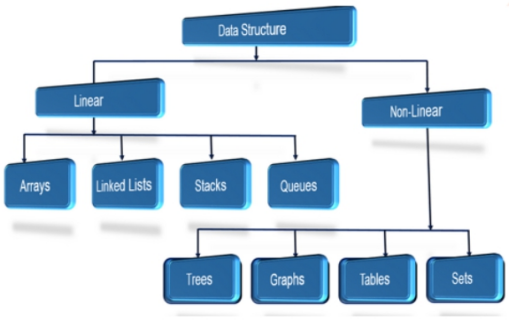
Overview



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!

Problem Solving

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

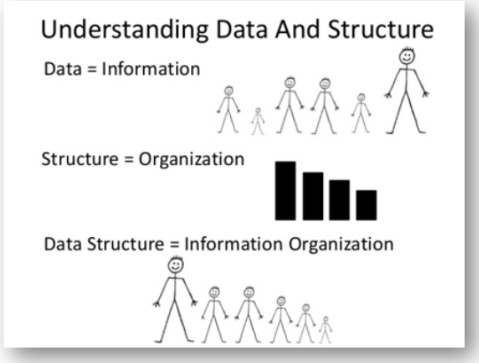
Data Structure

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

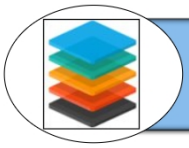
A step by step description for performing a task within a finite period of time

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.

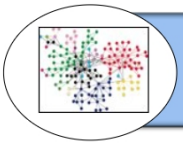
Abstract Data Type (ADT)

TEKNOTURF



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

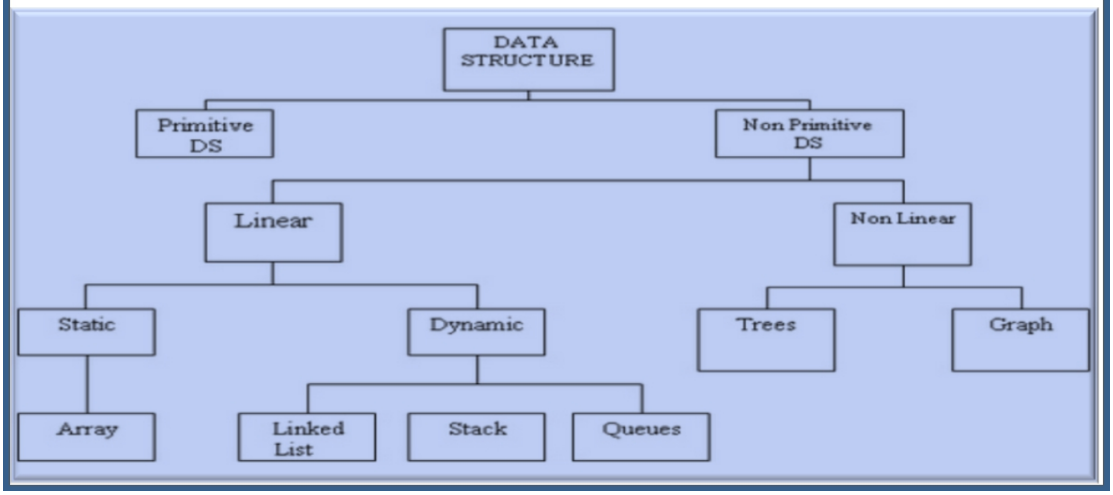
ADT is a logical description of how we view the data and the operations that are allowed regardless of how they will be implemented



Implementation of an ADT: Includes choosing a particular data structure

Data Structure Categories

TEKNOTURF



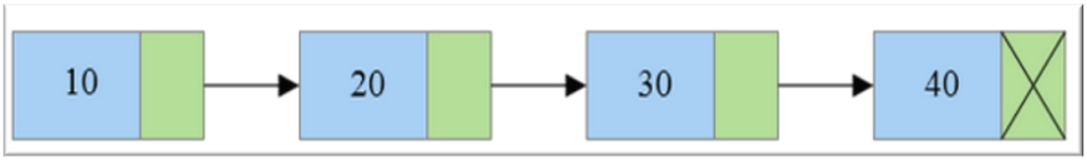
Linear Data Structure

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

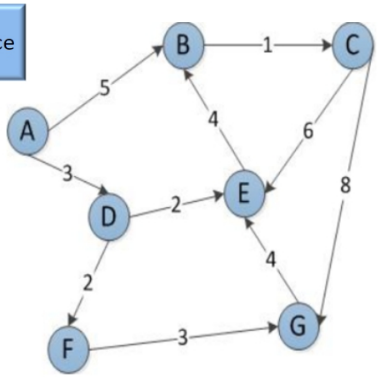
Non-Linear Data Structure

TEKNOTURF

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

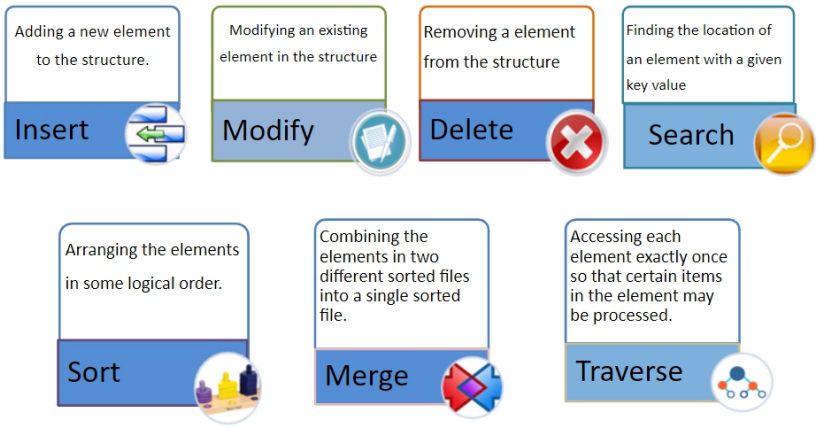
Every item might be attached with many other items

Data cannot be traversed in a single run



Types Non-Linear Data Structure

Data Structure Operations

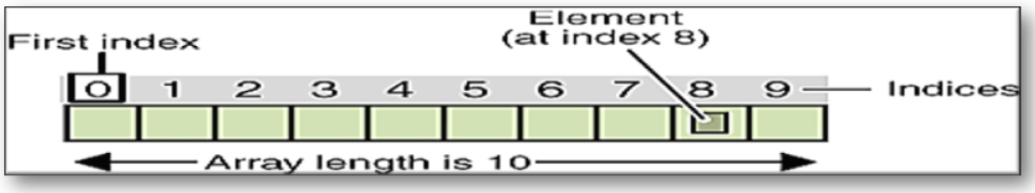


Array

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

Arrays have fixed size

Disadvantages
"Add element" is an expensive operation

Data must be shifted during insertions and deletions

Disadvantages
Fixed size

Linear List

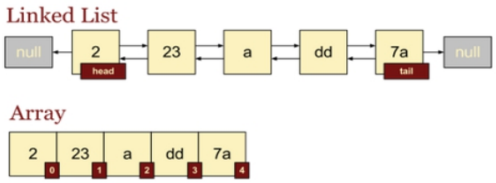
TEKNOTURF

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

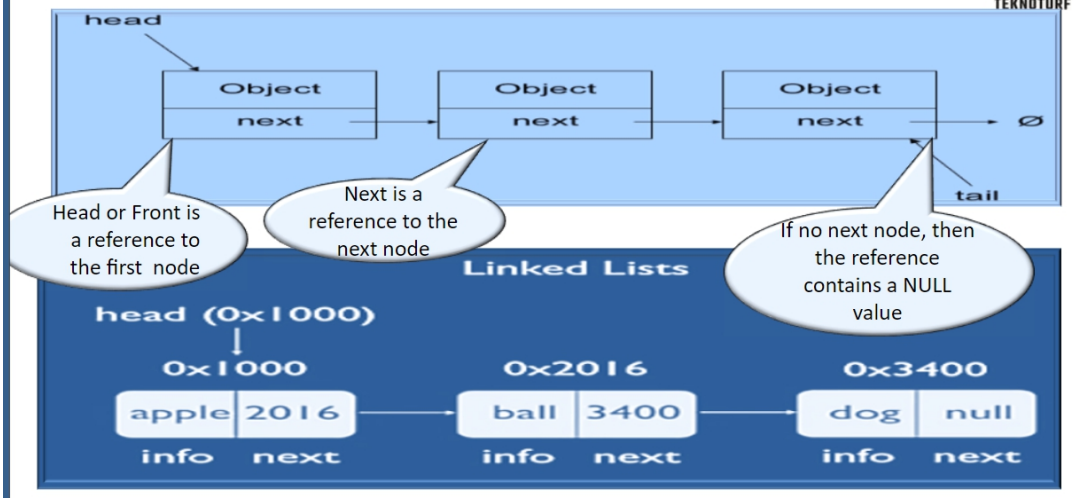
Does not require the shifting of items during insertions and deletions

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

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

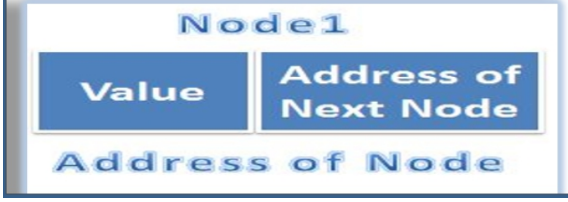
Singly Linked List Declaration

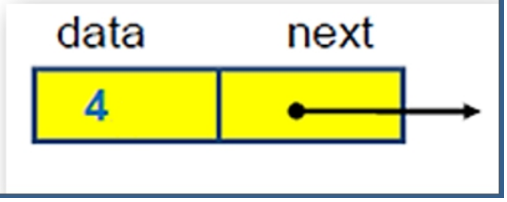
Syntax:

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

Example:

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

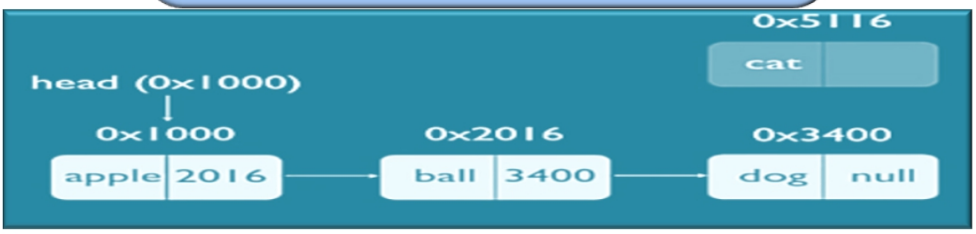




Insert a New Node to the List

Steps to insert a new node:

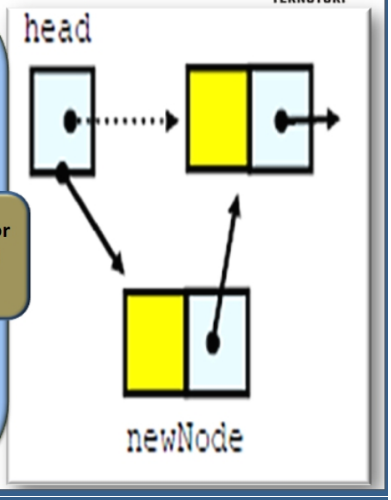
1. Find the location to insert the element
2. Allocate memory for the new node
3. Point the new node to its successor
4. Point the new node's predecessor to the new node



Insert at front or in empty list:

```
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;
          }
 }
```

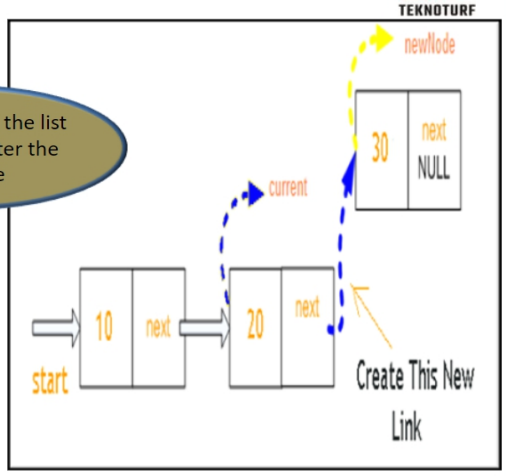
Allocate memory for the new node and put data in it



Insert at the end:

```
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

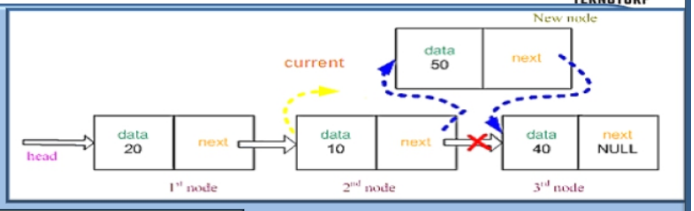


Insert in the middle

```
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

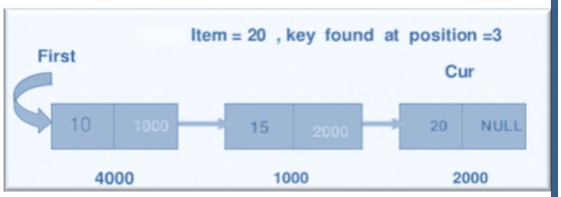
Count the number of nodes after which, the new node is to be inserted



Find and print elements

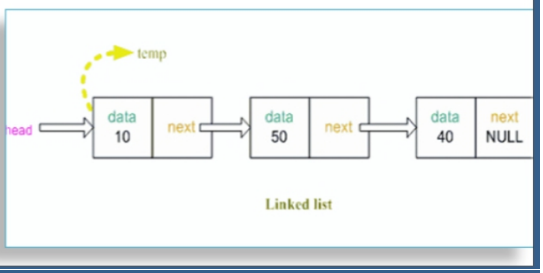
Find an element in the list

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



Display all elements in the list

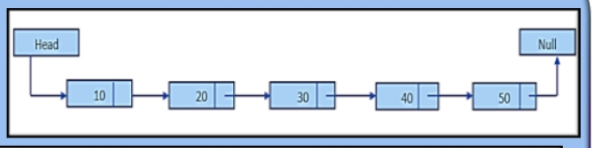
```
public void printList()  {
    Node temp = head;
    while (temp!= null)  {
        System.out.print(temp.data+" ");
        temp =temp.next;
    }
  }
```

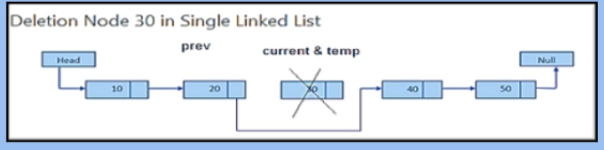


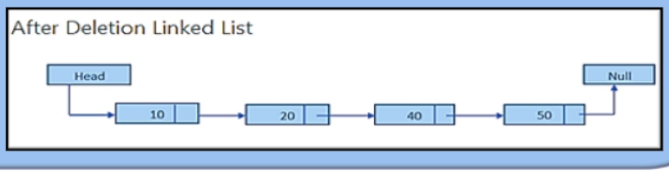
Delete an element

```
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



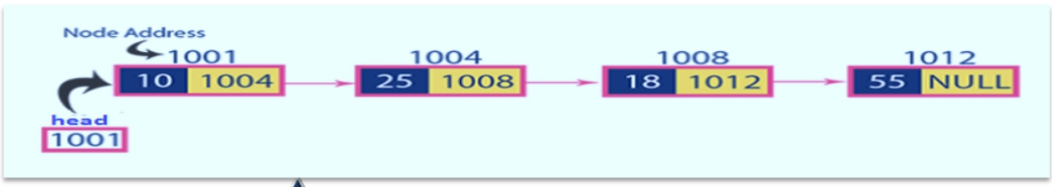




Linked list variations

TEKNOTURF

Singly Linked List



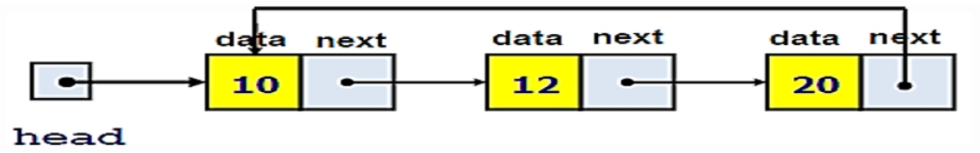
It consists of a list of elements, with a head and a tail; with each element referring to another of its own kind

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

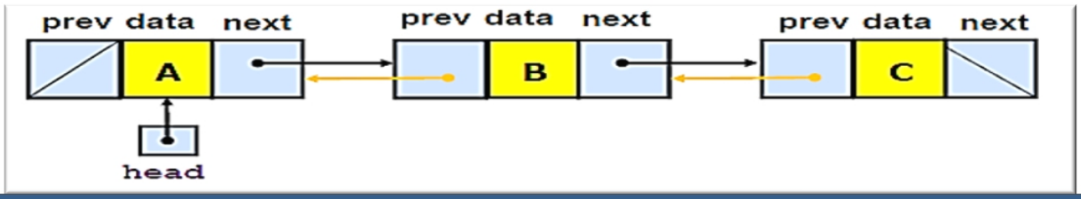
TEKNOTURF

Doubly Linked List

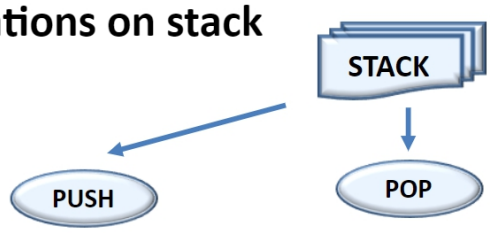
Each node in doubly linked list has two reference variables:
- One refers to the next node and
- The other refers to the previous node

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.



Operations on stack

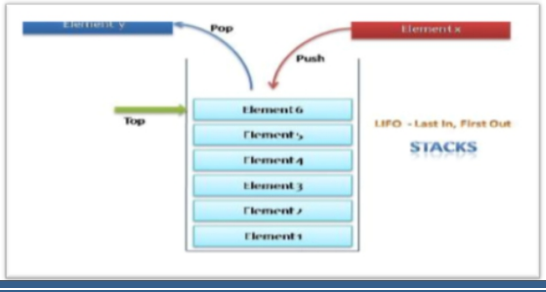


Insert an element into a stack

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

Stack can be implemented using array or linked list.

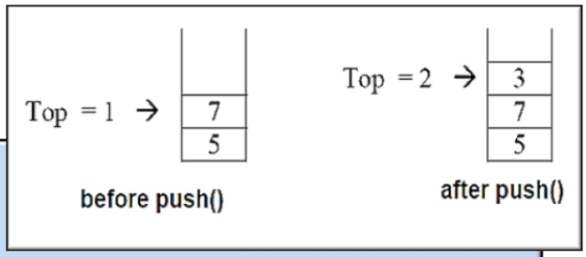
Array Implementation of stack

- 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

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

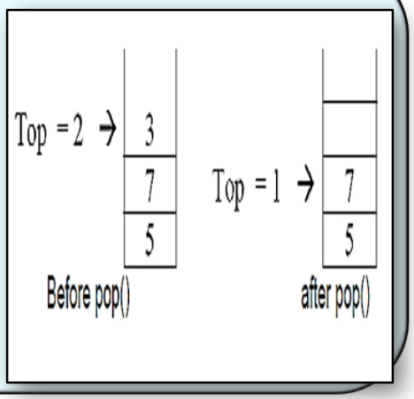
Stack Implementation using Array

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

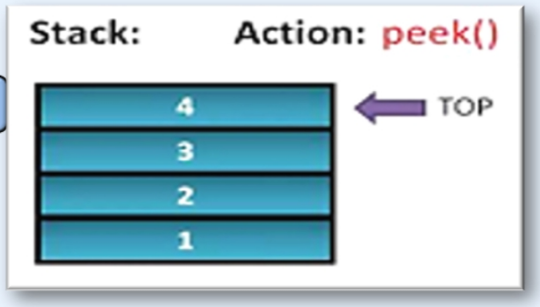


Stack Implementation using Array

Retrieve the topmost element :peek()

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

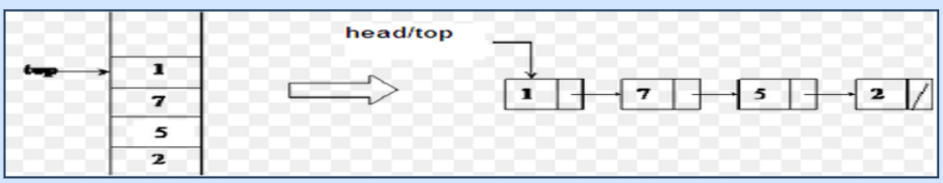
"Stack is Empty"



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

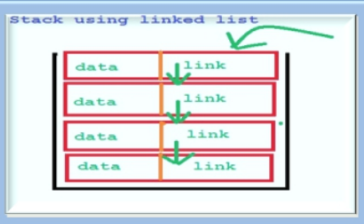


Linked List implementation of stack

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

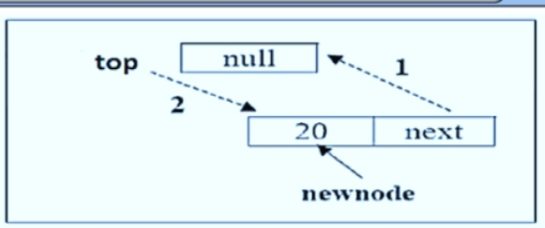


There are two possibilities for inserting an element into a stack:

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

Linked List implementation of stack

1. Insert an element into an empty stack

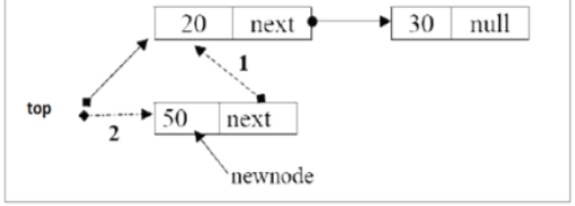


In this situation the new node being inserted, will become the first item in stack.

Linked List implementation of stack

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;

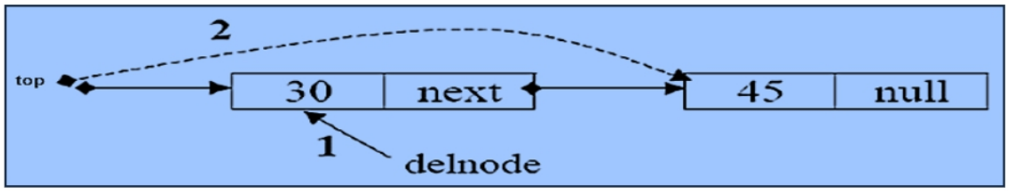
```
class linkedStack{
    protected Node top ;
    public void push(int n)  {
        Node newNode= new Node(n);
         newNode.next = top;
         top = newNode;
    }
}
```

Linked List implementation of stack

pop operation in stack :

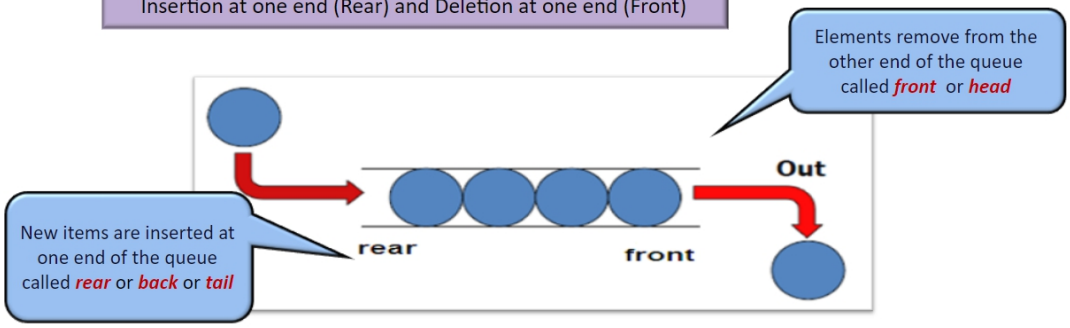
```
if(top==NULL)
        print Stack is empty.
        return
else
        step1 : delNode=top
        step 2: top=delNode.next;
        step 3:  return delNode.data;
```

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



Queue

- Linear Data Structure with restrictions
- First In First Out (FIFO)
- Insertion at one end (Rear) and Deletion at one end (Front)



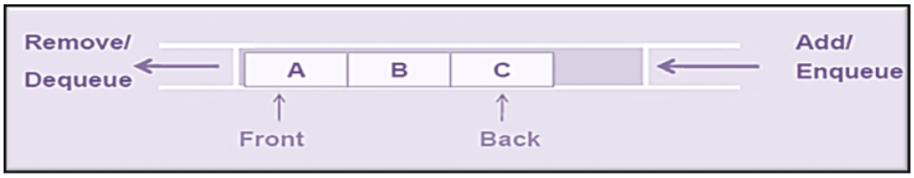
Applications of Queue

Real-World Applications	Computer Science Applications
• Cashier lines in any store	• Print lines of a document
• Check out at a bookstore	• Printer sharing between computers
• Bank/ATM	• Recognizing palindromes
• Call an airline	• Shared resource usage (CPU, memory access,...)

Queue operations

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

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;
}}
```

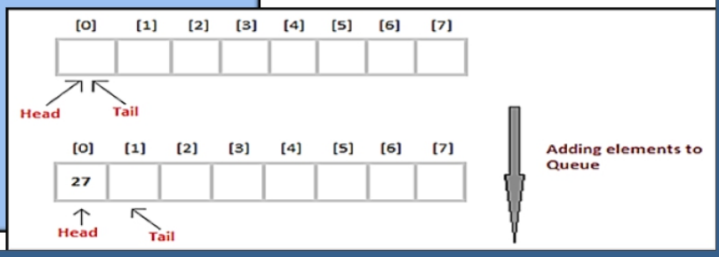
Queue Implementation using array

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



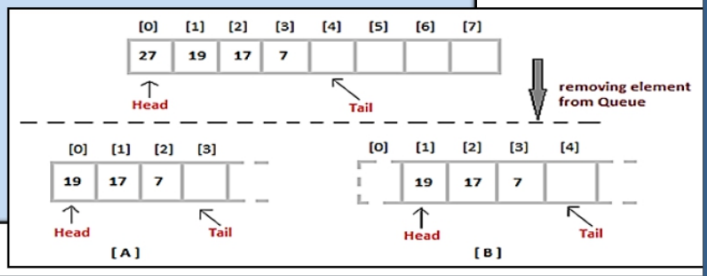
Queue Implementation using array

TEKNOTURF

Delete an element in a queue : deQueue()

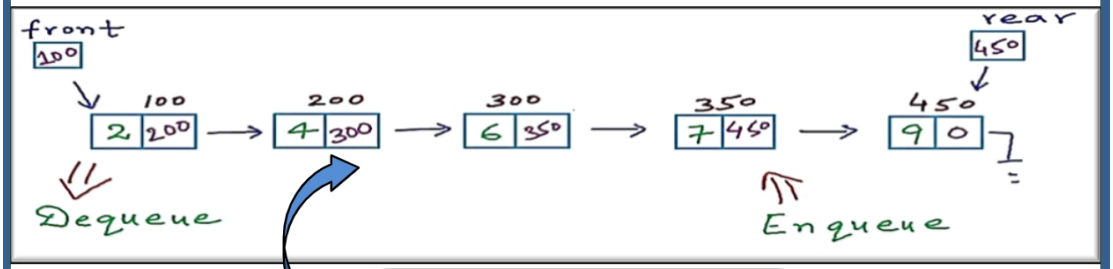
```
public int deQueue() {
    if (this.empty()) {
            System.out.printf("\nAttempt to remove from an empty queue\n");
            return 0;
     }
     int delElement = QA[front];
    front++ ;
    return delElement;

}
```

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



Queue – Linked List Implementation

TEKNOTURF




```
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

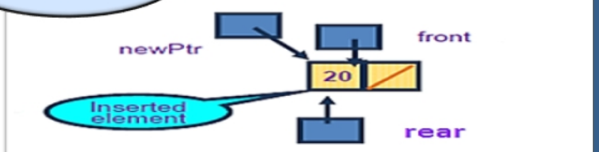
Queue Implementation using Linked List

TEKNOTURF

1. Insertion to an empty queue

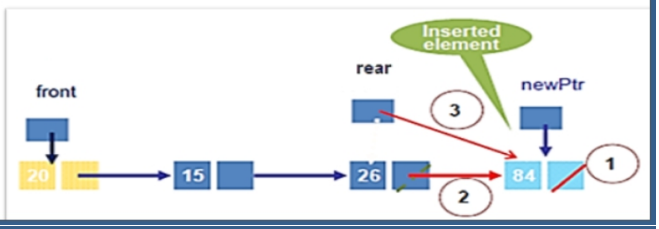
```
if(rear==NULL)
        front=newPtr;
        rear=newPtr
```

If the queue is an empty queue



2. Insertion to a non-empty queue

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

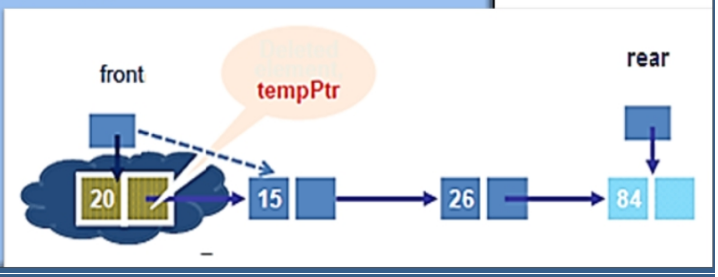


Queue Implementation using Linked List

TEKNOTURF

deQueue Operation:

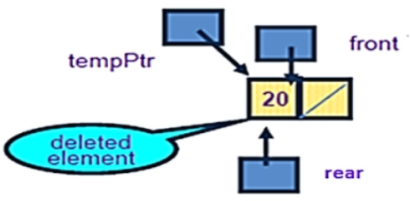
```
public Object deQueue() {
   if (this.empty()) {
      System.out.printf("\nAttempt to remove
                        from an empty queue\n");
       System.exit(1);
   }
   Object tempPtr= front.data;
   front = front.next;
   if (front == null)
          rear = null;
    return tempPtr;
}
```



Queue Implementation using Linked List

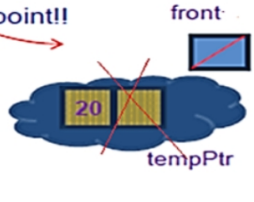
TEKNOTURF

If the deleted element is the last element in the queue, then need to add this statement:




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

After deletion, tempPtr has nowhere to point!!




```
if(front!=NULL)
        tempPtr=front;
        front=front.next;
        if(front==null)
                rear=null;
```

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

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);
    }
}
```

Output would be :
ACD