



SUMMER INTERNSHIP 2023

OOPS, Data structures and Algorithms

Bharathwaj Vasudevan, Principal Software Engineer I, Mr.Cooper

Harini Priya K, Senior Software Engineer, Mr.Cooper

**Object Oriented
Programming Systems
(OOPS)**

Variables are containers for storing data values

Access
Modifier

Type

Name/
Identifier

Statement
End

`private int doors;`

Types of variables:-

- Static\Class Variables
- Instance Variables
- Local Variables

```
class GFG {  
    public :  
        static int a ;  
        int b ;  
    public :  
        func ()  
        {  
            int c ;  
        },  
};
```

`static int a ;` → Static Variable

`int b ;` → Instance Variable

`int c ;` → Local Variable

Variables

(or)

Identifiers

(or)

Data Members

Methods

(or)

Functions

(or)

Member Functions

- Functions are small piece of code performing a particular task
- They help in modularizing our program
- A function is a small independent unit of a program
- Any program contains one or more function

```
int square(int num) {  
    return num * num;  
}  
...  
...  
result = square(10);  
// code
```

return value

method call

Consider you are
filling a form....

The diagram illustrates a form-filling process with a central form and three data cards. The central form is a rectangle with a blue triangular tab on its right side. It contains the following fields:

- Name : _____
- Age : _____
- Gender : _____
- Mobile : _____

Three data cards are positioned around the central form, each with a blue triangular tab on its top-right corner. They are connected to the central form by dashed lines:

- Top Card (Tony):** Connected to the top of the central form. It contains:
 - Name : Tony
 - Age : 24
 - Gender : Male
 - Mobile : 890-123-4567
- Bottom Card (Linda):** Connected to the bottom of the central form. It contains:
 - Name : Linda
 - Age : 29
 - Gender : Female
 - Mobile : 234-567-8901
- Right Card (Adam):** Connected to the right side of the central form. It contains:
 - Name : Adam
 - Age : 21
 - Gender : Male
 - Mobile : 123-456-7890

Encapsulation

Encapsulation is the practice of bundling related data into a structured unit, along with the methods used to work with that data.

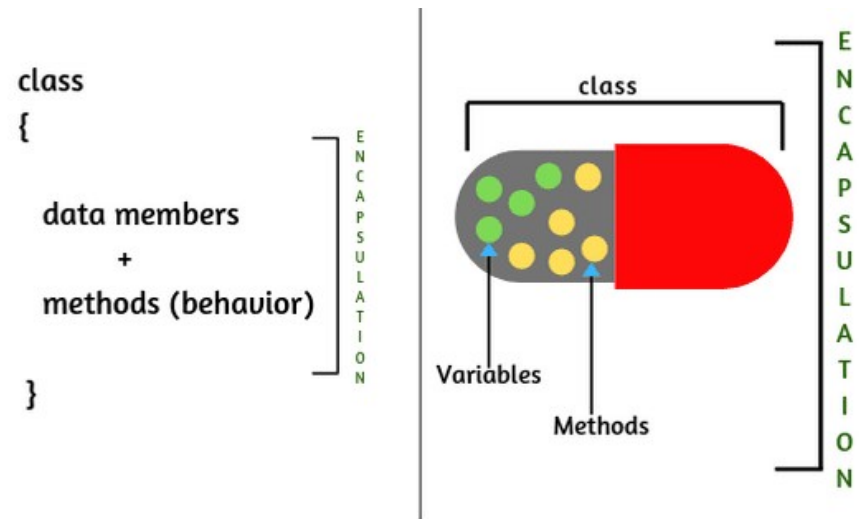
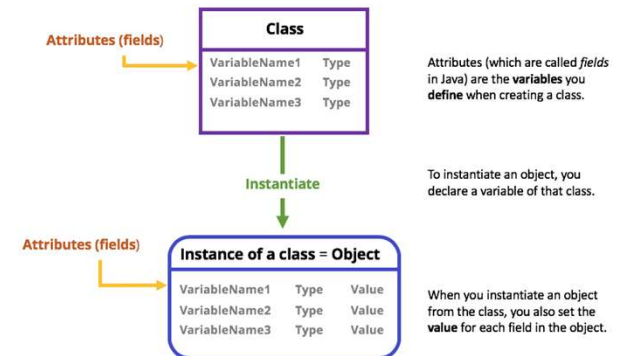
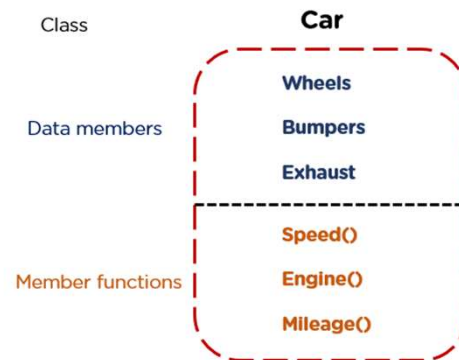


Fig: Encapsulation

Classes & Objects



```
class Bicycle
{
    public int gear = 5;

    public void braking()
    {
        System.out.println("Working of Braking");
    }
}
```

```
className object = new className();
```

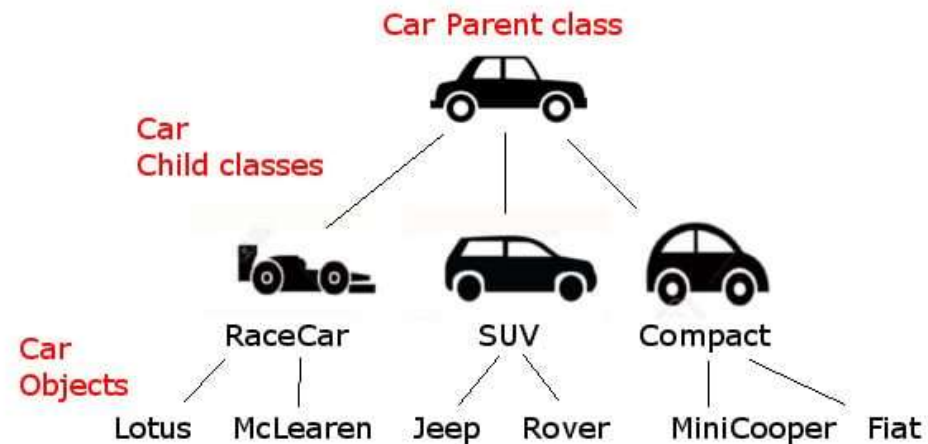
```
-----
```

```
Bicycle sportsBicycle = new Bicycle();
```

```
Bicycle touringBicycle = new Bicycle();
```

Inheritance

- Inheritance is one of the key features of OOP that allows us to create a new class from an existing class.
- The new class that is created is known as subclass (child or derived class) and the existing class from where the child class is derived is known as superclass (parent or base class).
- The extends keyword is used to perform inheritance in Java



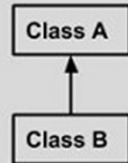
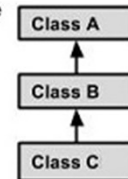
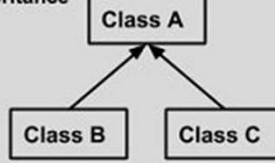
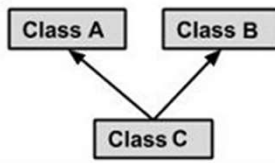
Inheritance

```
class Animal {  
    String name;  
    public void eat() {  
        System.out.println("I can eat");  
    }  
}  
  
class Dog extends Animal {  
    public void display() {  
        System.out.println("My name is " + name);  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Dog labrador = new Dog();  
        labrador.name = "Rohu";  
        labrador.display();  
        labrador.eat();  
    }  
}
```

Output:

```
My name is Rohu  
I can eat
```

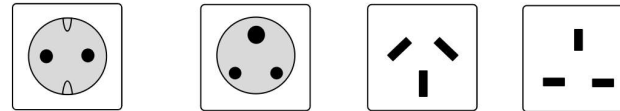
Types of Inheritance

Single Inheritance  <pre> graph BT B[Class B] --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } </pre>
Multi Level Inheritance  <pre> graph BT C[Class C] --> B[Class B] B --> A[Class A] </pre>	<pre> public class A {} public class B extends A {.....} public class C extends B {..... } </pre>
Hierarchical Inheritance  <pre> graph BT B[Class B] --> A[Class A] C[Class C] --> A </pre>	<pre> public class A {} public class B extends A {.....} public class C extends A {..... } </pre>
Multiple Inheritance  <pre> graph BT C[Class C] --> A[Class A] C --> B[Class B] </pre>	<pre> public class A {} public class B {.....} public class C extends A,B { } // Java does not support mutiple Inheritance </pre>

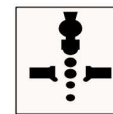
Polymorphism

- Ability of an object to respond differently to different messages
- With Polymorphism, a message is sent to multiple class objects, and every object responds appropriately according to the properties of the class

Without Polymorphism



With Polymorphism



Polymorphism

```
class Language {  
    public void displayInfo() {  
        System.out.println("Common English Language");  
    }  
}
```

```
class Java extends Language {  
    public void displayInfo() {  
        System.out.println("Java Programming Language");  
    }  
}
```

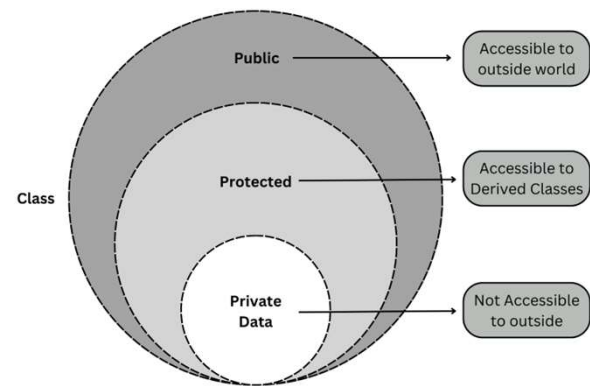
```
class Main {  
    public static void main(String[] args) {  
  
        // create an object of Java class  
        Java j1 = new Java();  
        j1.displayInfo();  
  
        // create an object of Language class  
        Language l1 = new Language();  
        l1.displayInfo();  
    }  
}
```

Output:

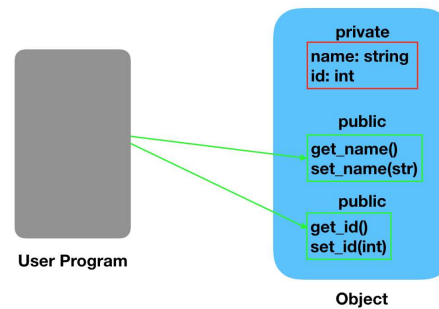
```
Java Programming Language  
Common English Language
```

Abstraction

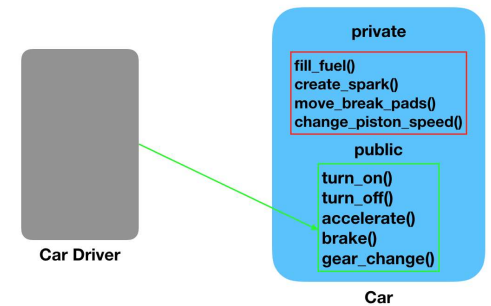
- Abstraction is a technique of hiding unnecessary details from the user.
- The user is only given access to the details that are relevant



Data Abstraction



Process Abstraction



Data Structures

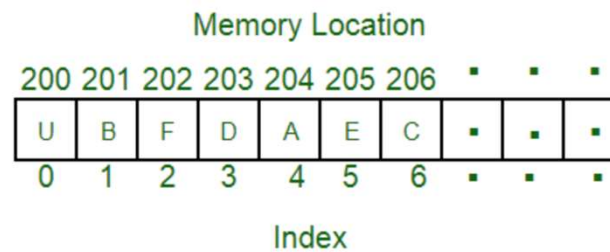
Data Structures

A **data structure** is a particular way of organizing data in a computer.

- **Array**
- **Stack**
- **Queue**
- **Linked List**
- **Trees**
- **Graph**

Arrays

- An array is a collection of items stored at **contiguous memory** locations.
- The idea is to store multiple items of the **same type** together.



Array Rotation

[1, 2, 3, 4, 5, 6, 7]

Rotation of the above array by 2 will make array

[3, 4, 5, 6, 7, 1, 2]

Method: Temp Array

Input arr[] = [1, 2, 3, 4, 5, 6, 7], d = 2, n = 7

1) Store the first d elements in a temp array temp[] = [1, 2]

2) Shift rest of the arr[] arr[] = [3, 4, 5, 6, 7, 6, 7]

3) Store back the d elements arr[] = [3, 4, 5, 6, 7, 1, 2]

Time complexity : $O(n)$

Auxiliary Space : $O(d)$

Array Rotation

[1, 2, 3, 4, 5, 6, 7]

Rotation of the above array by 2
will make array

[3, 4, 5, 6, 7, 1, 2]

Method: Rotate by one

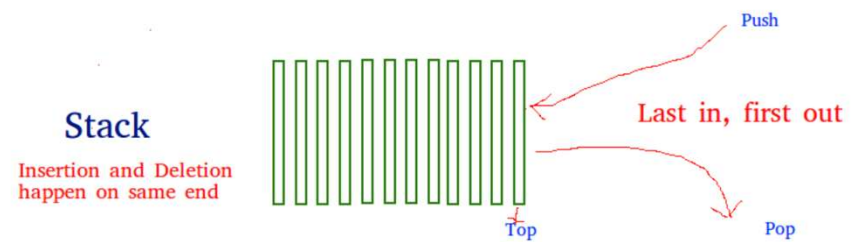
```
leftRotate(arr[], d, n)
start For i = 0 to i < d
    Left rotate all elements of arr[] by
    one
end
```

Time complexity : $O(n * d)$

Auxiliary Space : $O(1)$

Stack

- Stack is a linear data structure which follows a particular order in which the operations are performed.
- The order may be **LIFO**(Last In First Out) or **FILO**(First In Last Out).



Next Greater Number

[11, 13, 21, 3]

Output

11 -- 13

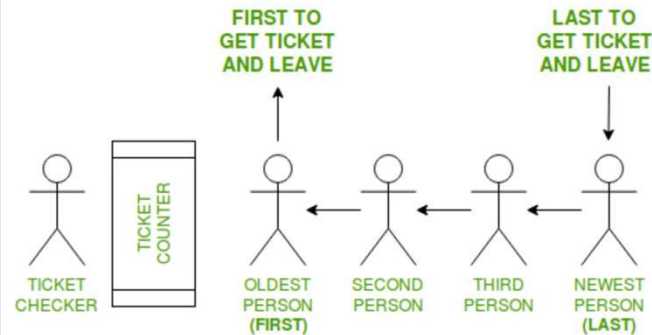
13 -- 21

21 -- -1

3 -- -1

```
void printNGE(int arr[], int n) {  
    stack < int > s;  
    s.push(arr[0]);  
  
    for (int i = 1; i < n; i++) {  
        if (s.empty()) {  
            s.push(arr[i]);  
            continue;  
        }  
        while (s.empty() == false && s.top() < arr[i])  
        {  
            cout << s.top() << " --> " << arr[i] << endl;  
            s.pop();  
        }  
        s.push(arr[i]);  
    }  
    while (s.empty() == false) {  
        cout << s.top() << " --> " << -1 << endl;  
        s.pop();  
    }  
}
```

Queue



```
int main() {  
    queue < int > q;
```

```
// Adds elements {0, 1, 2, 3, 4} to queue  
for (int i = 0; i < 5; i++)  
    q.push(i);  
print_queue(q);
```

```
// To remove the head of the queue.  
// In this the oldest element '0' will be removed  
int removedele = q.front();  
q.pop();  
cout << "removed element-" << removedele << endl;  
print_queue(q);
```

```
// To view the head of queue  
int head = q.front();  
cout << "head of queue-" << head << endl;  
int size = q.size();  
cout << "Size of queue-" << size;  
return 0;  
}
```

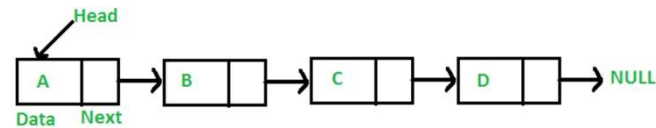
Linked List

- A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the image.

What is a pointer?

- A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location.

Like any variable or constant, you must declare a **pointer** before using it to store any variable address.



Find Middle of Linked List

- ```
void printMiddle(struct Node * head) {
 struct Node * slow_ptr = head;
 struct Node * fast_ptr = head;
 if (head != NULL) {
 while (fast_ptr != NULL &&
 fast_ptr -> next != NULL) {
 fast_ptr = fast_ptr -> next -> next;
 slow_ptr = slow_ptr -> next;
 }
 printf("The middle element is [%d]\n\n", slow_ptr -> data);
 }
}
```

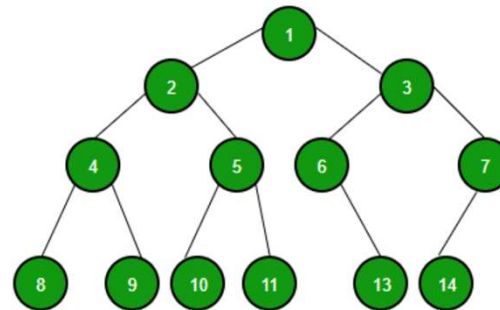
- **1->2->3->4->5->NULL**  
**The middle element is [3]**

# Binary Tree

- A tree whose elements have at most 2 children is called a binary tree.
- Element in a binary tree can have only 2 children, we typically name them the left and right child.

A Binary Tree node contains following parts.

- Data
- Pointer to left child
- Pointer to right child





# Tree Traversals

## Inorder

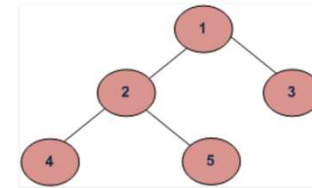
1. Traverse the left subtree,  
Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree,  
Inorder(right-subtree)

## Preorder

1. Visit the root.
2. Traverse the left subtree,  
Preorder(left-subtree)
3. Traverse the right subtree,  
Preorder(right-subtree)

## Postorder

1. Traverse the left subtree,  
call Postorder(left-subtree)
2. Traverse the right subtree,  
call Postorder(right-subtree)
3. Visit the root



(a) Inorder (Left, Root, Right) : 4 2 5 1 3  
(b) Preorder (Root, Left, Right) : 1 2 4 5 3  
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

# Symmetric Tree

```
bool isMirror(struct Node * root1,
struct Node * root2) {
 if (root1 == NULL && root2 == NULL)
 return true;
 if (root1 && root2 && root1 -> key ==
root2 -> key)
 return isMirror(root1 -> left, root2 -
> right) &&
 isMirror(root1 -> right, root2 ->
left);
 return false;
}

bool isSymmetric(struct Node * root) {
 return isMirror(root, root);
}
```

For example, this binary tree is symmetric:

```
 1
 / \
 2 2
 / \ / \
 3 4 4 3
```

But the following is not:

```
 1
 / \
 2 2
 \ \
 3 3
```

Given a binary tree, check whether it is a mirror of itself.

# Algorithms

- Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited.
- To do this a queue is used. All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.



# Queue based BFS in a binary tree

```
vector<vector<int>> levelOrder(TreeNode* root) {
 vector<vector<int>> finalans;
 queue<TreeNode*> q;
 if(root) q.push(root);

 while(!q.empty()){

 int l=q.size();
 vector<int> ans;
 for(int i=0;i<l;i++){
 TreeNode *f=q.front();
 if(f->left) q.push(f->left);
 if(f->right) q.push(f->right);
 ans.push_back(f->val);
 q.pop();

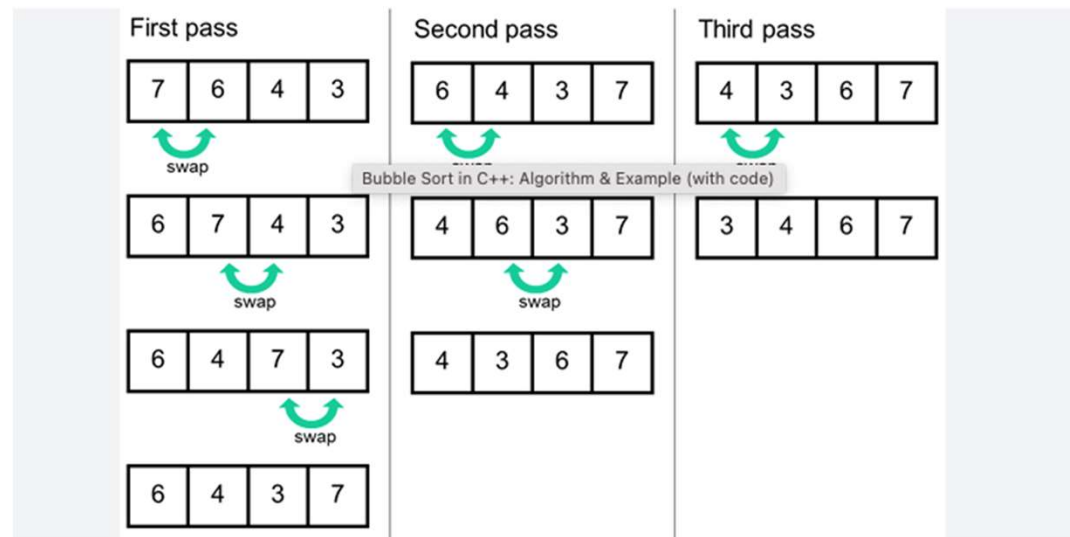
 }
 finalans.push_back(ans);

 }
 return finalans;
}
```

# Bubble Sort

*In this algorithm,*

- *traverse from left and compare adjacent elements and the higher one is placed at right side.*
- *In this way, the largest element is moved to the rightmost end at first.*
- *This process is then continued to find the second largest and place it and so on until the data is sorted.*



# Implementation

```
void bubbleSort(int array[], int size) {

 // loop to access each array element
 for (int step = 0; step < size; ++step) {

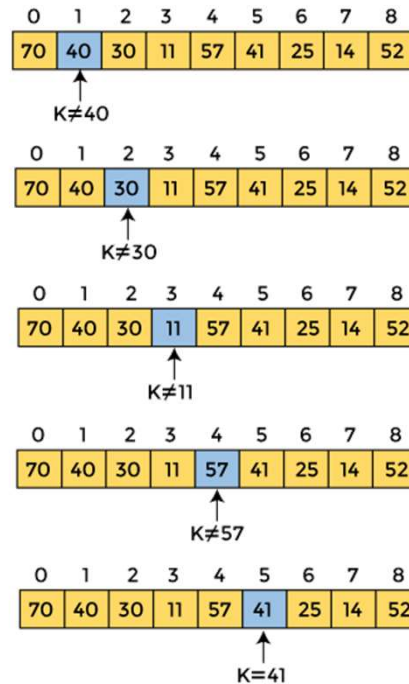
 // loop to compare array elements
 for (int i = 0; i < size - step; ++i) {

 // compare two adjacent elements
 // change > to < to sort in descending order
 if (array[i] > array[i + 1]) {

 // swapping elements if elements
 // are not in the intended order
 int temp = array[i];
 array[i] = array[i + 1];
 array[i + 1] = temp;
 }
 }
 }
}
```

# Searching Algorithm

## Linear Search



```
public static int search(int arr[], int N, int x)
{
 for (int i = 0; i < N; i++)
 {
 if (arr[i] == x)
 return i;
 }
 return -1;
}
```



# Searching Algorithm

## Binary Search

- Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.
- Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list.
- If the match is found then, the location of the middle element is returned.
- Otherwise, we search into either of the halves depending upon the result produced through the match.



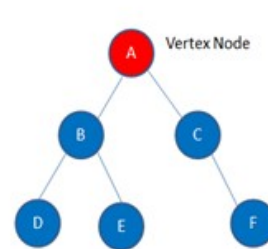
# Searching Algorithm

## Binary Search

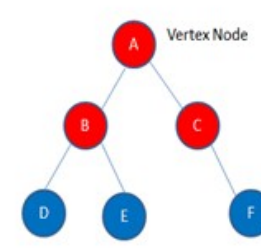
```
void binarySearch(int arr[], int first, int last, int key){
 int mid = (first + last)/2;
 while(first <= last){
 if (arr[mid] < key){
 first = mid + 1;
 } else if (arr[mid] == key){
 System.out.println("Element is found at index: " + mid);
 break;
 } else {
 last = mid - 1; }
 mid = (first + last)/2; }
 if (first > last){
 System.out.println("Element is not found!");
 }
}
```

# Depth First Search (DFS) Algorithm

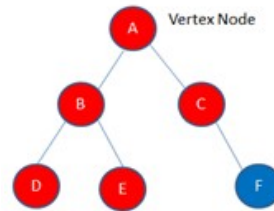
- In this article, we will discuss the DFS algorithm in the data structure.
- It is a recursive algorithm to search all the vertices of a tree data structure or a graph.
- The depth-first search (DFS) algorithm starts with the initial node of graph/tree and goes deeper until we find the goal node or the node with no children.
- Because of the recursive nature, stack data structure can be used to implement the DFS algorithm



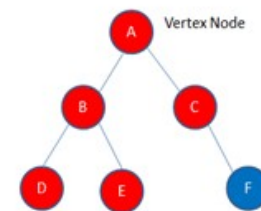
Iteration 1



Iteration 2



Iteration 3



Iteration 4



# Depth First Search (DFS) Algorithm

```
public void traversePreOrderWithoutRecursion() {
 Stack<Node> stack = new Stack<Node>();
 Node current = root;
 stack.push(root);
 while(!stack.isEmpty()) {
 current = stack.pop();
 visit(current.value);
 if(current.right != null) {
 stack.push(current.right);
 }
 if(current.left != null) {
 stack.push(current.left);
 }
 }
}
```

**QUESTIONS?**