**Demo Exam 2023 Algorithms and Data structure (Usman A):**

**NB ! Q1 vil mest sannsynlig komme på eksamen (lurt å kunne dette!!)**

**Graphical representation ( skissere (tegningene) sort algoritme løsningene)**

**Write the step-by-step algorithm when the question (Write an algorithm) is given**

**Q1:** **Write a step-by-step algorithm that finds the largest number in a list (an array) of n numbers.**

public void largestNumberInArray(){  
 int [] arr = {1,2,3,4,5,32,7,8,9,11};  
 int n = arr.length;  
 int max = arr[0];  
 for (int i = 0; i <n ; i++) {  
 if (arr[i]>max){  
 max=arr[i];  
  
 }  
 }  
 System.*out*.println("Largest number in array:"+max);  
}

**Step by Step algorithm:**

1. Initialize a variable in the type of an int and call it: “max” and set the max variable equal to arr[0] as default value.
2. Create a loop (for loop) that goes through each element in the array starting from the first element.
3. Create an if statement inside the loop, if element is greater than the max value, set the max value equal to the element.
4. Print the max value using System.out.println statement outside the loop and the answer should be displayed 😊

**Q2:**

<https://www.interviewkickstart.com/learn/difference-between-recursion-and-iteration#:~:text=Iteration%20is%20faster%20and%20more,using%20recursion%20and%20iteration%20both>.

<https://codeahoy.com/learn/recursionjava/ch6/>

**Iterations vs Recursion:**

**Iterations: (Uses loops to solve the problem)**

* Code structures that use loops is considered as iteration (or iterative codes)
* If the code repeatedly executes the set of instructions until the instructions or the loops becomes false
* Iterations has a polynomial time complexity

**Recursion: (When fuctions calls itself)**

* In recursion the functions calls itself in an infinite loop to solve the problem
* In recursive codes or methods, we try using smaller inputs to make the problem smaller
* It uses a base condition to stop the recursion, otherwise it could lead to a running out of stack memory
* Examples: Binary Search, Merge Sort, Quick Sort (Divide and Conquer), In-order/Pre-order/Post-order Tree Traversals, DFS of Graph, Fibonacci Series, and Factorial Problems, etc

**In Common:**

* The time complexity in Both recursion and iteration is O(N)
* Even though the time complexity is the same, iteration is faster since recursion takes a lot more time when the functions are calling themselves due to overhead of functions

**Which is more efficient (Iteration vs Recursion)?**

As mentioned before, in recursive code the function call itself which can cause memory loss due to overhead of functions. When a function calls other functions in an infinite loop, it will lead to take more space in stack memory. In this case the stack will run out of storage over time. Iteration on the other hand doesn’t use functions when solving the problem, instead it uses loops which repeats an action. Iterative codes /method doesn’t require stack memory to work like the recursive ones, since the loop structure occurs within the method. That is why it doesn’t use any memory or lead to any overhead in the memory. This is the reason why Iteration is more efficient than recursion.

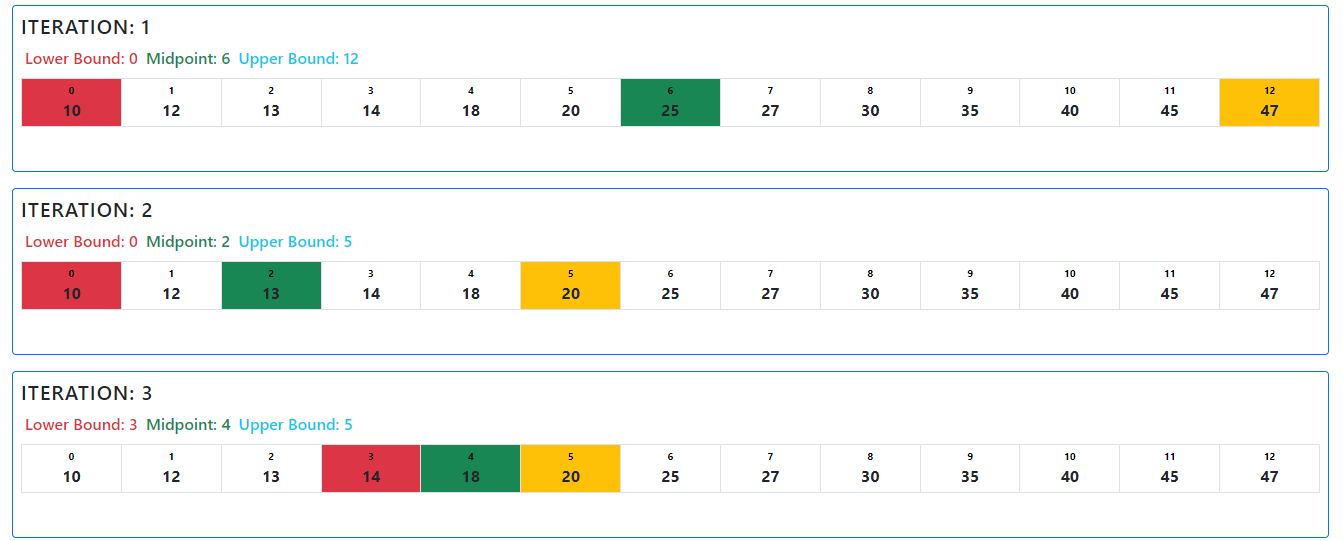
**Binary search:**

**Q3: i) Write an algorithm and solution to Binary Search with a Recursive version (10 points) Suppose x = 18, and we have the following array: [10,12,13,14,18,20,25,27,30,35,40,45,47]**

package Algoritmer.exercises;  
  
import java.util.Arrays;  
  
public class BinarySearch {  
  
 public void search(){  
 int [] arr ={1,2,3,4,5,6,7,8,9,10};  
 System.*out*.println(Arrays.*toString*(arr));  
 int key =9;  
 System.*out*.println("Key to be searched:"+key);  
  
 int low =0;  
 int high = arr.length-1;  
  
 int mid = (low+high)/2;  
  
 while (low <=high){  
 if (arr[mid]==key){  
 System.*out*.println("Element is found at index:" + mid);  
 break;  
 }if (arr[mid] < key) {  
 low = mid+1;  
 } else {  
 high=mid-1;  
 }  
 mid = (low+high)/2;  
  
 if (high>low){  
 System.*out*.println("Key is not found!");  
 }  
  
 }  
 }  
  
 public static void main(String[] args) {  
 BinarySearch bs = new BinarySearch();  
 bs.search();  
 }  
}

**Step by Step Binary Search Algorithm:**

1. Initial the variables low, high and key in the type of “int”.
2. Set low equal to 0 and high equal to ‘arr. length-1”.
3. Create a loop (While loop) to divide the list or array in half. While the low index is less or equal to the “high” index the binary search should be performed.
4. Use the formula (low+high)/2 to calculate the middle element from the array/list.
5. Compare the key element with the middle element.
6. Create an if statement inside the loop, if key is equal to middle element, then return the index position and the key is found!
7. Create an else-if statement, if key is greater than the middle element, then the key is located at the right subarray, repeat step 1-3 to find the key’s location.
8. Create a the last else-if statement, if key is lower than the middle element, choose the left subarray. Repeat step 1-3 to find it’s location 😊
9. Create an else statement, if the key is not found it’s the message (“Key’s not found”) should be displayed or return index-1.



18 is located on index x=4

**ii) When not to use Divide and Conquer (5 points) Short answer 3-4 sentences**

We cannot use divide and conquer in cases where the array is **unsorted**. To solve problems where divide and conquer is required, we have to sort the array otherwise it can cause errors when solving the problem. Other problems that can occur is when the **subproblems** cannot be solved separately, the subproblems have to be independent from each other to get solved otherwise the divide and conquer algorithm will fail. The last example is when the size of the array or the list **is small**, in this case it’s unnecessary to use the divide and conquer algorithm since in this type of lists the solution is usually simple, so we can’t divide the array into smaller subproblems. That’s why we can’t perform divide and conquer and it becomes useless.

**Divide and Conquer is a powerful algorithmic technique that involves breaking a problem down into subproblems that are easier to solve. However, there are some situations where Divide and Conquer may not be the most appropriate approach to use. Here are a few examples:**

**1. When the subproblems cannot be solved independently: In Divide and Conquer, the subproblems are independent of each other and can be solved separately. If the subproblems are interdependent and cannot be solved separately, then Divide and Conquer may not be the best approach.**

**2. When the problem size is small: Divide and Conquer is most effective when the problem can be divided into smaller subproblems. If the problem size is already small, then the overhead of dividing and combining the subproblems may outweigh the benefits.**

**3. When the problem has a simple solution: If the problem has a simple and straightforward solution that does not require the use of a complex algorithm like Divide and Conquer, then it may be more efficient to use the simple solution.**

**4. When the problem requires sequential processing: Divide and Conquer involves solving subproblems in parallel, which can be beneficial for some problems. However, for problems that require sequential processing, such as string matching, Divide and Conquer may not be the best approach.**

**5. When the problem has a non-uniform input distribution: Divide and Conquer works best when the input is uniformly distributed among the subproblems. If the input is non-uniformly distributed, then some subproblems may require significantly more computation than others, resulting in inefficient use of resources.**

**Q4:**

**(i) Why has Bubble sort proved inefficient compared to other sorting algorithms? (4 points)**

Bubble sort is the simplest sorting algorithm compared to Merge sort and Quick sort. Bubble sort can’t deal with larger datasets like the Merge sort or the Quicksort can and it’s only suitable for smaller list where the efficiency isn’t important. Bubble sort is a sorting algorithm that work by repeatedly swapping elements in the unsorted list and follows the O(N^2) time complexity whereas Merge Sort and Quick Sort follows the O(nlogn) time complexity. In bubble sort the list has to be unsorted in order to be solved, but if the list is already sorted, the bubble sort is unnecessary and cannot be performed. On the other hand, as mentioned before the Merge and the Quick sort can deal with larger lists / datasets and work by dividing the list into smaller subproblems before merging them back into a sorted list. This is why Merge sort and Quick sort has proved more efficiency than Bubble Sort.

**(ii) Sort the following list with Bubble Sort, Merge Sort and Quick Sort 123 34 189 56 150 12 9 240 (Skal man bruke java kode?? Send mail til foreleseren)**

**Bubble Sort:**

class BubbleSort {  
  
 void bubbleSort(int [] arr){  
 int n = arr.length;  
 for (int i = 0; i <n ; i++) {  
 for (int j = 0; j < n-i-1; j++) {  
 if (arr[j]>arr[j+1]){  
 int temp = arr[j];  
 arr[j] = arr[j+1];  
 arr[j+1]=temp;  
 }  
 }  
 }  
 }  
 void printArray(int [] arr){  
 int n = arr.length;  
 for (int i = 0; i <n ; i++) {  
 System.*out*.println(arr[i]);  
 }  
 }  
  
 public static void main(String[] args) {  
 int [] arr = {123, 34, 189, 56, 150 ,12 ,9 ,240};  
 BubbleSort ob = new BubbleSort();  
 System.*out*.println("Unsorted:");  
 ob.printArray(arr);  
 ob.bubbleSort(arr);  
 System.*out*.println("Sorted:");  
 ob.printArray(arr);  
 }  
}

**Quick Sort:**

package Algoritmer.exercises;  
  
import java.io.\*;  
 class QuickSort {  
  
 static void swap(int [] arr , int i , int j){  
 int temp = arr[i];  
 arr[i]=arr[j];  
 arr[j]=temp;  
 }  
   
 static int partition(int [] arr , int low , int high){  
 int pivot = arr[high];  
 int i = (low-1);  
  
 for (int j = low; j <=high-1 ; j++) {  
 if (arr[j]<pivot){  
 i++;  
 *swap*(arr , i, j);  
 }  
 }  
 *swap*(arr , i+1 ,high);  
 return (i+1);  
 }  
   
 static void quickSort(int [] arr , int low , int high){  
 if (low <high){  
 int pi =*partition*(arr , low ,high);  
   
 *quickSort*(arr , low ,pi-1);  
 *quickSort*(arr , pi+1 ,high);  
   
   
   
 }  
 }  
 static void printArray(int [] arr, int size){  
 for (int i = 0; i <size ; i++) {  
 System.*out*.println(arr[i]);  
 }  
 }  
  
 public static void main(String[] args) {  
 int [] arr = {123, 34, 189, 56, 150 ,12 ,9 ,240};  
 int n = arr.length;  
   
 System.*out*.println("Unsorted list:");  
 *printArray*(arr,n);  
 *quickSort*(arr, 0 , n-1);  
 System.*out*.println("Sorted list:");  
 *printArray*(arr , n);  
   
 }  
  
  
}

**Merge Sort:**

package Algoritmer.exercises;  
  
/\* Java program for Merge Sort \*/  
public class MergeSort {  
 void merge (int arr [] , int l , int m , int r){  
  
 int n1= m-l +1;  
 int n2 = r-m;  
  
 int L[] = new int [n1];  
 int R[] = new int[n2];  
  
 for (int i = 0; i < n1; i++) {  
 L[i]=arr[i+1];  
 }  
 for (int j = 0; j <n2 ; j++) {  
 R[j] = arr[m+1+j];  
 }  
  
 int i = 0 , j =0;  
  
 int k =l;  
  
 while (i <n1 && j < n2){  
  
 if (L[i] <=R[j]) {  
 arr[k] = L [i];  
  
 i++;  
 } else {  
 arr[k] = R [j];  
 j++;  
 }  
 k++;  
 }  
 while (i<n1){  
 arr[k] = L[i];  
 i++;  
 k++;  
 }  
  
 while (j <n2){  
 arr[k] = R[j];  
 j++;  
 k++;  
 }  
 }  
  
 void sort( int arr [] , int l , int r){  
  
 if ( l < r){  
 int m = l + (r-l)/2;  
  
 sort(arr , l , m);  
 sort(arr , m+1 , r);  
  
 merge(arr , l , m , r);  
 }  
 }  
  
 static void printArray( int [] arr){  
 int n = arr.length;  
 for (int i = 0; i <n ; i++) {  
 System.*out*.println(arr [i] + " ");  
  
 System.*out*.println();  
 }  
 }  
  
 public static void main(String[] args) {  
  
 int [] arr = {123, 34, 189, 56, 150 ,12 ,9 ,240};  
 int n = arr.length;  
 System.*out*.println("Unsorted list:");  
 *printArray*(arr);  
 System.*out*.println("Sorted list:");  
 MergeSort ms = new MergeSort();  
 ms.sort(arr , 0 ,n-1 );  
 *printArray*(arr);  
 }  
}

**(iii) provide a graphical representation of solving each of the sorting algorithms.**

(Se SkriveBoka for graphical representation av de ulike sorting algoritmene)

**Merge Sort Table format:**

**Arr = [123 34 189 56 150 12 9 240]**

|  |  |  |  |
| --- | --- | --- | --- |
| **k** | **U** | **V** | **Sorted (Result)** |
| **1** | 123,34,189,56 | 150, 12, 9, 240 | **9** |
| **2** | 123,34,189,56 | 150, 12, 9, 240 | **9,12** |
| **3** | 123,34,189,56 | 150, 12, 9, 240 | **9,12,34** |
| **4** | 123,34,189,56 | 150, 12, 9, 240 | **9,12,34,56** |
| **5** | 123,34,189,56 | 150, 12, 9, 240 | **9,12,34,56,123** |
| **6** | 123,34,189,56 | 150, 12, 9, 240 | **9,12,34,56,123,150** |
| **7** | 123,34,189,56 | 150, 12, 9, 240 | **9,12,34,56,123,150,189** |
| **----------------** | 123,34,189,56 | 150, 12, 9, 240 | **9,12,34,56,123,150,189,240**  **(Final values)** |

**Q6 : Short answers (3-4 sentences) are required for these 10 sub-questions (mangler) !!!**

**Algorithms notes:**

* What does it mean by algorithm efficiency: Time (How fast is it? What is the time complexity?) and Storage (Does it take a lot of storage?) (LO 1)

If an algorithm is efficient, it should be fast but also it should avoid using a lot of storage (not a requirement)

* **Big O notation** (means growth of functions) is a mathematical notation that describes the limiting behavior of a function when the argument or a problem tends towards a particular value or infinity.

In other words, it describes the time complexity in the code. F(n)=O(n^3) means the function grows asymptotic as fast as n3.

**Big O notation** is used to find the time complexity of algorithms. And it analyzes the algorithms based on their run time (how fast it is) or space (storage) requirements.

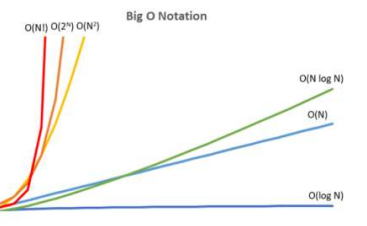
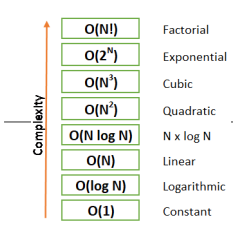
* **We can analyze how efficient** an algorithm is from the amount of time, storage, other resources it takes to run the algorithm, and a change in the input size. Big O Notation in Data Structure tells us how well an algorithm will perform in a particular situation. In other words, it gives an algorithm's upper-bound runtime or worst-case complexity.

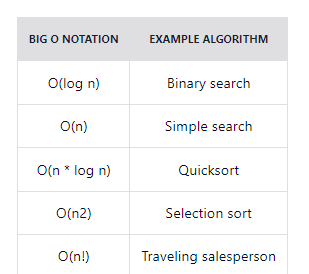
<https://www.simplilearn.com/big-o-notation-in-data-structure-article>

**From lecture slides:**

Avoiding all constants and will be written in Big Oh notation as T(n) = O(𝒏 𝟑 ), that means T(n) grows asymptotically no faster than n 3

* Linear search step by step algorithm can be found in Lecture slide (LO 2 & 5 searching efficiency)

****

****

**Screenshot taken from:**

[**https://www.freecodecamp.org/news/big-o-notation-explained-with-examples/**](https://www.freecodecamp.org/news/big-o-notation-explained-with-examples/)

**Deletion I binary search tree:**

* If the node is a **leaf node** (node without a child) is to be deleted. It can be directly removed
* If the node has **one child**: take the node you want to delete and swap it with the child node, and simply removed the node after that.
* If the node has **two children**: In this case you have to use in order traversal (left-root -right). After traversing, swap/replace the node that you want to delete with its greater node in the list. After that, simply remove the node.

**Traversals:**

* **In order** = Left -Root -Right.
* **Pre-order** = Root – Left – Right.
* **Post order** = Left -Right -Root.

**Insertion in binary search tree:**

* **Compare** the key value with the root (if less choose left subtree, else choose right)
* **Compare** the key value with the subtree root repeat step 1 until the leaf node is reached
* **When the leaf node is reached**, compare the key with the leaf node, if key is greater than the leaf node value, insert the key to the right side of the node else insert it to the left side.

**LO 3 Sorting Algorithms:**

**Quicksort: First item is the pivot element**

* Array recursively divided into two partitions and recursively sorted
* Division based on a pivot item, it can be any item, for simplicity choose first item as pivot item
* pivot item divides the array into two sub-arrays
* **All items <= pivot placed in sub-array before pivot (less)**
* **All items >= pivot placed in sub-array after pivot (greater)**
* Basic Operation:
* Comparison of S[i] with pivot item
* Input size n

**LO 6: Complexity Class:**

[LO 6\_Computability and Complexity\_PG4200.pdf](file:///C:\Users\Usman%20Ahmad\Downloads\LO%206_Computability%20and%20Complexity_PG4200.pdf)

* Some problems exist whose solutions are not yet found; The problems are divided into Complexity Classes
* Complexity classes help group problems based on how much time and space they require to solve problems and verify the solutions
* Common resources are **time and space** (How long the algorithm takes to solve the problem and how much space or memory)
* Time complexity of an algorithm is used to describe (number of steps required to solve a problem) and how long it takes to verify the answer.
* Space algorithm describes how much space/memory is required for the algorithm to operate
* **Types of Classes:**
* **P Class (Polynomial Time)**
* **NP Class (Non-deterministic Polynomial Time)**
* **Co-NP Class (Complement of NP class)**
* **NP -hard Class**
* **NP- Complete Class**

**P Class (Polynomial Time):**

* It is the collection of decision problems **(problems with a “yes” or “no” answer)** that can be solved by a deterministic machine in **polynomial time.**
* **Easy** to find
* P is often a class of computational problems that are **solvable** and **tractable**.
* **Tractable** means that the problems can be solved in **theory** as well as **in practice**. But the problems that can be solved in theory but not in practice are known as **intractable.**
* This class contains many natural problems like:
* (nlgn) for sorting.
* (lgn) for searching.

**NP Class:**

**NP Class (Non-Deterministic Polynomial Time)**

The NP in NP class stands for Non-deterministic Polynomial Time.

* It is the collection of decision problems that can **be solved by a non-deterministic machine in polynomial time.**
* **Features:**
* The solutions of the NP class are **hard to find but are easy** to verify**.**
* A Turing Test (TM) can verify problems of NP in polynomial time.
* **Example:**
* Suppose a company has a total of 1000 employees with unique employee IDs.
* Assume that there are 200 rooms available for them.
* A selection of 200 employees must be paired, but the company’s CEO has the data of some employees who can’t work in the same room for personal reasons.
* This is an example of an NP problem. Since it is easy to check if the given choice of 200 employees proposed by a co-worker is satisfactory or not, i.e., no pair taken from the co-worker list appears on the list given by the CEO.
* But generating such a list from scratch seems so hard to be completely impractical.
* It indicates that if someone can solve the problem, we can find the correct and incorrect pair in polynomial time.
* Thus, the answer is possible for the NP class problem, which can be calculated in polynomial time.

**Co-NP Class (Complement of NP)**

* Co-NP stands for the **complement of NP Class**.
* It means if the answer to a problem in **Co-NP is No**, then there is proof that can be checked in polynomial time.
* **Features:**
* If a problem X is in NP, then its complement X’ is also in CoNP. **(foreleseren forklarte dette I forelesningen husk å sjekke det)**
* For an **NP and CoNP** problem, **there is no need to verify** **all the answers** **at once** in polynomial time, there is a need to verify only **one particular answer, “yes” or “no”** in polynomial time for a problem to be in NP or CoNP.

**Some example problems for Co-NP are:**

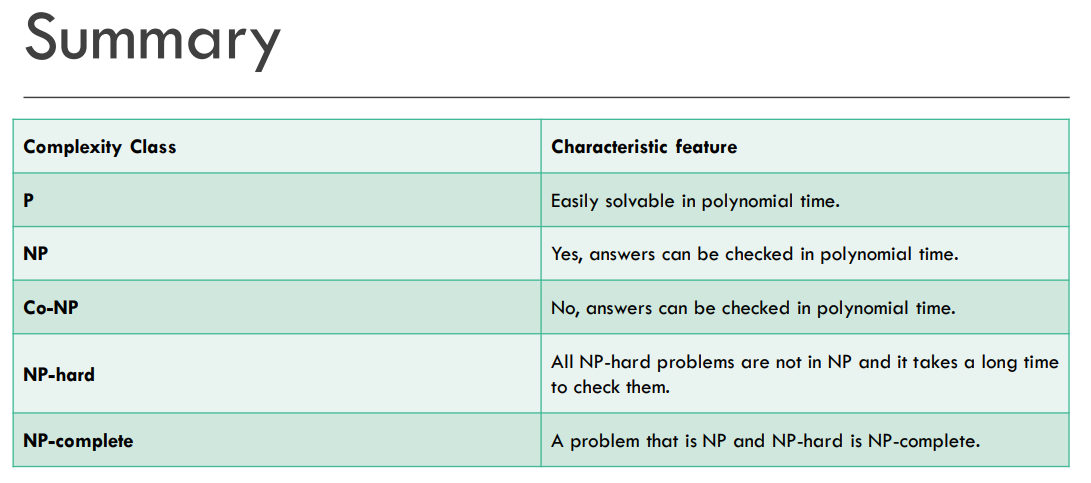
* To check the prime number.
* Integer Factorization.

**NP-hard Class**

* **An NP-hard problem is at least as hard as the hardest problem in NP**, and it is a class of problems such that every problem in NP reduces to NP-hard.
* **Features:**
* All NP-hard problems are not in NP.
* It takes a long time to check them. This means if a solution for an NP-hard problem is given, it takes a long time to check whether it is right or not.

**NP-complete Class**

* A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.
* **Features:**
* NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.
* If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.



**LO 1:**

* Data structures (a set of algorithms) stores and organize the data, whereas algorithms process the data meaningfully and efficiently.
* LinkedList contains two parts, i.e., the first is the data part, and the second is the address part.
* Types of linked lists: **Singly LinkedList, Doubly LinkedList, Circular Doubly LinkedList, Circular Singly Link List**
* Singly LinkedList: Only forward traversal is possible; we cannot reverse it.
* Stacks follows the LIFO principle (Last in First out)
* Queue follow the FIFO principle (First in First Out)
* Overflow in stack occurs when it’s full.