

# **Index: DSA < Part - 5 >**

---

1.Stacks

---

2.Queues

---

3.Trees

---

4.Heaps

---

5.Trie

---

6.Graphs

---

7.-----

---

8.-----

---

9.-----

---

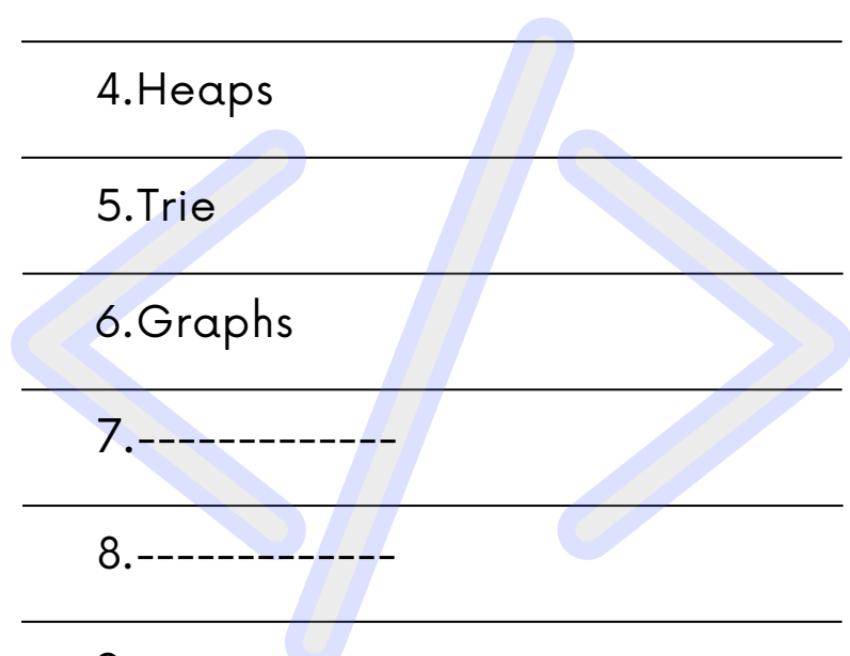
10.-----

---

11.-----

---

12.-----



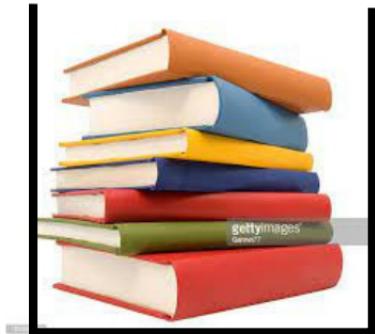
## 1. Stacks

---

### Definition of stack:

A stack is a linear data structure in which insertions and deletions are allowed only at the end, called the top of the stack.

As a stack is a linear data structure, we can implement it using an array or a linked list.



Stack of  
books



Stack of  
coins

Objects placed inside the glass jars

Both these examples have two things in common:

1. Any object (either a book or a coin) can be accessed only from the top.
2. Any object can be added only at the top.

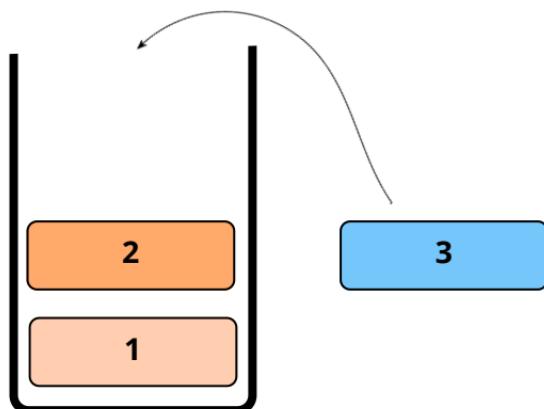
## **Stack as an ADT:**

When we define a stack as an ADT (or Abstract Data Type), then we are only interested in knowing the stack operations from the user of view.

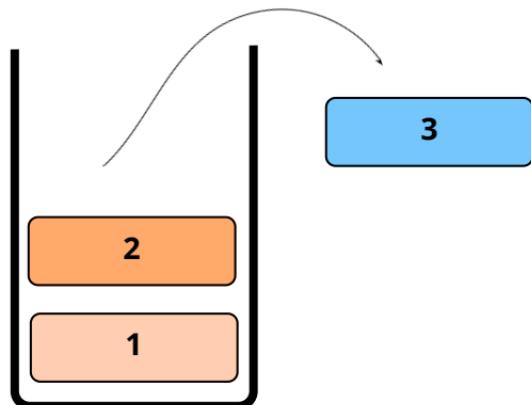
This means we are not interested in knowing the implementation details at this moment. We are only interested in knowing what types of operations we can perform on the stack.

## **Primary Stack Operations:**

**push(data):** Insert data onto stack.

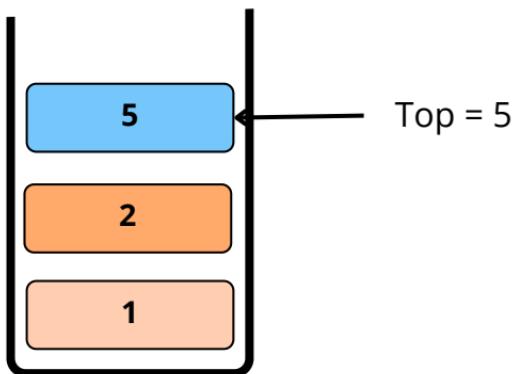


**pop():** Deletes the last inserted element from the stack.

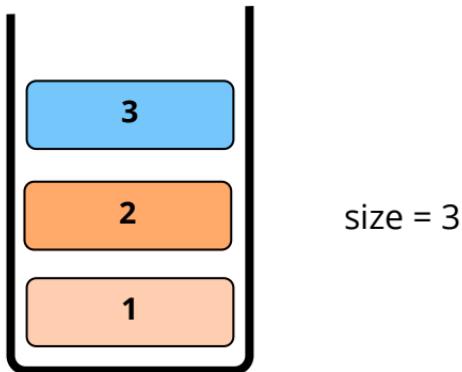


### Secondary Stack Operations:

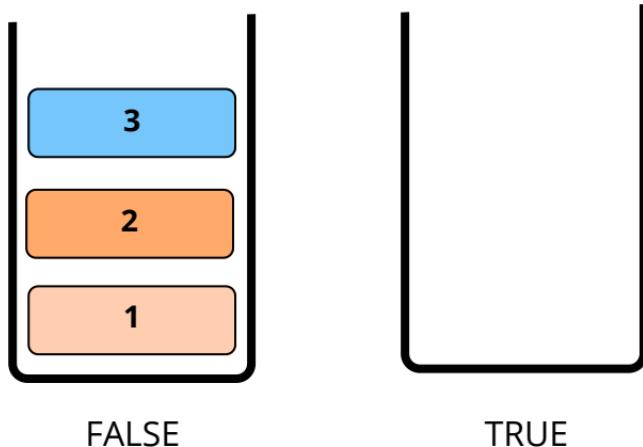
**top():** returns the last inserted element without removing it..



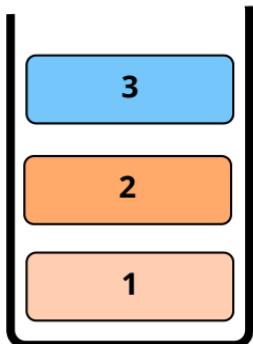
**size():** returns the size or the number of elements in the stack.



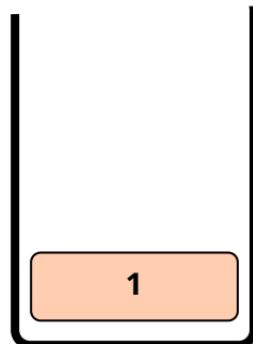
**isEmpty():** returns TRUE if the stack is empty, else returns FALSE.



**isFull():** returns TRUE if the stack is full, else returns FALSE.



TRUE



FALSE

- **Array Implementation of Stacks:**

stack\_arr =



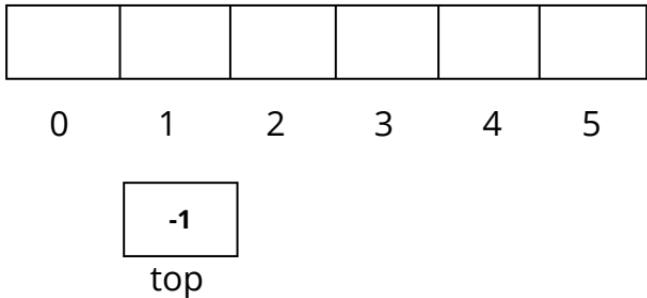
We know that stack\_arr[ ] is an array and insertion and deletion operations can be performed at any place but we want the stack\_arr[ ] to behave like a stack.

Hence, the insertion and deletion must be performed at the end.

For this, we will keep a variable top which will keep track of the last inserted element or the topmost element in an array.

### **Empty Stack:**

stack\_arr =

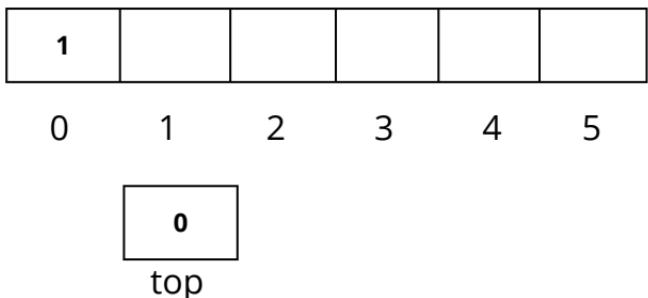


To indicate that the stack is empty, we will put -1 in the variable top.

**Note:** top = 0 indicates that the topmost element is at index 0 and this means is only one element in the stack.

## **push(1):**

stack\_arr =

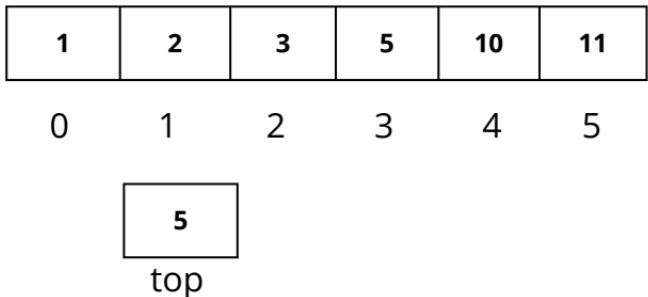


For the push operation:

- top is incremented by 1.
- New element is pushed at the position top.

Here,

stack\_arr =



push(2)

push(3)

push(5)

push(10)

push(11)

push(12) -----> stack overflow (New elements  
can't be pushed)

## **pop():**

stack\_arr =

1	2	3	5	10	
---	---	---	---	----	--

0      1      2      3      4      5

4
---

top

For the pop operation:

- The element at the position of top is deleted.
- top is decremented by 1.

Here,

stack\_arr =

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

0      1      2      3      4      5

-1
----

top

pop()

pop()

pop()

pop()

pop()

pop() -----> stack is empty. So, it can't be popped.

- **In built Stack example:**

```
import java.util.Stack;

public class Test {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();

        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println("Top element: " + stack.peek());

        System.out.println("Popped element: " + stack.pop());
        System.out.println("Popped element: " + stack.pop());

        System.out.println("Is stack empty? " +
                           stack.isEmpty());

        stack.push(40);
        stack.push(50);

        System.out.println("Size of stack: " + stack.size());

        System.out.print("Stack elements: ");
        while (!stack.isEmpty()) {
            System.out.print(stack.pop() + " ");
        }
    }
}
```

**Output:**

Top element: 30

Popped element: 30

Popped element: 20

Is stack empty? false

Size of stack: 3

Stack elements: 50 40 10

- **Array Implementation of Stacks in Java :**

```
import java.util.Arrays;

public class CustomStack {
    protected int[] data;
    private static final int DEFAULT_SIZE = 10;

    int ptr = -1;

    public CustomStack(){
        this(DEFAULT_SIZE);
    }

    public CustomStack(int size) {
        this.data = new int[size];
    }

    public boolean push(int item) {
        if (isFull()) {
            System.out.println("Stack is full!!!");
            return false;
        }
        ptr++;
        data[ptr] = item;
        return true;
    }
}
```

```
public int pop() throws Exception {  
    if (isEmpty()) {  
        throw new Exception("Cannot pop from an  
empty stack!!");  
    }  
    //    int removed = data[ptr];  
    //    ptr--;  
    //    return removed;  
    return data[ptr--];  
}  
  
public int peek() throws Exception {  
    if (isEmpty()) {  
        throw new Exception("Cannot peek from an  
empty stack!!");  
    }  
    return data[ptr];  
}  
  
public boolean isFull() {  
    return ptr == data.length - 1; // ptr is at last  
index  
}  
  
public boolean isEmpty() {  
    return ptr == -1;  
}  
  
public static void main(String[] args) throws  
Exception {  
    CustomStack stack = new CustomStack(5);  
}
```

```
stack.push(34);
stack.push(45);
stack.push(2);
stack.push(9);
stack.push(18);

System.out.println("peek = " + stack.peek());

System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());

}

}
```

### **Output:**

```
peek = 18
18
9
2
45
34
```

- **Array Implementation of DynamicStacks in Java :**

```
import java.util.Arrays;

public class CustomStack {
    protected int[] data;
    private static final int DEFAULT_SIZE = 10;

    int ptr = -1;

    public CustomStack(){
        this(DEFAULT_SIZE);
    }

    public CustomStack(int size) {
        this.data = new int[size];
    }

    public boolean push(int item) {
        if (isFull()) {
            System.out.println("Stack is full!!!");
            return false;
        }
        ptr++;
        data[ptr] = item;
        return true;
    }
}
```

```
public int pop() throws Exception {  
    if (isEmpty()) {  
        throw new Exception("Cannot pop from an  
empty stack!!");  
    }  
    //    int removed = data[ptr];  
    //    ptr--;  
    //    return removed;  
    return data[ptr--];  
}  
  
public int peek() throws Exception {  
    if (isEmpty()) {  
        throw new Exception("Cannot peek from an  
empty stack!!");  
    }  
    return data[ptr];  
}  
  
public boolean isFull() {  
    return ptr == data.length - 1; // ptr is at last  
index  
}  
  
public boolean isEmpty() {  
    return ptr == -1;  
}  
  
public static void main(String[] args) throws  
Exception {  
    CustomStack stack = new CustomStack(5);  
}
```

```
stack.push(34);
stack.push(45);
stack.push(2);
stack.push(9);
stack.push(18);

System.out.println("peek = " + stack.peek());

System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());

}

}
```

```
public class DynamicStack extends CustomStack{  
  
    public DynamicStack() {  
        super(); // it will call CustomStack()  
    }  
  
    public DynamicStack(int size) {  
        super(size); // it will call CustomStack(int size)  
    }  
  
    @Override  
    public boolean push(int item) {  
        // this takes care of it being full  
        if (this.isFull()) {  
            // double the array size  
            int[] temp = new int[data.length * 2];  
  
            // copy all previous items in new data  
            for (int i = 0; i < data.length; i++) {  
                temp[i] = data[i];  
            }  
  
            data = temp;  
        }  
  
        // at this point we know that array is not full  
        // insert item  
        return super.push(item);  
    }  
}
```

```
public static void main(String[] args) throws Exception {  
    DynamicStack stack = new DynamicStack(5);  
  
    stack.push(34);  
    stack.push(45);  
    stack.push(2);  
    stack.push(9);  
    stack.push(18);  
    stack.push(19);  
    stack.push(69);  
  
    System.out.println("peek = " + stack.peek());  
  
    System.out.println(stack.pop());  
    System.out.println(stack.pop());  
    System.out.println(stack.pop());  
    System.out.println(stack.pop());  
    System.out.println(stack.pop());  
  
}  
}
```

### **Output:**

peek = 69  
69  
19  
18  
9  
2

- **Linked List Implementation of Stacks in Java :**

```
public class StackAsLinkedList {  
  
    StackNode root;  
  
    static class StackNode {  
        int data;  
        StackNode next;  
  
        StackNode(int data) { this.data = data; }  
    }  
  
    public boolean isEmpty()  
    {  
        if (root == null) {  
            return true;  
        }  
        else  
            return false;  
    }  
  
    public void push(int data)  
    {  
        StackNode newNode = new StackNode(data);  
  
        if (root == null) {  
            root = newNode;  
        }  
        else {  
            StackNode temp = root;
```

```
root = newNode;
newNode.next = temp;
}
System.out.println(data + " pushed to stack");
}
```

```
public int pop()
{
int popped = Integer.MIN_VALUE;
if (root == null) {
    System.out.println("Stack is Empty");
}
else {
    popped = root.data;
    root = root.next;
}
return popped;
}
```

```
public int peek()
{
if (root == null) {
    System.out.println("Stack is empty");
    return Integer.MIN_VALUE;
}
else {
    return root.data;
}
}
```

```
// Driver code
public static void main(String[] args)
{
    StackAsLinkedList sll = new StackAsLinkedList();

    sll.push(10);
    sll.push(20);
    sll.push(30);

    System.out.println(sll.pop()
        + " popped from stack");

    System.out.println("Top element is " + sll.peek());
}
```

### **Output:**

```
10 pushed to stack
20 pushed to stack
30 pushed to stack
30 popped from stack
Top element is 20
```

- **Stack Application:**
- **Application of Stacks (Nested Brackets):**

- **Expression Evaluation and Conversion:**

5 + 2

<Operand> <Operator> <Operand>

p \* q

(5+2) \* 6

(5+2) \* (6/2)

### **Operator Precedence:**

1. Parenthesis [ ], {}, ()
2. Exponent (right to left)
3. Division/Multiplication (left to right)
4. Addition/Subtraction (left to right)

•  $5 - 2 * 3 / 6 - 10 + 2 / 7$

=>  $5 - \{ (2 * 3) / 6 \} - 10 + (2 / 7)$

Infix :    5\*3    <operand> <op> <operand>

Prefix : \*53    <op> <operand> <operand>

Postfix : 53\*    <operand> <operand> <op>

- **Evaluation of Postfix Expression using stack:**

Expression:

$5 * 2 / 3 - 4 + 6 * 17 + 2$

- **Infix to Postfix:**

$$\begin{aligned}5 * 2 / 3 - 4 + 6 * 17 + 2 \\= \{ (5 * 2) / 3 \} - 4 + (6 * 17) + 2 \\= \{ (52^*) / 3 \} - 4 + (6, 17^*) + 2 \\= \{ 52^* 3 / \} - 4 + (6, 17^*) + 2 \\= (52^* 3 / 4 -) + (6, 17^*) + 2 \\= (52^* 3 / 4 - 6, 17^* +) + 2 \\= 52^* 3 / 4 - 6, 17^* + 2 + \text{-----> Postfix}\end{aligned}$$

## Evaluate Postfix Expression:

52\*3/4 - 6, 17\* + 2 +

### Symbol

5

52

52\*

52\*3

52\*3/

52\*3/4

52\*3/4-

52\*3/4-6

52\*3/4-6 17

52\*3/4-6 17\*

52\*3/4-6 17\*+

52\*3/4-6 17\*+2

52\*3/4-6 17\*+2+

### Stack

5

5 2

10

10 3

3.33

3.33 4

-0.67

-0.67 6

-0.67 6 17

-0.67 102

101.33

101.33 2

103.33

op1=5, op2=2 = Push  
res=(op1\*op2)=10

- **Algorithm Evaluate Postfix Expression:**

- **Evaluation of Prefix Expression using stack:**

$$\begin{aligned} & A * B + C * D - E \\ &= (A * B) + (C * D) - E \\ &= (*AB) + (*CD) - E \\ &= \{ + (*AB) (*CD) \} - E \\ &= - \{ +(*AB) (*CD) \} E \\ &= - + * AB * CDE \end{aligned}$$

[Prefix Expression]

Let,  
A=2  
B=3  
C=4  
D=5  
E=7

### Evaluate Infix Expression:

$$\begin{aligned} & A * B + C * D - E \\ &= 2 * 3 + 4 * 5 - 7 \\ &= 6 + 20 - 7 \\ &= 19 \end{aligned}$$

## Evaluate Prefix Expression:

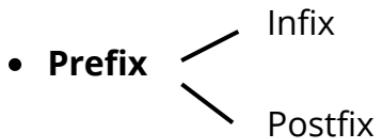
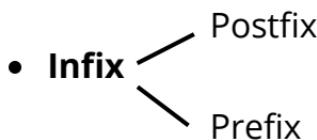
-+\*AB\*CDE

-+\*23\*457

Symbol	Stack					
7	<table border="1"><tr><td>7</td></tr></table>	7				
7						
5	<table border="1"><tr><td>7</td><td>5</td></tr></table>	7	5			
7	5					
4	<table border="1"><tr><td>7</td><td>5</td><td>4</td></tr></table>	7	5	4		
7	5	4				
*	<table border="1"><tr><td>7</td><td>20</td></tr></table>	7	20	op1=4, op2=5, res=20		
7	20					
3	<table border="1"><tr><td>7</td><td>20</td><td>3</td></tr></table>	7	20	3		
7	20	3				
2	<table border="1"><tr><td>7</td><td>20</td><td>3</td><td>2</td></tr></table>	7	20	3	2	
7	20	3	2			
*	<table border="1"><tr><td>7</td><td>20</td><td>6</td></tr></table>	7	20	6	op1=2, op2=3, res=6	
7	20	6				
+	<table border="1"><tr><td>7</td><td>26</td></tr></table>	7	26	op1=6, op2=20, res=26		
7	26					
-	<table border="1"><tr><td>19</td></tr></table>	19	op1=26, op2=7, res=19			
19						

- **Algorithm Evaluate Prefix Expression:**

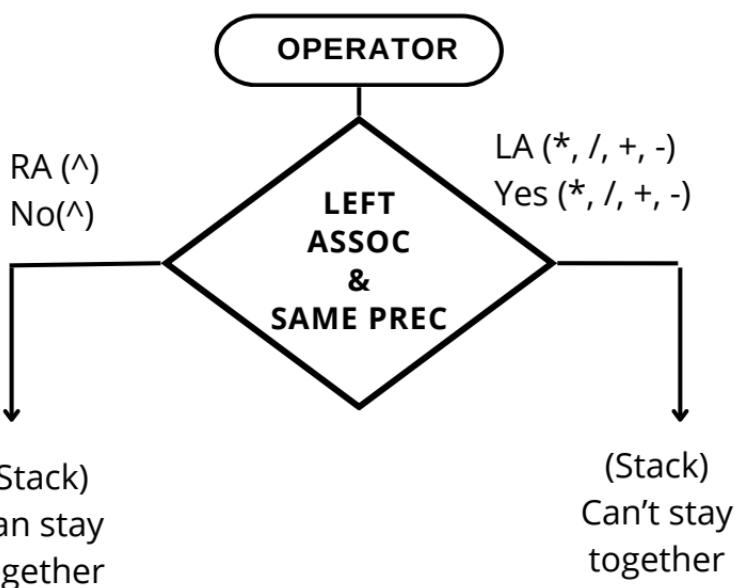
- **Converting Expression:**



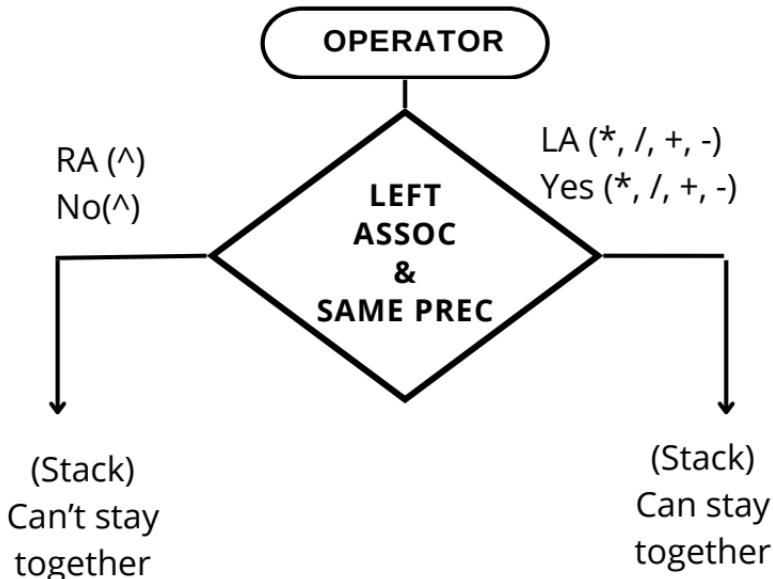
**Note:**

- For converting Postfix to Infix or Prefix to Infix we must need to use bracket ( () ) among two operands.
- For converting Postfix to Prefix or Prefix to Postfix we don't need to use bracket ( () ) among two operands.

- Infix to Postfix:



- Infix to Prefix:



- **Infix to Postfix:**

Scan the symbol of the expression from left to right and for each symbol, do the following:

a. If the symbol is an operand :

=> Print that symbol onto the screen.

b. If the symbol is a left parenthesis :

=> Push it on the stack.

c. If the symbol is a right parenthesis :

=> Pop all the operators from the stack up to the first left parenthesis and store the operators in the postfix array.

=> Discard the left and right parenthesis.

d. If the symbol is an operator :

=> If the precedence of the operators in the stack are greater than or equal to the current operator, then pop the operators out of the stack and print them onto the screen, and push the current operator onto the stack.

=> else

Push the current operator onto the stack.

- Convert Infix Expression to Postfix using Stack:

**Example:**

A+B\*C-D/E ---> Postfix ABC\*+DE/-

Symbol	Stack	Postfix
A		A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
-	-	ABC*+
D	-	ABC*+D
/	-/	ABC*+D
E	-/	ABC*+DE
	-	ABC*+DE/
		ABC*+DE-

- Convert Infix Expression to Postfix using Stack:

### **Example:**

$P^*Q-D/E+(F-H)/G+(H^*E)$

### **Infix:**

$$\begin{aligned}
 & P^*Q-D/E+(F-H)/G+(H^*E) \\
 &= P^*Q-D/E+(FH-)/G+(HE^*) \\
 &= (P^*Q)-(D/E)+\{(FH-)/G\}+(HE^*) \\
 &= (PQ^*)-(DE/)+\{(FH-)G/\}+(HE^*) \\
 &= \{(PQ^*) (DE/-)\} + \{(FH-)G/\} + (HE^*) \\
 &= [ \{(PQ^*) (DE/-)\} \{(FH-)G/\} ] + (HE^*) \\
 &= [ \{(PQ^*) (DE/-)\} \{(FH-)G/\} ] (HE^*)+ \\
 &= PQ^*DE/-FH-G/+HE^*+
 \end{aligned}$$

[Postfix Expression]

or,

$$\begin{aligned}
 & P^*Q-D/E+(F-H)/G+(H^*E) \\
 &= P^*Q-D/E+(FH-)/G+(HE^*) \\
 &= (P^*Q)-(D/E)+\{(FH-)/G\}+(HE^*) \\
 &= (PQ^*)-(DE/)+(FH-G/)+(HE^*) \\
 &= (PQ^*DE/-) + (FH-G/) + (HE^*) \\
 &= (PQ^*DE/-FH-G/+) + (HE^*) \\
 &= (PQ^*DE/-FH-G/+HE^*)+ \\
 &= PQ^*DE/-FH-G/+HE^*+
 \end{aligned}$$

## Conversion of Infix to Postfix Expression:

P\*Q-D/E+(F-H)/G+(H\*E)

Symbol	Stack	Postfix
P		P
*	*	P
Q	*	PQ
-	-	PQ*
D	-	PQ*D
/	-/	PQ*D
E	-/	PQ*DE
+	+	PQ*DE/-
(	+()	PQ*DE/-
F	+()	PQ*DE/-F
-	+(-	PQ*DE/-F
H	+(-	PQ*DE/-FH
)	+	PQ*DE/-FH-
/	+/	PQ*DE/-FH-

G	+ /	PQ*DE/-FH-G
+	+	PQ*DE/-FH-G/+
(	+(	PQ*DE/-FH-G/+
H	+ (	PQ*DE/-FH-G/+H
*	+ (*	PQ*DE/-FH-G/+H
E	+ (*	PQ*DE/-FH-G/+HE
)	+	PQ*DE/-FH-G/+HE*
		PQ*DE/-FH-G/+HE*+

- Convert Infix Expression to Postfix using Stack:

**Example:**

$F^*(C^A^B)$

**Infix:**

$F^*(C^A^B)$

=  $F^*\{C^{\{AB\}}\}$  [  $^*$  --> right to left Associativity]

=  $F^*\{C\{AB\}^{\}}$

= FCAB $^{^{\wedge \wedge}}^*$

### Conversion of Infix to Postfix Expression:

$F^*(C^A^B)$

Symbol	Stack	Postfix
F		F
*	*	F
(	*(	F
C	*(	FC
$^{\wedge}$	$*(^{\wedge}$	FC
A	$*(^{\wedge}$	FCA
$^{\wedge}$	$*(^{\wedge \wedge}$	FCA
B	$*(^{\wedge \wedge}$	FCAB
)	*	FCAB $^{^{\wedge \wedge}}^*$
		FCAB $^{^{\wedge \wedge}}^*$

- Convert Infix Expression to Postfix using Stack:

**Example:**

$P^*Q-D/E+(F-H+D+Q^P^E)+(H^*E/P^*Q)$

**Infix:**

$P^*Q-D/E+(F-H+D+Q^P^E)+(H^*E/P^*Q)$   
= [Postfix Expression]

or,

$$\begin{aligned} & P^*Q-D/E+(F-H)/G+(H^*E) \\ & = P^*Q-D/E+(FH-)/G+(HE^*) \\ & = (P^*Q)-(D/E)+\{(FH-)/G\}+(HE^*) \\ & = (PQ^*)-(DE/-)+(FH-G/)+(HE^*) \\ & = (PQ^*DE/-)+(FH-G/)+(HE^*) \\ & = (PQ^*DE/-FH-G/+)+(HE^*) \\ & = (PQ^*DE/-FH-G/+HE^*) \\ & = PQ^*DE/-FH-G/+HE^* \end{aligned}$$

## Conversion of Infix to Postfix Expression:

P\*Q-D/E+(F-H)/G+(H\*E)

Symbol	Stack	Postfix
P		P
*	*	P
Q	*	PQ
-	-	PQ*
D	-	PQ*D
/	-/	PQ*D
E	-/	PQ*DE
+	+	PQ*DE/-
(	+()	PQ*DE/-
F	+()	PQ*DE/-F
-	+(-	PQ*DE/-F
H	+(-	PQ*DE/-FH
)	+	PQ*DE/-FH-
/	+/	PQ*DE/-FH-

G	+ /	PQ*DE/-FH-G
+	+	PQ*DE/-FH-G/+
(	+(	PQ*DE/-FH-G/+
H	+ (	PQ*DE/-FH-G/+H
*	+ (*	PQ*DE/-FH-G/+H
E	+ (*	PQ*DE/-FH-G/+HE
)	+	PQ*DE/-FH-G/+HE*
		PQ*DE/-FH-G/+HE*+

- **Algorithm of Converting Infix to Postfix Expression:**

```
import java.util.Stack;
import java.util.Scanner;

public class InfixToPostfix {

    static final int MAX = 20;
    static Stack<Character> stack = new Stack<>();

    static boolean isOperator(char ch) {
        return ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '^';
    }

    static int precedence(char ch) {
        switch (ch) {
            case '+':
            case '-':
                return 1;
            case '*':
            case '/':
                return 2;
            case '^':
                return 3;
        }
        return -1;
    }

    static boolean isEmpty() {
```

```
        return stack.isEmpty();
    }

static boolean isFull() {
    return stack.size() == MAX;
}

static void push(char x) {
    if (isFull()) {
        System.out.println("Error: Stack is Full!!!!");
        return;
    }
    stack.push(x);
}

static char pop() {
    if (isEmpty()) {
        System.out.println("Error: Stack is Empty!!!!");
        return '\0'; // Return a null character instead of -1
    }
    return stack.pop();
}

static boolean isBalanced(String exp) {
    for (char ch : exp.toCharArray()) {
        if (ch == '(') {
            push(ch);
        } else if (ch == ')') {
            if (isEmpty()) {
                return false;
            }
        }
    }
}
```

```
        pop();
    }
}
return isEmpty();
}

static void infixToPostfix(String infix, String[] postfix) {
    int j = 0;
    for (char ch : infix.toCharArray()) {
        if (Character.isLetterOrDigit(ch)) {
            postfix[0] += ch;
        } else if (ch == '(') {
            push(ch);
        } else if (ch == ')') {
            while (!isEmpty() && stack.peek() != '(') {
                postfix[0] += pop();
            }
            pop();
        } else if (isOperator(ch)) {
            while (!isEmpty() && precedence(stack.peek()) >=
precedence(ch)) {
                postfix[0] += pop();
            }
            push(ch);
        }
    }
    while (!isEmpty()) {
        postfix[0] += pop();
    }
}
```

```
public static void main(String[] args) {  
    Scanner scanner = new Scanner(System.in);  
  
    System.out.print("Expression dao:\n");  
    String exp = scanner.nextLine();  
  
    if (isBalanced(exp)) {  
        System.out.println("Sob kichu thik thak ase...");  
    } else {  
        System.out.println("Bracket thik koro mia!!!...");  
    }  
  
    System.out.print("Infix expression dao:\n");  
    String infix = scanner.nextLine();  
    String[] postfix = new String[1];  
    infixToPostfix(infix, postfix);  
    System.out.println(postfix[0]);  
}  
}
```

### **Output:**

Expression dao:

(12+5)-(56/7

Bracket thik koro mia!!!...Infix expression dao:

(12+5)-(56/7)

125+567/-

- Convert Infix Expression to Prefix using Stack:

**Example:**

$(M+N)*D-A+B$

**Infix:**

$$\begin{aligned}
 & (M+N)*D-A+B \\
 &= (+MN)*D-A+B \\
 &= \{*(+MN)\}-A+B \\
 &= [-\{*(+MN)D\}A]+B \\
 &= +-*+MNDAB
 \end{aligned}$$

[Prefix Expression]

**Conversion of Infix to Prefix Expression:**

Reverse :  $B+A-D*(N+M)$

Symbol	Stack	Prefix
B		B
+	+	B
A	+	BA
-	+ -	BA
D	+ -	BAD
*	+ - *	BAD

(	+-*(	BAD
N	+-*(	BADN
+	+-*(+	BADN
M	+-*(+	BADNM
)	+-*	BADNM+*
	+-	BADNM+*-
	+	BADNM+*-+

After reverse, Prefix : +-\*+MNDAB

- Convert Infix Expression to Prefix using Stack:

**Example:**

$F^*(C^A^B)$

**Infix:**

$$\begin{aligned} & F^*(C^A^B) \\ &= F^*\{C^{\{A^B\}}\} \\ &= F^*\{\{C(A^B)\}\} \\ &= *F^C^A^B \end{aligned}$$

[Prefix Expression]

**Conversion of Infix to Prefix Expression:**

Reverse :  $(B^A^C)^*F$

Symbol	Stack	Prefix
(	(	
B	(	B
$\wedge$	( $\wedge$	B
A	( $\wedge$	BA
$\wedge$	( $\wedge$	BA $\wedge$
C	( $\wedge$	BA $\wedge$ C

)		BA <sup>A</sup> C <sup>A</sup>
*	*	BA <sup>A</sup> C <sup>A</sup>
F		BA <sup>A</sup> C <sup>A</sup> F
		BA <sup>A</sup> C <sup>A</sup> F*

After reverse, Prefix : \*F<sup>A</sup>C<sup>A</sup>AB

- **Algorithm of Converting Infix to Prefix Expression:**

- Convert Postfix Expression to Infix using Stack:

### Postfix:

ABC\*DEF^G\*-H\*+

Symbol	Stack
A	A
B	A B
C	A B C
*	A (B*C)
D	A (B*C) D
E	A (B*C) D E
F	A (B*C) D E F
^	A (B*C) D (E^F)
/	A (B*C) (D/(E^F))
G	A (B*C) (D/(E^F)) G
*	A (B*C) ((D/(E^F))*G)
-	A ((B*C)-((D/(E^F))*G))

H

A	$((B^C) - ((D/(E^F))^G))$	H
---	---------------------------	---

\*

A	$(((B^C) - ((D/(E^F))^G))^H)$
---	-------------------------------

+

(A+((B*C)-((D/(E^F))^G))^H))
------------------------------

Postfix: (A+((B\*C)-((D/(E^F))^G))^H))

- **Algorithm of Converting Postfix to Infix Expression:**

- Convert Prefix Expression to Infix using Stack:

**Prefix:**

/A+BC

Symbol	Stack		
C	<table border="1"><tr><td>C</td></tr></table>	C	
C			
B	<table border="1"><tr><td>C</td><td>B</td></tr></table>	C	B
C	B		
+	<table border="1"><tr><td>(B+C)</td></tr></table>	(B+C)	
(B+C)			
A	<table border="1"><tr><td>(B+C)</td><td>A</td></tr></table>	(B+C)	A
(B+C)	A		
/	<table border="1"><tr><td>(A/(B*C))</td></tr></table>	(A/(B*C))	
(A/(B*C))			

Prefix: (A/(B\*C))

- Convert Prefix Expression to Infix using Stack:

**Prefix:**

+P\*-\*QR\*/S^TUVW

Symbol	Stack
W	W
V	W V
U	W V U
T	W V U T
^	W V (T^U)
S	W V (T^U) S
/	W V (S/(T^U))
*	W ((S/(T^U))*V)
R	W ((S/(T^U))*V) R
Q	W ((S/(T^U))*V) R Q
*	W ((S/(T^U))*V) (Q*R)
-	W ((Q*R)-((S/(T^U))*V))

\*

$$(((Q^*R)-((S/(T^U))^*V))^*W)$$

P

$$(((Q^*R)-((S/(T^U))^*V))^*W) \quad | \quad P$$

+

$$(P+(((Q^*R)-((S/(T^U))^*V))^*W))$$

Prefix:  $(P+(((Q^*R)-((S/(T^U))^*V))^*W))$

- **Algorithm of Converting Prefix to Infix Expression:**

- Convert Prefix Expression to Postfix using Stack:

**Prefix:**

+P\*-\*QR\*/S^TUVW

Symbol	Stack
W	W
V	W V
U	W V U
T	W V U T
^	W V TU^
S	W V TU^ S
/	W V STU^/
*	W STU^/V*
R	W STU^/V* R
Q	W STU^/V* R Q
*	W STU^/V* QR*
-	W QR*STU^/V*-

\*

QR\*STU^/V\*-W\*

P

QR\*STU^/V\*-W\*

P

+

PQR\*STU^/V\*-W\*+

Postfix: PQR\*STU^/V\*-W\*+

- **Algorithm of Converting Prefix to Postfix Expression:**

- Convert Postfix Expression to Prefix using Stack:

**Postfix:**

PQR\*STU<sup>^</sup>/V\*-W\*+

Symbol	Stack
P	P
Q	P Q
R	P Q R
*	P *QR
S	P *QR S
T	P *QR S T
U	P *QR S T U
^	P *QR S ^TU
/	P *QR /S^TU
V	P *QR /S^TU V
*	P *QR */S^TUV
-	P -*QR*/S^TUV

W

P	-*QR*/S^TUV	W
---	-------------	---

\*

P	*-*QR*/S^TUVW
---	---------------

+

+P*-*QR*/S^TUVW
-----------------

Prefix: +P\*-\*QR\*/S^TUVW

- **Algorithm of Converting Postfix to Prefix Expression:**

- **Bracket problem(with position) of expression :**

```
import java.util.Scanner;
import java.util.Stack;

public class Test {

    static boolean isBalanced(String s) {
        Stack<Character> stack = new Stack<>();

        for (int i = 0; i < s.length(); i++) {
            char ch = s.charAt(i);
            if (ch == '(' || ch == '{' || ch == '[') {
                // System.out.println("Push"+ " "+ch);
                stack.push(ch);
            } else if (ch == ')' || ch == '}' || ch == ']') {
                if (stack.isEmpty()) {
                    System.out.println(i + " position " + ch + " beshi
keno??");
                    return false;
                }
                char c, ct=stack.pop();
                // System.out.println("pop "+ i+" "+ch);
                if(ct=='('){
                    c=')';
                }else if(ct=='{'){
                    c='}';
                }else if(ct=='['){
                    c=']';
                }else{
                    c='.';
                }
            }
        }
    }
}
```

```
}

if (c != ch) {
    return false;
}
}

return stack.isEmpty();
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    System.out.print("Expression dao\n");
    String s = scanner.nextLine();

    if (isBalanced(s)) {
        System.out.println("All is OK!");
    } else {
        System.out.println("braket thik kor Mia :D");
    }

    System.out.println(s.length());
}
}
```

### **Output:**

Expression dao  
(46+7)+8-  
9 position ) beshi keno??  
braket thik kor Mia :D  
10

## **2. Queues**

---

### **Definition of queue:**

A queue is a linear data structure in which elements are inserted from one end called the rear end and deleted from another end called the front end.

It follows FIFO (First In First Out) technique.  
Insertion in the queue is called enqueue and deletion in the queue is called dequeue.

Queues can be implemented in two ways: Array-based queues and list-based queues.

### **Queue Implementations:**

- Queue As Array
- Queue As Circular Array
- Queue As Linked List

In array-based queues are implemented using the arrays. In list-based queues are implemented using a linked list.

- **Array-Based vs List-Based Queues:**

Sno.	Array-Based Queues	List-Based Queues
1.	The size of the queue should be known in advance.	It's not necessary to know the size of the queue in advance.
2.	Insertion at the end is easier but insertion at the beginning is difficult.	Insertion at both end and beginning is easy.
3.	Elements can be accessed randomly.	Elements can be accessed sequentially only.
4.	Resizing array-based queues is complex.	Resizing list-based queues is simple.
5.	Requires less memory.	Requires more memory.
6.	It is faster compared to list-based queues.	It is slower compared to array-based queues.

Property	Array-Based Queues	List-Based Queues
Underlying Data Structure	Array	Linked List
Access Time	Constant Time	$O(n)$ Time
Memory Management	Fixed Size, May Require Resizing	Dynamic Size
Insertion/Deletion	Rear: $O(1)$ Time, Front: $O(n)$ Time	Rear: $O(1)$ Time, Front: $O(1)$ Time
Implementation	Easier to Implement	More Complex to Implement
Advantages	Constant Access Time, Efficient for Small Queues	Dynamic Size, Efficient for Large Queues
Disadvantages	Fixed Size, May Require Resizing, Inefficient for Large Queues	Slower Access Time, More Complex to Implement

Queues are linear data structures in which we add elements to one end and remove them from the other end.



### Queue Data Structure

The first item to be inserted (en-queued) is the first to be deleted (de-queued). A queue is therefore called a First In First Out (FIFO) data structure.

Queue operations:

- **Enqueue(value):** insert an element at the rear of the queue
- **Dequeue():** delete an element at the front of the queue
- **GetHead():** get the element at the front without deleting it

- **Enqueue(value):**

1. Empty Queue



2. Enqueue(7)



Front  
Node



Rear  
Node



Pointer  
Representation

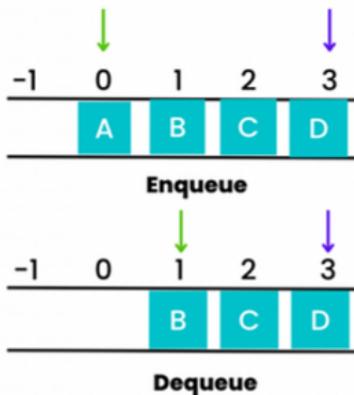
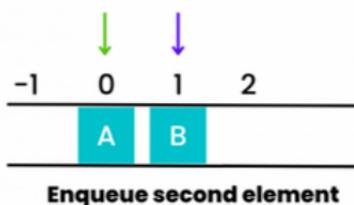
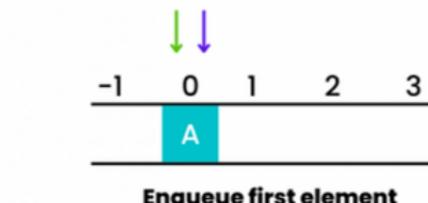
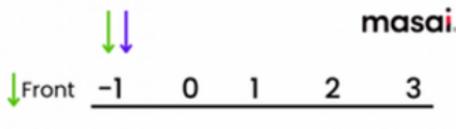
3. Enqueue(2)



4. Enqueue(-9)



- **Dequeue():**



- **In built Queue :**

```
import java.util.*;  
  
public class InBuiltExamples {  
  
    public static void main(String[] args) {  
  
        Queue<Integer> queue = new LinkedList<>();  
        queue.add(3);  
        queue.add(6);  
        queue.add(5);  
        queue.add(19);  
        queue.add(1);  
  
        System.out.println(queue.remove());  
        System.out.println(queue.remove());  
  
    }  
}
```

**Output:**

3  
6

- **Queue Implementation :**

Queue Implementations:

- Queue As Array
- Queue As Circular Array
- Queue As Linked List

- **QueueAsArray Implementation :**

```
public class CustomQueue {  
    private int[] data;  
  
    private static final int DEFAULT_SIZE = 10;  
  
    int end = 0;  
  
    public CustomQueue(){  
        this(DEFAULT_SIZE);  
    }  
  
    public CustomQueue(int size) {  
        this.data = new int[size];  
    }  
  
    public boolean isFull() {  
        return end == data.length; // ptr is at last index  
    }  
  
    public boolean isEmpty() {  
        return end == 0;  
    }  
  
    public boolean insert(int item) {  
        if (isFull()) {  
            return false;  
        }  
        data[end++] = item;  
        return true;  
    }  
}
```

```
public int remove() throws Exception {  
    if (isEmpty()) {  
        throw new Exception("Queue is empty");  
    }  
  
    int removed = data[0];  
  
    // shift the elements to left  
    for (int i = 1; i < end; i++) {  
        data[i-1] = data[i];  
    }  
    end--;  
    return removed;  
}  
  
public int front() throws Exception{  
    if (isEmpty()) {  
        throw new Exception("Queue is empty");  
    }  
    return data[0];  
}  
  
public void display() {  
    for (int i = 0; i < end; i++) {  
        System.out.print(data[i] + " <- ");  
    }  
    System.out.println("END");  
}
```

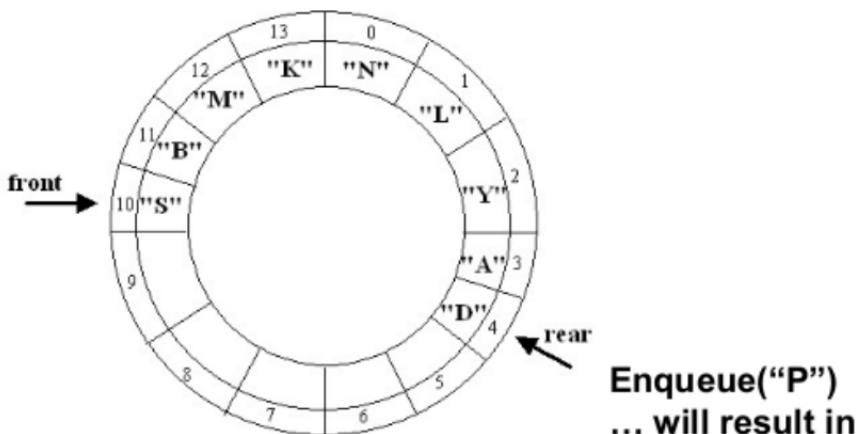
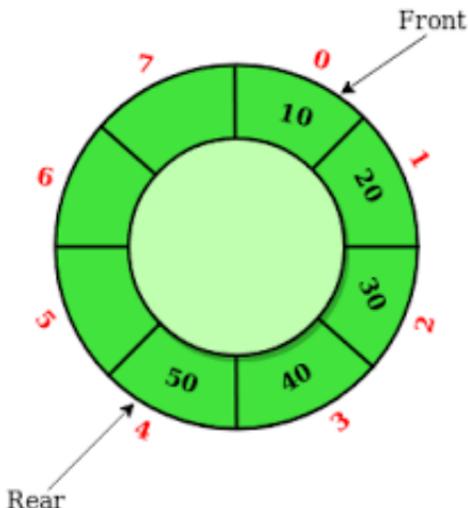
```
public static void main(String[] args) throws Exception {  
    CustomQueue queue = new CustomQueue(5);  
    queue.insert(3);  
    queue.insert(6);  
    queue.insert(5);  
    queue.insert(19);  
    queue.insert(1);  
  
    queue.display();  
  
    System.out.println(queue.remove());  
    queue.insert(133);  
    queue.display();  
  
    System.out.println(queue.remove());  
    queue.insert(99);  
    queue.display();  
  
}  
}  
}
```

### **Output:**

```
3 <- 6 <- 5 <- 19 <- 1 <- END  
3  
6 <- 5 <- 19 <- 1 <- 133 <- END  
6  
5 <- 19 <- 1 <- 133 <- 99 <- END
```

- **QueueAsCircularArray Implementation :**

By using modulo arithmetic for computing array indexes, we can have a queue implementation in which each of the operations enqueue, dequeue, and getHead has complexity O(1)





## DEQUEUE OPERATION

- What is the output of the following Java program fragment:

```
public class CircularQueue {  
    protected int[] data;  
    private static final int DEFAULT_SIZE = 10;  
  
    protected int end = 0;  
    protected int front = 0;  
    private int size = 0;  
  
    public CircularQueue(){  
        this(DEFAULT_SIZE);  
    }  
  
    public CircularQueue(int size) {  
        this.data = new int[size];  
    }  
  
    public boolean isFull() {  
        return size == data.length; // ptr is at last index  
    }  
  
    public boolean isEmpty() {  
        return size == 0;  
    }  
  
    public boolean insert(int item) {  
        if (isFull()) {  
            return false;  
        }  
        data[end++] = item;  
        end = end % data.length;  
    }  
}
```

```
size++;
return true;
}

public int remove() throws Exception {
    if (isEmpty()) {
        throw new Exception("Queue is empty");
    }

    int removed = data[front++];
    front = front % data.length;
    size--;
    return removed;
}

public int front() throws Exception{
    if (isEmpty()) {
        throw new Exception("Queue is empty");
    }
    return data[front];
}

public void display() {
    if (isEmpty()) {
        System.out.println("Empty");
        return;
    }
    int i = front;
    do {
        System.out.print(data[i] + " -> ");
        i++;
    }
}
```

```
        i %= data.length;
    } while (i != end);
    System.out.println("END");
}

public static void main(String[] args) throws Exception {
    CircularQueue queue = new CircularQueue(5);
    queue.insert(3);
    queue.insert(6);
    queue.insert(5);
    queue.insert(19);
    queue.insert(1);

    queue.display();

    System.out.println(queue.remove());
    queue.insert(133);
    queue.display();

    System.out.println(queue.remove());
    queue.insert(99);
    queue.display();

}
}
```

### **Output:**

3 -> 6 -> 5 -> 19 -> 1 -> END

3

6 -> 5 -> 19 -> 1 -> 133 -> END

6

5 -> 19 -> 1 -> 133 -> 99 -> END

- **DynamicQueueAsArray Implementation :**

```
public class CircularQueue {  
    protected int[] data;  
    private static final int DEFAULT_SIZE = 10;  
  
    protected int end = 0;  
    protected int front = 0;  
    private int size = 0;  
  
    public CircularQueue(){  
        this(DEFAULT_SIZE);  
    }  
  
    public CircularQueue(int size) {  
        this.data = new int[size];  
    }  
  
    public boolean isFull() {  
        return size == data.length; // ptr is at last index  
    }  
  
    public boolean isEmpty() {  
        return size == 0;  
    }  
  
    public boolean insert(int item) {  
        if (isFull()) {  
            return false;  
        }  
        data[end++] = item;  
        end = end % data.length;  
    }  
}
```

```
size++;
return true;
}

public int remove() throws Exception {
    if (isEmpty()) {
        throw new Exception("Queue is empty");
    }

    int removed = data[front++];
    front = front % data.length;
    size--;
    return removed;
}

public int front() throws Exception{
    if (isEmpty()) {
        throw new Exception("Queue is empty");
    }
    return data[front];
}

public void display() {
    if (isEmpty()) {
        System.out.println("Empty");
        return;
    }
    int i = front;
    do {
        System.out.print(data[i] + " -> ");
        i++;
    }
}
```

```
        i %= data.length;
    } while (i != end);
    System.out.println("END");
}

}
}
```

```
public class DynamicQueue extends CircularQueue{

    public DynamicQueue() {
        super();
    }

    public DynamicQueue(int size) {
        super(size);
    }

    @Override
    public boolean insert(int item) {

        // this takes care of it being full
        if (this.isFull()) {
            // double the array size
            int[] temp = new int[data.length * 2];

            // copy all previous items in new data
            for (int i = 0; i < data.length; i++) {
```

```
        temp[i] = data[(front + i) % data.length];
    }
    front = 0;
    end = data.length;
    data = temp;
}

// at this point we know that array is not full
// insert item
return super.insert(item);
}

public static void main(String[] args) throws Exception {
    DynamicQueue queue = new DynamicQueue(5);
    queue.insert(3);
    queue.insert(6);
    queue.insert(5);
    queue.insert(19);
    queue.insert(1);

    queue.display();

    System.out.println(queue.remove());
    queue.insert(133);
    queue.display();

    System.out.println(queue.remove());
    queue.insert(99);
    queue.display();
}

}
```

**Output:**

3 -> 6 -> 5 -> 19 -> 1 -> END

3

6 -> 5 -> 19 -> 1 -> 133 -> END

6

5 -> 19 -> 1 -> 133 -> 99 -> END



- **QueueAsLinkedList Implementation :**

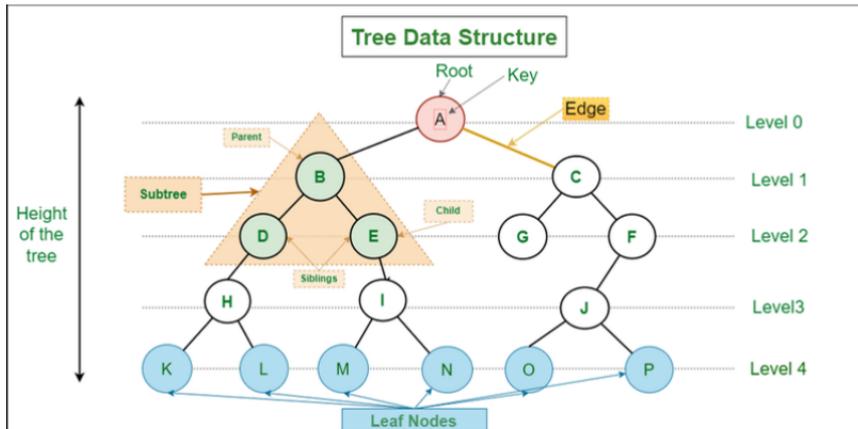
### **3. Trees**

---

A tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure

This data structure is a specialized method to organize and store data in the computer to be used more effectively. It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has roots, branches, and leaves connected with one another.



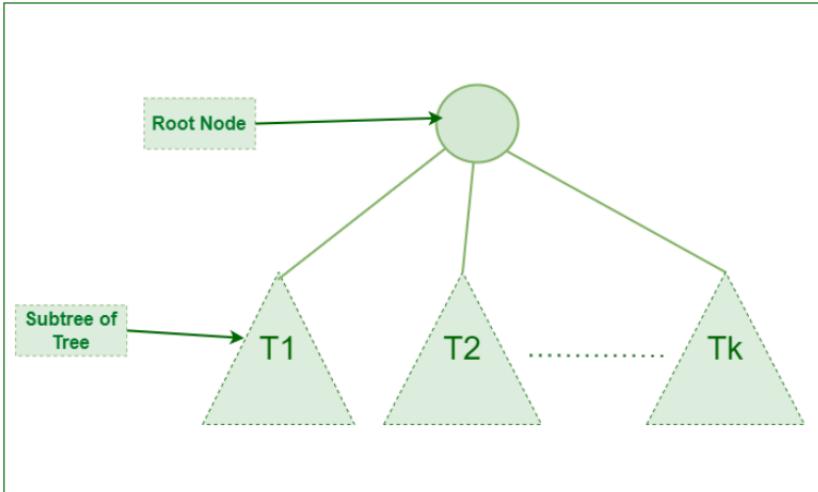
## Basic Terminologies In Tree Data Structure:

- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {B} is the parent node of {D, E}.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {D, E} are the child nodes of {B}.
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {K, L, M, N, O, P, G} are the leaf nodes of the tree.

- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {A,B} are the ancestor nodes of the node {E}
- **Descendant:** Any successor node on the path from the leaf node to that node. {E,I} are the descendants of the node {B}.
- **Sibling:** Children of the same parent node are called siblings. {D,E} are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. Level = (height of the root - height of the node). The root node has level 0.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree:** Any node of the tree along with its descendant.
- **Height:** Maximum number of edges from the node and leaf node.

- **Representation of Tree Data Structure:**

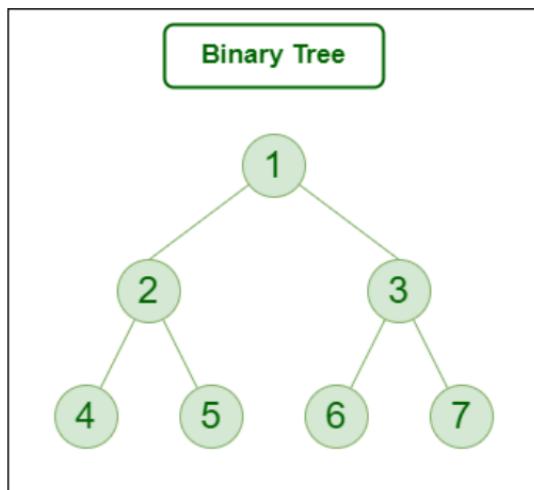
A tree consists of a root, and zero or more subtrees  $T_1, T_2, \dots, T_k$  such that there is an edge from the root of the tree to the root of each subtree.



- **Representation of a Node in Tree Data Structure:**

```
struct Node  
{  
    int data;  
    struct Node *first_child;  
    struct Node *second_child;  
    struct Node *third_child;  
    .  
    .  
    .  
    struct Node *nth_child;  
};
```

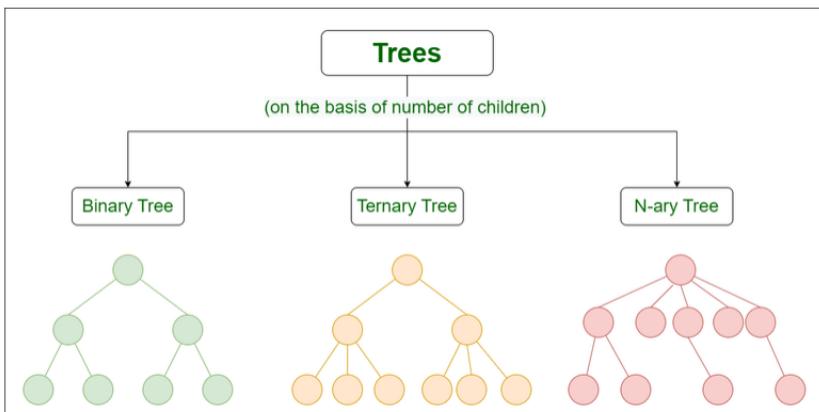
### **Example of Tree data structure:**



Here,

- Node 1 is the root node
- 1 is the parent of 2 and 3
- 2 and 3 are the siblings
- 4, 5, 6, and 7 are the leaf nodes
- 1 and 2 are the ancestors of 5

- **Types of Tree data structures:**



**1. Binary tree:** In a binary tree, each node can have a maximum of two children linked to it. Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees.

**2. Ternary Tree:** A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as "left", "mid" and "right".

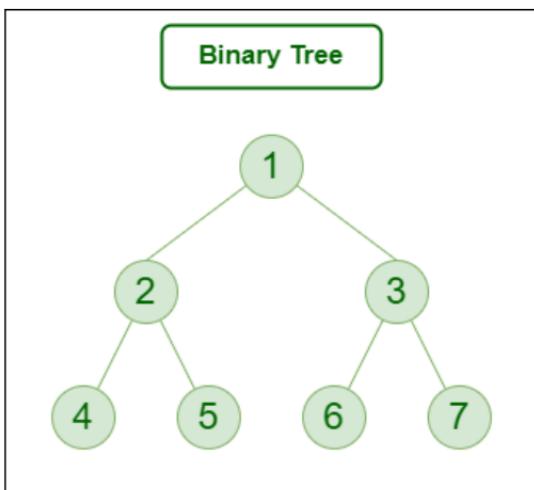
**3. N-ary Tree or Generic Tree:** Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

## **1. Binary tree:**

In a binary tree, each node can have a maximum of two children linked to it. Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees.

### **Example:**

Consider the tree below. Since each node of this tree has only 2 children, it can be said that this tree is a Binary Tree



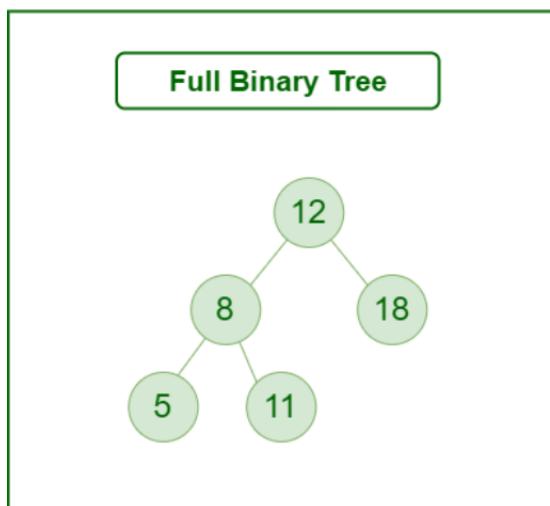
- **Types of Binary Tree:**

1. Full/ Strict Binary Tree
2. Complete Binary Tree
3. Perfect Binary Tree
4. Height Balanced Binary Tree
5. Skewed Binary Tree
6. Ordered Binary Tree

### **1. Full/ Strict Binary Tree:**

A full binary tree is a binary tree with either zero or two child nodes for each node.

A full binary tree, on the other hand, does not have any nodes that have only one child node.



Full Binary Tree Theorem:

Let  $T$  be a nonempty, full binary tree Then:

- If  $T$  has  $I$  internal nodes, the number of leaves is  $L = I + 1$ .

This is known as the full binary tree theorem.

Facts derived from the theorem:

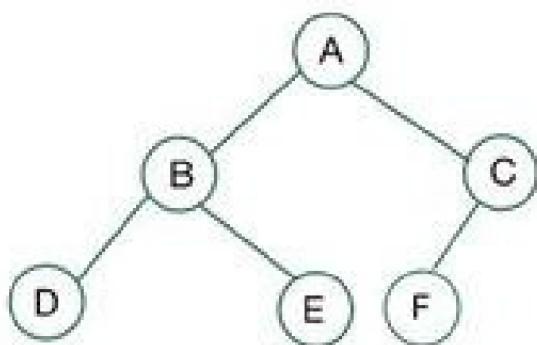
- If  $T$  has  $I$  internal nodes, the total number of nodes is  $N = 2I + 1$ .
- If  $T$  has a total of  $N$  nodes, the number of internal nodes is  $I = (N - 1)/2$ .
- If  $T$  has a total of  $N$  nodes, the number of leaves is  $L = (N + 1)/2$ .
- If  $T$  has  $L$  leaves, the total number of nodes is  $N = 2L - 1$ .
- If  $T$  has  $L$  leaves, the number of internal nodes is  $I = L - 1$ .

Some other properties:

- There are a maximum of  $2^k$  nodes in level  $k$  for every  $k \geq 0$ .
- The binary tree with  $\lambda$  levels has maximum of  $(2^\lambda) - 1$  nodes.
- The binary tree with  $N$  nodes has the number of levels at least  $[\log(N + 1)]$ .
- The binary tree with  $L$  leaves has the number of leaves at least  $[\log L] + 1$ .

## **2.Complete Binary Tree:**

A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled from left as possible.



## Some terminology of Complete Binary Tree:

- **Root** - Node in which no edge is coming from the parent. Example -node A
- **Child** - Node having some incoming edge is called child. Example – nodes B, F are the child of A and C respectively.
- **Sibling** - Nodes having the same parent are sibling. Example- D, E are siblings as they have the same parent B.
- **Degree of a node** - Number of children of a particular parent. Example- Degree of A is 2 and Degree of C is 1. Degree of D is 0.
- **Internal/External nodes** - Leaf nodes are external nodes and non leaf nodes are internal nodes.
- **Level** - Count nodes in a path to reach a destination node. Example- Level of node D is 2 as nodes A and B form the path.
- **Height** - Number of edges to reach the destination node, Root is at height 0. Example – Height of node E is 2 as it has two edges from the root.

## Properties of Complete Binary Tree:

- A complete binary tree is said to be a proper binary tree where all leaves have the same depth.
- In a complete binary tree number of nodes at depth  $d$  is  $2^d$ .
- In a complete binary tree with  $n$  nodes height of the tree is  $\log(n+1)$ .
- All the levels except the last level are completely full.

### **3.Perfect Binary Tree:**

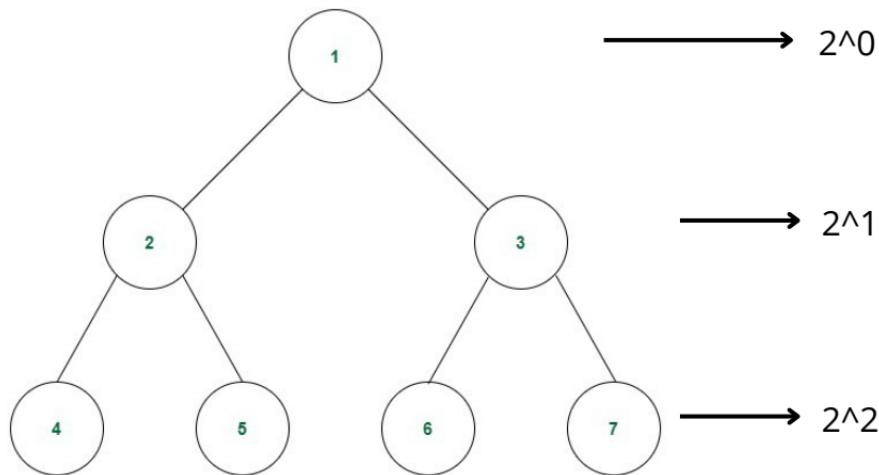
A perfect binary tree is a special type of binary tree in which all the leaf nodes are at the same depth, and all non-leaf nodes have two children. In simple terms, this means that all leaf nodes are at the maximum depth of the tree, and the tree is completely filled with no gaps.

The maximum number of nodes in a perfect binary tree is given by the formula  $2^{(d+1)} - 1$ , where d is the depth/ height of the tree. This means that a perfect binary tree with a depth of n has  $2^n$  leaf nodes and a total of  $2^{(n+1)} - 1$  nodes.

Perfect binary trees have a number of useful properties that make them useful in various applications. For example, they are often used in the implementation of heap data structures, as well as in the construction of threaded binary trees. They are also used in the implementation of algorithms such as heapsort and merge sort.

In other words, it can be said that each level of the tree is completely filled by the nodes.

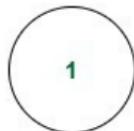
### Examples of Perfect Binary Tree:



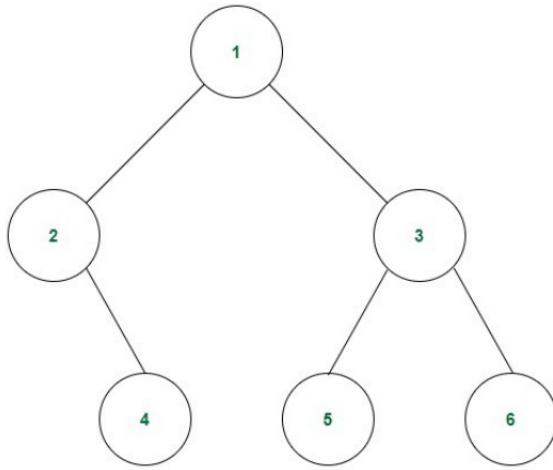
$$2^0 + 2^1 + 2^2 + \dots + 2^h = 2^{(h+1)} - 1$$

GP(Geometric Progression) series

A tree with only the root node is also a perfect binary tree.



The following tree is not a perfect binary tree because the last level of the tree is not completely filled.



## Properties of a Perfect Binary Tree:

- **Degree:** The degree of a node of a tree is defined as the number of children of that node. All the internal nodes have a degree of 2. The leaf nodes of a perfect binary tree have a degree of 0.
- **Number of leaf nodes:**  
Height =  $h$   
The number of leaf nodes =  $2^h$   
because the last level is completely filled.
- **Depth of a node:** Average depth of a node in a perfect binary tree is  $\Theta(\ln(n))$ .
- **Relation between leaf nodes and non-leaf nodes:** No. of leaf nodes = No. of non-leaf nodes +1.
- **Total number of nodes:**  
Height =  $h$   
Total nodes =  $2^{(h+1)} - 1$ .  
Each node of the tree is filled. So total number of nodes can be calculated as  $2^0 + 2^1 + \dots + 2^h = 2^{(h+1)} - 1$ .
- **Height of the tree:** The height of a perfect binary tree with  $N$  number of nodes =  $\log(N + 1) - 1 = \Theta(\ln(n))$ . This can be calculated using the relation shown while calculating the total number of nodes in a perfect binary tree.

Check whether a tree is a Perfect Binary Tree or not:

- **Check the depth of the tree:** A perfect binary tree is defined as a tree where all leaf nodes are at the same depth, and all non-leaf nodes have two children. To check whether a tree is a perfect binary tree, you can first calculate the depth of the tree.
- **Check the number of nodes at each level:** Once you have calculated the depth of the tree, you can then check the number of nodes at each level. In a perfect binary tree, the number of nodes at each level should be a power of 2 (e.g. 1, 2, 4, 8, etc.). If any level has a different number of nodes, the tree is not a perfect binary tree.

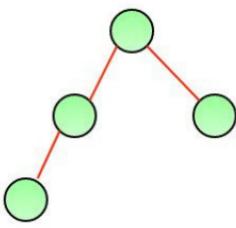
#### **4. Height Balanced Binary Tree:**

A height balanced binary tree is a binary tree in which the height of the left subtree and right subtree of any node does not differ by more than 1 and both the left and right subtree are also height balanced.

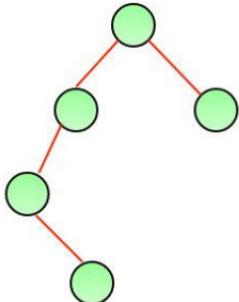
In this article, we will look into methods how to determine if given Binary trees are height-balanced

#### **Examples:**

The tree on the left is a height balanced binary tree. Whereas the tree on the right is not a height balanced tree. Because the left subtree of the root has a height which is 2 more than the height of the right subtree.



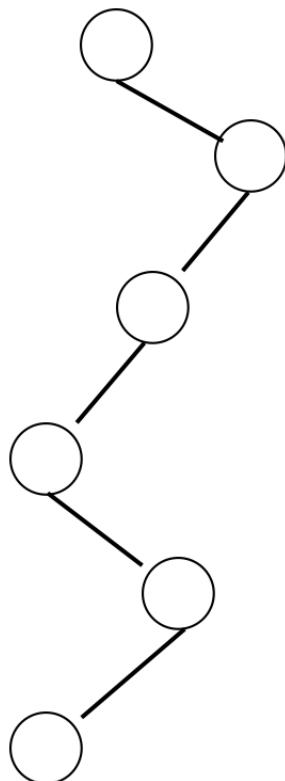
A height balanced tree



Not a height balanced tree

## 5. Skewed Binary Tree:

A skewed binary tree is a specific type of binary tree where each node has at most one child. In other words, it's a binary tree in which the left or right subtree of every node is empty. This results in a tree that is essentially a linked list, where each node has only one child.



The skewness of the tree depends on whether the left or right subtree is consistently chosen. If every node has a left child, it's a left-skewed binary tree; if every node has a right child, it's a right-skewed binary tree.

These skewed binary trees don't provide the advantages of a balanced binary tree in terms of search and retrieval operations, as the time complexity can degrade to  $O(n)$  in the worst case (where  $n$  is the number of nodes) instead of the more efficient  $O(\log n)$  of a balanced binary tree.

## **6.Ordered Binary Tree:**

An ordered binary tree, also known as a binary search tree (BST), is a type of binary tree with the following properties:

**1. Value Ordering:** Each node in the tree has a value, and for every node:

- All values in its left subtree are less than its own value.
- All values in its right subtree are greater than its own value.

**2. Unique Values:** No two nodes in the tree have the same value.

- These properties make searching, insertion, and deletion operations efficient, as they can be performed in  $O(\log n)$  time on average, where  $n$  is the number of nodes in the tree.
- The ordered nature of the tree allows for quick searches by narrowing down the search space based on the comparison of values. It's a fundamental data structure used in many applications where efficient search is a critical requirement.

- **Uses of Binary Trees:**

1. File systems
2. Databases
3. Algorithms/ Networking
4. Maths
5. Machine Learning Algorithm
6. Future Data Structure (Graphs, Heap)

## **Basic Operation Of Tree Data Structure:**

- **Create** - create a tree in the data structure.
- **Insert** - Inserts data in a tree.
- **Search** – Searches specific data in a tree to check whether it is present or not.
- **Traversal:**
  1. **Preorder Traversal** – perform Traveling a tree in a pre-order manner in the data structure.
  2. **In order Traversal** – perform Traveling a tree in an in-order manner.
  3. **Post-order Traversal** – perform Traveling a tree in a post-order manner.

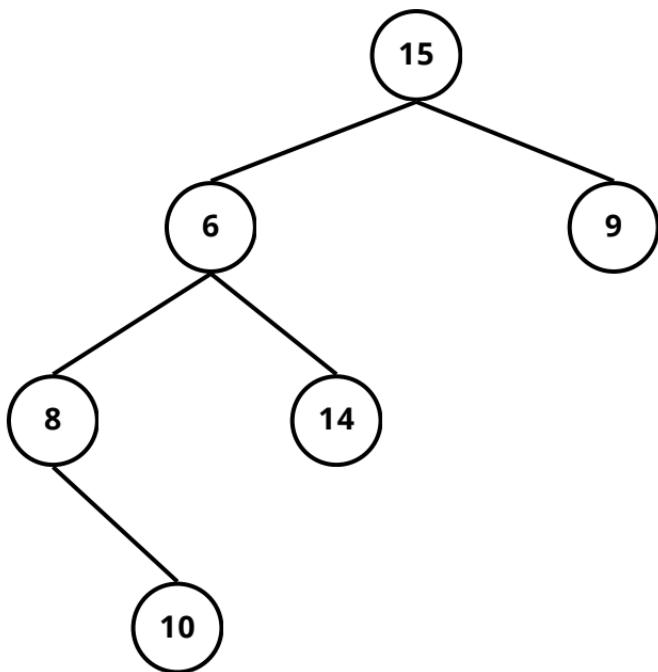
## **Implementation:**

**There are two ways to implement Trees:**

1. LinkedList representation
2. Sequential representation => Using Array

## 1. LinkedList representation of Tree:

- Make a program that will implement a Binary Tree and show the output of it.



```
import java.util.Scanner;

class BinaryTree {

    public BinaryTree() {

    }

    private static class Node {
        int value;
        Node left;
        Node right;

        public Node(int value) {
            this.value = value;
        }
    }

    private Node root;

    // insert elements
    public void populate(Scanner scanner) {
        System.out.println("Enter the root Node: ");
        int value = scanner.nextInt();
        root = new Node(value);
        populate(scanner, root);
    }

    private void populate(Scanner scanner, Node node) {
        System.out.println("Do you want to enter left of "
+ node.value);
```

```
boolean left = scanner.nextBoolean();
if (left) {
    System.out.println("Enter the value of the left of " +
node.value);
    int value = scanner.nextInt();
    node.left = new Node(value);
    populate(scanner, node.left);
}

System.out.println("Do you want to enter right of " +
node.value);
boolean right = scanner.nextBoolean();
if (right) {
    System.out.println("Enter the value of the right of " +
node.value);
    int value = scanner.nextInt();
    node.right = new Node(value);
    populate(scanner, node.right);
}

// display tree's elements
public void prettyDisplay() {
    prettyDisplay(root, 0);
}

private void prettyDisplay(Node node, int level) {
    if (node == null) {
        return;
    }
```

```
prettyDisplay(node.right, level + 1);

if (level != 0) {
    for (int i = 0; i < level - 1; i++) {
        System.out.print("|\t\t");
    }
    System.out.println("----->" + node.value);
} else {
    System.out.println(node.value);
}
prettyDisplay(node.left, level + 1);
}
```

```
public void preOrder() {
    preOrder(root);
}
```

```
private void preOrder(Node node) {
    if (node == null) {
        return;
    }
    System.out.print(node.value + " ");
    preOrder(node.left);
    preOrder(node.right);
}
```

```
public void inOrder() {
    preOrder(root);
}
```

```
private void inOrder(Node node) {  
    if (node == null) {  
        return;  
    }  
    preOrder(node.left);  
    System.out.print(node.value + " ");  
    preOrder(node.right);  
}  
  
public void postOrder() {  
    preOrder(root);  
}  
  
private void postOrder(Node node) {  
    if (node == null) {  
        return;  
    }  
    preOrder(node.left);  
    preOrder(node.right);  
    System.out.print(node.value + " ");  
}  
  
}  
  
public class Test {  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        BinaryTree tree = new BinaryTree();  
        tree.populate(scanner);  
        tree.prettyDisplay();  
    }  
}
```

## **Output:**

Enter the root Node:

15

Do you want to enter left of 15

true

Enter the value of the left of 15

6

Do you want to enter left of 6

true

Enter the value of the left of 6

8

Do you want to enter left of 8

false

Do you want to enter right of 8

true

Enter the value of the right of 8

10

Do you want to enter left of 10

false

Do you want to enter right of 10

false

Do you want to enter right of 6

true

Enter the value of the right of 6

14

Do you want to enter left of 14

false

Do you want to enter right of 14

false

Do you want to enter right of 15

true

Enter the value of the right of 15

9

Do you want to enter left of 9

false

Do you want to enter right of 9

false

|----->9

15

|----->14

|----->6

|----->10

|----->8

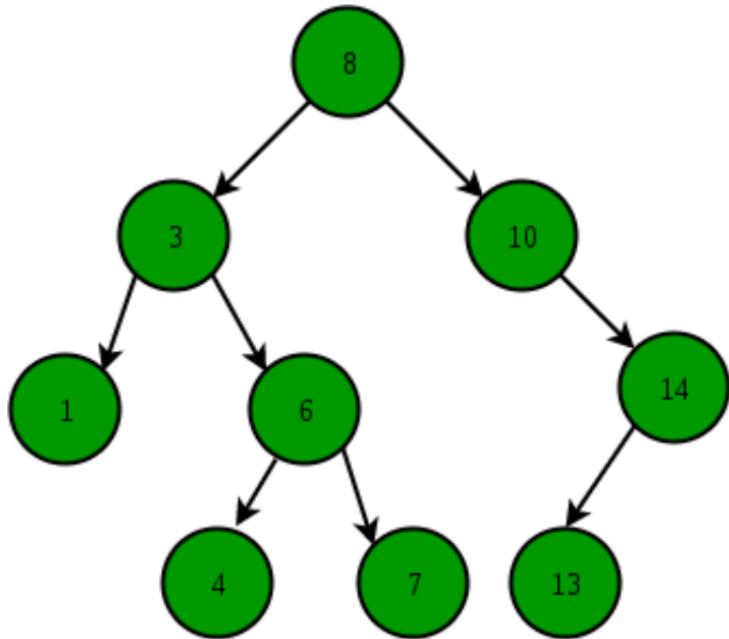
- **Binary Search Tree:**

A Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node's value and the right child has a value greater than the node's value. This property is called the BST property and it makes it possible to efficiently search, insert, and delete elements in the tree.

The root of a BST is the node that has the smallest value in the left subtree and the largest value in the right subtree. Each left subtree is a BST with nodes that have smaller values than the root and each right subtree is a BST with nodes that have larger values than the root.

Binary Search Tree is a node-based binary tree data structure that has the following properties:

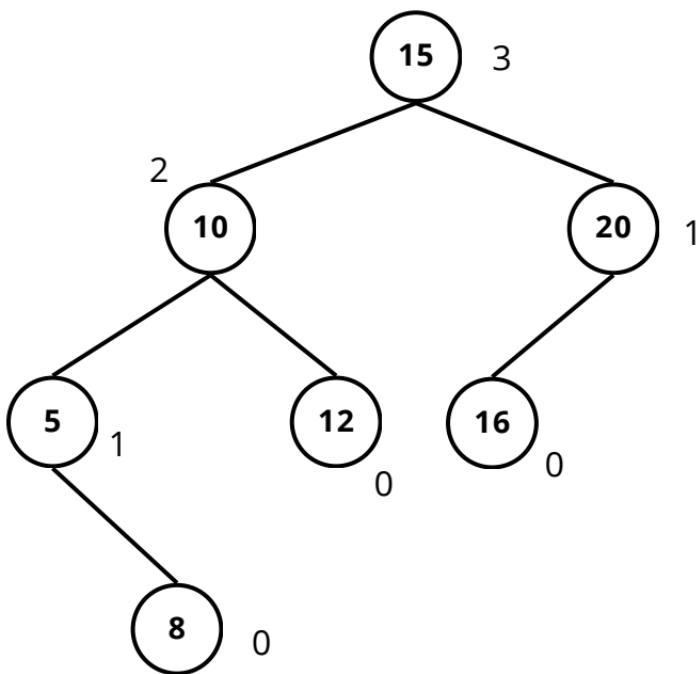
- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- This means everything to the left of the root is less than the value of the root and everything to the right of the root is greater than the value of the root. Due to this performing, a binary search is very easy.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes(BST may have duplicate values with different handling approaches)



Handling approach for Duplicate values in the Binary Search tree:

- You can not allow the duplicated values at all.
- We must follow a consistent process throughout i.e either store duplicate value at the left or store the duplicate value at the right of the root, but be consistent with your approach.
- We can keep the counter with the node and if we found the duplicate value, then we can increment the counter

- Make a program that will implement Binary Search Tree and show the output of it.



```
class BST {  
    public class Node {  
        private int value;  
        private Node left;  
        private Node right;  
        private int height;  
  
        public Node(int value) {  
            this.value = value;  
        }  
  
        public int getValue() {  
            return value;  
        }  
    }  
  
    private Node root;  
  
    public BST() {  
    }  
  
    public int height(Node node) {  
        if (node == null) {  
            return -1;  
        }  
        return node.height;  
    }  
  
    public boolean isEmpty() {  
        return root == null;  
    }  
}
```

```
public void insert(int value) {  
    root = insert(value, root);  
}  
  
private Node insert(int value, Node node) {  
    if (node == null) {  
        node = new Node(value);  
        return node;  
    }  
  
    if (value < node.value) {  
        node.left = insert(value, node.left);  
    }  
  
    if (value > node.value) {  
        node.right = insert(value, node.right);  
    }  
  
    node.height = Math.max(height(node.left),  
                           height(node.right)) + 1;  
    return node;  
}  
  
public void populate(int[] nums) {  
    for (int i = 0; i < nums.length; i++) {  
        this.insert(nums[i]);  
    }  
}  
  
public void populatedSorted(int[] nums) {  
    populatedSorted(nums, 0, nums.length);  
}
```

```
// When array is sorted
private void populatedSorted(int[] nums, int start,
int end) {
    if (start >= end) {
        return;
    }

    int mid = (start + end) / 2;

    this.insert(nums[mid]);
    populatedSorted(nums, start, mid);
    populatedSorted(nums, mid + 1, end);
}

public boolean balanced() {
    return balanced(root);
}

private boolean balanced(Node node) {
    if (node == null) {
        return true;
    }
    return Math.abs(height(node.left) -
height(node.right)) <= 1 && balanced(node.left) &&
balanced(node.right);
}

public void display() {
    display(this.root, "Root Node: ");
}
```

```
private void display(Node node, String details) {  
    if (node == null) {  
        return;  
    }  
    System.out.println(details + node.value);  
    display(node.left, "Left child of " + node.value + " : ");  
    display(node.right, "Right child of " + node.value + " : ");  
}  
  
}  
  
public class Test {  
    public static void main(String[] args) {  
        BST tree = new BST();  
        int[] nums = { 15, 10, 20, 12, 5, 16, 8 };  
        tree.populate(nums);  
        tree.display();  
  
        BST tree1 = new BST();  
        int[] nums1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
        tree1.populatedSorted(nums1);  
        tree1.display();  
    }  
}
```

**Output:**

Root Node: 15

Left child of 15 : 10

Left child of 10 : 5

Right child of 5 : 8

Right child of 10 : 12

Right child of 15 : 20

Left child of 20 : 16

Root Node: 6

Left child of 6 : 3

Left child of 3 : 2

Left child of 2 : 1

Right child of 3 : 5

Left child of 5 : 4

Right child of 6 : 9

Left child of 9 : 8

Left child of 8 : 7

Right child of 9 : 10

- **Tree Traversal methods:**

- 1.BFS (Breath First Search) -----> Level-Order-Traversal
- 2.DFS (Depth First Search)

### **1. BFS (Breath First Search):**

Breadth-First Search (BFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node and explores the neighbor nodes at the present depth prior to moving on to nodes at the next depth level. It visits nodes level by level.

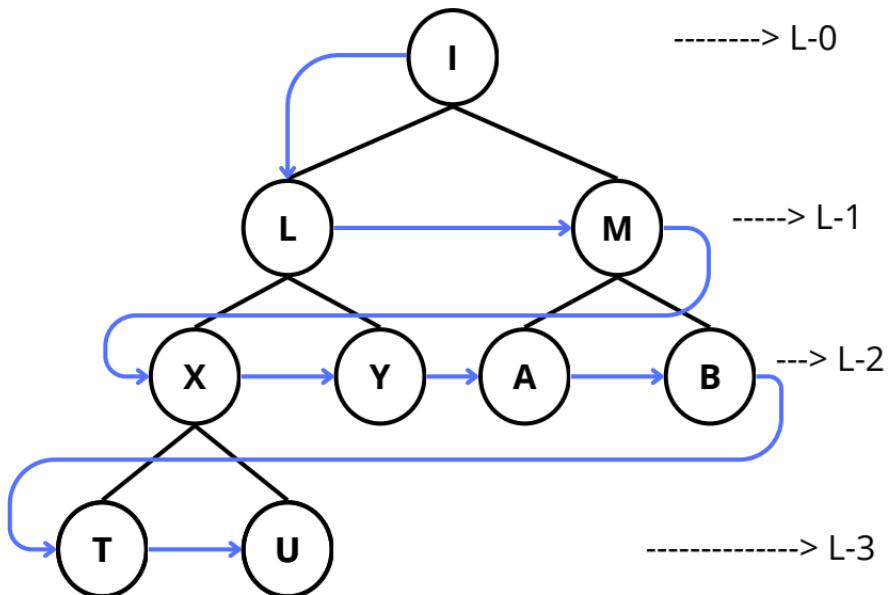
Here's a basic outline of how BFS works:

- Start at the Root: Begin at the root node of the tree or the starting node of the graph.
- Visit the Node: Process (or "visit") the current node.
- Explore Neighbors: Explore all the neighbors (adjacent nodes) of the current node. These neighbors are at the same depth level.
- Move to the Next Level: Move to the next level of the tree or graph and repeat steps 2-3 for the nodes at that level.
- Continue Until Completion: Continue this process until all nodes have been visited.

BFS uses a queue data structure to keep track of which nodes to visit next. The order in which nodes are added to and removed from the queue ensures that nodes are processed in a breadth-first manner.

BFS is often used to find the shortest path between two nodes in an unweighted graph, as it guarantees that the first time a node is encountered during the search, it is via the shortest possible path.

Here's a simple example of a tree traversal using BFS:



### **Output:**

Level-Order:

I->L->M->X->Y->A->B->T->U

## **2. DFS (Depth First Search):**

Depth-First Search (DFS) is another algorithm for traversing or searching tree or graph data structures. Unlike Breadth-First Search (BFS), DFS explores as far as possible along each branch before backtracking. It goes as deep as possible before exploring siblings.

### **Types of DFS:**

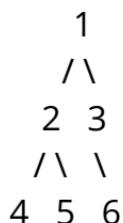
1. Pre-order Traversal
2. In-order Traversal
3. Post-order Traversal

Here's a basic outline of how DFS works:

- Start at the Root: Begin at the root node of the tree or the starting node of the graph.
- Visit the Node: Process (or "visit") the current node.
- Explore Neighbors: Recursively explore all the neighbors (adjacent nodes) of the current node. This involves going as deep as possible along each branch before backtracking.
- Backtrack: After reaching the end of a branch, backtrack to the previous node and explore its other neighbors.
- Continue Until Completion: Repeat steps 2-4 until all nodes have been visited.

DFS can be implemented using either recursion or an explicit stack data structure. When implemented with recursion, the call stack effectively takes the place of an explicit stack.

Here's a simple example of a tree traversal using DFS:



Starting from the root (1), a possible DFS traversal would be: 1, 2, 4, 5, 3, 6.

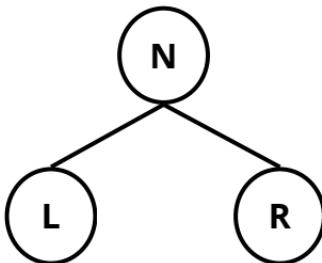
The order in which nodes are visited depends on the specific implementation and can vary. DFS has different variants, such as pre-order, in-order, and post-order traversals, which specify when the current node is visited concerning its children during the recursive exploration.

DFS is often used in problems that involve searching through possibilities and can be employed in various applications, such as maze-solving, topological sorting, and graph component analysis. However, it may not necessarily find the shortest path between two nodes in a graph, unlike BFS.

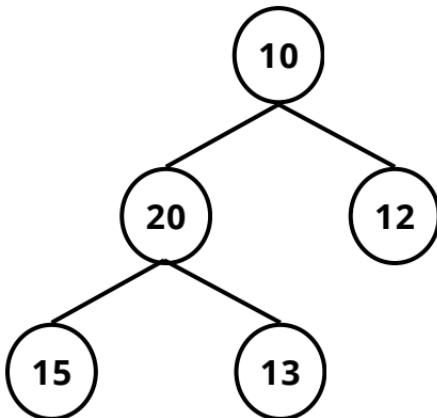
## 1. Pre-order Traversal:

It will display like,

Node (N) ---> Left (L) ---> Right (R)



N ---> L ---> R



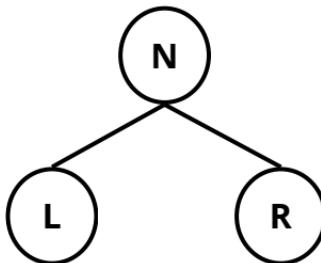
10 ---> 20 ---> 15 --> 13 --> 12

- Used for evaluating mathematical expressions or making a copy
- Serialization your array

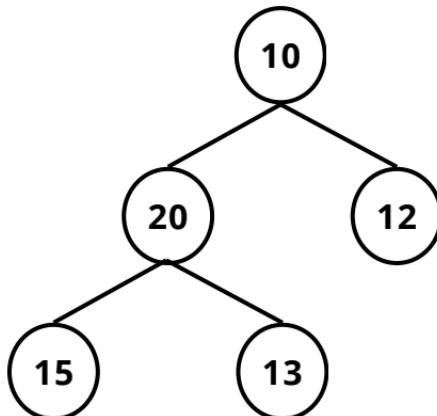
## 2. In-order Traversal:

It will display like,

Left (L) ---> Node (N) ---> Right (R)



L ---> N ---> R



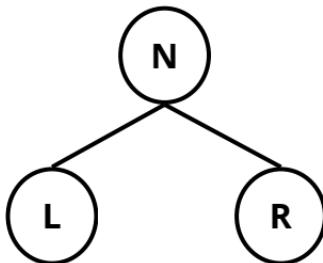
15 ---> 20 ---> 13 --> 10 --> 12

- In BST, we can visit node in sorted manner

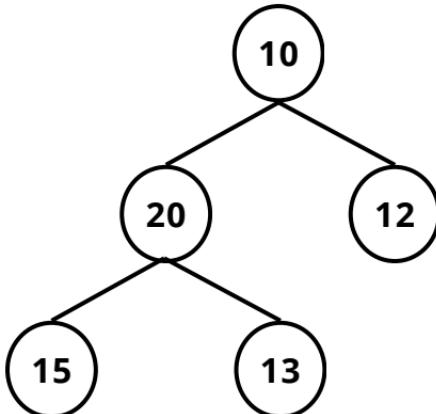
### 3. Post-order Traversal:

It will display like,

Left (L) ---> Right (R) ---> Node (N)



L ---> R ---> N



15 ---> 13 ---> 20 ---> 12 ---> 10

- Deleting binary tree
- Bottom-up calculation

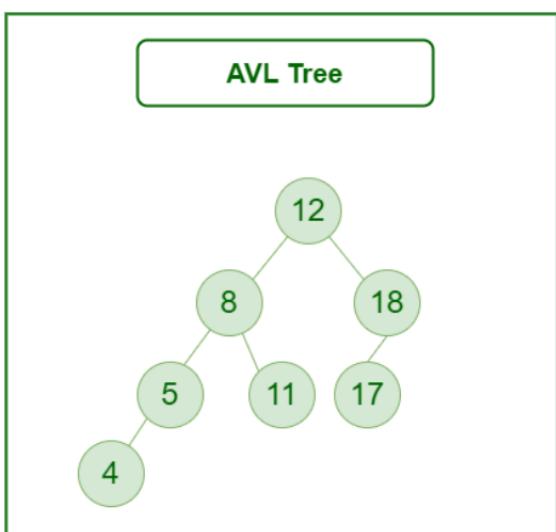
- **AVL Tree:**

An AVL tree defined as a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.

The difference between the heights of the left subtree and the right subtree for any node is known as the balance factor of the node.

The AVL tree is named after its inventors, Georgy Adelson-Velsky and Evgenii Landis, who published it in their 1962 paper “An algorithm for the organization of information”.

Example of AVL Trees:



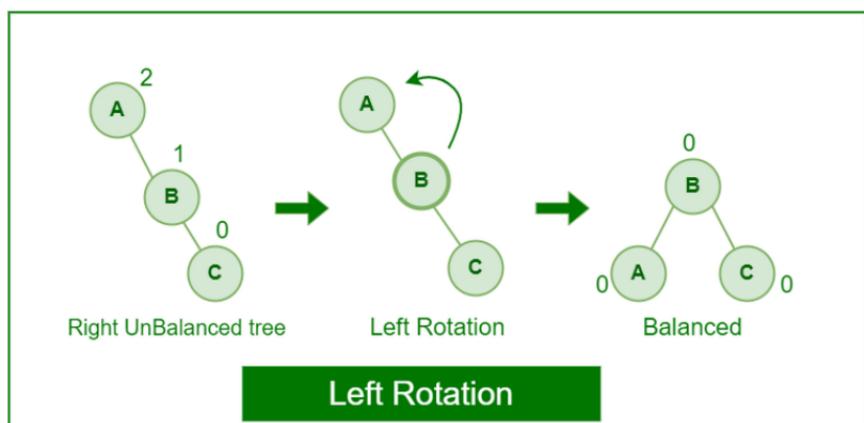
The above tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1. (-1 or +1 or 0).

- **Rotating the subtrees in an AVL Tree:**

An AVL tree may rotate in one of the following four ways to keep itself balanced:

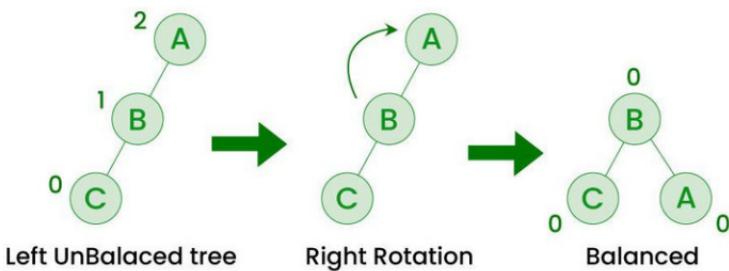
### **Left Rotation:**

When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation.



## Right Rotation:

If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.

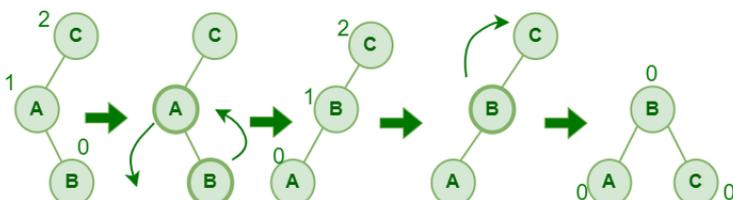


AVL Tree

∞

## Left-Right Rotation:

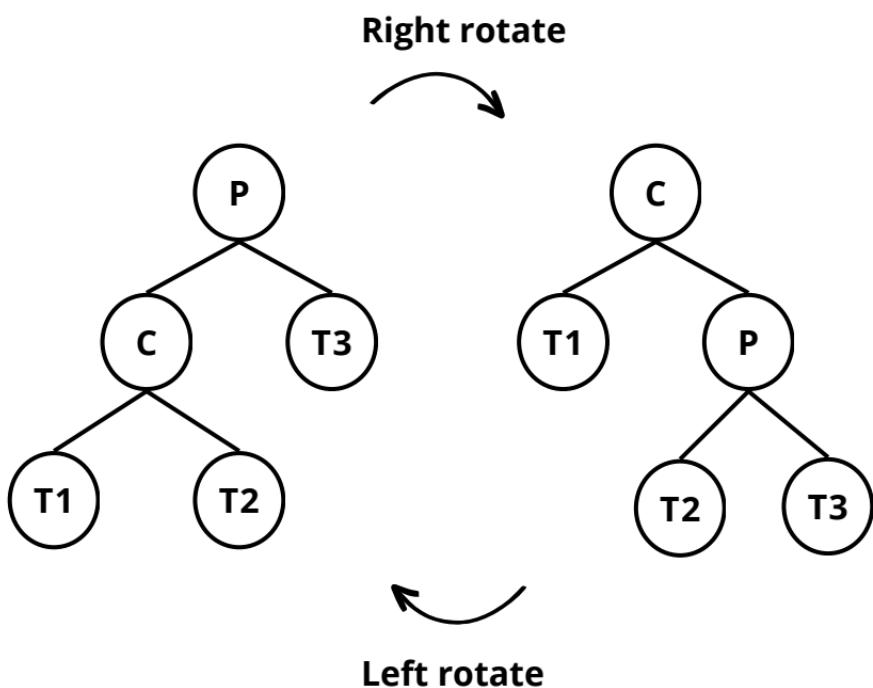
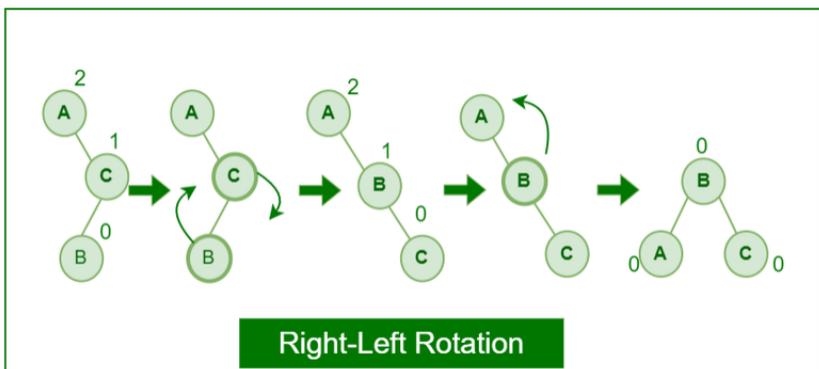
A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.



Left-Right Rotation

## Right-Left Rotation:

A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.



## **Advantages of AVL Tree:**

1. AVL trees can self-balance themselves.
2. It is surely not skewed.
3. It provides faster lookups than Red-Black Trees
4. Better searching time complexity compared to other trees like binary tree.
5. Height cannot exceed  $\log(N)$ , where, N is the total number of nodes in the tree.

## **Disadvantages of AVL Tree:**

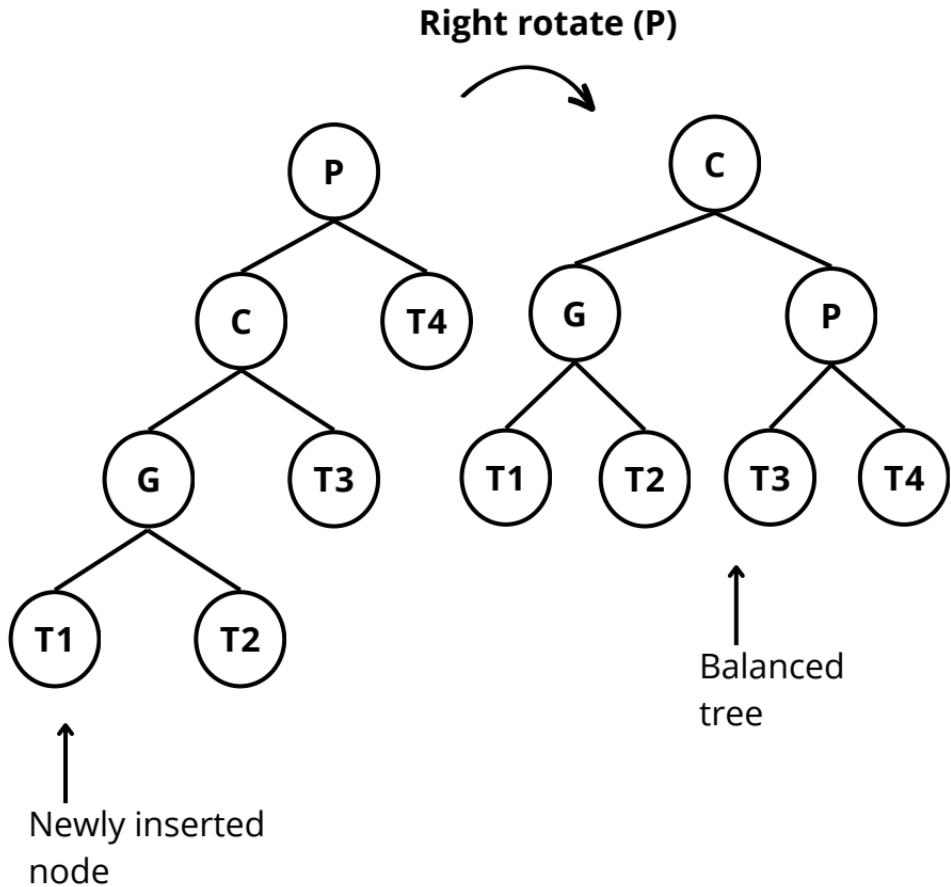
1. It is difficult to implement.
2. It has high constant factors for some of the operations.
3. Less used compared to Red-Black trees.
4. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.
5. Take more processing for balancing.

## **Applications of AVL Tree:**

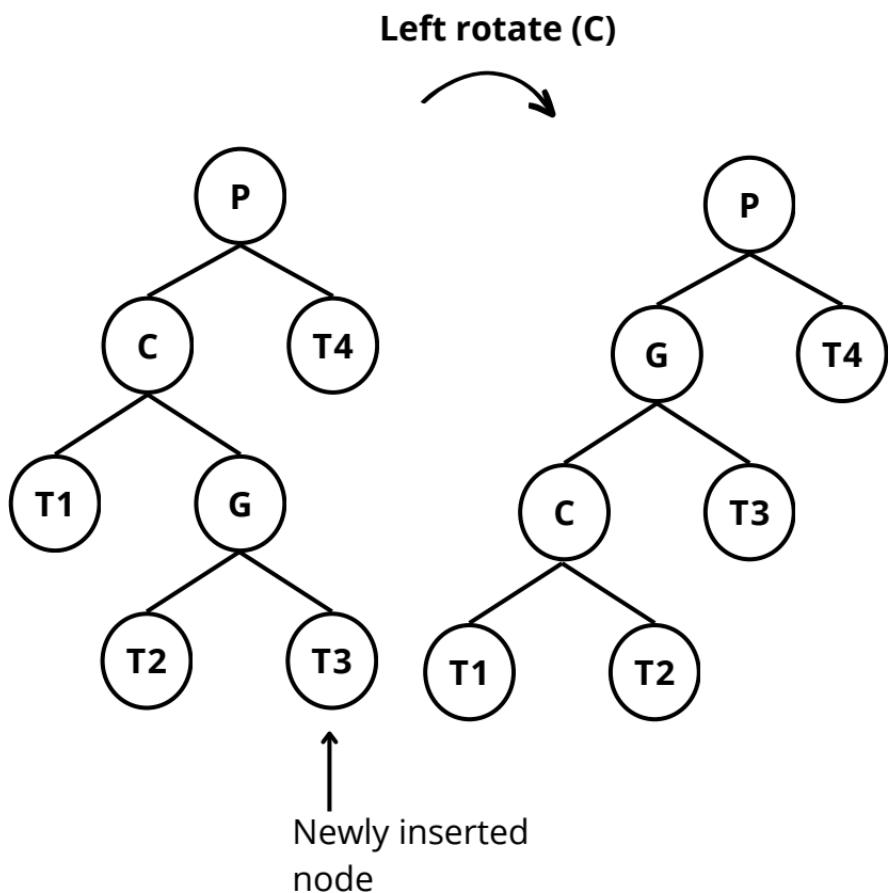
1. It is used to index huge records in a database and also to efficiently search in that.
2. For all types of in-memory collections, including sets and dictionaries, AVL Trees are used.
3. Database applications, where insertions and deletions are less common but frequent data lookups are necessary
4. Software that needs optimized search.
5. It is applied in corporate areas and storyline games.

**4 rules:**

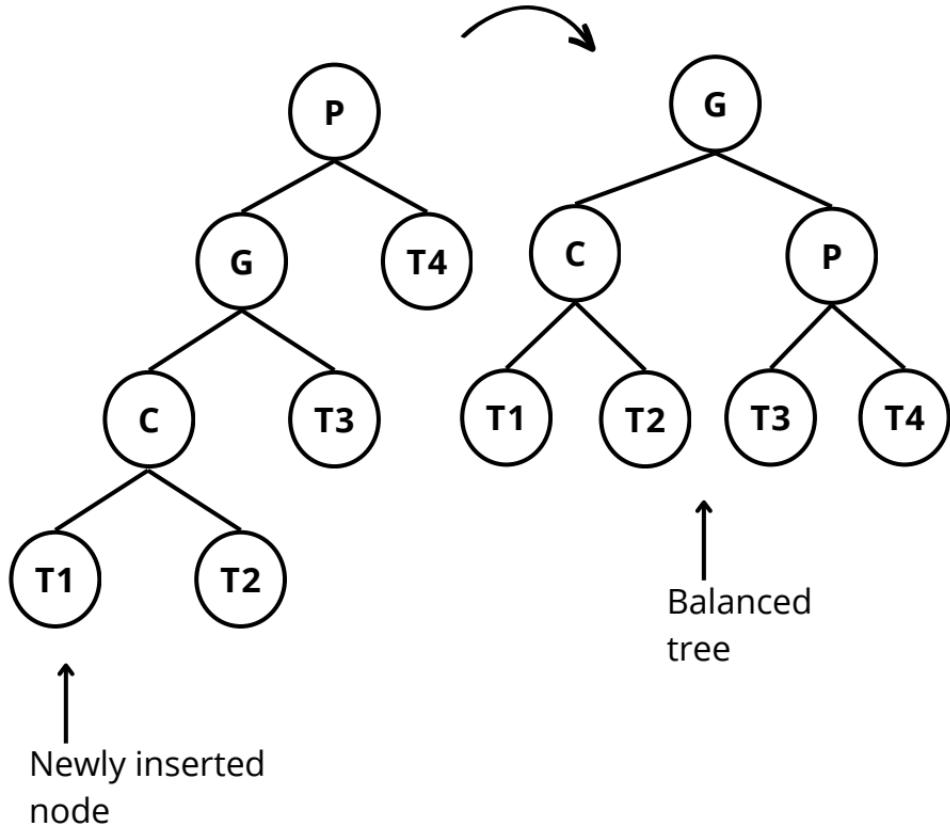
**1. Left- Left case:**



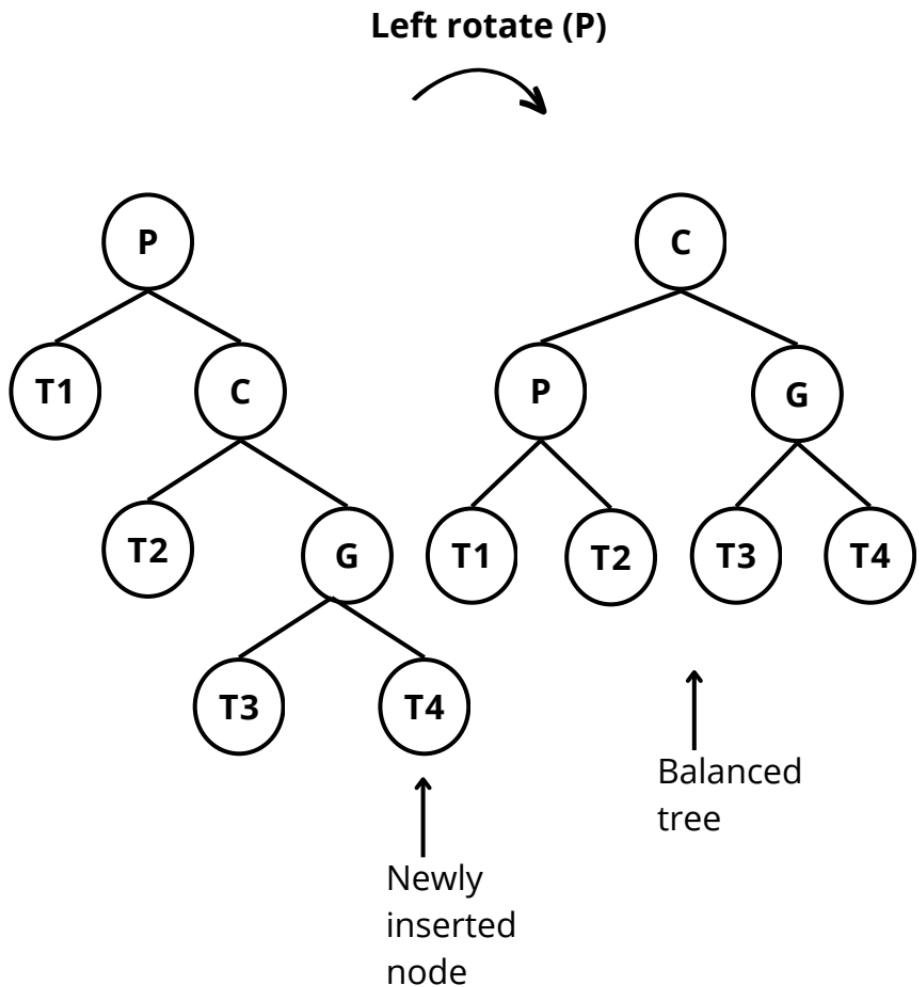
## 2. Left- Right case:



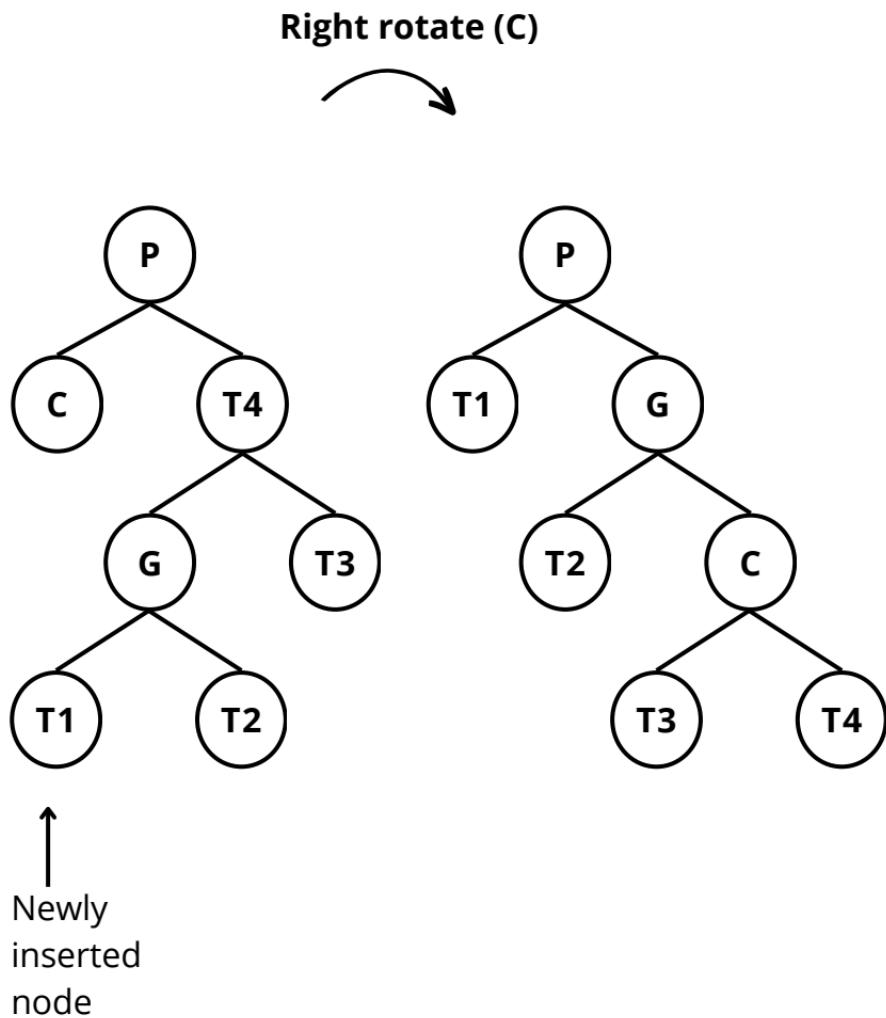
### Right rotate (P)



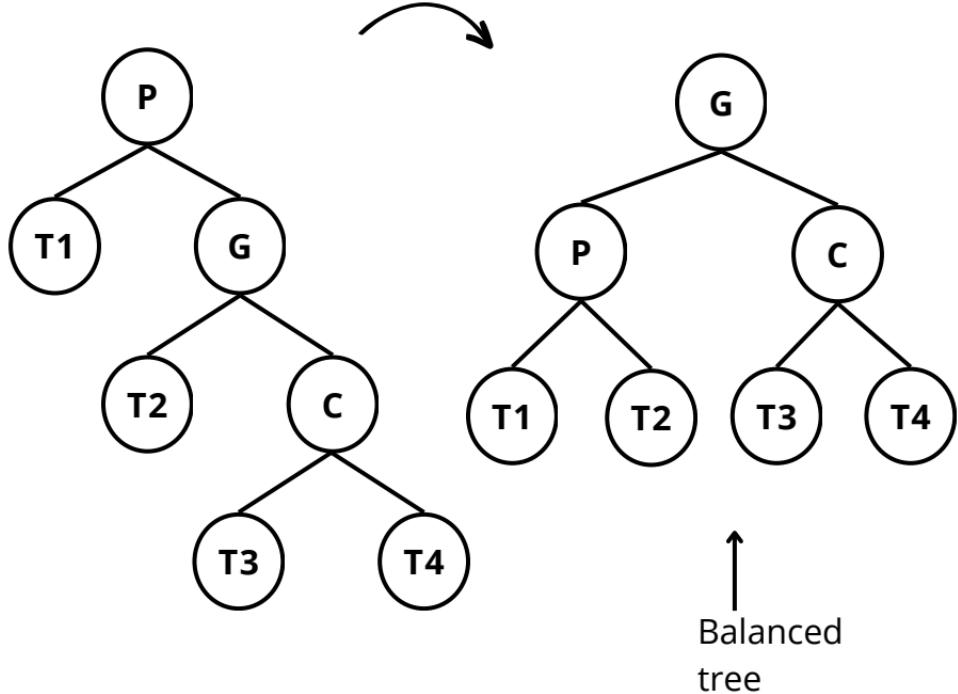
### 3. Right- Right case:



#### 4. Right- Left case:



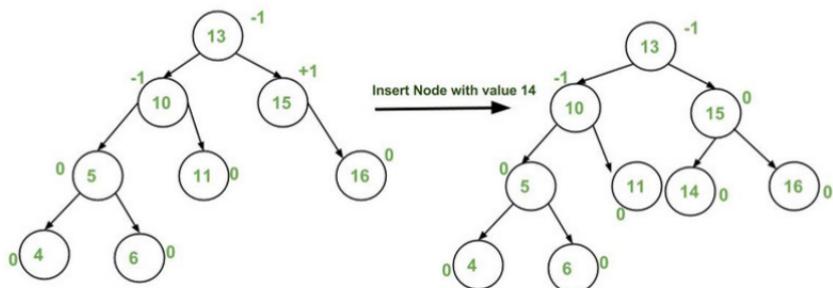
**Left rotate (P)**



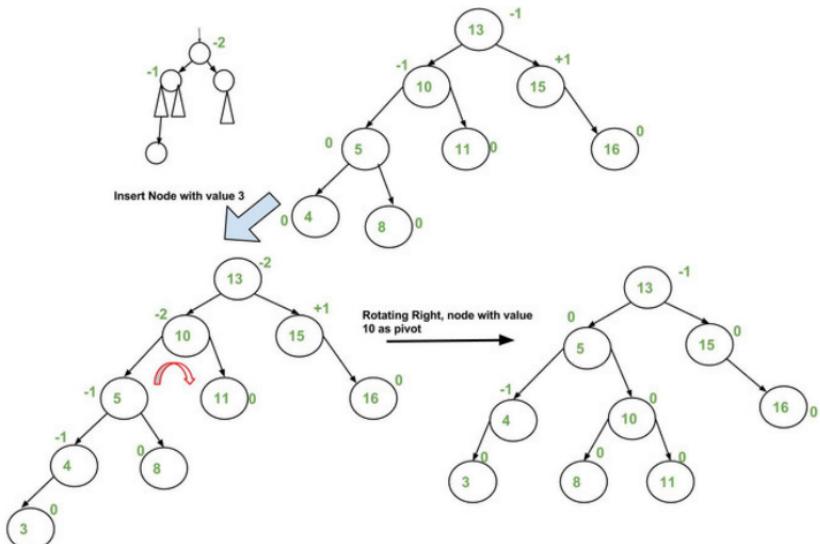
## Example:

### Illustration of Insertion at AVL Tree

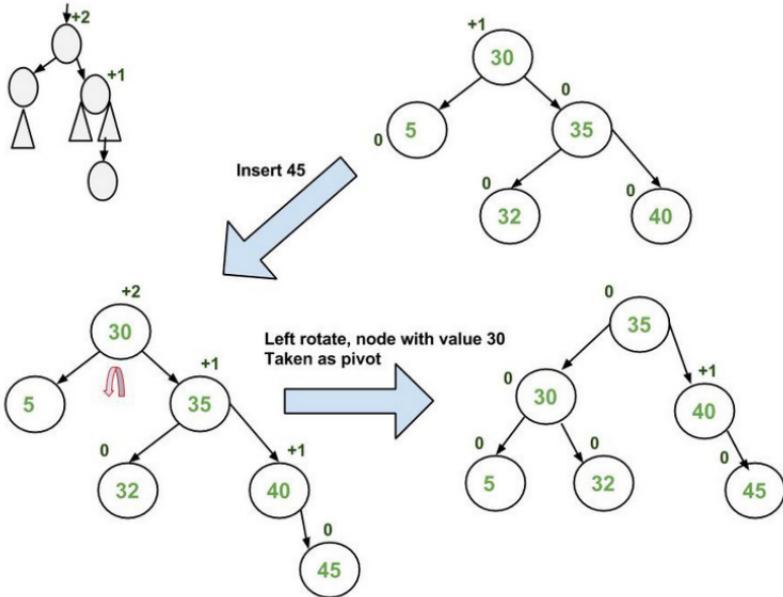
=>



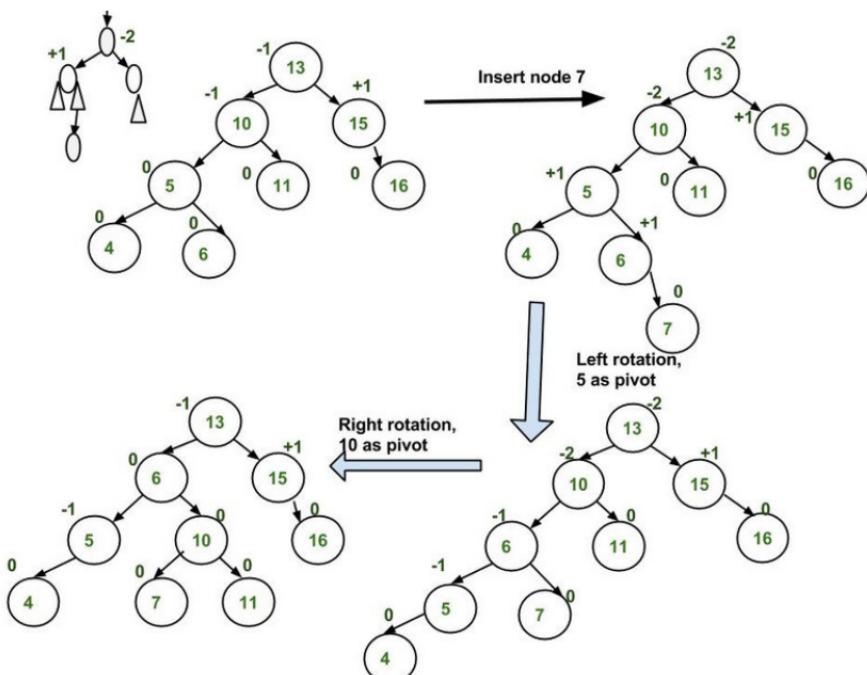
=>



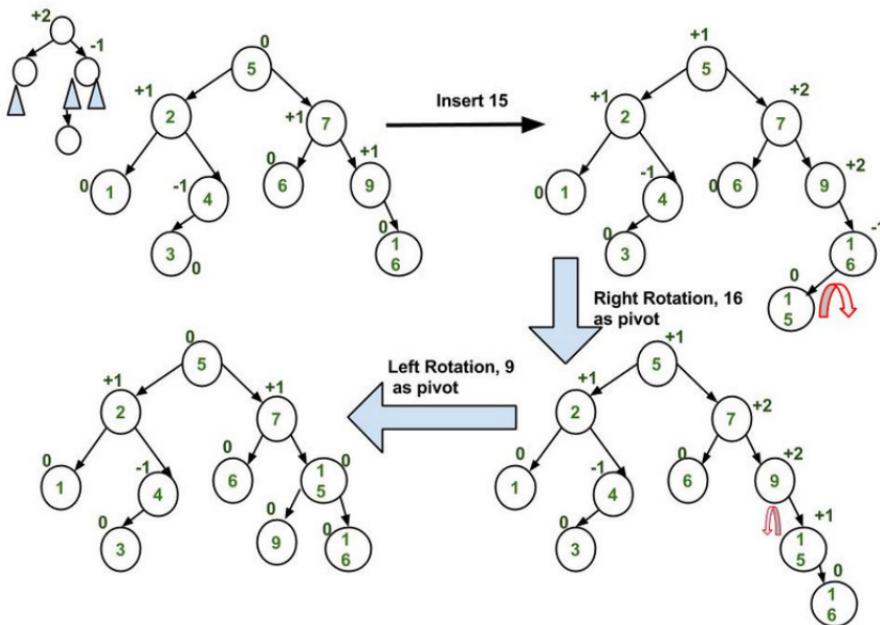
=>



=>



=>



## **Operations on an AVL Tree:**

- Insertion
- Deletion
- Searching [It is similar to performing a search in BST]

## **Algorithm of AVL Tree:**

1. Insert normally node (n)
2. Start from node (n) and find the node that makes the tree unbalanced, bottom-up.
3. Using one of the 4 rules; rotate.

## **Time complex of AVL Tree:**

$O(\log(n))$

- **Make a AVL tree program.**

```
class AVL {  
  
    public class Node {  
        private int value;  
        private Node left;  
        private Node right;  
        private int height;  
  
        public Node(int value) {  
            this.value = value;  
        }  
  
        public int getValue() {  
            return value;  
        }  
    }  
  
    private Node root;  
  
    public AVL() {  
    }  
  
    public int height() {  
        return height(root);  
    }  
    private int height(Node node) {  
        if (node == null) {  
            return -1;  
        }
```

```
        }

    return node.height;
}

public void insert(int value) {
    root = insert(value, root);
}

private Node insert(int value, Node node) {
    if (node == null) {
        node = new Node(value);
        return node;
    }

    if (value < node.value) {
        node.left = insert(value, node.left);
    }

    if (value > node.value) {
        node.right = insert(value, node.right);
    }

    node.height = Math.max(height(node.left),
height(node.right)) + 1;
    return rotate(node);
}

private Node rotate(Node node) {
    if (height(node.left) - height(node.right) > 1) {
        // left heavy
        if(height(node.left.left) - height(node.left.right) >
0) {
```

```
// left left case
    return rightRotate(node);
}
if(height(node.left.left) - height(node.left.right) < 0) {
    // left right case
    node.left = leftRotate(node.left);
    return rightRotate(node);
}

if (height(node.left) - height(node.right) < -1) {
    // right heavy
    if(height(node.right.left) - height(node.right.right) < 0) {
        // right right case
        return leftRotate(node);
    }
    if(height(node.right.left) - height(node.right.right) > 0) {
        // left right case
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }
}

return node;
}

public Node rightRotate(Node p) {
    Node c = p.left;
    Node t = c.right;

    c.right = p;
```

```
p.left = t;  
  
p.height = Math.max(height(p.left), height(p.right) + 1);  
c.height = Math.max(height(c.left), height(c.right) + 1);  
  
return c;  
}  
  
public Node leftRotate(Node c) {  
    Node p = c.right;  
    Node t = p.left;  
  
    p.left = c;  
    c.right = t;  
  
    c.height = Math.max(height(c.left), height(c.right) + 1);  
    p.height = Math.max(height(p.left), height(p.right) + 1);  
  
    return p;  
}  
  
public void populate(int[] nums) {  
    for (int i = 0; i < nums.length; i++) {  
        this.insert(nums[i]);  
    }  
}  
  
public void populatedSorted(int[] nums) {  
    populatedSorted(nums, 0, nums.length);  
}
```

```
private void populatedSorted(int[] nums, int start, int end) {
    if (start >= end) {
        return;
    }

    int mid = (start + end) / 2;

    this.insert(nums[mid]);
    populatedSorted(nums, start, mid);
    populatedSorted(nums, mid + 1, end);
}

public void display() {
    display(this.root, "Root Node: ");
}

private void display(Node node, String details) {
    if (node == null) {
        return;
    }
    System.out.println(details + node.value);
    display(node.left, "Left child of " + node.value + " : ");
    display(node.right, "Right child of " + node.value + " : ");
}

public boolean isEmpty() {
    return root == null;
}
```

```
public boolean balanced() {  
    return balanced(root);  
}  
  
private boolean balanced(Node node) {  
    if (node == null) {  
        return true;  
    }  
    return Math.abs(height(node.left) -  
height(node.right)) <= 1 && balanced(node.left) &&  
balanced(node.right);  
}  
  
}  
  
class Test {  
    public static void main(String[] args) {  
        AVL tree = new AVL();  
  
        for(int i=0; i < 1000; i++) {  
            tree.insert(i);  
        }  
  
        System.out.println(tree.height());  
    }  
}
```

## **Output:**

9

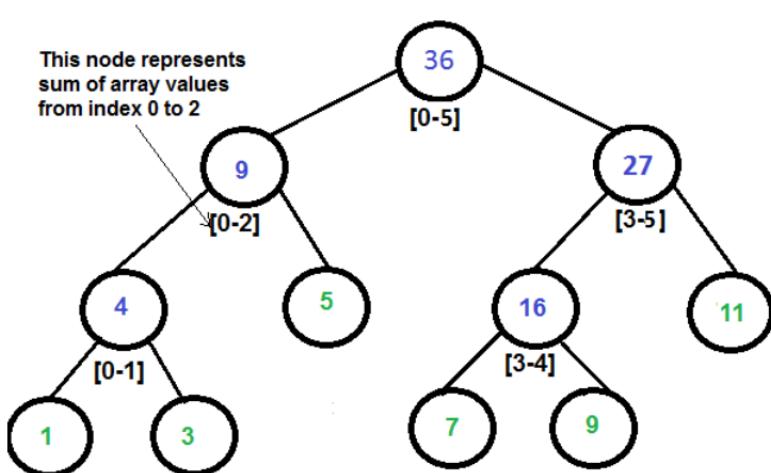
- **Segment Tree:**

In computer science, a Segment Tree, also known as a statistic tree, is a tree data structure used for storing information about intervals, or segments. It allows querying which of the stored segments contain a given point. It is, in principle, a static structure; that is, it's a structure that cannot be modified once it's built. A similar data structure is the interval tree.

A segment tree for a set  $I$  of  $n$  intervals uses  $O(n \log n)$  storage and can be built in  $O(n \log n)$  time.

Segment trees support searching for all the intervals that contain a query point in time  $O(\log n + k)$ ,  $k$  being the number of retrieved intervals or segments.

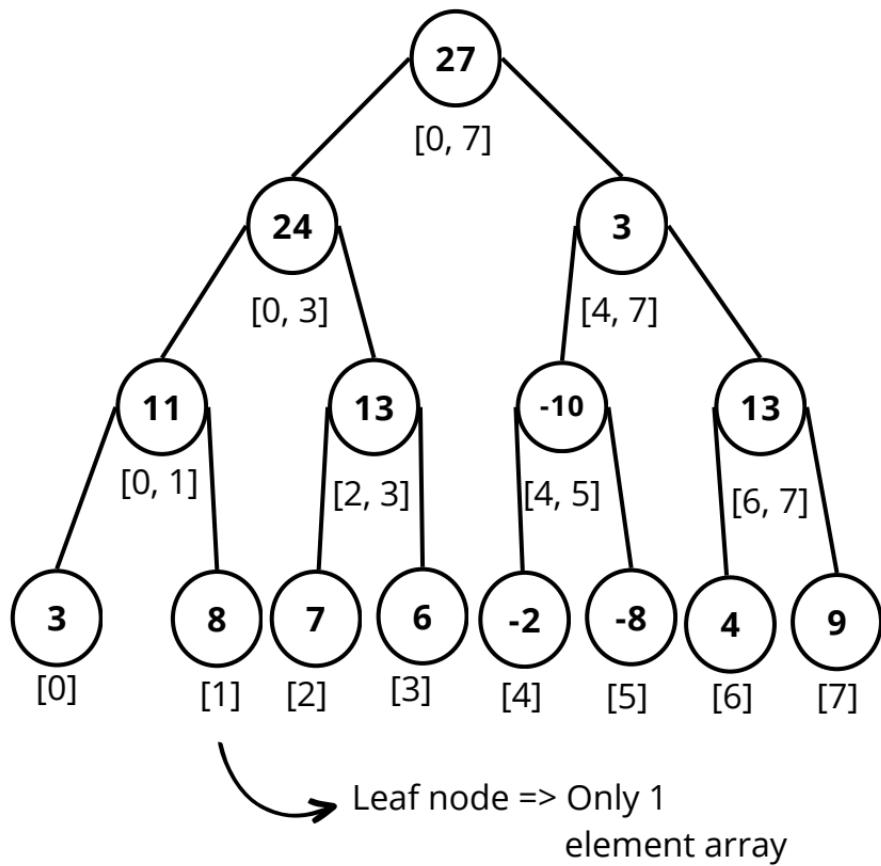
Segment => Perform query on a range (Sum, max, avg, min, product of all the numbers) in  $O(\log(N))$   
Update =>  $O(\log(N))$



Segment Tree for input array {1, 3, 5, 7, 9, 11}

- Some between two integer:

arr = [3, 8, 7, 6, -2, -8, 4, 9] , N = 8



So, Segment tree is a Full Binary Tree.

Leaf nodes = N

Internal nodes = N-1

Total nodes =  $2N - 1$

$$\frac{q}{\underline{\underline{=}}} \text{ Sum } b t u [2, 6] = [2, 3] + [$$

## Cases:

1. Node interval is inside query interval

Example: [4, 5] --> return the value

2. Node interval is completely outside the query interval. (node start index) > (query end index) or, (node end index) > (query start index)

[2, 6] --> [7, 7]

|

|-----> return the default value of the quarry.

In this case = 0 .

3. Overlapping:

## How to update in O(log(n)) time:

Change index 3 with 14 => (3, 14)

1. Check whether index lies in interval [0, 7]

2. If yes, then check child node. If child node is out, no change in value, just return.

3. In the end you will reach leafs. Update leafs and recursion will update the tree.

- **Make a program that will implement a Segment tree of given array data and query [0, 6]**

```
class SegmentTree {
```

```
    private static class Node {  
        int data;  
        int startInterval;  
        int endInterval;  
        Node left;  
        Node right;
```

```
        public Node (int startInterval, int endInterval) {  
            this.startInterval = startInterval;  
            this.endInterval = endInterval;  
        }  
    }
```

```
    Node root;
```

```
    public SegmentTree(int[] arr) {  
        // create a tree using this array  
        this.root = constructTree(arr, 0, arr.length - 1);  
    }
```

```
    private Node constructTree(int[] arr, int start, int end) {  
        if(start == end) {  
            // leaf node  
            Node leaf = new Node(start, end);  
            leaf.data = arr[start];  
            return leaf;  
        }  
    }
```

```
// create new node with index you are at
Node node = new Node(start, end);

int mid = (start + end) / 2;

node.left = this.constructTree(arr, start, mid);
node.right = this.constructTree(arr, mid + 1, end);

node.data = node.left.data + node.right.data;
return node;
}

public void display() {
    display(this.root);
}

private void display(Node node) {
    String str = "";

    if(node.left != null) {
        str = str + "Interval=[" + node.left.startInterval +
"- " + node.left.endInterval + "] and data: " +
node.left.data + " => ";
    } else {
        str = str + "No left child";
    }

    // for current node
    str = str + "Interval=[" + node.startInterval + "-" +
node.endInterval + "] and data: " + node.data + " <=
";
```

```
if(node.right != null) {
    str = str + "Interval=[" + node.right.startInterval
+ "-" + node.right.endInterval + "] and data: " +
node.right.data;
} else {
    str = str + "No right child";
}

System.out.println(str + '\n');

// call recursion
if(node.left != null) {
    display(node.left);
}

if(node.right != null) {
    display(node.right);
}
}

// query
public int query(int qsi, int qei) {
    return this.query(this.root, qsi, qei);
}

private int query(Node node, int qsi, int qei) {
    if(node.startInterval >= qsi && node.endInterval <=
qei) {
        // node is completely lying inside query
        return node.data;
    } else if (node.startInterval > qei || 
node.endInterval < qsi) {
```

```
// completely outside
return 0;
} else {
    return this.query(node.left, qsi, qei) +
    this.query(node.right, qsi, qei);
}
}

// update
public void update(int index, int value) {
    this.root.data = update(this.root, index, value);
}
private int update(Node node, int index, int value) {
    if (index >= node.startInterval && index <=
        node.endInterval){
        if(index == node.startInterval && index ==
            node.endInterval) {
            node.data = value;
            return node.data;
        } else {
            int leftAns = update(node.left, index, value);
            int rightAns = update(node.right, index, value);
            node.data = leftAns + rightAns;
            return node.data;
        }
    }
    return node.data;
}
```

```
public class Test {  
    public static void main(String[] args) {  
        int[] arr = {3, 8, 6, 7, -2, -8, 4, 9};  
        SegmentTree tree = new SegmentTree(arr);  
        tree.display();  
  
        System.out.println(tree.query(1, 6));  
    }  
}
```

### **Output:**

Interval=[0-3] and data: 24 => Interval=[0-7] and data: 27 <= Interval=[4-7] and data: 3

Interval=[0-1] and data: 11 => Interval=[0-3] and data: 24 <= Interval=[2-3] and data: 13

Interval=[0-0] and data: 3 => Interval=[0-1] and data: 11 <= Interval=[1-1] and data: 8

No left childInterval=[0-0] and data: 3 <= No right child

No left childInterval=[1-1] and data: 8 <= No right child

Interval=[2-2] and data: 6 => Interval=[2-3] and data: 13 <= Interval=[3-3] and data: 7

No left childInterval=[2-2] and data: 6 <= No right child

No left childInterval=[3-3] and data: 7 <= No right child

Interval=[4-5] and data: -10 => Interval=[4-7] and data: 3 <= Interval=[6-7] and data: 13  
nterval=[5-5] and data: -8  
nterval=[5-5] and data: -8

No left childInterval=[4-4] and data: -2 <= No right child

No left childInterval=[5-5] and data: -8 <= No right child

Interval=[6-6] and data: 4 => Interval=[6-7] and data: 13 <= Interval=[7-7] and data: 9

No left childInterval=[6-6] and data: 4 <= No right child

No left childInterval=[7-7] and data: 9 <= No right child

## **4. Heaps**

---

### **Definition of Heap data structure:**

A Heap is a special Tree-based Data Structure in which the tree is a complete binary tree.

### **Types of heaps:**

Generally, heaps are of two types.

1. Max-Heap
2. Min-Heap

#### **1. Max-Heap:**

In this heap, the value of the root node must be the greatest among all its child nodes and the same thing must be done for its left and right sub-tree also.

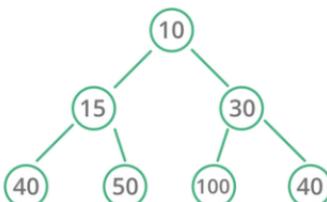
The total number of comparisons required in the max heap is according to the height of the tree. The height of the complete binary tree is always  $\log n$ ; therefore, the time complexity would also be  $O(\log n)$ .

## 2. Min-Heap:

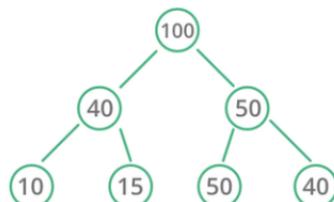
In this heap, the value of the root node must be the smallest among all its child nodes and the same thing must be done for its left and right sub-tree also.

The total number of comparisons required in the min heap is according to the height of the tree. The height of the complete binary tree is always  $\log n$ ; therefore, the time complexity would also be  $O(\log n)$ .

### Heap Data Structure



Min Heap



Max Heap

## **Properties of Heap:**

Heap has the following Properties:

- Complete Binary Tree: A heap tree is a complete binary tree, meaning all levels of the tree are fully filled except possibly the last level, which is filled from left to right. This property ensures that the tree is efficiently represented using an array.
- Heap Property: This property ensures that the minimum (or maximum) element is always at the root of the tree according to the heap type.
- Parent-Child Relationship: The relationship between a parent node at index 'i' and its children is given by the formulas: left child at index  $2i+1$  and right child at index  $2i+2$  for 0-based indexing of node numbers.
- Efficient Insertion and Removal: Insertion and removal operations in heap trees are efficient. New elements are inserted at the next available position in the bottom-rightmost level, and the heap property is restored by comparing the element with its parent and swapping if necessary. Removal of the root element involves replacing it with the last element and heapifying down.
- Efficient Access to Extremal Elements: The minimum or maximum element is always at the root of the heap, allowing constant-time access.

## **Operations Supported by Heap:**

Operations supported by min – heap and max – heap are same. The difference is just that min-heap contains minimum element at root of the tree and max – heap contains maximum element at the root of the tree.

### **Heapify:**

It is the process to rearrange the elements to maintain the property of heap data structure. It is done when a certain node creates an imbalance in the heap due to some operations on that node. It takes  $O(\log N)$  to balance the tree.

- For max-heap, it balances in such a way that the maximum element is the root of that binary tree and
- For min-heap, it balances in such a way that the minimum element is the root of that binary tree.

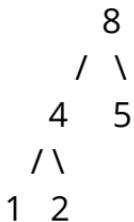
### **Insertion:**

- If we insert a new element into the heap since we are adding a new element into the heap so it will distort the properties of the heap so we need to perform the heapify operation so that it maintains the property of the heap.

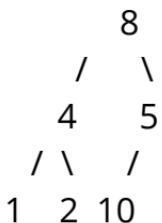
This operation also takes  $O(\log N)$  time.

## **Examples:**

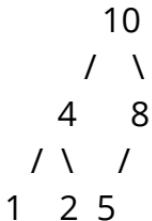
Assume initially heap(taking max-heap) is as follows



Now if we insert 10 into the heap



After heapify operation final heap will be look like this



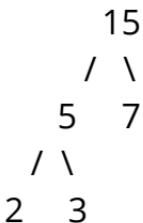
## **Deletion:**

- If we delete the element from the heap it always deletes the root element of the tree and replaces it with the last element of the tree.
- Since we delete the root element from the heap it will distort the properties of the heap so we need to perform heapify operations so that it maintains the property of the heap.

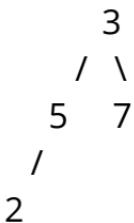
It takes  $O(\log N)$  time.

## **Example:**

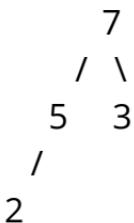
Assume initially heap(taking max-heap) is as follows



Now if we delete 15 into the heap it will be replaced by leaf node of the tree for temporary.



After heapify operation final heap will be look like this



## **getMax (For max-heap) or getMin (For min-heap):**

It finds the maximum element or minimum element for max-heap and min-heap respectively and as we know minimum and maximum elements will always be the root node itself for min-heap and max-heap respectively. It takes O(1) time.

## **removeMin or removeMax:**

This operation returns and deletes the maximum element and minimum element from the max-heap and min-heap respectively. In short, it deletes the root element of the heap binary tree.

## **Implementation of Heap Data Structure:-**

The following code shows the implementation of a max-heap.

Let's understand the maxHeapify function in detail:-

maxHeapify is the function responsible for restoring the property of the Max Heap. It arranges the node i, and its subtrees accordingly so that the heap property is maintained.

- Suppose we are given an array, arr[] representing the complete binary tree. The left and the right child of ith node are in indices  $2*i+1$  and  $2*i+2$ .

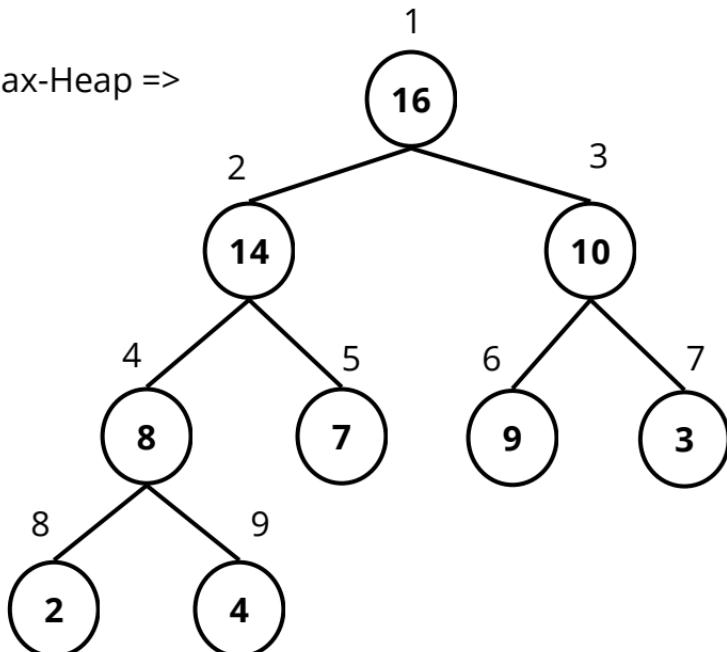
- We set the index of the current element,  $i$ , as the 'MAXIMUM'.
- If  $\text{arr}[2 * i + 1] > \text{arr}[i]$ , i.e., the left child is larger than the current value, it is set as 'MAXIMUM'.
- Similarly if  $\text{arr}[2 * i + 2] > \text{arr}[i]$ , i.e., the right child is larger than the current value, it is set as 'MAXIMUM'.
- Swap the 'MAXIMUM' with the current element.
- Repeat steps 2 to 5 till the property of the heap is restored.

## Heap:

1. Complete Binary Tree
2. In max-heap every node value  $\geq$  All of its children node value  
or,  
In min-heap every node value  $\leq$  All of its children node value
3. Root  $\Rightarrow i = 1$
4. parent ( $i$ ) =  $i/2$
5. left ( $i$ ) =  $2 * i$
6. right ( $i$ ) =  $(2 * i) + 1$
7. Height =  $\log(N)$

1	2	3	4	5	6	7	8	9	10
arr =	16	14	10	8	7	9	3	2	4

Max-Heap =>

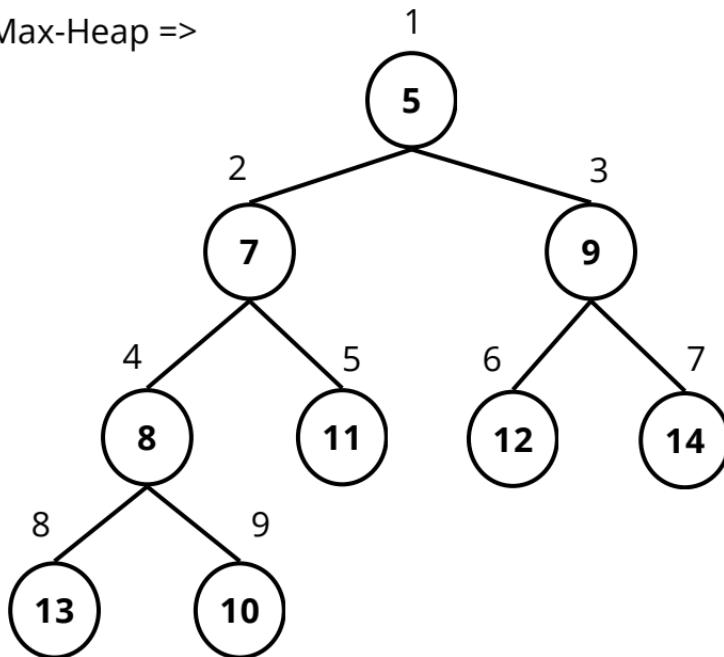


	1	2	3	4	5	6	7	8	9	10
arr =	7	8	9	10	11	12	14	13	5	

After insert 5:

	1	2	3	4	5	6	7	8	9	10
arr =	5	7	9	8	11	12	14	13	10	

Max-Heap =>



- **Implementation of Heap Data Structure + Heapsort + Priority Queue.**

```
import java.util.ArrayList;

class Heap<T extends Comparable<T>> {

    private ArrayList<T> list;

    public Heap() {
        list = new ArrayList<>();
    }

    private void swap(int first, int second) {
        T temp = list.get(first);
        list.set(first, list.get(second));
        list.set(second, temp);
    }

    private int parent(int index) {
        return (index - 1) / 2;
    }

    private int left(int index) {
        return index * 2 + 1;
    }

    private int right(int index) {
        return index * 2 + 2;
    }
}
```

```
public void insert(T value) {  
    list.add(value);  
    upheap(list.size() - 1);  
}  
  
private void upheap(int index) {  
    if(index == 0) {  
        return;  
    }  
    int p = parent(index);  
    if(list.get(index).compareTo(list.get(p)) < 0) {  
        swap(index, p);  
        upheap(p);  
    }  
}  
  
public T remove() throws Exception {  
    if (list.isEmpty()) {  
        throw new Exception("Removing from an empty  
        heap!");  
    }  
  
    T temp = list.get(0);  
  
    T last = list.remove(list.size() - 1);  
    if (!list.isEmpty()) {  
        list.set(0, last);  
        downheap(0);  
    }  
  
    return temp;  
}
```

```
private void downheap(int index) {  
    int min = index;  
    int left = left(index);  
    int right = right(index);  
  
    if(left < list.size() &&  
        list.get(min).compareTo(list.get(left)) > 0) {  
        min = left;  
    }  
  
    if(right < list.size() &&  
        list.get(min).compareTo(list.get(right)) > 0) {  
        min = right;  
    }  
  
    if(min != index) {  
        swap(min, index);  
        downheap(min);  
    }  
}  
  
public ArrayList<T> heapSort() throws Exception {  
    ArrayList<T> data = new ArrayList<>();  
    while(!list.isEmpty()) {  
        data.add(this.remove());  
    }  
    return data;  
}
```

```
class Test {  
    public static void main(String[] args) throws Exception{  
        Heap<Integer> heap = new Heap<>();  
  
        heap.insert(34);  
        heap.insert(45);  
        heap.insert(22);  
        heap.insert(89);  
        heap.insert(76);  
  
        ArrayList list = heap.heapSort();  
        System.out.println(list);  
  
    }  
}
```

**Output:**

[22, 34, 45, 76, 89]

## 5. Trie

---

A Trie (pronounced "try"), also known as a prefix tree or digital tree or k-ary search tree or retrieval tree, is a multiway tree data structure used for storing strings over an alphabet. It is used to store a large amount of strings. The pattern matching can be done efficiently using tries.

It's particularly useful for tasks involving prefix-based operations, such as searching for words with specific prefixes or autocomplete suggestions.

The trie shows words like allot, alone, ant, and, are, bat, bad. The idea is that all strings sharing common prefix should come from a common node. The tries are used in spell checking programs.

- Preprocessing pattern improves the performance of pattern matching algorithm. But if a text is very large then it is better to preprocess text instead of pattern for efficient search.
- A trie is a data structure that supports pattern matching queries in time proportional to the pattern size.

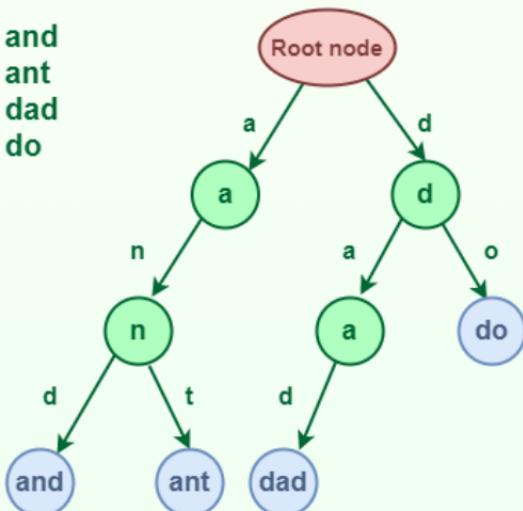
If we store keys in a binary search tree, a well balanced BST will need time proportional to  $M * \log N$ , where M is the maximum string length and N is the number of keys in the tree. Using Trie, the

key can be searched in  $O(M)$  time. However, the penalty is on Trie storage requirements (Please refer to Applications of Trie for more details).

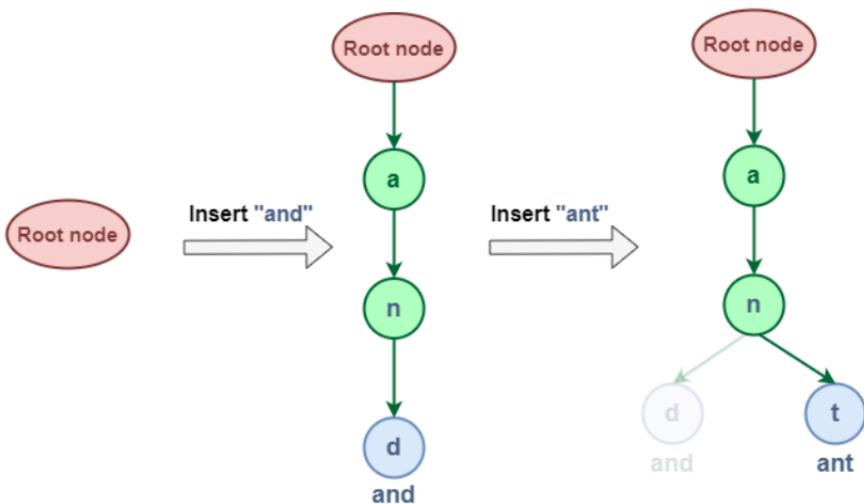
Trie is also known as digital tree or prefix tree.  
Refer to this article for more detailed information.

### Trie Data Structure

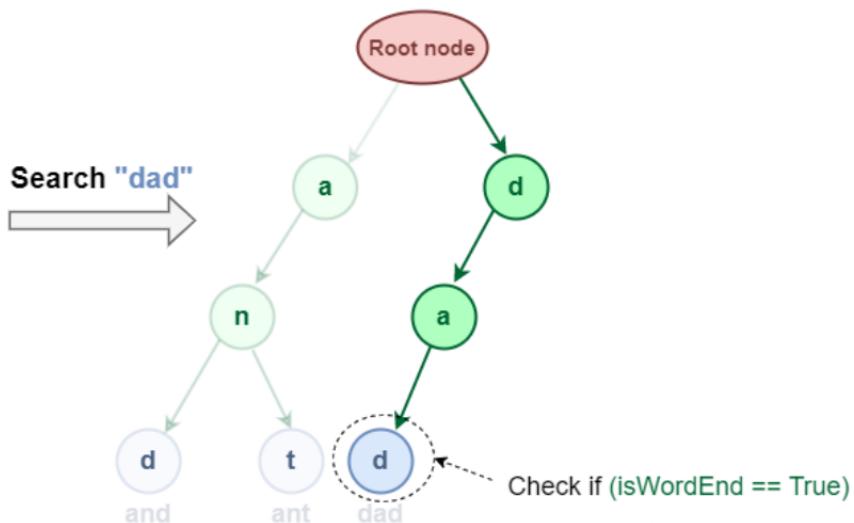
- and
- ant
- dad
- do



## Insert Operation in Trie:



## Search Operation in Trie:



**Time complexity:**

$O(\text{key\_length})$

**Space complexity:**

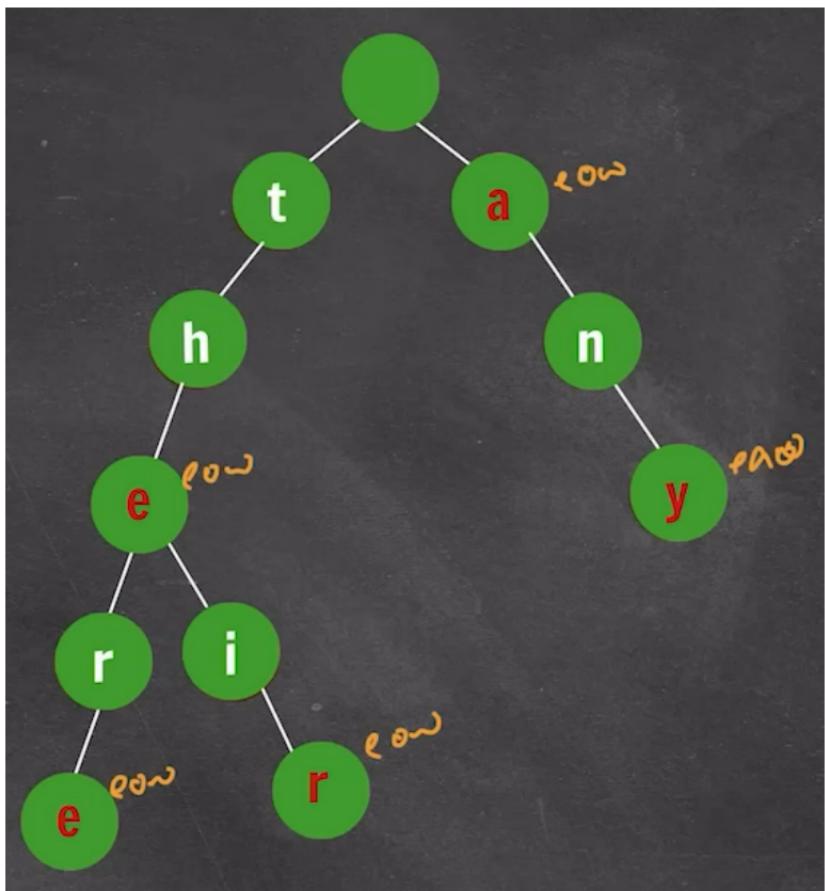
$O(\text{ALPHABET\_SIZE} * \text{key\_length} * N)$

where **N** is the **number of keys in Trie**. There are efficient representations of trie nodes (e.g. compressed trie, ternary search tree, etc.) to minimize the memory requirements of the trie.

- **Trie code implementation:**

words[ ] = “the”, “a”, “there”, “their”, “any”

Trie of these words :



```
public class Test {  
    static class Node {  
        Node[] children;  
        boolean eow; // endOfWord  
  
        public Node() {  
            children = new Node[26];  
            for (int i=0; i<26; i++) {  
                children[i] = null;  
            }  
        }  
    }  
  
    public static Node root = new Node();  
  
    public static void insert(String word) { //O(wordLength)  
        Node curr = root;  
  
        for(int i = 0; i<word.length(); i++) {  
            int idx = word.charAt(i)-'a';  
            if(curr.children[idx] == null) {  
                curr.children[idx] = new Node();  
            }  
            curr = curr.children[idx];  
        }  
        curr.eow = true;  
    }  
  
    public static boolean search(String key) {  
        //O(wordLength)  
        Node curr = root;
```

```
        for(int i = 0; i<key.length(); i++) {  
            int idx = key.charAt(i)-'a';  
            if(curr.children[idx] == null) {  
                return false;  
            }  
            curr = curr.children[idx];  
        }  
  
        return curr.eow == true;  
    }  
  
    public static void main(String args[]) {  
        String words[] = {"the", "a", "there", "their", "any",  
"thee"};  
  
        for (String word : words) {  
            insert(word);  
            System.out.println("inserted " + word);  
        }  
  
        System.out.println("thee -> " + search("thee"));  
        System.out.println("thor -> " + search("thor"));  
    }  
}
```

**Output:**

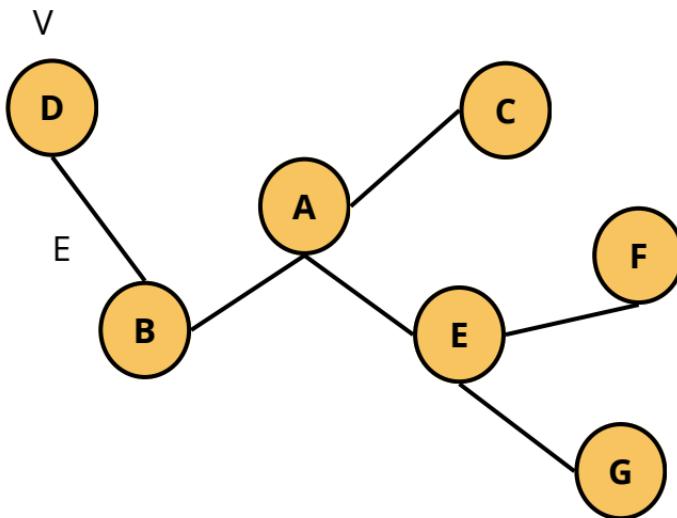
inserted the  
inserted a  
inserted there  
inserted their  
inserted any  
inserted thee  
thee -> true  
thor -> false

## 6. Graphs

---

### Definition of Graphs:

- A graph  $G=(V, E)$  consists a set of vertices,  $V$ , and a set of edges,  $E$ .
- Each edge is a pair of  $(v, w)$ , where  $v, w$  belongs to  $V$
- If the pair is unordered, the graph is undirected; otherwise it is directed



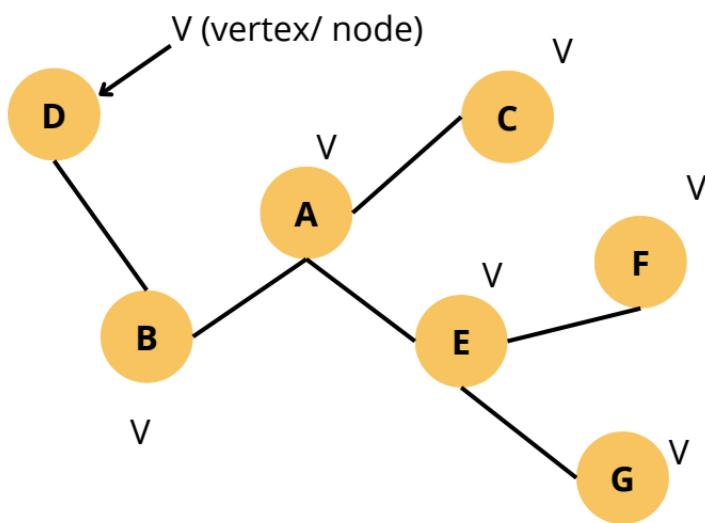
## **Applications of Graphs:**

1. Maps
2. Social Network
3. Delivery Network

## Basics of Graphs:

**Vertex:**

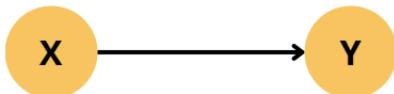
V



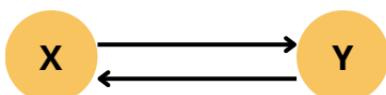
**Edge:**

E

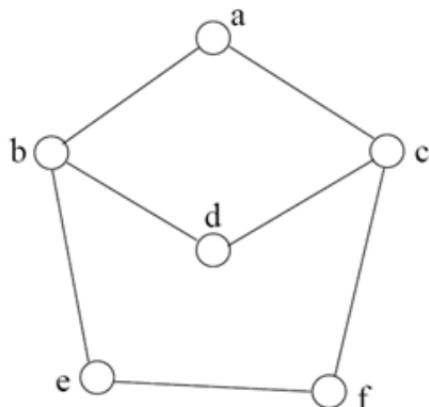
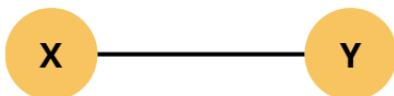
1. Uni-directional



2. Bi-directional / Undirectional



or,



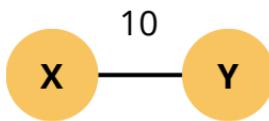
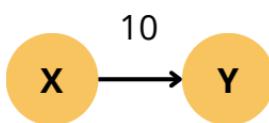
$$V = \{a, b, c, d, e, f\}$$

$$E = \{\{a, b\}, \{a, c\}, \{a, d\}, \{a, e\}, \{a, f\}, \{b, c\}, \{b, d\}, \{b, e\}, \{b, f\}, \{c, d\}, \{c, e\}, \{c, f\}, \{d, e\}, \{d, f\}, \{e, f\}\}$$

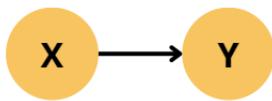
**Weight:**

W

1. Weighted:



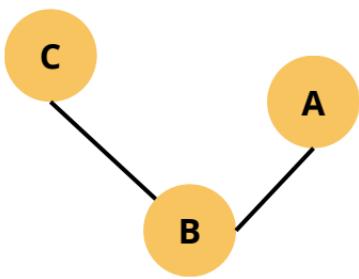
2. Unweighted:



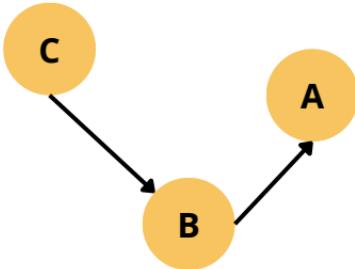
## Types of Graphs:

Variations:

- A **connected graph** has a path from every vertex to every other.
- In an **undirected graph**:
  - > Edge  $(u,v) = \text{Edge } (v,u)$
  - > No self-loops
- In a **directed** graph:
  - > Edge  $(u,v)$  goes from vertex  $u$  to vertex  $v$ , notated  $u \rightarrow v$
- A **weighted graph** associates weights with either the edges or the vertices
  - > E.g., a road map: edges might be weighted w distance
- A **multi-graph** allows multiple edges between the same vertices
  - > E.g., the call graph in a program (a function can get called from multiple points in another function)



Undirected weighted Graph



Directed unweighted Graph

- We will typically express running times in terms of  $|E|$  and  $|V|$  (often dropping the '|s)
  - > If  $|E| \approx |V|^2$  the graph is dense
  - > If  $|E| \approx |V|$  the graph is sparse
- If you know you are dealing with dense or sparse graphs, different data structures may make sense

- Complete Graph:
  - > How many edges are there in an  $N$ -vertex complete graph?
- Bipartite Graph:
  - > What is its property? How can we detect?
- Path:
- Tour:
- Degree of a vertices
  - > Indegree
  - > Outdegree
  - > Indegree+outdegree = Even (why??)

## Storing a Graph / Graph Representation:

1. Adjacency List
2. Adjacency Matrix
3. Edge List
4. 2D Matrix (Implicit Graph)

Two popular computer representations of a graph.  
Both represent the vertex set and the edge set, but  
in different ways.

### **1. Adjacency Matrix:**

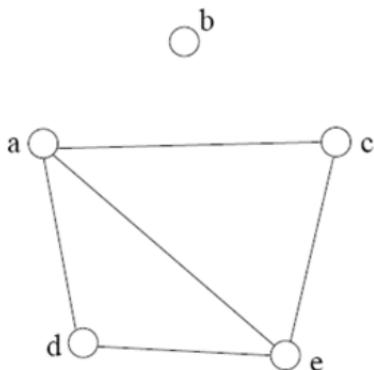
Use a 2D matrix to represent the graph

### **2. Adjacency List:**

Use a 1D array of linked lists

## 1. Adjacency Matrix:

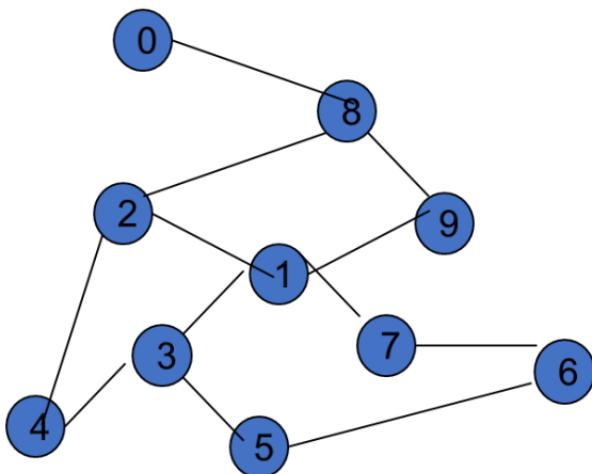
2D Matrix



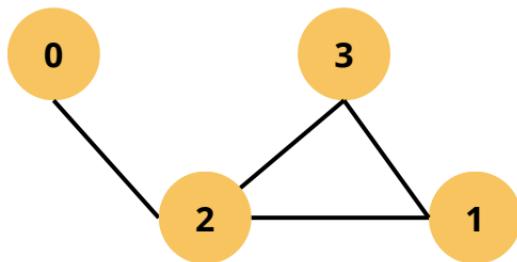
	a	b	c	d	e
a	0	0	1	1	1
b	0	0	0	0	0
c	1	0	0	0	1
d	1	0	0	0	1
e	1	0	1	1	0

- 2D array  $A[0..n-1, 0..n-1]$ , where  $n$  is the number of vertices in the graph
- Each row and column is indexed by the vertex id
- e.g.  $a=0, b=1, c=2, d=3, e=4$
- $A[i][j]=1$  if there is an edge connecting vertices  $i$  and  $j$ ; otherwise,  $A[i][j]=0$
- The storage requirement is  $\Theta(n^2)$ . It is not efficient if the graph has few edges. An adjacency matrix is an appropriate representation if the graph is dense:  
 $|E|=\Theta(|V|^2)$
- We can detect in  $O(1)$  time whether two vertices are connected.

## Adjacency Matrix Example:

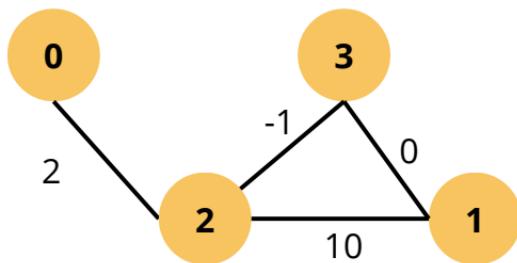


## Adjacency Matrix Example:



	0	1	2	3
0	0	0	1	0
1	0	0	1	1
2	1	1	0	1
3	0	1	1	0

## Adjacency Matrix Example:



	0	1	2	3
0	0	0	2	0
1	0	0	10	0
2	2	10	0	-1
3	0	0	-1	0

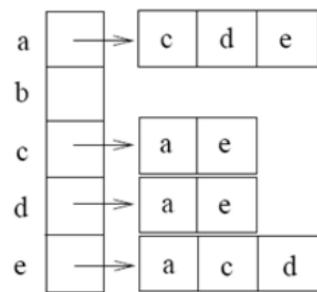
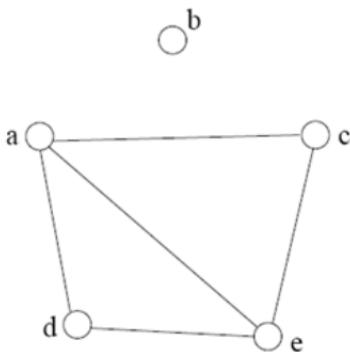
## **Simple Questions on Adjacency Matrix:**

- Is there a direct link between A and B?
- What is the indegree and outdegree for a vertex A?
- How many nodes are directly connected to vertex A?
- Is it an undirected graph or directed graph?
- Suppose ADJ is an NxN matrix. What will be the result if we create another matrix ADJ2 where  $ADJ2=ADJ \times ADJ$ ?

## 2. Adjacency List:

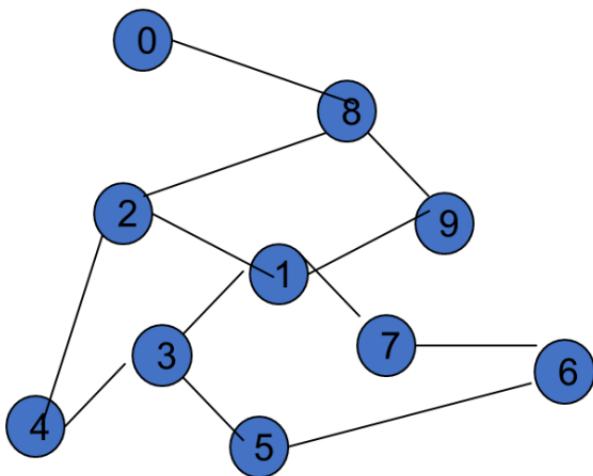
List of Lists

It is optimise than others.



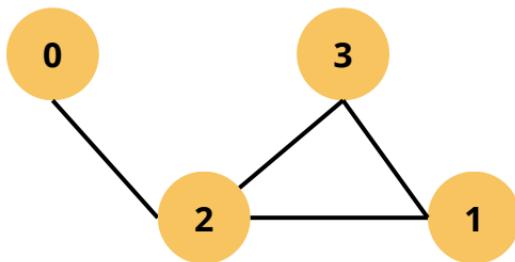
- If the graph is not dense, in other words, **sparse**, a better solution is an adjacency list
- The adjacency list is **an array  $A[0..n-1]$  of lists**, where  $n$  is the number of vertices in the graph.
- Each array entry is indexed by the vertex id
- Each **list  $A[i]$**  stores the **ids of the vertices adjacent to vertex  $i$**

## Adjacency List Example:



0	→	8
1	→	2 3 7 9
2	→	1 4 8
3	→	1 4 5
4	→	2 3
5	→	3 6
6	→	5 7
7	→	1 6
8	→	0 2 9
9	→	1 8

## Adjacency List code Example:

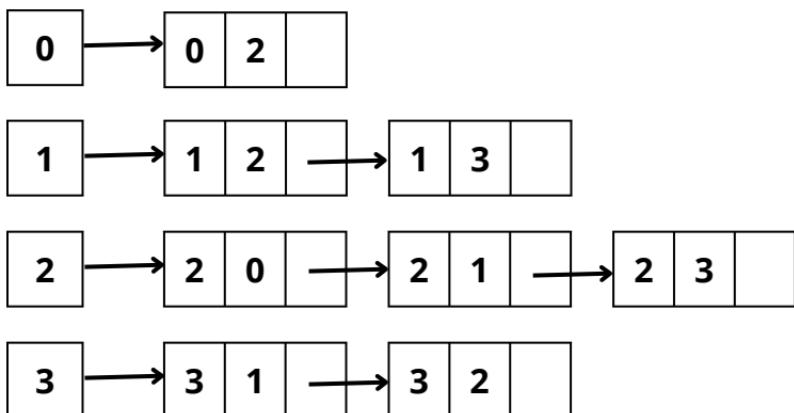


0 --> {0,2}

1 --> {1,2}, {1,3}

2 --> {2, 0}, {2,1}, {2,3}

3 --> {3,1}, {3,2}



```
import java.util.ArrayList;

public class Test{
    static class Edge{
        int src;
        int dest;

        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[])
    {
        // creating empty ArrayList of graph
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<Edge>();
        }

        graph[0].add(new Edge(0, 2));

        graph[1].add(new Edge(1, 2));
        graph[1].add(new Edge(1, 3));

        graph[2].add(new Edge(2, 0));
        graph[2].add(new Edge(2, 1));
        graph[2].add(new Edge(2, 3));

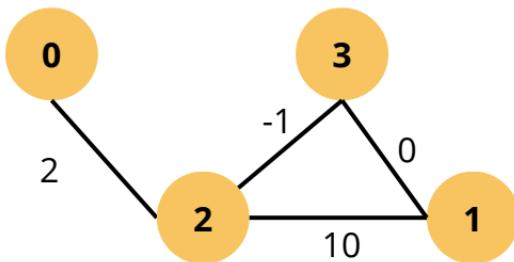
        graph[3].add(new Edge(3, 1));
        graph[3].add(new Edge(3, 2));
    }
}
```

```
public static void main(String[] args) {  
    int v = 4;  
  
    ArrayList<Edge> graph[] = new ArrayList[v];  
    createGraph(graph);  
  
    // print 2's neighbours  
    for(int i=0; i<graph[2].size(); i++){  
        Edge e = graph[2].get(i);  
        System.out.print(e.dest+" ");  
    }  
}  
}
```

### **Output:**

0 1 3

## Adjacency List code Example:

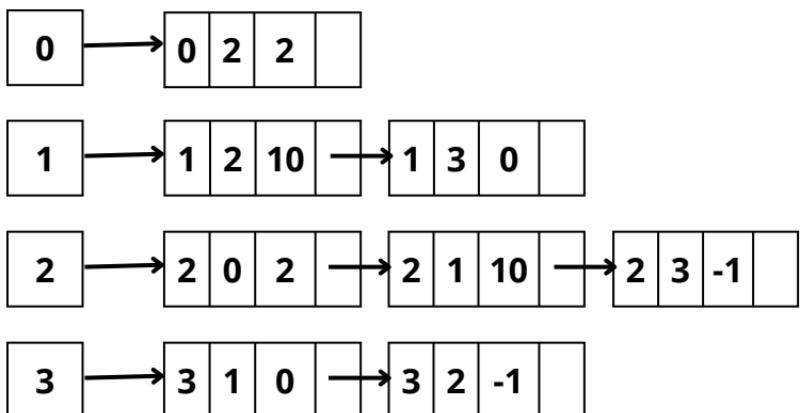


0 --> {0,2,2}

1 --> {1,2,10}, {1,3,0}

2 --> {2, 0,2}, {2,1,10}, {2,3,-1}

3 --> {3,1,0}, {3,2,-1}



```
import java.util.ArrayList;
```

```
public class Test{
```

```
    static class Edge{
```

```
        int src;
```

```
        int dest;
```

```
        int weight;
```

```
        public Edge(int s, int d, int w){
```

```
            this.src = s;
```

```
            this.dest = d;
```

```
            this.weight = w;
```

```
        }
```

```
}
```

```
    public static void createGraph(ArrayList<Edge> graph[])
```

```
{
```

```
        // creating empty ArrayList of graph
```

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

```
            graph[i] = new ArrayList<Edge>();
```

```
        }
```

```
        graph[0].add(new Edge(0, 2, 2));
```

```
        graph[1].add(new Edge(1, 2, 10));
```

```
        graph[1].add(new Edge(1, 3, 0));
```

```
        graph[2].add(new Edge(2, 0, 2));
```

```
        graph[2].add(new Edge(2, 1, 10));
```

```
        graph[2].add(new Edge(2, 3, -1));
```

```
graph[3].add(new Edge(3, 1, 0));
graph[3].add(new Edge(3, 2, -1));
}

public static void main(String[] args) {
    int v = 4;

    ArrayList<Edge> graph[] = new ArrayList[v];
    createGraph(graph);

    // print 2's neighbours with weight
    for(int i=0; i<graph[2].size(); i++){
        Edge e = graph[2].get(i);
        System.out.println(e.dest+", "+e.weight);
    }

}

}
```

### **Output:**

0, 2  
1, 10  
3, -1

## Storage of Adjacency List:

- The array takes up  $\Theta(n)$  space
- Define **degree** of  $v$ ,  $\deg(v)$ , to be the number of edges incident to  $v$ . Then, the total space to store the graph is proportional to:

$$\sum_{\text{vertex } v} \deg(v)$$

- An edge  $e=\{u,v\}$  of the graph contributes a count of 1 to  $\deg(u)$  and contributes a count 1 to  $\deg(v)$
- Therefore,  $\sum_{\text{vertex } v} \deg(v) = 2m$ , where  $m$  is the total number of edges
- In all, the **adjacency list takes up  $\Theta(n+m)$  space**
  - > If  $m = O(n^2)$  (i.e. dense graphs), both adjacent matrix and adjacent lists use  $\Theta(n^2)$  space.
  - > If  $m = O(n)$ , adjacent list outperform adjacent matrix
- However, one cannot tell in  $O(1)$  time whether two vertices are connected

## **Adjacency List vs. Matrix:**

### **Adjacency List:**

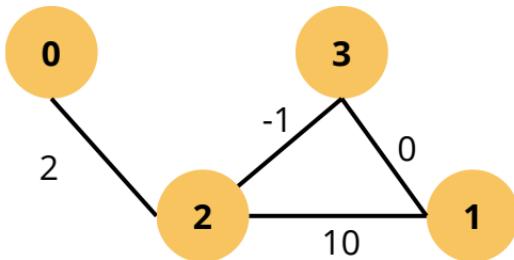
- More compact than adjacency matrices if graph has few edges
- Requires more time to find if an edge exists

### **Adjacency Matrix:**

- Always require  $n^2$  space
  - > This can waste a lot of space if the number of edges are sparse
- Can quickly find if an edge exists

### 3. Edge List:

Edge ArrayList / LinkedList

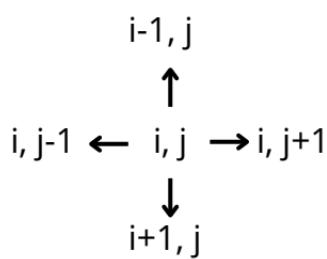


Edge = {{0, 2}, {1, 2}, {1, 3}, {2, 3}}

We use this method in Minimum Spanning Tree(MST) --> Edges sort

#### 4. Implicit Graph:

2D array



		X	

## **Path between Vertices:**

- A **path** is a sequence of vertices  $(v_0, v_1, v_2, \dots, v_k)$  such that:
  - > For  $0 \leq i < k$ ,  $\{v_i, v_{i+1}\}$  is an edge

Note: a path is allowed to go through the same vertex or the same edge any number of times!

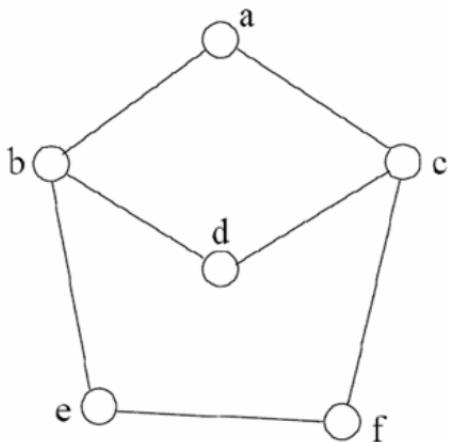
- The **length** of a path is the number of edges on the path

## Types of paths:



- A path is **simple** if and only if it does not contain a vertex more than once.
- A path is a **cycle** if and only if  $v_0 = v_k$   
    > The beginning and end are the same vertex!
- A path contains a cycle as its sub-path if some vertex appears twice or more

## Path Examples:



Are these paths?

Any cycles?

What is the path's length?

1. {a,c,f,e}
2. {a,b,d,c,f,e}
3. {a, c, d, b, d, c, f, e}  $\square$
4. {a,c,d,b,a}
5. {a,c,f,e,b,d,c,a}

## Graph Traversal:

- Application example:
  - > Given a graph representation and a vertex  $s$  in the graph
  - > Find paths from  $s$  to other vertices
  
- Two common graph traversal algorithms:
  1. Breadth-First Search (BFS):
    - > Find the shortest paths in an unweighted graph
  2. Depth-First Search (DFS):
    - > Topological sort
    - > Find strongly connected components

## **1. Breadth-First Search:**

Go to the immediate neighbors first.

- Explore" a graph, turning it into a tree
  - > One vertex at a time
  - > Expand frontier of explored vertices across the breadth of the frontier
- Builds a tree over the graph
  - > Pick a source vertex to be the root
  - > Find ("discover") its children, then their children, etc.

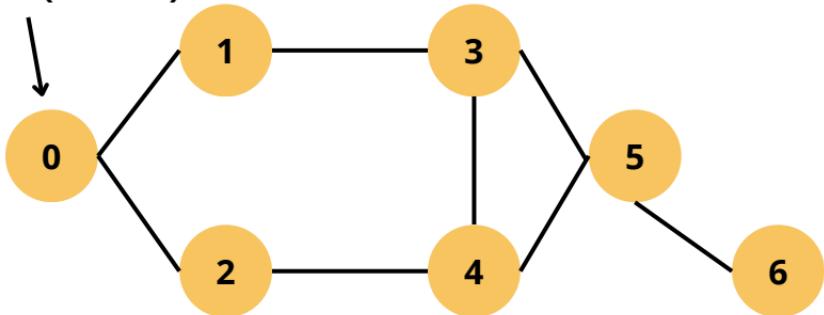
Here, Queue(FIFO) data structure is used for BFS.

Application of BFS:

- Find the shortest path in an undirected/directed unweighted graph.
- Find the bipartiteness of a graph.
- Find cycle in a graph.
- Find the connectedness of a graph.

## BFS example:

S (Source)



Visit step:

1. print (Current node),
2. Visit (Current node) = T,
3. Current node neighbors add in the Queue.

Step 1:

$Q =$	<del>0</del>	1	2				
-------	--------------	---	---	--	--	--	--

0	1	2	3	4	5	6
vis =	T	F	F	F	F	F

### Step 2:

Q =	<del>1</del>	2	0	3			
-----	--------------	---	---	---	--	--	--

0      1      2      3      4      5      6

vis =	T	T	F	F	F	F	F
-------	---	---	---	---	---	---	---

### Step 3:

Q =	<del>2</del>	0	3	0	4		
-----	--------------	---	---	---	---	--	--

0      1      2      3      4      5      6

vis =	T	T	T	F	F	F	F
-------	---	---	---	---	---	---	---

### Step 4:

Q =	<del>0</del>	3	0	4			
-----	--------------	---	---	---	--	--	--

0      1      2      3      4      5      6

vis =	T	T	T	F	F	F	F
-------	---	---	---	---	---	---	---

### Step 5:

Q =	<del>3</del>	0	4	1	4	5	
-----	--------------	---	---	---	---	---	--

	0	1	2	3	4	5	6
vis =	T	T	T	T	F	F	F

### Step 6:

Q =	<del>0</del>	4	1	4	5		
-----	--------------	---	---	---	---	--	--

	0	1	2	3	4	5	6
vis =	T	T	T	T	F	F	F

### Step 7:

Q =	<del>4</del>	1	4	5	2	3	5
-----	--------------	---	---	---	---	---	---

	0	1	2	3	4	5	6
vis =	T	T	T	T	T	F	F

### Step 8:

Q =	<del>1</del>	4	5	2	3	5	
-----	--------------	---	---	---	---	---	--

0      1      2      3      4      5      6

vis =	T	T	T	T	T	F	F
-------	---	---	---	---	---	---	---

### Step 9:

Q =	<del>4</del>	5	2	3	5		
-----	--------------	---	---	---	---	--	--

0      1      2      3      4      5      6

vis =	T	T	T	T	T	F	F
-------	---	---	---	---	---	---	---

### Step 10:

Q =	<del>5</del>	2	3	5	3	4	6
-----	--------------	---	---	---	---	---	---

0      1      2      3      4      5      6

vis =	T	T	T	T	T	T	F
-------	---	---	---	---	---	---	---

### Step 11:

Q =	<del>2</del>	3	5	3	4	6	
-----	--------------	---	---	---	---	---	--

	0	1	2	3	4	5	6
vis =	T	T	T	T	T	T	F

### Step 12:

Q =	<del>3</del>	5	3	4	6		
-----	--------------	---	---	---	---	--	--

	0	1	2	3	4	5	6
vis =	T	T	T	T	T	T	F

### Step 13:

Q =	<del>5</del>	3	4	6			
-----	--------------	---	---	---	--	--	--

	0	1	2	3	4	5	6
vis =	T	T	T	T	T	T	F

### Step 14:

Q =	<del>3</del>	4	6				
-----	--------------	---	---	--	--	--	--

0      1      2      3      4      5      6

vis =	T	T	T	T	T	T	F
-------	---	---	---	---	---	---	---

### Step 15:

Q =	<del>4</del>	6					
-----	--------------	---	--	--	--	--	--

0      1      2      3      4      5      6

vis =	T	T	T	T	T	T	F
-------	---	---	---	---	---	---	---

### Step 16:

Q =	<del>6</del>	5					
-----	--------------	---	--	--	--	--	--

0      1      2      3      4      5      6

vis =	T	T	T	T	T	T	T
-------	---	---	---	---	---	---	---

## Step 17:

Q =	<del>5</del>						
-----	--------------	--	--	--	--	--	--

	0	1	2	3	4	5	6
vis =	T	T	T	T	T	T	T

## Time complexity:

Total running time:  $O(V+E)$

## **BFS code for connected or disconnected Graph:**

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

public class Test{
    static class Edge{
        int src;
        int dest;

        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[])
    {
        // creating empty ArrayList of graph
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<Edge>();
        }

        graph[0].add(new Edge(0, 1));
        graph[0].add(new Edge(0, 2));

        graph[1].add(new Edge(1, 0));
        graph[1].add(new Edge(1, 3));

        graph[2].add(new Edge(2, 0));
        graph[2].add(new Edge(2, 4));
    }
}
```

```
graph[3].add(new Edge(3, 1));
graph[3].add(new Edge(3, 4));
graph[3].add(new Edge(3, 5));

graph[4].add(new Edge(4, 2));
graph[4].add(new Edge(4, 3));
graph[4].add(new Edge(4, 5));

graph[5].add(new Edge(5, 3));
graph[5].add(new Edge(5, 4));
graph[5].add(new Edge(5, 6));
}

public static void bfs(ArrayList<Edge> graph[], int v,
boolean vis[], int start){
Queue<Integer> q = new LinkedList<>();

q.add(start);

while(!q.isEmpty()){
    int curr = q.remove();
    if(vis[curr] == false){
        System.out.print(curr+" ");
        vis[curr] = true;

        for (int i=0; i<graph[curr].size(); i++) {
            Edge e = graph[curr].get(i);
            q.add(e.dest);
        }
    }
}
}
```

```
public static void main(String[] args) {  
    int v = 7;  
  
    ArrayList<Edge> graph[] = new ArrayList[v];  
    createGraph(graph);  
  
    boolean vis[] = new boolean[v];  
  
    // for disconnected graph  
    for(int i=0; i<v; i++){  
        if(vis[i] == false){  
            bfs(graph, v, vis, i);  
        }  
    }  
  
    System.out.println();  
}  
}
```

### **Output:**

0 1 2 3 4 5 6

## 2. Depth-First Search:

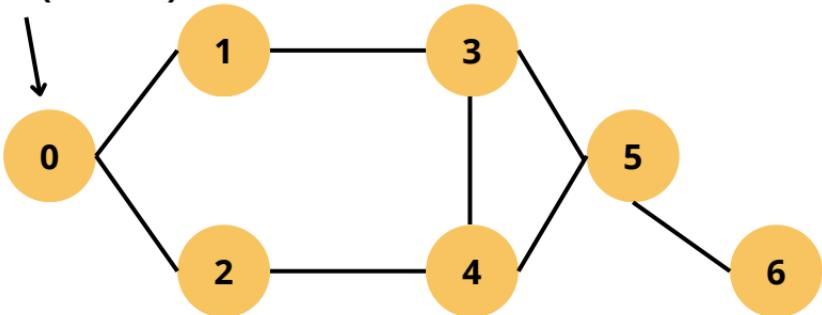
Keep going to the 1st neighbor.

- Search “deeper” in the graph whenever possible
- Edges are explored out of the most recently discovered vertex  $v$  that still has unexplored edges
- *After all edges of  $v$  have been explored, the search “backtracks” from the parent of  $v$*
- *The process continues until all vertices reachable from the original source have been discovered*
- *If undiscovered vertices remain, choose one of them as a new source and repeat the search from that vertex*
- *DFS creates a “depth-first forest”*

Here, Stack / Recursion used for DFS.

## DFS example:

S (Source)

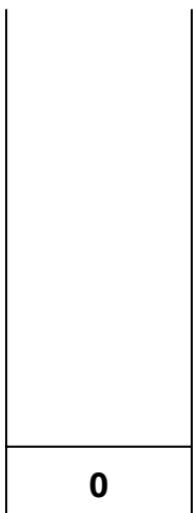


Visit step:

1. print (Current node),
2. Visit (Current node) = T,
3. Neighbor visiting by the help of Stack/Recursion

Step 1:

0	1	2	3	4	5	6
vis =	T	F	F	F	F	F



**Step 0:**

0	1	2	3	4	5	6
vis =	F	F	F	F	F	F

Stack

**Step 1:**

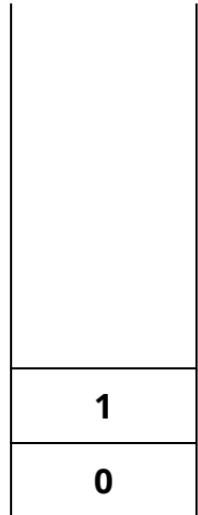
0	1	2	3	4	5	6
vis =	T	F	F	F	F	F

0

Stack

## Step 2:

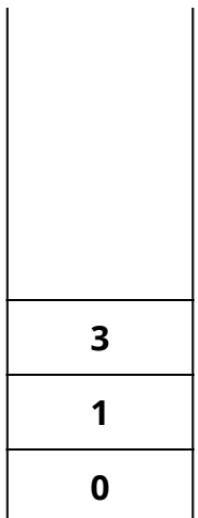
	0	1	2	3	4	5	6
vis =	T	T	F	F	F	F	F



Stack

## Step 3:

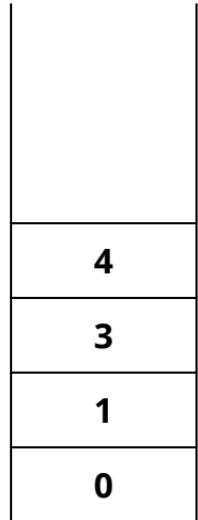
	0	1	2	3	4	5	6
vis =	T	T	F	T	F	F	F



Stack

### **Step 4:**

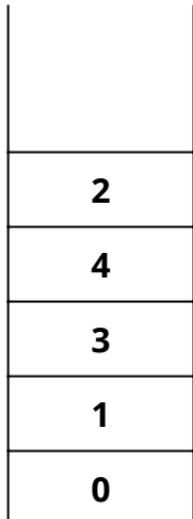
	0	1	2	3	4	5	6
vis =	T	T	F	T	T	F	F



Stack

### **Step 4:**

	0	1	2	3	4	5	6
vis =	T	T	T	T	T	F	F



Stack

### **Step 5:**

	0	1	2	3	4	5	6
vis =	T	T	T	T	T	F	F

4
3
1
0

Stack

### **Step 6:**

	0	1	2	3	4	5	6
vis =	T	T	T	T	T	T	F

5
4
3
1
0

Stack

### **Step 7:**

vis =

0	1	2	3	4	5	6
T	T	T	T	T	T	T

6
5
4
3
1
0

Stack

### **Step 7:**

vis =

0	1	2	3	4	5	6
T	T	T	T	T	T	T

5
4
3
1
0

Stack

### **Step 8:**

	0	1	2	3	4	5	6
vis =	T	T	T	T	T	T	T

4
3
1
0

Stack

### **Step 9:**

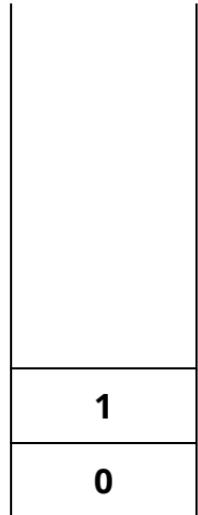
	0	1	2	3	4	5	6
vis =	T	T	T	T	T	T	T

3
1
0

Stack

### **Step 10:**

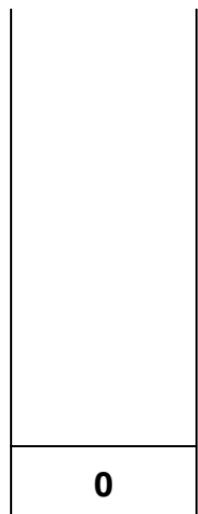
	0	1	2	3	4	5	6
vis =	T	T	T	T	T	T	T



Stack

### **Step 11:**

	0	1	2	3	4	5	6
vis =	T	T	T	T	T	T	T



Stack

## Step 12:

	0	1	2	3	4	5	6
vis =	T	T	T	T	T	T	T

Stack

## Time complexity:

Total running time:  $O(V+E)$

## **DFS code for connected or disconnected Graph:**

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

public class Test{
    static class Edge{
        int src;
        int dest;

        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[])
    {
        // creating empty ArrayList of graph
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<Edge>();
        }

        graph[0].add(new Edge(0, 1));
        graph[0].add(new Edge(0, 2));

        graph[1].add(new Edge(1, 0));
        graph[1].add(new Edge(1, 3));
    }
}
```

```
graph[2].add(new Edge(2, 0));
graph[2].add(new Edge(2, 4));

graph[3].add(new Edge(3, 1));
graph[3].add(new Edge(3, 4));
graph[3].add(new Edge(3, 5));

graph[4].add(new Edge(4, 2));
graph[4].add(new Edge(4, 3));
graph[4].add(new Edge(4, 5));

graph[5].add(new Edge(5, 3));
graph[5].add(new Edge(5, 4));
graph[5].add(new Edge(5, 6));
}

public static void dfs(ArrayList<Edge> graph[], int curr, boolean vis[]){
    System.out.print(curr+" ");
    vis[curr] = true;

    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i);

        if(vis[e.dest] == false){
            dfs(graph, e.dest, vis);
        }
    }
}
```

```
public static void main(String[] args) {  
    int v = 7;  
  
    ArrayList<Edge> graph[] = new ArrayList[v];  
    createGraph(graph);  
  
    boolean vis[] = new boolean[v];  
  
    // for disconnected graph  
    for(int i=0; i<v; i++){  
        if(vis[i] == false){  
            dfs(graph, i, vis);  
        }  
    }  
  
    System.out.println();  
}  
}
```

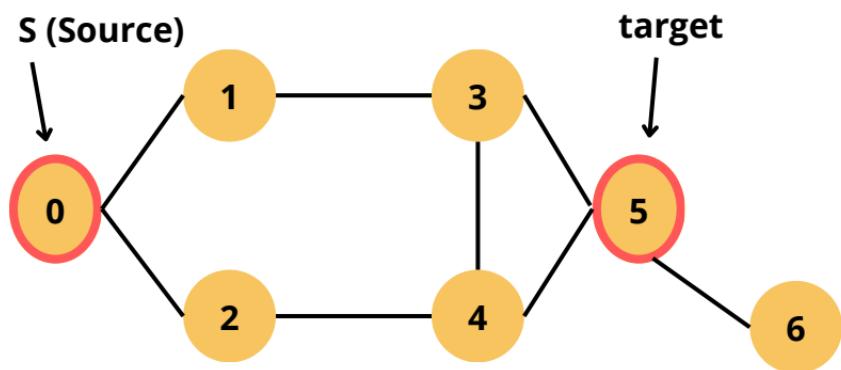
**Output:**

0 1 3 4 2 5 6

- **All paths from Source to Target:**

For given src & tar, print all paths that exist from src to tar/.

src = 0, tar = 5



```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

public class Test{
    static class Edge{
        int src;
        int dest;

        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[])
    {
        // creating empty ArrayList of graph
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<Edge>();
        }

        graph[0].add(new Edge(0, 1));
        graph[0].add(new Edge(0, 2));

        graph[1].add(new Edge(1, 0));
        graph[1].add(new Edge(1, 3));

        graph[2].add(new Edge(2, 0));
        graph[2].add(new Edge(2, 4));
    }
}
```

```
graph[3].add(new Edge(3, 1));
graph[3].add(new Edge(3, 4));
graph[3].add(new Edge(3, 5));

graph[4].add(new Edge(4, 2));
graph[4].add(new Edge(4, 3));
graph[4].add(new Edge(4, 5));

graph[5].add(new Edge(5, 3));
graph[5].add(new Edge(5, 4));
graph[5].add(new Edge(5, 6));
}

// modified dfs
public static void printAllPath(ArrayList<Edge>
graph[], boolean vis[], int curr, String path, int tar){
    if(curr == tar){
        System.out.println(path);
        return;
    }

    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        if(!vis[e.dest]){
            vis[curr] = true;
            printAllPath(graph, vis, e.dest, path+
"+e.dest, tar);
            vis[curr] = false;
        }
    }
}
```

```
public static void main(String[] args) {  
    int v = 7;  
  
    ArrayList<Edge> graph[] = new ArrayList[v];  
    createGraph(graph);  
  
    int src = 0, tar = 5;  
    printAllPath(graph, new boolean[v], src, "0", tar);  
}  
}
```

**Output:**

0 1 3 4 5  
0 1 3 5  
0 2 4 3 5  
0 2 4 5

**Time complexity:**

$O(V^V)$  for printAllPath() function.

- **Cycle Detection in Graph:**

Undirected graph:

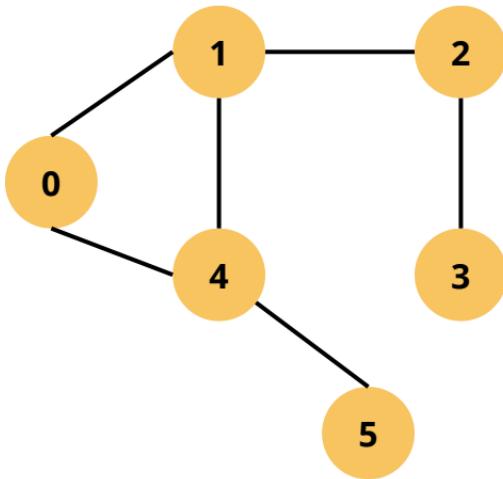
- DFS
- BFS
- DSU (Disjoint Set Union)

Directed graph:

- DFS
- BFS
- Topological Sorting (Kahn's Algorithm)
- Graph coloring

- **Cycle Detection(Undirected Graph):**

Approach : Modified DFS.



```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class Test{
    static int n=0;
    static class Edge{
        int src;
        int dest;

        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[])
    {
        // creating empty ArrayList of graph
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<Edge>();
        }

        graph[0].add(new Edge(0, 1));
        graph[0].add(new Edge(0, 4));

        graph[1].add(new Edge(1, 0));
        graph[1].add(new Edge(1, 2));
        graph[1].add(new Edge(1, 4));
    }
}
```

```
graph[2].add(new Edge(2, 1));
graph[2].add(new Edge(2, 3));

graph[3].add(new Edge(3, 2));

graph[4].add(new Edge(4, 0));
graph[4].add(new Edge(4, 1));
graph[4].add(new Edge(4, 5));

graph[5].add(new Edge(5, 4));

}

public static boolean
isCycleUndirected(ArrayList<Edge> graph[], boolean vis[],
int curr, int par){
    vis[curr] = true;

    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        if(vis[e.dest] && e.dest != par){
            return true;
        } else if(!vis[e.dest]){
            if(isCycleUndirected(graph, vis, e.dest, curr)){
                return true;
            }
        }
    }
}
```

```
        return false;
    }

public static void main(String[] args) {
    int v = 6;

    ArrayList<Edge> graph[] = new ArrayList[v];
    createGraph(graph);

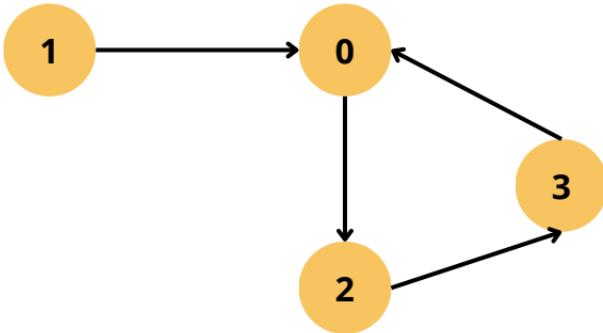
    System.out.println(isCycleUndirected(graph, new
boolean[v], 0, -1));
}
}
```

**Output:**

true

- **Cycle Detection(Directed Graph):**

Approach : Modified DFS.



```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

public class Test{
    static class Edge{
        int src;
        int dest;

        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[])
    {
        // creating empty ArrayList of graph
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<Edge>();
        }

        graph[0].add(new Edge(0, 2));

        graph[1].add(new Edge(1, 0));

        graph[2].add(new Edge(2, 3));

        graph[3].add(new Edge(3, 0));
    }
}
```

```
// modified dfs
public static boolean
isCycleDirected(ArrayList<Edge> graph[], boolean
vis[], int curr, boolean rec[]){
    vis[curr] = true;
    rec[curr] = true;

    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        if(rec[e.dest]){ //cycle
            return true;
        } else if(!vis[e.dest]){
            if (isCycleDirected(graph, vis, e.dest, rec)) {
                return true;
            }
        }
    }

    rec[curr] = false;
    return false;
}

public static void main(String[] args) {
    int v = 4;

    ArrayList<Edge> graph[] = new ArrayList[v];
    createGraph(graph);

    boolean vis[] = new boolean[v];
```

```
for(int i=0; i<v; i++){  
    boolean rec[] = new boolean[v];  
  
    if(!vis[i]){  
        System.out.println(isCycleDirected(graph,  
vis, i, rec));  
    }  
}  
}  
}
```

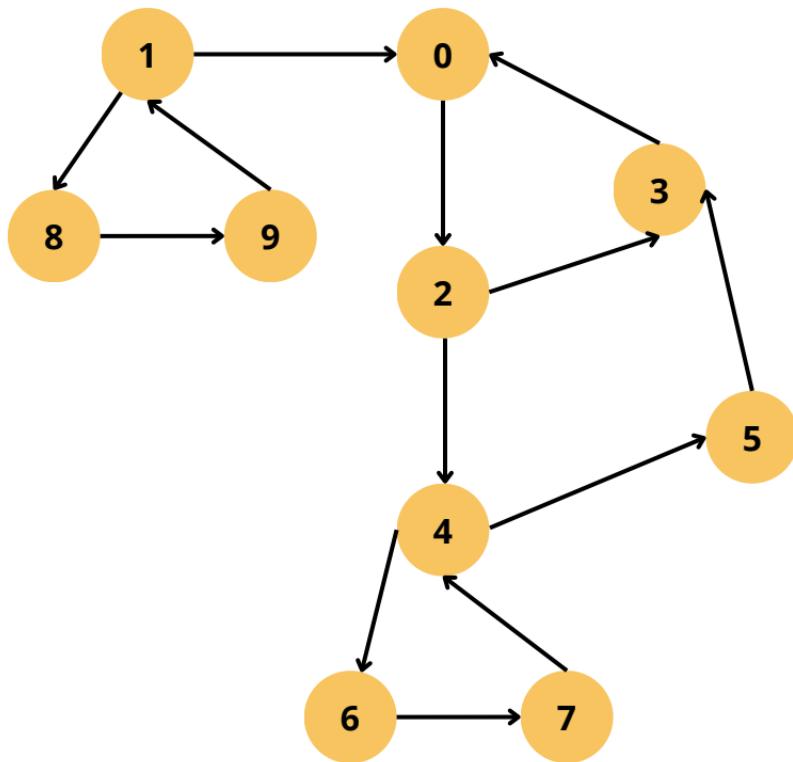
### **Output:**

true

false

- **Cycle Detection(Directed Graph):**

How many cycles for every single vertex are in this graph but the condition is once you visited a graph you can't visit it again.



```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;
```

```
public class Test{
    static int n=0;
    static class Edge{
        int src;
        int dest;

    public Edge(int s, int d){
        this.src = s;
        this.dest = d;
    }
}
```

```
public static void createGraph(ArrayList<Edge> graph[]){
    // creating empty ArrayList of graph
    for(int i=0; i<graph.length; i++){
        graph[i] = new ArrayList<Edge>();
    }
}
```

```
graph[0].add(new Edge(0, 2));
```

```
graph[1].add(new Edge(1, 0));
graph[1].add(new Edge(1, 8));
graph[2].add(new Edge(2, 3));
graph[2].add(new Edge(2, 4));
```

```
graph[3].add(new Edge(3, 0));
```

```
graph[4].add(new Edge(4, 5));
graph[4].add(new Edge(4, 6));

graph[5].add(new Edge(5, 3));

graph[6].add(new Edge(6, 7));

graph[7].add(new Edge(7, 4));

graph[8].add(new Edge(8, 9));

graph[9].add(new Edge(9, 1));

}

// modified dfs
public static int isCycleDirected(ArrayList<Edge> graph[],
boolean vis[], int curr, boolean rec[], int par){
vis[curr] = true;
rec[curr] = true;

for(int i=0; i<graph[curr].size(); i++){
Edge e = graph[curr].get(i);
if(rec[e.dest] && e.dest == par){ //cycle
n++;
} else if(!vis[e.dest]){
isCycleDirected(graph, vis, e.dest, rec, par);
}
}
```

```
rec[curr] = false;
return n;
}

public static void main(String[] args) {
int v = 10;

ArrayList<Edge> graph[] = new ArrayList[v];
createGraph(graph);

boolean vis[] = new boolean[v];

for(int i=0; i<v; i++){
boolean rec[] = new boolean[v];

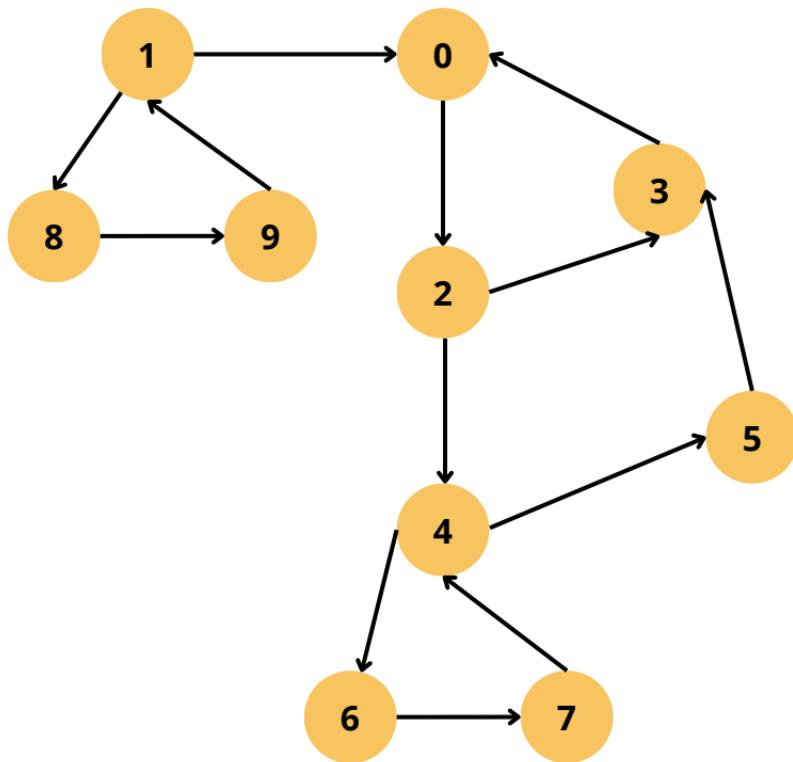
if(!vis[i]){
System.out.println(isCycleDirected(graph, vis, i, rec, i));
n=0;
}
}
}
}
```

## **Output:**

1  
1

- **Cycle Detection(Directed Graph):**

How many cycles for every single vertex are in this graph. ( Double loop not accepted)



```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;
```

```
public class Test{
    static int n=0;
    static class Edge{
        int src;
        int dest;

        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }
}
```

```
public static void createGraph(ArrayList<Edge> graph[])
{
    // creating empty ArrayList of graph
    for(int i=0; i<graph.length; i++){
        graph[i] = new ArrayList<Edge>();
    }

    graph[0].add(new Edge(0, 2));

    graph[1].add(new Edge(1, 0));
    graph[1].add(new Edge(1, 8));
    graph[2].add(new Edge(2, 3));
    graph[2].add(new Edge(2, 4));

    graph[3].add(new Edge(3, 0));
```

```
graph[4].add(new Edge(4, 5));
graph[4].add(new Edge(4, 6));

graph[5].add(new Edge(5, 3));

graph[6].add(new Edge(6, 7));

graph[7].add(new Edge(7, 4));

graph[8].add(new Edge(8, 9));

graph[9].add(new Edge(9, 1));

}

// modified dfs
public static int isCycleDirected(ArrayList<Edge>
graph[], boolean vis[], int curr, int v){
    vis[curr] = true;

    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        if(vis[e.dest] && v == e.dest){ //cycle
            n++;
        } else if(!vis[e.dest]){
            isCycleDirected(graph, vis, e.dest, v);
        }
    }

}
```

```
    vis[curr] = false;
    return n;
}

public static void main(String[] args) {
    int v = 10;

    ArrayList<Edge> graph[] = new ArrayList[v];
    createGraph(graph);

    for(int i=0; i<v; i++){
        boolean vis[] = new boolean[v];

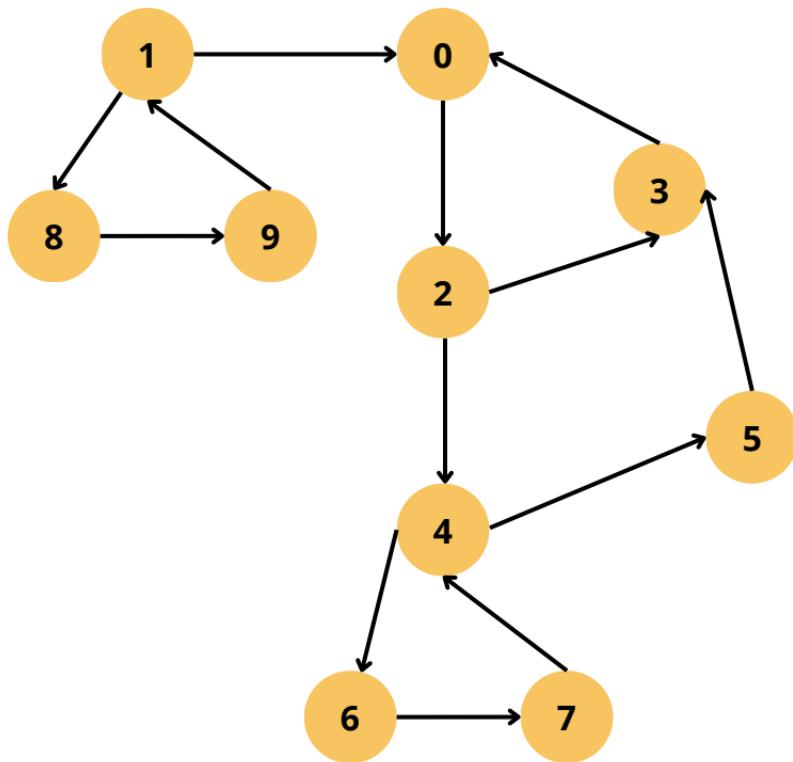
        System.out.println(i+" = "+isCycleDirected(graph,
vis, i, i));
        n=0;
    }
}
}
```

## **Output:**

```
0 = 2
1 = 1
2 = 2
3 = 2
4 = 2
5 = 1
6 = 1
7 = 1
8 = 1
9 = 1
```

- **Cycle Detection(Directed Graph):**

How many cycles for every single vertex are in this graph. ( Double loop accepted and simple vertices multiple time visiting accepted for visiting other vertices)



```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

public class Test{
    static int n=0;
    static class Edge{
        int src;
        int dest;

        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[])
    {
        // creating empty ArrayList of graph
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<Edge>();
        }

        graph[0].add(new Edge(0, 2));

        graph[1].add(new Edge(1, 0));
        graph[1].add(new Edge(1, 8));
        graph[2].add(new Edge(2, 3));
        graph[2].add(new Edge(2, 4));

        graph[3].add(new Edge(3, 0));
    }
}
```

```
graph[4].add(new Edge(4, 5));
graph[4].add(new Edge(4, 6));

graph[5].add(new Edge(5, 3));

graph[6].add(new Edge(6, 7));

graph[7].add(new Edge(7, 4));

graph[8].add(new Edge(8, 9));

graph[9].add(new Edge(9, 1));

}

// modified dfs
public static int isCycleDirected(ArrayList<Edge>
graph[], boolean vis[], int curr, int v){
    vis[curr] = true;

    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        if(vis[e.dest] && v == e.dest){ //cycle
            n++;
        }

        } else if(!vis[e.dest]){
            isCycleDirected(graph, vis, e.dest, v);
        } else {
            for(int j=0; j<graph[e.dest].size(); j++){
                Edge b = graph[e.dest].get(j);
            }
        }
    }
}
```

```
        if(vis[b.dest] && v == b.dest){ //cycle
            n++;
        }
        if(!vis[b.dest]){
            isCycleDirected(graph, vis, b.dest, v);
        }
    }
}

vis[curr] = false;
return n;
}

public static void main(String[] args) {
    int v = 10;

    ArrayList<Edge> graph[] = new ArrayList[v];
    createGraph(graph);

    for(int i=0; i<v; i++){
        boolean vis[] = new boolean[v];

        System.out.println(i+" =
"+isCycleDirected(graph, vis, i, i));
        n=0;
    }
}
}
```

**Output:**

0 = 3

1 = 1

2 = 3

3 = 3

4 = 2

5 = 2

6 = 2

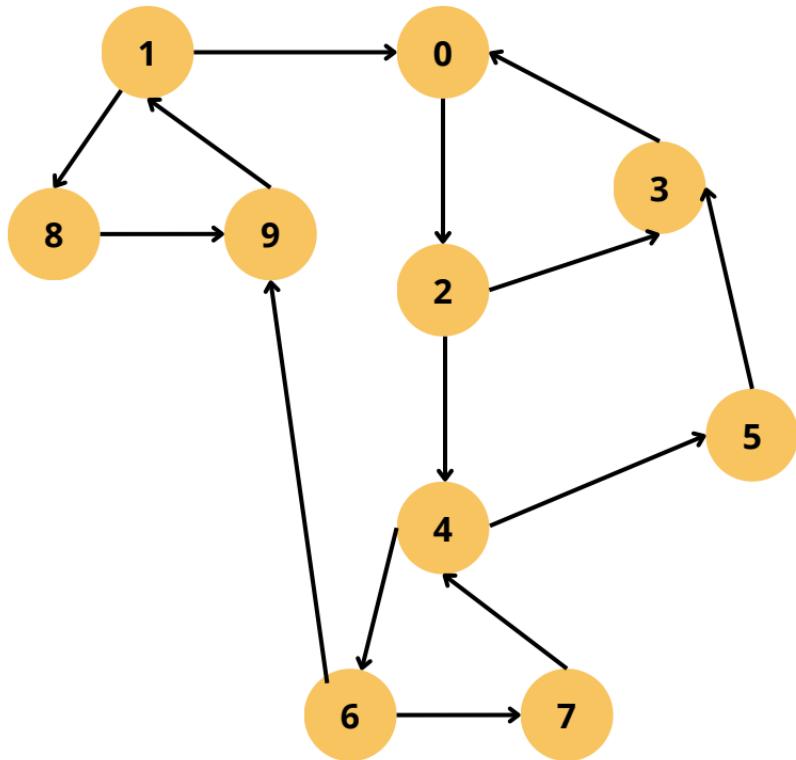
7 = 2

8 = 1

9 = 1

- **Cycle Detection(Directed Graph):**

How many cycles for every single vertex are in this graph. ( Double loop accepted)



```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;

public class Test{
    static int n=0;
    static class Edge{
        int src;
        int dest;

        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[])
    {
        // creating empty ArrayList of graph
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<Edge>();
        }

        graph[0].add(new Edge(0, 2));

        graph[1].add(new Edge(1, 0));
        graph[1].add(new Edge(1, 8));
        graph[2].add(new Edge(2, 3));
        graph[2].add(new Edge(2, 4));

        graph[3].add(new Edge(3, 0));
    }
}
```

```
graph[4].add(new Edge(4, 5));
graph[4].add(new Edge(4, 6));

graph[5].add(new Edge(5, 3));

graph[6].add(new Edge(6, 7));
graph[6].add(new Edge(6, 9));

graph[7].add(new Edge(7, 4));

graph[8].add(new Edge(8, 9));

graph[9].add(new Edge(9, 1));

}

// modified dfs
public static int isCycleDirected(ArrayList<Edge>
graph[], boolean vis[], int curr, int v){
    vis[curr] = true;

    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        if(vis[e.dest] && v == e.dest){ //cycle
            n++;

        } else if(!vis[e.dest]){
            isCycleDirected(graph, vis, e.dest, v);
        } else {
            for(int j=0; j<graph[e.dest].size(); j++){
                Edge b = graph[e.dest].get(j);
            }
        }
    }
}
```

```
        if(vis[b.dest] && v == b.dest){ //cycle
            n++;
        }
        if(!vis[b.dest]){
            isCycleDirected(graph, vis, b.dest, v);
        }
    }

}

vis[curr] = false;
return n;
}

public static void main(String[] args) {
    int v = 10;

    ArrayList<Edge> graph[] = new ArrayList[v];
    createGraph(graph);

    for(int i=0; i<v; i++){
        boolean vis[] = new boolean[v];

        System.out.println(i+" = "+isCycleDirected(graph,
vis, i, i));
        n=0;
    }
}
```

**Output:**

0 = 4

1 = 2

2 = 4

3 = 3

4 = 3

5 = 2

6 = 3

7 = 2

8 = 1

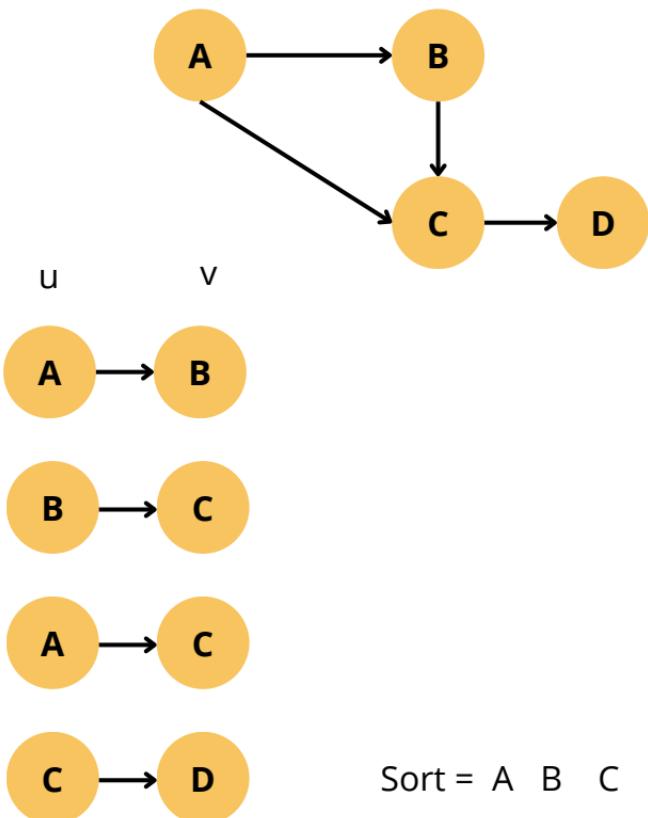
9 = 2

- **Topological Sorting:**

Directed Acyclic Graph(DAG) is a directed graph with no cycles.

Topological sorting is used only for DAGs (not for non-DAGs)

It is linear order of vertices such that every directed edge  $u \rightarrow v$ , the vertex  $u$  comes before  $v$  in the order.



## **Topological sort dependency :**

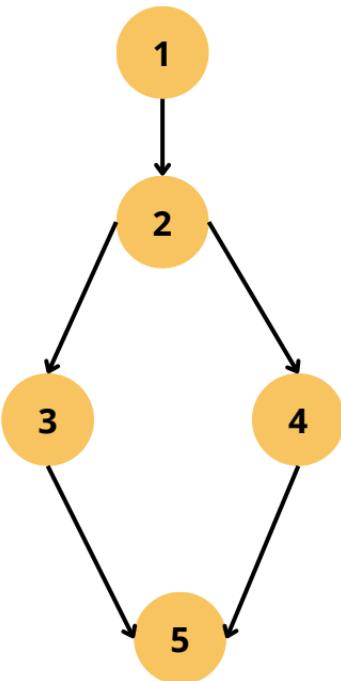
Action 1 - Buy laptop

Action 2 - install OS

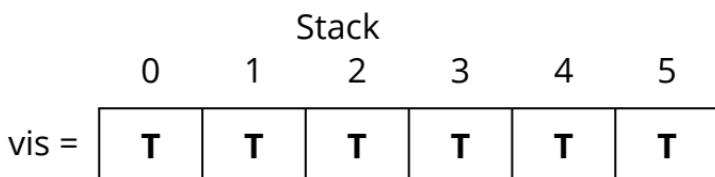
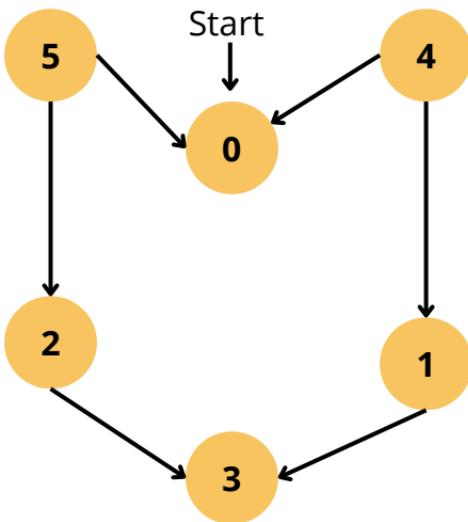
Action 3 - install code editor

Action 4 - install Java

Action 5 - write code



- Topological sort example:



```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class Test{
    static int n=0;
    static class Edge{
        int src;
        int dest;

        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[])
    {
        // creating empty ArrayList of graph
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<Edge>();
        }

        graph[2].add(new Edge(2, 3));

        graph[3].add(new Edge(3, 1));

        graph[4].add(new Edge(4, 0));
        graph[4].add(new Edge(4, 1));
    }
}
```

```
graph[5].add(new Edge(5, 0));
graph[5].add(new Edge(5, 2));

}
```

```
public static void topSortUtil(ArrayList<Edge> graph[],
int curr, boolean vis[], Stack<Integer> stack){
    vis[curr] = true;

    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i);

        if(!vis[e.dest]){
            topSortUtil(graph, e.dest, vis, stack);
        }
    }

    stack.push(curr);
}
```

```
public static void topSort(ArrayList<Edge> graph[], int v)
{
    boolean vis[] = new boolean[v];
    Stack<Integer> stack = new Stack<>();

    for(int i=0; i<v; i++){
        if(!vis[i]){
            topSortUtil(graph, i, vis, stack);
        }
    }
}
```

```
while(!stack.isEmpty()){
    System.out.print(stack.pop()+" ");
}
}

public static void main(String[] args) {
    int v = 6;

    ArrayList<Edge> graph[] = new ArrayList[v];
    createGraph(graph);

    topSort(graph, v);
}
}
```

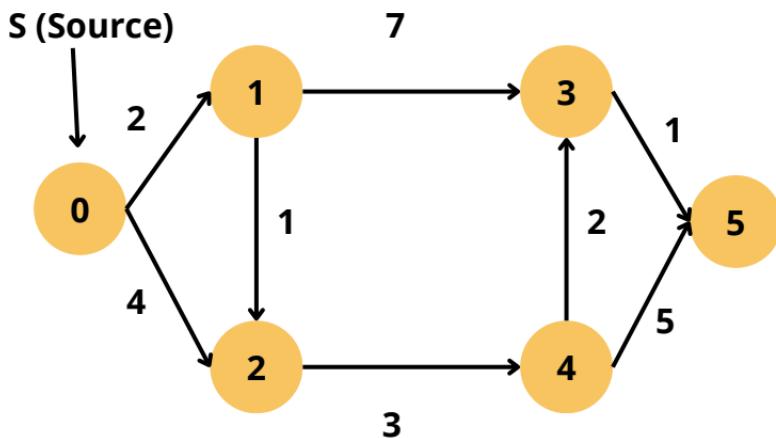
**Output:**

5 4 2 3 1 0

- Single Source Shortest Path(SSSP) **Dijkstra's Algorithm:**

Shortest distance from the source to all vertices.  
Here, we will use BFS of Greedy Algorithm.

Dijkstra = BFS + Priority Queue + Array.  
It will work for both undirected and directed graph.  
If any edge weight is negative then it will not work.



i	Vis	0	1	2	3	4	5
0	0	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
1	1		2	4	$\infty$	$\infty$	$\infty$
2	2			3	9	$\infty$	$\infty$
3	4				9	6	$\infty$
4	3				8		11
5	5						9

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Queue;

public class Test{
    static class Edge{
        int src;
        int dest;
        int wt;
        public Edge(int s, int d, int w){
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[]){
        // creating empty ArrayList of graph
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<Edge>();
        }

        graph[0].add(new Edge(0, 1, 2));
        graph[0].add(new Edge(0, 2, 4));

        graph[1].add(new Edge(1, 3, 7));
        graph[1].add(new Edge(1, 2, 1));

        graph[2].add(new Edge(2, 4, 3));
    }
}
```

```
graph[3].add(new Edge(3, 5, 1));

graph[4].add(new Edge(4, 3, 2));
graph[4].add(new Edge(4, 5, 5));

}

public static class Pair implements Comparable<Pair>{
    int node;
    int dist;

    public Pair(int n, int d){
        this.node = n;
        this.dist = d;
    }

    @Override
    public int compareTo(Pair p2){
        return this.dist - p2.dist; // ascending order
    }
}

// modified bfs
public static void dijkstra(ArrayList<Edge> graph[], int src, int V){
    PriorityQueue<Pair> pq = new PriorityQueue<>(); // <---
    O(ElogV)
    int dist[] = new int[V];

    for(int i=0; i<V; i++){
        if(i != src){


```

```
    dist[i] = Integer.MAX_VALUE;
}
}

boolean vis[] = new boolean[V];

pq.add(new Pair(0, 0));

while ((!pq.isEmpty())) {
    Pair curr = pq.remove(); // shortest
    if(!vis[curr.node]){
        vis[curr.node] = true;

        for(int i=0; i<graph[curr.node].size(); i++){
            Edge e = graph[curr.node].get(i);
            int u = e.src;
            int v = e.dest;
            if(dist[u] + e.wt < dist[v]){ // relaxation
                dist[v] = dist[u] + e.wt;
                pq.add(new Pair(v,dist[v]));
            }
        }
    }
}

// printing distance
for(int i=0; i<V; i++){
    System.out.print("0 to "+i+": \n-----\n");
    System.out.print("Total distance --> "+dist[i]+"\n\n");
}
```

```
}

public static void main(String[] args) {
    int v = 6;

    ArrayList<Edge> graph[] = new ArrayList[v];
    createGraph(graph);

    dijkstra(graph, 0, v);
}
}
```

## **Output:**

0 to 0:  
-----

Total distance --> 0

0 to 1:  
-----

Total distance --> 2

0 to 2:  
-----

Total distance --> 3

0 to 3:  
-----

Total distance --> 8

0 to 4:  
-----

Total distance --> 6

0 to 5:  
-----

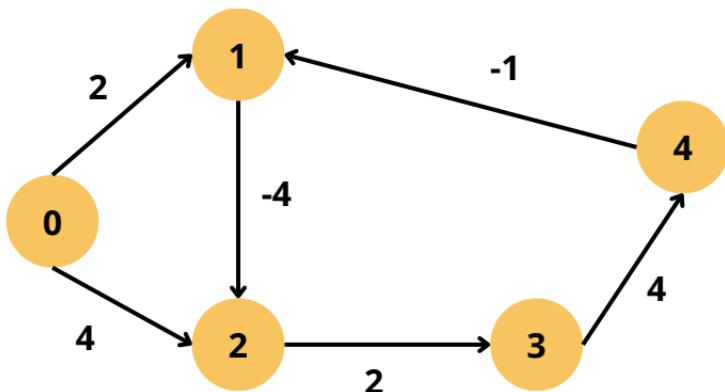
Total distance --> 9

- **Bellman Ford Algorithm:**

Shortest distance from the source to all vertices.

It works in both positive and negative edges.

Dynamic programming approach used here.



1st iteration:

i	Vis	0	1	2	3	4
0	T	0	$\infty$	$\infty$	$\infty$	$\infty$
1	T	0	2	4	$\infty$	$\infty$
2	T	0	2	-2	$\infty$	$\infty$
3	T	0	2	-2	0	$\infty$
4	T	0	2	-2	0	4
5	T	0	2	-2	0	4

Here, we need to continue this iteration  $V-1$  ( $V =$  Vertices) times.

In the upper graph  $V = 5$

So, number of iteration,  $n = V - 1 = 5 - 1 = 4$

2nd iteration:

i	Vis	0	1	2	3	4
0	T	0	2	-2	0	4
1	T	0	2	-2	0	4
2	T	0	2	-2	0	4
3	T	0	2	-2	0	4
4	T	0	2	-2	0	4
5	T	0	2	-2	0	4

3rd iteration:

i	Vis	0	1	2	3	4
0	T	0	2	-2	0	4
1	T	0	2	-2	0	4
2	T	0	2	-2	0	4
3	T	0	2	-2	0	4
4	T	0	2	-2	0	4
5	T	0	2	-2	0	4

4th iteration:

i	Vis	0	1	2	3	4
0	T	0	2	-2	0	4
1	T	0	2	-2	0	4
2	T	0	2	-2	0	4
3	T	0	2	-2	0	4
4	T	0	2	-2	0	4
5	T	0	2	-2	0	4

**Time complexity :**

O(V.E)

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Queue;

public class Test{
    static class Edge{
        int src;
        int dest;
        int wt;

        public Edge(int s, int d, int w){
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[])
    {
        // creating empty ArrayList of graph
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<Edge>();
        }

        graph[0].add(new Edge(0, 1, 2));
        graph[0].add(new Edge(0, 2, 4));

        graph[1].add(new Edge(1, 2, -4));

        graph[2].add(new Edge(2, 3, 2));
    }
}
```

```
graph[3].add(new Edge(3, 4, 4));

graph[4].add(new Edge(4, 1, -1));

}

// modified BFS
public static void bellmanFord(ArrayList<Edge> graph[],
int src, int V){
    int dist[] = new int[V];

    for(int i=0; i<V; i++){
        if(i != src){
            dist[i] = Integer.MAX_VALUE;
        }
    }

    for(int k=0; k<V-1; k++){ // O(V)
        // O(E)
        for(int i=0; i<V; i++){
            for(int j=0; j<graph[i].size(); j++){
                Edge e = graph[i].get(j);
                int u = e.src;
                int v = e.dest;

                if(dist[u]+e.wt < dist[v] && dist[u] !=
Integer.MAX_VALUE){
                    dist[v] = dist[u] + e.wt;
                }
            }
        }
    }
}
```

```
        for(int i=0; i<dist.length; i++){
            System.out.print(dist[i]+" ");
        }
        System.out.println();
    }

public static void main(String[] args) {
    int v = 5;

    ArrayList<Edge> graph[] = new ArrayList[v];
    createGraph(graph);

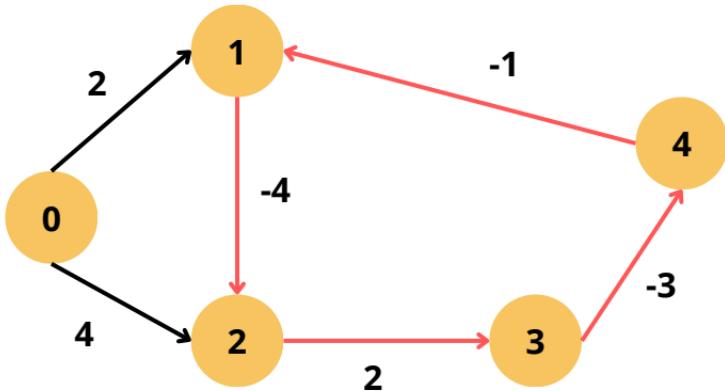
    bellmanFord(graph, 0, v);
}
}
```

**Output:**

0 2 -2 0 4

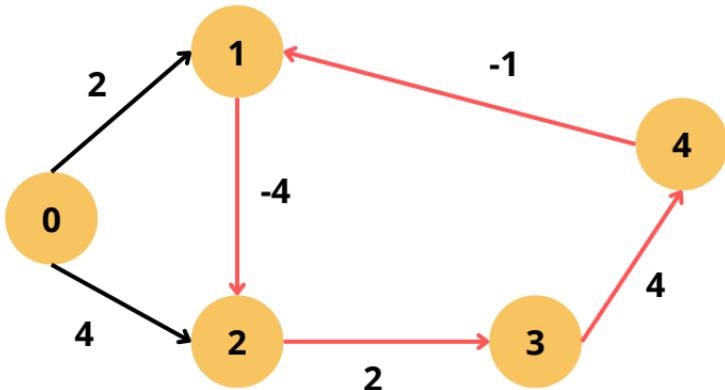
## Note:

Bellman Ford Algorithm doesn't work for Negative Weight Cycles.



$$-4 + 2 + (-3) + (-1) = -6$$

So, in this graph Bellman Ford Algorithm will be failed.



$$-4 + 2 + 4 + (-1) = 1$$

So, in this graph Bellman Ford Algorithm will work.

- **Minimum Spanning Tree (MST) :**

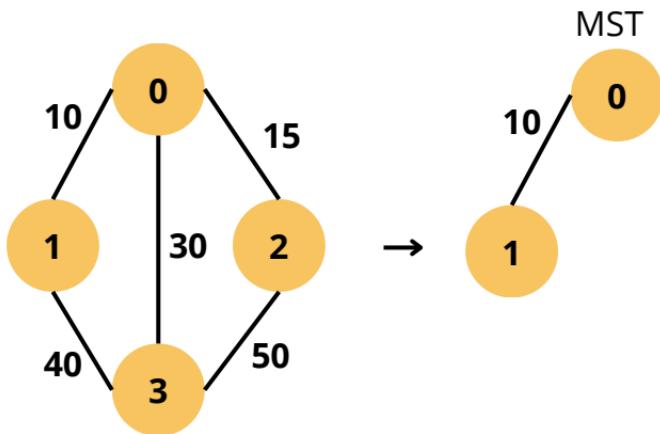
A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight.

Characteristics of MST:

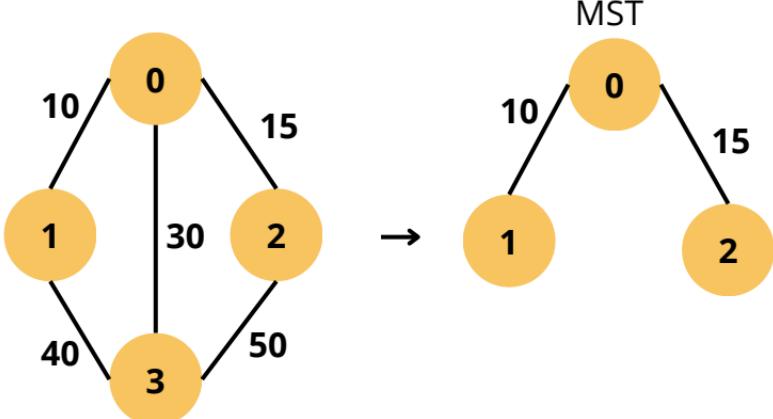
- Subgraph(MST)
- Undirected
- Vertices
- Connected
- No Cycles
- Edge weighted(Minimum)

A MST can be grown from the current spanning tree by adding the nearest vertex and the edge connecting the nearest vertex to the MST. (Prim's algorithm)□

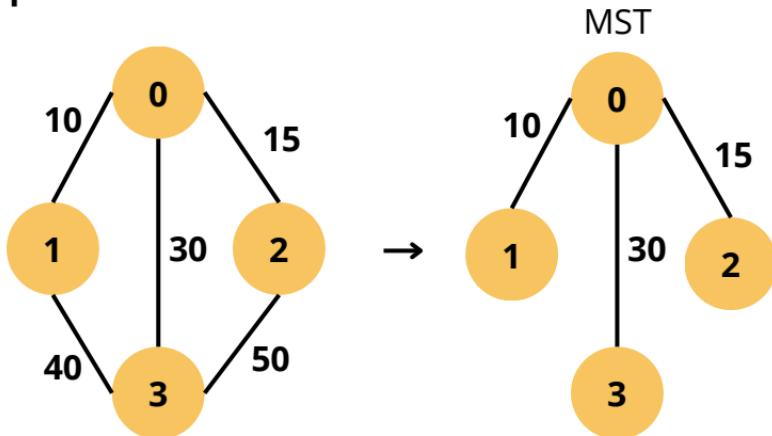
**Step - 1:**



**Step - 2:**



### Step - 3:



$$\text{Cost} = 0 + 15 + 10 + 30 = 55$$

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Queue;

public class Test{
    static class Edge{
        int src;
        int dest;
        int wt;
        public Edge(int s, int d, int w){
            this.src = s;
            this.dest = d;
            this.wt = w;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[])
    {
        // creating empty ArrayList of graph
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<Edge>();
        }

        graph[0].add(new Edge(0, 1, 10));
        graph[0].add(new Edge(0, 2, 15));
        graph[0].add(new Edge(0, 3, 30));

        graph[1].add(new Edge(1, 0, 10));
        graph[1].add(new Edge(1, 3, 40));
    }
}
```

```
graph[2].add(new Edge(2, 0, 15));
graph[2].add(new Edge(2, 3, 50));

graph[3].add(new Edge(3, 1, 40));
graph[3].add(new Edge(3, 2, 50));

}

public static class Pair implements Comparable<Pair> {
    int node;
    int cost;

    public Pair(int n, int c){
        this.node = n;
        this.cost = c;
    }

    @Override
    public int compareTo(Pair p2){
        return this.cost - p2.cost; // ascending
    }
}

// modified BFS
public static void primsAlgo(ArrayList<Edge> graph[], int V){ // ElogE
    PriorityQueue<Pair> pq = new PriorityQueue<>(); // non-mst
    boolean vis[] = new boolean[V]; // mst
    pq.add(new Pair(0, 0));

    int mstCost = 0;
```

```
while(!pq.isEmpty()){
    Pair curr = pq.remove();
    if(!vis[curr.node]){
        vis[curr.node] = true;
        mstCost += curr.cost;

        for(int i=0; i<graph[curr.node].size(); i++){
            Edge e = graph[curr.node].get(i);
            if(!vis[e.dest]){
                pq.add(new Pair(e.dest, e.wt));
            }
        }
    }
}

System.out.println("min cost of mst = "+mstCost);
}

public static void main(String[] args) {
    int v = 5;

    ArrayList<Edge> graph[] = new ArrayList[v];
    createGraph(graph);

    primsAlgo(graph, v);
}
}
```

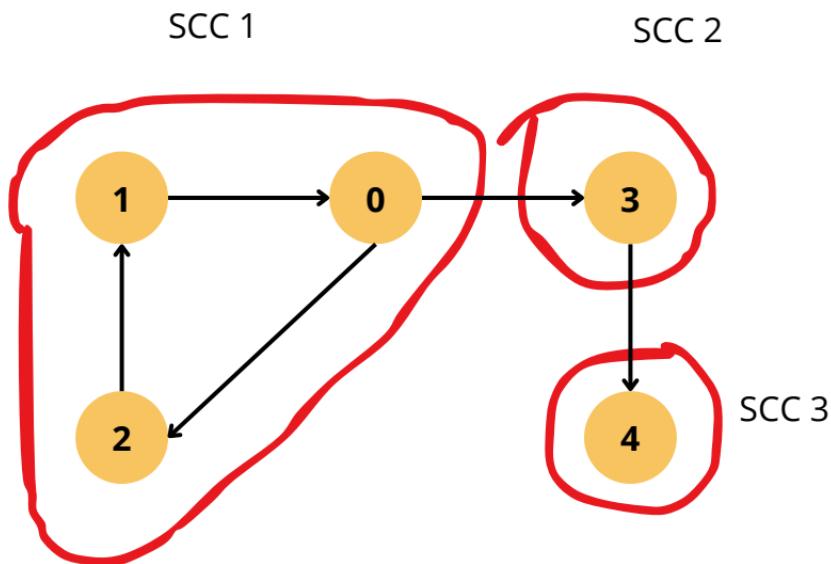
## **Output:**

min cost of mst = 55

**Time complexity;**  
 $E \log E$

- **Strongly Connected Component:**

SCC is a component in which we can reach every vertex of the component from every other vertex in that component.



0 1 2  $\rightarrow$  SCC 1

3  $\rightarrow$  SCC 2

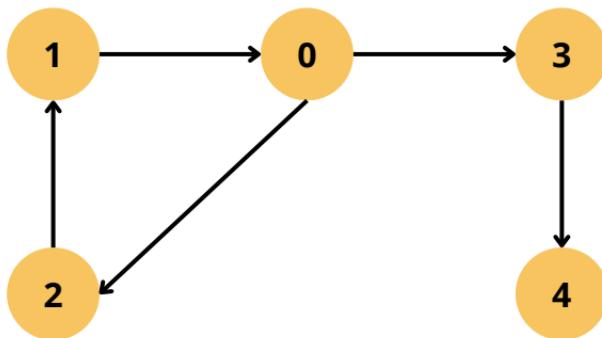
4  $\rightarrow$  SCC 3

Here, we will find SCC using “Kosaraju’s Algorithm”.

## Kosaraju's Algorithm:

Step:

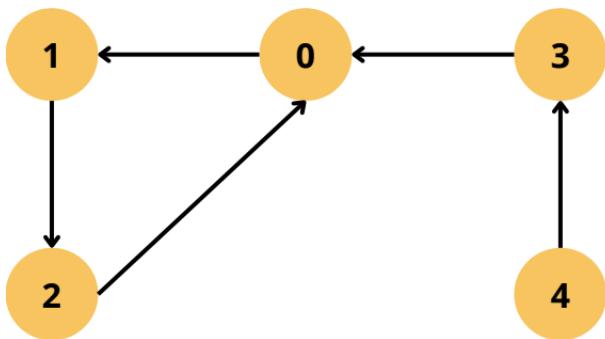
1. Get nodes in the stack using topological sort.  
 $O(V+E)$
2. Transpose the graph.  $O(V+E)$
3. Do DFS according to stack nodes on the transpose graph.  $O(V+E)$



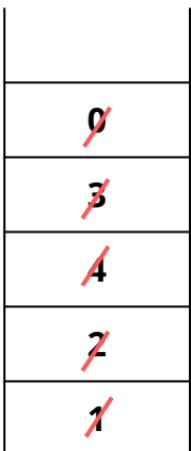
Step - 1:



## Step - 2:



## Step - 3:



0 1 2 --> SCC 1

3 --> SCC 2

4 --> SCC 3

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Stack;

public class Test{
    static class Edge{
        int src;
        int dest;

        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[])
    {
        // creating empty ArrayList of graph
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<Edge>();
        }

        graph[0].add(new Edge(0, 2));
        graph[0].add(new Edge(0, 3));

        graph[1].add(new Edge(1, 0));

        graph[2].add(new Edge(2, 1));

        graph[3].add(new Edge(3, 4));
    }
}
```

```
}
```

```
public static void topSort(ArrayList<Edge> graph[], int curr, boolean vis[], Stack<Integer> stack){
```

```
    vis[curr] = true;
```

```
    for(int i=0; i<graph[curr].size(); i++){
```

```
        Edge e = graph[curr].get(i);
```

```
        if(!vis[e.dest]){
```

```
            topSort(graph, e.dest, vis, stack);
```

```
        }
```

```
}
```

```
    stack.push(curr);
```

```
}
```

```
public static void dfs(ArrayList<Edge> graph[], int curr, boolean vis[]){
```

```
    vis[curr] = true;
```

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

```
    for(int i=0; i<graph[curr].size(); i++){
```

```
        Edge e = graph[curr].get(i);
```

```
        if(!vis[e.dest]){
```

```
            dfs(graph, e.dest, vis);
```

```
        }
```

```
}
```

```
}
```

```
public static void KosarajuAlgo(ArrayList<Edge>
graph[], int V){
    // Step1
    Stack<Integer> s = new Stack<>();
    boolean vis[] = new boolean[V];
    for(int i=0; i<V; i++){
        if(!vis[i]){
            topSort(graph, i, vis, s);
        }
    }

    // Step2
    ArrayList<Edge> transpose[] = new ArrayList[V];
    for(int i=0; i<graph.length; i++){
        transpose[i] = new ArrayList<Edge>();
        vis[i] = false;
    }

    for(int i=0; i<V; i++){
        for(int j=0; j<graph[i].size(); j++){
            Edge e = graph[i].get(j); // e.src(i) -> e.dest
            transpose[e.dest].add(new Edge(e.dest, e.src));
        }
    }

    // Step3
    while (!s.isEmpty()) {
        int curr = s.pop();
        if(!vis[curr]){
            dfs(transpose, curr, vis);
            System.out.println();
        }
    }
}
```

```
        }
    }
}

public static void main(String[] args) {
    int v = 5;

    ArrayList<Edge> graph[] = new ArrayList[v];
    createGraph(graph);

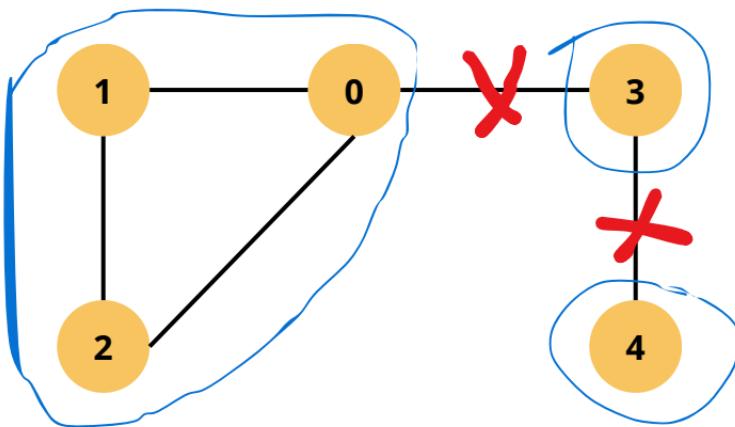
    KosarajuAlgo(graph, v);
}
}
```

### **Output:**

0 1 2  
3  
4

- **Bridge in Graph:**

Bridge is an edge whose deletion increases the graph's number of connected components.



Here, we will use Tarjan's Algorithm for finding bridges.

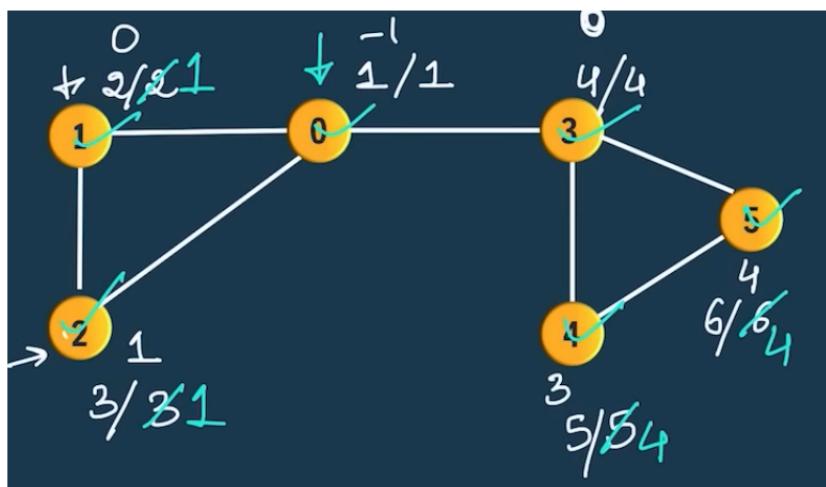
## Tarjan's Algorithm:

Algorithm --> DFS

1.  $dt[ ]$

2.  $low[ ]$

lowest distance time of all neighbors.



```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Stack;

public class Test{
    static class Edge{
        int src;
        int dest;

        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[])
    {
        // creating empty ArrayList of graph
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<Edge>();
        }

        graph[0].add(new Edge(0, 1));
        graph[0].add(new Edge(0, 2));
        graph[0].add(new Edge(0, 3));

        graph[1].add(new Edge(1, 0));
        graph[1].add(new Edge(1, 2));
    }
}
```

```
graph[2].add(new Edge(2, 0));
graph[2].add(new Edge(2, 1));

graph[3].add(new Edge(3, 0));
graph[3].add(new Edge(3, 4));
graph[3].add(new Edge(3, 5));

graph[4].add(new Edge(4, 3));
graph[4].add(new Edge(4, 5));

graph[5].add(new Edge(5, 3));
graph[5].add(new Edge(5, 4));

}

public static void dfs(ArrayList<Edge> graph[], int curr,
boolean vis[], int dt[], int low[], int time, int par){
    vis[curr] = true;
    dt[curr] = low[curr] = ++time;

    for(int i=0; i<graph[curr].size(); i++){
        Edge e = graph[curr].get(i);
        if(e.dest == par){
            continue;
        } else if(!vis[e.dest]){
            dfs(graph, e.dest, vis, dt, low, time, curr);
            low[curr] = Math.min(low[curr], low[e.dest]);
            if(dt[curr] < low[e.dest]){
                System.out.println("Bridge is : "+ curr + "---"+
e.dest);
            }
        }
    }
}
```

```
        } else {
            low[curr] = Math.min(low[curr], dt[e.dest]);
        }
    }
}

public static void getBridge(ArrayList<Edge>
graph[], int V){
    int dt[] = new int[V];
    int low[] = new int[V];
    int time = 0;
    boolean vis[] = new boolean[V];

    for(int i=0; i<V; i++){
        if(!vis[i]){
            dfs(graph, i, vis, dt, low, time, -1);
        }
    }
}

public static void main(String[] args) {
    int v = 6;

    ArrayList<Edge> graph[] = new ArrayList[v];
    createGraph(graph);

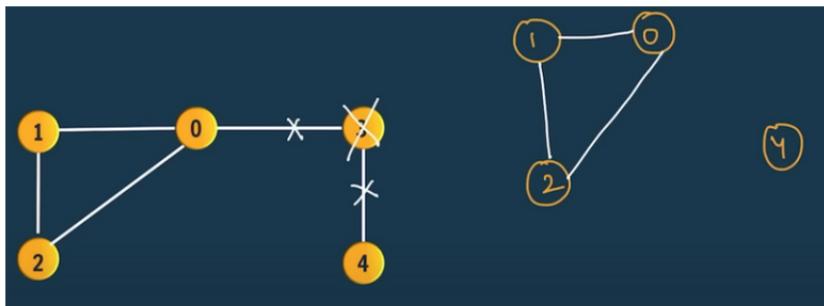
    getBridge(graph, v);
}
}
```

**Output:**

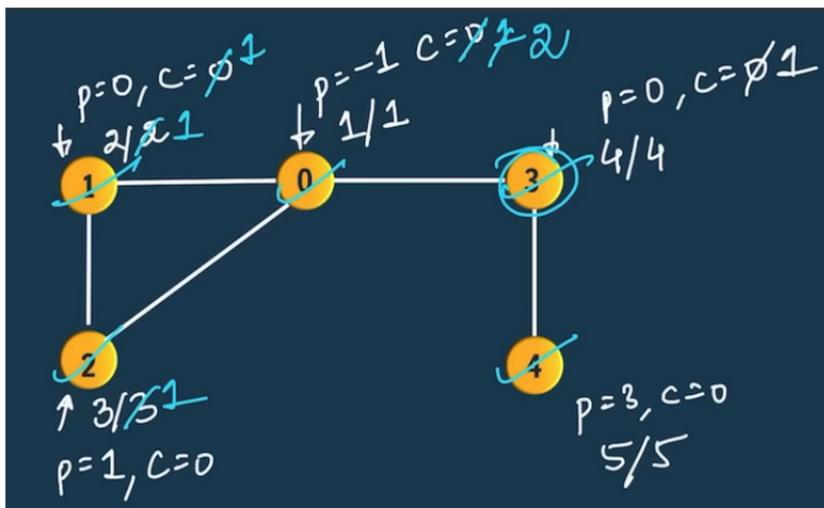
Bridge is : 0---3

- **Articulation Point:**

A vertex in an undirected connected graph is an articulation point (or cut vertex) if removing it (and edges through it) disconnects the graph.



Here, we will use **Tarjan's Algorithm** for finding articulation point.



```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Stack;

public class Test{
    static class Edge{
        int src;
        int dest;

        public Edge(int s, int d){
            this.src = s;
            this.dest = d;
        }
    }

    public static void createGraph(ArrayList<Edge> graph[])
    {
        // creating empty ArrayList of graph
        for(int i=0; i<graph.length; i++){
            graph[i] = new ArrayList<Edge>();
        }

        graph[0].add(new Edge(0, 1));
        graph[0].add(new Edge(0, 2));
        graph[0].add(new Edge(0, 3));

        graph[1].add(new Edge(1, 0));
        graph[1].add(new Edge(1, 2));

        graph[2].add(new Edge(2, 0));
    }
}
```

```
graph[2].add(new Edge(2, 1));  
  
graph[3].add(new Edge(3, 0));  
graph[3].add(new Edge(3, 4));  
  
graph[4].add(new Edge(4, 3));  
  
}  
  
public static void dfs(ArrayList<Edge> graph[], int curr,  
int par, int dt[], int low[], boolean vis[], int time, boolean  
ap[]){  
    vis[curr] = true;  
    dt[curr] = low[curr] = ++time;  
    int children = 0;  
  
    for(int i=0; i<graph[curr].size(); i++){  
        Edge e = graph[curr].get(i);  
        int neigh = e.dest;  
  
        if(par == neigh){  
            continue;  
        } else if(vis[neigh]){  
            low[curr] = Math.min(low[curr], dt[neigh]);  
        } else {  
            dfs(graph, neigh, curr, dt, low, vis, time, ap);  
            low[curr] = Math.min(low[curr], low[neigh]);  
            if(dt[curr] <= low[neigh] && par != -1){  
                ap[curr] = true;  
            }  
            children++;  
        }  
    }  
}
```

```
        }
    }

    if(par == -1 && children > 1){
        ap[curr] = true;
    }
}

// O(V+E)
public static void getAP(ArrayList<Edge> graph[], int V){
    int dt[] = new int[V];
    int low[] = new int[V];
    int time = 0;
    boolean vis[] = new boolean[V];
    boolean ap[] = new boolean[V];

    for(int i=0; i<V; i++){
        if(!vis[i]){
            dfs(graph, i, -1, dt, low, vis, time, ap);
        }
    }

    for(int i=0; i<V; i++){
        if(ap[i]){
            System.out.println("AP : "+i);
        }
    }
}

public static void main(String[] args) {
    int v = 5;
```

```
ArrayList<Edge> graph[] = new ArrayList[v];
createGraph(graph);

    getAP(graph, v);
}

}
```

**Output:**

AP : 0

AP : 3

**Time complexity:**

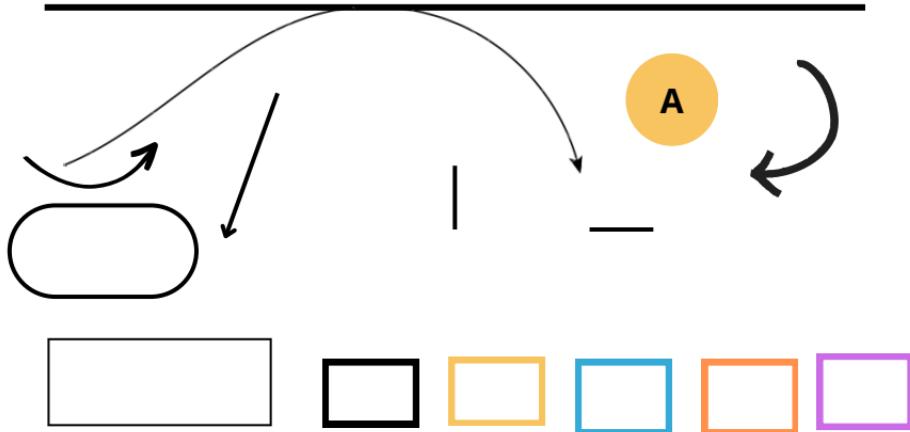
$O(V+E)$

## Operator Precedence and Associativity:

---

- Make a program that will show String palindrome.


---



Output:

Here,

Jim



- What is the output of the following Java program fragment:

Operator



JAVA  
(FIFTH PART)  
T.I.M. HAMIM

OBZTRON  
PROKASHONI