

The background of the slide features a complex, abstract network of glowing blue and purple lines and nodes, resembling a data structure or a neural network, set against a dark, textured background.

Data Structure

Sep23 : Day 7

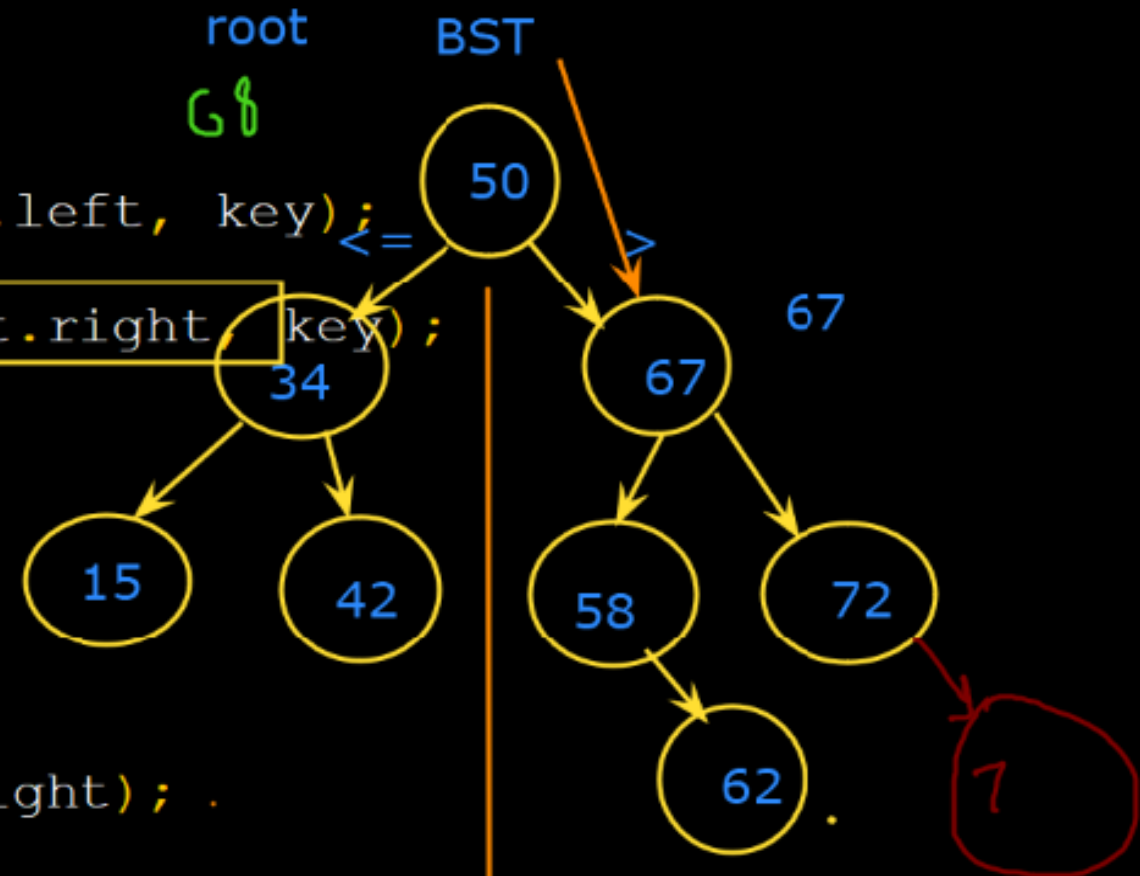
Kiran Waghmare
CDAC Mumbai

```

if(root == null)
    return root;
if(key < root.data)
    root.left = deletedata(root.left, key);
else if(key > root.data)
    root.right = deletedata(root.right, key);
else{
    if(root.left == null)
        return root.right;
    else if(root.right == null)
        return root.left;

    //case 3
    root.data = minvalue(root.right);
    root.right = deletedata(root.right, root.data);
}
return root;
}

```



15 34 42 50 58 62 67 72

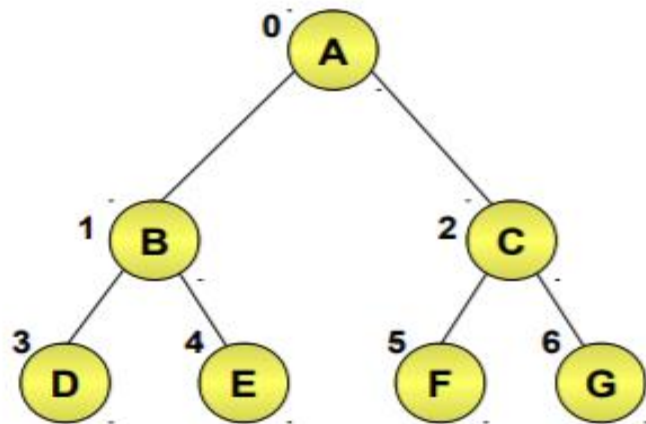
Heap



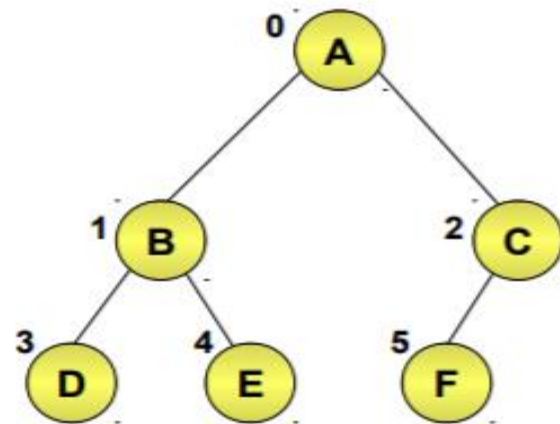
Defining Binary Trees (Contd.)

◆ Complete binary tree:

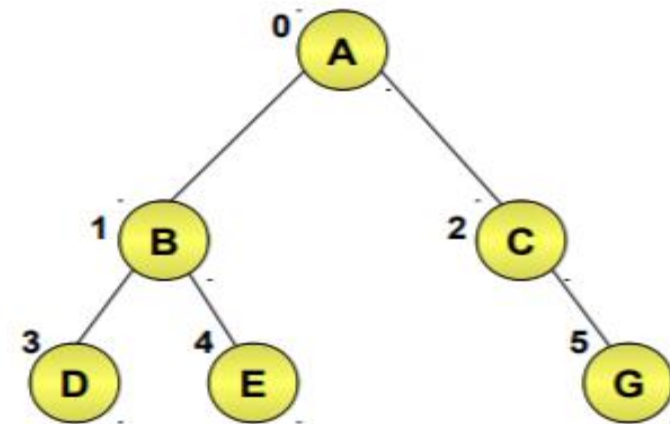
- ◆ A binary tree with n nodes and depth d whose nodes correspond to the nodes numbered from 0 to $n - 1$ in the full binary tree of depth k .



Full Binary Tree



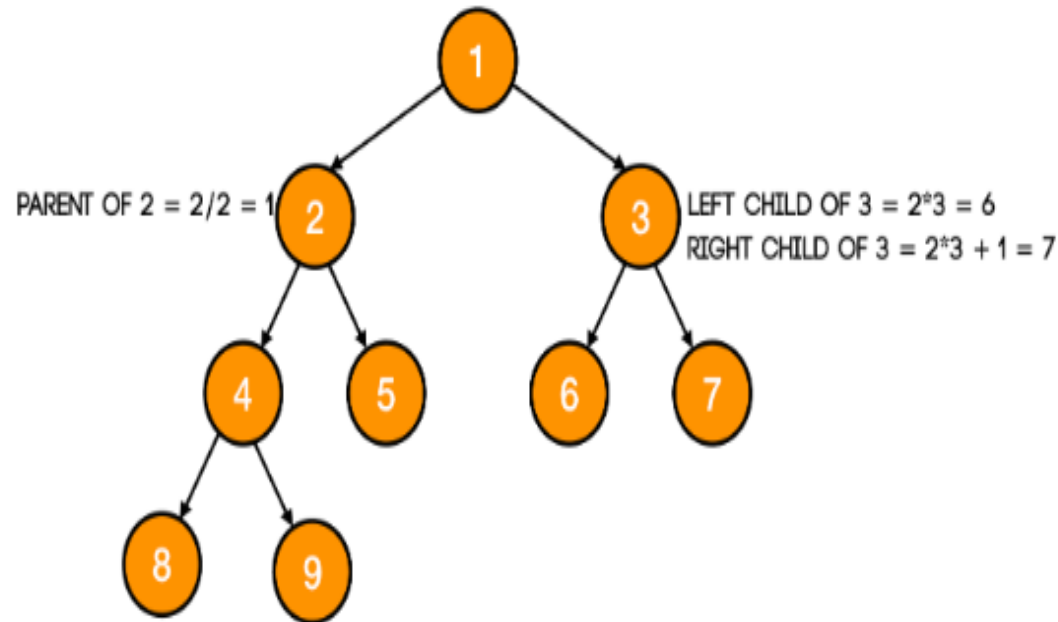
Complete Binary Tree



Incomplete Binary Tree

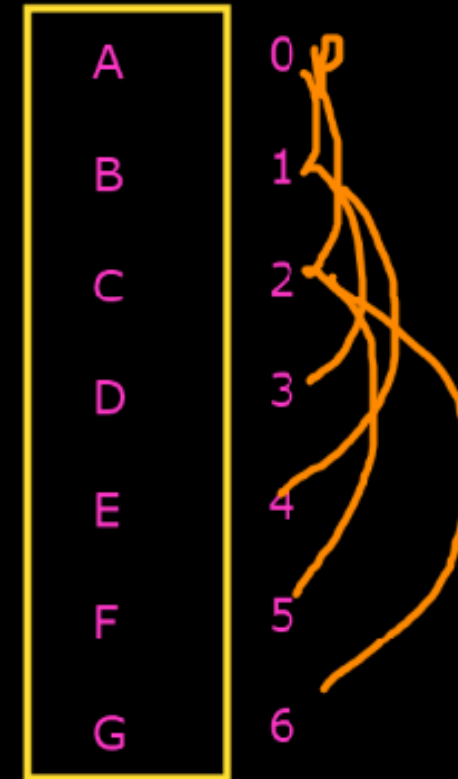
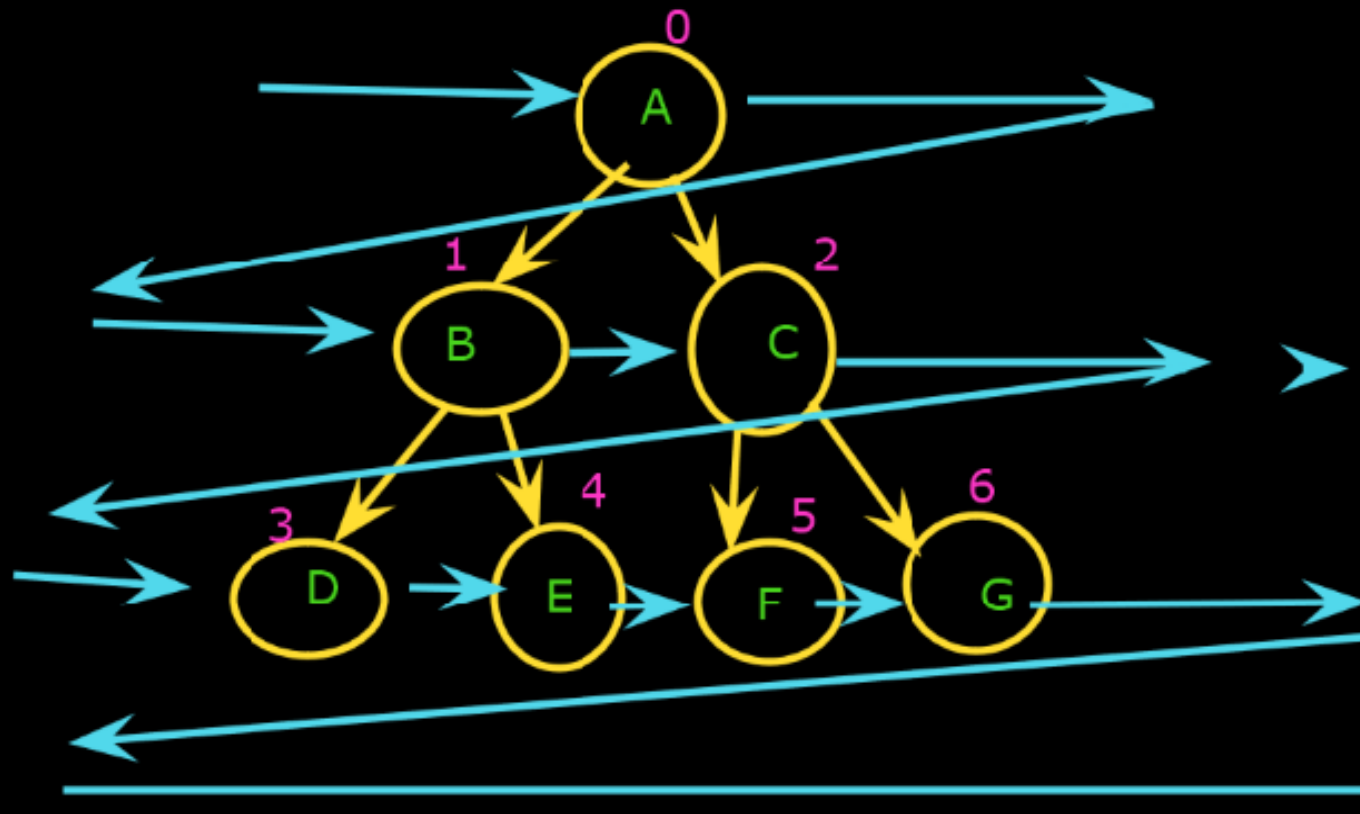
A complete binary tree also holds some important properties. So, let's look at them.

- The **parent of node i** is $\lfloor \frac{i}{2} \rfloor$. For example, the parent of node 4 is 2 and the parent of node 5 is also 2.
- The **left child of node i** is $2i$.
- The **right child of node i** is $2i + 1$



Binary Tree:

A binary tree is a tree in which every node has at most two children.
0,1,2,2



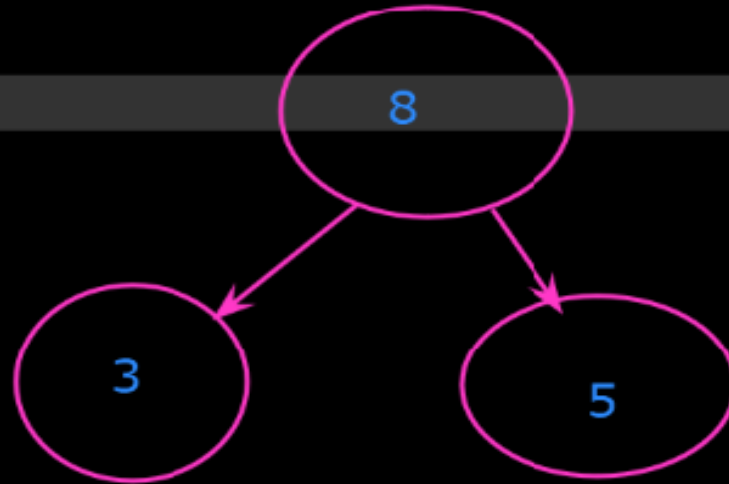
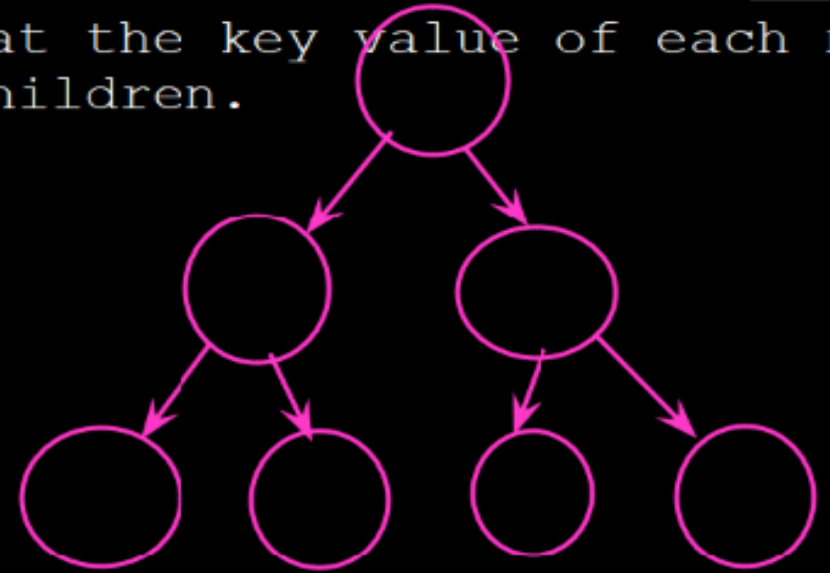
Array Representation of Binary Tree

Definition:

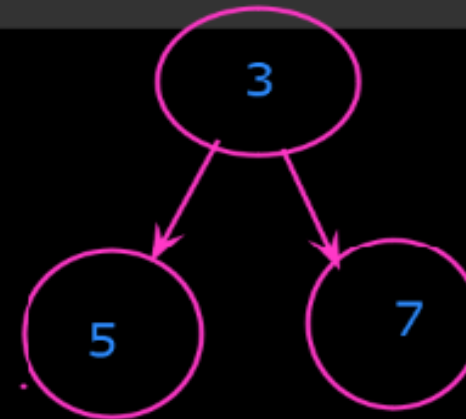
-a special form of complete binary tree that the key value of each node is smaller (larger) than the key value of its children.

Types of heap:

1. Max-Heap: root node has largest value
2. Min-Heap: root node has smallest value



Max Heap



Min Heap

Heap

Heap

- **Definition in Data Structure**

- **Heap:** A special form of **complete binary tree** that key value of each node is no smaller (larger) than the key value of its children (if any).

- **Max-Heap: root node has the largest key.**

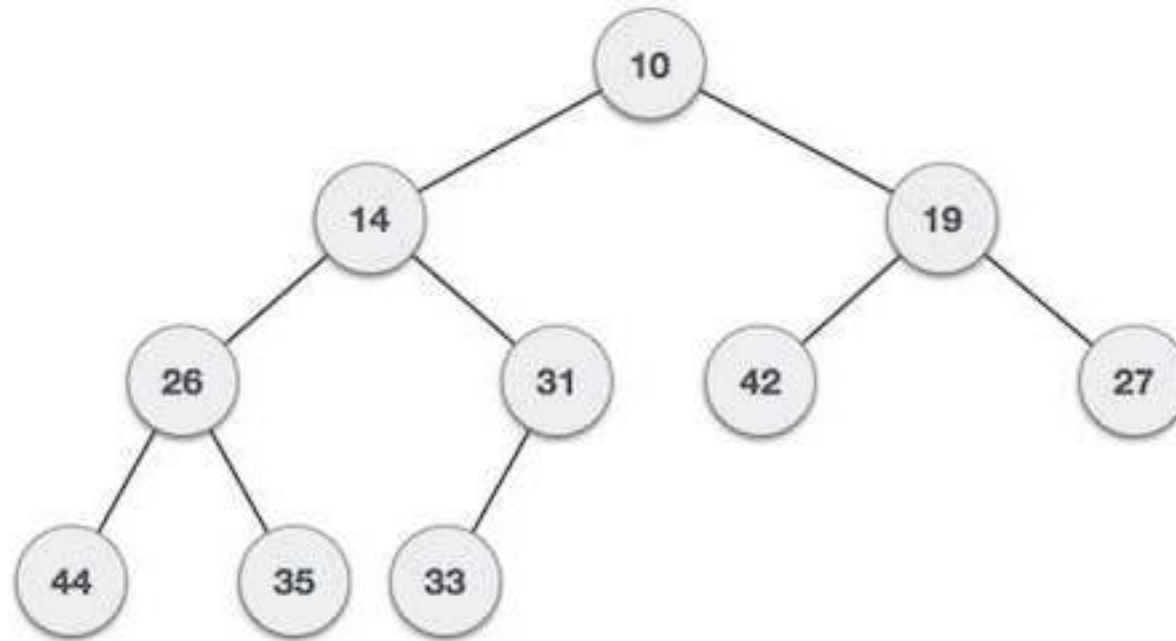
- A **max tree** is a tree in which the key value in each node is **no smaller than** the key values in its children.
- A **max heap** is a **complete binary tree** that is also a max tree.

- **Min-Heap: root node has the smallest key.**

- A **min tree** is a tree in which the key value in each node is **no larger than** the key values in its children.
- A **min heap** is a **complete binary tree** that is also a min tree.

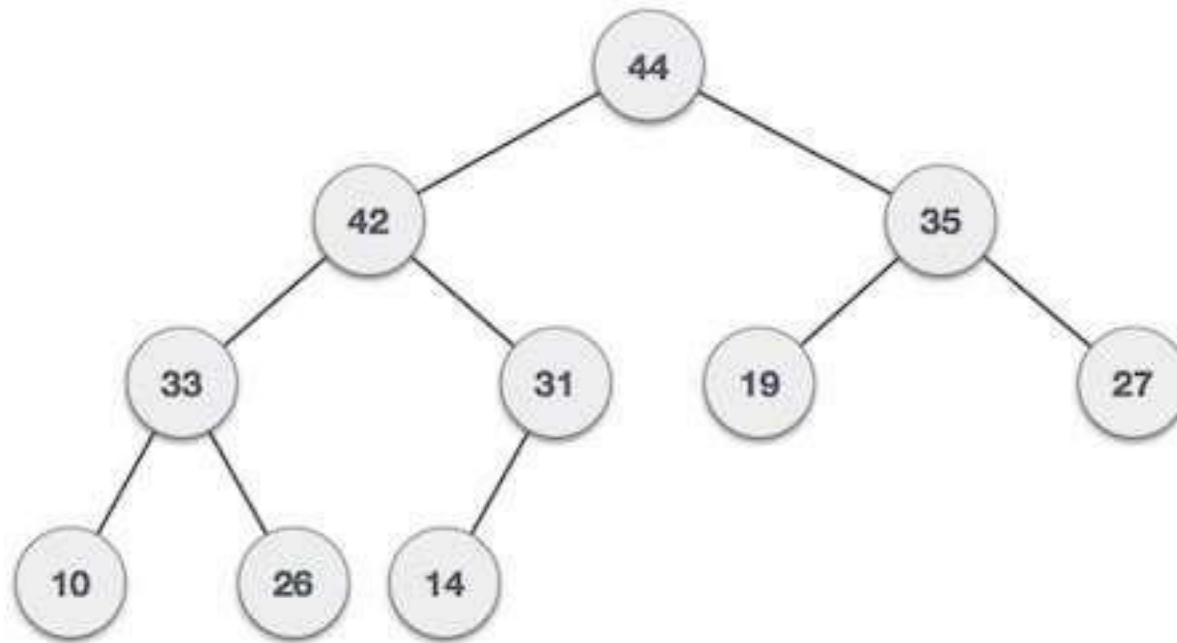
Heap

- Min-Heap
 - Where the value of the root node is less than or equal to either of its children
 - For input 35 33 42 10 14 19 27 44 26 31



Heap

- Max-Heap –
 - where the value of root node is greater than or equal to either of its children.
 - For input 35 33 42 10 14 19 27 44 26 31



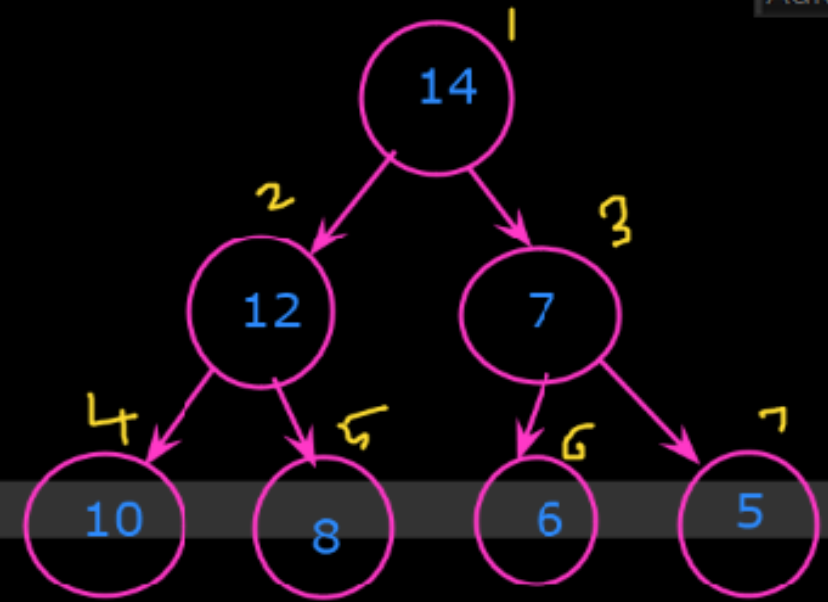
Types of heap:

1. Max-Heap: root node has largest value
2. Min-Heap: root node has smallest value

$$\text{Parent} = i/2$$

$$\text{Lc} = 2i$$

$$\text{RC} = 2i+1$$



Max heap

14 12 7 10 8 6 5

1 2 3 4 5 6 7

Types of heap:

1. Max-Heap: root node has largest value
2. Min-Heap: root node has smallest value

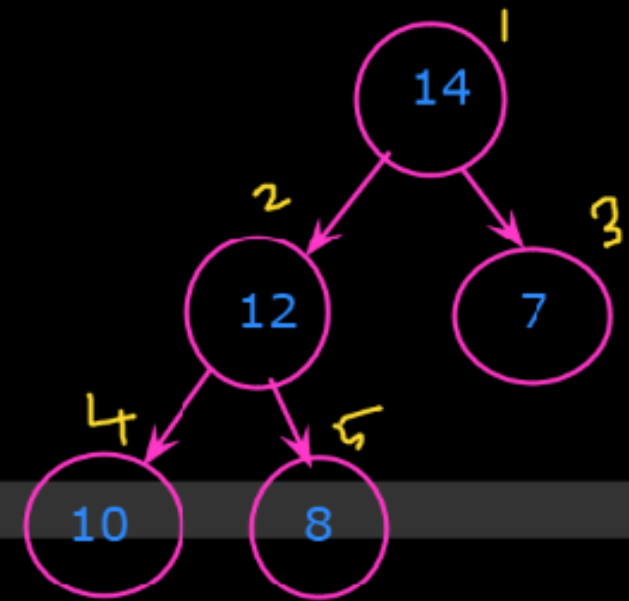
$$\text{Parent} = i/2$$

$$\text{Lc} = 2i$$

$$\text{RC} = 2i+1$$

14 12 7 10 8

1 2 3 4 5 6 7



Max heap

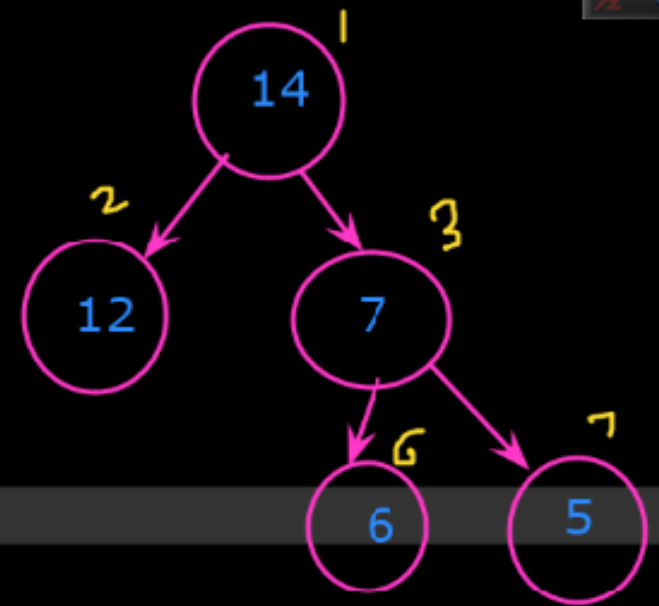
Types of heap:

1. Max-Heap: root node has largest value
2. Min-Heap: root node has smallest value

$$\text{Parent} = i/2$$

$$\text{Lc} = 2i$$

$$\text{RC} = 2i+1$$



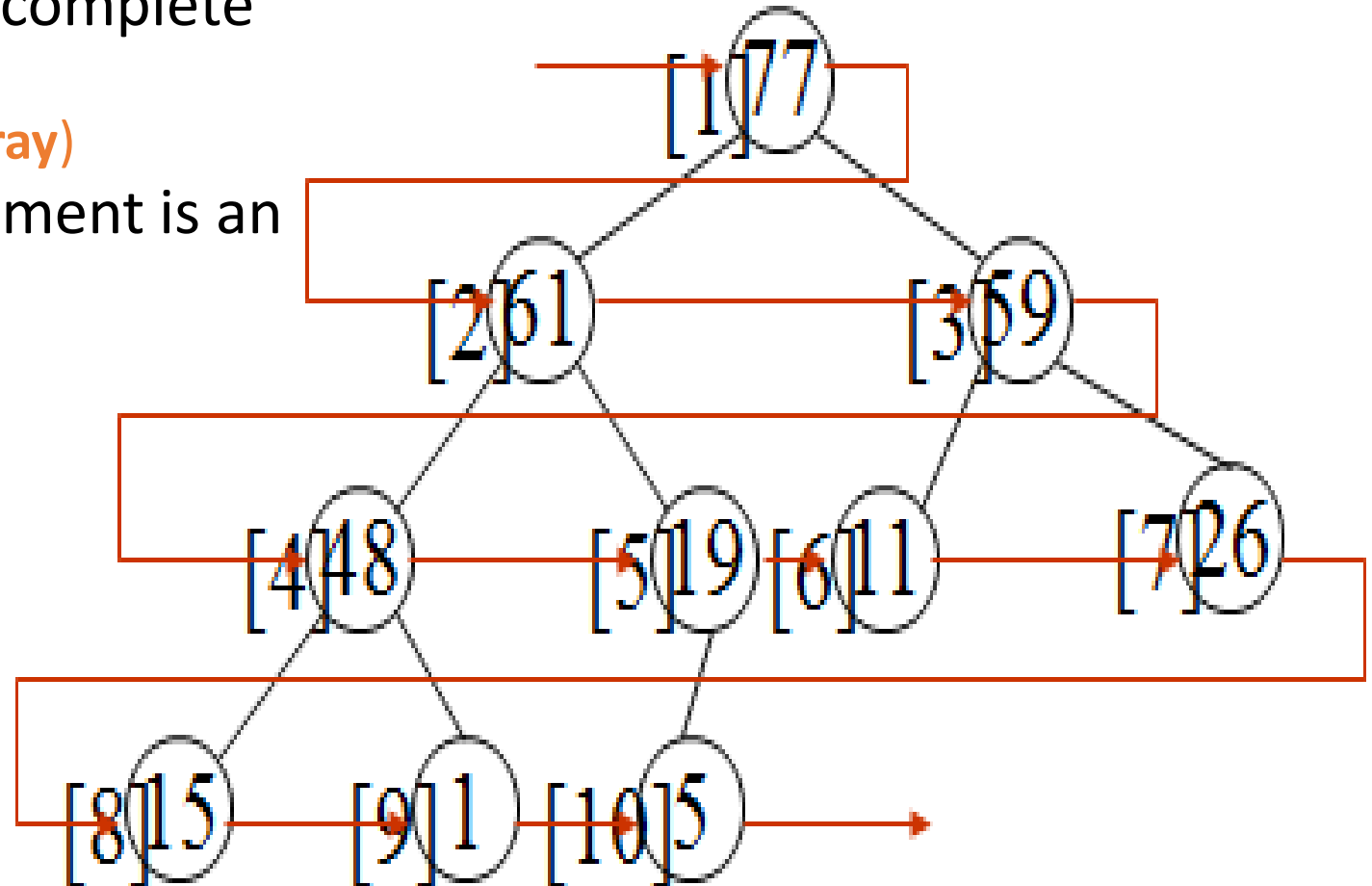
Max heap

14 12 7 - - 6 5

1 2 3 4 5 6 7

- **Note:**

- Heap in data structure is a complete binary tree!
 - (Nice representation in Array)
- Heap in C program environment is an array of memory.



— Stored using array in C

index	1	2	3	4	5	6	7	8	9	10
value	77	61	59	48	19	11	26	15	1	5


```

for(int i=n/2-1; i>=0; i--)
    heapify(arr,n,i);

//n-1 elements consider ka
for(int i=n-1; i>0; i--)
{
    //replacement of last
    int temp = arr[0];
    arr[0] = arr[i];
    arr[i] = temp;
    //balancing of maxheap
    heapify(arr,i,0);
}

```

10 20 30 40 50

```

void display(int arr[])
{
    int n = arr.length;
    for(int i=0; i<n; i++)

```

C:\Windows\system32\cmd.e: x + v

Microsoft Windows [Version 10.0.22621.2134]
(c) Microsoft Corporation. All rights reserved.

D:\Test>javac HSort.java

D:\Test>java HSort

Unsorted array:

10
40
30
50
20

D:\Test>javac HSort.java

D:\Test>java HSort

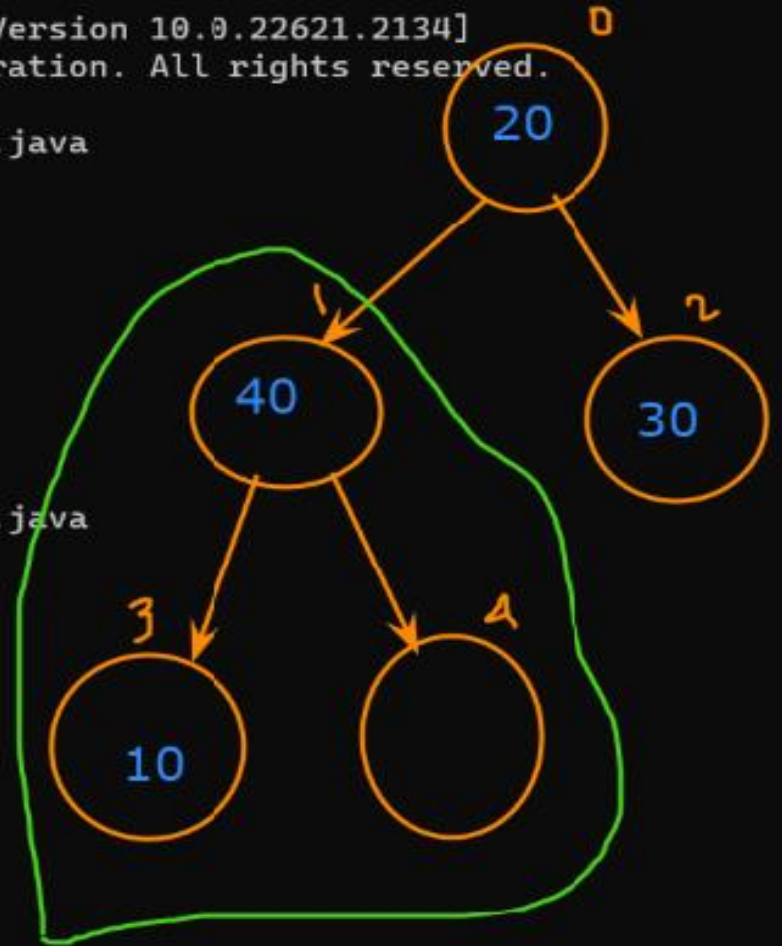
Unsorted array:

10
40
30
50
20

Sorted array:

10
20
30
40
50

D:\Test>

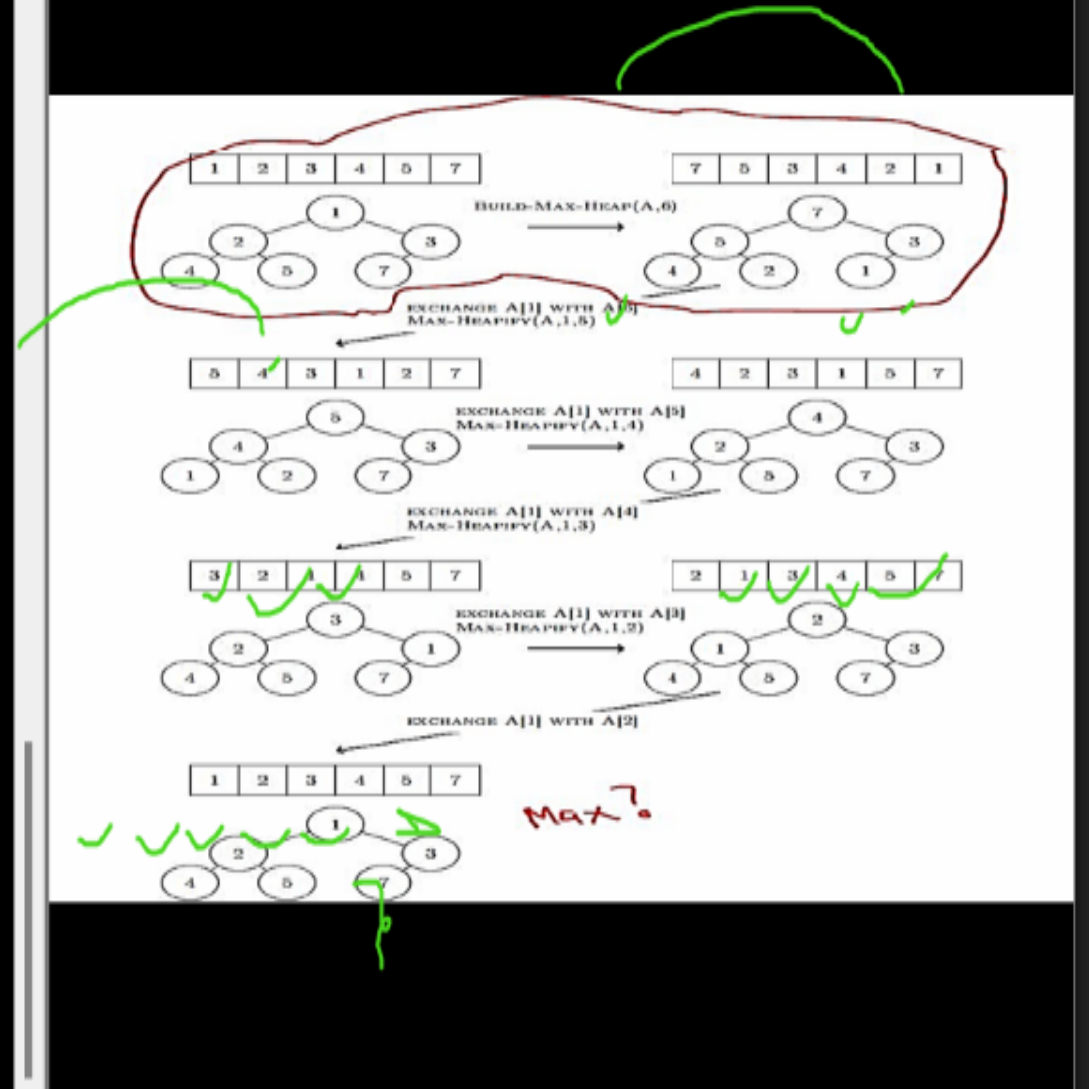


50

```

public static void main(String args[])
{
    HSort h1 = new HSort();
    int arr[] = {10, 40, 30, 50, 20};
    System.out.println("Unsorted array:");
    h1.display(arr);
    h1.heapsort(arr);
    System.out.println("Sorted array:");
    h1.display(arr);
}

```



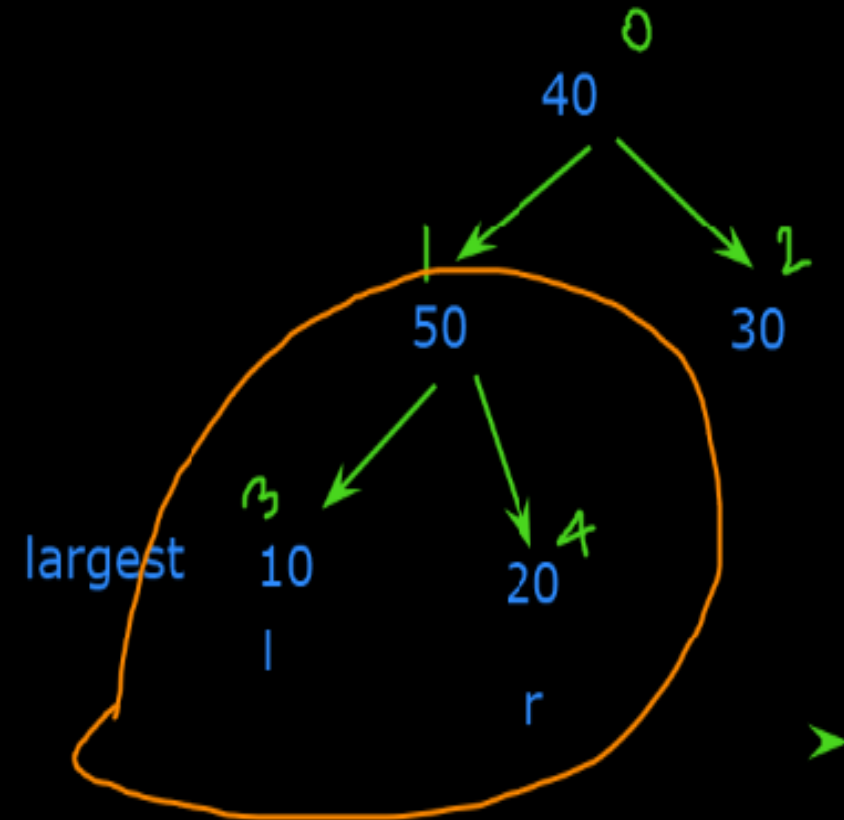
```

//to construct max heap at Root, LC and RC
void heapify(int arr[],int n, int i)
{
    //Max heap
    int largest = i; //i/2 : Parent
    int l = 2*i+1; //2*i : LC
    int r = 2*i+2; //2*i+1 : RC

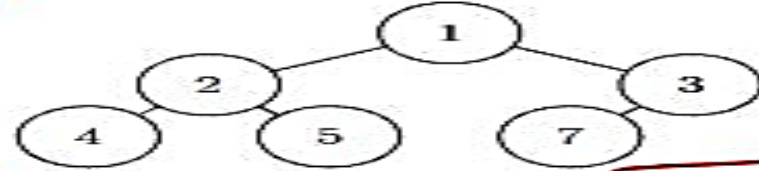
    if(l < n && arr[l] > arr[largest])
        largest = l;
    if(r < n && arr[r] > arr[largest])
        largest = r;

    if(largest != i)
    {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;
        heapify(arr,n,largest);
    }
}

```

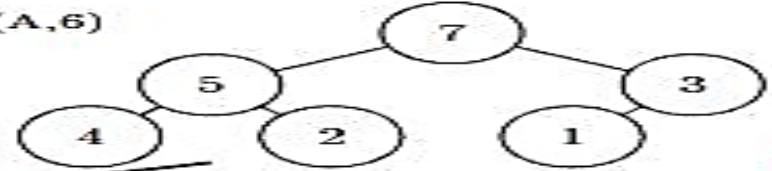


1	2	3	4	5	7
---	---	---	---	---	---



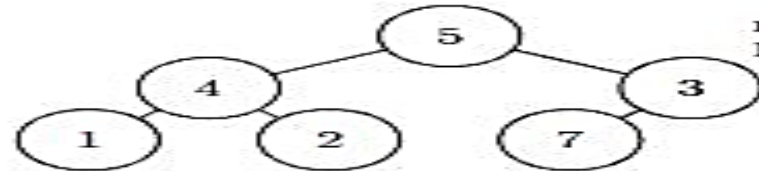
BUILD-MAX-HEAP(A,6)

7	5	3	4	2	1
---	---	---	---	---	---



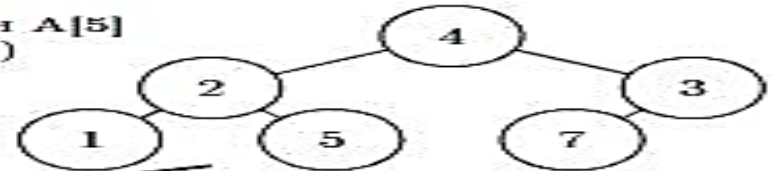
EXCHANGE A[1] WITH A[6]
MAX-HEAPIFY(A,1,5)

5	4	3	1	2	7
---	---	---	---	---	---



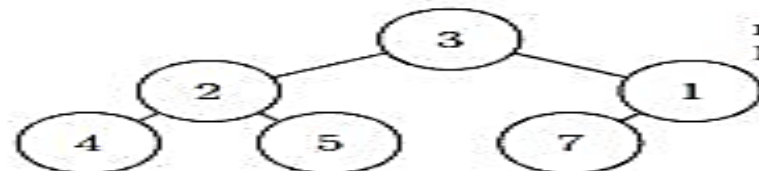
EXCHANGE A[1] WITH A[5]
MAX-HEAPIFY(A,1,4)

4	2	3	1	5	7
---	---	---	---	---	---



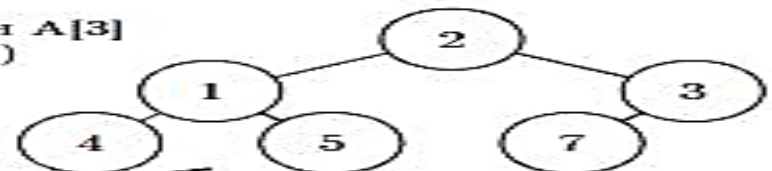
EXCHANGE A[1] WITH A[4]
MAX-HEAPIFY(A,1,3)

3	2	1	4	5	7
---	---	---	---	---	---



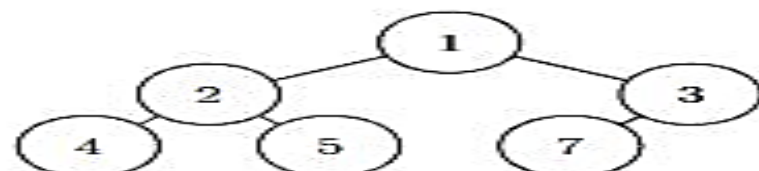
EXCHANGE A[1] WITH A[3]
MAX-HEAPIFY(A,1,2)

2	1	3	4	5	7
---	---	---	---	---	---



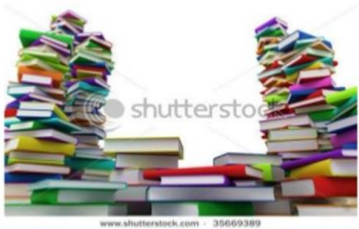
EXCHANGE A[1] WITH A[2]

1	2	3	4	5	7
---	---	---	---	---	---



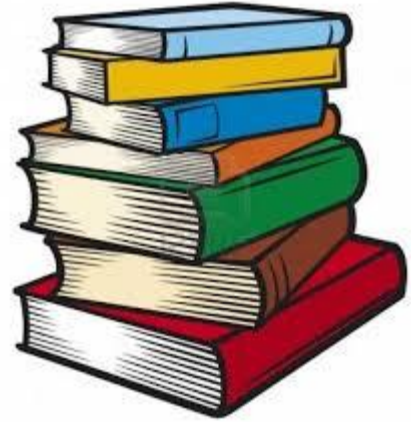
MAX?

Examples of stack



Stacks

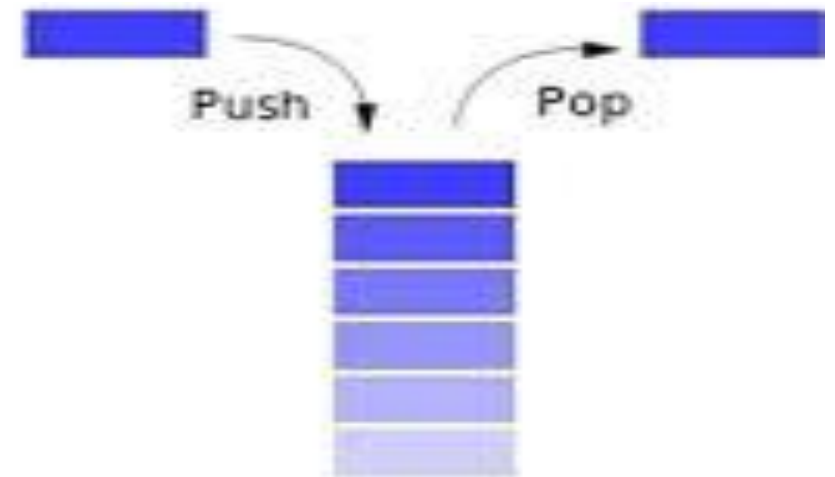
Kiran Waghmare



Stack of books



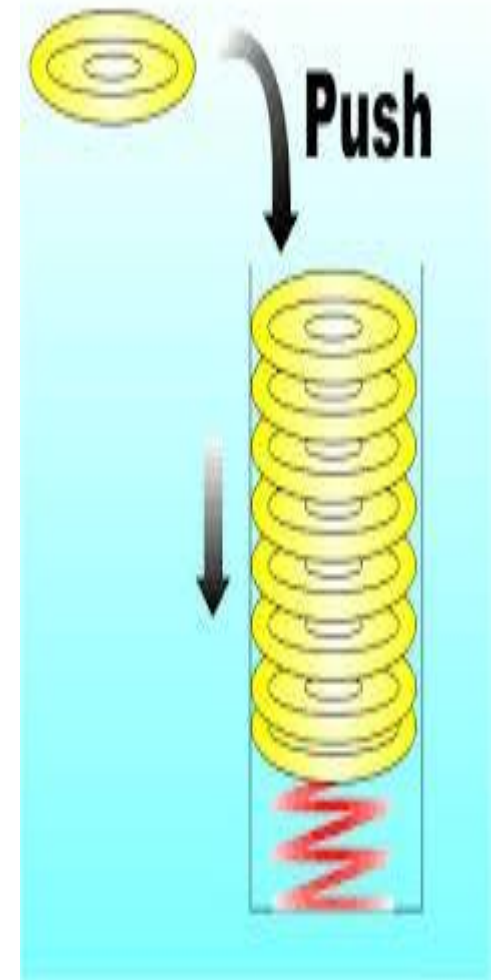
Stack of Coins



Memory stack

Stack

- Stack is an ordered list of similar data type.
- Stack is a LIFO structure. (Last in First out).
- push() function is used to insert new elements into the Stack and pop() is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called Top.
- Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

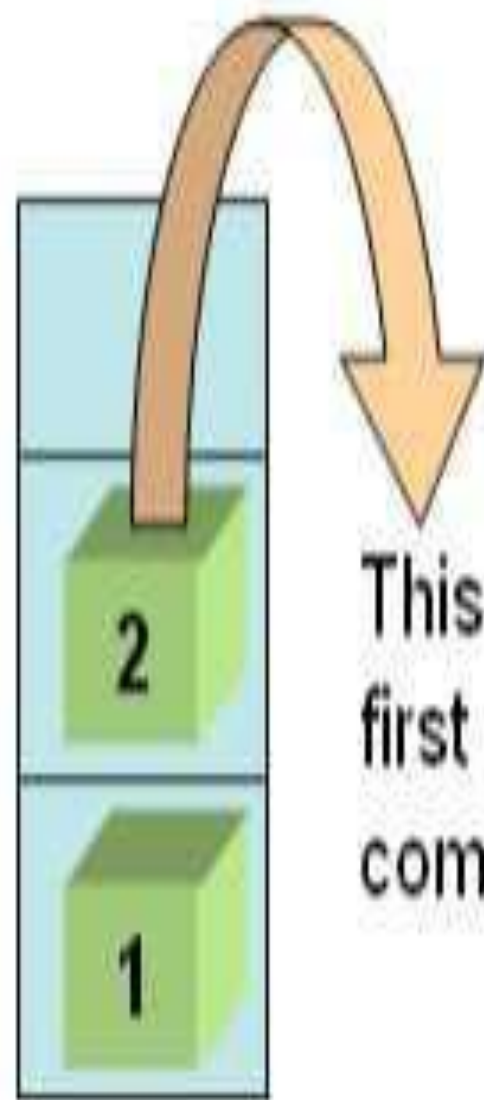
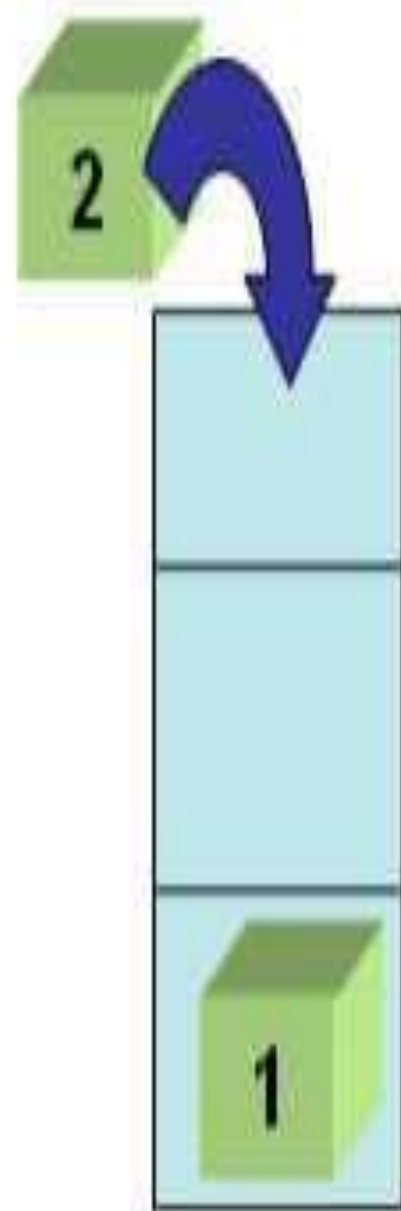


Standard Stack Operations

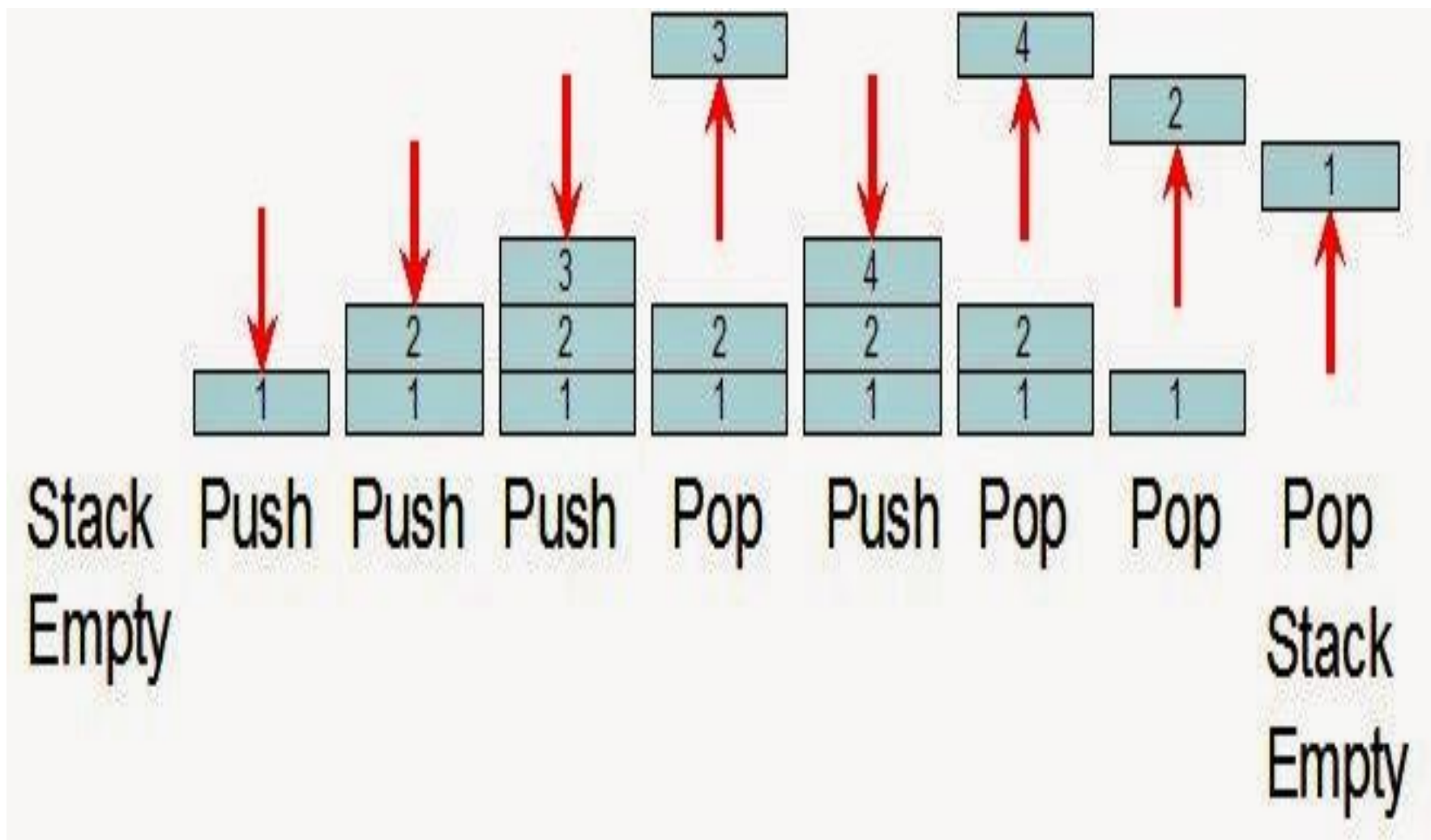
- The following are some common operations implemented on the stack:
- **push():**
 - When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():**
 - When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():**
 - It determines whether the stack is empty or not.
- **isFull():**
 - It determines whether the stack is full or not.'
- **peek():**
 - It returns the element at the given position.
- **count():**
 - It returns the total number of elements available in a stack.
- **change():**
 - It changes the element at the given position.
- **display():**
 - It prints all the elements available in the stack.



Empty Stack

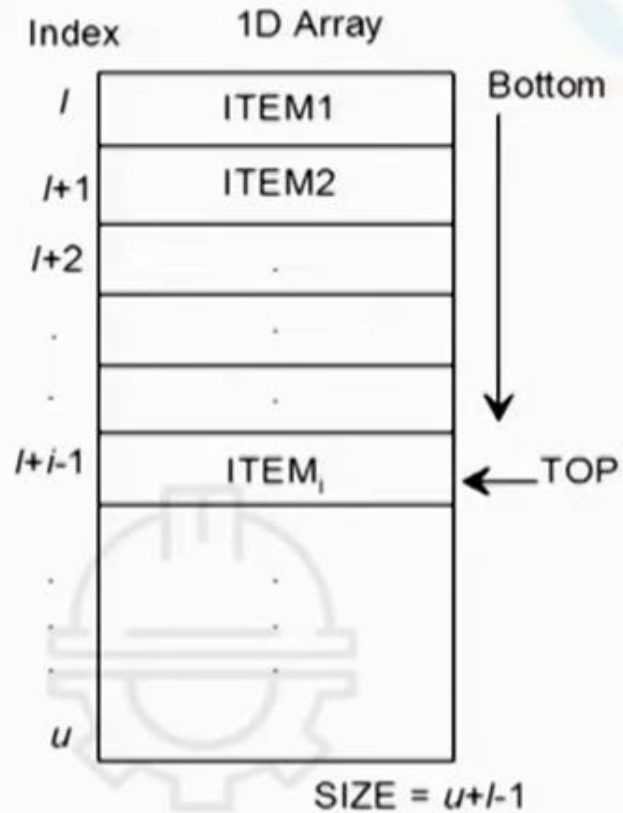


This will be the
first object to
come out.

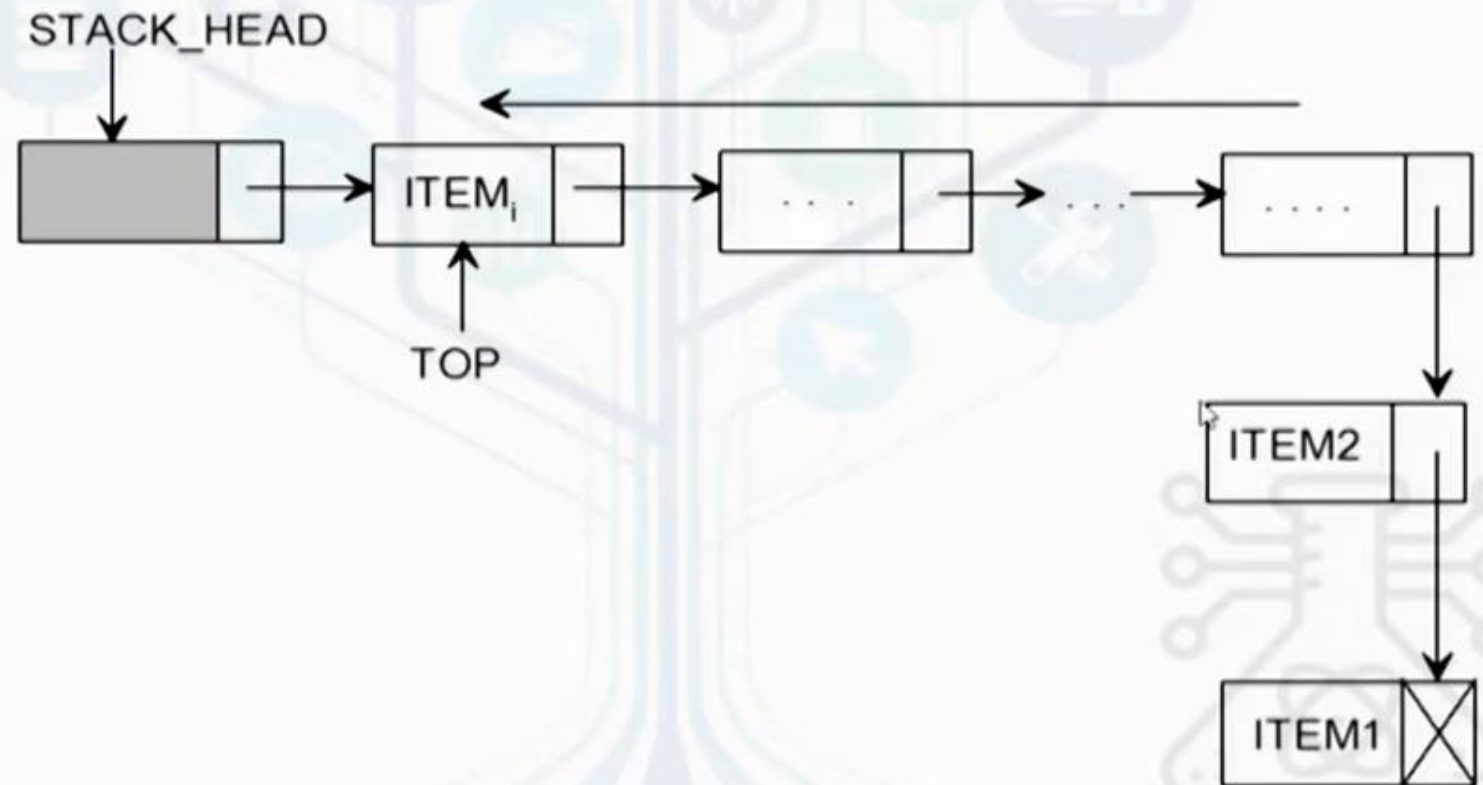


Memory representations

Array representation



Linked list representation



```
void push(long j)
{
    stackArray[++top] = j;
}

long pop()
{
    return stackArray[top--];
}

long peek()
{
    return stackArray[top];
}

boolean isEmpty()
{
    return (top == -1);
}
```

Applications of Stack

- The following are the applications of the stack:
- **Balancing of symbols:**
 - Stack is used for balancing a symbol. For example, we have the following program:
 - As we know, each program has an opening and closing braces;
 - when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack.
 - Therefore, the net value comes out to be zero.
 - If any symbol is left in the stack, it means that some syntax occurs in a program.
- **String reversal:**
 - Stack is also used for reversing a string.
 - For example, we want to reverse a "javaTpoint" string, so we can achieve this with the help of a stack.
 - First, we push all the characters of the string in a stack until we reach the null character.
 - After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.

- **UNDO/REDO:**

- It can also be used for performing UNDO/REDO operations.
- For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc.
- So, there are three states, a, ab, and abc, which are stored in a stack.
- There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
- If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.

- **Recursion:**

- The recursion means that the function is calling itself again.
- To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.

- **DFS(Depth First Search):**

- This search is implemented on a Graph, and Graph uses the stack data structure.

- **Backtracking:**

- Suppose we have to create a path to solve a maze problem.
- If we are moving in a particular path, and we realize that we come on the wrong way.
- In order to come at the beginning of the path to create a new path, we have to use the stack data structure.

- **Expression conversion:**

- Stack can also be used for expression conversion.
- This is one of the most important applications of stack.
- The list of the expression conversion is given below:
 - Infix to prefix
 - Infix to postfix
 - Prefix to infix
 - Prefix to postfix
 - Postfix to infix

- **Memory management:**

- The stack manages the memory.
- The memory is assigned in the contiguous memory blocks.
- The memory is known as stack memory as all the variables are assigned in a function call stack memory.
- The memory size assigned to the program is known to the compiler.

When the function is created, all its variables are assigned in the stack memory.

When the function completed its execution, all the variables assigned in the stack are released.

Parenthesis matching problem : Algorithm

1. For each character of input string
2. If character is opening parenthesis '(', put it on stack.
3. If character is closing parenthesis ')'
 - a. Check top of stack, if it is '(', pop and move to next character.
 - b. If it is not '(', return false
4. After scanning the entire string, check if stack is empty. If stack is empty, return true else return false.

Check for Balanced Brackets in an expression (well-formedness) using Stack

- Given an expression string `exp`, write a program to examine whether the pairs and the orders of “{”, “}”, “(”, “)”, “[”, “]” are correct in the given expression.
- Example:
- Input: `exp = “[()]{}[()()](){}”`
- Output: **Balanced**
- Explanation: all the brackets are well-formed
- Input: `exp = “[()]”`
- Output: **Not Balanced**
- Explanation: 1 and 4 brackets are not balanced because
- there is a closing ‘]’ before the closing ‘(‘

Polish Notations

1. Infix Notation : $A+B$

2. Prefix Notation: $+AB$

3. Postfix Notation : $AB+$

- **Operator Precedence:**

1. **BODMAS Rule**

2. **Brackets, Exponential, $(* / \%)$, $(+ -)$**

- **Rules: Infix to Postfix Conversion**

1. Parenthesize the expression starting from left to right.

2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in above expression $B * C$ is parenthesized first before $A+B$.

3. The sub-expression (part of expression) which has been converted into postfix is to be treated as single operand.

4. Once the expression is converted to postfix from remove the parenthesis.

ALGORITHM:

- Scan infix expression from left to right.
- If there is a character as operand, output it.
- if not
 - 1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
 - 2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
- If the scanned character is an '(', push it to the stack.
- If the character character is an ')', pop the stack and and output it until a '(' is encountered, and discard both the parenthesis.
- Repeat steps 2-6 until infix expression is scanned.
- display the output
- Pop and output from the stack until it is not empty.

ADVANTAGE OF POSTFIX:

- **Any formula can be expressed without parenthesis.**
- **It is very convenient for evaluating formulas on computer with stacks.**
- **Postfix expression doesn't have the operator precedence.**
- **Postfix is slightly easier to evaluate.**
- **It reflects the order in which operations are performed.**
- **You need to worry about the left and right associativity.**

Application: Evaluation of a postfix expression

Steps:

1. Append a special delimiter '#' at the end of the expression
2. $\text{item} = E.\text{ReadSymbol}()$ // Read the first symbol from E
3. **While** ($\text{item} \neq \text{'\#'}$) **do**
4. **If** ($\text{item} = \text{operand}$) **then**
5. **PUSH**(item) // Operand is the first push into the stack
6. **Else**
7. $\text{op} = \text{item}$ // The item is an operator
8. $y = \text{POP}()$ // The right-most operand of the current operator
9. $x = \text{POP}()$ // The left-most operand of the current operator
10. $t = x \text{ op } y$ // Perform the operation with operator 'op' and operands x, y
11. **PUSH**(t) // Push the result into stack
12. **EndIf**
13. $\text{item} = E.\text{ReadSymbol}()$ // Read the next item from E
14. **EndWhile**
15. $\text{value} = \text{POP}()$ // Get the value of the expression
16. **Return**(value)
17. **Stop**

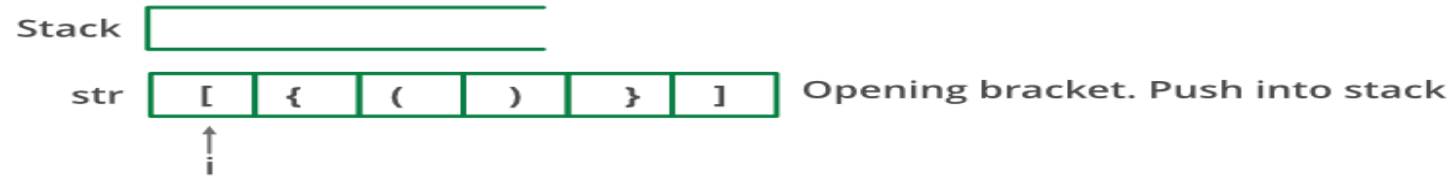
Balance Parenthesis

- Given an expression string *exp*, write a program to examine whether the pairs and the orders of “{“, “}”, “(“, “)”, “[“, “]” are correct in *exp*.
- **Example:**
- *Input: exp = “[()]”*
Output: Not Balanced
- *Input: exp = “[()]{}{[()()]()}"*
Output: Balanced

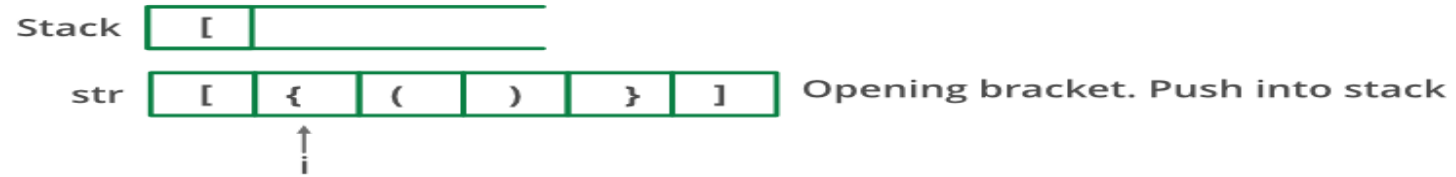
Algorithm:

- Declare a character stack S.
- Now traverse the expression string exp.
 - If the current character is a starting bracket ('(' or '{' or '[') then push it to stack.
 - If the current character is a closing bracket (')' or '}' or ']') then pop from stack and if the popped character is the matching starting bracket then fine else brackets are not balanced.
- After complete traversal, if there is some starting bracket left in stack then “not balanced”

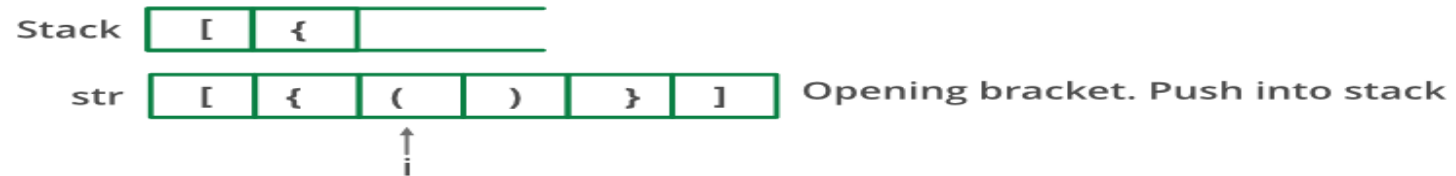
Initially :



Step 1:



Step 2:



Step 3:



Step 4:



Step 5:



Thanks