("a");

Output:

false

1. **boolean.endsWith(string “ha”)** – Returns true if name ends with string “ ha” otherwise false.

class Str {

    public static void main(String[] args) {

        String str = "  java  ";

        boolean p = str.endsWith("a");

        System.out.println(p);

Output:

true

10. **char.charAt(int index)** – Returns character at a given index position .

class Str {

    public static void main(String[] args) {

        String str = " java ";

        char i = str.charAt(4);

        System.out.println(i);

Output:

v

11. **int.indexOf(string “m” )** – Returns the index of the given string.

class Str {

    public static void main(String[] args) {

        String str = " java ";

        int j = str.indexOf("a");

        System.out.println(j);

12. **int.indexOf(“s”,3)** – Returns the index of the given string starting from the index 3 (int). (otherwise return -1 in don’t match condition). If a specific char is start with given index so return true if it’s not match so return false.

class str{

   public static void main(String[] args) {

    String s = "Programming";

    int sc = s.indexOf("r", 0); //Here the compiler is check r on which index after inx 0

    System.out.println(sc);

   }

}

Output:

1

13.**int.lastIndexOf(“s”)**Returnsthelastindexofthegivenstring.The lastIndexOf() method returns the position of the last occurrence of specified character(s) in a string.

class str{

   public static void main(String[] args) {

    String s = "Programming";

    int sc = s.lastIndexOf("r");

    System.out.println(sc);

   }

}

Output:

1

14.**int.lastIndexOf(“s” ,2)** – Returns the last index of the given string before index 2.

class str{

   public static void main(String[] args) {

    String s = "Programming";

    int sc = s.lastIndexOf("r",3);

    System.out.println(sc);

   }

}

Outpur:

1

15.**boolean.equals(“java”)** – Returns true if the given string is equal to next string another otherwise false [ canse sensitive]

class str{

   public static void main(String[] args) {

    String str1 = "Programming";

    String str2 = "java";

    boolean q = str1.equals(str2);

    System.out.println(q);

   }

}

16.**boolean.equalsIgnoreCase(“sumit”)** – Returns true if two strings are equal ignoring the case of characters. The equalsIgnoreCase() method compares two strings, ignoring lower case and upper case differences.

class str{

   public static void main(String[] args) {

    String str1 = "JAVA";

    String str2 = "java";

    boolean q = str1.equalsIgnoreCase(str2);

    System.out.println(q);

   }

}

Output:

true

**17. compareTo()-**The compareTo() method compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The method returns 0 if the string is equal to the other string. A value less than 0 is returned if the string is less than the other string (less characters) and a value greater than 0 if the string is greater than the other string (more characters).

**Tip:** Use [compareToIgnoreCase()](https://www.w3schools.com/java/ref_string_comparetoignorecase.asp) to compare two strings lexicographyically, ignoring lower case and upper case differences.

**Tip:** Use the [equals()](https://www.w3schools.com/java/ref_string_equals.asp) method to compare two strings without consideration of Unicode values.

String myStr1 = "Hello";

String myStr2 = "Hello";

System.out.println(myStr1.compareTo(myStr2)); // Returns 0 because they are equal

**toString() -** The toString() method converts the object into a [string](https://www.programiz.com/java-programming/string) and returns it.

**StringBuffer class in Java**

StringBuffer is a class in Java that represents a mutable sequence of characters. It provides an alternative to the immutable String class, allowing you to modify the contents of a string without creating a new object every time.

**There are several advantages of using StringBuffer over regular String objects in Java:**

1. **Mutable:** StringBuffer objects are mutable, which means that you can modify the contents of the object after it has been created. In contrast, String objects are immutable, which means that you cannot change the contents of a String once it has been created.
2. **Efficient:** Because StringBuffer objects are mutable, they are more efficient than creating new String objects each time you need to modify a string. This is especially true if you need to modify a string multiple times, as each modification to a String object creates a new object and discards the old one.
3. **Thread-safe:** StringBuffer objects are thread-safe, which means multiple threads can access it simultaneously( they can be safely accessed and modified by multiple threads simultaneously). In contrast, String objects are not thread-safe, which means that you need to use synchronization if you want to access a String object from multiple threads.

**StringBuilder Class in Java**

StringBuilder in Java represents a mutable sequence of characters. Since the String Class in Java creates an immutable sequence of characters, the StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters. The function of StringBuilder is very much similar to the StringBuffer class, as both of them provide an alternative to String Class by making a mutable sequence of characters. However, the StringBuilder class differs from the StringBuffer class on the basis of synchronization. The StringBuilder class provides no guarantee of synchronization whereas the StringBuffer class does.

Where possible, it is recommended that this class be used in preference to StringBuffer as it will be faster under most implementations. Instances of StringBuilder are not safe for use by multiple threads. If such synchronization is required then it is recommended that StringBuffer be used. String Builder is not thread-safe and high in performance compared to String buffer.

**Note:** All the methods of String class is also work with String Builder and String Buffer Class but it have own some extra methos also.

**append():-**The append() method concatenates the given argument with this string.

**insert():-**The insert() method inserts the given string with this string at the given position.

**replace():-**The replace() method replaces the given string from the specified beginIndex and endIndex-1.

**delete():-**The delete() method of the StringBuffer class deletes the string from the specified beginIndex to endIndex-1.

**reverse():-**The reverse() method of the StringBuilder class reverses the current string.

**capacity():**

* The capacity() method of the StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of characters increases from its current capacity, it increases the capacity by (oldcapacity\*2)+2.
* For instance, if your current capacity is 16, it will be (16\*2)+2=34.

**StringBuffer Example:**

class StringDemo {

    public static void main(String[] args) {

        // append method

        StringBuffer sb = new StringBuffer("Hello ");

        sb.append("Java"); // now original string is changed

        System.out.println(sb);

        // insert method

        StringBuffer sb = new StringBuffer("Hello ");

        sb.insert(1, "Java");

        // Now original string is changed

        System.out.println(sb);

        // replace method

        StringBuffer sb = new StringBuffer("Hello");

        sb.replace(1, 3, "Java");

        System.out.println(sb);

        // delete method

        StringBuffer sb = new StringBuffer("Hello");

        sb.delete(1, 3);

        System.out.println(sb);

        // reverse method

        StringBuffer sb = new StringBuffer("Hello");

        sb.reverse();

        System.out.println(sb);

        // capacity method

        StringBuffer sb = new StringBuffer();

        System.out.println(sb.capacity()); // default 16

        sb.append("Hello");

        System.out.println(sb.capacity()); // now 16

        sb.append("java is my favourite language");

        System.out.println(sb.capacity());

        // Now (16\*2)+2=34 i.e (oldcapacity\*2)+2

        // set method

        // create a StringBuilder object

        // with a String pass as parameter

        StringBuilder str = new StringBuilder("WelcomeGeeks");

        // print string

        System.out.println("String = "

                + str.toString());

        // set char at index 2 to 'L'

        str.setCharAt(2, 'L');

        // print string

        System.out.println("After setCharAt() String = "

                + str.toString());

        // delete method

        StringBuffer sb = new StringBuffer("Hello");

        sb.deleteCharAt(1);

        System.out.println(sb);

    }

}

class stringDemo {

    // LOOPING IN JAVA:

    public static void printLetters(String str) {

        for (int i = 0; i < str.length(); i++) {

            System.out.print(str.charAt(i));

        }

        System.out.println();

    }

    public static void main(String[] args) {

        String str = "This is a string:";

        // STRING CONCATENATION:

        String firstName = "Sumit";

        String lastName = "Kushwaha";

        String fullname = firstName + " " + lastName;

        System.out.println(fullname);

        printLetters(fullname);

        // STRING COMPARATION:

        String sc = "HI";

        String sc1 = "HI";

        String sc2 = new String("HI");

        // IT'S CHECH EQUALITY ON OBJECT LEVEL

        if (sc == sc1) {

            System.out.println("It's equal");

        } else {

            System.out.println("It's not equal");

        }

        if (sc == sc2) {

            System.out.println("It's equal");

        } else {

            System.out.println("It's not equal");

        }

        // //CHECK EQUALITY ON VALUES LEVEL:

        if (sc.equals(sc2)) {

            System.out.println("It's equal");

        } else {

            System.out.println("NOt equal");

        }

        printLetters(str);

    }

}

dfghjk

REcursion

**What is Recursion?**   
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. Examples of such problems are [Towers of Hanoi (TOH)](https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/), [Inorder/Preorder/Postorder Tree Traversals](https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/), [DFS of Graph](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/), etc. A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems.

**Properties of Recursion:**

* Performing the same operations multiple times with different inputs.
* In every step, we try smaller inputs to make the problem smaller.
* Base condition is needed to stop the recursion otherwise infinite loop will occur.

**How are recursive functions stored in memory?**

Recursion uses more memory, because the recursive function adds to the stack with each recursive call, and keeps the values there until the call is finished. The recursive function uses LIFO (LAST IN FIRST OUT) Structure just like the stack data structure.

class Recuresion {

    static void fun1() {

        System.out.println("Hi");

        fun1();

    }

public static void main(String[] args) {

        fun1();

    }

}

class Recuresion {

    static void fun1(int n) {

        if (n == 0)

            return;

        System.out.println("Hello");

       fun1(n - 1);

    }

    public static void main(String[] args) {

        fun1(2);

    }

}

//  recursive solution to print numbers from N to 1 using recursion.

class Recuresion {

    static void printToN(int n) {

        if (n == 0)

            return;

        System.out.print(n + " ");

        printToN(n - 1);

    }

    public static void main(String[] args) {

        int n = 4;

        printToN(n);

    }

}

// recursive solution to print numbers from 1 to N using recursion.

class Recuresion {

    static void printToN(int n) {

        if (n == 0)

            return;

        printToN(n - 1);

        System.out.print(n + " ");

    }

    public static void main(String[] args) {

        int n = 4;

        printToN(n);

    }

}

**What is Tail Recursion**

**Tail recursion** is defined as a recursive function in which the recursive call is the last statement that is executed by the function. So basically nothing is left to execute after the recursion call.

// Tail Recursive Code for print 1 to N

class Recuresion {

   static void fun(int n, int k) {

        if (n == 0)

            return;

        System.out.println(k);

        fun(n - 1, k + 1);

    }

    public static void main(String[] args) {

        fun(4, 1);

    }

}

// Tail Recursive code for factorial

class Recuresion {

    static int fact(int n, int k) {

        if (n == 0 || n == 1)

            return k;

        return fact(n - 1, k \* n);

    }

    public static void main(String[] args) {

        System.out.println(fact(4, 01));

    }

}

//recursive function to check if a string is palindrome or not.

class Recuresion {

    static boolean isPalindrome(String str, int start, int end) {

        if (start >= end)

            return true;

        return ((str.charAt(start) == str.charAt(end)) &&

                isPalindrome(str, start + 1, end - 1));

    }

    public static void main(String[] args) {

        String s = "suhanahus";

        System.out.println(isPalindrome(s, 0, s.length() - 1));

    }

}

//Rope cutting problem

class Recuresion {

    static int maxCuts(int n, int a, int b, int c) {

        if (n == 0)

            return 0;

        if (n <= -1)

            return -1;

        int res = Math.max(maxCuts(n - a, a, b, c),

                Math.max(maxCuts(n - b, a, b, c),

                        maxCuts(n - c, a, b, c)));

        if (res == -1)

            return -1;

        return res + 1;

    }

    public static void main(String[] args) {

        int n = 5, a = 2, b = 1, c = 5;

        System.out.println(maxCuts(n, a, b, c));

    }

}

//SOME OF DIGITES  USING RECURSION:

class Recuresion {

    static int fun(int n) {

        if (n < 10)

            return n;

        return fun(n / 10) + n % 10;

    }

    public static void main(String[] args) {

        System.out.println(fun(82734));

    }

}

Backtracking

**Backtracking Algorithm**

A backtracking algorithm is a problem-solving algorithm that uses a **brute force approach** for finding the desired output. The Brute force approach tries out all the possible solutions and chooses the desired/best solutions. The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach.

**When to Use a Backtracking Algorithm?**

Backtracking algorithms are best used to solve problems that have the following characteristics:

* There are multiple possible solutions to the problem.
* The problem can be broken down into smaller subproblems.
* The subproblems can be solved independently.

// Appling backtracking on array where exist index 1 to 4 which have value 0 to 4

//Now we want to backtrack(Traverasal) -2 with each element of arrays.

public class Backtracking {

    public static void changeArr(int arr[], int i, int val) {

        if (i == arr.length) {

            printArr(arr);

            return;

        }

        arr[i] = val;

        changeArr(arr, i + 1, val + 1);

        arr[i] = arr[i] - 2;

    }

    public static void printArr(int arr[]) {

        for (int i = 0; i < arr.length; i++) {

            System.out.println((arr[i] + " "));

        }

        System.out.println();

    }

    public static void main(String[] args) {

        int arr[] = new int[5];

        changeArr(arr, 0, 1);

        printArr(arr);

    }

}

// Place N Queens on an NxN chessboard such that no 2 queens can attack each other.

public class N\_Queen {

    public static boolean isSafe(char board[][], int row, int col) {

        // Vertical up

        for (int i = row - 1; i >= 0; i--) {

            if (board[i][col] == 'Q') {

                return false;

            }

        }

        // Diagonal left up

        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {

            if (board[i][j] == 'Q') {

                return false;

            }

        }

        // Digoanal right up

        for (int i = row - 1, j = col + 1; i >= 0 && j < board.length; i--, j++) {

            if (board[i][j] == 'Q') {

                return false;

            }

        }

        return true;

    }

    public static void nQueens(char board[][], int row) {

        // Base case

        if (row == board.length) {

            printBoard(board);

            return;

        }

        // coloumn loop

        for (int j = 0; j < board.length; j++) {

            if (isSafe(board, row, j)) {

                board[row][j] = 'Q';

                nQueens(board, row + 1); // function call

                board[row][j] = 'X'; // Backtracking step

            }

        }

    }

    public static void printBoard(char board[][]) {

        System.out.println("--------Chess Board---------");

        for (int i = 0; i < board.length; i++) {

            for (int j = 0; j < board.length; j++) {

                System.out.print(board[i][j] + " ");

            }

            System.out.println();

        }

    }

    public static void main(String[] args) {

        int n = 4;

        char board[][] = new char[n][n];

        // Initialize

        for (int i = 0; i < n; i++) {

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

                board[i][j] = 'X';

            }

        }

        nQueens(board, 0);

    }

}

// Count total number of ways in which we can solve N queens problem.

public class N\_QueenCount {

    public static boolean isSafe(char board[][], int row, int col) {

        // Vertical up

        for (int i = row - 1; i >= 0; i--) {

            if (board[i][col] == 'Q') {

                return false;

            }

        }

        // Diagonal left up

        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {

            if (board[i][j] == 'Q') {

                return false;

            }

        }

        // Digoanal right up

        for (int i = row - 1, j = col + 1; i >= 0 && j < board.length; i--, j++) {

            if (board[i][j] == 'Q') {

                return false;

            }

        }

        return true;

    }

    public static void nQueens(char board[][], int row) {

        // Base case

        if (row == board.length) {

            count++;

            return;

        }

        // coloumn loop

        for (int j = 0; j < board.length; j++) {

            if (isSafe(board, row, j)) {

                board[row][j] = 'Q';

                nQueens(board, row + 1); // function call

                board[row][j] = 'X'; // Backtracking step

            }

        }

    }

    public static void printBoard(char board[][]) {

        System.out.println("--------Chess Board---------");

        for (int i = 0; i < board.length; i++) {

            for (int j = 0; j < board.length; j++) {

                System.out.print(board[i][j] + " ");

            }

            System.out.println();

        }

    }

    static int count = 0;

    public static void main(String[] args) {

        int n = 10;

        char board[][] = new char[n][n];

        // Initialize

        for (int i = 0; i < n; i++) {

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

                board[i][j] = 'X';

            }

        }

        nQueens(board, 0);

        System.out.println("Totel ways of solution is:" + count);

    }

}

//Checkk if problem can be solved & print only 1 solution to N queens problem.

public class N\_QueenOneSolution {

    public static boolean isSafe(char board[][], int row, int col) {

        // Vertical up

        for (int i = row - 1; i >= 0; i--) {

            if (board[i][col] == 'Q') {

                return false;

            }

        }

        // Diagonal left up

        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {

            if (board[i][j] == 'Q') {

                return false;

            }

        }

        // Digoanal right up

        for (int i = row - 1, j = col + 1; i >= 0 && j < board.length; i--, j++) {

            if (board[i][j] == 'Q') {

                return false;

            }

        }

        return true;

    }

    public static boolean nQueens(char board[][], int row) {

        // Base case

        if (row == board.length) {

            count++;

            return true;

        }

        // coloumn loop

        for (int j = 0; j < board.length; j++) {

            if (isSafe(board, row, j)) {

                board[row][j] = 'Q';

                if (nQueens(board, row + 1)) {

                    return true;

                }

                board[row][j] = 'X'; // Backtracking step

            }

        }

        return false;

    }

    public static void printBoard(char board[][]) {

        System.out.println("--------Chess Board---------");

        for (int i = 0; i < board.length; i++) {

            for (int j = 0; j < board.length; j++) {

                System.out.print(board[i][j] + " ");

            }

            System.out.println();

        }

    }

    static int count = 0;

    public static void main(String[] args) {

        int n = 10;

        char board[][] = new char[n][n];

        // Initialize

        for (int i = 0; i < n; i++) {

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

                board[i][j] = 'X';

            }

        }

        if (nQueens(board, 0)) {

            System.out.println("solution is possible");

            printBoard(board);

        } else {

            System.out.println("solution is not possible");

        }

    }

}

Bit magic

**Bitwise operators:** Bitwise operators are used to deal with binary operations.

|  |  |  |
| --- | --- | --- |
| **Name** | **Operator** | **Example** |
| **Bitwise AND** | & | a & b |
| **Bitwise OR** | | | a | b |
| **Bitwise NOT** | ~ | ~a |
| **Bitwise XOR** | ^ | a ^ b |
| **Bitwise right shift** | >> | a>> |
| **Bitwise left shift** | << | b<< |
| **Unsigned right shift** | >>> | a>>> |

**<<Bitwise AND:** X: 0000………..OO11 🡪3

Y: 0000…………0110 🡪6

(X&Y): 0000………...0010 🡪2

public class bitMagic {

    public static void main(String[] args) {

        int x = 3, y = 6;

        // Both value should be true for true output:

        System.out.println(x & y);

    }

}

Output: 2

**Bitwise OR:** X: 0000………..OO11 🡪3

Y: 0000…………0110 🡪6

(X|Y): 0000………...0111 🡪7

public class bitMagic {

    public static void main(String[] args) {

        int x = 3, y = 6;

       System.out.println(x | y); // Atleast One value should be true for true output:

     }

   } Output: 7

**Bitwise XOR:** X: 0000………..OO11 🡪3

Y: 0000…………0110 🡪6

(X^Y): 0000………...0101 🡪5

public class bitMagic {

    public static void main(String[] args) {

        int x = 3, y = 6;

        // Same value give false output and different value gives true output:

       System.out.println(x | y);

     }

   }

**Bitwise NOT:** X: 0000………..O101 🡪5

~X: 0000………..1010

In java negative number are stored in 2s’ complement represtion.(2^32-1).

public class bitMagic {

    public static void main(String[] args) {

        int x = 5;

        // 1s' complement output:

        System.out.println(~x);

        // 2s'complement output:

        System.out.println(~x + 1);

    }

}

**Bitwise Left Shift:** X: 0000………..OO11 🡪3

(X<<1):0000……..….0110 🡪6

(X<<2):0000……....01100 🡪12

(X<<4):0000…...0110000 🡪48

* a<<b = a\*2^b

public class bitMagic {

    public static void main(String[] args) {

        int x = 3;

        System.out.println(x << 1);

        System.out.println(x << 2);

        System.out.println(x << 4);

    }

}

Output: 6, 12, 48

**Bitwise Right Shift:** X: 0000………..100001 🡪3

(X>>1): 00000……..….010000 🡪26

(X>>2): 000000……....001000 🡪8

* a>>b = a/2^b

public class bitMagic {

    public static void main(String[] args) {

        int x = 33;

        System.out.println(x >> 1);

        System.out.println(x >> 2);

    }

}

Output: 16, 8

The right shift fill leading bits of negative numbers with 1 but unsigned right shift shift is fill leading bits with 0 means that unsigned right shift consider negative numbers also as positive number.

**Bitwise Unsignet Right Shift:** X: 111………..10 🡪 -2

(X>>>1): 011……..….11 🡪 2147483647

(X>>>2): 001……......11 🡪 1073741823

public class bitMagic {

    public static void main(String[] args) {

        int x = -2;

        System.out.println(x >>> 1);

        System.out.println(x >>> 2);

    }

}

// <<<<<<<<<<<<<-------CHECK IF ODD OR EVEN------->>>>>>>>>>>>>>

public class bitMagic1 {

    static void evenOdd(int n) {

        int bitMask = 1;

        if ((n & bitMask) == 0) {

            System.out.println("Even");

        } else {

            System.out.println("Odd");

        }

    }

    public static void main(String[] args) {

        evenOdd(4345623);

        evenOdd(1338475);

    }

}

// <<<<<<<<<<<<<<------ GET nTH BIT ------------>>>>>>>>>>>>>

class bitMagic1 {

    static int getNthBit(int n, int i) {

        int bitMask = 1 << i;

        if ((n & bitMask) == 0) {

            return 0;

        } else {

            return 1;

        }

    }

    public static void main(String[] args) {

        System.out.println(getNthBit(10, 03));

        // getNthBit(10, 2);

    }

}

// <<<<<<<<<<<<<----------SET NTH BIT -------------->>>>>>>>>>>.

class bitMagic1 {

    static int setNthBit(int n, int i) {

        int bitMask = 1 << i;

        return n | bitMask;

    }

    public static void main(String[] args) {

        System.out.println(setNthBit(10, 02));

    }

}

// <<<<<<<<<<<<<<-----------CLEAR nTH BIT--------->>>>>>>>>>>>>>>>>>

class bitMagic1 {

    static int clearNthBit(int n, int i) {

        int bitMask = ~(1 << i);

        return n & bitMask;

    }

    public static void main(String[] args) {

        System.out.println(clearNthBit(10, 02));

    }

}

// <<<<<<<<<<<<----------CLEAR N BIT ---------------->>>>>>>>>>>>>>>>>>>

class bitMagic1 {

    static int clearBit(int n, int i) {

        int bitmask = (~0) << i;

        return n & bitmask;

    }

    public static void main(String[] args) {

        System.out.println(clearBit(10, 02));

    }

}

// <<<<<<<<<<-----------CHECK IF A NUMBER IS POWER OF 2 OR

class bitMagic1 {

    public static boolean isPowerOfTwo(int n) {

        return (n & (n - 1)) == 0;

    }

    public static void main(String[] args) {

        System.out.println(isPowerOfTwo(15));

    }

}

// <<<<<<<<<<-----OTHER WAY FOR POWER CHECK------------->>>>>>>>>Naive way:

class bitMagic1 {

    static boolean isPow2(int n) {

        if (n == 0) {

            return false;

        }

        while (n != 1) {

            if (n % 2 != 0)

                return false;

            n = n / 2;

        }

        return true;

    }

    public static void main(String[] args) {

        System.out.println(isPow2(4));

    }

}

// <<<<<<<<<<<------------Other way------------->>>>>>>>>>.

class bitMagic1 {

    static boolean isPow2(int n) {

        if (n == 0) {

            return false;

        }

        return ((n & (n - 1)) == 0);

    }

    public static void main(String[] args) {

        System.out.println(isPow2(8));

    }

}

// <<<<<<<<<<-----------COUNT HOW MANY BIT'S ARE SET IN A NUMBER----------->>>>>>>>>>

// T.C = O(logn)

class bitMagic1 {

    public static int countSetBits(int n) {

        int count = 0;

        while (n > 0) {

            if ((n & 1) != 0) { // check out LSB

                count++;

            }

            n = n >> 1;

        }

        return count;

    }

    public static void main(String[] args) {

        System.out.println(countSetBits(15));

    }

}

// <<<<<<<<<<<<------OTHER WAY------------>>>>>>>>>>>>

class bitMagic1 {

    static int countset(int n) {

        int res = 0;

        while (n > 0) {

            if (n % 2 != 0) {

                res++;

            }

            n = n / 2;

        }

        return res;

    }

    public static void main(String[] args) {

        System.out.println(countset(15));

    }

}

// T.C = 0(total bit's in n ):

// <<<<<<<<<<<--------------OTHER WAY { Brian kerningam's algo}

class bitMagic1 {

    static int countBits(int n) {

        int res = 0;

        while (n > 0) {

            n = (n & (n - 1));

            res++;

        }

        return res;

    }

    public static void main(String[] args) {

        System.out.println(countBits(15));

    }

}

// T.C = (set bit count)

//takes an array of integers and displays the number that has odd occurrences in the array.

// Naive way:

class OddOcurring {

    static int findOdd(int arr[], int n) {

        for (int i = 0; i < n; i++) {

            int count = 0;

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

                if (arr[i] == arr[j])

                    count++;

            }

            if (count % 2 != 0)

                return arr[i];

        }

        return 0;

    }

    public static void main(String[] args) {

        int arr[] = { 4, 3, 4, 4, 4, 5, 5, 3, 3 }, n = 9;

        System.out.println(findOdd(arr, n));

    }

}

// Efficiant way:

class OddOcurring {

    static int findOdd(int arr[], int n) {

        int res = 0;

        for (int i = 0; i < n; i++) {

            res = res ^ arr[i];

        }

        return res;

    }

    public static void main(String[] args) {

        int arr[] = { 4, 3, 4, 4, 4, 5, 5, 3, 3 }, n = 9;

        System.out.println(findOdd(arr, n));

    }

}

//This program takes an array of integers and displays the two number that has odd occurrences in the array.

//Efficiant way:

class OddOcurring {

    static void oddAppearing(int arr[], int n) {

        int xor = 0, res1 = 0, res2 = 0;

        for (int i = 0; i < n; i++)

            xor = xor ^ arr[i];

        int sn = xor & (~(xor - 1));

        for (int i = 0; i < n; i++) {

            if ((arr[i] & sn) != 0)

                res1 = res1 ^ arr[i];

            else

                res2 = res2 ^ arr[i];

        }

        System.out.println(res1 + " " + res2);

    }

    public static void main(String[] args) {

        int arr[] = { 3, 4, 3, 4, 5, 4, 4, 6, 7, 7 }, n = 10;

        oddAppearing(arr, n);

    }

}

// Naive Way:

class OddOcurring {

    static void oddAppearing(int arr[], int n) {

        for (int i = 0; i < n; i++) {

            int count = 0;

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

                if (arr[i] == arr[j])

                    count++;

            }

            if (count % 2 != 0)

                System.out.print(arr[i] + " ");

        }

    }

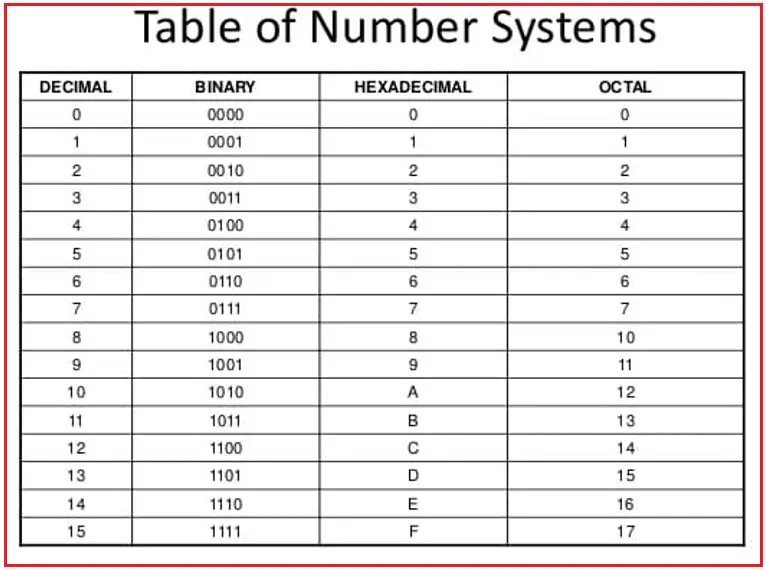
    public static void main(String[] args) {

        int arr[] = { 3, 4, 3, 4, 5, 4, 4, 6, 7, 7 }, n = 10;

        oddAppearing(arr, n);

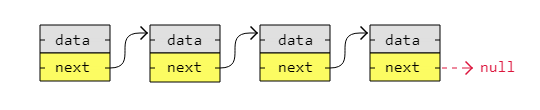
    }

}



Linked\_list

**Linked Lists:-** A linked list consists of nodes with some sort of data, and a pointer, or link, to the next node.



Node1 Node2 Node3 Node4

**Head** **Tail**

A big benefit with using linked lists is that nodes are stored wherever there is free space in memory, the nodes do not have to be stored contiguously right after each other like elements are stored in arrays. Another nice thing with linked lists is that when adding or removing nodes, the rest of the nodes in the list do not have to be shifted.

**Advantages of Linked Lists**

* **Dynamic Size**: The size of the linked list can grow or shrink as needed, unlike arrays which have a fixed size.
* **Efficient Insertions/Deletions**: Adding or removing elements from the beginning or end is more efficient compared to arrays since it does not require shifting elements.

**Disadvantages of Linked Lists**

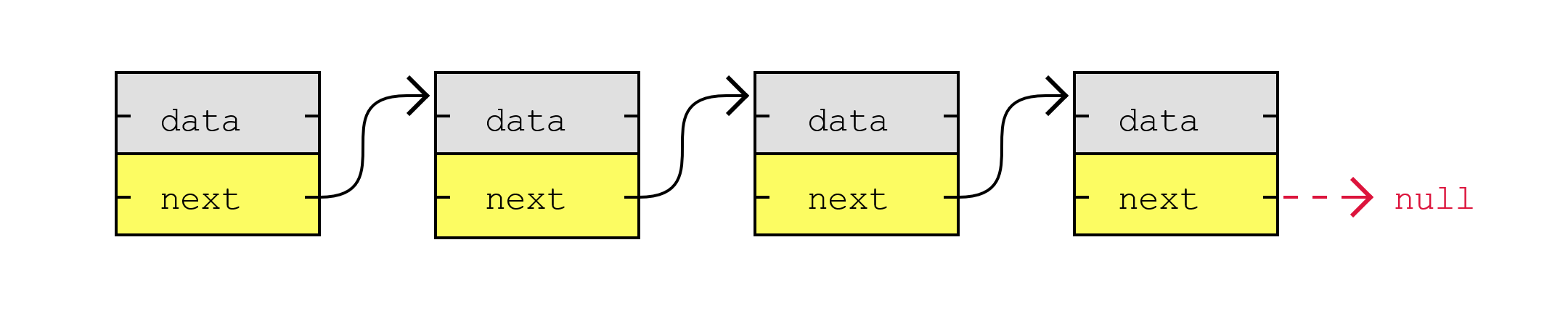
* **Memory Overhead**: Each node requires extra memory for the pointer.
* **Sequential Access**: Elements cannot be accessed directly by index; traversal from the head is required.
* **Cache Performance**: Linked lists have poor cache performance compared to arrays due to non-contiguous memory allocation.

**Types of Linked Lists**

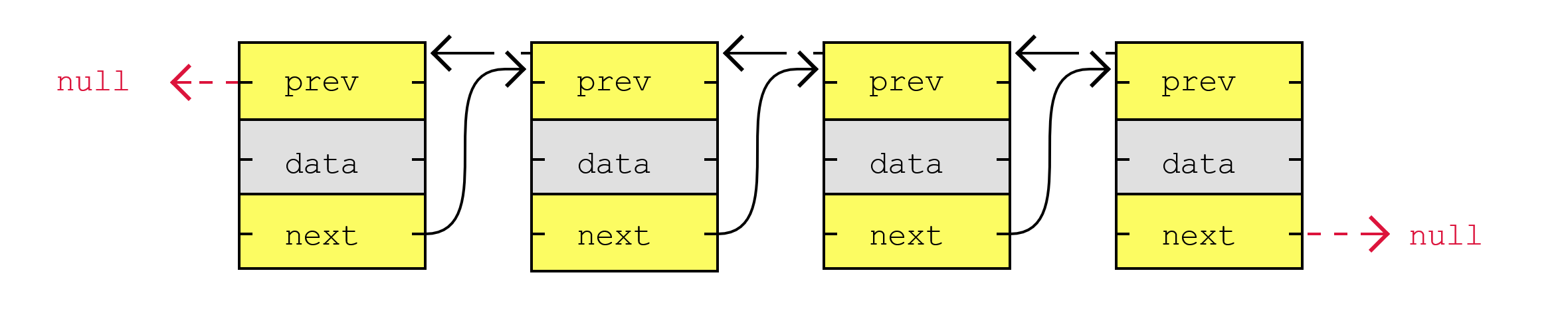
There are three basic forms of linked lists:

1. Singly linked lists
2. Doubly linked lists
3. Circular linked lists

A **singly linked list** is the simplest kind of linked lists. It takes up less space in memory because each node has only one address to the next node, like in the image below.



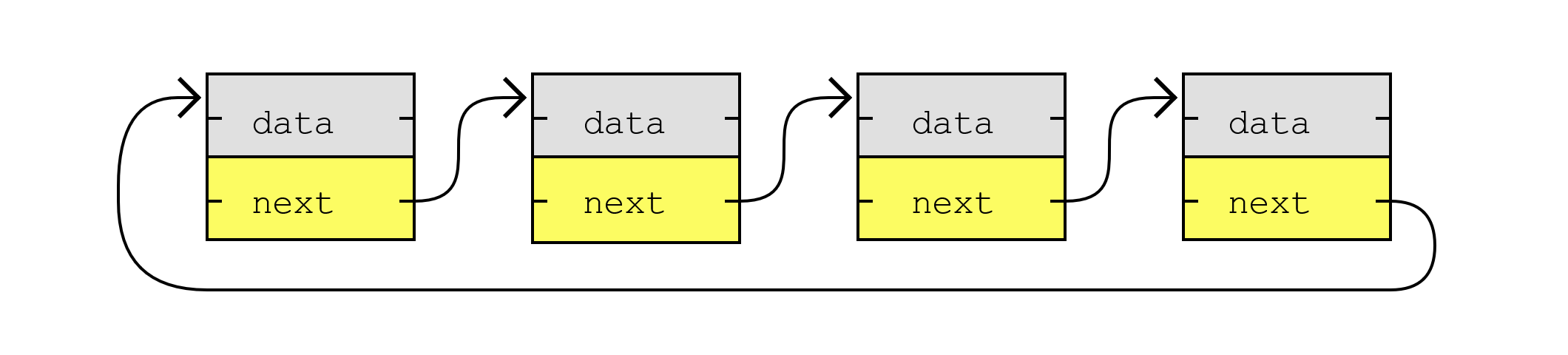
A **doubly linked list** has nodes with addresses to both the previous and the next node, like in the image below, and therefore takes up more memory. But doubly linked lists are good if you want to be able to move both up and down in the list.



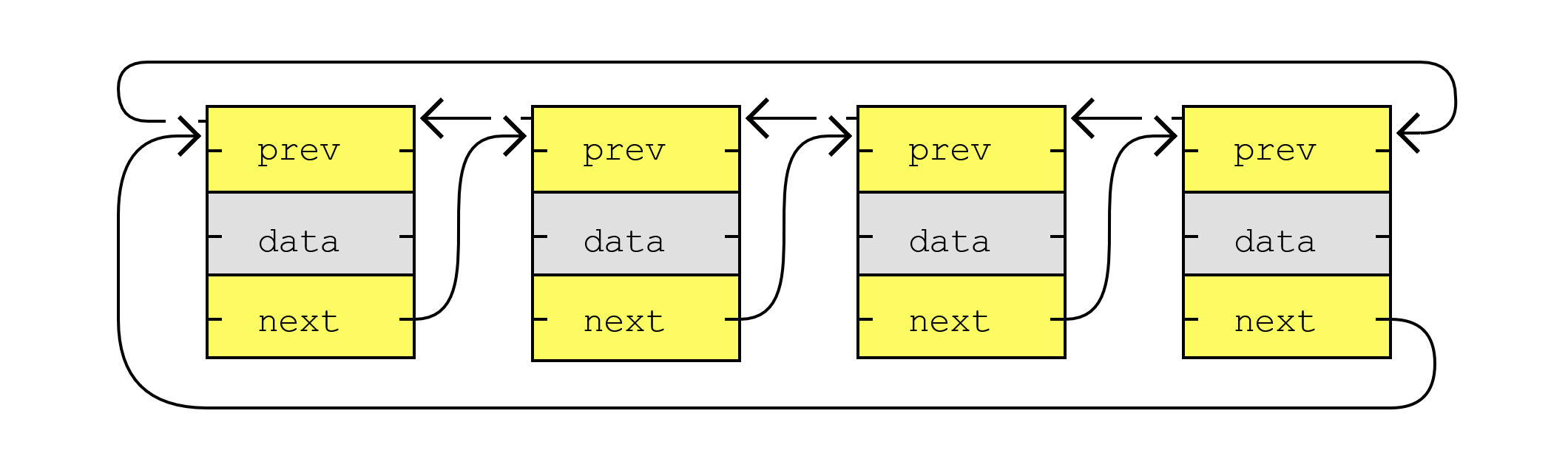
A **circular linked list** is like a singly or doubly linked list with the first node, the "head", and the last node, the "tail", connected.

In singly or doubly linked lists, we can find the start and end of a list by just checking if the links are null. But for circular linked lists, more complex code is needed to explicitly check for start and end nodes in certain applications. Circular linked lists are good for lists you need to cycle through continuously.

The image below is an example of a singly circular linked list:



The image below is an example of a doubly circular linked list:



**Linked Lists in Memory**

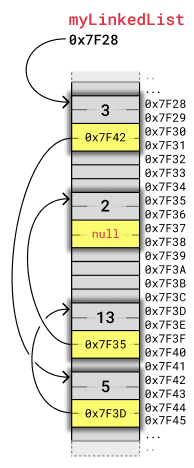
Instead of storing a collection of data as an array, we can create a linked list.

Linked lists are used in many scenarios, like dynamic data storage, stack and queue implementation or graph representation, to mention some of them.

A linked list consists of nodes with some sort of data, and at least one pointer, or link, to other nodes.

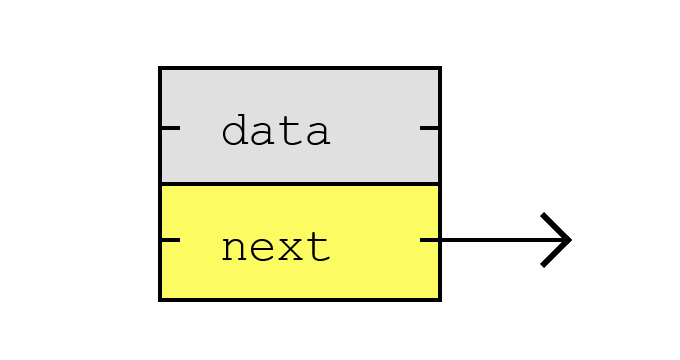
A big benefit with using linked lists is that nodes are stored wherever there is free space in memory, the nodes do not have to be stored contiguously right after each other like elements are stored in arrays. Another nice thing with linked lists is that when adding or removing nodes, the rest of the nodes in the list do not have to be shifted.

* The image below shows how a linked list can be stored in memory. The linked list has four nodes with values 3, 5, 13 and 2, and each node has a pointer to the next node in the list.

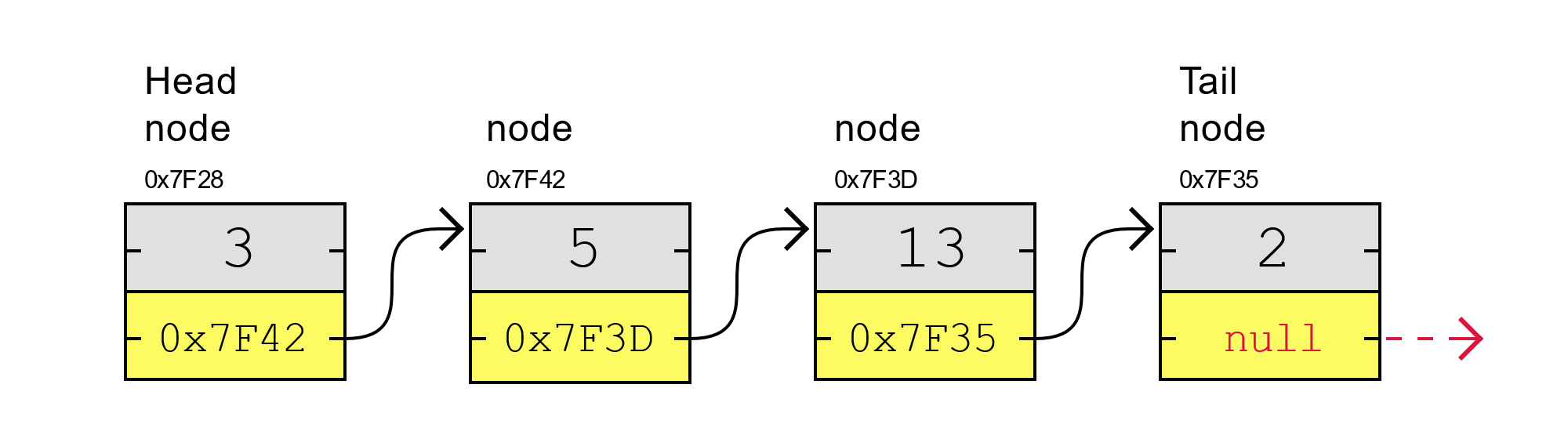


Each node takes up four bytes. Two bytes are used to store an integer value, and two bytes are used to store the address to the next node in the list. As mentioned before, how many bytes that are needed to store integers and addresses depend on the architecture of the computer. This example, like the previous array example, fits with a simple 8-bit microcontroller architecture.

To make it easier to see how the nodes relate to each other, we will display nodes in a linked list in a simpler way, less related to their memory location, like in the image below:



If we put the same four nodes from the previous example together using this new visualization, it looks like this:



As you can see, the first node in a linked list is called the "Head", and the last node is called the "Tail".

Unlike with arrays, the nodes in a linked list are not placed right after each other in memory. This means that when inserting or removing a node, shifting of other nodes is not necessary, so that is a good thing.

Something not so good with linked lists is that we cannot access a node directly like we can with an array by just writing myArray[5] for example. To get to node number 5 in a linked list, we must start with the first node called "head", use that node's pointer to get to the next node, and do so while keeping track of the number of nodes we have visited until we reach node number 5.

Learning about linked lists helps us to better understand concepts like memory allocation and pointers.

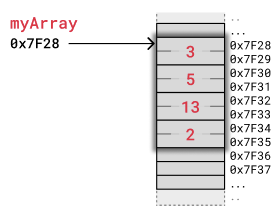
Linked lists are also important to understand before learning about more complex data structures such as trees and graphs, that can be implemented using linked lists.

**Arrays in Memory**

To understand linked lists, it is useful to first know how arrays are stored in memory.

Elements in an array are stored contiguously in memory. That means that each element is stored right after the previous element.

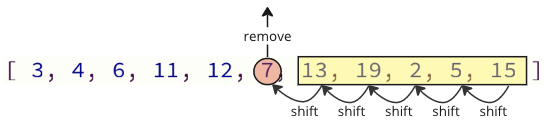
The image below shows how an array of integers myArray = [3,5,13,2] is stored in memory. We use a simple kind of memory here with two bytes for each integer, like in the previous example, just to get the idea.



The computer has only got the address of the first byte of myArray, so to access the 3rd element with code myArray[2] the computer starts at 0x7F28 and jumps over the two first integers. The computer knows that an integer is stored in two bytes, so it jumps 2x2 bytes forward from 0x7F28 and reads value 13 starting at address 0x7F32.

When removing or inserting elements in an array, every element that comes after must be either shifted up to make place for the new element, or shifted down to take the removed element's place. Such shifting operations are time consuming and can cause problems in real-time systems for example.

The image below shows how elements are shifted when an array element is removed.



The table below compares linked lists with arrays to give a better understanding of what linked lists are.

|  |  |  |
| --- | --- | --- |
|  | **Arrays** | **Linked Lists** |
| *An existing data structure in the programming language* | Yes | No |
| *Fixed size in memory* | Yes | No |
| *Elements, or nodes, are stored right after each other in memory (contiguously)* | Yes | No |
| *Memory usage is low (each node only contains data, no links to other nodes)* | Yes | No |
| *Elements, or nodes, can be accessed directly (random access)* | Yes | No |
| *Elements, or nodes, can be inserted or deleted in constant time, no shifting operations in memory needed.* | No | Yes |

**Implementation of LinkedList from scratch:**

package Test\_Demo;

public class Linked\_List\_Learn {

    public static class Node {// we create a class which makes a Node of LL

        int data; // These are the properties of this class

        Node next;

        public Node(int data) { //constructor which initializing next with by default cnull

            this.data = data;

            this.next = null;

        }

    }

    public static Node head; // creating a head and tail of the LL

    public static Node tail;

    // Method to calculate the size of the linked list

    public int size() { //O(n)

        int count = 0;

        Node current = head;

        while (current != null) {

            count++;

            current = current.next;

        }

        return count;

    }

    public static void main(String[] args) {

        Linked\_List\_Learn list = new Linked\_List\_Learn(); // create a object of LL with as class name

        // we not use the adding operation like this we use functions for every opeartions

        list.head = new Node(9);

        list.head.next = new Node(4);

        list.head.next.next = new Node(3);

        System.out.println("Size of this linkelist " + list.size());

//We can also create LL using this approach:

//Node a = new Node(5); // create Node

// Node b = new Node(6);

// Node c = new Node(9);

// Node c = new Node(2);

// a.next = b; //Link the Node

// b.next = c;

// c.next = d

    }

}

Output:

Size of this linkelist 3

**LinkedList in java collections framework:**

The LinkedList class of the [Java collections framework](https://www.programiz.com/java-programming/collections)provides the functionality of the linked list data structure.

Linked List is a part of the Collection framework present in java.util package. This class is an implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node.

Each element in a linked list is known as a **node**. It consists of 3 fields:

* **Prev** - stores an address of the previous element in the list. It is null for the first element
* **Next** - stores an address of the next element in the list. It is null for the last element
* **Data** - stores the actual data

**How Does LinkedList work Internally?**

Since a LinkedList acts as a dynamic array and we do not have to specify the size while creating it, the size of the list automatically increases when we dynamically add and remove items. And also, the elements are not stored in a continuous fashion. Therefore, there is no need to increase the size. Internally, the LinkedList is implemented using the doubly linked list data structure

**Time complexity of Linkedlist methods:**

|  |  |
| --- | --- |
| Methods | Time complexity |
|  |  |
| contains() | O(n) |
| remove() | O(n) |
| indexOf(object) | O(n) |
| lastIndexOf(object) | O(n) |
| clear() | O(n) |
| add(object) | O(1) |
| isEmpty() | O(1) |
| add(index, object) | O(index) |
| remove(index) | O(index) |
| get(index) | O(index) |
| set(index, object) | O(index) |
|  |  |

**Note:** In java collections framework no need to make all methods form scratch all the methods are pre-written and optimize simply use it.

**Example:**

import java.util.LinkedList;

class linklistDemo {

    public static void main(String[] args) {

        LinkedList<Integer> link = new LinkedList<>(); //Integer type LL we can create any type of LL like String, Integer, bollean etc.

        link.add(10);

        link.add(20);

        link.add(30);

        link.add(2, 40);

        System.out.println(link);

        System.out.println(link.contains(30));

        link.remove(0);

        link.remove("10");

        link.get(1);

        link.set(0, 50);

        link.indexOf(30);

        link.indexOf(10);

        link.lastIndexOf(20);

        link.isEmpty();

        link.clear();

    }

}

**Creating the all operations of Singly LinkedList from scratch:**

package LinkedList;

public class SinglyLinkedList\_Operations {

    public static class Node {

        int data;

        Node next;

        public Node(int data) {

            this.data = data;

            this.next = null;

        }

    }

    public static Node head;

    public static Node tail;

    static int size;

    // ADD VALUES IN THE LINKEDLIST AT FIRST NODE:

    public void addFirst(int data) {

        Node newNode = new Node(data);

        size++;

        if (head == null) {

            head = tail = newNode;

            return;

        }

        newNode.next = head;

        head = newNode;

    }

    // ADD VALUES IN THE LINKEDLIST AT LAST NODE:

    public void addLast(int data) {

        Node NN = new Node(data);

        size++;

        if (head == null) {

            head = tail = NN;

            return;

        }

        tail.next = NN;

        tail = NN;

    }

    // ADD VALUES IN THE LINKEDLIST AT SPECIFIC NODE:

    public void addAtindex(int index, int data) {

        // If we want to add elements on head of LL:

        if (index == 0) {

            addFirst(data);

            return;

        }

        // if(index < 0 || index > size()){

        // System.out.println("wrong index");

        // }

        Node new\_Node = new Node(data);

        Node temp = head;

        for (int i = 0; i <= index - 1; i++) {

            temp = temp.next;

        }

        new\_Node.next = temp.next;

        temp.next = new\_Node;

    }

    // Method to delete the first node of the linked list

    public void deleteFirst() {

        if (head == null) {

            System.out.println("Linked List is empty, nothing to delete.");

            return;

        }

        if (head == tail) { // If there's only one node in the list

            head = tail = null;

        } else {

            head = head.next;

        }

    }

    // Method to delete the last node of the linked list

    public void deleteLast() {

        if (head == null) {

            System.out.println("Linked List is empty, nothing to delete.");

            return;

        }

        if (head == tail) { // If there's only one node in the list

            head = tail = null;

        } else {

            Node temp = head;

            while (temp.next != tail) {

                temp = temp.next;

            }

            temp.next = null;

            tail = temp;

        }

    }

    // Delete at specific index in LL

    public void deletAtindex(int index) {

        if (index == 0) {

            head = head.next;

            return;

        }

        Node temp = head;

        for (int i = 0; i <= index - 1; i++) {

            temp = temp.next;

        }

        temp.next = temp.next.next;

        tail = temp;

    }

    // SEARCHING IN THE LL

    public int search\_in\_Linkedlist(int key) {

        Node temp = head;

        int i = 0;

        while (temp != null) {

            if (temp.data == key) {

                return i;

            }

            temp = temp.next;

            i++;

        }

        // if key is not found

        return -1;

    }

    // displaying in normal way:

    public static void display(Node head) {

        while (head != null) {

            System.out.println(head.data + " ");

            head = head.next;

        }

    }

    // DISPLAYING LL USING RECURSION:

    public static void display1(Node head) {

        if (head == null)

            return;

        System.out.println(head.data + " ");

        display1(head.next);

    }

    // // DISPLAYING OF A LINKEDLIST IN REVERSE ORDER USING RECURSION:

    public static void displayReverse(Node head) {

        if (head == null)

            return;

        displayReverse(head.next);

        System.out.println(head.data + " ");

    }

    public static void main(String[] args) {

        SinglyLinkedList\_Operations ll = new SinglyLinkedList\_Operations();

        ll.addFirst(3);

    }

}

**Implimention of Doubly Linked List from scratch:**

class Node {

    int data;

    Node prev;

    Node next;

    Node(int d) {

        data = d;

        prev = null;

        next = null;

    }

}

class Test {

    public static void main(String args[]) {

        Node head = new Node(10);

        Node temp1 = new Node(20);

        Node temp2 = new Node(30);

        head.next = temp1;

        temp1.prev = head;

        temp1.next = temp2;

        temp2.prev = temp1;

        printlist(head);

    }

    public static void printlist(Node head) {

        Node curr = head;

        while (curr != null) {

            System.out.print(curr.data + " ");

            curr = curr.next;

        }

        System.out.println();

    }

}

**Creating the all operations of Doubly LinkedList from scratch:**

STACK:

Implementation using List

Implementation using Quese

---: using array, linkedlist , quese

---: using stack class<collections>

**>>Operations on Array:**

package Arrays;

public class OperationsOnArray {

    // Traversing on array:

    public static void main(String[] args) {

        int arr[] = { 3, 4, 7, 6, 8, 7, 9, 8, 0, 1, 2, 3, 4, 8, 65 };

        for (int i = 0; i < arr.length; i++) {

            // Values in index

            System.out.println(arr[i]);

        }

    }

}

// Index

// System.out.println(i);