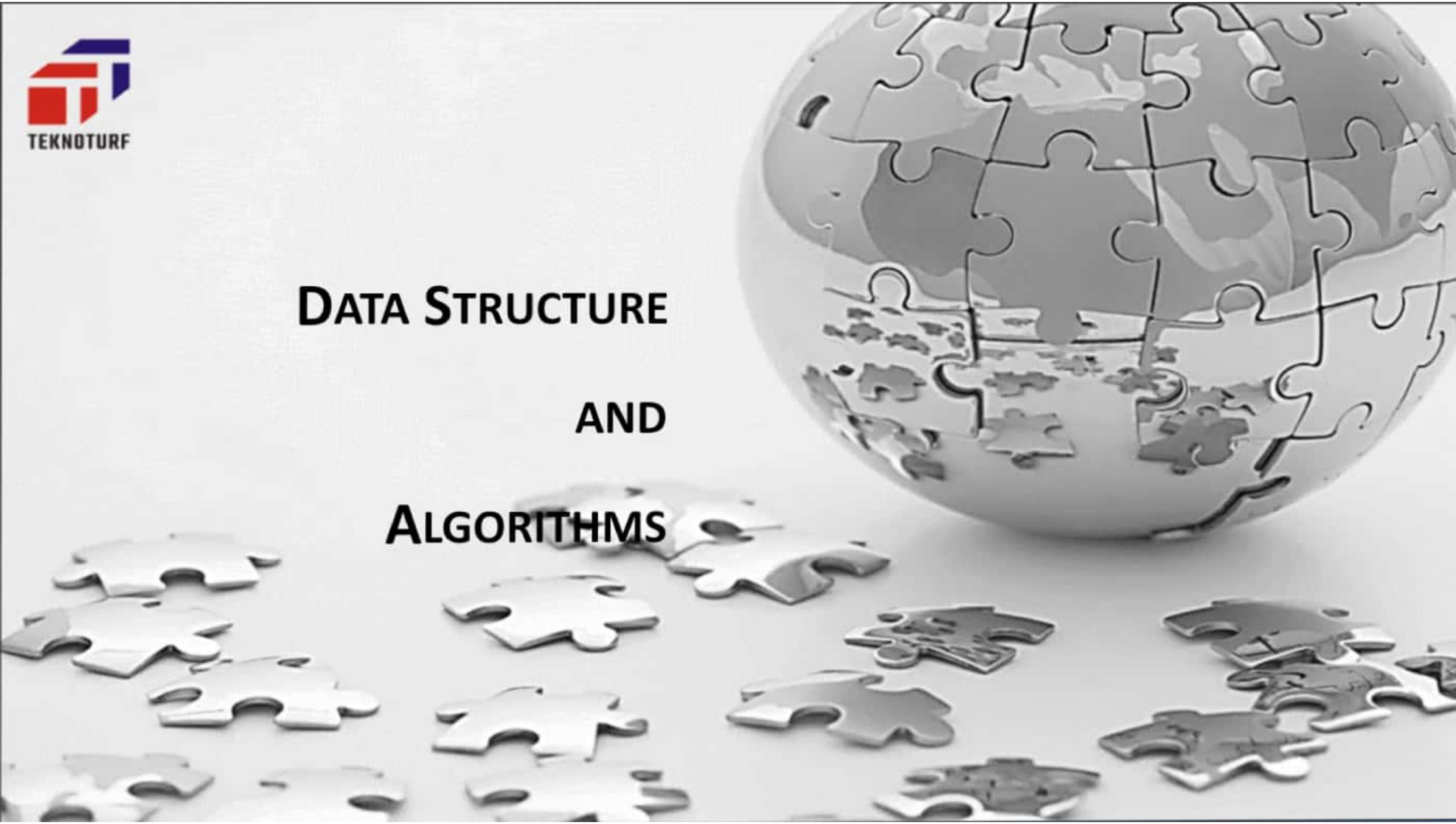
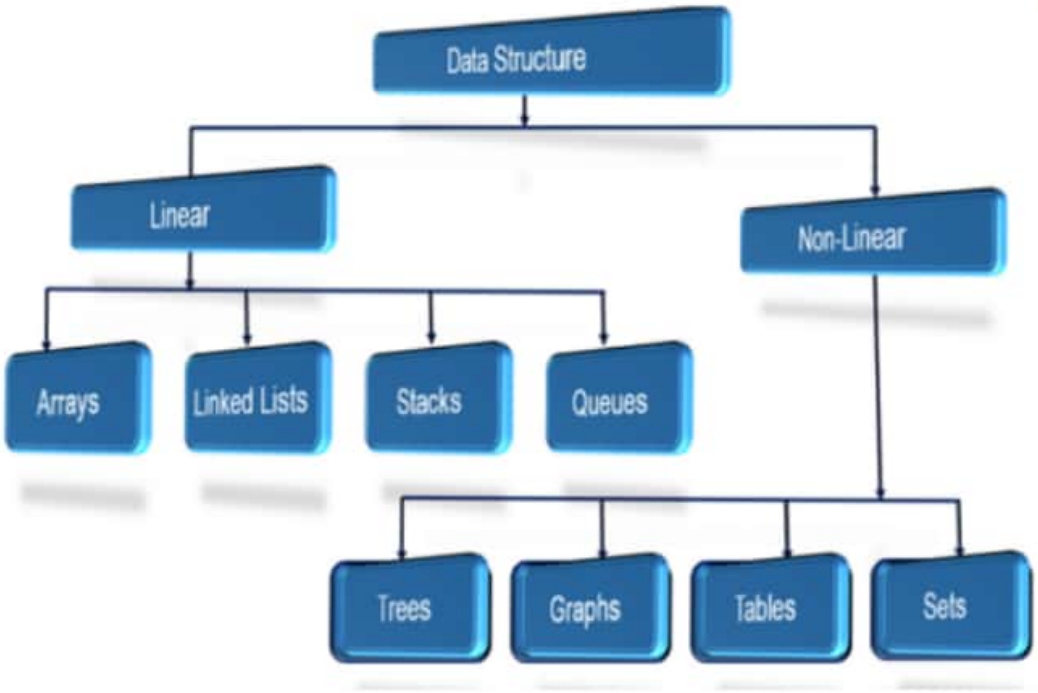


Overview



Likewise; In Programming,
if the data are arranged in a particular order, it will be easier to store and retrieve data faster. **Data Structure** helps us to do it!!!
Let's try to understand Data Structure through this session!

In this module you will learn

TEKNOTURF

- Introduction to Data Structures and Algorithm
- Problem solving
- Abstract Data types
- Linear Data Structures
  - Arrays
  - Linked List
  - Stack
  - Queue

Problem Solving

TEKNOTURF

In the programming world,
a problem solution has 2 main components

Data Structures — Appropriate Algorithm

Note:

- DS has implementation of algorithm and algorithm demands appropriate DS
- There is no ultimate solution for a problem; only better solutions.

Algorithms + Data Structures = Programs

Data Structure

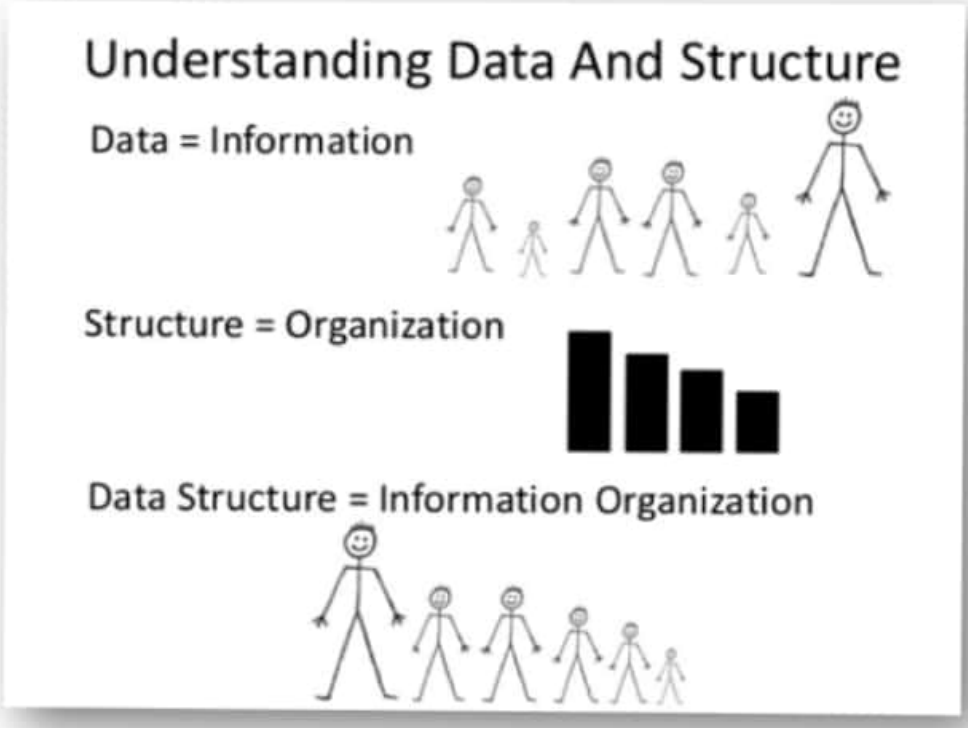
TEKNOTURF

Data Structure is a method of storing and organizing data in a computer that can be retrieved and used efficiently

Choosing the right data structure will allow the most efficient algorithm to be used

A well-designed data structure :

- allows a variety of critical operations to be performed (algorithms)
- monitors both execution time and memory space (analysis)



Algorithms

TEKNOTURF

It is a problem solving procedure

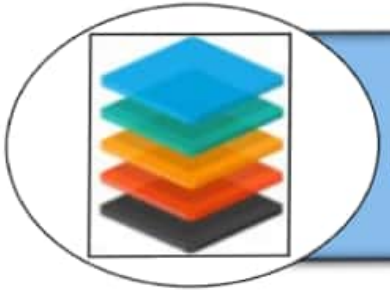
A step by step description for performing a task within a finite period of time

It often operates on collection of data, which is stored in a structured way in the computer's memory

We need algorithms for efficiently performing operations, without any wastage of resources and time.

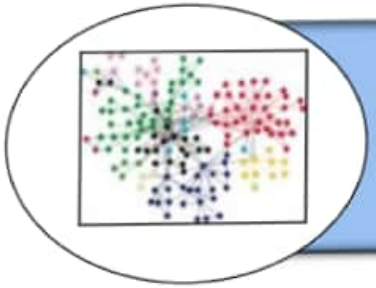
Abstract Data Type (ADT)

TEKNOTURF



Represents a collection of data and set of operations on that data

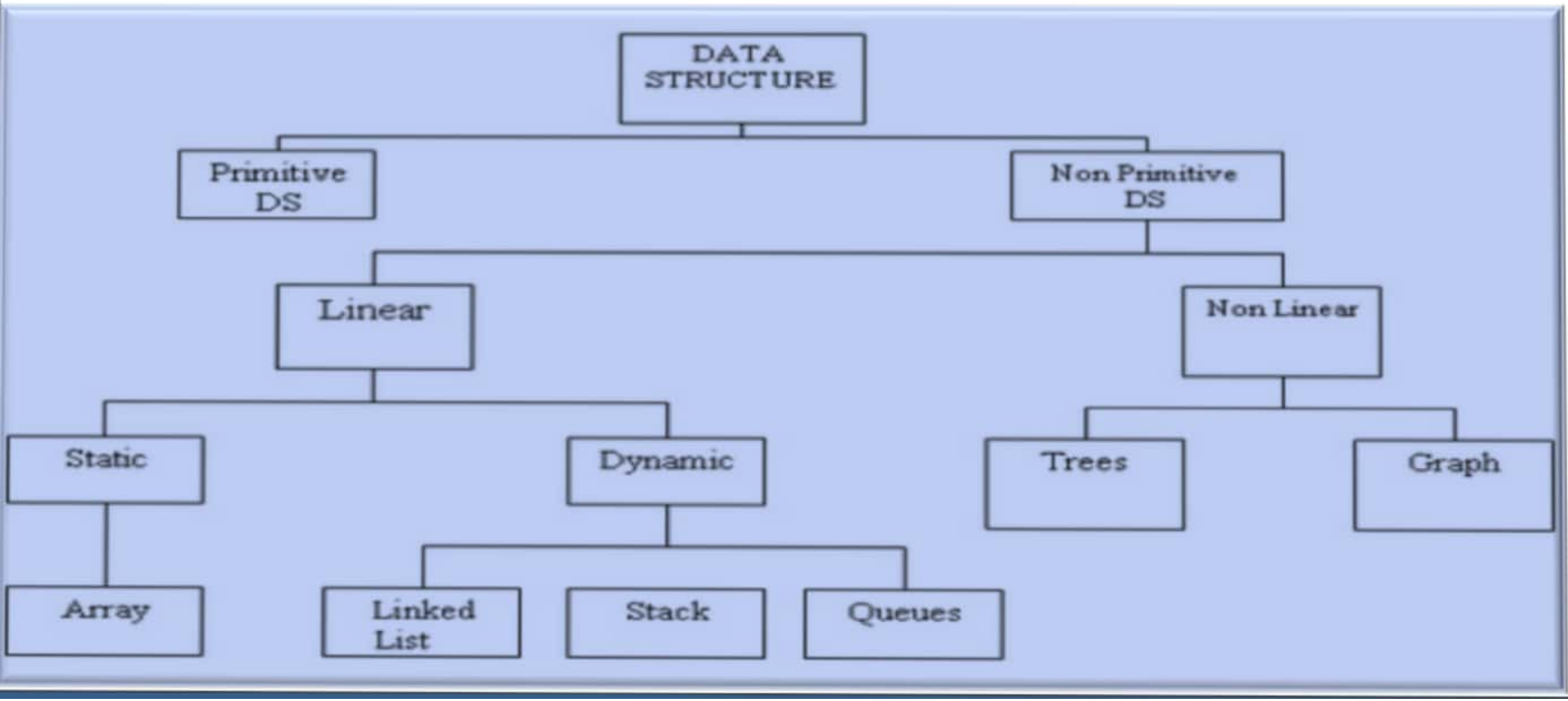
ADT is a logical description of how we view the data and the operations that are allowed regardless of how they will be implemented



Implementation of an ADT: Includes choosing a particular data structure

Data Structure Categories

TEKNOTURF



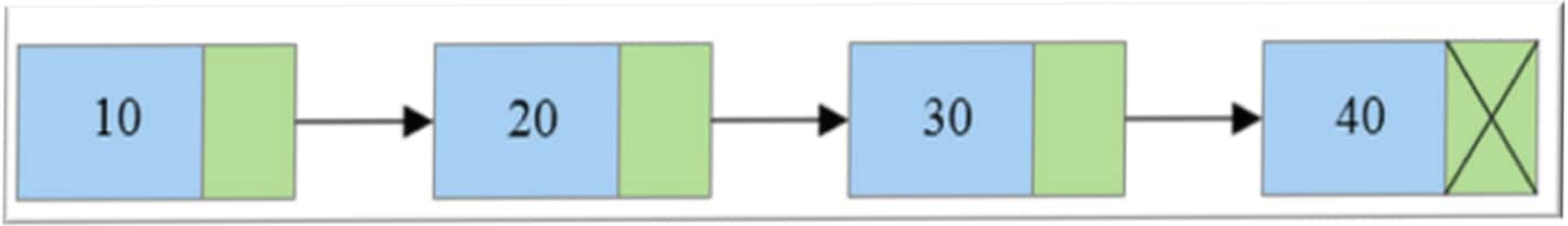
Linear Data Structure

TEKNOTURF

Data items can be traversed in a single run

In linear data structure, data is arranged in linear sequence

In linear data structure, elements are accessed or placed in contiguous (together in sequence) memory locations



Types of Linear Data Structure

TEKNOTURF

Array

Linked List

Stack

Queue

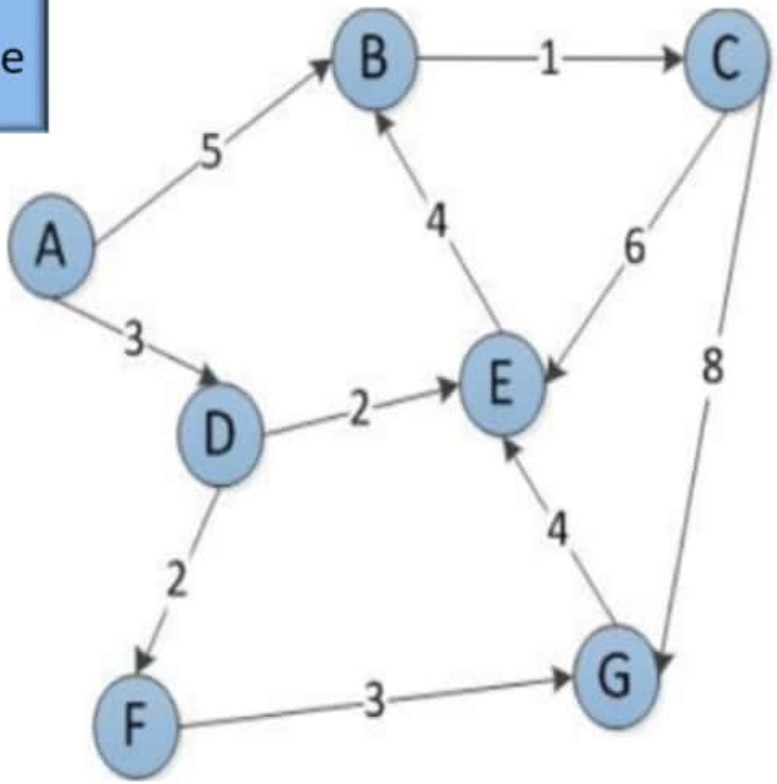
Non-Linear Data Structure

TEKNOTURF

In non-linear data structure, data is not arranged in linear sequence

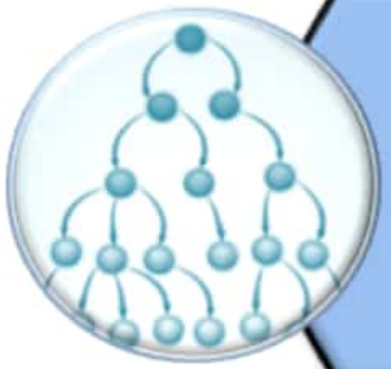
Every item might be attached with many other items

Data cannot be traversed in a single run

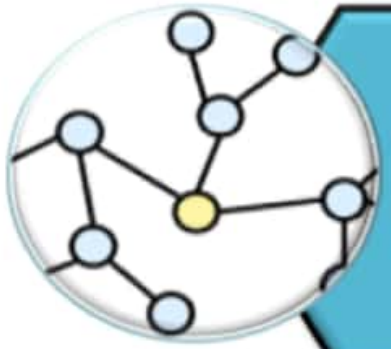


Types Non-Linear Data Structure

TEKNOTURF



Tree



Graph

Data Structure Operations

TEKNOTURF

Insert — Adding a new element to the structure.

Modify — Modifying an existing element in the structure

Delete — Removing a element from the structure

Search — Finding the location of an element with a given key value

Sort — Arranging the elements in some logical order.

Merge — Combining the elements in two different sorted files into a single sorted file.

Traverse — Accessing each element exactly once so that certain items in the element may be processed.

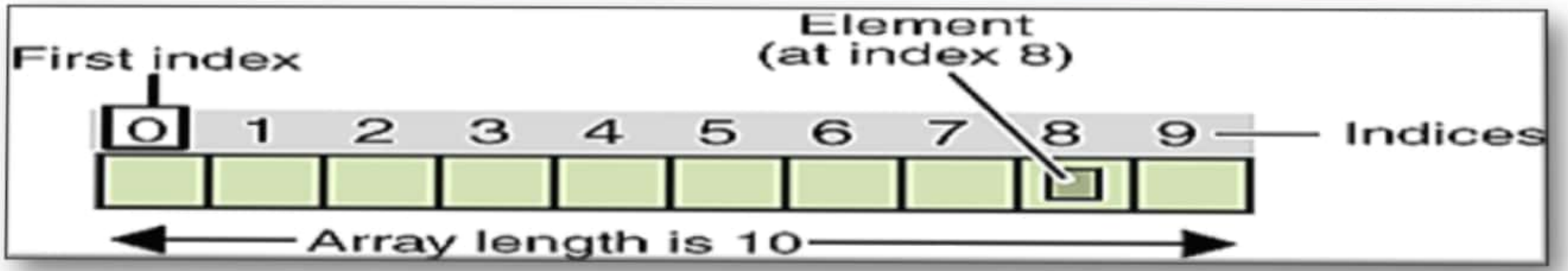
Array

TEKNOTURF

An array is a structured collection of components(called array elements), all of the **same data type, given a single name**, and **stored in adjacent memory locations**.

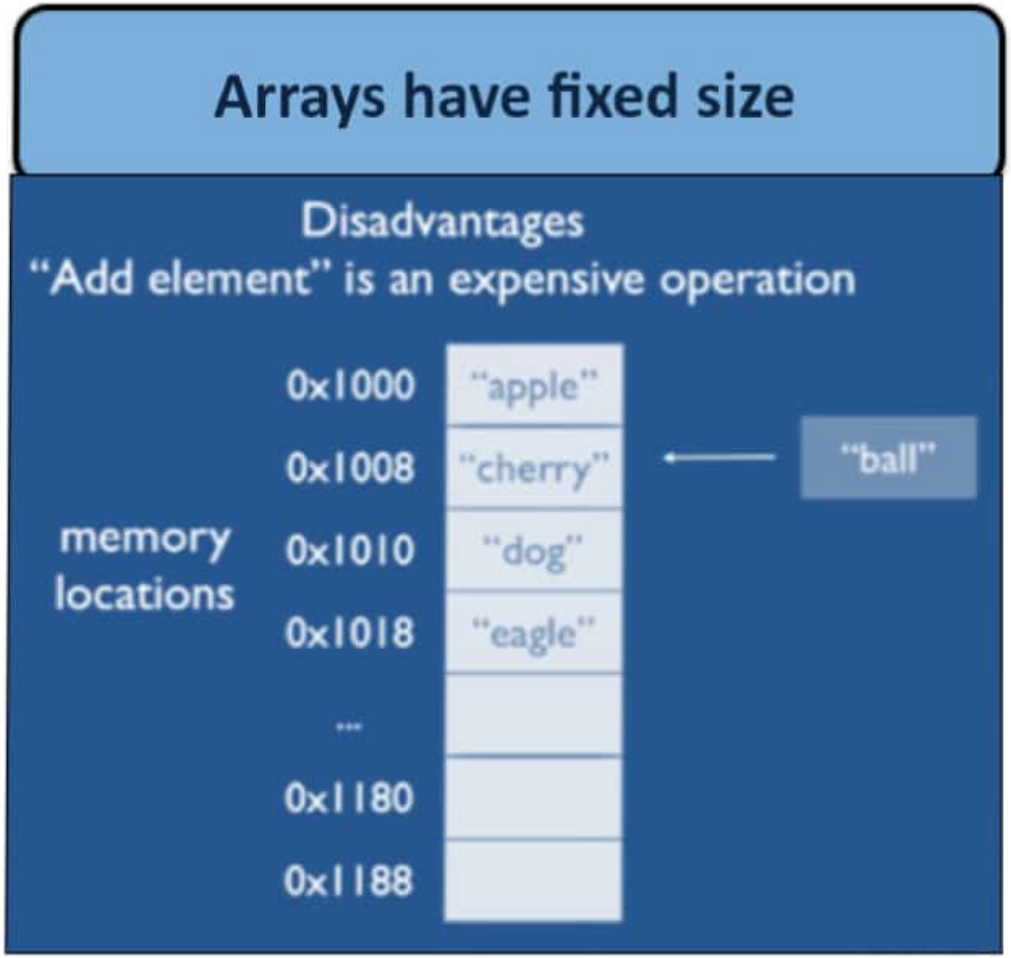
The individual components are accessed by using the array name together with an integral valued index in square brackets.

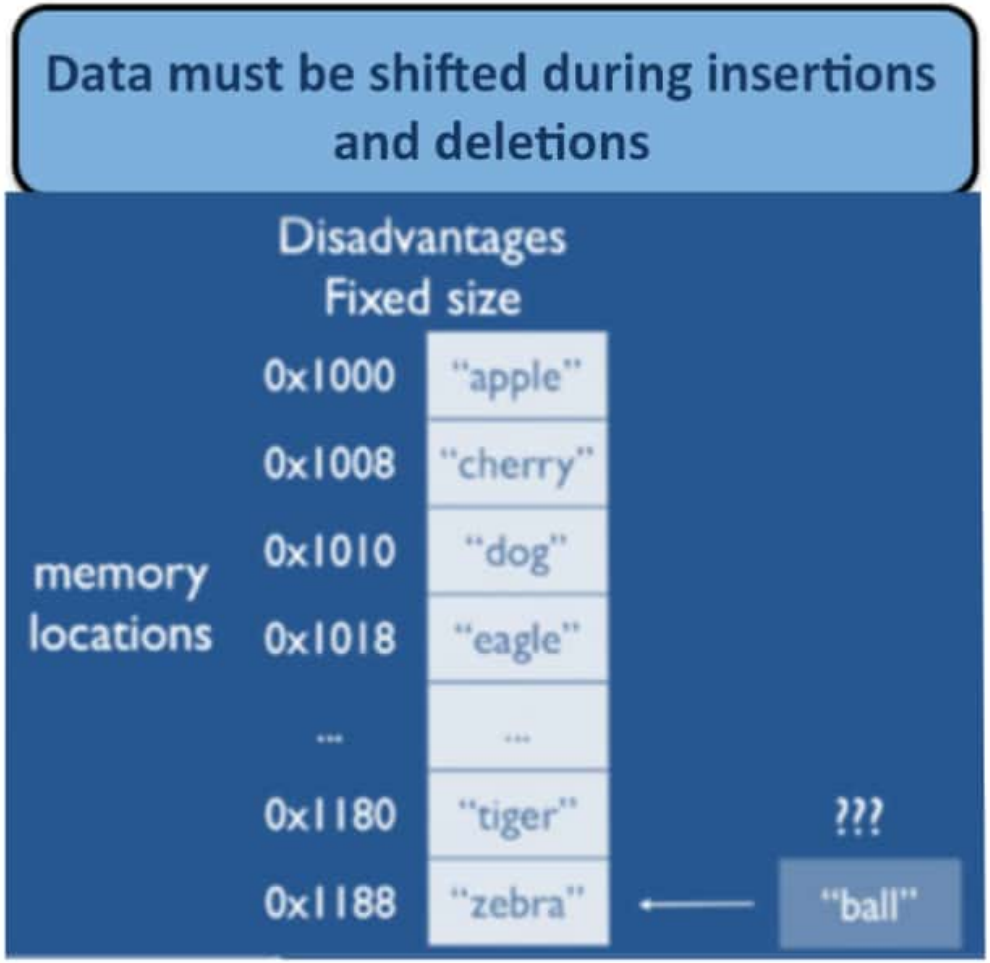
The **index indicates the position** of the component within the collection.



Array

TEKNOTURF





Linear List

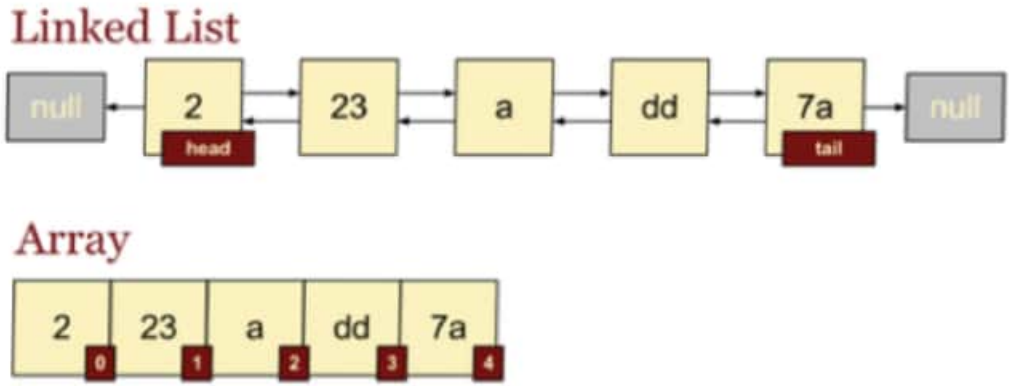
TEKNOTURF

In Computer Science, a list is:

An ordered collection of values (items, entries, elements)

Where a value may occur more than once

How are lists implemented?



Linked List

TEKNOTURF

Linked list is a collection of elements called nodes, arranged in linear sequence

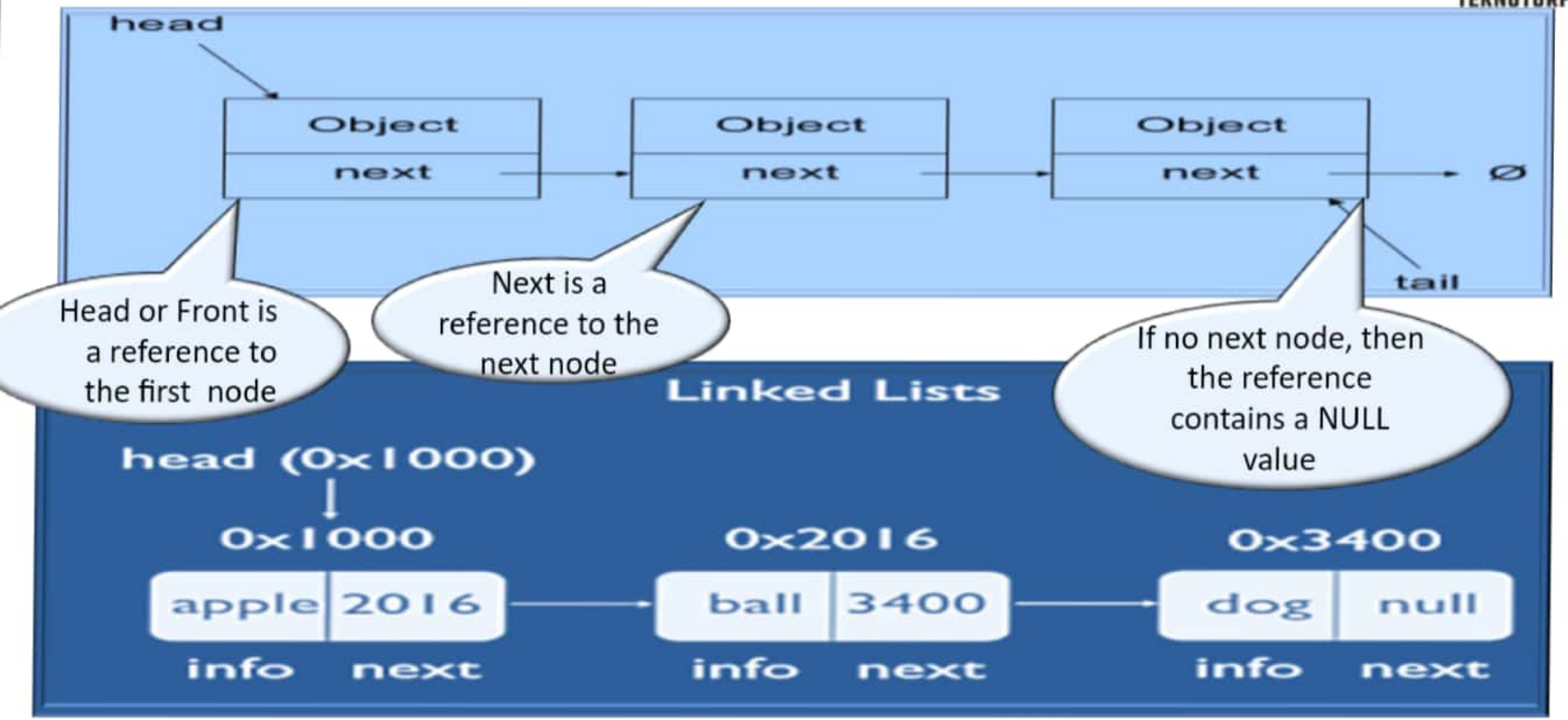
Does not require the shifting of items during insertions and deletions

Linked List is able to grow in size as needed

Each node contains :

- one value and
- one pointer which keeps a pointer to the next node of the list.

Linked List



Operations on Linked List

TEKNOTURF

Insert a new Item	Delete an item	Modify an item
• Insert at the beginning of the list, or • Insert at the end of the list, or • Insert at a specified location of the list	• The item can be specified by position or by some value	• Search and locate the item, then display or modify or remove the item

Linked List Implementation

TEKNOTURF

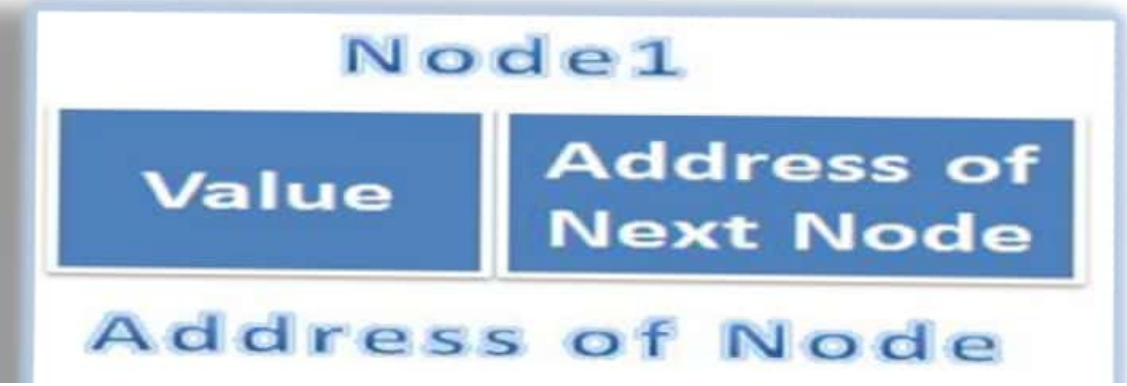
Singly Linked List Declaration

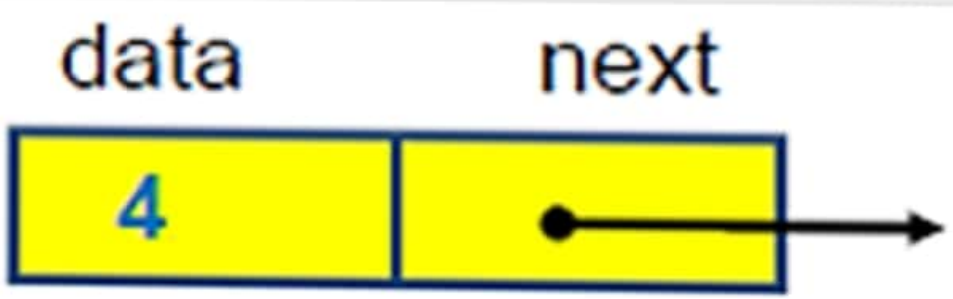
Syntax:

```
class class-name
{
    object declaration;
    class-name Reference to next object;
}
```

Example:

```
class Node {
        Object data;
        Node next;
}
```



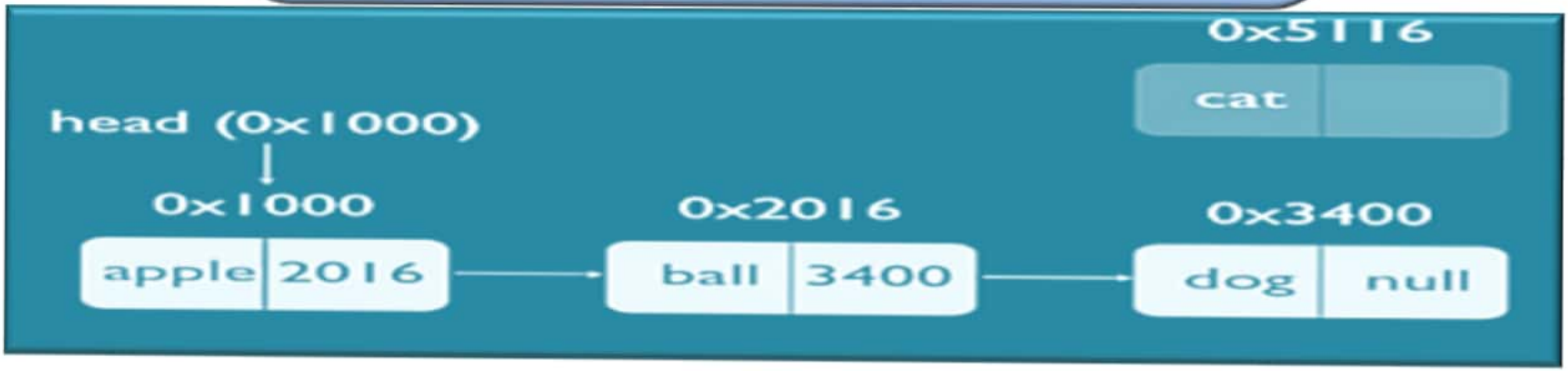


Insert a New Node to the List

TEKNOTURF

Steps to insert a new node:

1. Find the location to insert the element
2. Allocate memory for the new node
3. Point the new node to its successor
4. Point the new node's predecessor to the new node

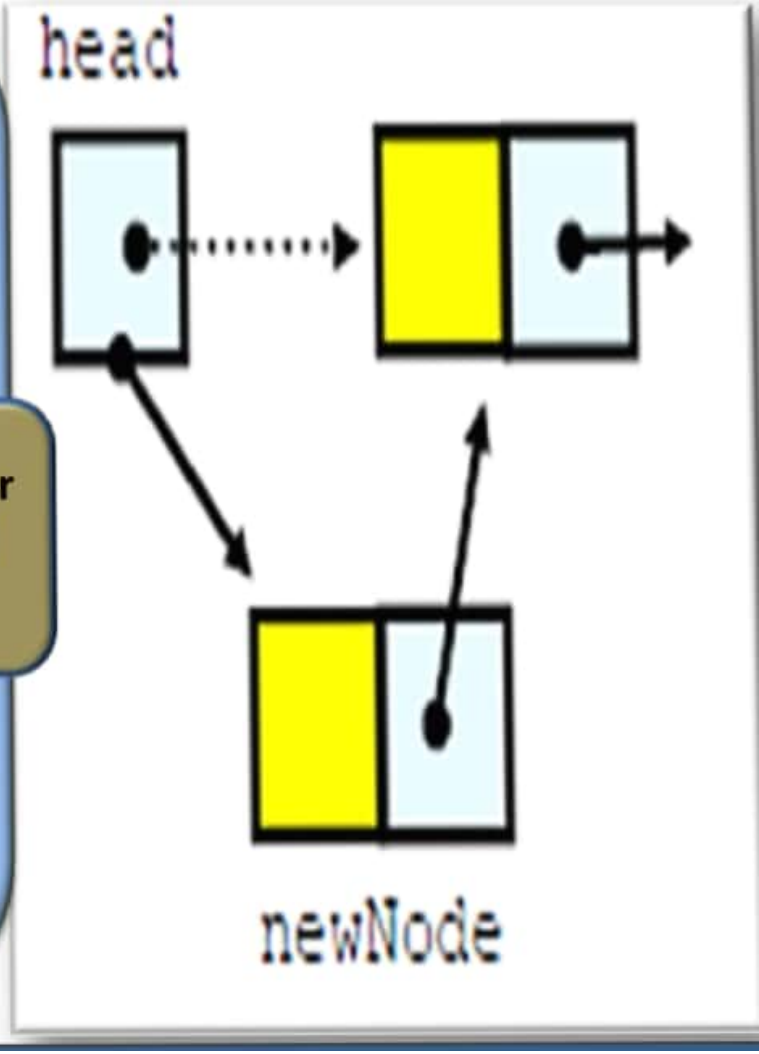


Insert at front or in empty list:

TEKNOTURF

```
class Node {
   Object data;
   Node next;
   public Node(Object data)
  {
       this.data=data;
       next=null;
   }
}
public class LinkedList {
          Node head;
          public void insertInBegin( Object data) {
                Node newNode=new Node(data);
                newNode.next=head;
                head=newNode;
          }
 }
```

Allocate memory for the new node and put data in it

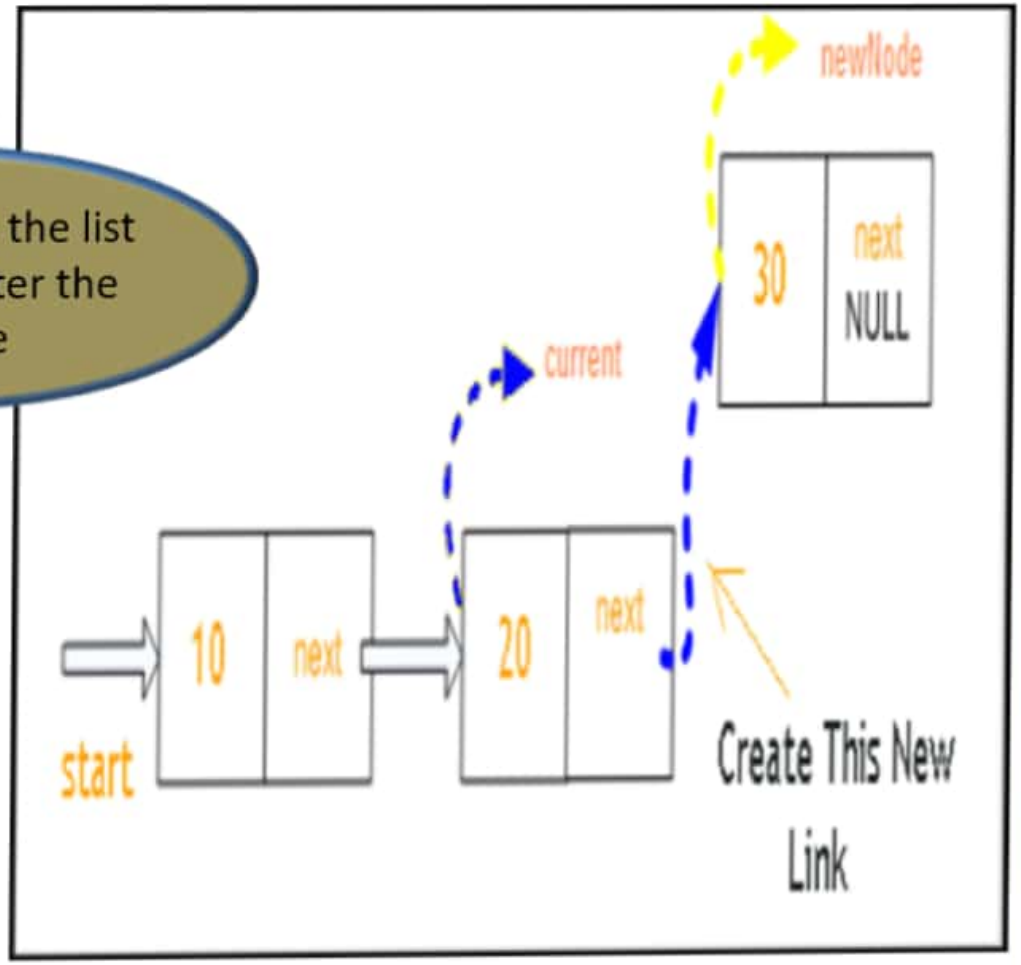


Insert at the end:

TEKNOTURF

```
class LinkedList {
      Node head;
     public void insertAtEnd(Object data) {
            Node newNode=new Node(data);
            Node current=head;
            if(head==null)            {
               head=newNode;
               return;
            }
            else  {
                        while(current.next!=null)  {
                            current=current.next;
                        }
                        current.next=newNode;
                        return;
            }
      }
}
```

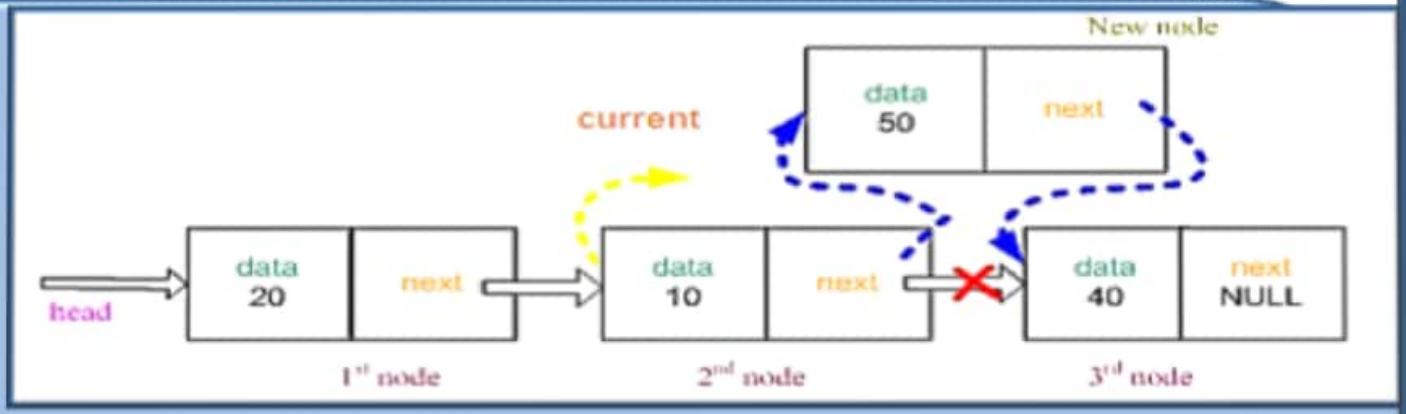
Iterate through the list till we encounter the last node



Insert in the middle

TEKNOTURF

```
public void insertAtMid(Object data){
    if (head == null)
        head = new Node(data);
    else {
        Node newNode = new Node(data);
        Node current = head;
        int len = 0;
        while (current != null) {
            len++;
            current = current.next;
         }
         int count = ((len % 2) == 0) ? (len / 2) : (len + 1) / 2;    current= head;
         while (count > 1) {
                    current= current.next;
                    count--;    }
            newNode.next = current.next;
            current.next = newNode;
        }}
}
```



Calculate length of the linked list, i.e., the number of nodes

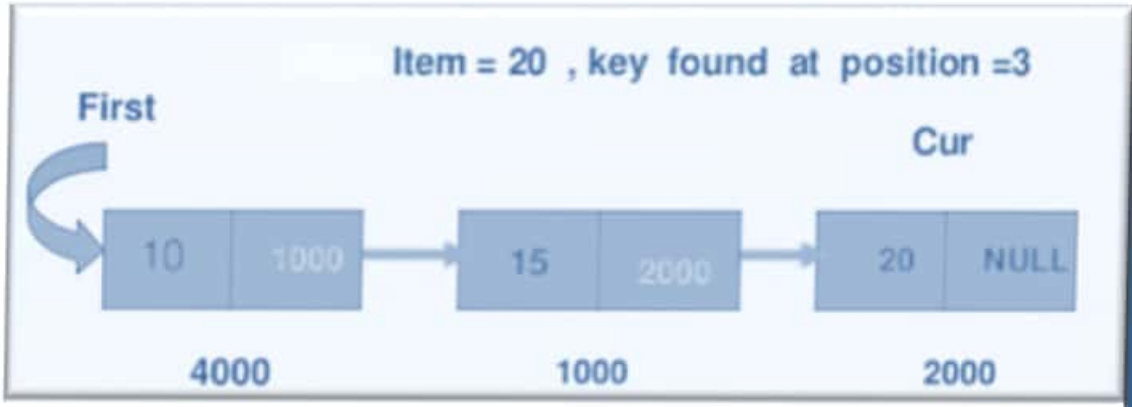
Count the number of nodes after which, the new node is to be inserted

Find and print elements

TEKNOTURF

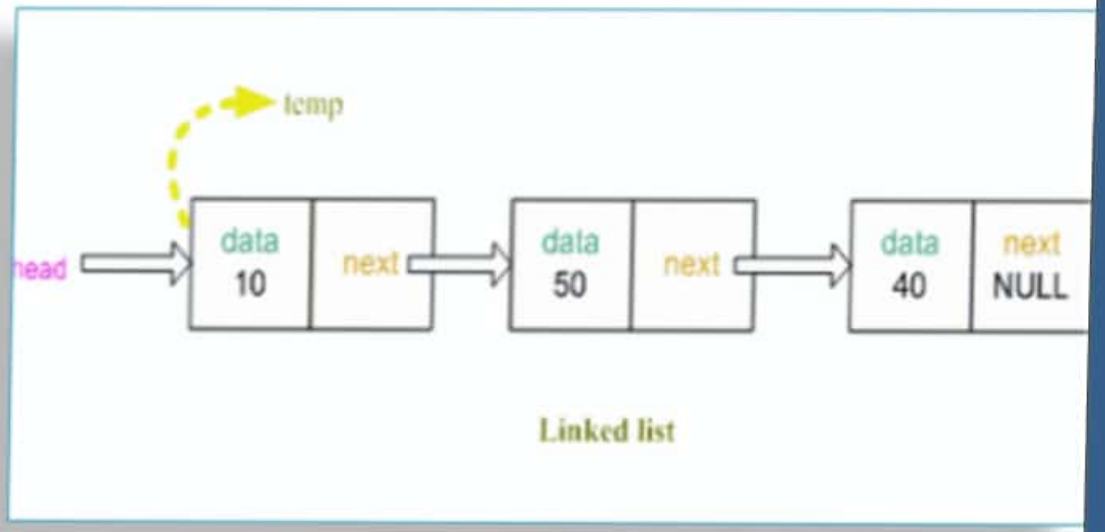
Find an element in the list

```
public static Node search(Object key)  {
        while (head != null && key != head.item)
                head = head.next;
        return head;
}
```



Display all elements in the list

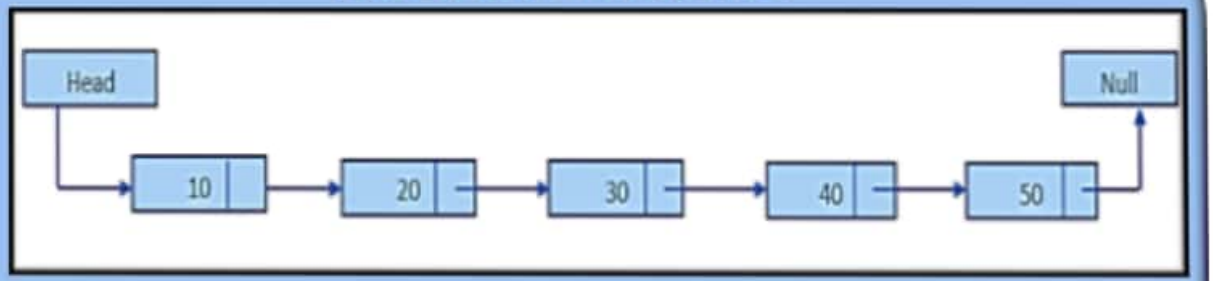
```
public void printList()  {
      Node temp = head;
      while (temp!= null)  {
         System.out.print(temp.data+" ");
         temp =temp.next;
      }
   }
```

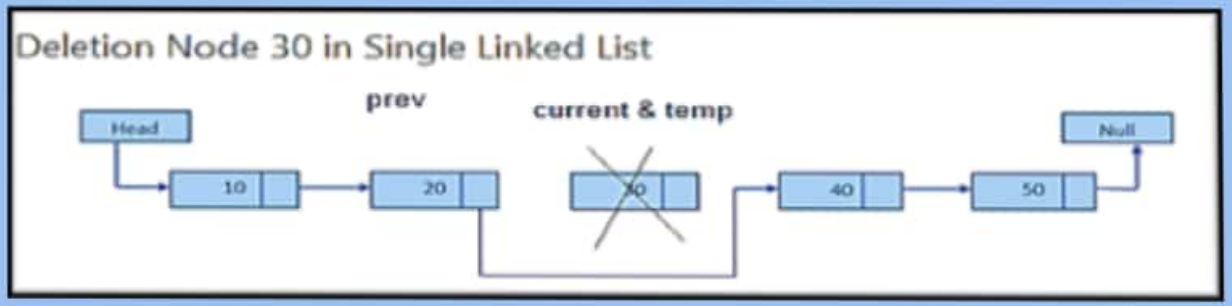


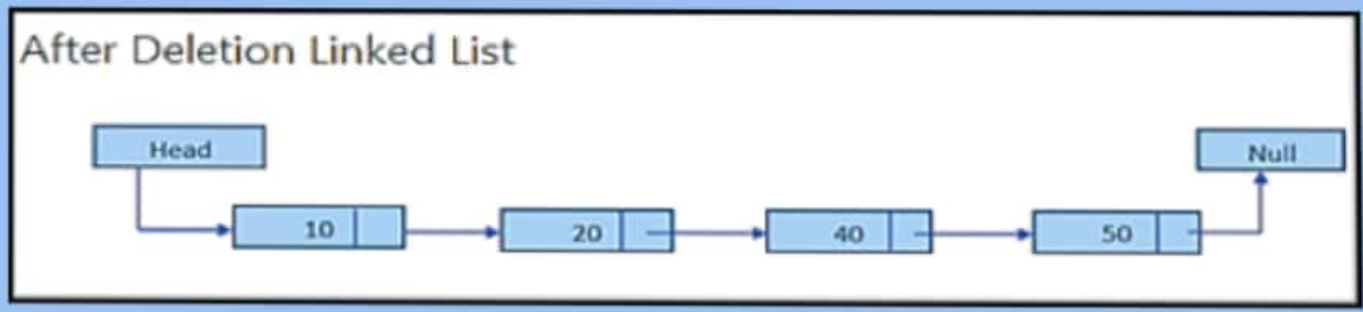
Delete an element

```
public void delete(Object key){
    Node temp = head, prev = null;
    if (temp != null && temp.item == key)  {
        head = temp.next; // Changed head
            return;
    }
    while (temp != null && temp.item != key)
    {
        prev = temp;
        temp = temp.next;
    }
    if (temp == null)
            return;
      prev.next = temp.next;
}
```

Deletion Procedure



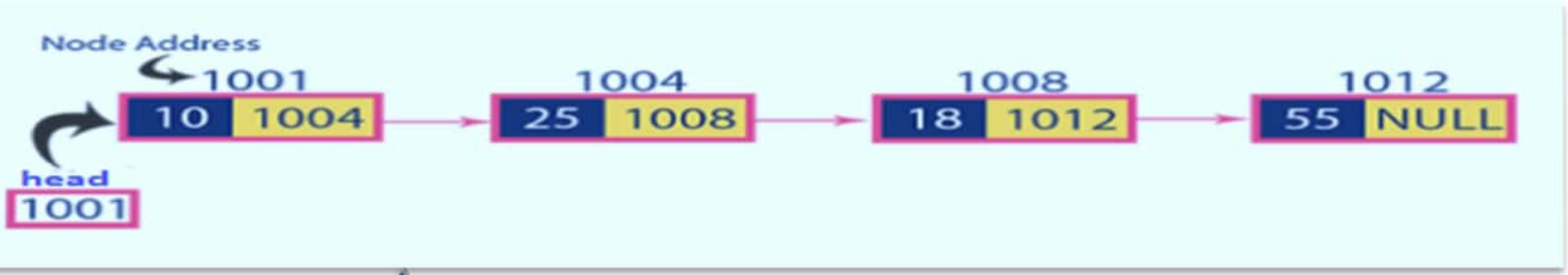




Linked list variations

TEKNOTURF

Singly Linked List



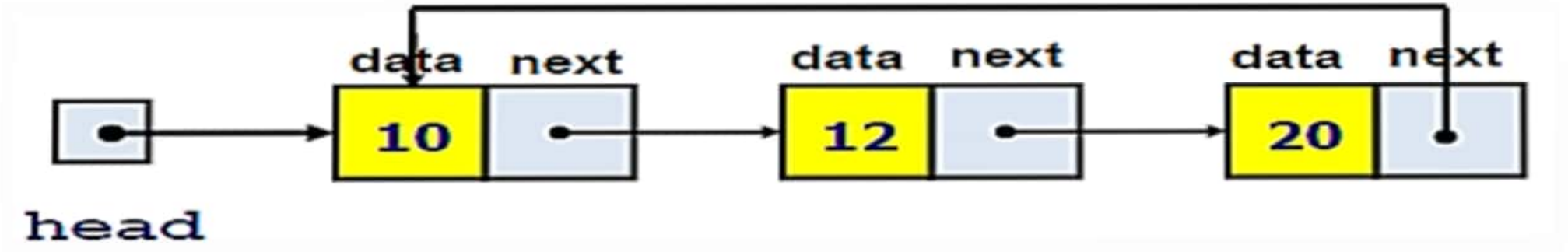
It consists of a list of elements, with a head and a tail; with each element referring to another of its own kind

Each node contains at least
– A piece of data of any type
– reference to the next node

Linked list variations

TEKNOTURF

Circular Linked List



Circular linked list contains a series of connected nodes with the last node referring to the first node of the list.

Linked list variations

TEKNOTURF

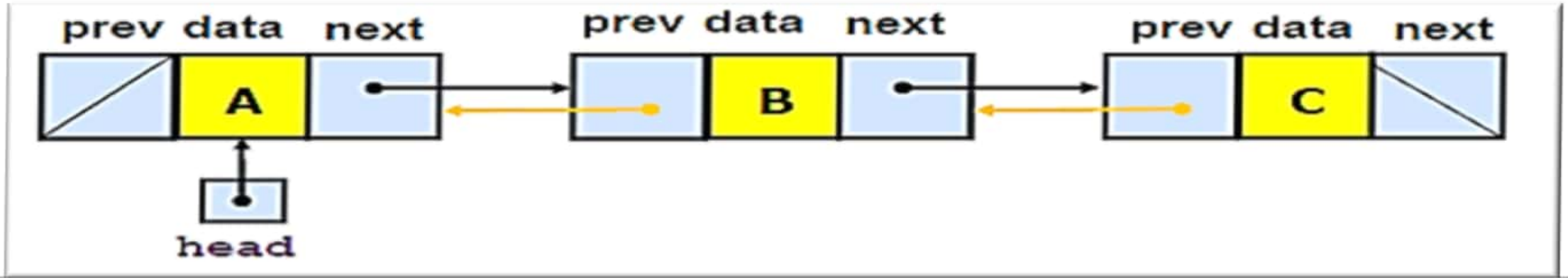
Doubly Linked List

Each node in doubly linked list has two reference variables:

- One refers to the next node and
- The other refers to the previous node

There are two NULLs: First and last nodes in the list

Advantages: For a given a node, it is easy to visit its predecessor and the successor. This makes it convenient to traverse lists backwards and forwards.



Choose the correct data structure whose size is fixed and insertion and deletions are bit difficult because we have to shift the elements upward or downward to avoid the wastage of memory.

- ✔ (●) Array
- ( ) Linked List
- ( ) Tree
- ( ) Graph

Correct

That's right! You selected the correct response.

_____is a header list where the last node points back to the header node.

- Grounder Header List
- Doubly header List
- ✔ Circular Header List
- Singly header List

Correct

That's right! You selected the correct response.

_____ operation is not performed in linear list.
1. Insertion 2. Deletion 3. Retrieval 4. Traversal

- [x] None of these options
- [ ] Only 3
- [ ] Only 1 and 2
- [ ] Only 1, 2 and 3

Correct

That's right! You selected the correct response.

A linear collection of data elements where the linear node is given by means of a reference is called

- Array
- node list
- ✔ linked list
- primitive list

Correct

That's right! You selected the correct response.

which of the following is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented?

- ✔ (●) Abstract Data Types
- ( ) Array Data Types
- ( ) None of these options
- ( ) Advanced Data types

Correct

That's right! You selected the correct response.

Operations on stack

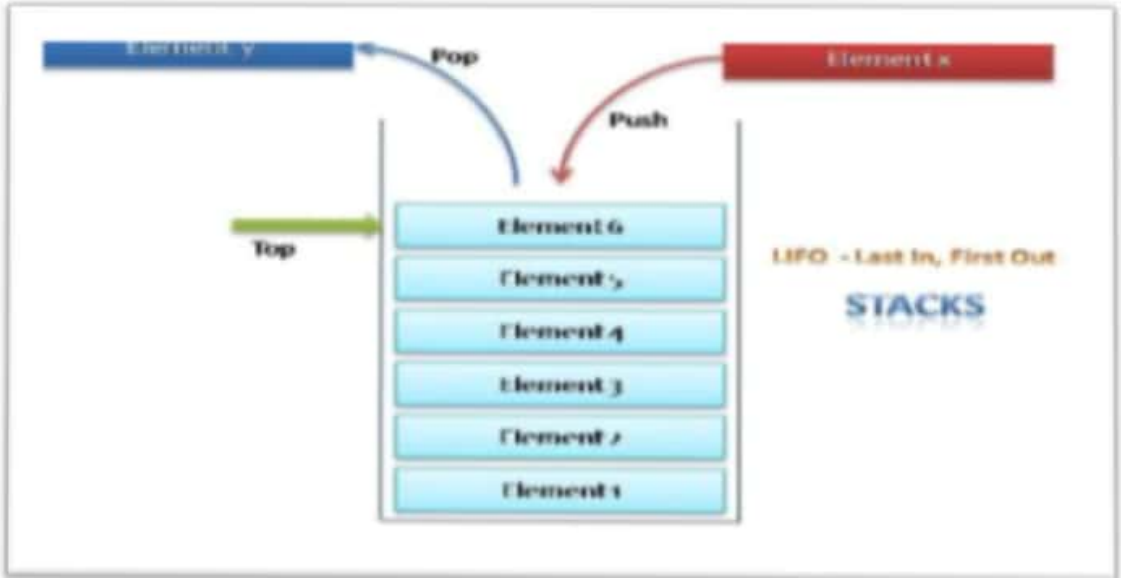
TEKNOTURF

Insert an element into a stack

Remove an element from a stack

Get the topmost element of the stack

A stack is open at one end (the top) only. You can push an element onto the top, or pop the top element out of the stack.



Stack Implementation

TEKNOTURF

Stack can be implemented using array or linked list.

Array Implementation of stack

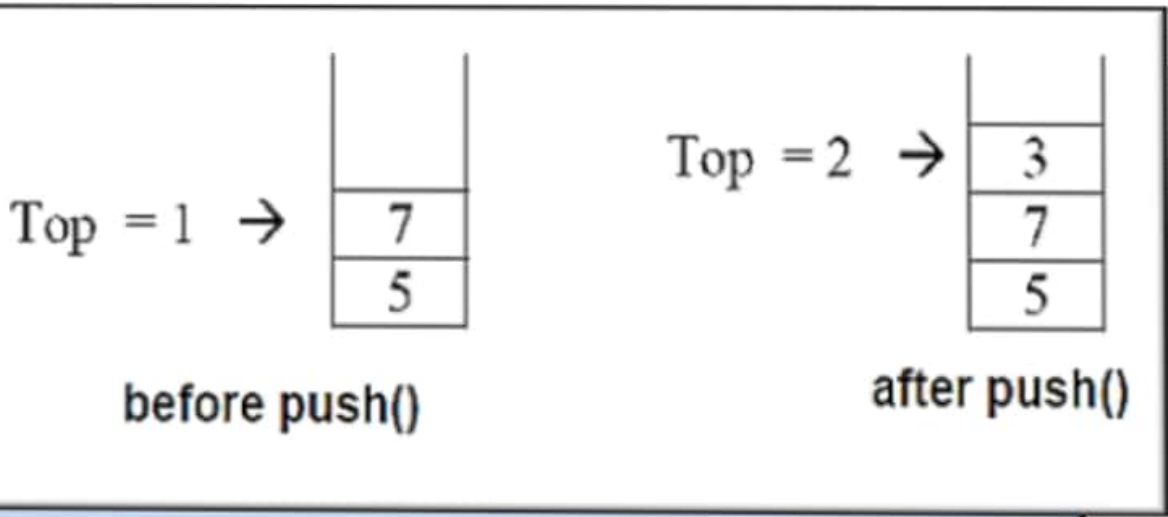
- Size of stack is ***fixed*** during declaration
- Item can be pushed if there is some space available.
- Needs a variable called 'top' to keep track of the top element of a stack.
- Stack is empty when the value of ***top is –1.***

Stack Implementation using Array

TEKNOTURF

Inserting an element into a stack: push()

```
class Stack {
        final static int MaxStack = 100;
        int[] stack = new int[100];
       int top = -1; //set the tops as -1 (empty)
}
```



Insert an Element :push ()

```
public int push(int n)
{
if (top == MaxStack - 1) {
System.out.printf("\nStack Overflow\n");
retutn 0);
}
++top;
stack[top] = n;
return 1;
}
```

Stack is full

top will be increased by 1 and new item will be inserted at the top

return 1 represents successful insertion
return 0 represents unsuccessful insertion

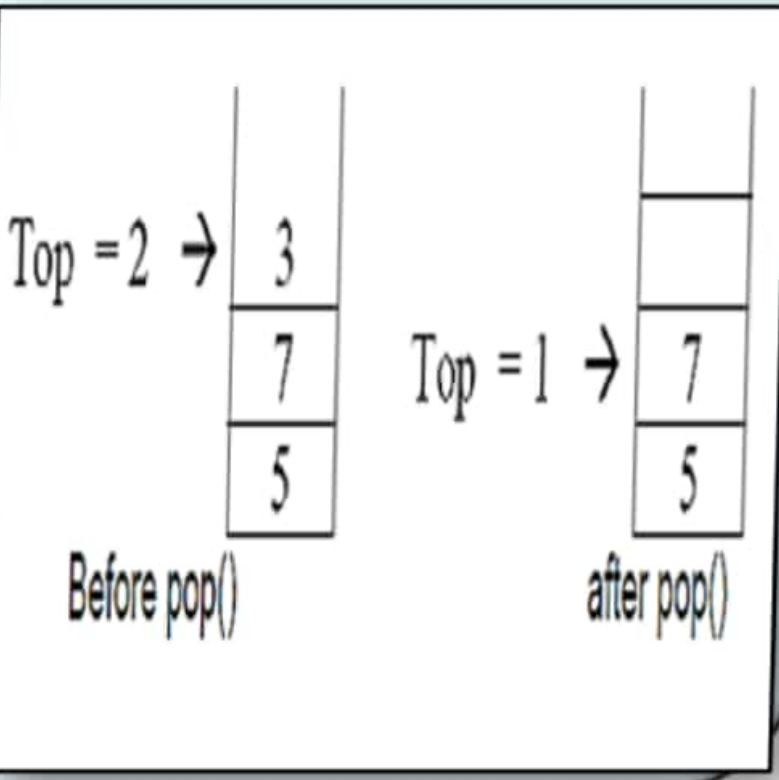
Stack Implementation using Array

TEKNOTURF

Remove an Element from stack :pop ()

```
public int pop()
{
        int temp=0;
        if(top==-1)  {
            return 0;
         }
        else {
              temp=stack[top--];
              return temp;
         }
 }
```

pop() operation will decrease the value of top by 1:



'return temp' represents successful pop operation
and
return 0; represents unsuccessful deletion

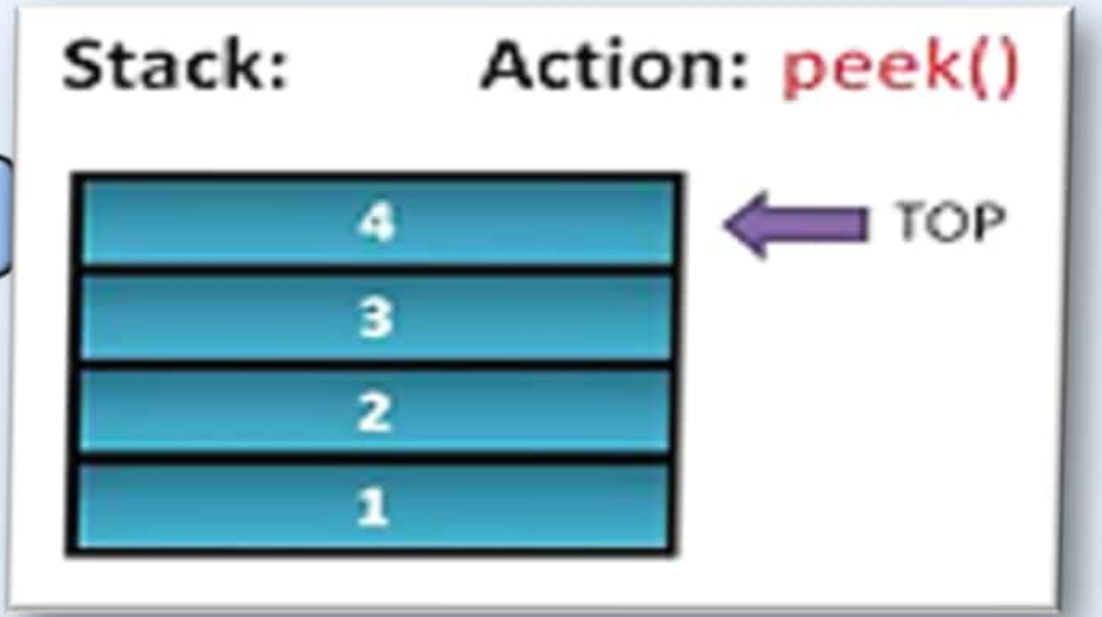
Stack Implementation using Array

TEKNOTURF

Retrieve the topmost element :peek()

```
void peek( )
{
        if(top==-1) {
            return 0;
        }
        else {
            int value=stack[top];
            return value;
        }
  }
```

"Stack is Empty"

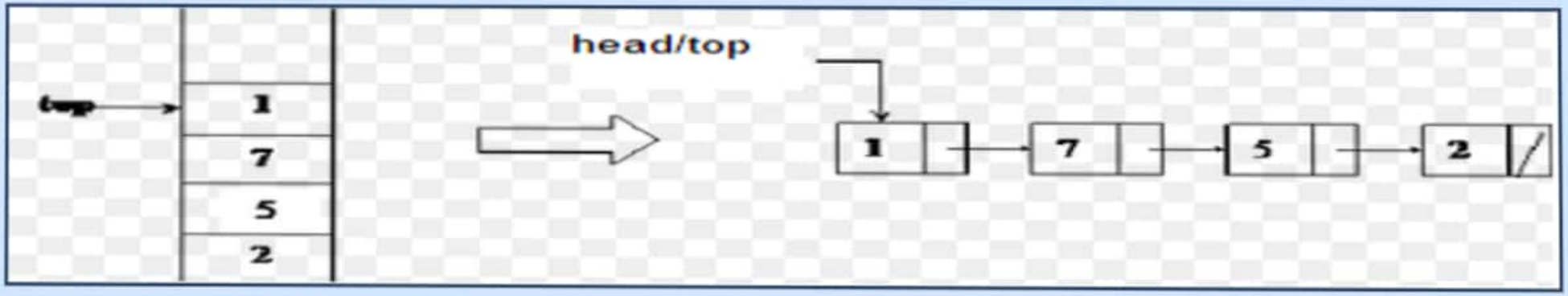


Stack Implementation

Linked List implementation of stack

- Size of stack is ***flexible***. Item can be pushed and popped dynamically.
- Needs a pointer, called ***top*** to point to the top of the stack.
- Here, memory will be assigned to stack when a new node is pushed into stack, and memory will be released when an element is being popped from the stack
- Each node in a stack must contain at least 2 attributes:

  Data (object) and reference to the next node



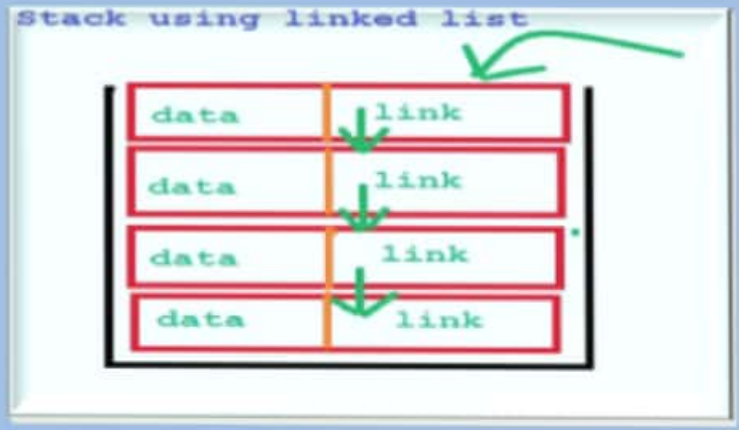
Linked List implementation of stack

TEKNOTURF

Create a class called node with two fields - data and reference object

```
class Node {
        int data;
        Node next;
        public Node(int data) {
                this.data = data;
                next = null;
        }
}
```

Include a constructor to initialize the values



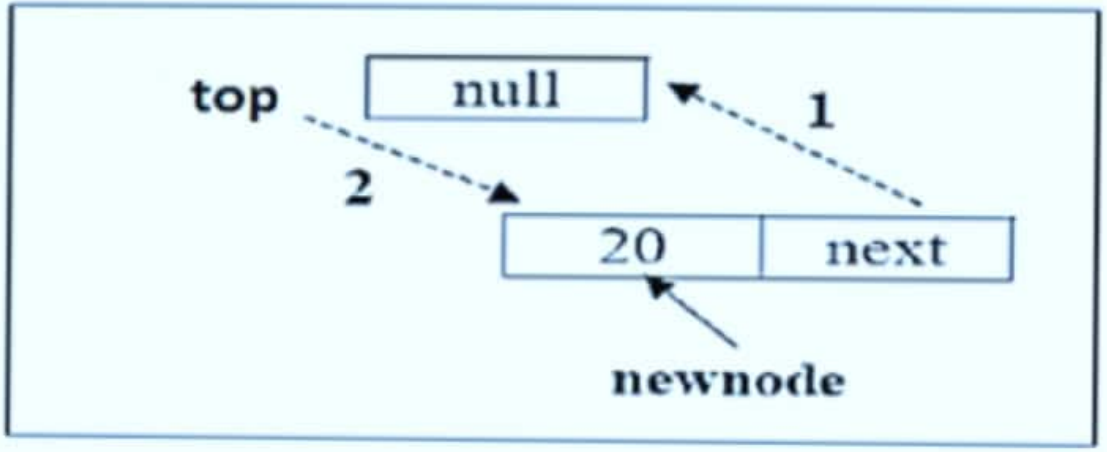
There are two possibilities for inserting an element into a stack:

1. Insertion to an empty stack
2. Insertion to a non empty stack

Linked List implementation of stack

TEKNOTURF

1. Insert an element into an empty stack



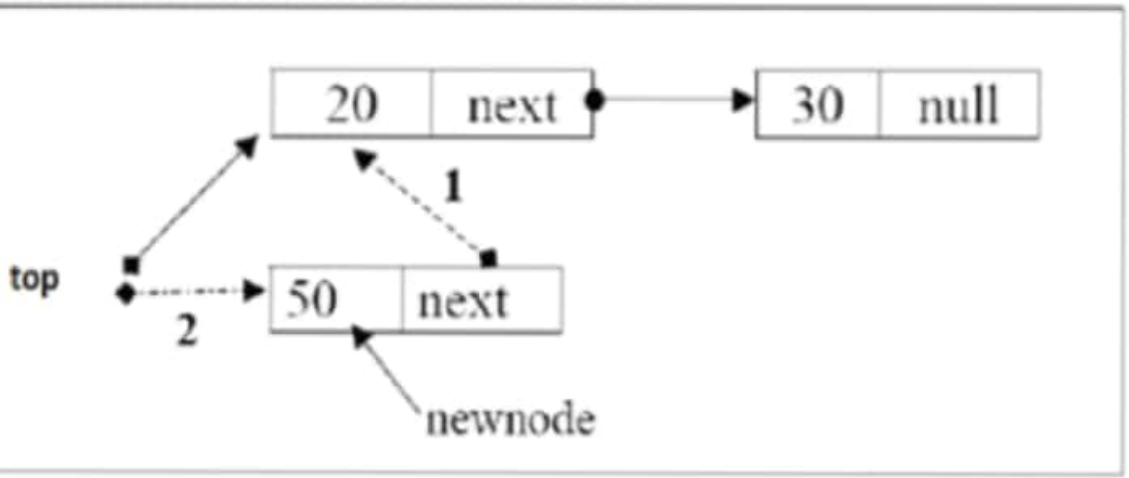
In this situation the new node being inserted, will become the first item in stack.

Linked List implementation of stack

TEKNOTURF

2. Insert an element into a non-empty stack

This operation is similar to insert an element in front of a linked list.



STEP 1 : newNode.next=top;
STEP 2 : top=newNewNode;

```
class linkedStack{
    protected Node top ;
    public void push(int n)  {
        Node newNode= new Node(n);
         newNode.next = top;
         top = newNode;
    }
}
```

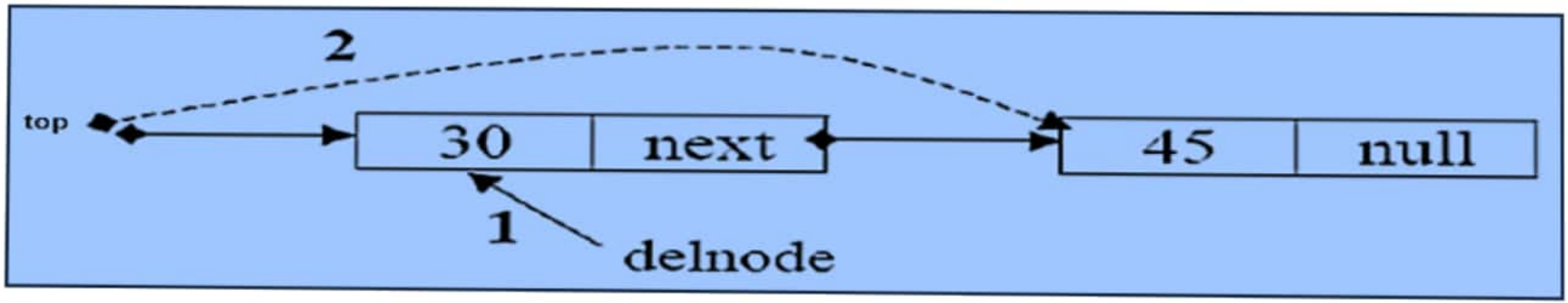
Linked List implementation of stack

TEKNOTURF

pop operation in stack :

```
if(top==NULL)
        print Stack is empty.
        return
else
        step1 : delNode=top
        step 2: top=delNode.next;
        step 3:  return delNode.data;
```

```
public int pop() {
   if (if(top==NULL)
            return 0;
   int deletedValue= top.data;
   top = top.next;
   return deletedValue;
}
```



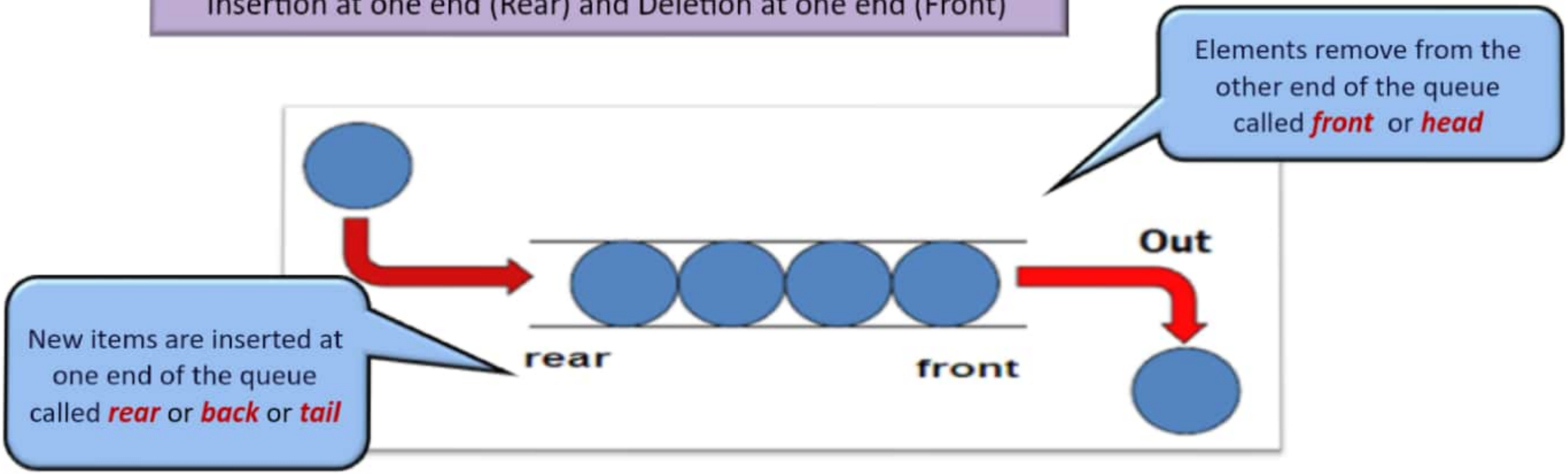
Queue

TEKNOTURF

Linear Data Structure with restrictions

First In First Out (FIFO)

Insertion at one end (Rear) and Deletion at one end (Front)



Applications of Queue

TEKNOTURF

Real-World Applications

- Cashier lines in any store
- Check out at a bookstore
- Bank/ATM
- Call an airline

Computer Science Applications

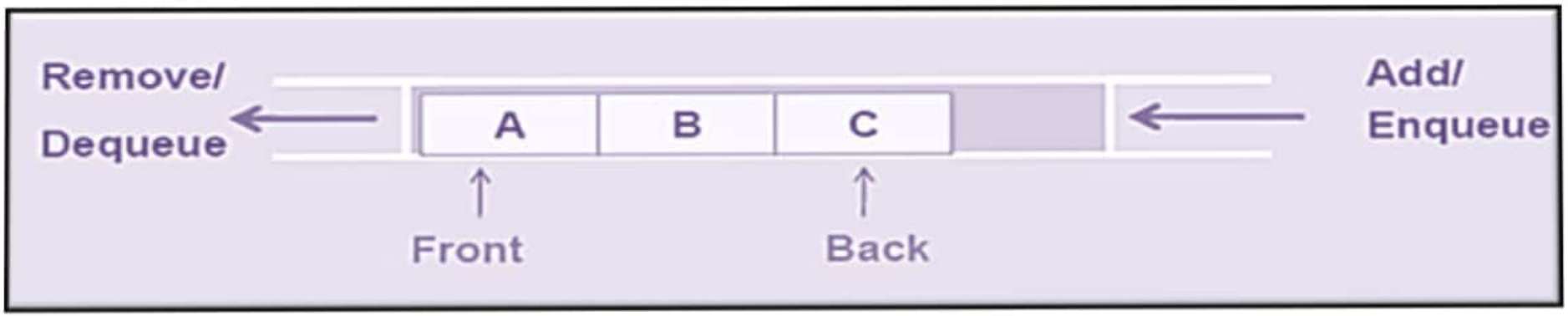
- Print lines of a document
- Printer sharing between computers
- Recognizing palindromes
- Shared resource usage (CPU, memory access,…)

Queue operations

TEKNOTURF

enqueue ()- The process of adding an element into a queue

dequeue() -the process of removing an element from a queue



Queue can be implemented using an *Array* or a *Linked List*

Queue Implementation using array

TEKNOTURF

Point to Remember:

- Number of elements in Queue are fixed during declaration
- Need **to check** whether a queue is full or not
- Initially the head(FRONT) and the tail(REAR) of the queue points to the first index of the array

Creation of a Queue

To create a Queue of MAX size, Initialize the front and the rear to 0 as:

```
class Queue {
        final static int MaxQ = 100;
        int front = 0, rear = 0;
        int[] QA = new int[MaxQ];

        public boolean empty() {
                return front == rear;
}}
```

Queue Implementation using array

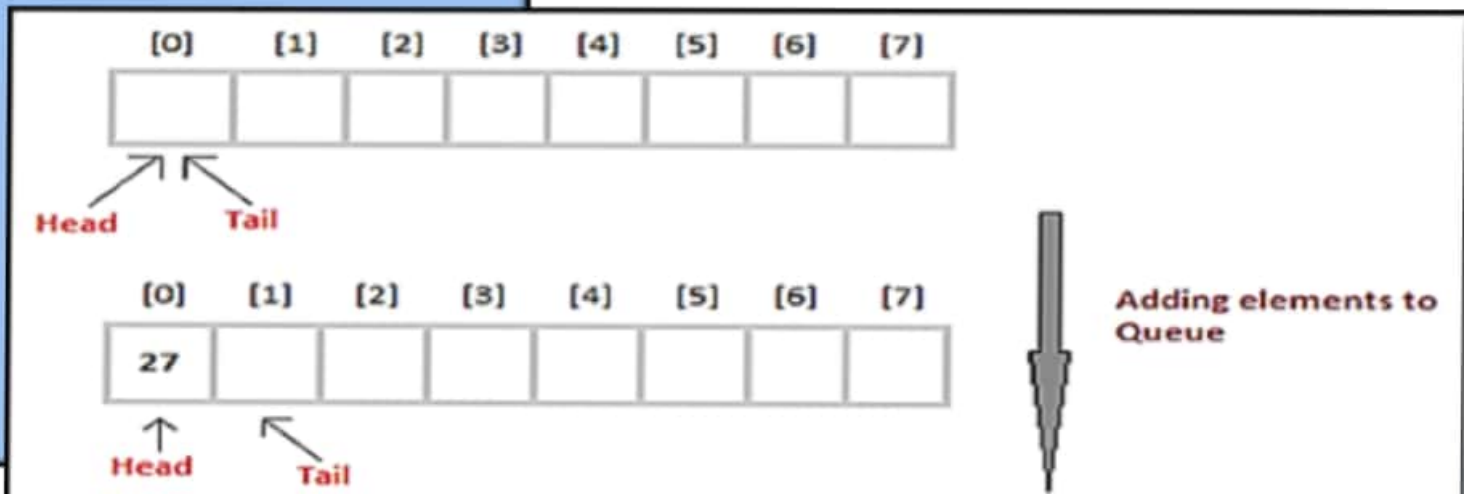
TEKNOTURF

Insert an element into a queue : enQueue()

```
public int enQueue(int n) {
    if (rear == front) {
        System.out.printf("\nQueue is full\n");
        return 0;
    }
    QA[rear] = n;
    rear++;
    return 1;
}
```

queue is full, so cannot insert the element, so return 0

After insertion increment the tail/rear



Queue Implementation using array

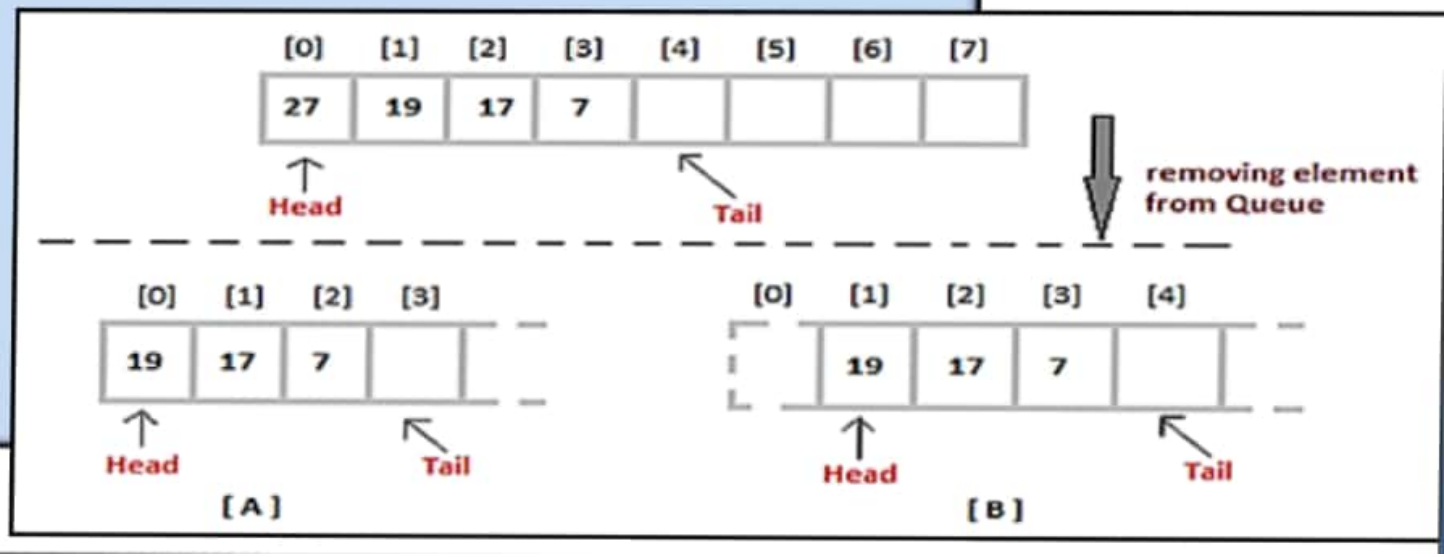
TEKNOTURF

Delete an element in a queue : deQueue()

```
public int deQueue() {
    if (this.empty()) {
            System.out.printf("\nAttempt to remove from an empty queue\n");
            return 0;
     }
     int delElement = QA[front];
    front++ ;
    return delElement;

}
```

queue is empty, so cannot dequeue the element, so return 0



Choose the Linear Data types:

- [ ] Tree
- [x] Queue
- [ ] Graph
- [x] Arrays
- [x] Stack

Correct

That's right! You selected the correct response.

Which of the following form of access is used to add and remove nodes from a queue?

- [x] FIFO, First In First Out
- [ ] FILO, First in Last Out
- [ ] LIFO, Last In First Out
- [ ] None of these options

Correct

That's right! You selected the correct response.

Match the following items with their descriptions:

1. push()	1. Insertion in stack
2. enque()	2. Insertion in queue
3. peek()	3. Access topmost element on stack
4. dequeue()	4. Deletion in queue
5. pop()	5. Deletion in stack

Correct

That's right! You selected the correct response.

Which Data Structure Should be used for implementing LRU cache?

- (●) stack
- ( ) Graph
- ( ) Queue
- ( ) Tree

Incorrect

You did not select the correct response.

Choose the correct real-time application which uses Queue for its implementation?

- Keeping track of local variables at run time
- ✔ Process Scheduling
- A parentheses balancing program
- Syntax analyzer for a compiler

Correct

That's right! You selected the correct response.

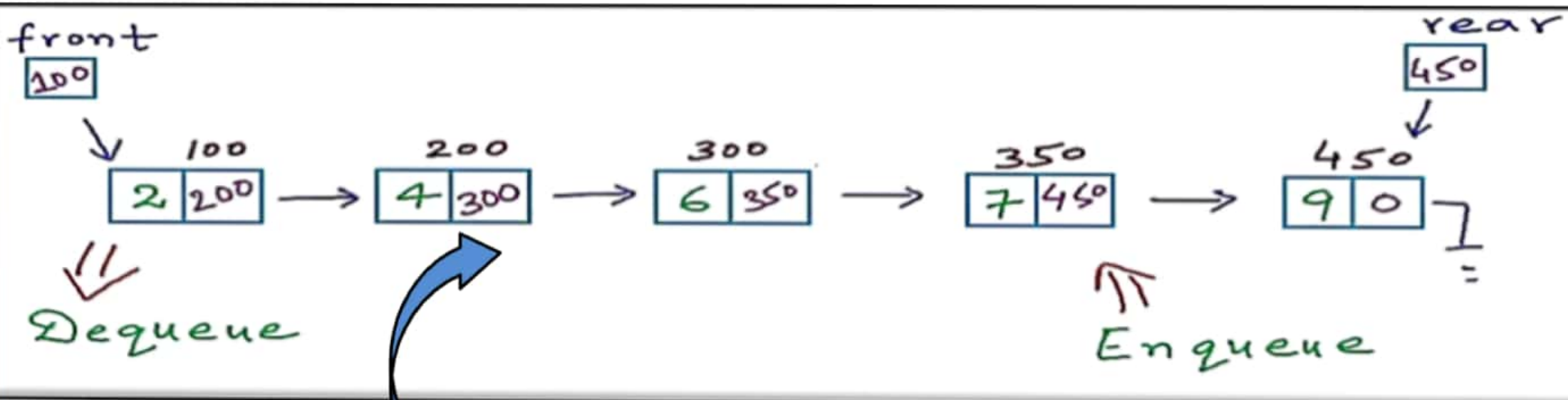
Which Data Structure Should be used for implementing LRU cache?

- Graph
- ✔ Queue
- stack
- Tree

Correct

That's right! You selected the correct response.

Queue – Linked List Implementation




```
class Node {
Object data;
Node next;
}
```

```
public Node(Object d)
{
data = d;
next = null;
}}
```

Node creation

Queue Implementation using Linked List

TEKNOTURF

EnQueue Operation:

1. Create Queue Node:

```
class Node {
    Object data;
    Node next;
   public Node(Object d) {
          data = d;
          next = null;
   } }
}
```

Two External pointer to be maintained: ***front*** and ***rear***

2. Allocate the new node and store the value

```
class Queue {
    Node front = null, rear = null;
   public void enQueue(Object nd)  {
       Node newNode = new Node(nd);
       if (front==null) {
            front = newnode;
            rear = newNode;
       }
       else  {
            rear.next = newNode;
            rear = newNode;
   }}
```

Equivalent to inserting an element at end of a linked list

There are two possibilities for inserting an element into a queue:

1. Insertion to an empty queue
2. Insertion to a non empty queue

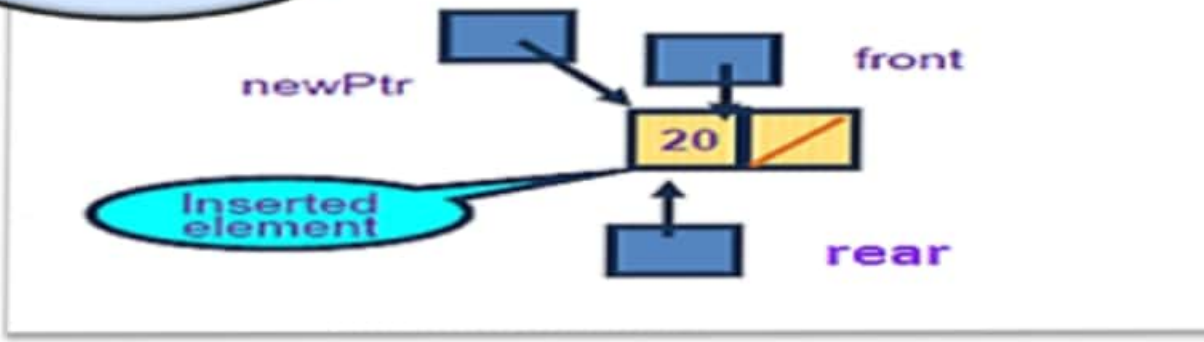
Queue Implementation using Linked List

TEKNOTURF

1. Insertion to an empty queue

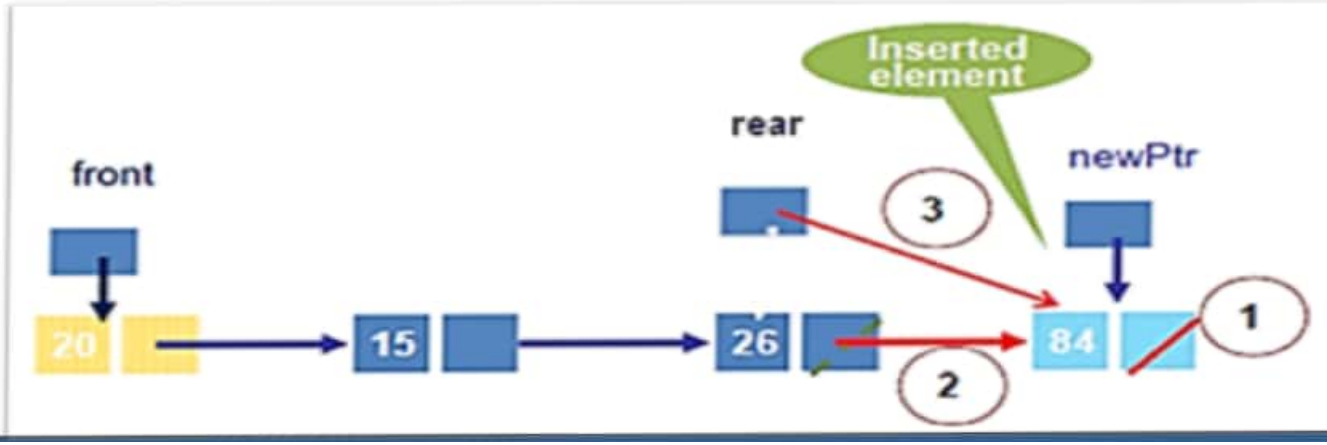
```
if(rear==NULL)
        front=newPtr;
        rear=newPtr
```

If the queue is an empty queue



2. Insertion to a non-empty queue

```
if(rear!=NULL)
        rear.next=newPtr;
        rear=newPtr;
```

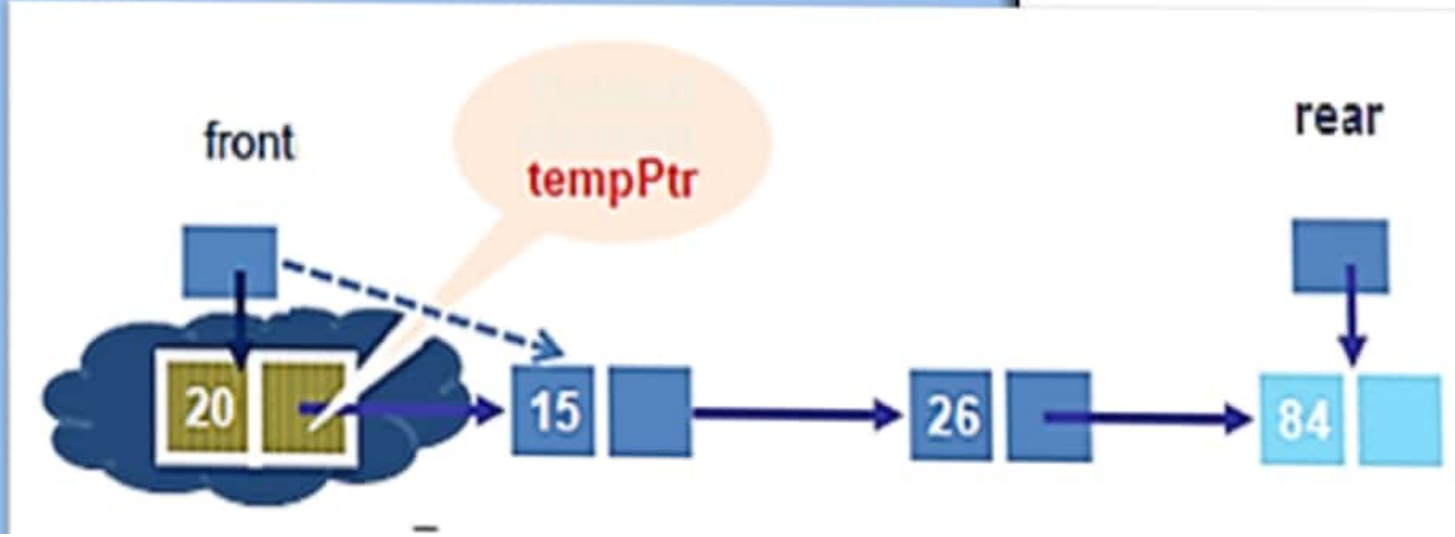


Queue Implementation using Linked List

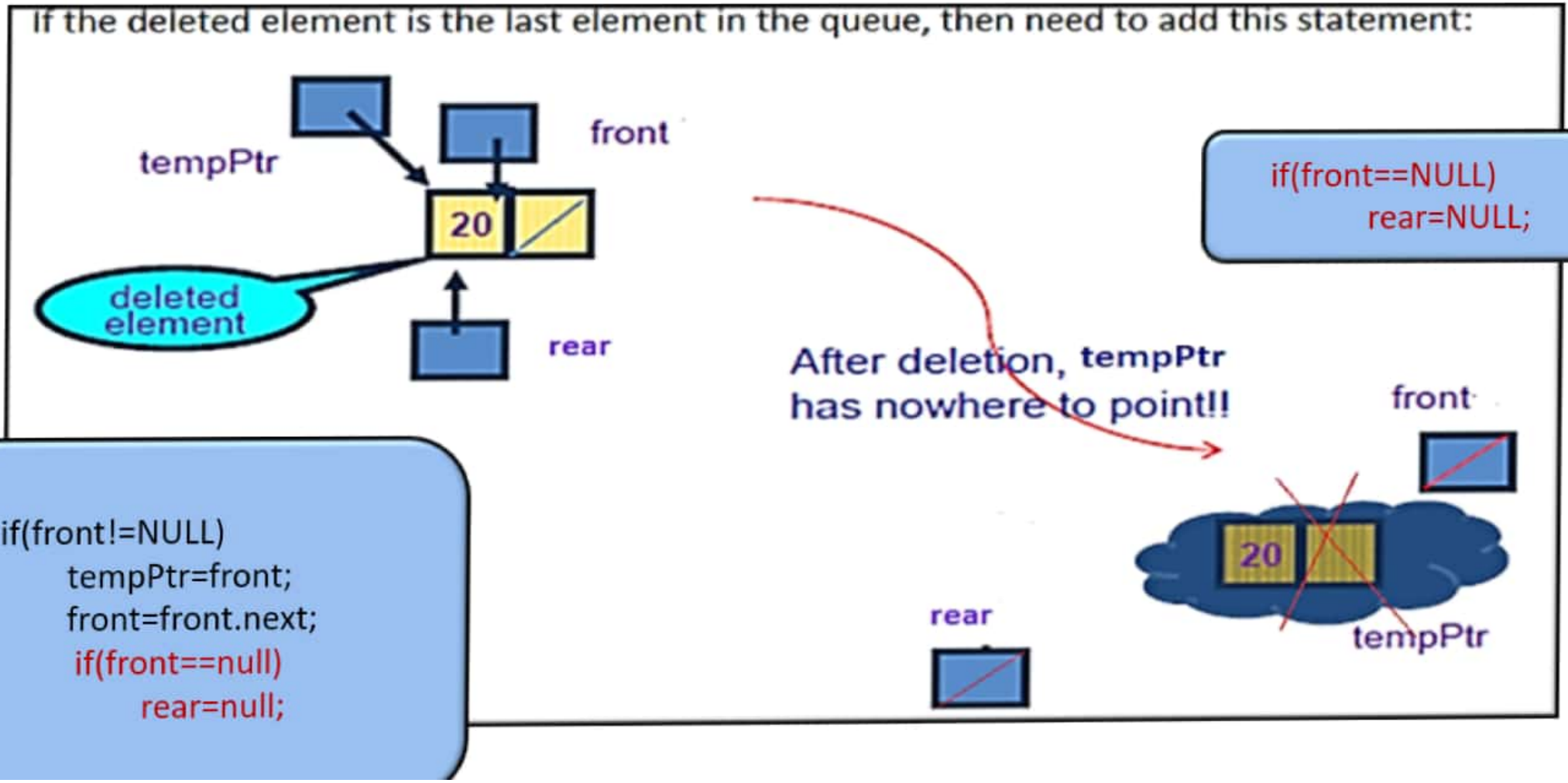
TEKNOTURF

deQueue Operation:

```
public Object deQueue() {
   if (this.empty()) {
      System.out.printf("\nAttempt to remove
                        from an empty queue\n");
       System.exit(1);
   }
  Object tempPtr= front.data;
  front = front.next;
  if (front == null)
          rear = null;
   return tempPtr;
}
```



Queue Implementation using Linked List



LinkedList Class in Java

- LinkedList class is part of the Java API package java.util
- It is a double-linked list that implements the List interface
- In the LinkedList class, the iterators are bi-directional: they can move either forward (to the next element) or backward (to the previous element).

- Because the LinkedList class implements the List interface and LinkedList objects support a variety of index-based methods
- The index always start at 0

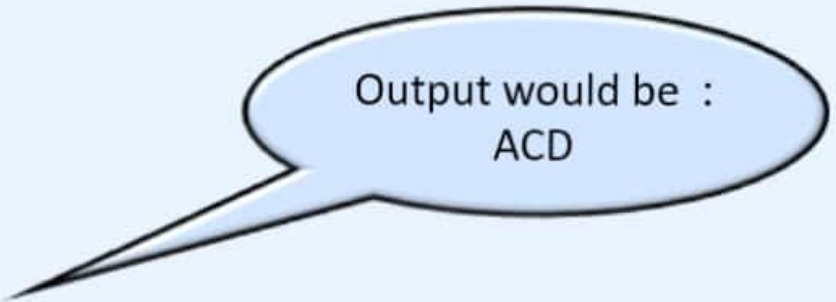
Method	Behavior
`public void add(int index, E obj)`	Inserts object `obj` into the list at position `index`
`public void addFirst(E obj)`	Inserts object `obj` as the first element of the list
`public void addLast(E obj)`	Adds object `obj` to the end of the list
`public E get(int index)`	Returns the item at position `index`
`public E getFirst()`	Gets the first element in the list. Throws `NoSuchElementException` if the list is empty
`public E getLast()`	Gets the last element in the list. Throws `NoSuchElementException` if the list is empty
`public boolean remove(E obj)`	Removes the first occurrence of object `obj` from the list. Returns `true` if the list contained object `obj`; otherwise, returns `false`
`public int size()`	Returns the number of objects contained in the list

LinkedList Class in Java

TEKNOTURF

Sample Code which implements LinkedList Class In Java:

```
import java.util.*;
public class Test {
    public static void main(String args[]) {
        LinkedList<String> object = new LinkedList<String>();
        object.add("A");
        object.add("B");
        object.addLast("C");
        object.add("D");
        object.add("E");
        object.addFirst("F")
        object.remove("B");
        object.removeFirst();
        object.removeLast();
        System.out.println("Linked list after deletion: " + object);
    }
}
```



Summary

TEKNOTURF

In this Module we have learnt about:

- Data Structure concepts
- Abstract Data Types
- Linear Data Structures
  - Arrays
  - Linked List Implementation
  - Stack Implementation
  - Queue Implementation

TEKNOTURF

Thank You

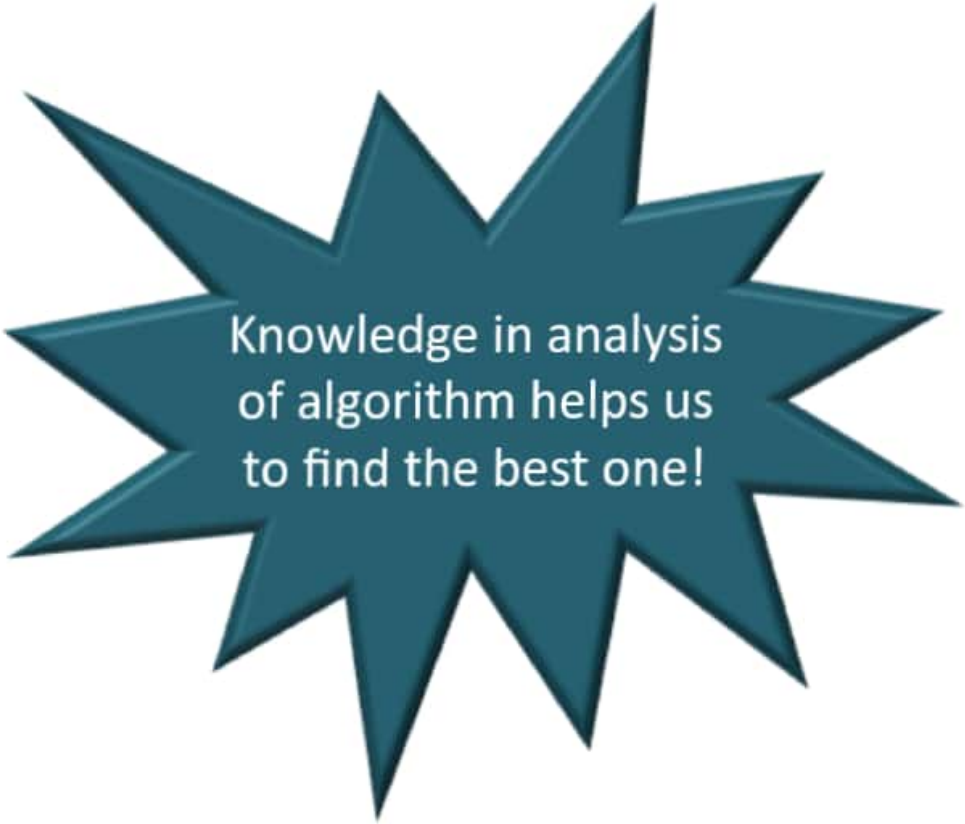
TEKNOTURF

ANALYSIS OF ALGORITHMS

Analysis of Algorithm

TEKNOTURF

There are now two solutions for finding a prime number. So which one will be the best algorithm?



In this module you will learn

TEKNOTURF

- Algorithm Efficiency Analysis
- Time Complexity
- Searching and Sorting Algorithms
  - Types of Searching Algorithm
  - Types of Sorting Algorithm

Algorithm efficiency

TEKNOTURF

Efficiency of an algorithm is based on:

CPU time usage

Memory usage

Disk usage

Network usage

Algorithm efficiency

TEKNOTURF

To find a better solution to a problem, we may use algorithms.
To choose the correct algorithm, we need to analyze the efficiency of the algorithm.

How do we analyze the algorithm?

Analysis of the algorithms are based on 3 things:

1. **Quality** and **accuracy** of the algorithm
2. How much time do the algorithms take to perform the operations (**time complexity**)
3. How much memory it requires (**Space Complexity**)

Time complexity

TEKNOTURF

Time complexity of an algorithm signifies the total time required by the algorithm to complete its execution with provided resources

It is measured in terms of number of operations rather than computer time; because computer time depends on the hardware, processor, etc.

Time for most algorithms depends on the amount of data or size of the problem. If the volume of data is doubled, the time needed for the computations is also doubled.

Time needed is proportional to the amount of data

Algorithm's performance may vary with different types of input data.
Hence, we usually use the worst-case Time complexity of an algorithm because that is the maximum time taken for any input size

Cases in Algorithm Analysis

TEKNOTURF

Three cases to analyze an algorithm:

Worst Case Analysis:

- The worst case analysis calculates upper bound on running time of an algorithm
- It is the maximum length of running time for any input of size n
- We usually concentrate on finding only the worst-case running time

Average Case Analysis:

- Average case complexity is the complexity of an algorithm that is averaged over all possible inputs
- It is the average running time for any input of size n

Best Case Analysis:

- Best case analysis calculates lower bound on running time of an algorithm
- We must know the case that causes minimum number of operations to be executed

Big Oh Notation

TEKNOTURF

- The most common metric for calculating time complexity is Big O notation
- It expresses the amount of time required by the algorithm
- It can be denoted by the symbol 'O'
- Big Oh denotes "fewer than or the same as" <expression> iterations

Time complexity for different operations can be denoted as:

O(1) : An algorithm which executes on constant time period
O(n) : An algorithm which executes in linear time period
$O(n^2)$: An algorithm which executes in quadratic time period
O(log n) : An algorithm which executes in a logarithmic time period

n?

Big Oh removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity

Determining the Time complexity

TEKNOTURF

```
statement;
```

The running time of the statement will not change in relation to N.
Then its Time Complexity will be **Constant. (**O(1))

```
for(i=0; i < N; i++)
{
      statement;
}
```

The running time of the loop is directly proportional to N. When N doubles, so does the running time. So the time complexity for this algorithm will be **Linear** . (O(n))

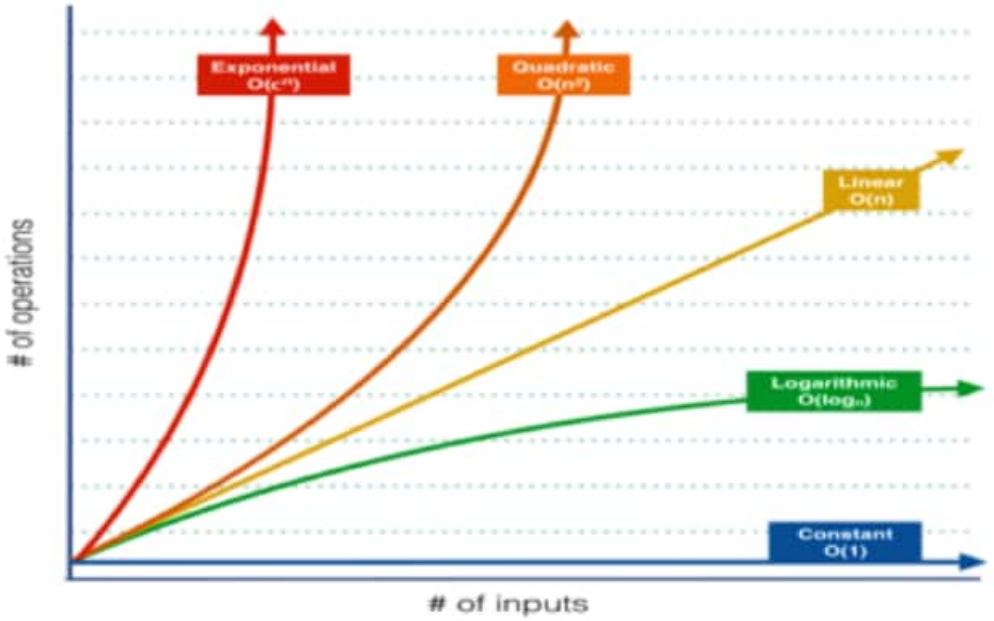
```
for(i=0; i < N; i++) {
for(j=0; j < N; j++) {
     statement;
} }
```

The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N. The time complexity for this code will be **Quadratic** . ($O(n^2)$)

```
while(low <= high) {
    mid = (low + high) / 2;
   if (target < list[mid])
        high = mid - 1;
   else if (target > list[mid])
        low = mid + 1; else break;
}
```

This is an algorithm to break a set of numbers into halves. The running time of the algorithm is proportional to the number of times N can be divided by 2. This algorithm will have a **Logarithmic** Time Complexity. (O(log n))

Measure



General order that we may consider:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n!) < O(n^2)$$

Recursion Overview

TEKNOTURF

Just like these Russian dolls, we'll see how to design an algorithm to solve a problem by solving a smaller instance of the same problem. We call this technique **recursion**.

Recursion

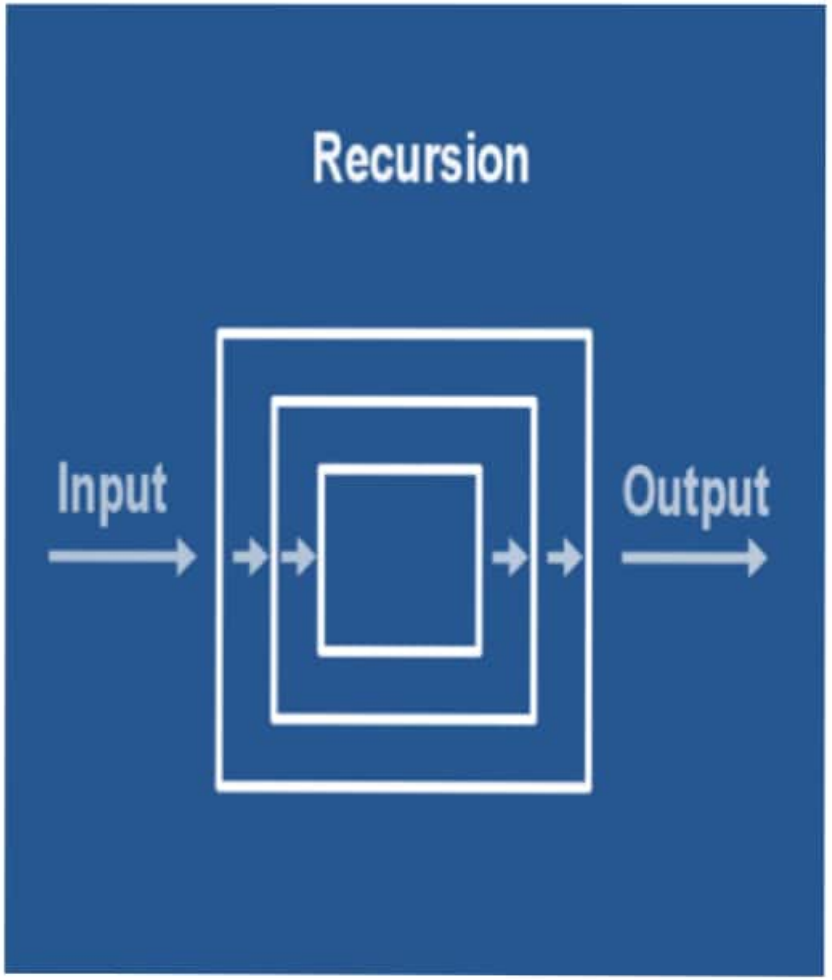
It is an algorithm which calls itself with “smaller ” input values, and obtains the result for the current input by applying simple operations to the returned value for the smaller inputs.

Solves difficult problems easily and

provides readability

but

needs lot of memory



Recursion – Base case

Function call ends when condition meets base value

if (answer is known) provide the answer	base case
else make a recursive call to solve a **smaller** version of the same problem	recursive case

Factorial using recursion

TEKNOTURF

How to find Factorial of 6 ?

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

```
public int  factorial(int n)
    {
            if (n <= 1)
                  return 1;
             else
                  n*factorial(n-1);

    }
```

Exit / Base condition

Recursive call

Iteration Vs. Recursion

TEKNOTURF

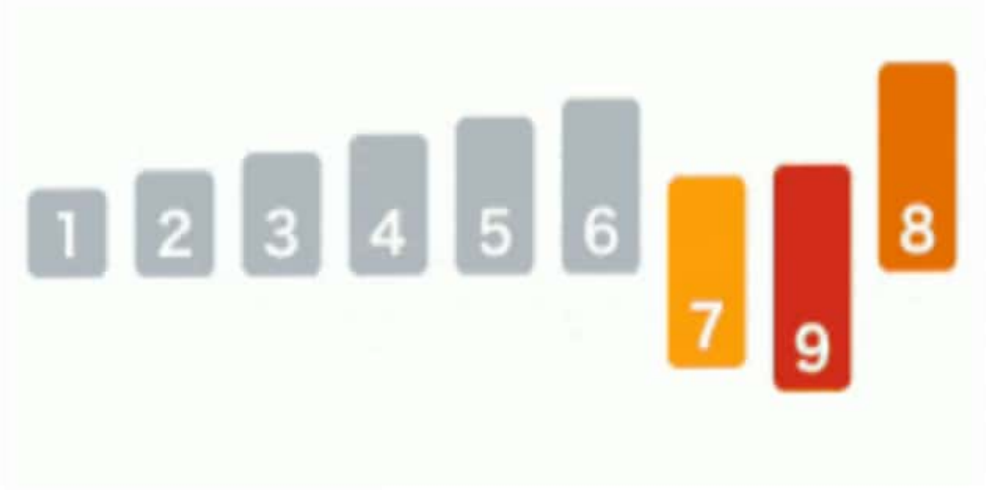
ITERATION

- It is a process of executing statements repeatedly, until some specific condition is specified
- Iteration involves four clear cut steps, initialization, condition, execution and updating
- Any recursive problem can be solved iteratively
- Iterative computer part of a problem is more efficient in terms of memory utilization and execution speed

RECURSIVE

- Recursion is a technique of defining anything in terms of itself
- There must be an exclusive if statement inside the recursive function specifying stopping condition
- Not all problems has recursive solution
- Recursion is generally a worst option to go for simple program or problems not recursive in nature

Searching and Sorting



In computer science, a searching algorithm is an algorithm that retrieves information stored within some data structure.

A sorting algorithm is an algorithm that puts elements of a list in a certain order

Searching

TEKNOTURF

The appropriate search algorithm often depends on the data structure being searched

Search algorithms can be classified based on their mechanism of searching

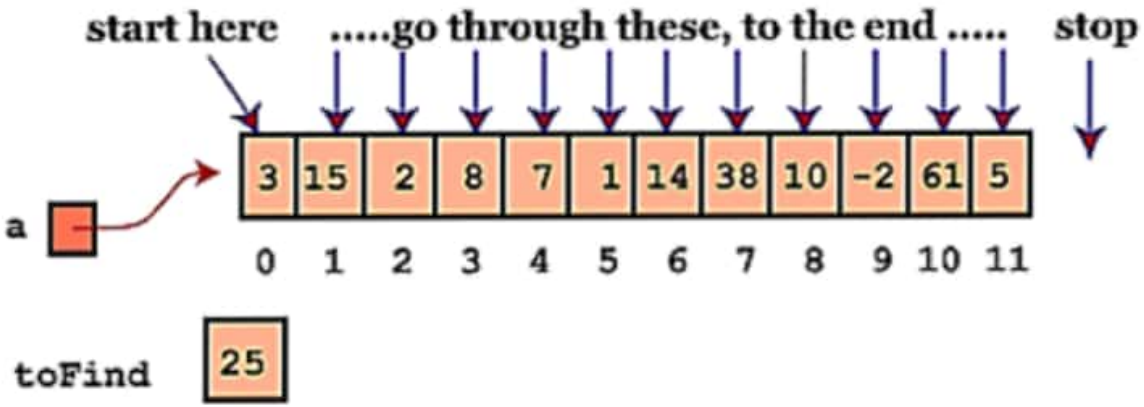
What are the most popular searching methods?

- Sequential / Linear Search
- Binary Search

Linear Search

TEKNOTURF

Linear search is a very simple search algorithm.
It searches for data by comparing each item in a list one after the other.



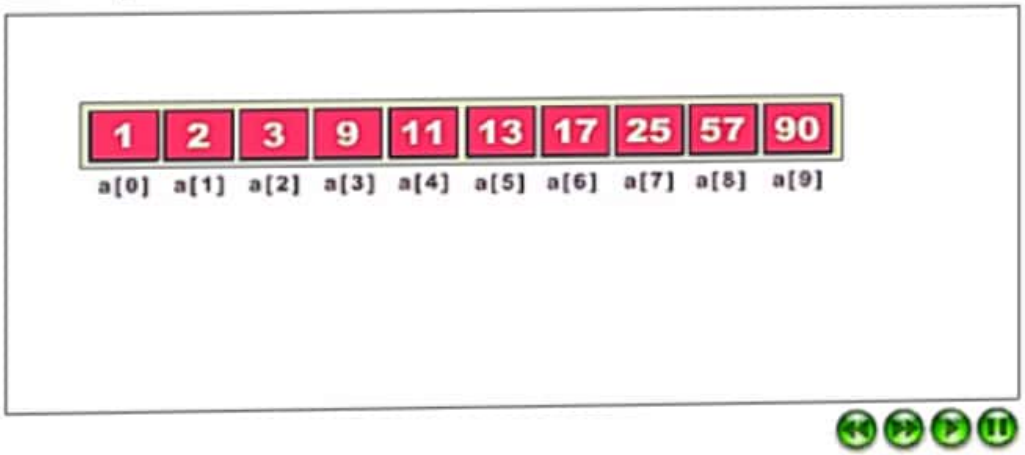

```
int Search(int a[], int size, int toFind)
{
    int index , retVal = -1;
    for( index=0; index < size; index++)
    {
        if ( toFind == a[index] )
        {
            retVal = index;
            break;
        }
    }
    return retVal;
}
```

returns -1
for this example, because 25 is not found in the given array

Binary Search

- This search algorithm works on the principle of divide and conquer
- It is an efficient algorithm for finding an item from an ordered list (sorted) of items

Binary search



Binary Searching Procedure

TEKNOTURF

Binary search compares the target value to the middle element of the array

If the value is matched, then it returns the value

If the value is less than the middle element, then it must lie in the lower half of the array and if it's greater than the element then it must lie in the upper half of the array

Repeat this procedure on the lower (or upper) half of the array.

Binary Search is useful when there are large numbers of elements in an array

Binary Search

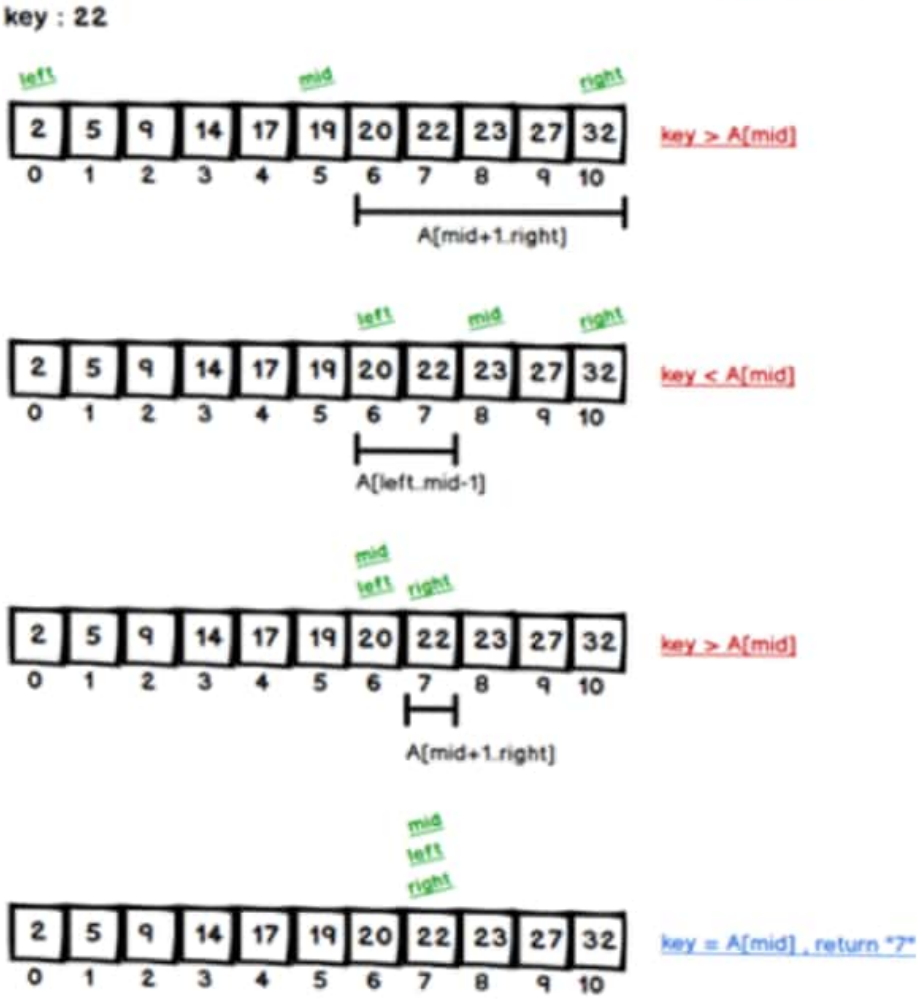
```
int binarySearch(int A[], int size, int key)
{
   int left = 0, right = size-1;
   while (left <= right) {
      mid = ( left + right)/2;
      if( key == A[mid]
            return mid;
      else if ( key < A[mid])
            right = mid – 1;
       else
            left = mid + 1;
   }
  return -1;
}
```

Returns index when key is found

When search value is less than the mid value

When search value is greater than the mid value

Returns -1 when key is not found



Which of the following is the correct list where we can appply Binary search algorithm.

- Unsorted linked list
- ✔ Sorted Array
- None of these options
- Sorted linked list

Correct

That's right! You selected the correct response.

The time factor when determining the efficiency of algorithm is measured by_____.

- counting the number of statements
- counting the kilobytes of algorithm
- counting microseconds
- ✔ counting the number of key operations

Correct

That's right! You selected the correct response.

The Team leader wanted Raj to develop an efficient algorithm for an appliation. To check the efficiency of the algorithm which Raj developed, what are the two main factors he needs to consider?

- ○ Processor and memory
- ○ Complexity and capacity
- ○ Data and space
- ✔ ◉ Time and space

Correct

That's right! You selected the correct response.

Match the following:

1. Space Complexity	1. How much memory need to perform the operation
2. Time Complexity	2. How long does it take to find a solution
3. Completeness	3. Is the strategy guaranteed to find the solution when there in one
4. Accuracy	4. producing the correct solution

Correct

That's right! You selected the correct response.

What is the Worst case scenario occur in linear search algorithm?

- [x] Item is the last element in the array or is not there at all
- [ ] Item is not in the array at all
- [ ] Item is the last element in the array
- [ ] Item is somewhere in the middle of the array

Correct

That's right! You selected the correct response.

Sorting

TEKNOTURF

Sorting is a process that organizes a collection of data into either ascending or descending order

Types of Sorting

TEKNOTURF

Insertion Sort

Bubble Sort

Quick Sort

Merge Sort

Heap Sort

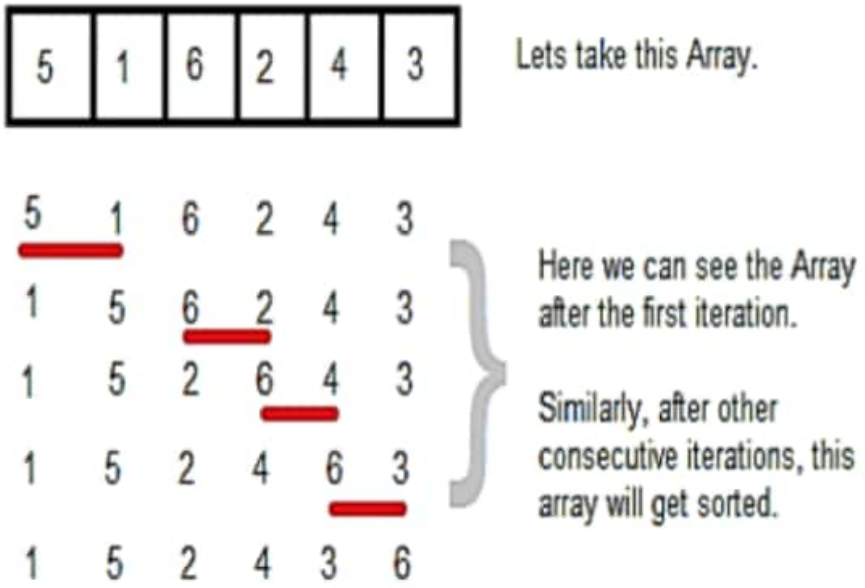
Bubble sort

TEKNOTURF

Bubble Sort takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

Why is it called Bubble sort?

With each iteration, the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface

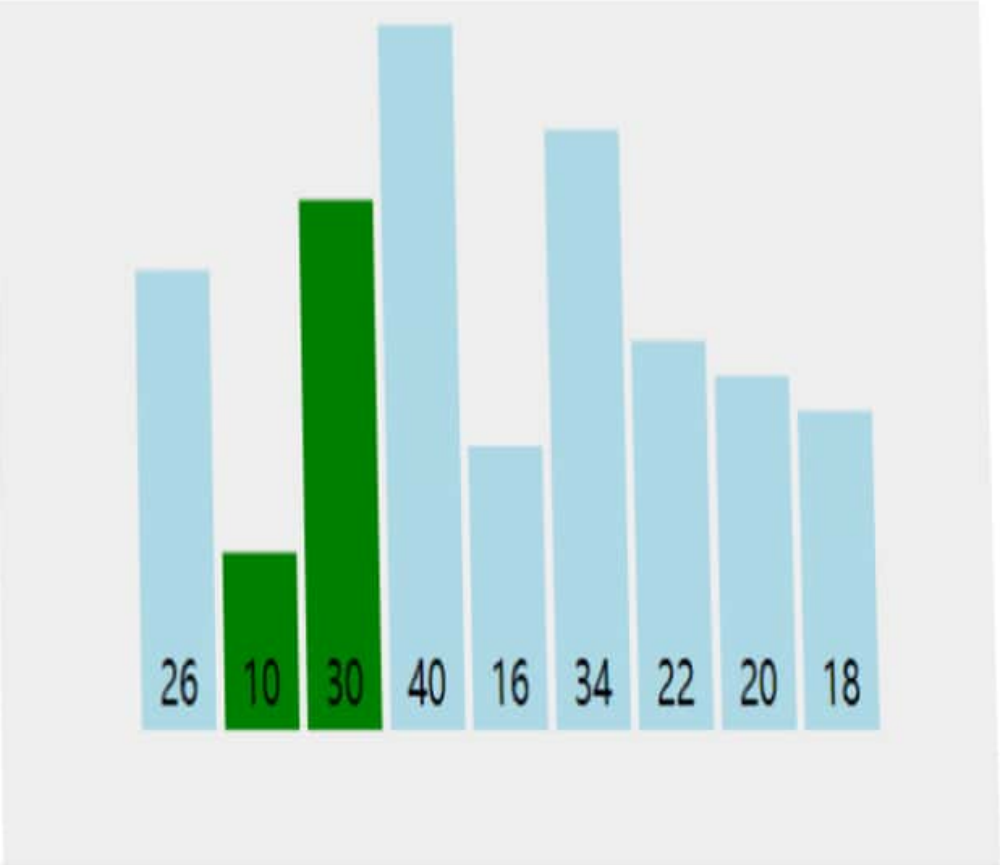


Bubble sort

TEKNOTURF

Bubble Sort Algorithm

```
bubbleSort(A,n)
    for i = 0 to n-1
        for j = 0 to n-i-1
            if A[j] >A[ j+1 ]
                swap a[j] <-> A[j+1]
```



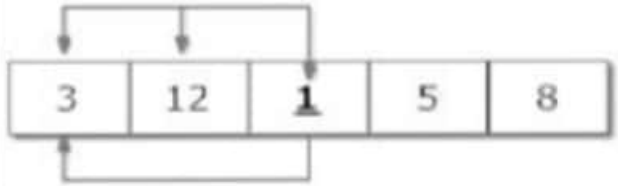
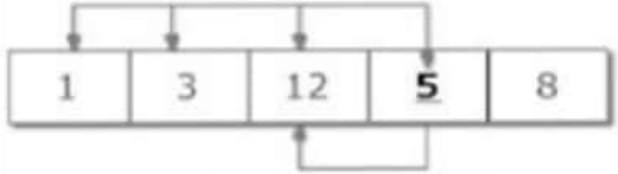
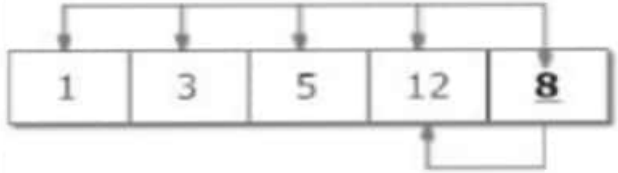
Insertion Sort

TEKNOTURF

It is a simple Sorting algorithm which sorts the array by shifting elements one by one

It is efficient for smaller data sets, but very inefficient for larger lists

How Insertion Sort works?

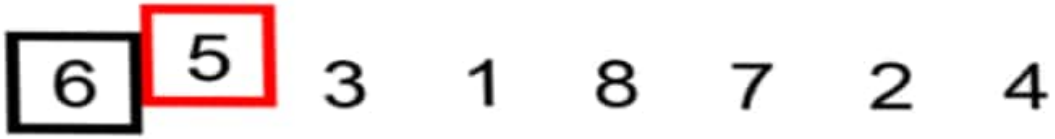
Step 1		**Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.**
Step 2		**Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3.**
Step 3		**Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.**
Step 4		**Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.**
		**Sorted Array in Ascending Order**

Insertion Sort

TEKNOTURF

Insertion Sort Algorithm

```
insertionSort(A: an array of items)
        for i = 1 to length(A)-1
            item = A[i]
            j = i
            while j> 0 and  A[j – 1] > item
                A[j] = A[j – 1]
                j = j – 1
            A[j] = item
```



Quick Sort

TEKNOTURF

Quick Sort sorts any list very quickly

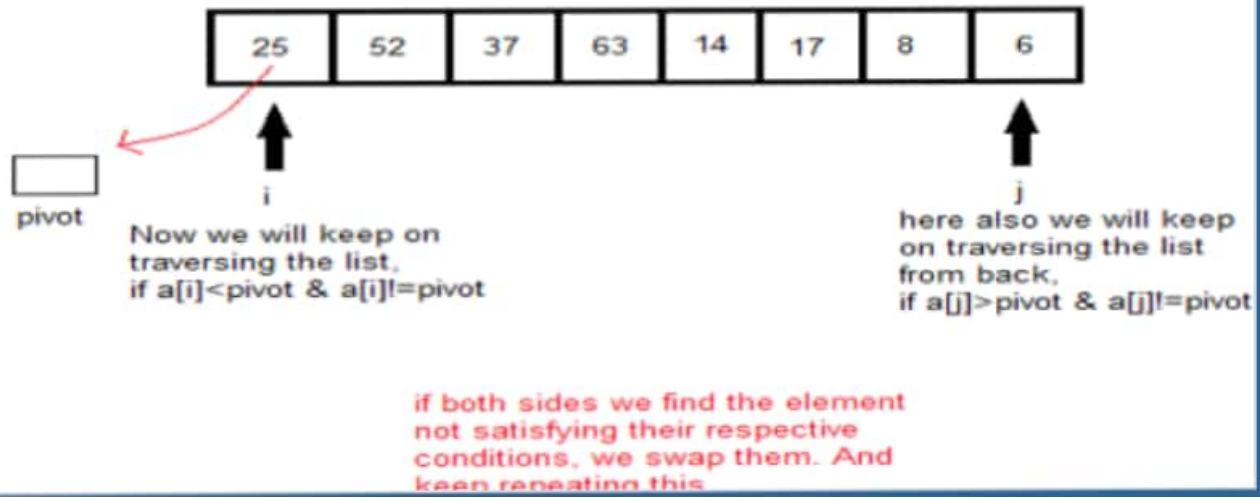
It is based on the rule of **Divide and Conquer**

This algorithm divides the list into three main parts :

- Elements less than the Pivot element
- Pivot element
- Elements greater than the pivot element

Strategy

- Choose a pivot (first element in the array)
- Partition the array about the pivot
  - items < pivot
  - items >= pivot
  - Pivot is now in correct sorted position
- Sort the left section again until there is one item left
- Sort the right section again until there is one item left



Quick Sort

TEKNOTURF

Quick Sort Algorithm

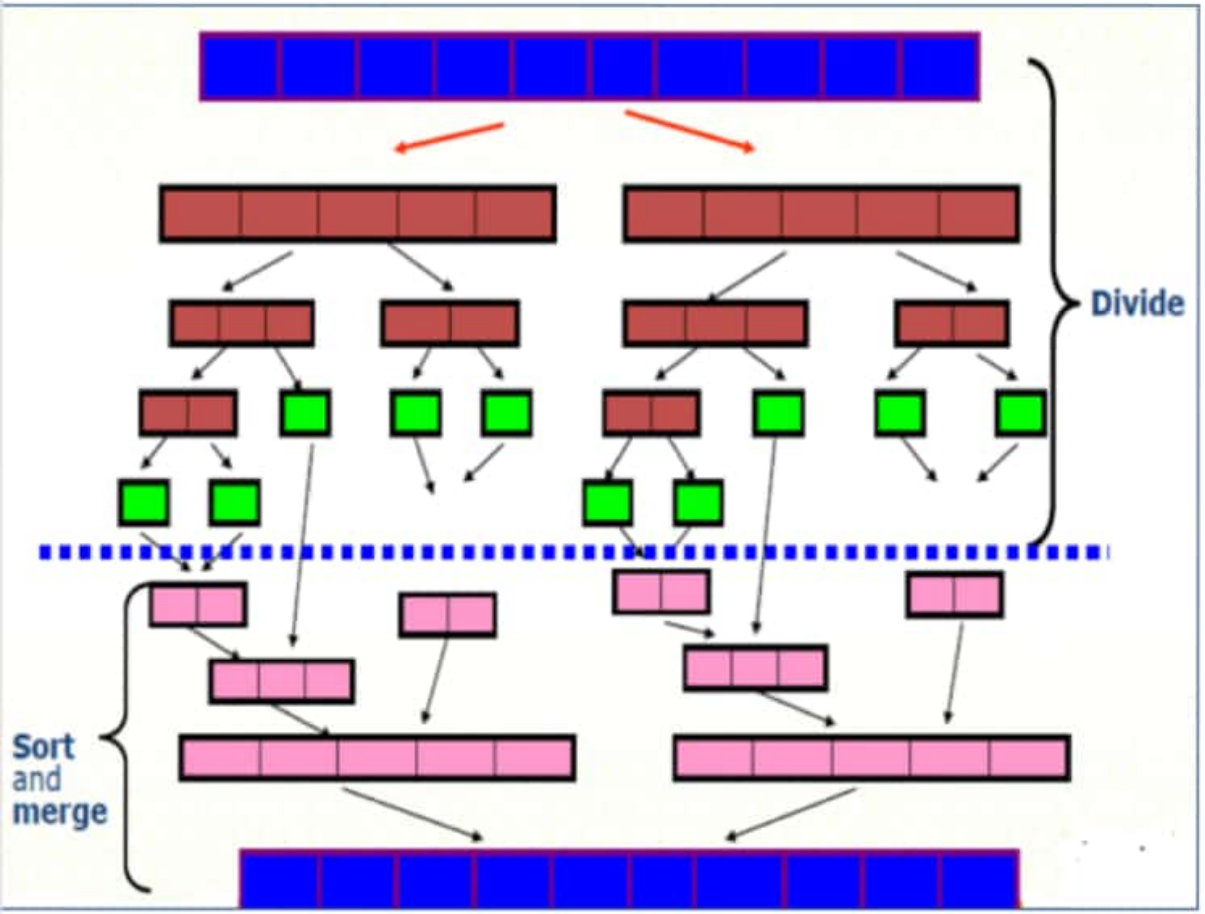
```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Merge Sort

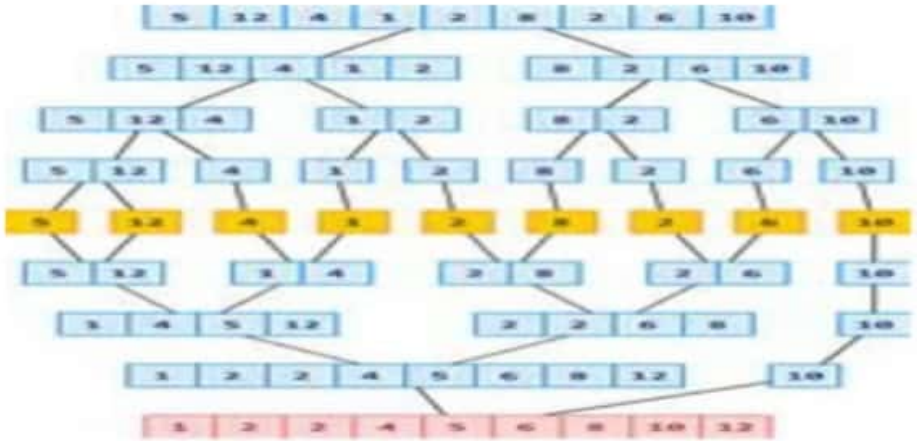
TEKNOTURF

Merge sort applies divide and conquer strategy



Three main steps in Merge Sort algorithm:

- **Divide** an array into halves until only one piece left
- **Sort** each half
- **Merge** the sorted halves into one sorted array



Merge Sort

TEKNOTURF

Merge Sort Algorithm

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a

   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]

   l1 = mergesort( l1 )
   l2 = mergesort( l2 )

   return merge( l1, l2 )
end procedure
```


```
procedure merge( var a as array, var b as array )

   var c as array

   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
   end while

   while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a
   end while
```

Heap Sort

Heap Sort algorithm is the best sorting method in-place with no quadratic worst-case scenarios

This algorithm is divided into two basic parts:

- Creating a Heap of the unsorted list
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array.

The heap is reconstructed after each removal

Heap is a special ***complete binary tree*** -based data structure that satisfies the special heap properties.

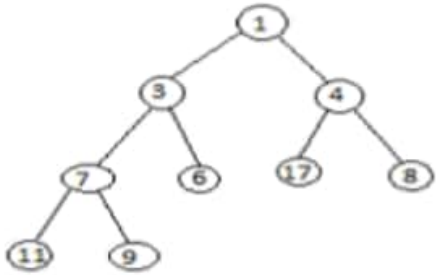
Heap Property : All nodes are either *greater than or equal to* or *less than or equal to* each of its children

Heap Sort

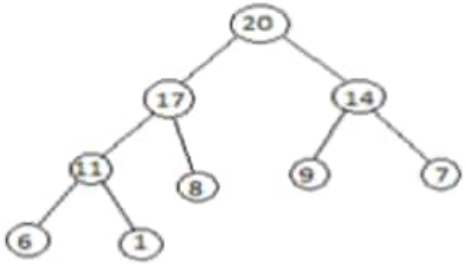
TEKNOTURF

Max-Heap - If the parent nodes are greater than their child nodes

Min-Heap - if the parent nodes are smaller than their child nodes



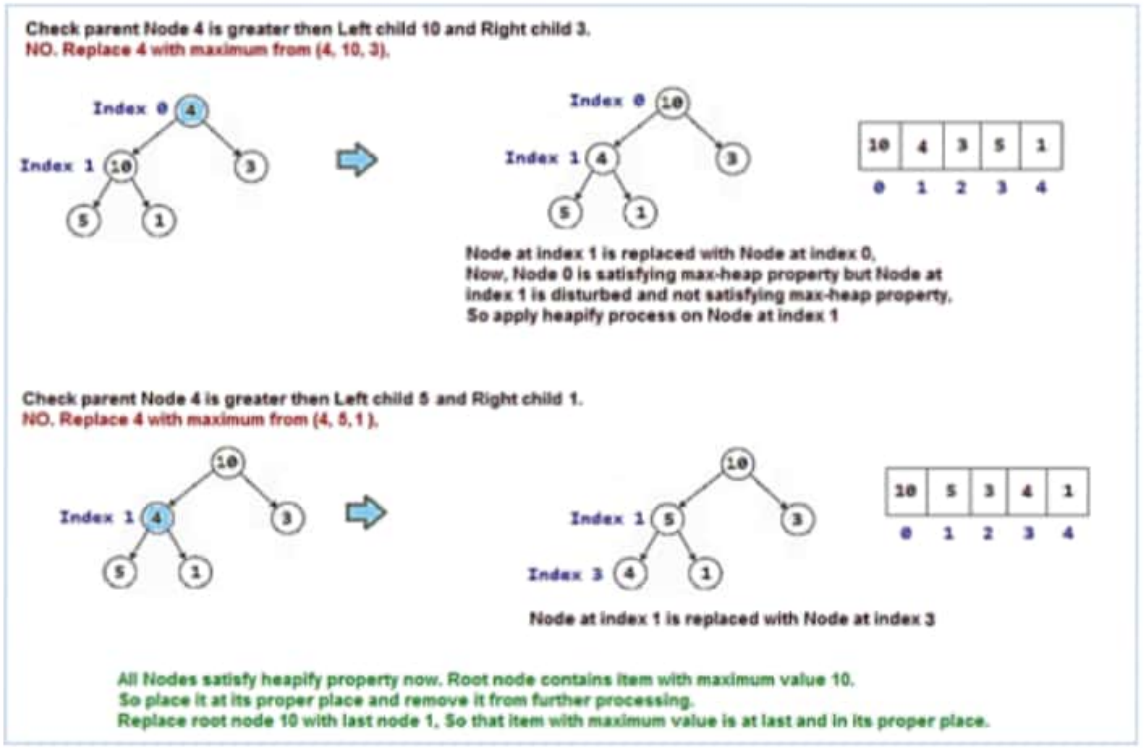
Min-Heap



Max-Heap

What is heapifying process?

Heapify picks the largest or smallest child node and compares it to its parent node. If the parent node is larger or smaller than its child node, heapify quits, otherwise it swaps the parent node with the largest child node.



Heap Sort

How heap sort works?

TEKNOTURF

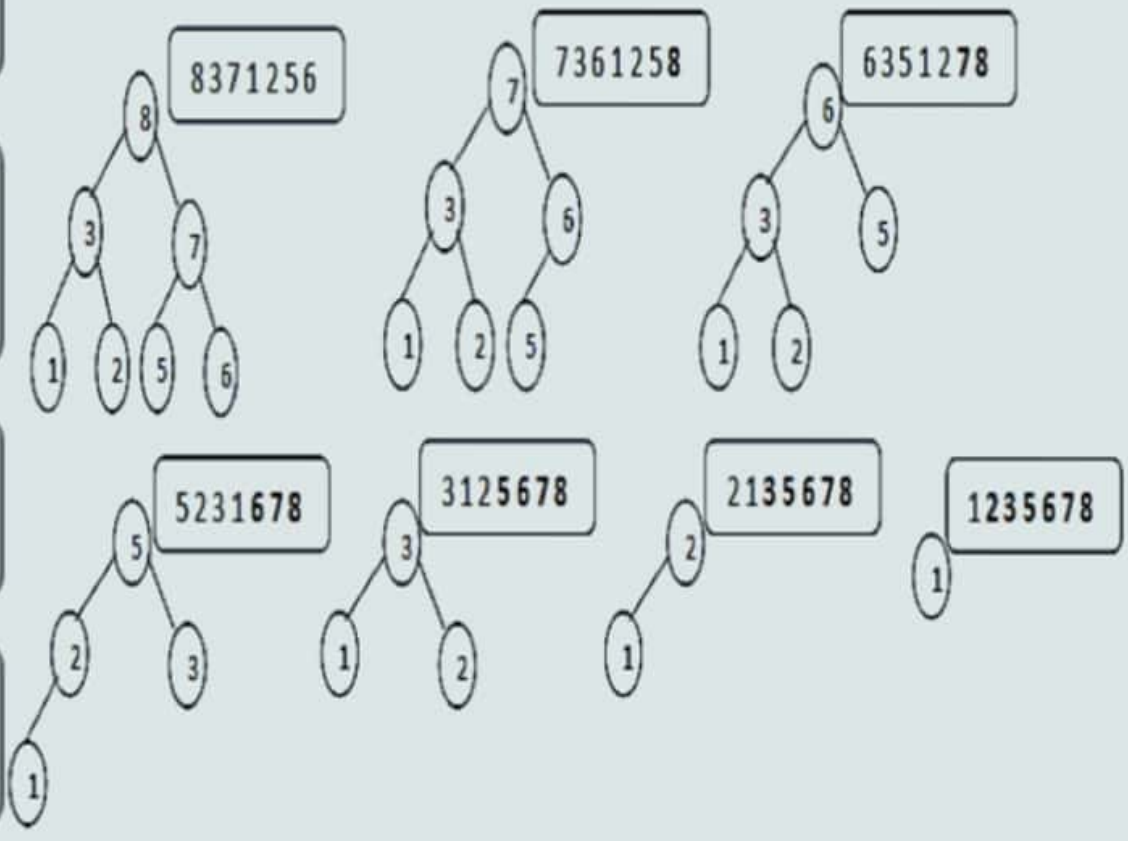
The first step in heap sort is to create a Heap data structure (Max-Heap or Min-Heap) on the unsorted list

Once the heap is built, the first element of the Heap is either largest (Max-Heap) or smallest (Min-Heap). Next, it places the first element of the heap into the sorted array.

Then it again makes the heap using the remaining elements and picks the first element of the heap and puts it into the array.

It repeats this process until the array becomes completely sorted.

Example:- The fig. shows steps of heap-sort for list (2 3 7 1 8 5 6)



Choose the correct sorting algortihm wich is Efficient for smaller data sets but not for larger sets.

- Quick Sort
- Bubble Sort
- ✔ Insertion Sort
- Merge Sort
- Heap Sort

Correct

That's right! You selected the correct response.

The running time of quick sort depends heavily on the selection of

- Arrangement of elements in array
- Size of elements
- No of inputs
- ✔ Pivot element

Correct

That's right! You selected the correct response.

The number of interchanges required to sort 6, 2, 7, 3 5 in ascending order using Bubble Sort is _____

- 7
- 6
- 5
- 3

Incorrect

You did not select the correct response.

Match the following items with their descriptions:

1. Heap Sort	1. Complete Binary Tree based
2. Quick Sort	2. Pivot value
3. Bubble Sort	3. Compare Adjacent data
4. Insertion Sort	4. Best for small data sets

Correct

That's right! You selected the correct response.

How many number of comparisons are required in insertion sort to sort a file if the file is sorted in reverse order?

- N
- ✔ N^2
- N/2
- N-1

Correct

That's right! You selected the correct response.

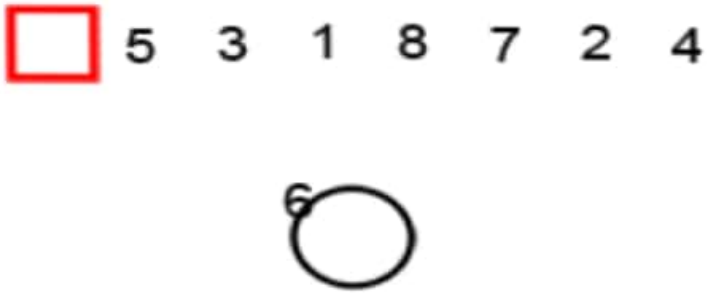
Heap Sort

Heap Sort Algorithm

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4      largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7      largest = r
 8  if largest ≠ i
 9      exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



Heap Sort

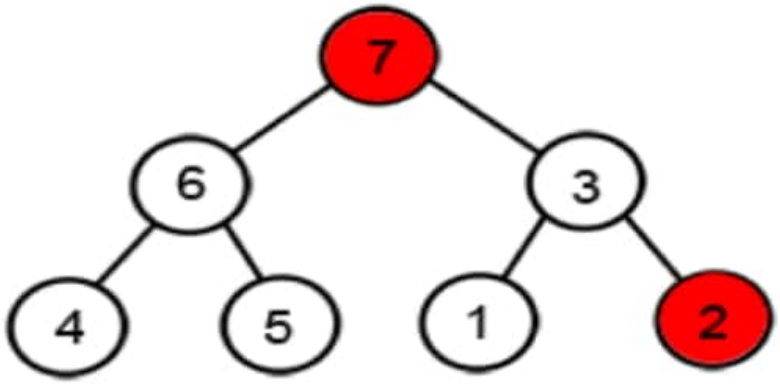
TEKNOTURF

Heap Sort Algorithm

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4      largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7      largest = r
 8  if largest ≠ i
 9      exchange A[i] with A[largest]
10      MAX-HEAPIFY(A, largest)
```

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```



Sorting Comparison

TEKNOTURF

Sorting Algorithm Complexity

	Worst Case	Average Case	Best Case
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Insertion Sort	$O(n^2)$	$O(n^2)$	$O(n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

Summary

TEKNOTURF

In this Module we have learnt about:

- Algorithm Efficiency Analysis
- Time Complexity
- Searching and Sorting Algorithms
- Types of Searching Algorithm
- Types of Sorting Algorithm

TEKNOTURF

Thank You