Introduction to DSA

We will learn about:

1) Basic Data Structures
2) Big O Notation
3) Searching Algorithms
4) Sorting Algorithms
5) Graphs
6) Trees

Data Structure: A **named location** that can be used to **store and organize data**.
example: Family(A hierarchy of family relationships.)
example: array(A collection of elements stored at contiguous memory locations)
Algorithm: A **collection of steps to solve a problem**.
example: (Baking a pizza)(it will be done step by step)
example: Linear Search.

Why DSA?

1) write code that is both time and memory efficient.
2) For job interviews.
3) To get familiar with trees and graphs to get going in the field.

Stack

```
import java.util.Stack;

public class App {
    //Stack: LIFO (Last in first out) or FILO (First in last out) data structure.

    //Useful functions:
    //      push() to add on the top.
    //      pop() to get and remove the top value.
    //      peek() to get the top value.
    //      isEmpty() or empty()  returns a boolean value (true or false)
    //      search() returns the index starting from 1 from top to bottom.

    //Features of stack:
    //1) Undo/redo feature.
    //2) browser forward/backward moving.
    //3) backtracking algos.
    //4) calling functions (functions call stack)

    public static void main(String[] args) {
```

```
        Stack<String> stack = new Stack<String>();

        System.out.println(stack.empty()); // true

        stack.push("A");
        stack.push("B");
        stack.push("C");
        stack.push("D");
        stack.push("E");

        System.out.println(stack.isEmpty()); // false

        System.out.println(stack);   //[A, B, C, D, E]

        // String lastValue = stack.peek();

        System.out.println("Top of the stack: "+stack.peek());   //E

        while(!stack.isEmpty())
        {
            System.out.println("Popped value: "+stack.pop());
        }

        stack.push("A");
        stack.push("B");
        stack.push("C");
        stack.push("D");
        stack.push("E");    //On top right now

        System.out.println("Position of D: "+stack.search("D"));  //2
    }

}
```

Queue

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

public class App {
    public static void main(String[] args) throws Exception {

        //FIFO - First in first out.
```

```
//A collection of design for holding elements prior to processing.
//It's a linear data structure.

//offer()       Tail   just added to the queue(enqueue)(add)
//poll()        Head   about to leave(dequeue)(remove)
//element()     same as poll() but throws an exception that is not the case in poll().
//peek()        returns head(not remove)
//isEmpty()     true/false
//size()        no. of elements in a queue.
//contains(obj) true/false.

// Why Queue?
// keyboard buffer
// printer queue
// used in LinkedList, priority queue, BFS.

Queue<String> queue = new LinkedList<String>();
// Queue<String> queue = new Queue<String>();      //does not work   //it's an interface and we can not instantiate an object of interface.

queue.offer("a");
queue.offer("b");
queue.offer("c");
queue.offer("d");
queue.offer("e");

System.out.println("Queue: "+queue); //[a, b, c, d, e]
System.out.println("Peek: "+queue.peek()); //a

System.out.println("poll: "+queue.poll());   //a
System.out.println("poll: "+queue.poll());   //b
System.out.println("poll: "+queue.poll());   //c
System.out.println("poll: "+queue.poll());   //d
System.out.println("poll: "+queue.poll());   //e
System.out.println("poll: "+queue.poll());   //null

//System.out.println("element: "+queue.element());   //throws exception  (same as poll).

System.out.println("Empty: "+queue.isEmpty());   //true

queue.offer("a");
queue.offer("b");
queue.offer("c");
queue.offer("d");
```

```
        queue.offer("e");

        System.out.println("Size: "+queue.size());       //5

        System.out.println("Contains: "+queue.contains("a"));   //true

    }
}
```

Priority Queue

```
// import java.util.LinkedList;
import java.util.Collections;
import java.util.PriorityQueue;
import java.util.Queue;

public class App {
    public static void main(String[] args) throws Exception {

        //Priority Queue

        //It's FIFO data structure that serves
        //elements with the highest priorities first
        //before elements with lower priority

        //PriorityQueue (it will arrange in ascending order automatically.(by default))

        System.out.println("Double ascending order: ");
        Queue<Double> queue = new PriorityQueue<Double>();

        queue.offer(3.1);
        queue.offer(3.4);
        queue.offer(3.5);
        queue.offer(3.3);
        queue.offer(3.2);
        queue.offer(3.7);

        System.out.println("Queue: "+queue);   //[3.1, 3.2, 3.5, 3.4, 3.3, 3.7]

        while(!queue.isEmpty())
        {
            System.out.println(queue.poll());
        }
```

```
System.out.println("Double descending order:");
Queue<Double> queue2 = new PriorityQueue<Double>(Collections.reverseOrder());

queue2.offer(3.1);
queue2.offer(3.4);
queue2.offer(3.5);
queue2.offer(3.3);
queue2.offer(3.2);
queue2.offer(3.7);

while(!queue2.isEmpty())
{
   System.out.println(queue2.poll());
}

System.out.println("String ascending order:");
Queue<String> queue3 = new PriorityQueue<String>();

queue3.offer("Huzaifa");
queue3.offer("Haider");
queue3.offer("Hassan");
queue3.offer("Rahil");
queue3.offer("Ali");
queue3.offer("Abbas");
queue3.offer("Jawad");

while(!queue3.isEmpty())
{
   System.out.println(queue3.poll());
}

System.out.println("String descending order:");
Queue<String> queue4 = new PriorityQueue<String>(Collections.reverseOrder());

queue4.offer("Huzaifa");
queue4.offer("Haider");
queue4.offer("Hassan");
queue4.offer("Rahil");
queue4.offer("Ali");
queue4.offer("Abbas");
queue4.offer("Jawad");

while(!queue4.isEmpty())
{
```

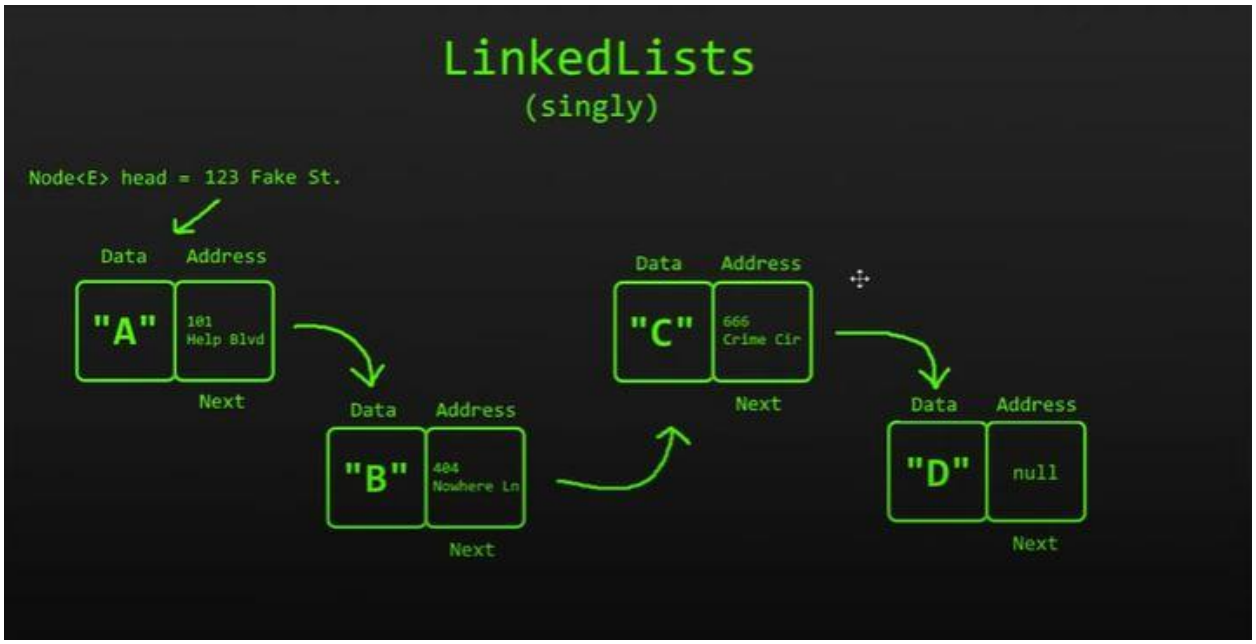
```
        System.out.println(queue4.poll());
    }

  }
}
```

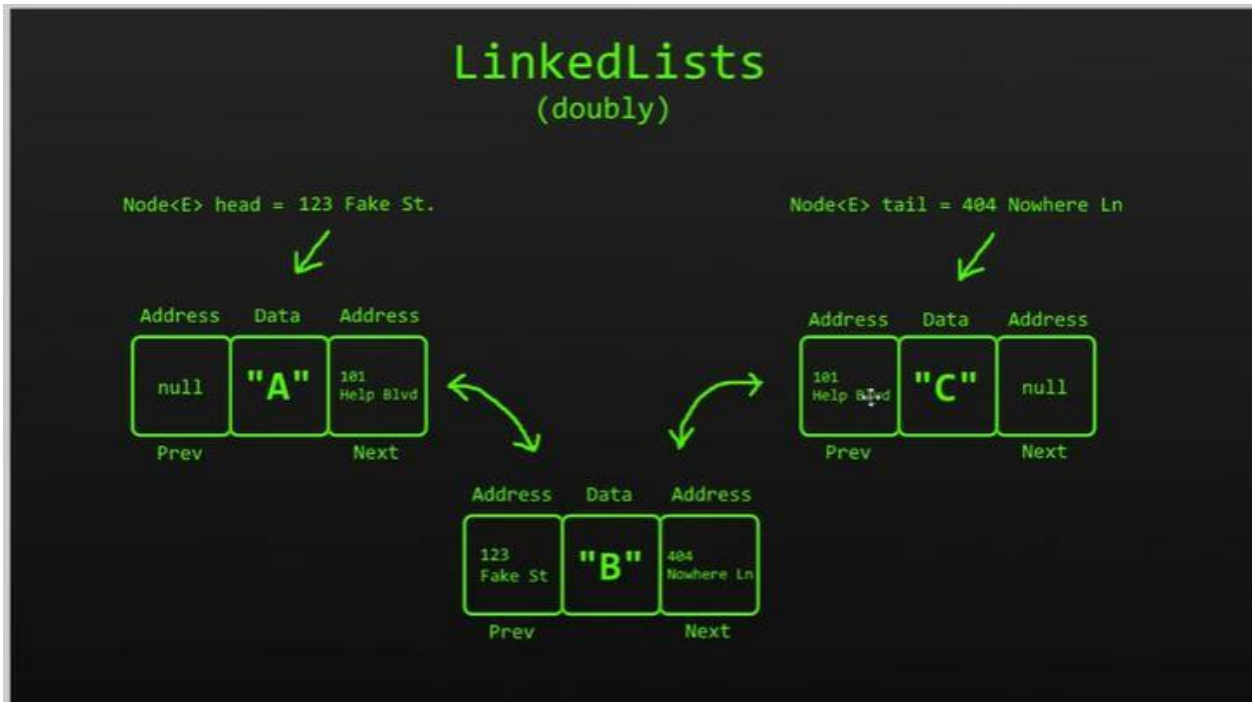
Linked List

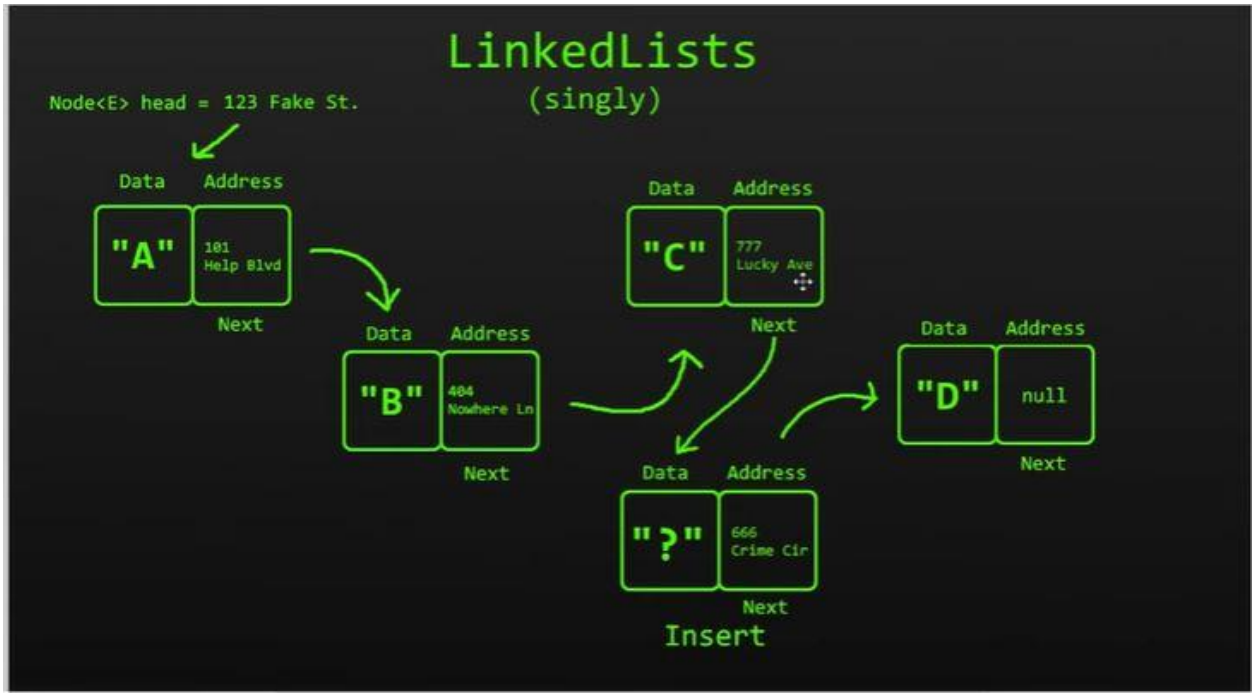
```
import java.util.LinkedList;

public class App {
  public static void main(String[] args) throws Exception {

    // ArrayLists:
    // contiguous memory location
    // Takes time in insert and delete into ArrayList(shifting happens here)

    //LinkedList:
    //at non contiguous memory locations.
    //No shifting of elements required in insert/delete

    //doubly linked list:
    //can traverse backward and forward but takes even more memory.


    // *******************************************************
    // LinkedList =  Nodes are in 2 parts (data + address)
    //               Nodes are in non-consecutive memory locations
    //               Elements are linked using pointers

    //  advantages?
    //  1. Dynamic Data Structure (allocates needed memory while running)
    //  2. Insertion and Deletion of Nodes is easy. O(1)
    //  3. No/Low memory waste

    //  disadvantages?
    //  1. Greater memory usage (additional pointer)
    //  2. No random access of elements (no index [i])
    //  3. Accessing/searching elements is more time consuming. O(n)

    //  uses?
    //  1. implement Stacks/Queues
    //  2. GPS navigation
    //  3. music playlist
```

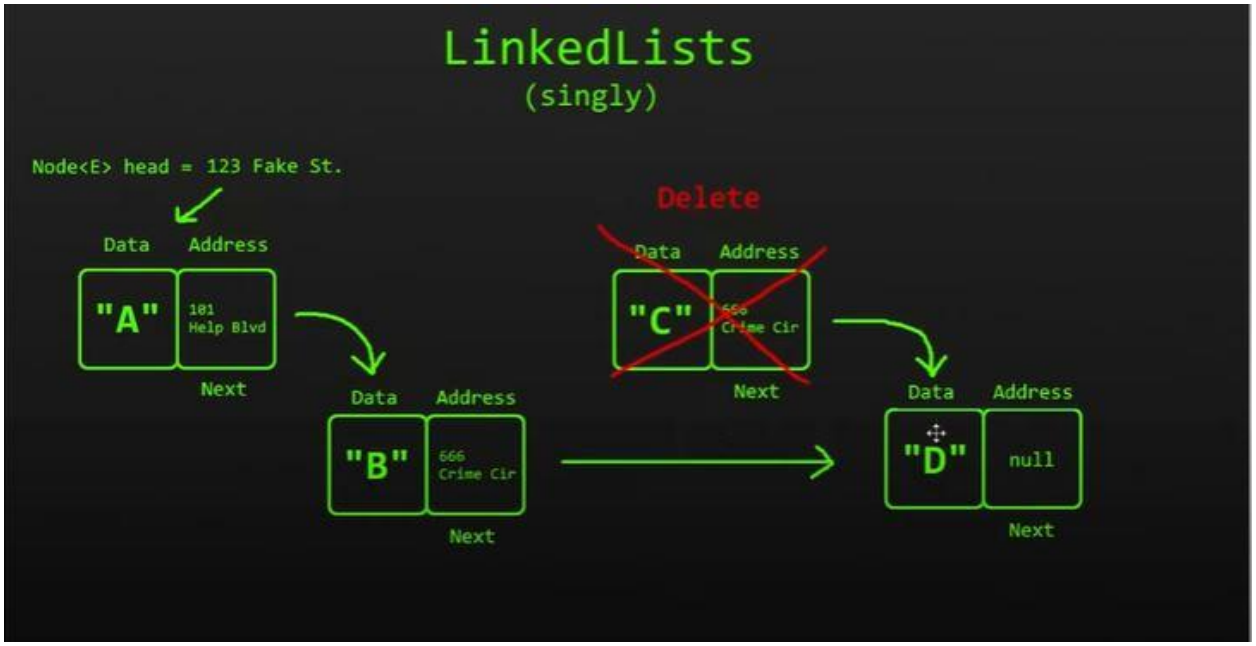
```
// ******************************************************

LinkedList<String> linkedlist = new LinkedList<String>();  //it's a doubly linked list.
//can behave like a stack or queue.

//stack
linkedlist.push("A");
linkedlist.push("B");
linkedlist.push("C");
linkedlist.push("D");
linkedlist.push("E");

System.out.println(linkedlist);  //[E,D,C,B,A]

System.out.println(linkedlist.pop());  //E
// System.out.println(linkedlist.poll()); //same as above

System.out.println(linkedlist);   //[D,C,B,A]

linkedlist.pop();
linkedlist.pop();
linkedlist.pop();
linkedlist.pop();

//queue
linkedlist.offer("A");
linkedlist.offer("B");
linkedlist.offer("C");
linkedlist.offer("D");
linkedlist.offer("E");

System.out.println(linkedlist);   //[A,B,C,D,E]

System.out.println(linkedlist.poll());  //A

System.out.println(linkedlist);   //[B,C,D,E]

linkedlist.add(3,"d");

System.out.println(linkedlist);    //[B,C,D,d,E]

linkedlist.remove("D");

System.out.println(linkedlist);     //[B,C,d,E]
```

```
        System.out.println("Index of d: "+linkedlist.indexOf("d"));  //2

        //System.out.println("First element: "+linkedlist.peek());  //same as below one.
        System.out.println("First element: "+linkedlist.peekFirst());  //B
        System.out.println("Last element: "+linkedlist.peekLast());  //E

        linkedlist.addFirst("0");
        linkedlist.addLast("F");

        System.out.println(linkedlist);    //[0,B,C,d,E,F]

        String first = linkedlist.removeFirst();
        String last = linkedlist.removeLast();

        System.out.println(linkedlist);  //[B,C,d,E]
    }
}
```

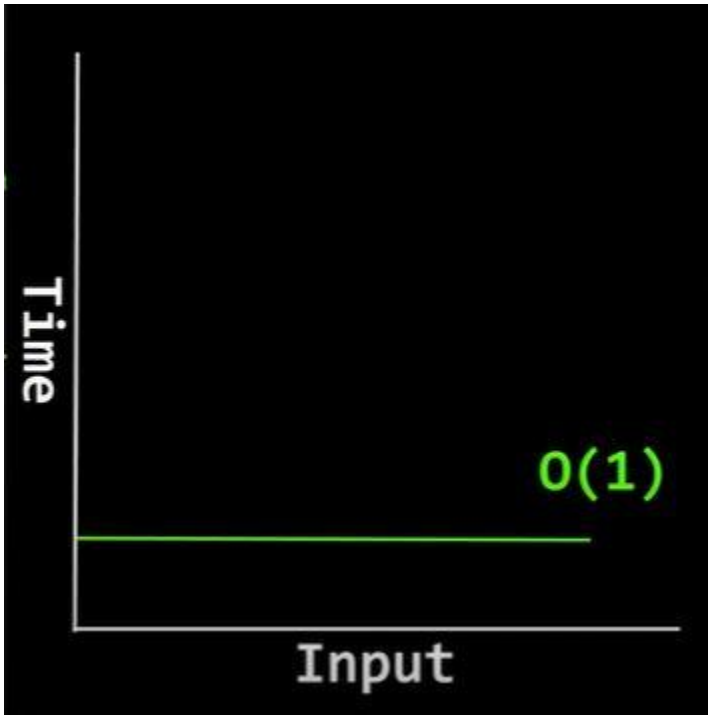



LinkedLists
(doubly)
Node<E> head = 123 Fake St.
Node<E> tail = 404 Nowhere Ln
Address
Data
Address
null
"A"
101
Help Blvd
Prev
Next
Address
Data
Address
101
Help Blvd
"C"
null
Prev
Next
Address
Data
Address
123
Fake St
"B"
404
Nowhere Ln
Prev
Next

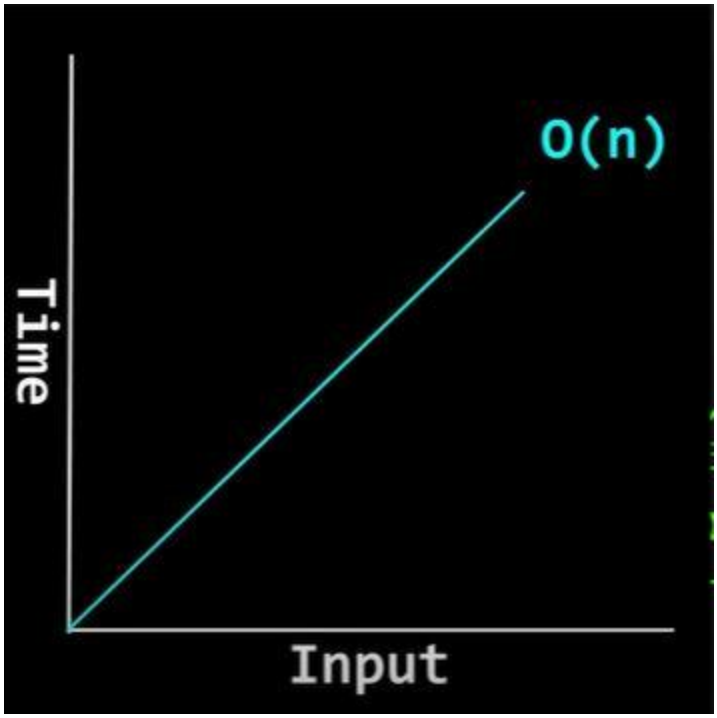

LinkedLists
(singly)
Node<E> head = 123 Fake St.
Data
Address
"A"
101
Help Blvd
Next
Data
Address
"B"
404
Nowhere Ln
Next
Data
Address
"C"
777
Lucky Ave
Next
Data
Address
"D"
null
Next
Data
Address
"?"
666
Crime Cir
Next
Insert



Insertion or Deletion T(n) = O(1):



searching T(n) = O(n):



Dynamic Array

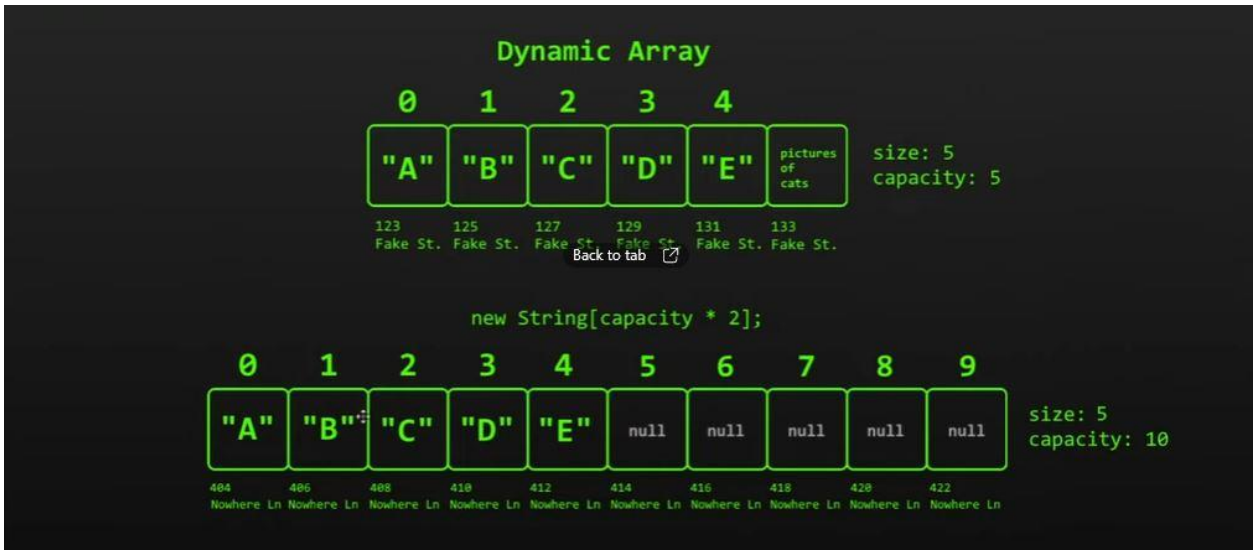
```
import java.util.ArrayList;

public class App {
    public static void main(String[] args) throws Exception {

        // ArrayList<String> arrayList = new ArrayList<String>();      //it's also a dynamic array to work with.

        DynamicArray dynamicArray = new DynamicArray();  //default size is 10, Or we can give our size here.

        dynamicArray.add("1");
        dynamicArray.add('2');
        dynamicArray.add(3);

        dynamicArray.insert(0, "0");
        dynamicArray.insert(40, "5");
        dynamicArray.delete("1");

        System.out.println("2 is at: "+dynamicArray.search('2'));
        System.out.println(dynamicArray);
        System.out.println("Size: "+dynamicArray.getSize());
```

```
        System.out.println("Capacity: "+dynamicArray.getCapacity());
        System.out.println("Is Empty: "+dynamicArray.isEmpty());

    }
}
/**
 * DynamicArray
 */
public class DynamicArray {

    int size;      //default value size = 0
    int capacity = 10;
    Object[] array;

    public DynamicArray() {
        this.array = new Object[capacity];
    }

    public DynamicArray(int capacity) {
        this.capacity = capacity;
        this.array = new Object[capacity];
    }

    public int getSize() {
        return size;              //size of array.
    }

    public int getCapacity() {
        return capacity;              //capacity of array.
    }

    public Object get(int index) {
        return array[index];
    }

    public void add(Object data)
    {
        if(size >= capacity)
        {
            grow();
        }
        array[size] = data;
        size++;
    }
```

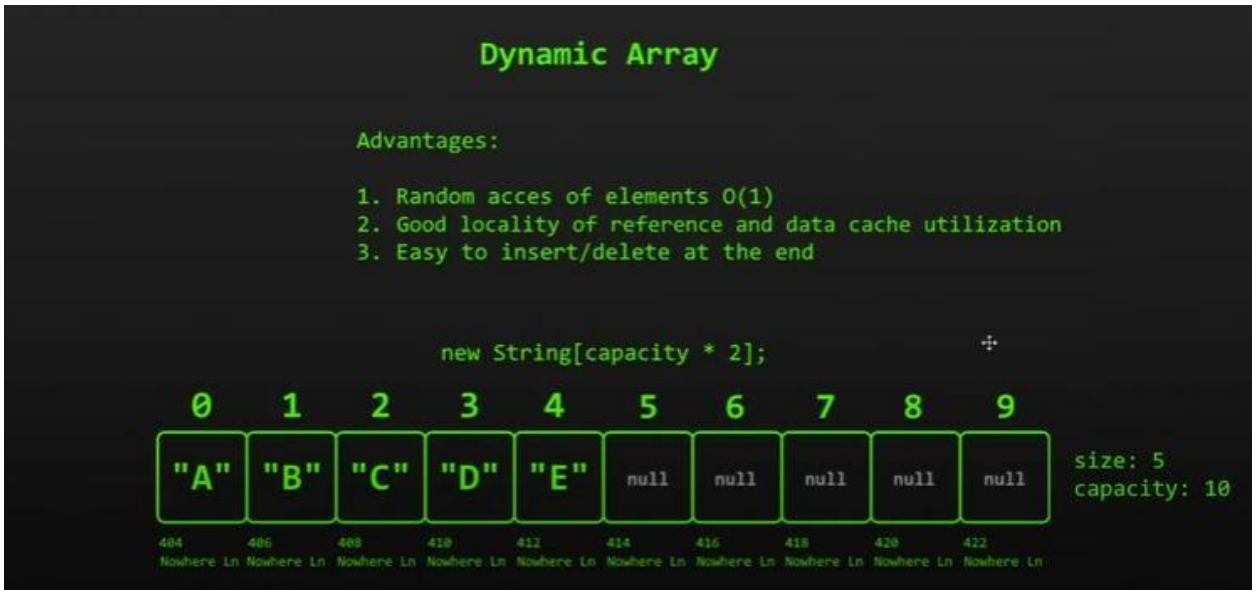
```
public void insert(int index, Object data)
{
   if(index<size)
   {
      if(size >= capacity)
      {
         grow();
      }
      for(int i = size; i > index; i--)
      {
         array[i] = array[i - 1];
      }
      array[index] = data;
      size++;
   }
   else
   {
      System.out.println(data+" can't be added at index "+index);
   }
}

public void delete(Object data)
{
   for(int i = 0; i < size; i++) {
      if(array[i] == data) {
         // for(int j = 0; j < (size - i - 1); j++){
         //  array[i + j] = array[i + j + 1];
         // }
         for(int j=i;j<size-1;j++)   //same as above.
         {
            array[j] = array[j+1];
         }
         array[size - 1] = null;
         size--;
         if(size <=(int) (capacity/3)) {
            shrink();
         }
         break;
      }
   }
}

public int search(Object data) {
```

```
    for(int i = 0; i < size; i++) {
        if(array[i] == data) {
            return i;
        }
    }
    return -1;
}

private void grow() {

    int newCapacity = (int)(capacity * 2);
    Object[] newArray = new Object[newCapacity];

    for(int i = 0; i < size; i++) {
        newArray[i] = array[i];
    }
    capacity = newCapacity;
    array = newArray;
}

private void shrink() {

    int newCapacity = (int)(capacity / 2);
    Object[] newArray = new Object[newCapacity];

    for(int i = 0; i < size; i++) {
        newArray[i] = array[i];
    }
    capacity = newCapacity;
    array = newArray;
}

public boolean isEmpty() {
    return size == 0;
}

public String toString() {

    String string = "";

    for(int i = 0; i < capacity; i++) {
        string += array[i] + ", ";
    }
```

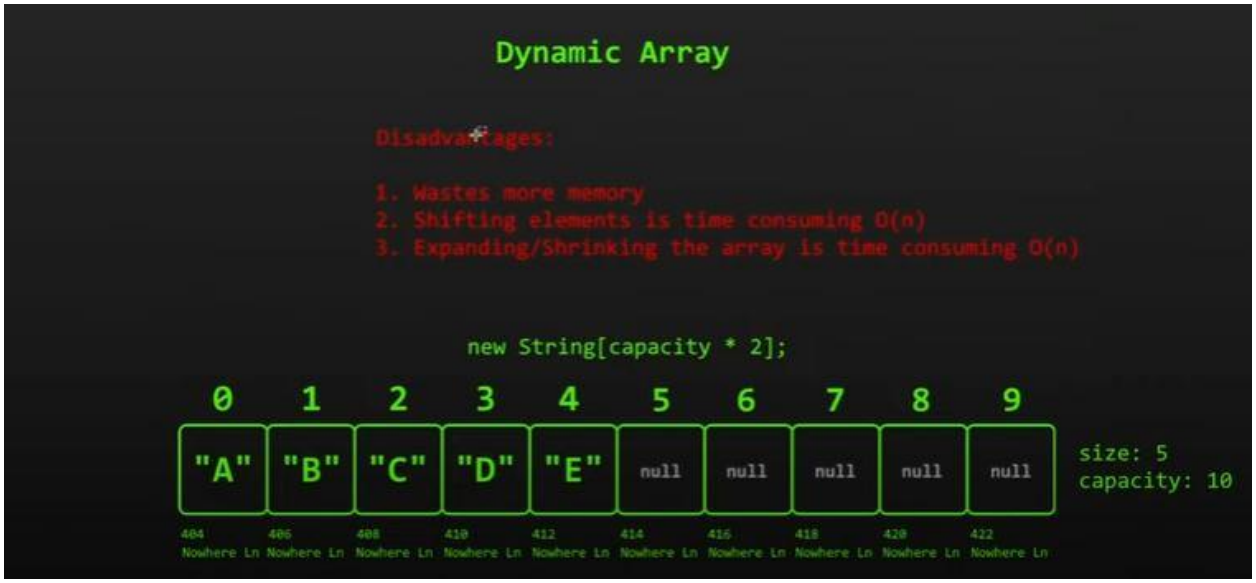
```
        if(string != "") {
            string = "[" + string.substring(0, string.length() - 2) + "]";
        }
        else {
            string = "[]";
        }
        return string;
    }
}
```



Advantages:



Disadvantages:



LinkedList vs ArrayList

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

public class App {
    public static void main(String[] args) throws Exception {

        LinkedList<Integer> linkedList =  new LinkedList<Integer>();
        ArrayList<Integer> arrayList = new ArrayList<Integer>();

        long startTime;
        long endTime;
        long elapsedTime;

        for(int i=0; i<10000; i++){
            linkedList.add(i);
            arrayList.add(i);
        }

        //----------------------------LinkedList----------------------------
        startTime = System.nanoTime();
```

```
        //do something.

        //-----------------much time
        // linkedList.get(0);          //21600
        // linkedList.get(5000);       //185000
        // linkedList.get(9999);       //34400


        // linkedList.remove(0);       //39000
        // linkedList.remove(5000);    //191600
        linkedList.remove(9999);    //34800

        endTime = System.nanoTime();
        elapsedTime = endTime - startTime;

        System.out.println("Time taken by LinkedList: "+elapsedTime);

        //------------------------------ArrayList-----------------------------

        startTime = System.nanoTime();
        //do something.

        //------------Less time
        // arrayList.get(0);            //14100
        // arrayList.get(5000);         //70400
        // arrayList.get(9999);         //24400

        // arrayList.remove(0);         //288200  //much time because of shifting
        // arrayList.remove(5000);      //73500 //less time
        arrayList.remove(9999);    //24000      //less time because of no shifting


        endTime = System.nanoTime();
        elapsedTime = endTime - startTime;

        System.out.println("Time taken by ArrayList: "+elapsedTime);

    }
}
```

Big O Notation

Big O notation

"How code slows as data grows."

1. Describes the performance of an algorithm as the amount of data increases
2. Machine independent (# of steps to completion)
3. Ignore smaller operations O(n + 1) -> O(n)

example:
O(1)
O(n)
O(log n)
O(n^2)

n = amount of data
(it's a variable like x)

Big O notation

"How code slows as data grows."

O(n) linear time

```
int addUp(int n){

    int sum = 0;
    for(int i = 0; i <= n; i++) {
        sum += i;
    }
    return sum;
}
```

n = 1000000

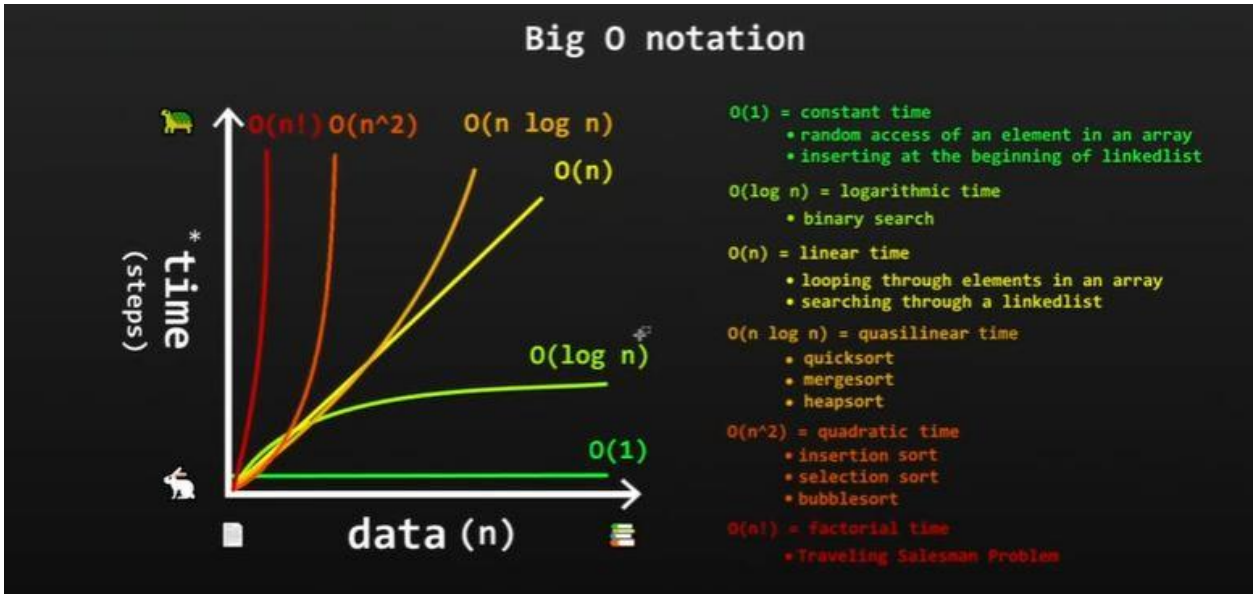
~1000000 steps

O(1) constant time

```
int addUp(int n){

    int sum = n * (n + 1) / 2;
    return sum;
}
```

n = 1000000

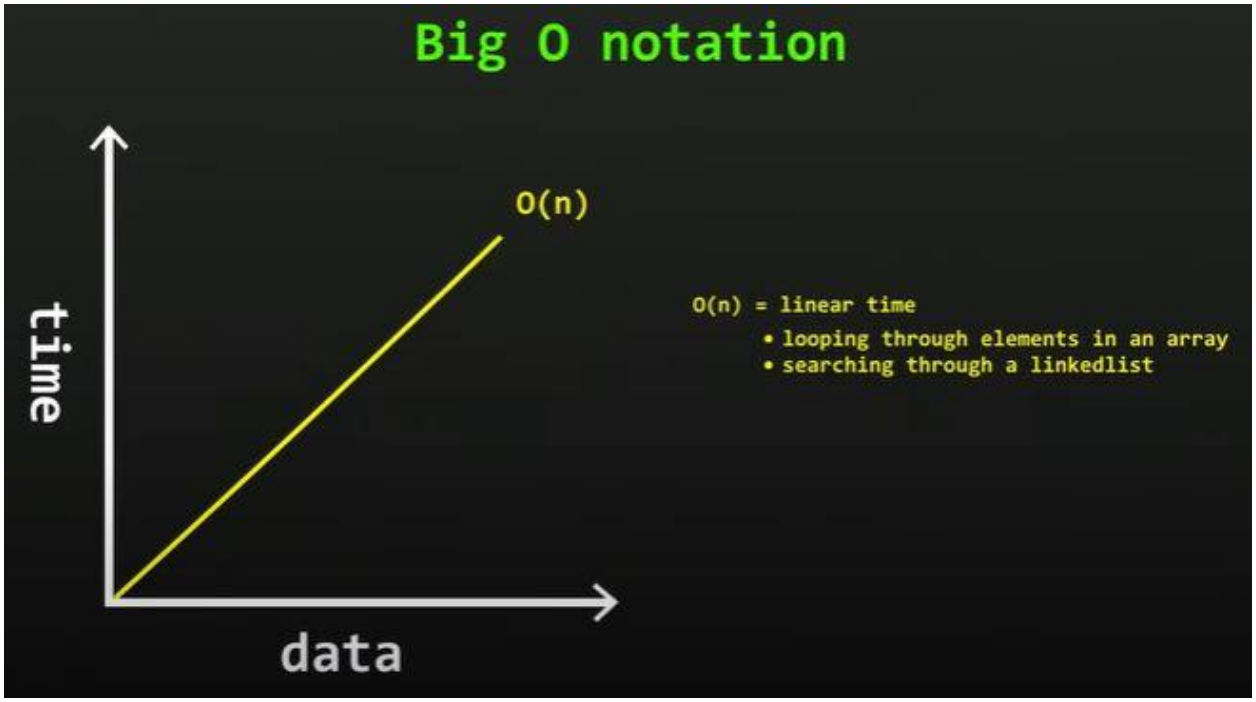
3 steps



Linear Search

```
import java.util.Scanner;

public class App {
    public static void main(String[] args) throws Exception {

        //Linear Search =   //Iterate through a collection one element at a time.

        //runtime complexity : O(n)

        //Disadvantages:
        //slow for large data set.
        //elements are not necessary to be sorted.

        //Advantages:
        //Fast for searches of small to medium dataset.
        //Does not need to be sorted.
        //Useful for data structures that do not have random access.

        int[] array = new int[10];

        for(int i=0;i<array.length;i++){
            array[i]=i+1;
        }

        System.out.print("Enter a number to search in array: ");
```

```
        Scanner scanner = new Scanner(System.in);

        int val = scanner.nextInt();

        int index = LinearSearch(array,val);

        if(index!=-1)
        {
            System.out.println("Element found at : "+index);
        }
        else
        {
            System.out.println("Element was nowhere in the picture.");
        }

        scanner.close();
    }

    public static int LinearSearch(int[] arr, int val)
    {
        for(int i=0;i<arr.length;i++)
        {
            if(arr[i]==val)
            {
                return i;
            }
        }
        return -1;
    }
}
```



Binary Search

```
import java.util.Arrays;

public class App {
    public static void main(String[] args) throws Exception {
        // binary search  = Search algorithm that finds the position
        //                  of a target value within a sorted array.
        //                  Half of the array is eliminated during each "step"
        //                  larger the dataset binary search becomes more efficient.
        int array[] = new int[100];

        int target = 42;

        for(int i=0;i<array.length;i++)
        {
            array[i] = i;
        }

        // int index = Arrays.binarySearch(array, target);   //built in (cheap way to do it.)
        //Let's create our own function now.
        int index = binarySearch(array, target);
```

```
        if(index == -1)
        {
            System.out.println(target +" not found.");
        }
        else
        {
            System.out.println(target +" found at index: "+index);
        }
    }

    public static int binarySearch(int[] array, int target)
    {
        int low = 0;
        int high = array.length-1;

        while(low<=high)
        {
            int middle = low+(high-low)/2;
            int value = array[middle];

            System.out.println("Middle value is: "+value);

            if(value<target) low =middle + 1;
            else if (value>target) high = middle - 1;
            else return middle;
        }

        return -1;
    }
}
```

Binary Search

Search algorithm that finds the position of a target value within a sorted array. Half of the array is eliminated during each "step"

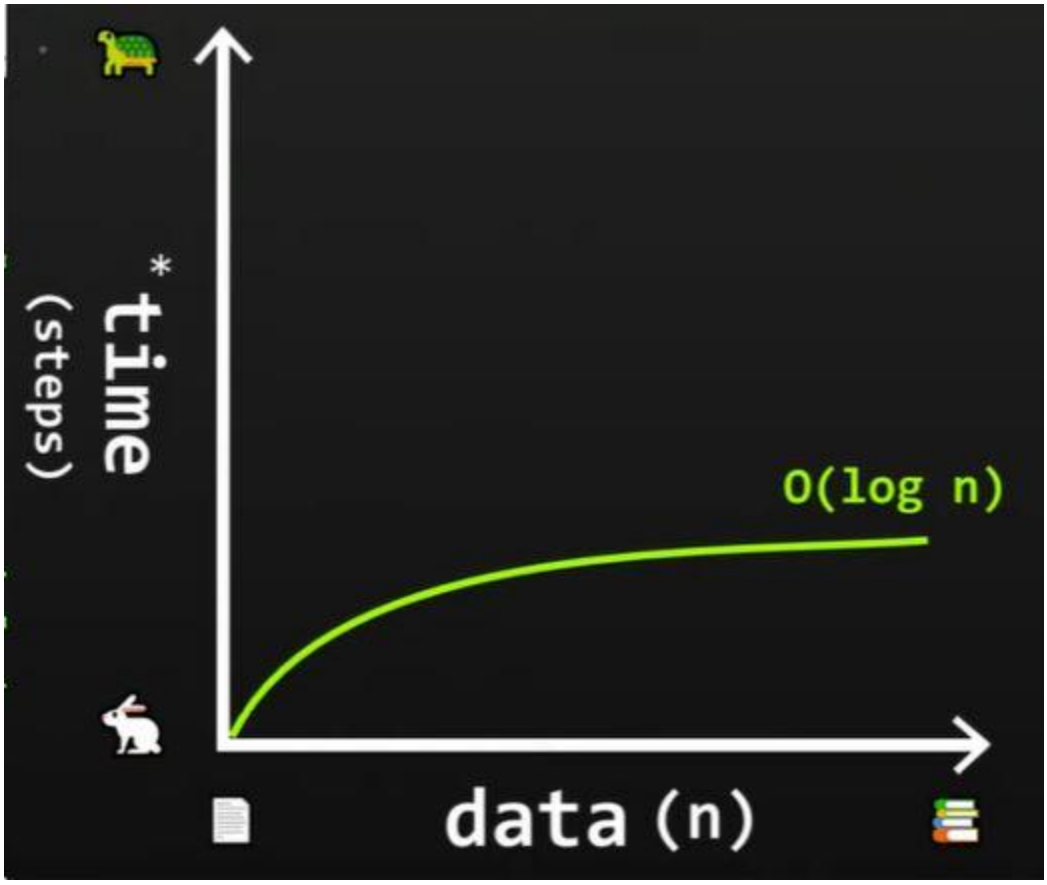
```
value = "H";
index = ?;
```

Binary Search

Search algorithm that finds the position of a target value within a sorted array. Half of the array is eliminated during each "step"

```
value = "H";
index = ?;
```

after 4 iterations



Interpolation Search

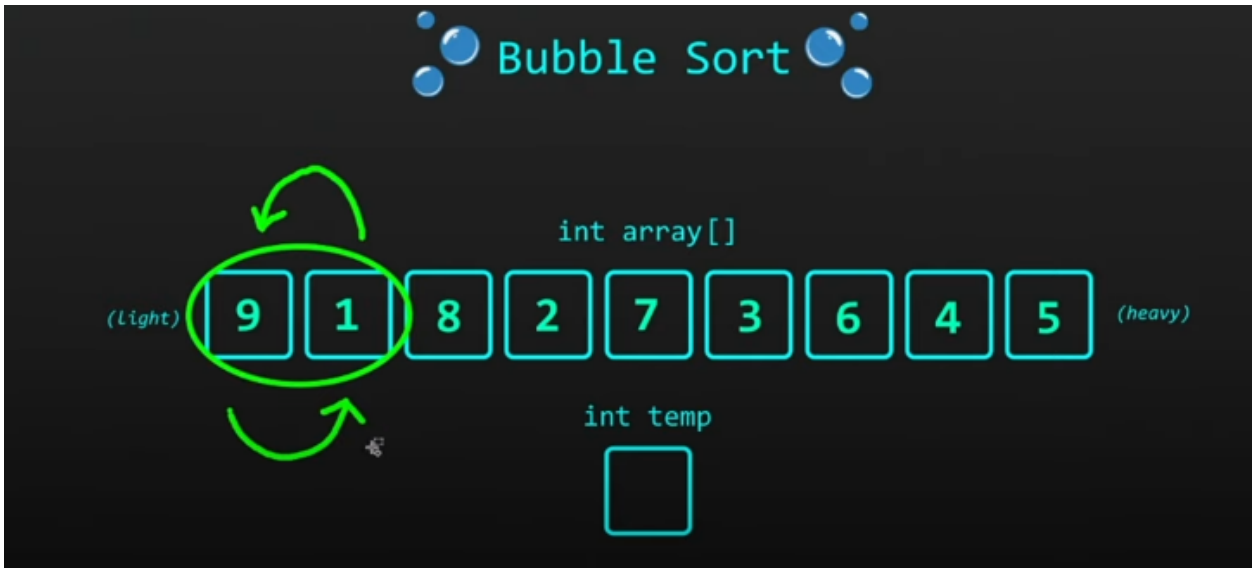
```
public class App {
    public static void main(String args[]){

        //interpolation search =  improvement over binary search best used for "uniformly" distributed data
        //                    "guesses" where a value might be based on calculated probe results
        //                       if probe is incorrect, search area is narrowed, and a new probe is calculated

        //                    average case: O(log(log(n)))
        //                    worst case: O(n) [values increase exponentially]

        int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9};

        int index = interpolationSearch(array, 8);

        if(index != -1) {
            System.out.println("Element found at index: "+ index);
```

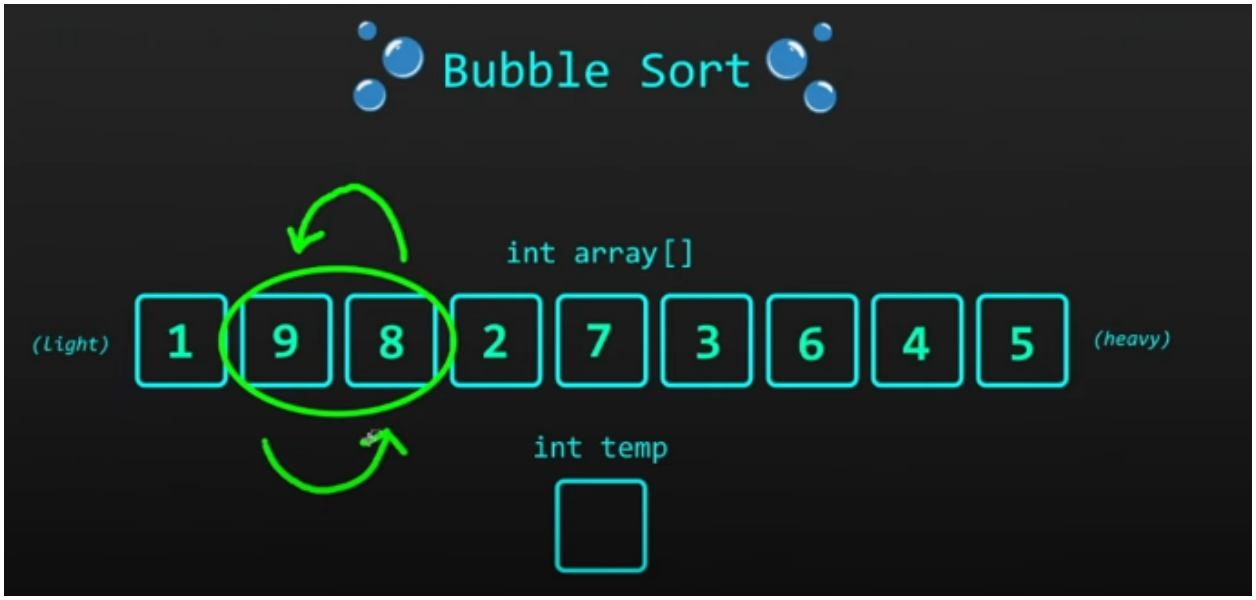
```
        }
        else {
            System.out.println("Element not found");
        }
    }

    private static int interpolationSearch(int[] array, int target) {

        int low = 0;
        int high = array.length - 1;

        while(target >= array[low] && target <= array[high] && low <= high)
        {
            int probe = low + (high - low) * (target - array[low]) / (array[high] - array[low]);

            int value = array[probe];

            System.out.println("value at probe: " + value + " with probe: "+probe);

            if(value < target) low = probe + 1;
            else if(value > target) high = probe - 1;
            else return probe;
        }

        return -1;
    }
}
```

Bubble sort

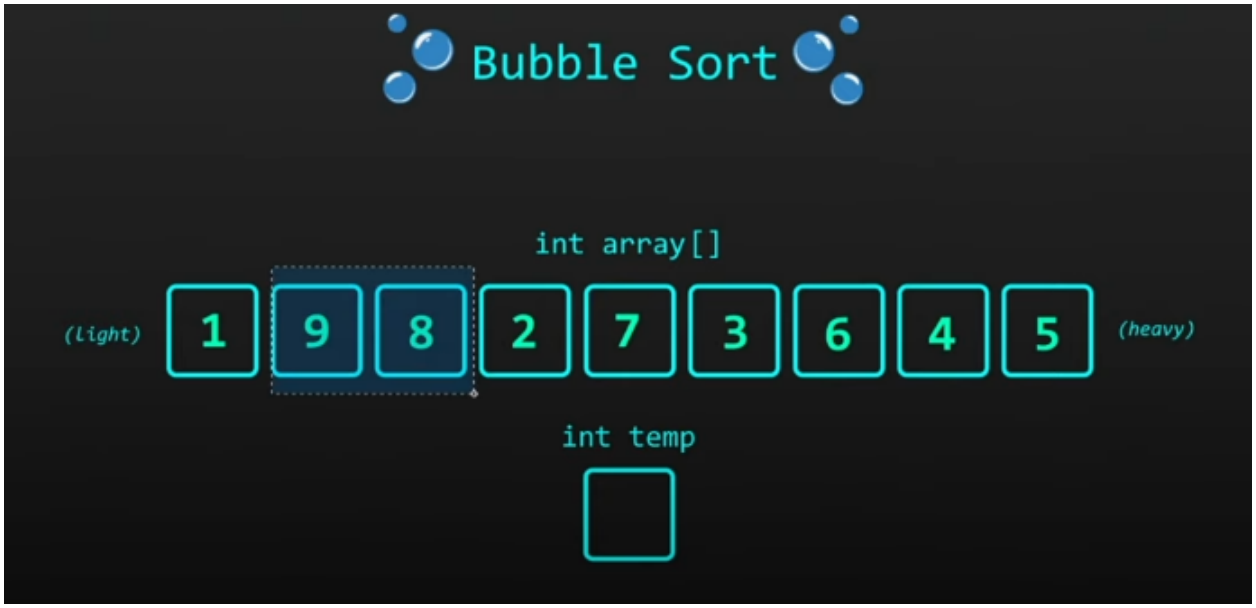
```
// bubble sort = pairs of adjacent elements are compared, and the elements
        //              swapped if they are not in order.

        //    Quadratic time O(n^2)
        //    small data set = okay-ish
        //    large data set = BAD (plz don't)

        int []arr = {3,8,4,7,9,5,1,6,2};

        bubbleSort(arr);

        System.out.println("Ascending Order: ");  //new line
```

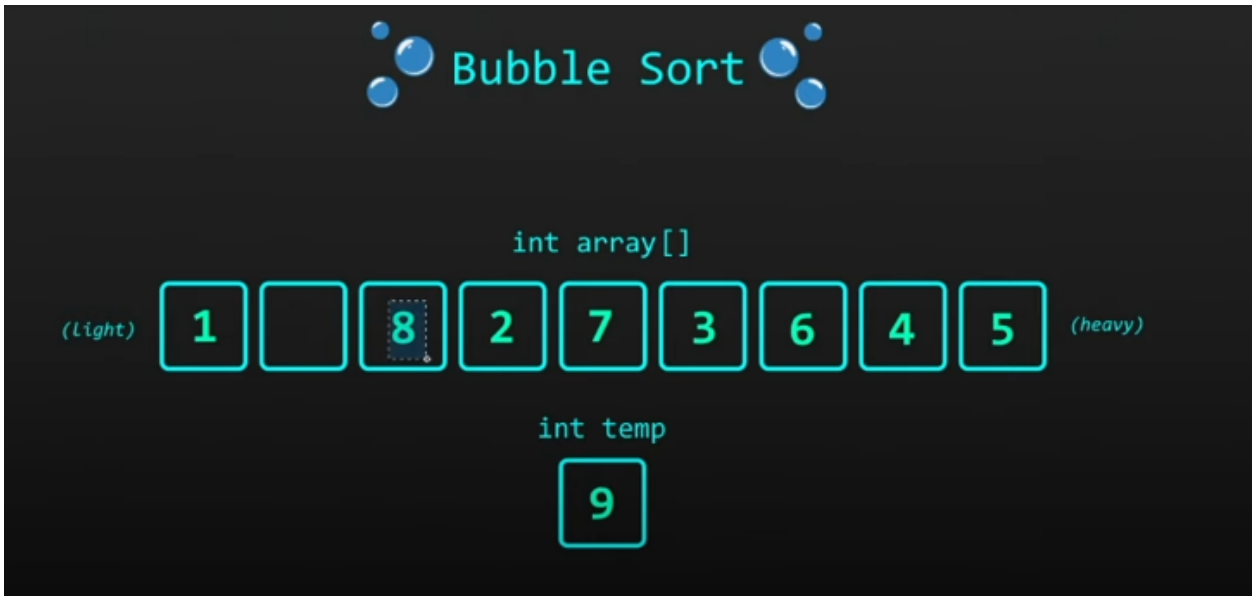
```
    for(int i : arr)
    {
        System.out.print(i);
    }
public static void  bubbleSort(int[] arr)
{
    for(int i=0;i<arr.length-1; i++)
    {
        for(int j=0;j<arr.length-i-1;j++)
        {
            if(arr[j]>arr[j+1])
            {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

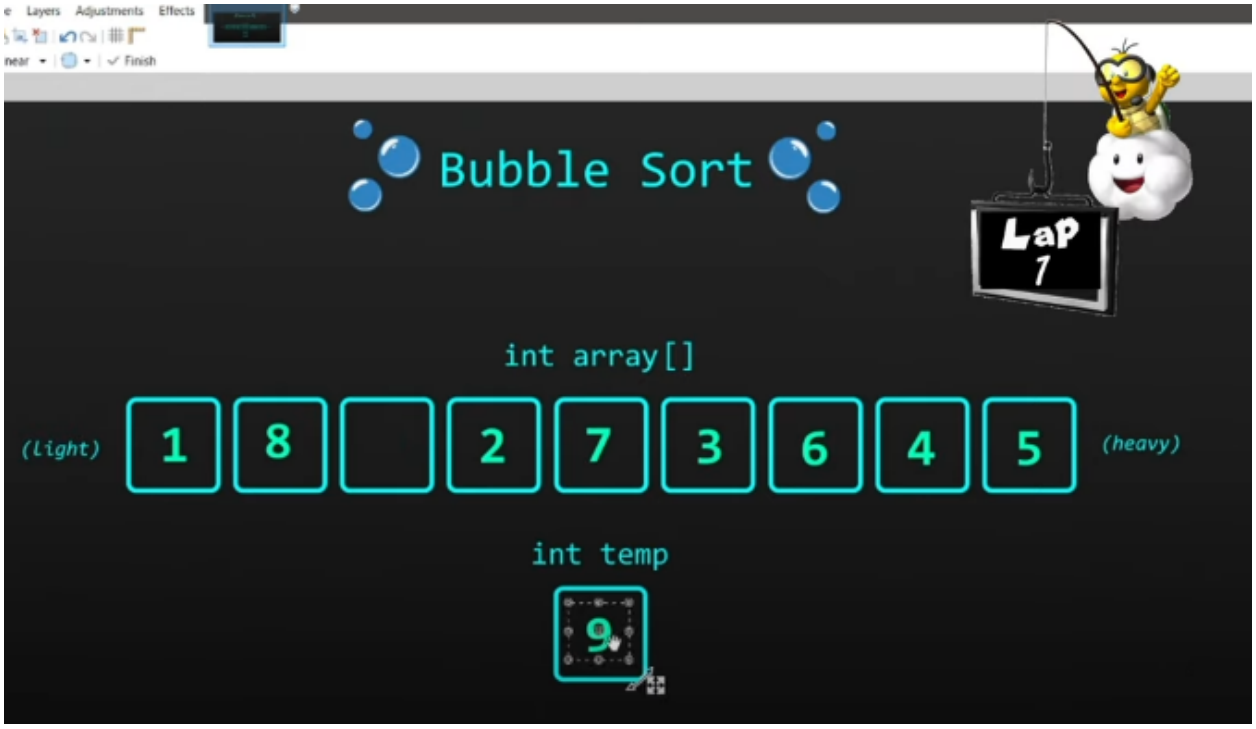


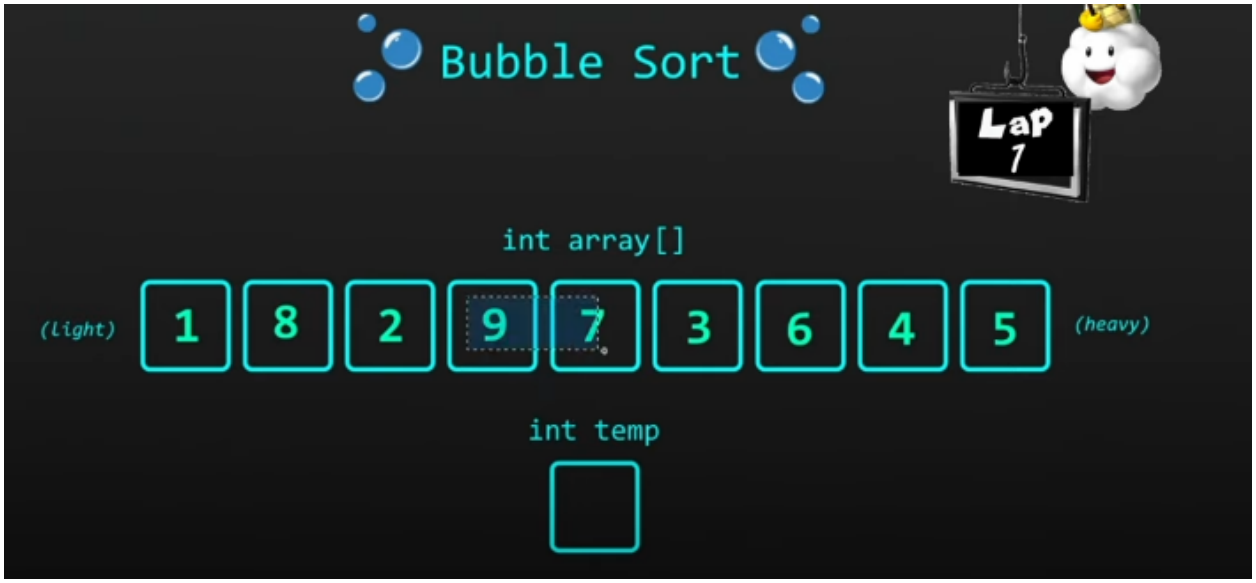

Bubble Sort
int array[]
(light)
1
9
8
2
7
3
6
4
5
(heavy)
int temp

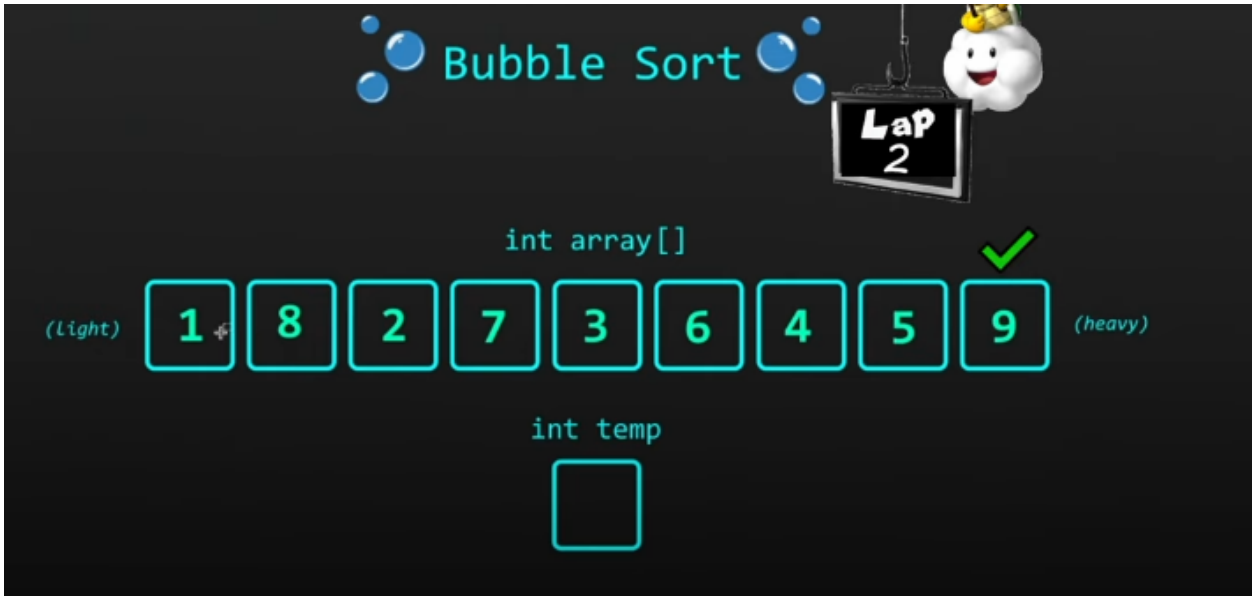

Bubble Sort
int array[]
(light)
1
8
2
7
3
6
4
5
(heavy)
int temp
9

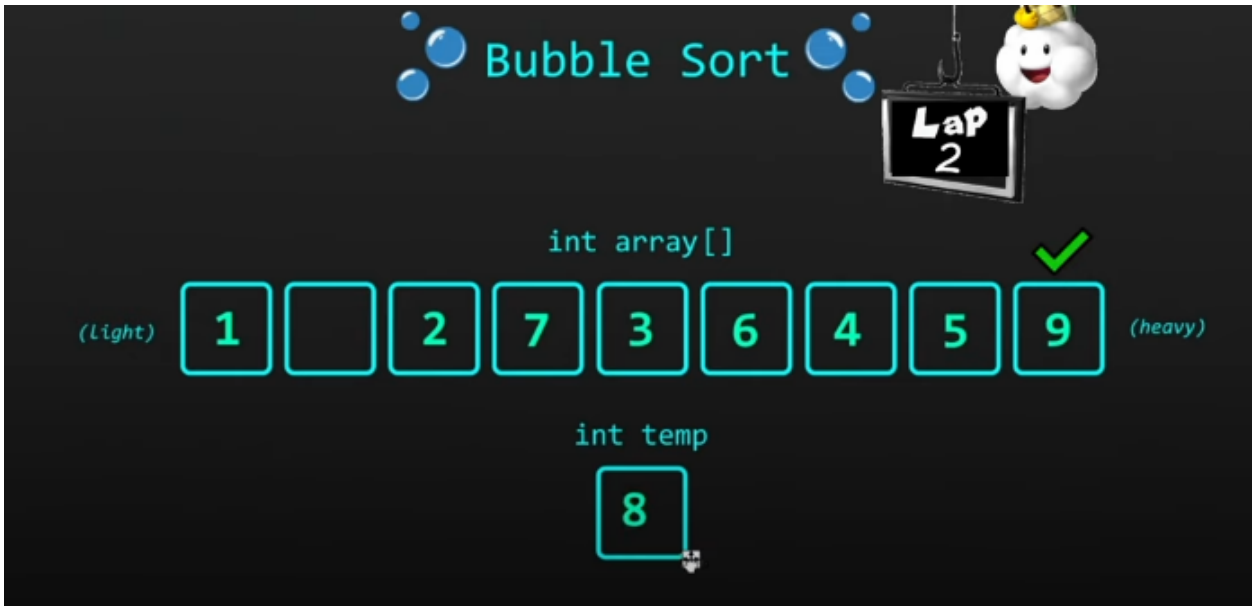

Bubble Sort
int array[]
(light)
1
9
8
2
7
3
6
4
5
(heavy)
int temp

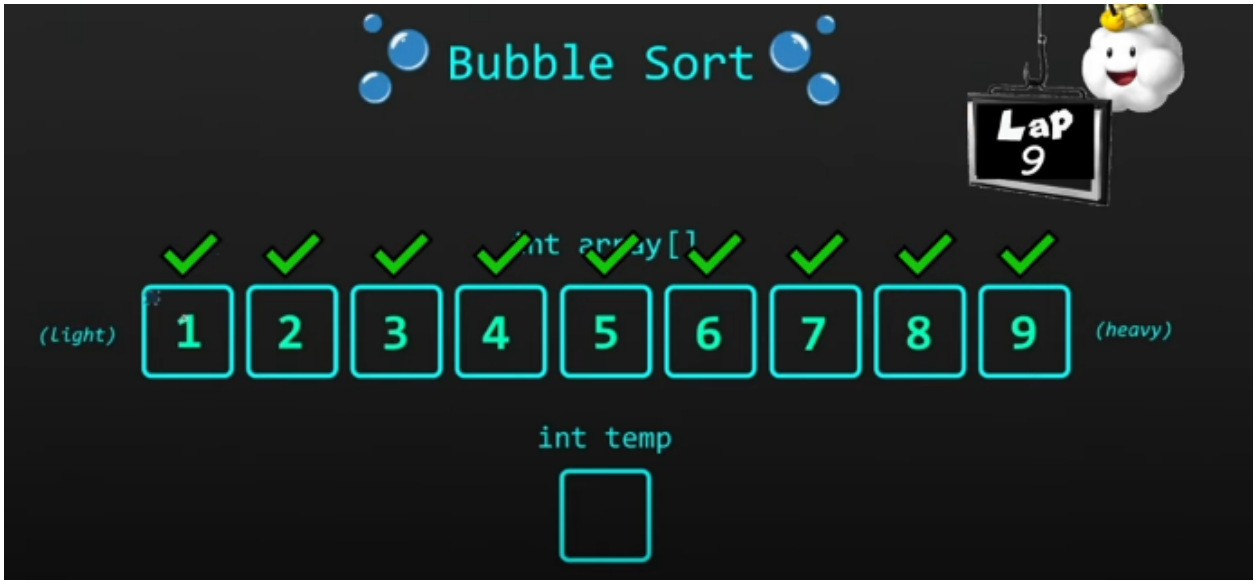

Bubble Sort
int array[]
(light)
1
8
2
7
3
6
4
5
(heavy)
int temp
9

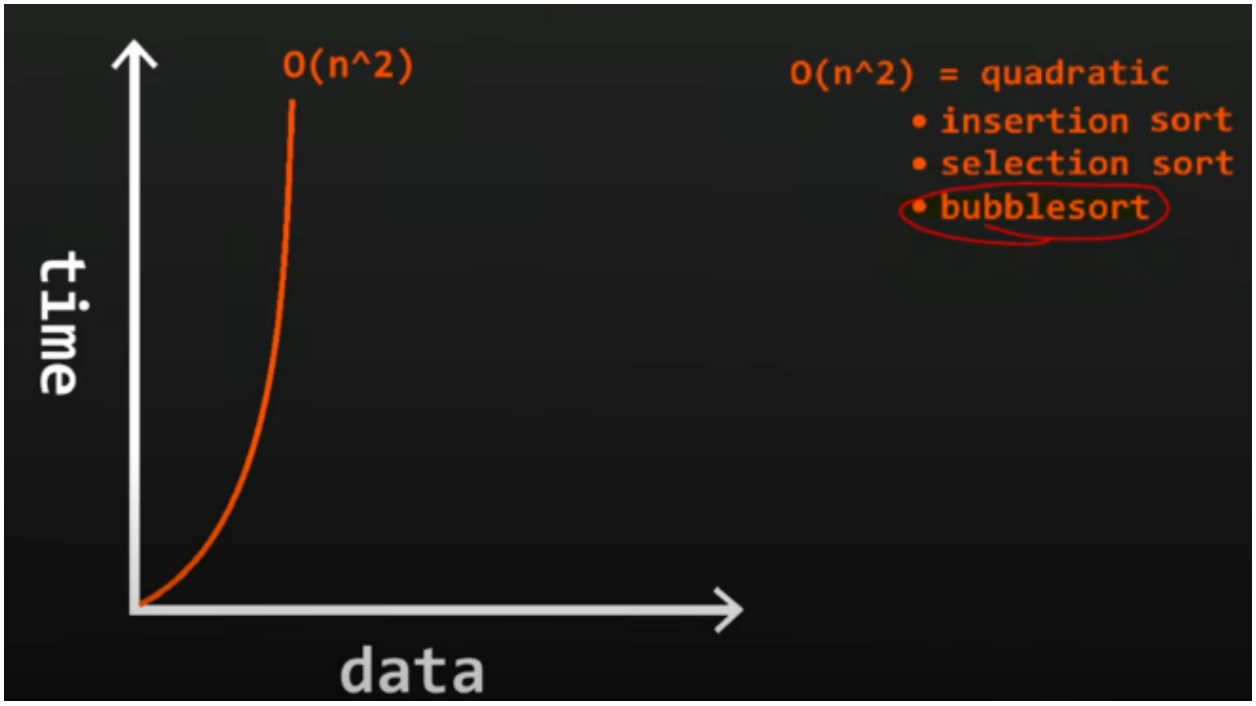

Layers
Adjustments
Effects
Finish
Bubble Sort
Lap 1
int array[]
(light)
1
8
2
7
3
6
4
5
(heavy)
int temp
9


Bubble Sort
Lap 1
int array[]
(light)
1
8
2
9
7
3
6
4
5
(heavy)
int temp


Bubble Sort
Lap 2
int array[]
(light)
1
8
2
7
3
6
4
5
9
(heavy)
int temp


Bubble Sort
Lap 2
int array[]
(light)
1
2
7
3
6
4
5
9
(heavy)
int temp
8



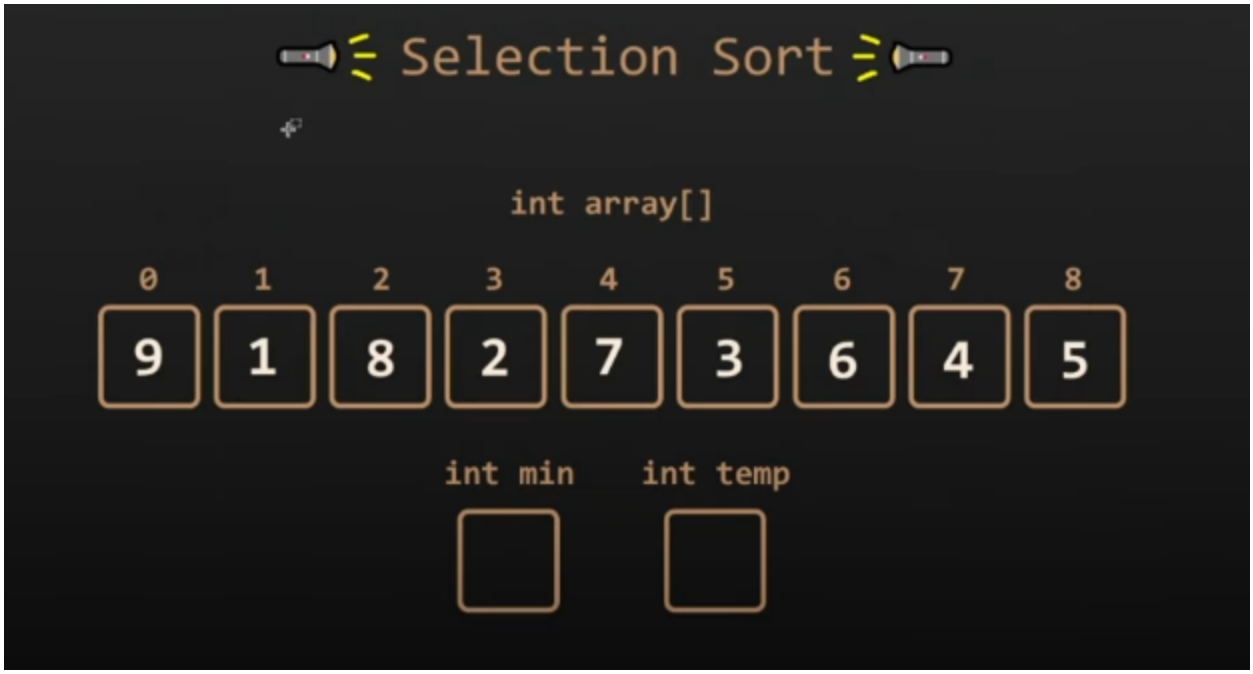


Selection Sort

// selection sort = search through an array and keep track of the minimum value during
// each iteration. At the end of each iteration, we swap values.

// Quadratic time O(n^2)
// small data set = okay
// large data set = BAD

```
public static void main(String[] args) {

    int array[] = {8, 7, 9, 2, 3, 1, 5, 4, 6};

    System.out.println("Ascending order: ");

    selectionSort(array);

    for(int i : array) {
        System.out.print(i);
    }
}

private static void selectionSort(int[] array) {

    for(int i = 0; i < array.length - 1; i++) {
        int min = i;
        for(int j = i + 1; j < array.length; j++) {
            if(array[min] > array[j]) {
                min = j;
            }
        }

        int temp = array[i];
        array[i] = array[min];
        array[min] = temp;
    }

}
```

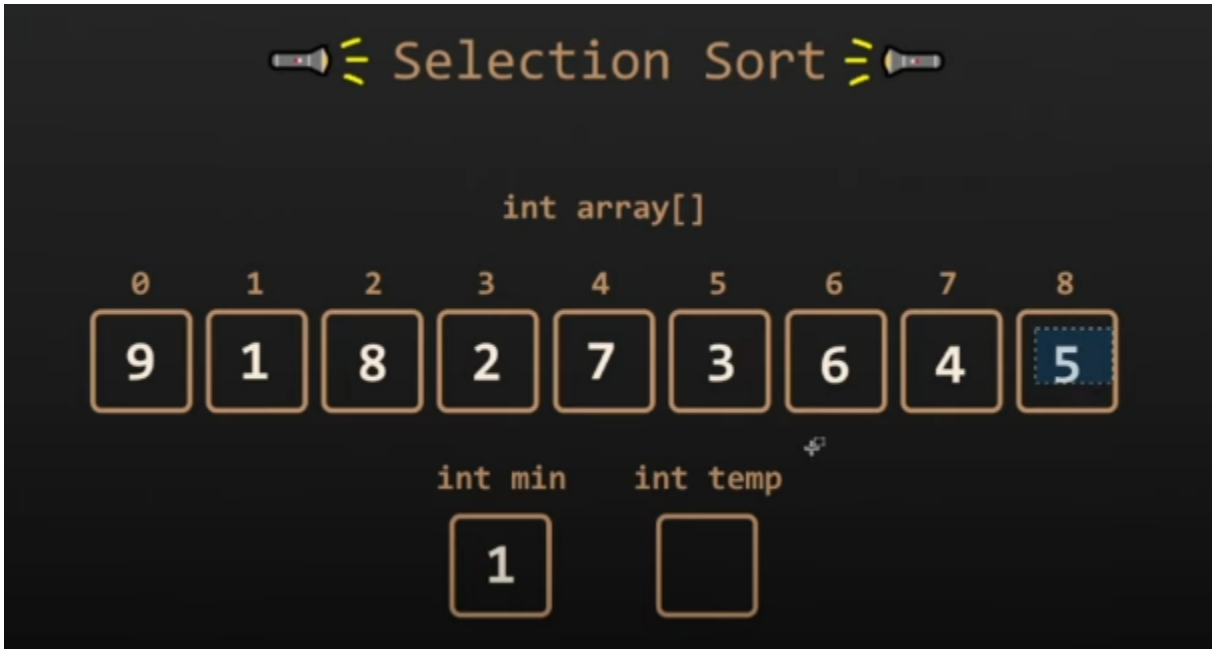

Selection Sort
int array[]
0 1 2 3 4 5 6 7 8
9 1 8 2 7 3 6 4 5
int min
int temp

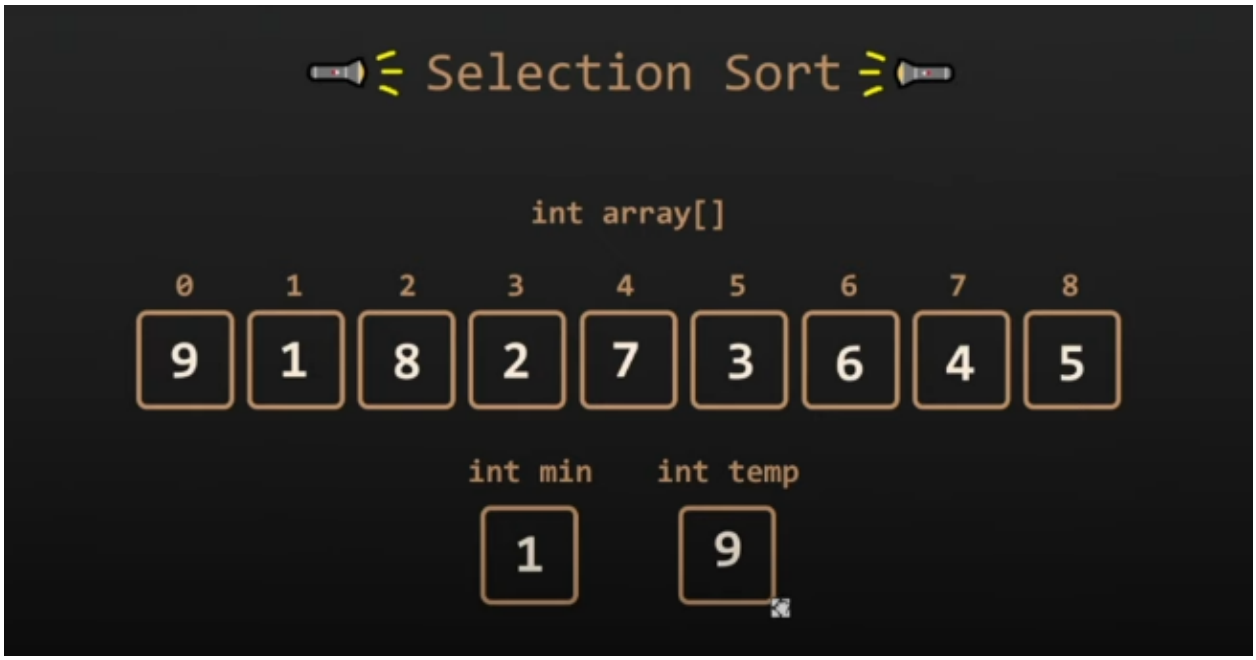

Selection Sort
int array[]
0 1 2 3 4 5 6 7 8
9 1 8 2 7 3 6 4 5
int min
int temp
9


Selection Sort
int array[]
0 1 2 3 4 5 6 7 8
9 1 8 2 7 3 6 4 5
int min
int temp
9

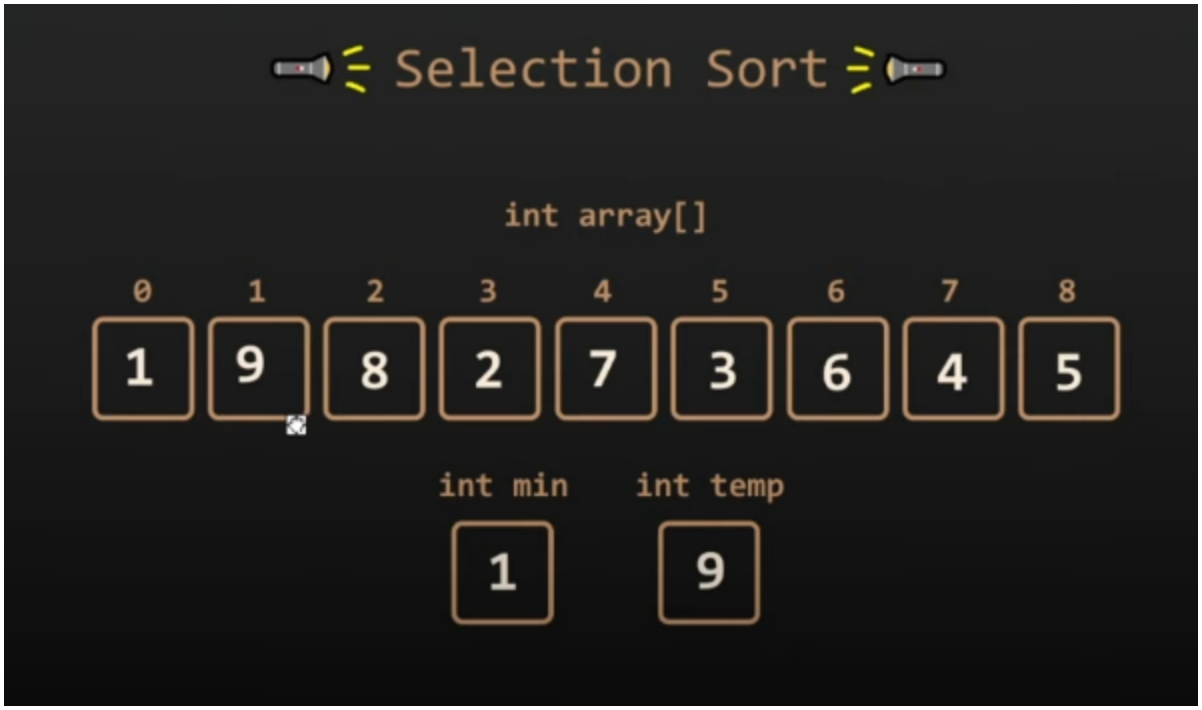

Selection Sort
int array[]
0 1 2 3 4 5 6 7 8
9 1 8 2 7 3 6 4 5
int min
int temp
1


Selection Sort
int array[]
0 1 2 3 4 5 6 7 8
9 1 8 2 7 3 6 4 5
int min
int temp
1


Selection Sort
int array[]
0 1 2 3 4 5 6 7 8
9 1 8 2 7 3 6 4 5
int min
int temp
1

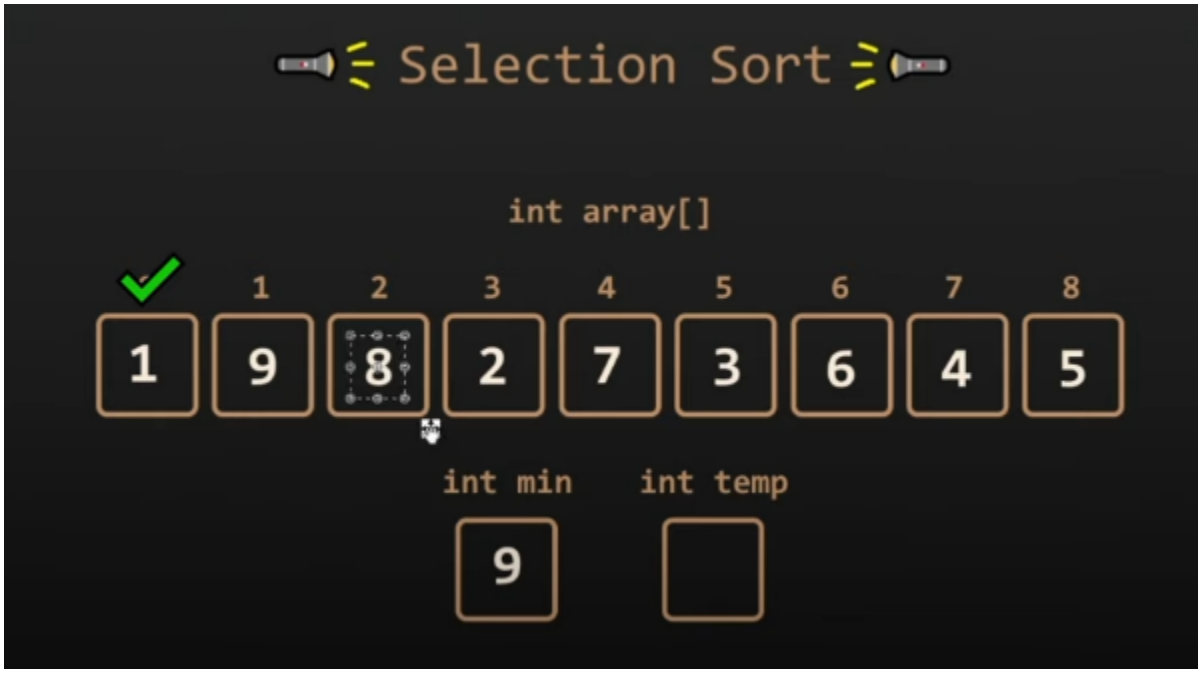

Selection Sort
int array[]
0 1 2 3 4 5 6 7 8
9 1 8 2 7 3 6 4 5
int min
int temp
1
9

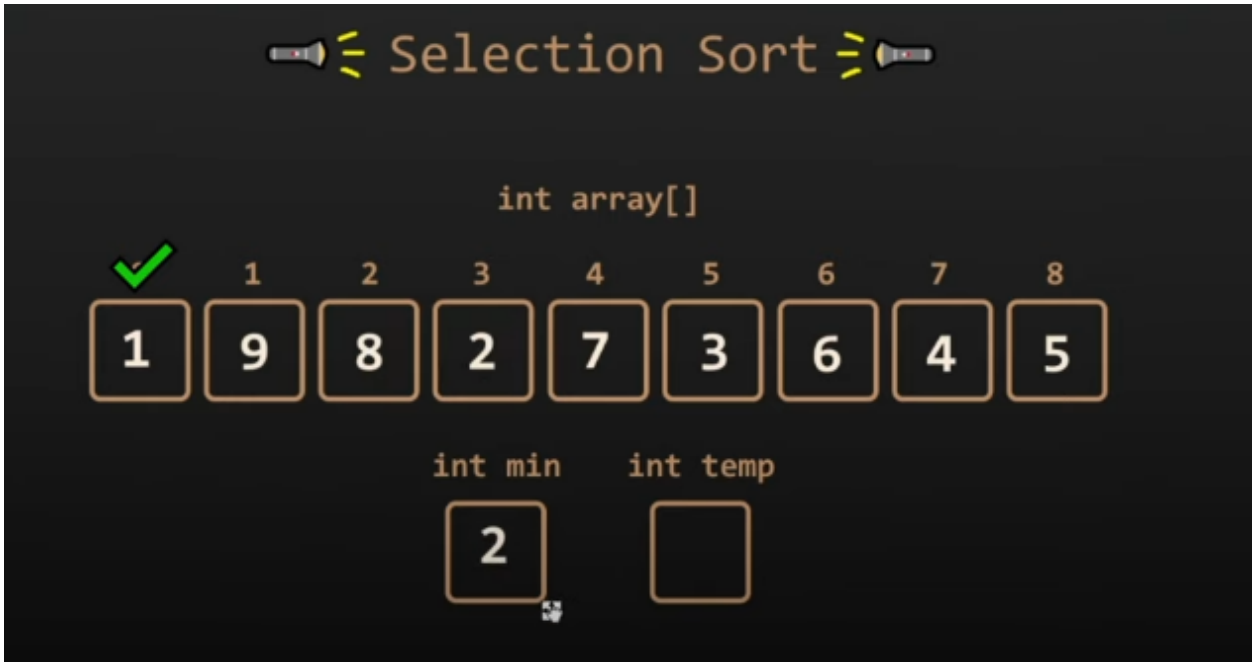

Selection Sort
int array[]
0 1 2 3 4 5 6 7 8
1 1 8 2 7 3 6 4 5
int min
int temp
1
9


Selection Sort
int array[]
0 1 2 3 4 5 6 7 8
1 9 8 2 7 3 6 4 5
int min
int temp
1
9


Selection Sort
int array[]
1 2 3 4 5 6 7 8
1 9 8 2 7 3 6 4 5
int min
int temp
1
9


Selection Sort
int array[]
1 2 3 4 5 6 7 8
1 9 8 2 7 3 6 4 5
int min
int temp

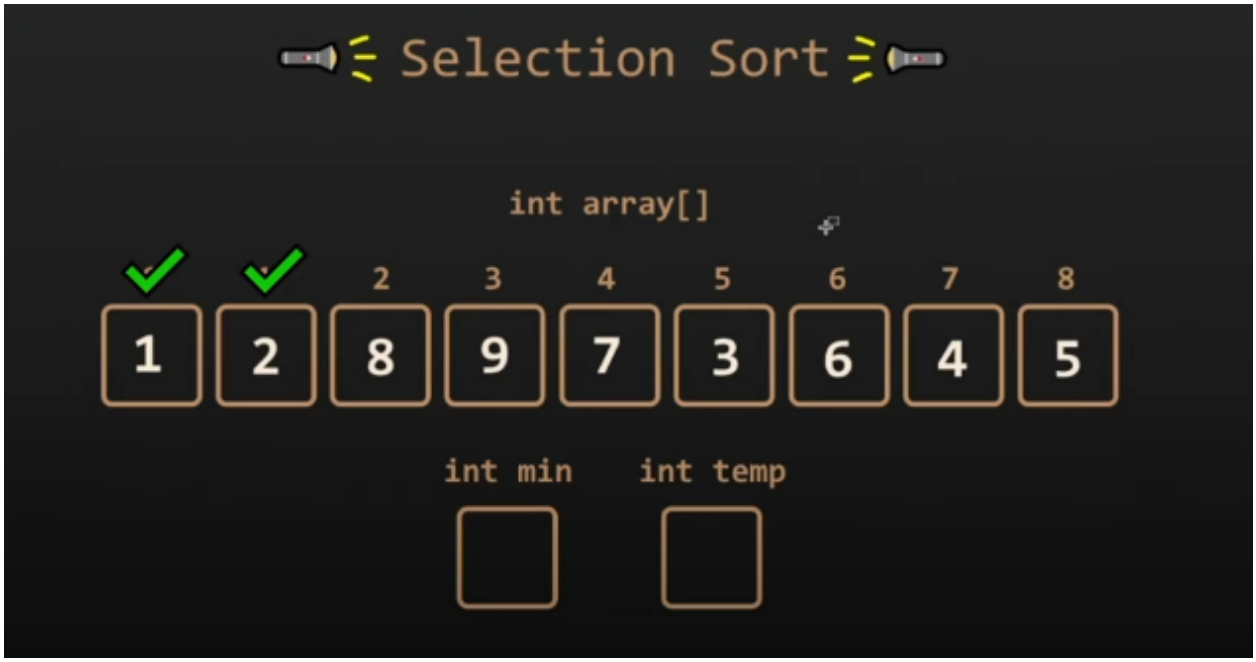

Selection Sort
int array[]
1 2 3 4 5 6 7 8
1 9 8 2 7 3 6 4 5
int min
int temp
9


Selection Sort
int array[]
1 2 3 4 5 6 7 8
1 9 8 2 7 3 6 4 5
int min
int temp
2

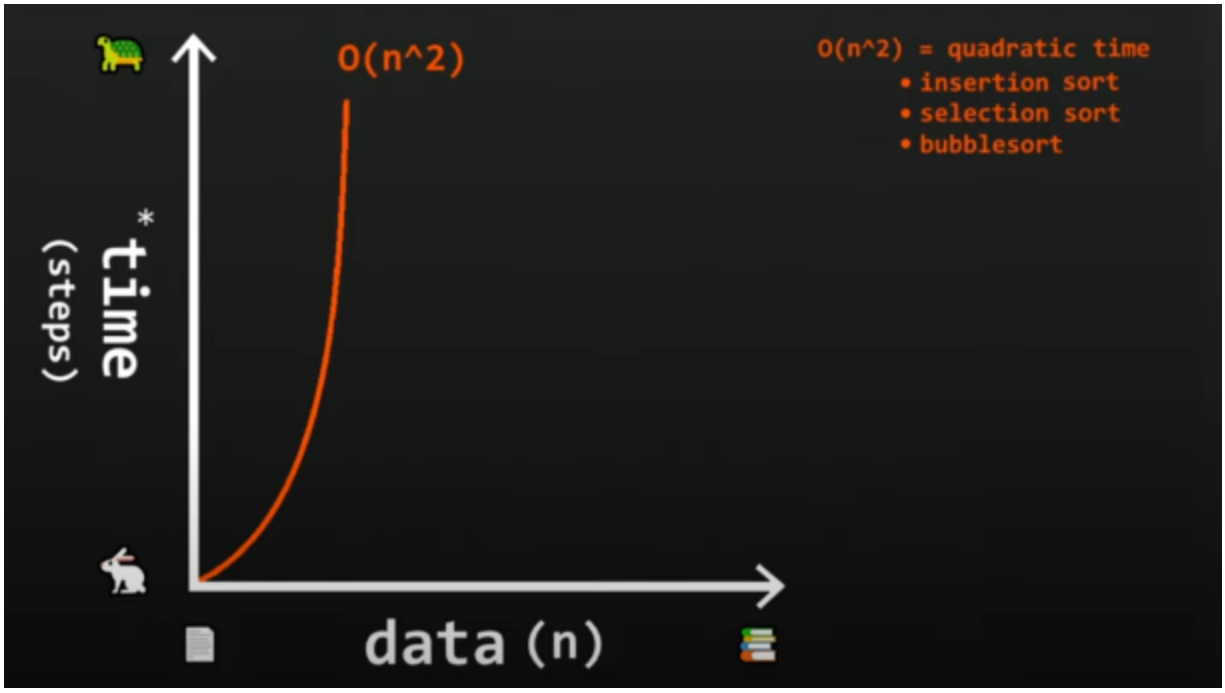

Selection Sort
int array[]
1 2 3 4 5 6 7 8
1 9 8 2 7 3 6 4 5
int min
int temp
2
9


Selection Sort
int array[]
1 2 3 4 5 6 7 8
1 2 8 2 7 3 6 4 5
int min
int temp
2
9


Selection Sort
int array[]
1 2 3 4 5 6 7 8
1 2 8 9 7 3 6 4 5
int min
int temp
2
9


Selection Sort
int array[]
2
3
4
5
6
7
8
1
2
8
9
7
3
6
4
5
int min
int temp

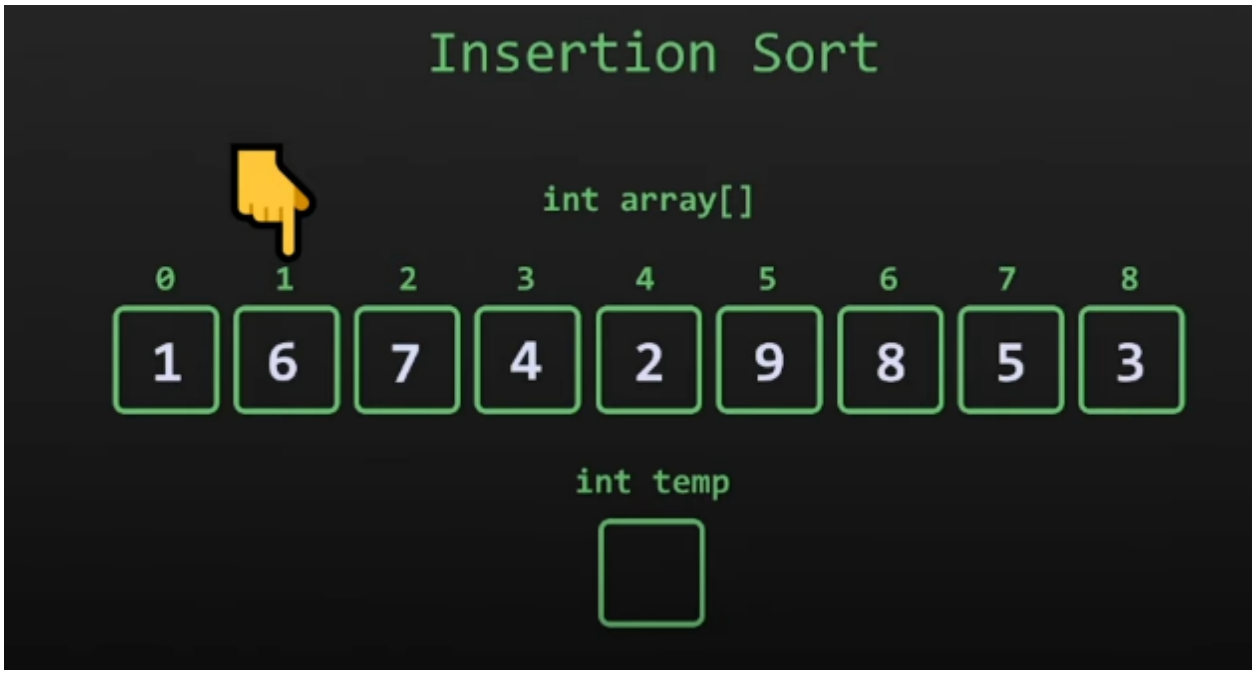

Selection Sort
int array[]
1
2
3
4
5
6
7
8
9
int min
int temp

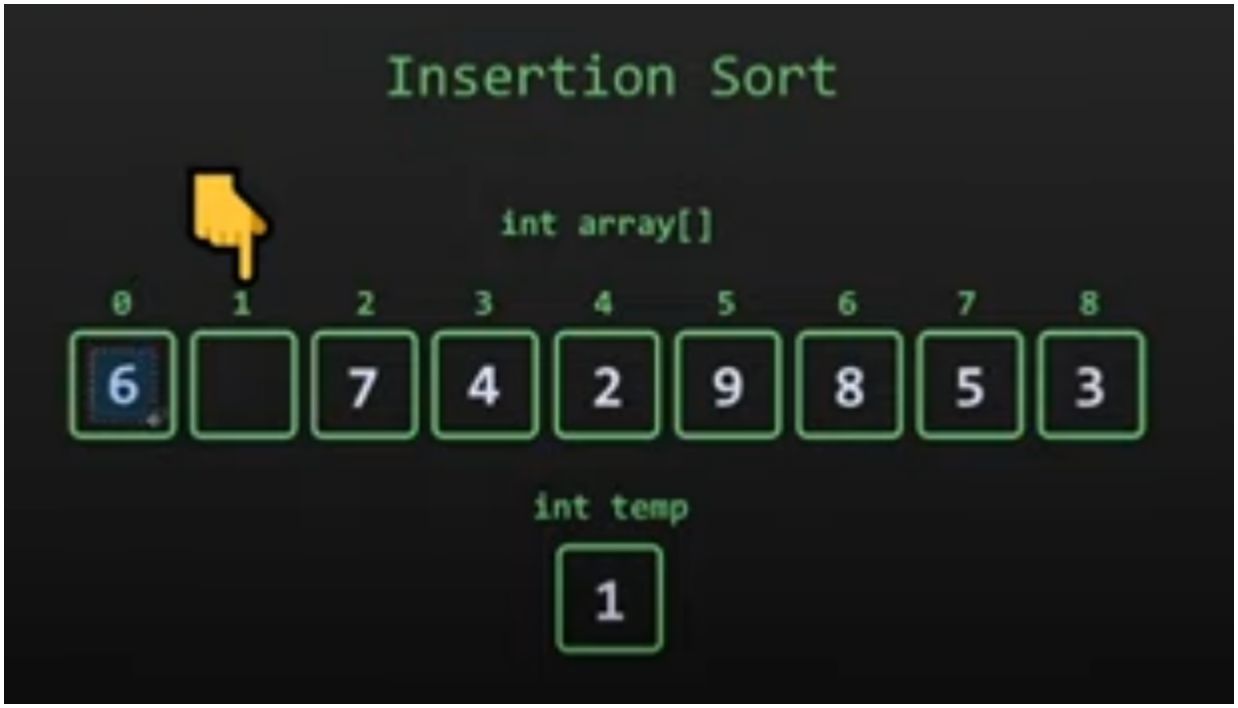


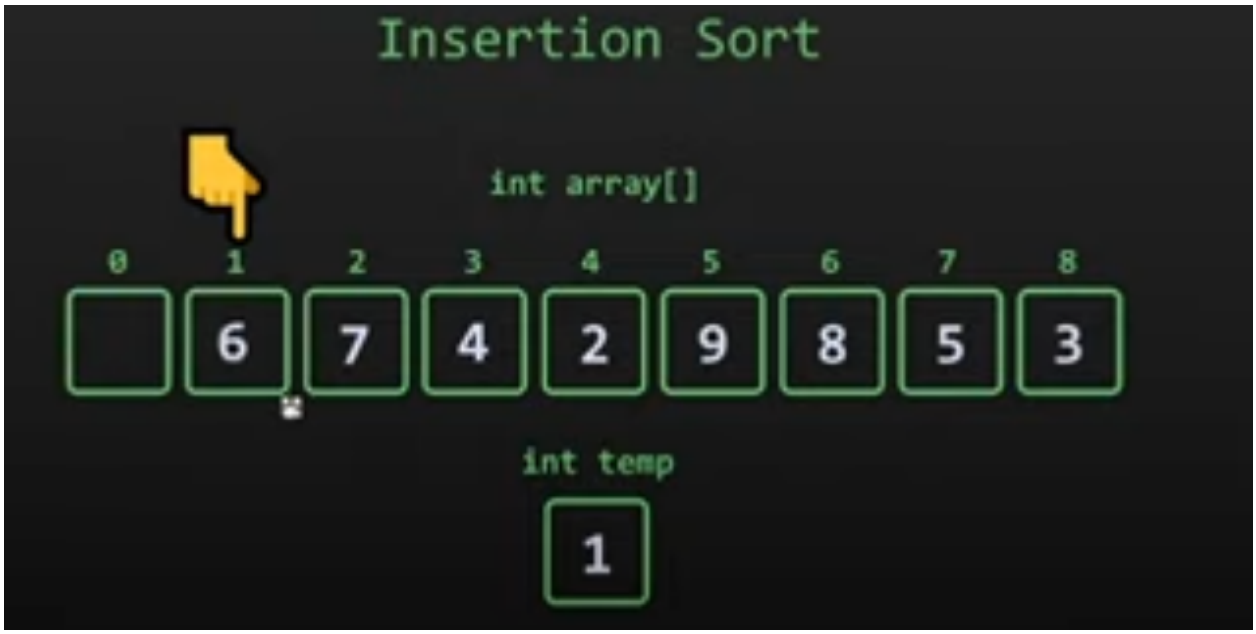
Insertion Sort

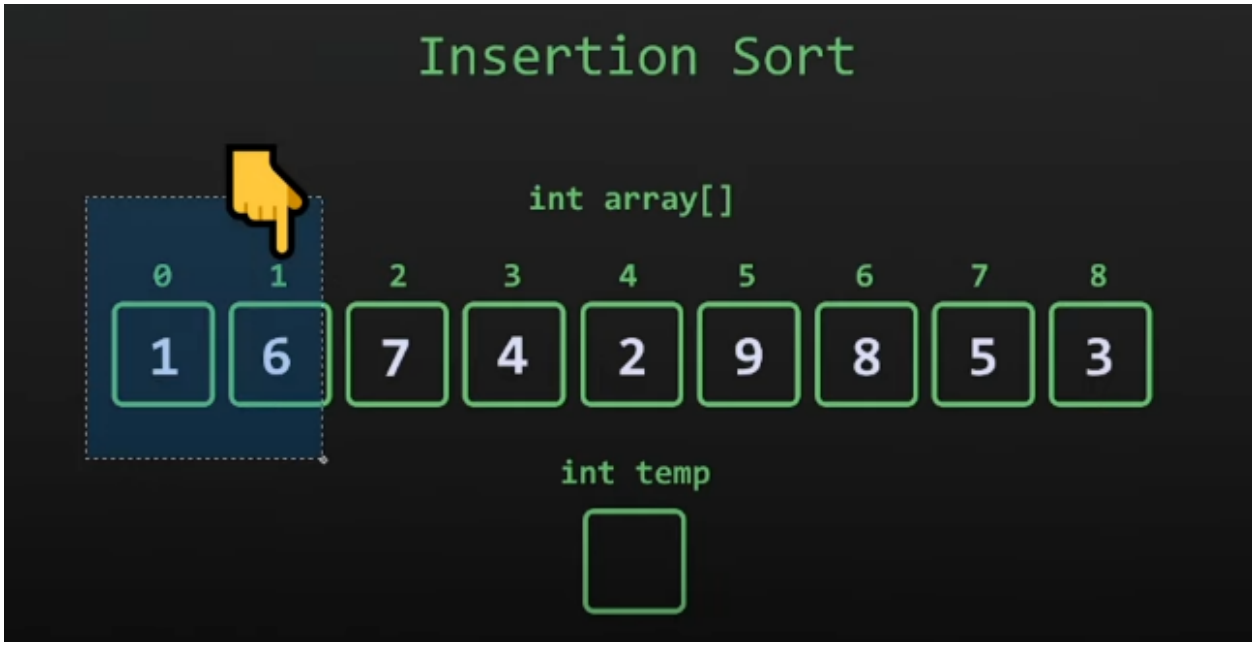
```
// Insertion sort = after comparing elements to the left,
    //          shift elements to the right to make room to insert a value

    //          Quadratic time O(n^2)
    //          small data set = decent
    //          large data set = BAD

    //          Less steps than Bubble sort
    //          Best case is O(n) compared to Selection sort O(n^2)

    public static void main(String[] args) {

        int array[] = {9, 1, 8, 2, 7, 3, 6, 5, 4};

        System.out.println("Ascending order: ");

        insertionSort(array);

        for(int i : array) {
            System.out.print(i);
        }
    }
```

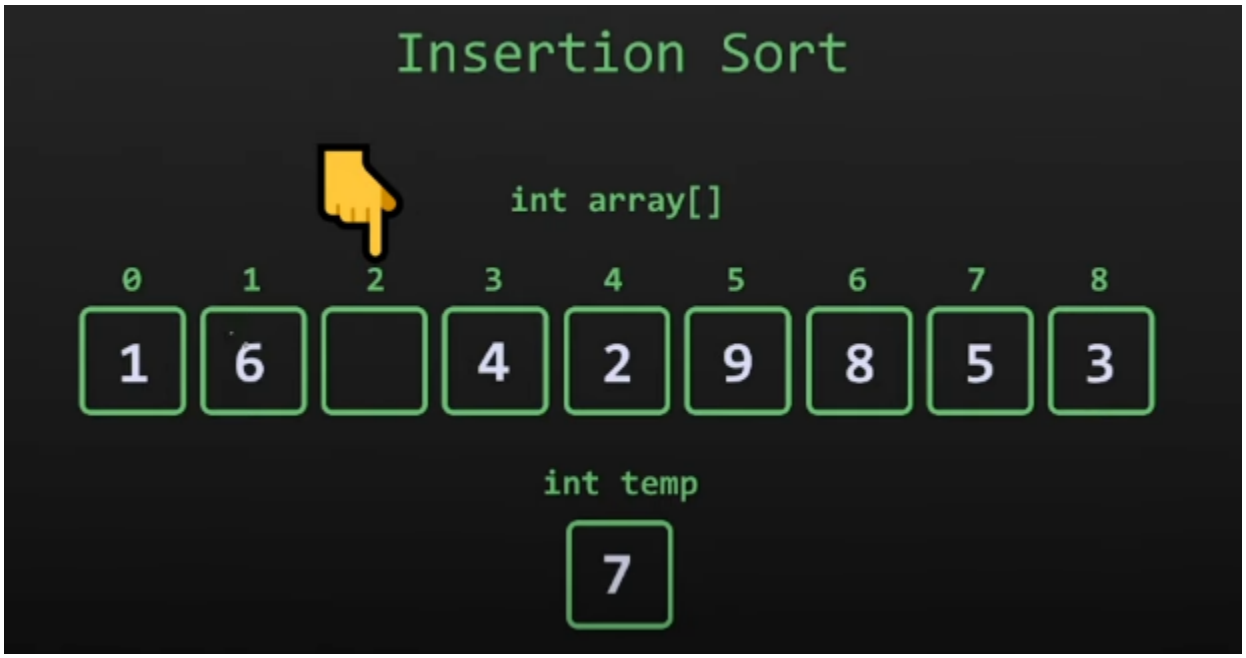
```
private static void insertionSort(int[] array) {

    for(int i = 1; i < array.length; i++) {
        int temp = array[i];
        int j = i - 1;

        while(j >= 0 && array[j] > temp) {
            array[j + 1] = array[j];
            j--;
        }
        array[j + 1] = temp;
    }
}
```

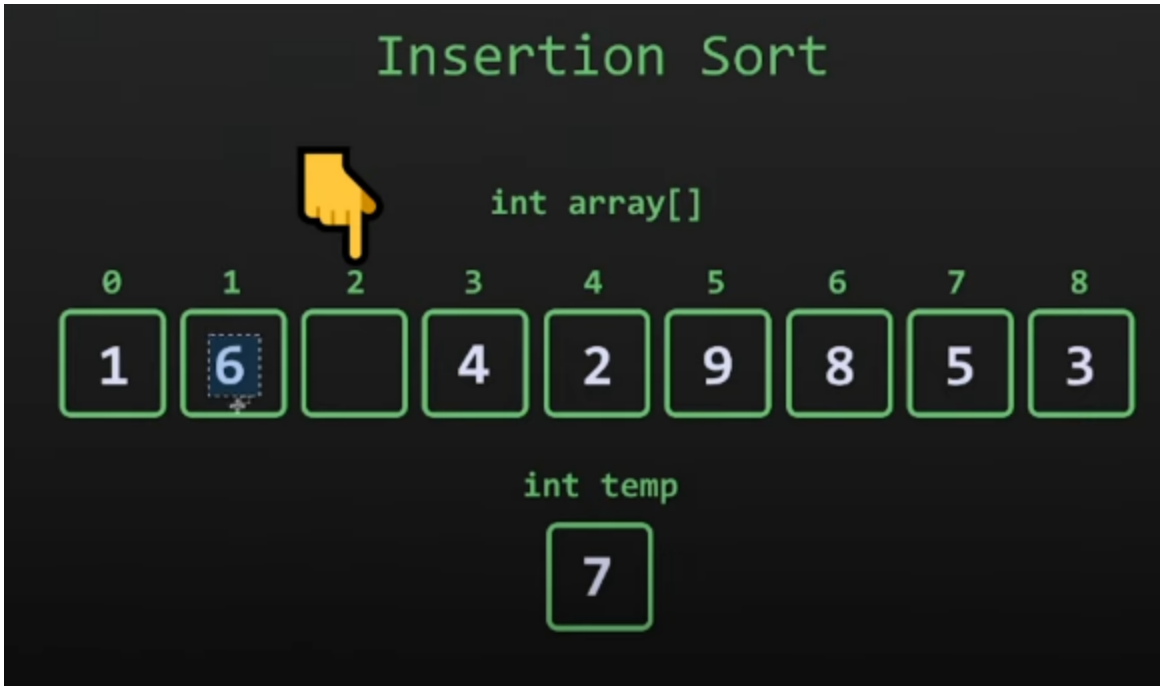


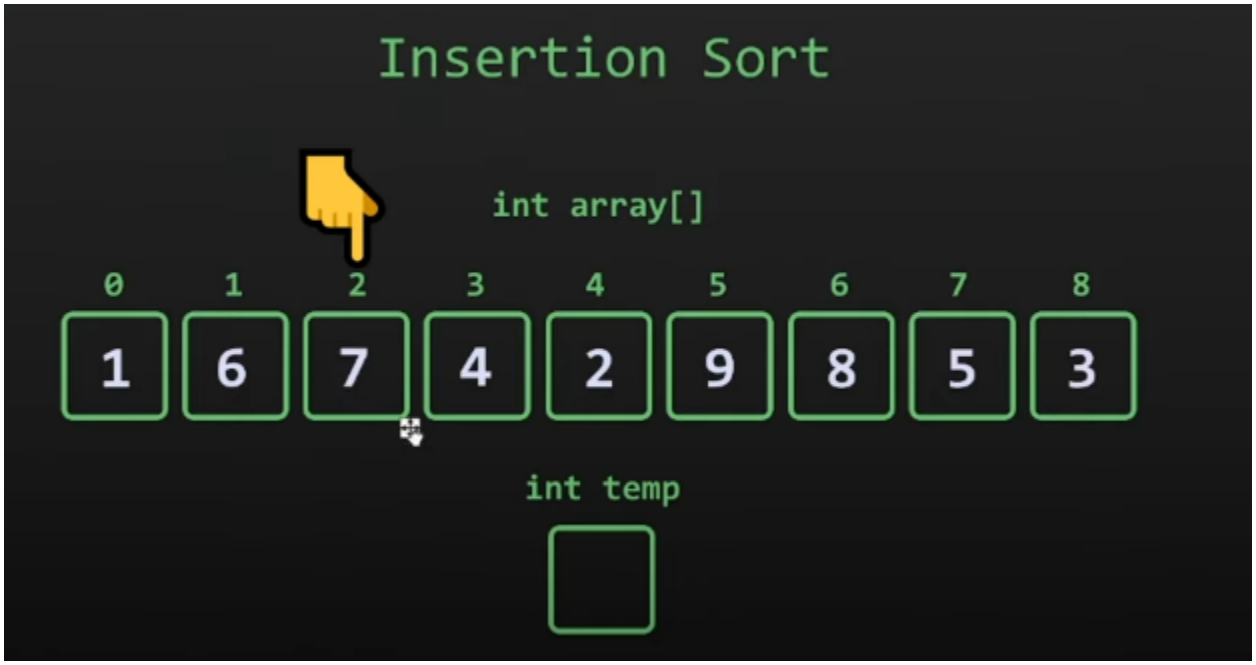

Insertion Sort
int array[]
0 1 2 3 4 5 6 7 8
6 7 4 2 9 8 5 3
int temp
1

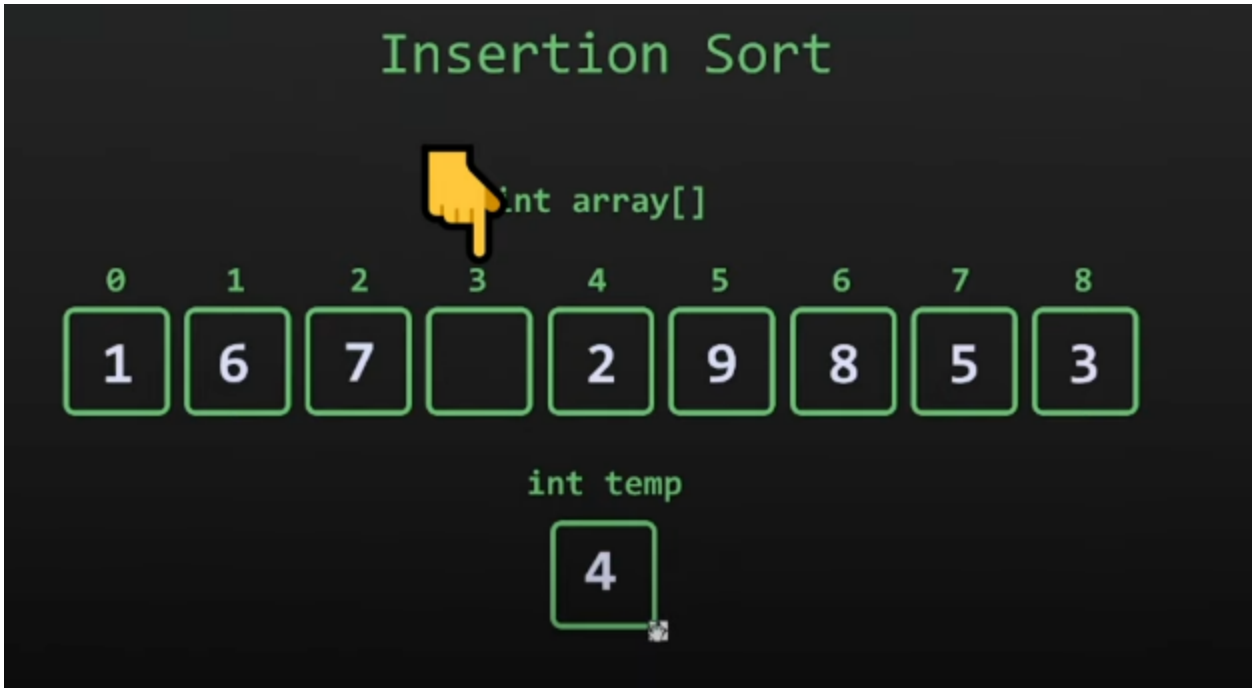

Insertion Sort
int array[]
0 1 2 3 4 5 6 7 8
6 7 4 2 9 8 5 3
int temp
1

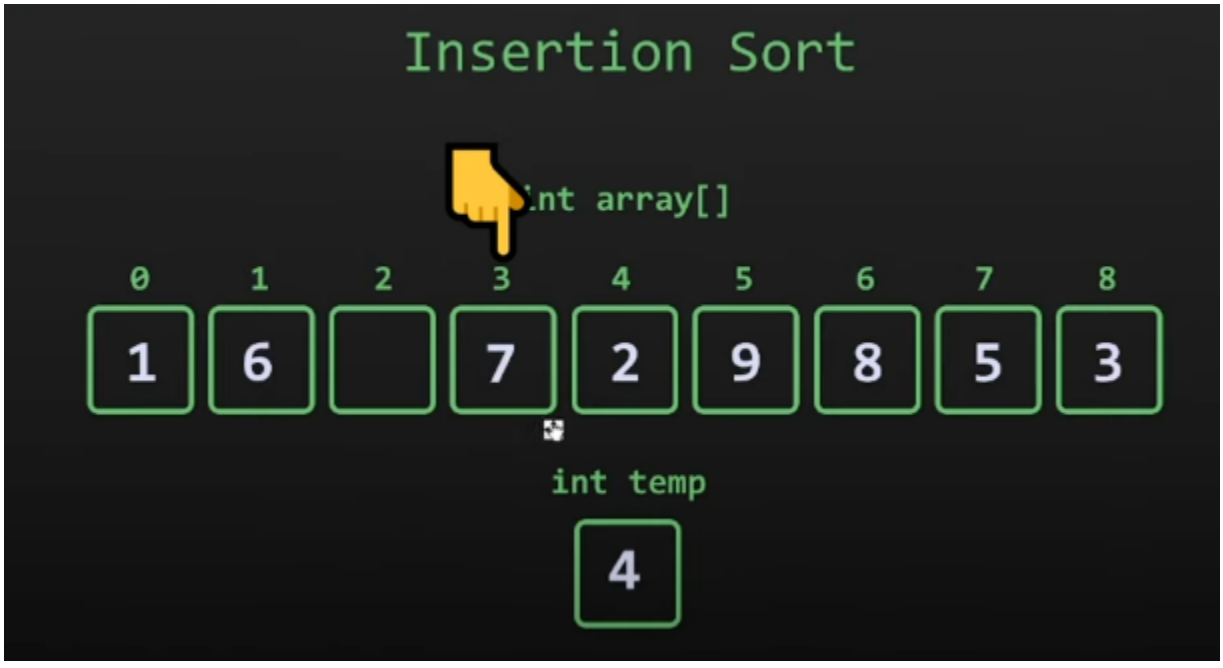

Insertion Sort
int array[]
0 1 2 3 4 5 6 7 8
1 6 7 4 2 9 8 5 3
int temp

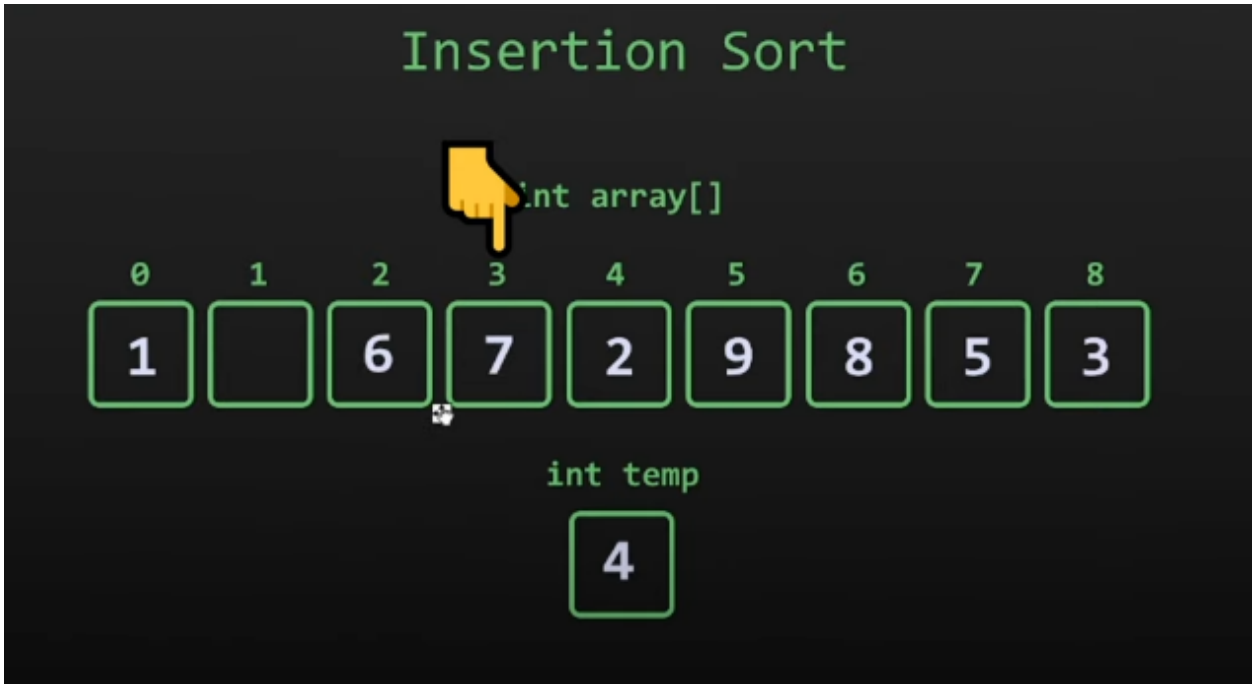

Insertion Sort
int array[]
0 1 2 3 4 5 6 7 8
1 6 4 2 9 8 5 3
int temp
7

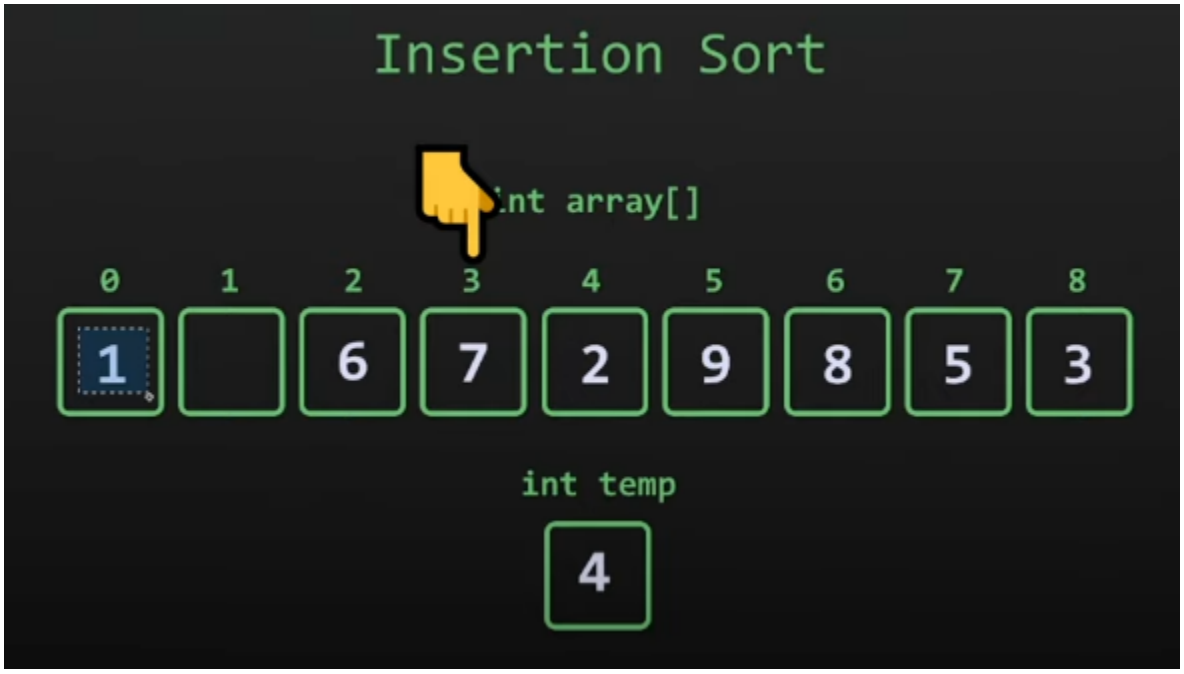

Insertion Sort
int array[]
0 1 2 3 4 5 6 7 8
1 6 4 2 9 8 5 3
int temp
7

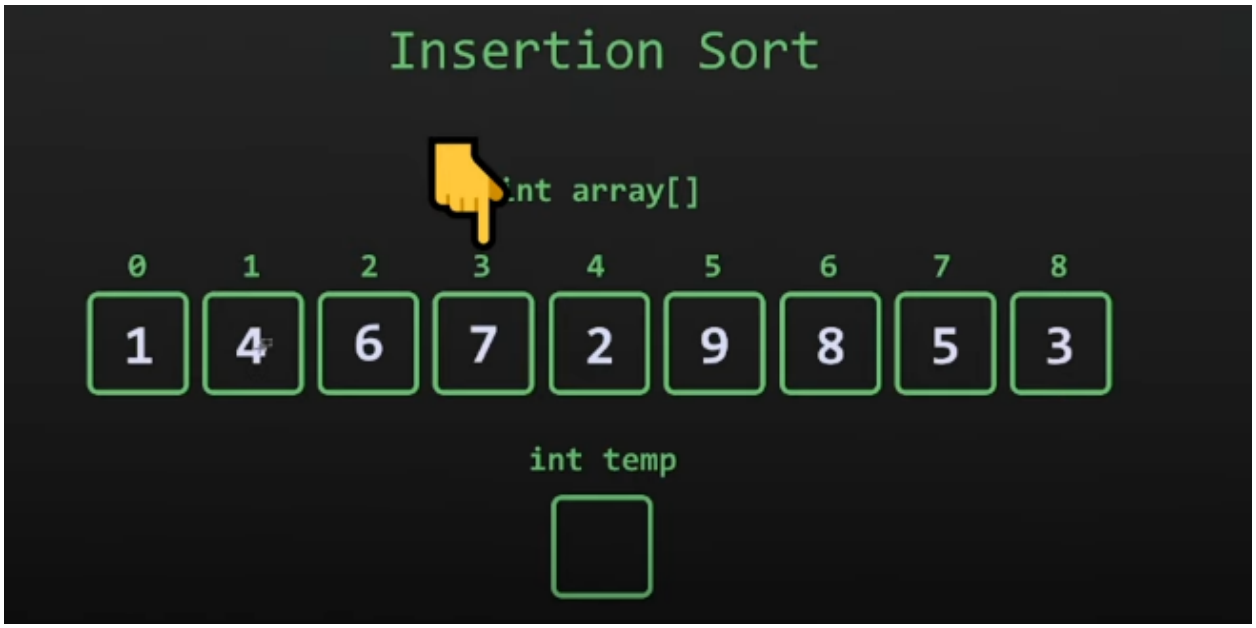

Insertion Sort
int array[]
0 1 2 3 4 5 6 7 8
1 6 7 4 2 9 8 5 3
int temp

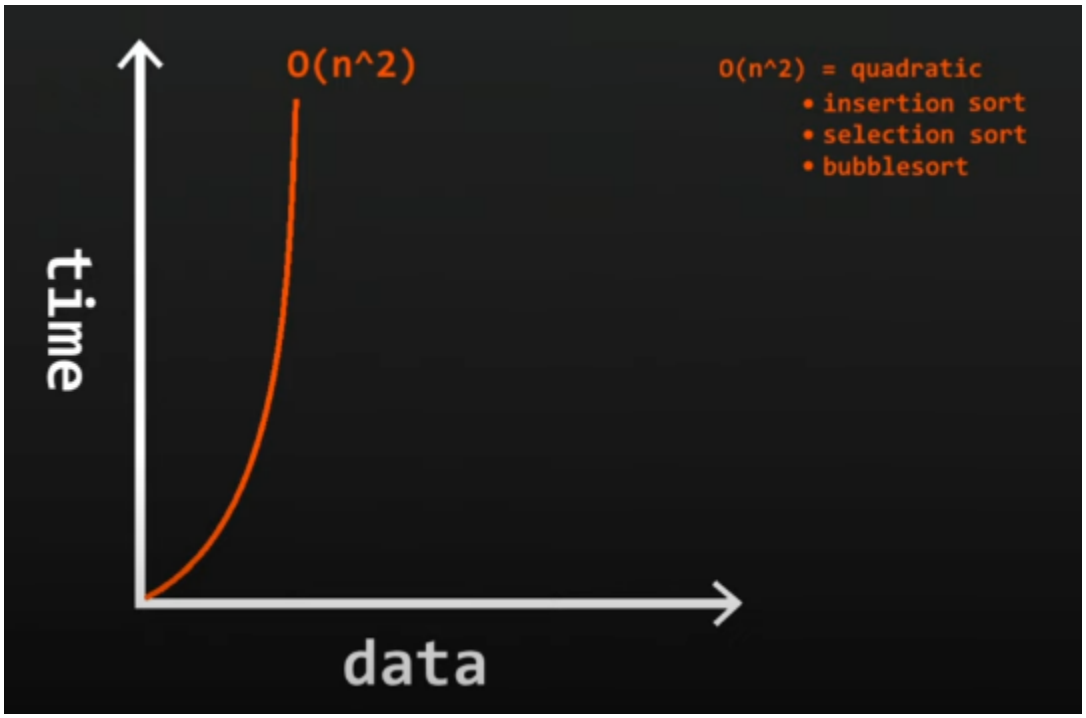

Insertion Sort
int array[]
0 1 2 3 4 5 6 7 8
1 6 7 2 9 8 5 3
int temp
4


Insertion Sort
int array[]
0 1 2 3 4 5 6 7 8
1 6 7 2 9 8 5 3
int temp
4

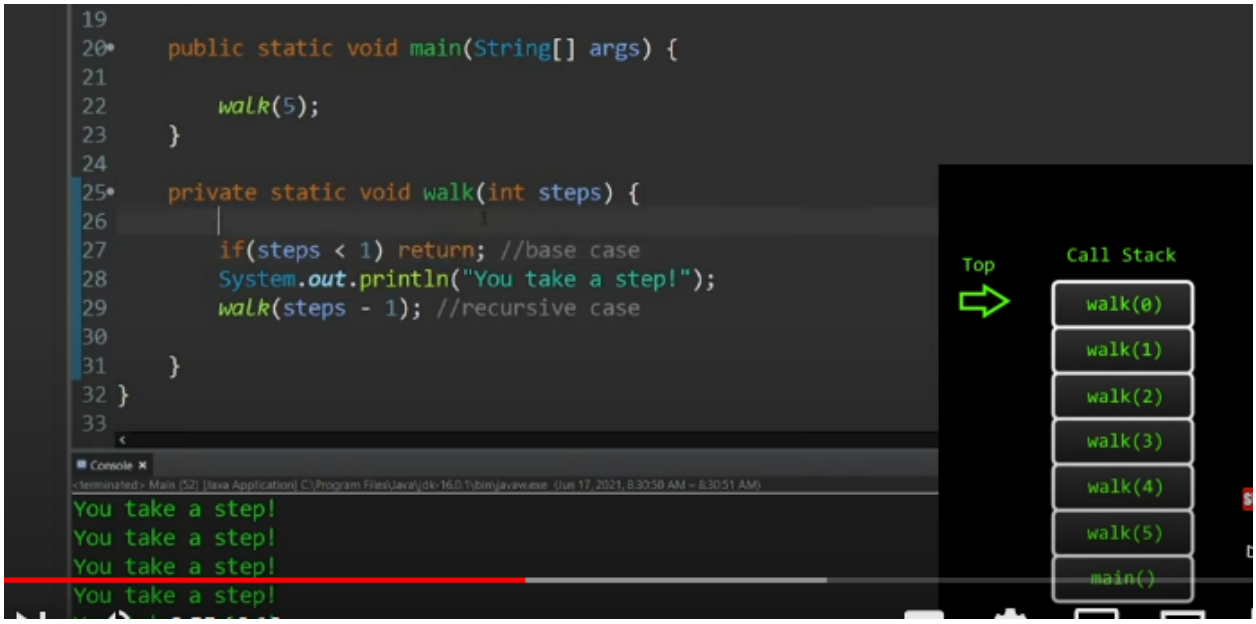

Insertion Sort
int array[]
0 1 2 3 4 5 6 7 8
1 6 7 2 9 8 5 3
int temp
4


Insertion Sort
int array[]
0 1 2 3 4 5 6 7 8
1 6 7 2 9 8 5 3
int temp
4


Insertion Sort
int array[]
0 1 2 3 4 5 6 7 8
1 4 6 7 2 9 8 5 3
int temp


O(n^2)
O(n^2) = quadratic
• insertion sort
• selection sort
• bubblesort
time
data

Recursion



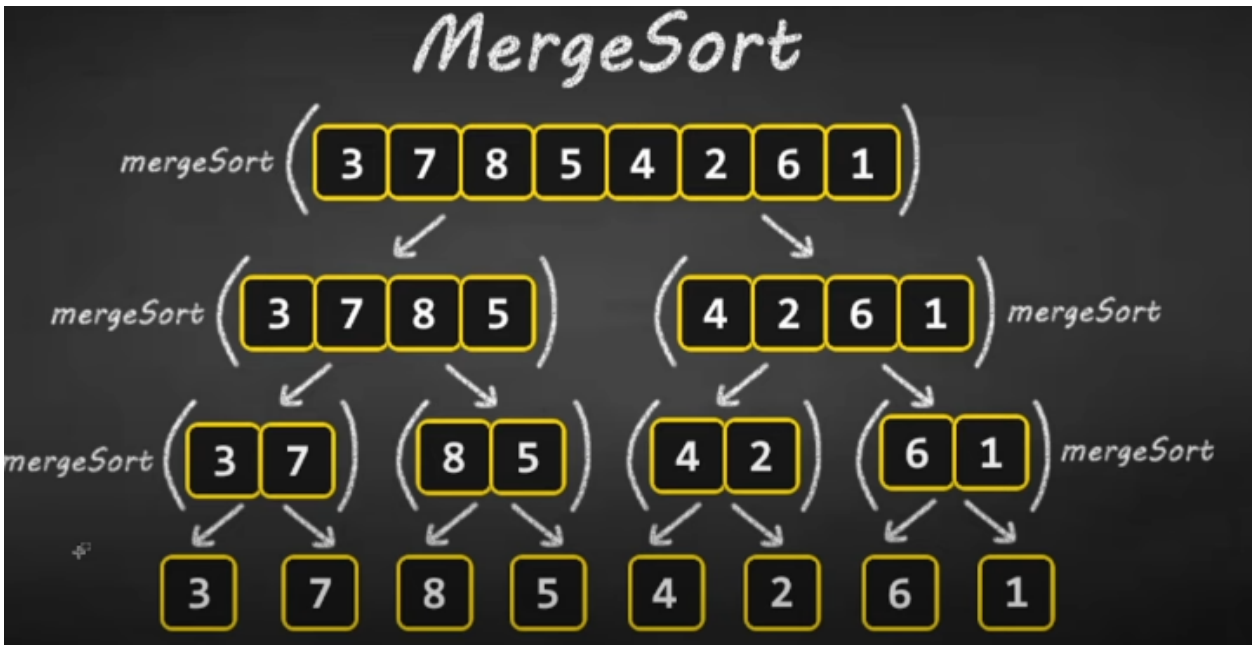
class Main{

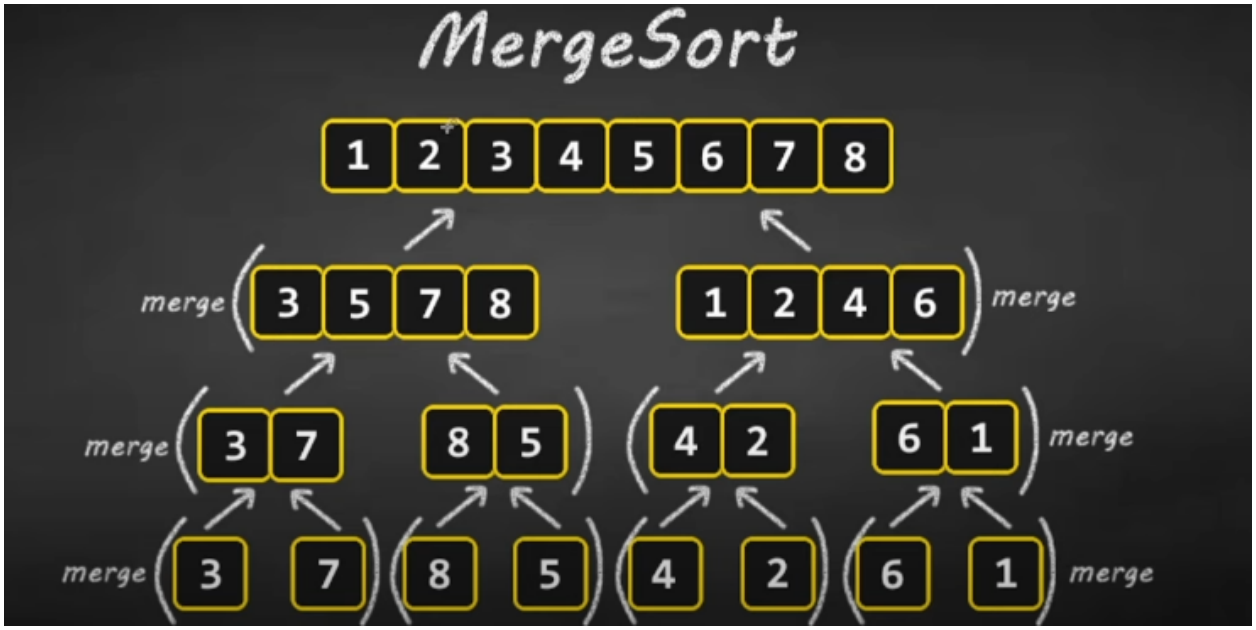
recursion = When

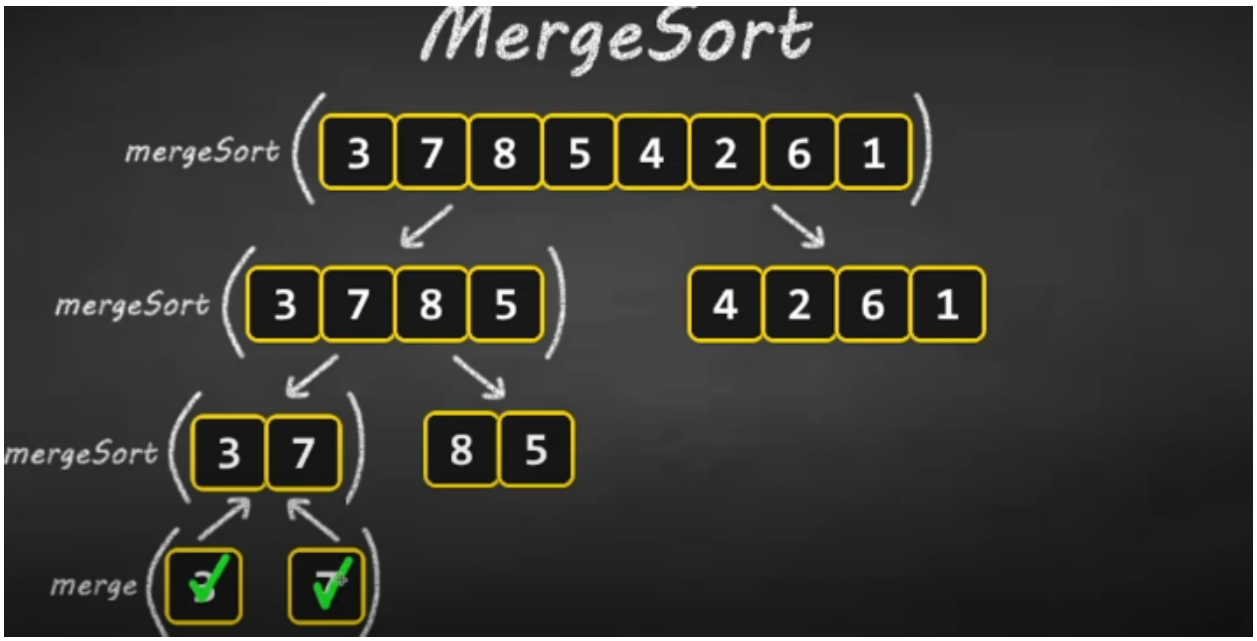
	iteration	recursion
implemention	uses loops	calls itself
state	control index (int steps = 0)	parameter values walk(int steps)
progression	moves toward value in condition	converge towards base case
termination	index satisfies condition	base case = true
size	more code less memory	less code more memory
speed	faster	slower

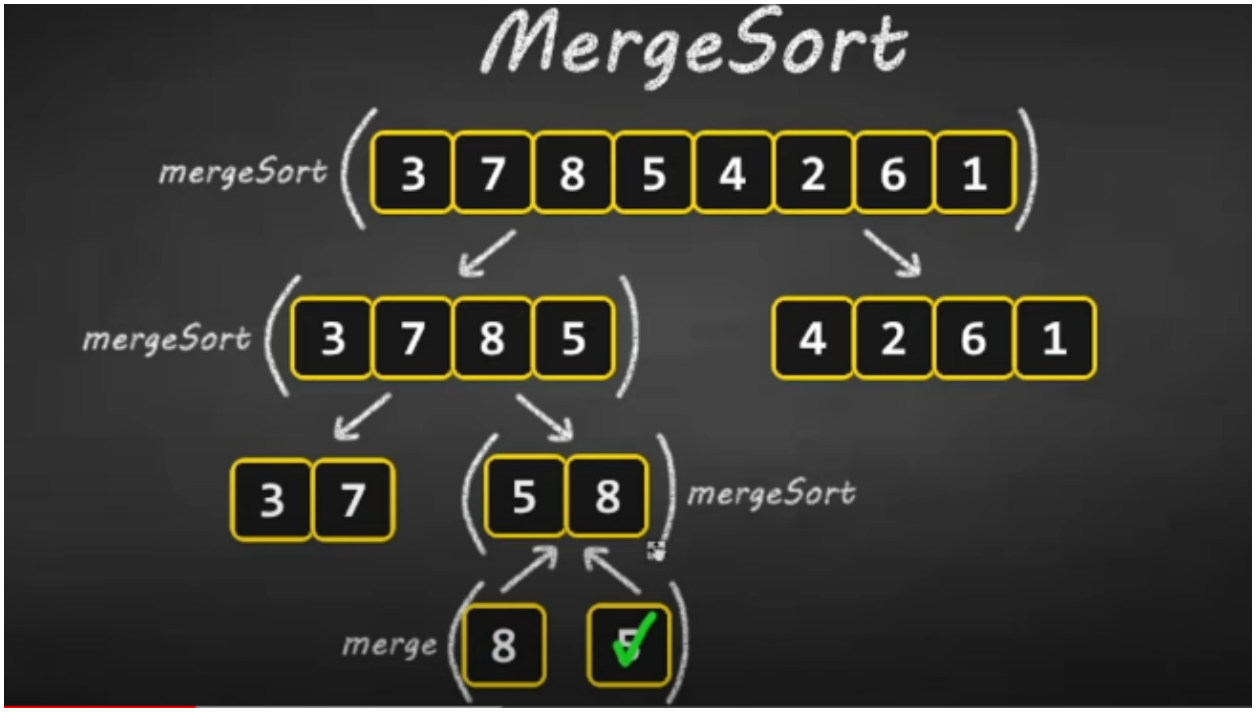
```
// recursion = When a thing is defined in terms of itself. - Wikipedia
//             Apply the result of a procedure, to a procedure.
//             A recursive method calls itself. Can be a substitute for iteration.
//             Divide a problem into sub-problems of the same type as the original.
//             Commonly used with advanced sorting algorithms and navigating trees

//             Advantages
//             ----------
//             easier to read/write
//             easier to debug

//             Disadvantages
//             ----------
//             sometimes slower
//             uses more memory
```

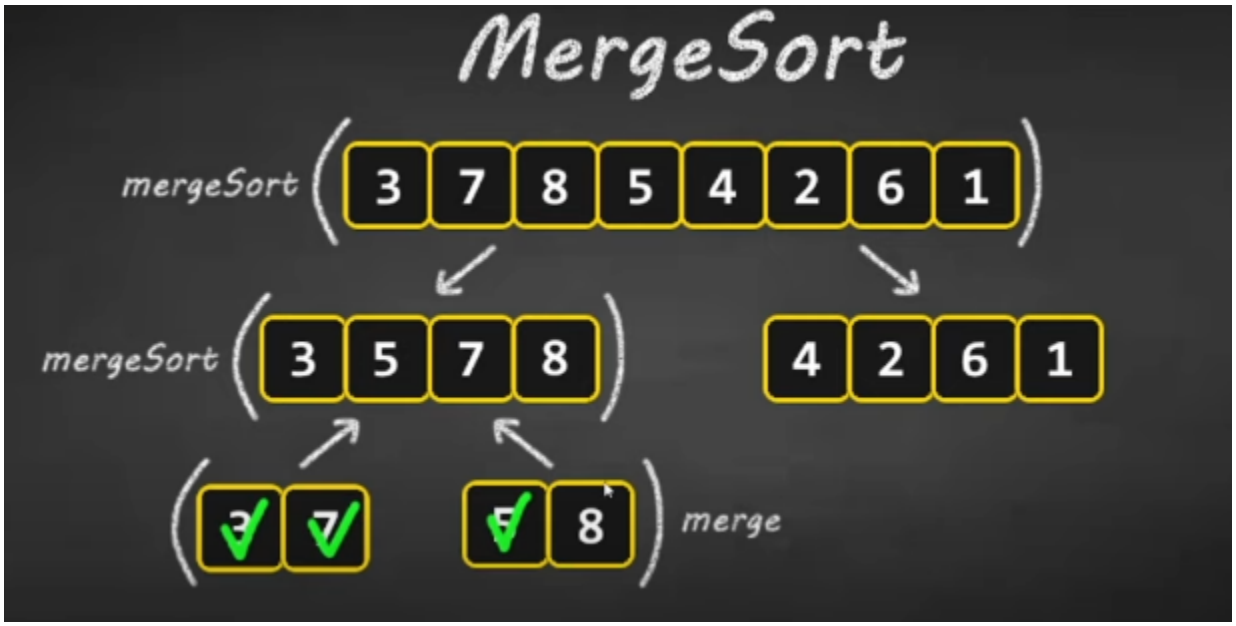
Merge Sort

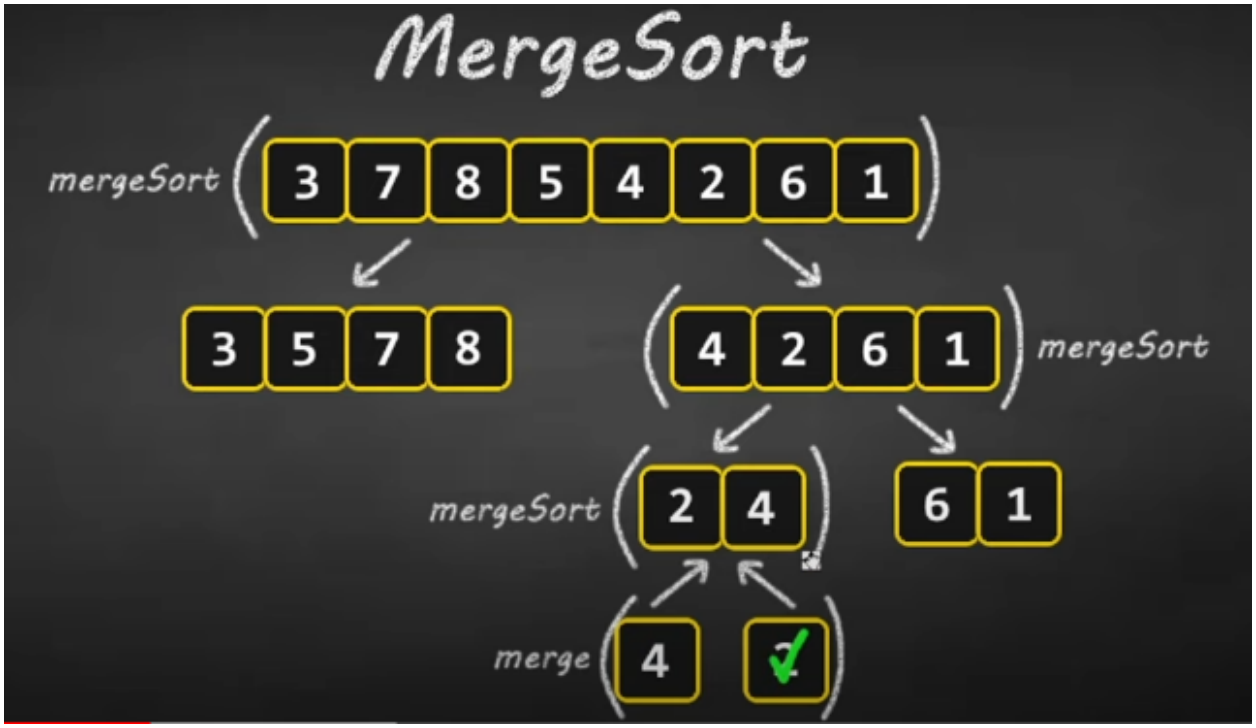


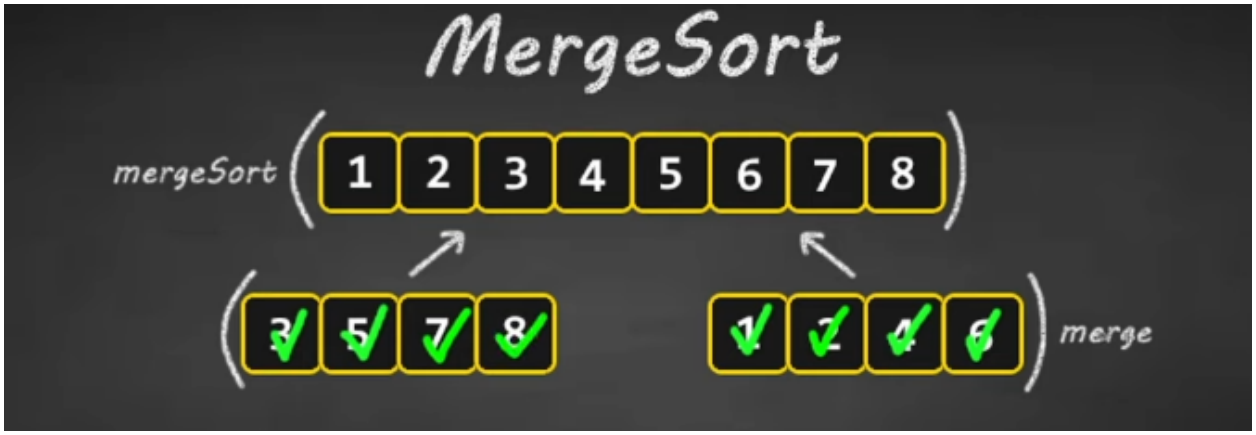

MergeSort
1 2 3 4 5 6 7 8
merge
3 5 7 8
1 2 4 6
merge
merge
3 7
8 5
4 2
6 1
merge
merge
3
7
8
5
4
2
6
1
merge

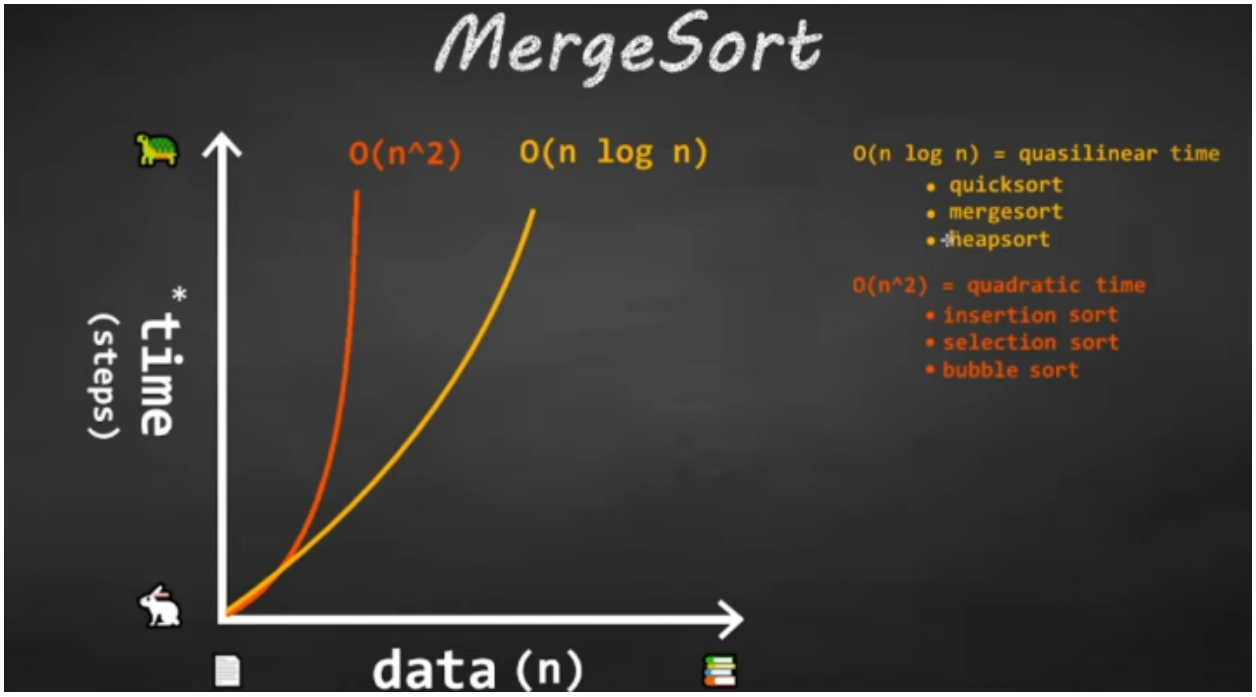

MergeSort
mergeSort
3 7 8 5 4 2 6 1
mergeSort
3 7 8 5
4 2 6 1
mergeSort
3 7
8 5
merge

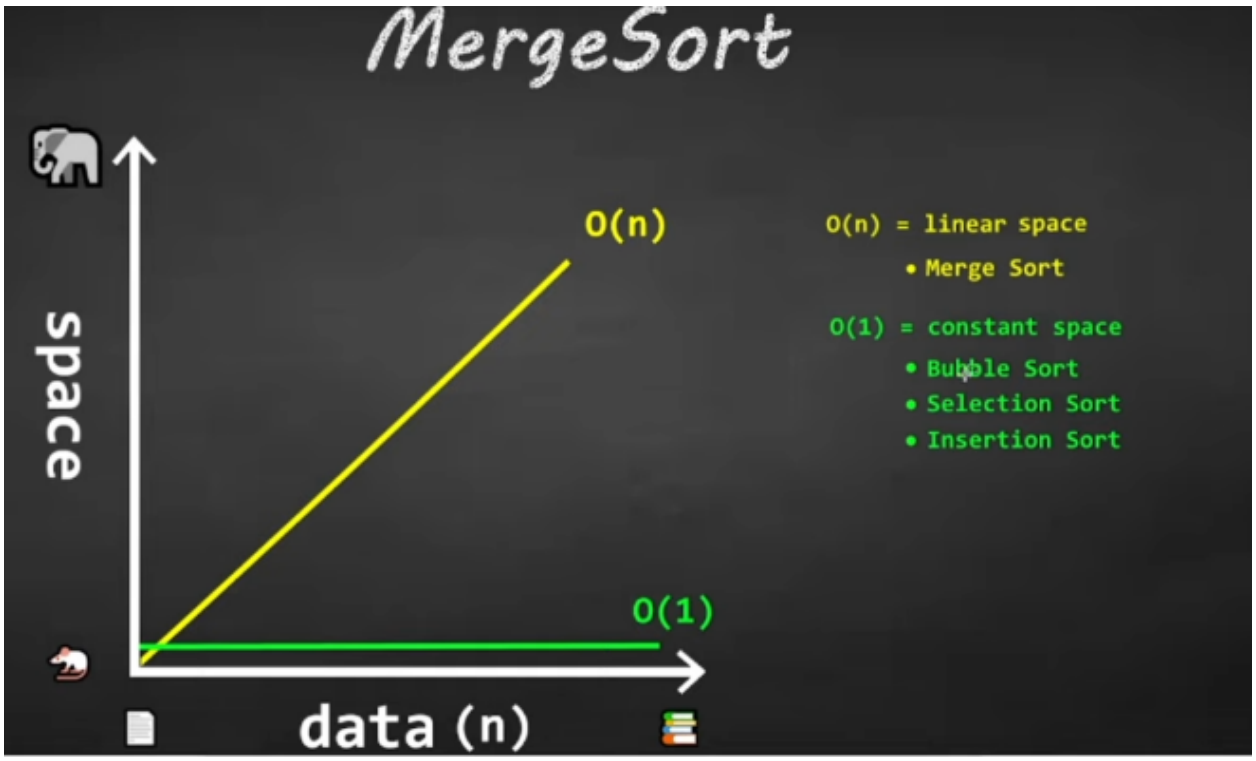

MergeSort
mergeSort
3 7 8 5 4 2 6 1
mergeSort
3 7 8 5
4 2 6 1
3 7
5 8
mergeSort
merge
8


MergeSort
mergeSort
3 7 8 5 4 2 6 1
mergeSort
3 5 7 8
4 2 6 1
8
merge


MergeSort
mergeSort
3 7 8 5 4 2 6 1
3 5 7 8
4 2 6 1
mergeSort
mergeSort
2 4
6 1
merge
4


MergeSort
mergeSort
1 2 3 4 5 6 7 8
merge


MergeSort
O(n^2)
O(n log n)
time (steps)
data (n)
O(n log n) = quasilinear time
• quicksort
• mergesort
• heapsort
O(n^2) = quadratic time
• insertion sort
• selection sort
• bubble sort


MergeSort
O(n)
O(1)
space
data (n)
O(n) = linear space
• Merge Sort
O(1) = constant space
• Bubble Sort
• Selection Sort
• Insertion Sort

Merge Sort

Algorithm 1 Merge-Sort(A, p, r)

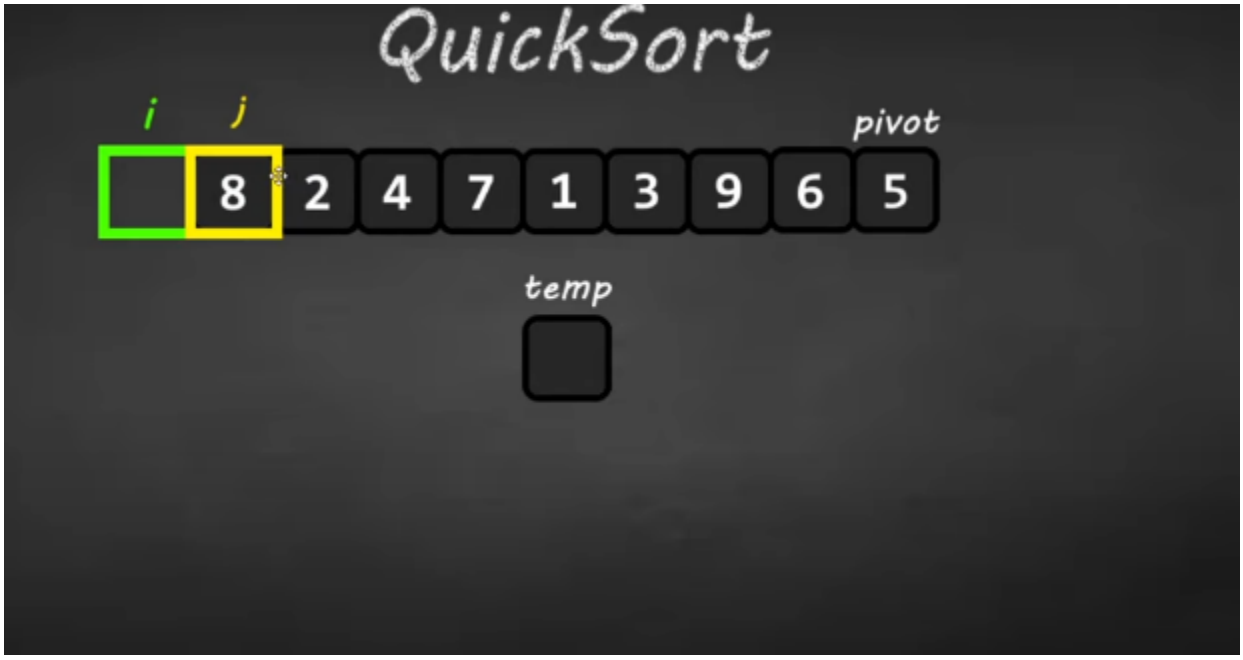
```
1: if p < r then
2:     q ← ⌊(p + r)\2⌋
3:     Merge-Sort(A, p, q)
4:     Merge-Sort(A, q + 1, r)
5:     Merge(A, p, q, r)
6: end if
```

Merge Sort

Algorithm 2 Merge(A, p, q, r)

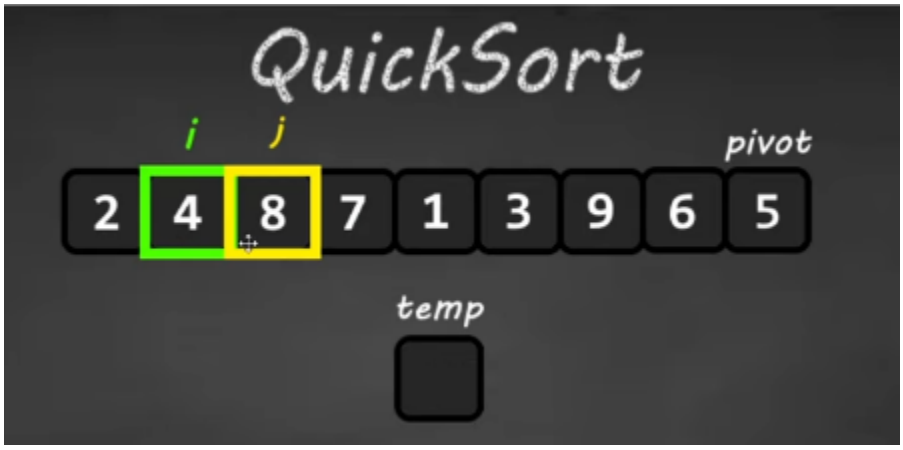
```
1: n1 ← q − p + 1
2: n2 ← r − q
3: let L[1 . . . n1 + 1] and R[1 . . . n2 + 1] be new arrays
4: for i ← 1, n1 do
5:     L[i] ← A[p + i − 1]
6: end for
7: for j ← 1, n2 do
8:     R[j] ← A[q + j]
9: end for
10: L[n1 + 1] ← ∞
11: R[n2 + 1] ← ∞
12: i ← 1
13: j ← 1
14: for k ← p, r do
15:     if L[i] ≤ R[j] then
16:         A[k] ← L[i]
17:         i ← i + 1
18:     else
19:         A[k] ← R[j]
20:         j ← j + 1
21:     end if
22: end for
```

Quick Sort(divide and conquer)

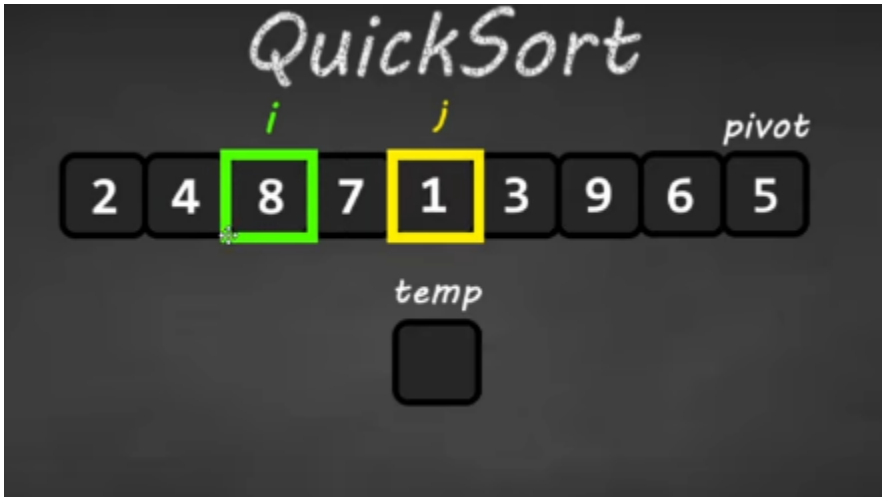


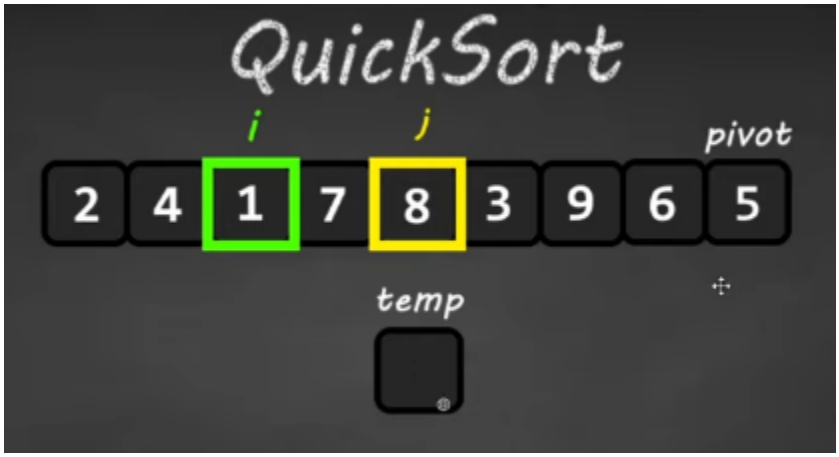

QuickSort
i
j
pivot
2
8
4
7
1
3
9
6
5
temp

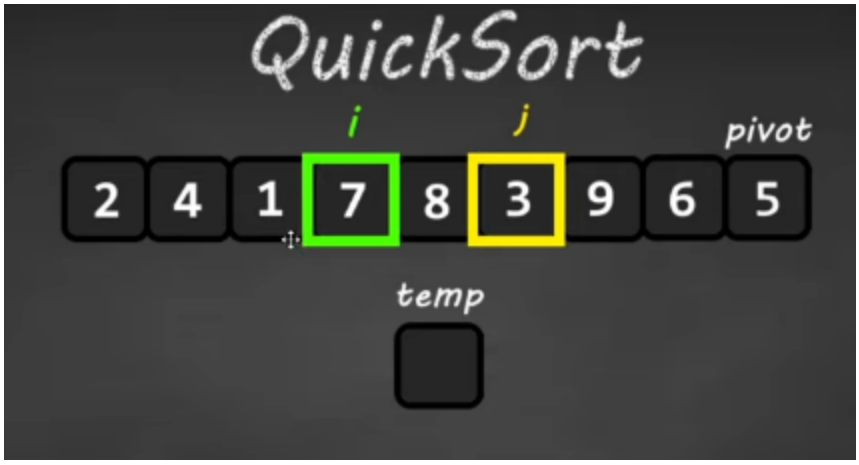

QuickSort
i
j
pivot
2
8
4
7
1
3
9
6
5
temp

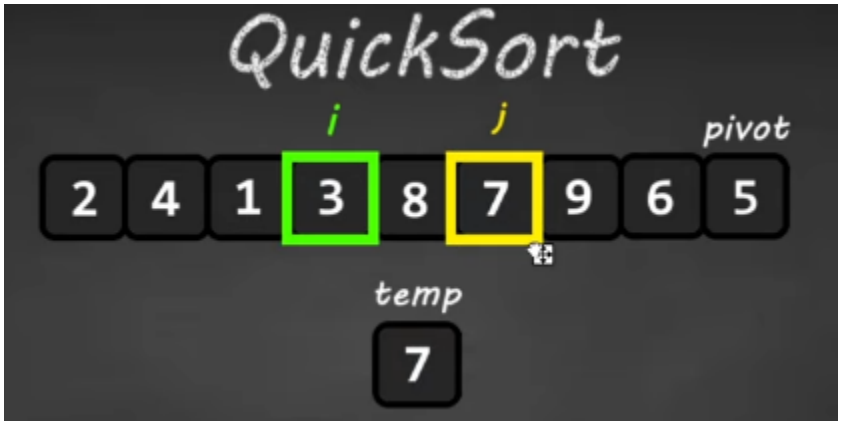

QuickSort
i
j
pivot
2
4
8
7
1
3
9
6
5
temp


QuickSort
i
j
pivot
2 4 8 7 1 3 9 6 5
temp

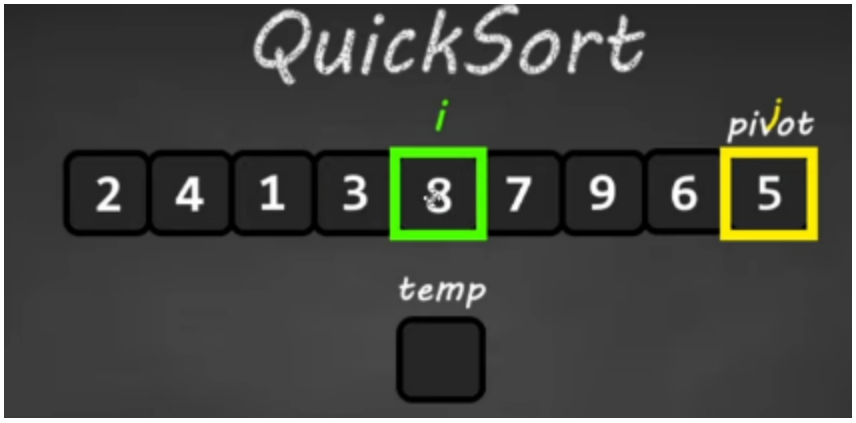

QuickSort
i
j
pivot
2 4 8 7 1 3 9 6 5
temp


QuickSort
i
j
pivot
2 4 1 7 8 3 9 6 5
temp


QuickSort
i
j
pivot
2 4 1 7 8 3 9 6 5
temp

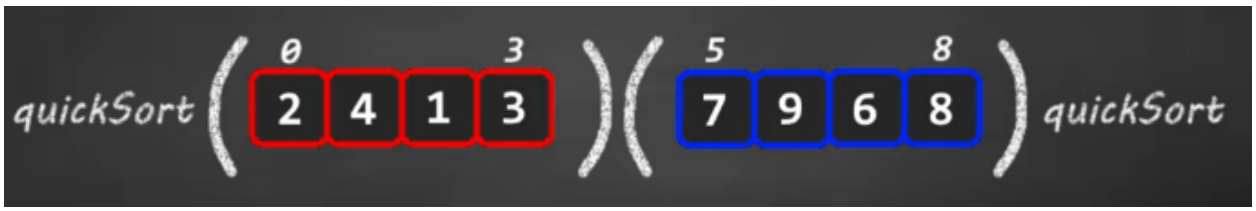

QuickSort
i
j
pivot
2 4 1 3 8 7 9 6 5
temp
7

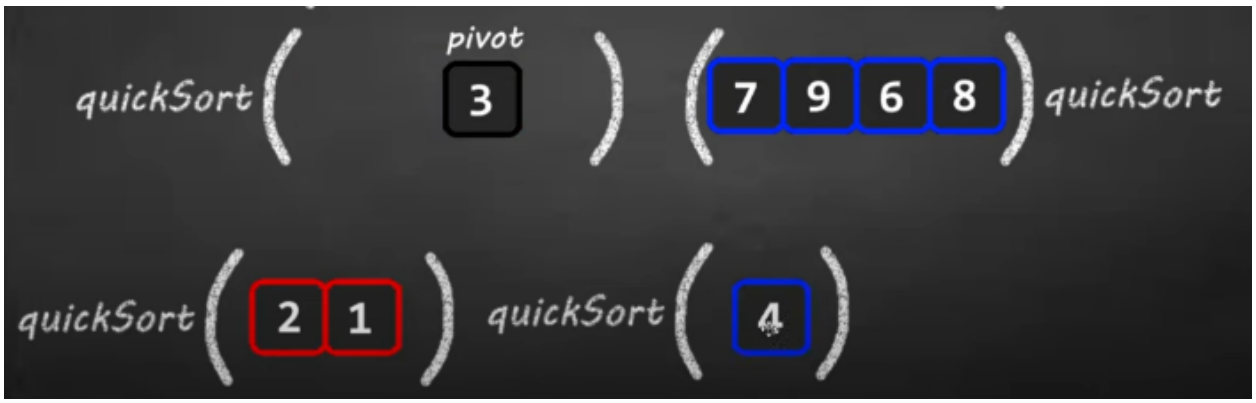

QuickSort
i
j
pivot
2 4 1 3 8 7 9 6 5
temp

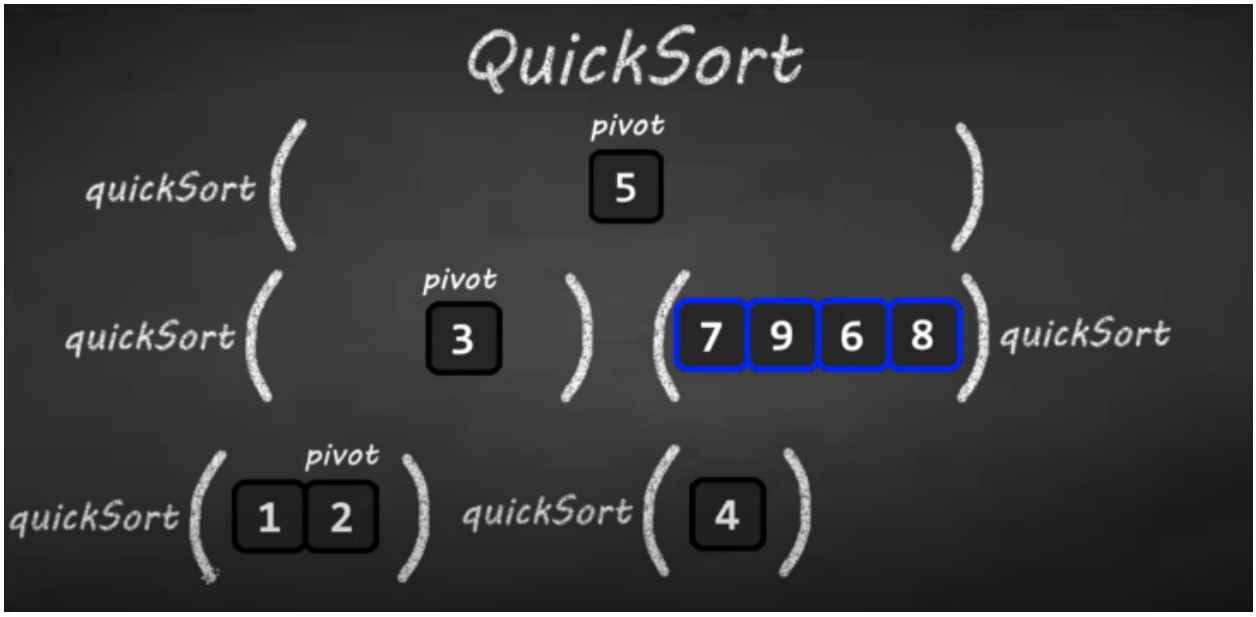

QuickSort
i
pivot
2 4 1 3 8 7 9 6 5
temp

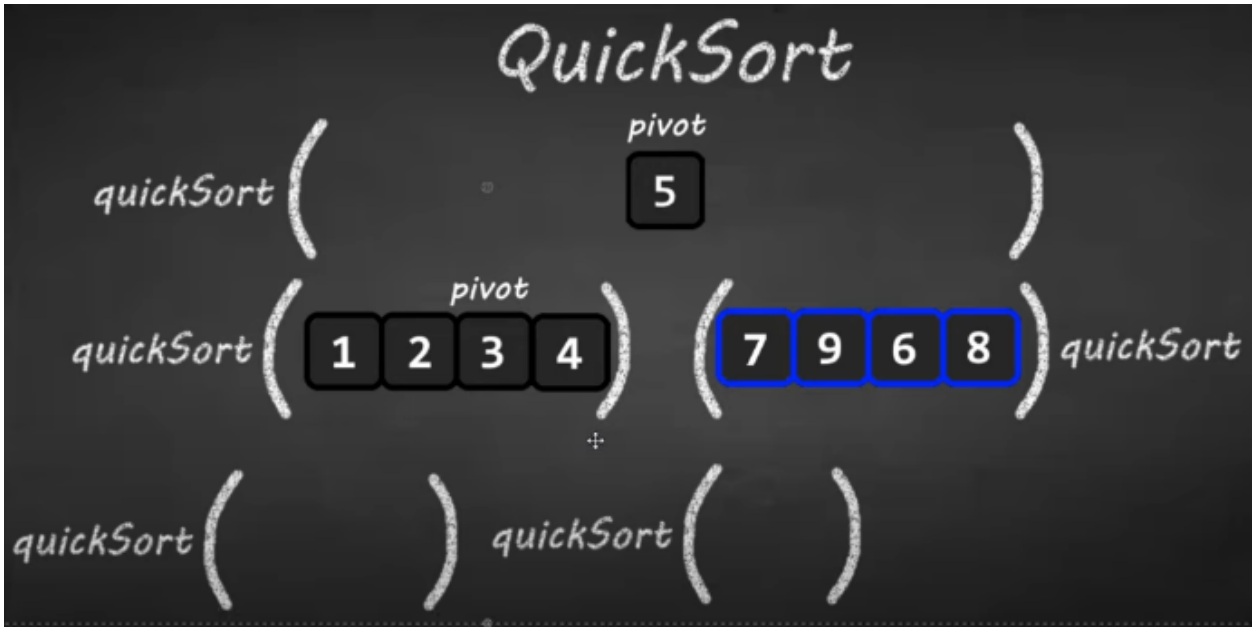

QuickSort
pivot
2 4 1 3 5 7 9 6 8

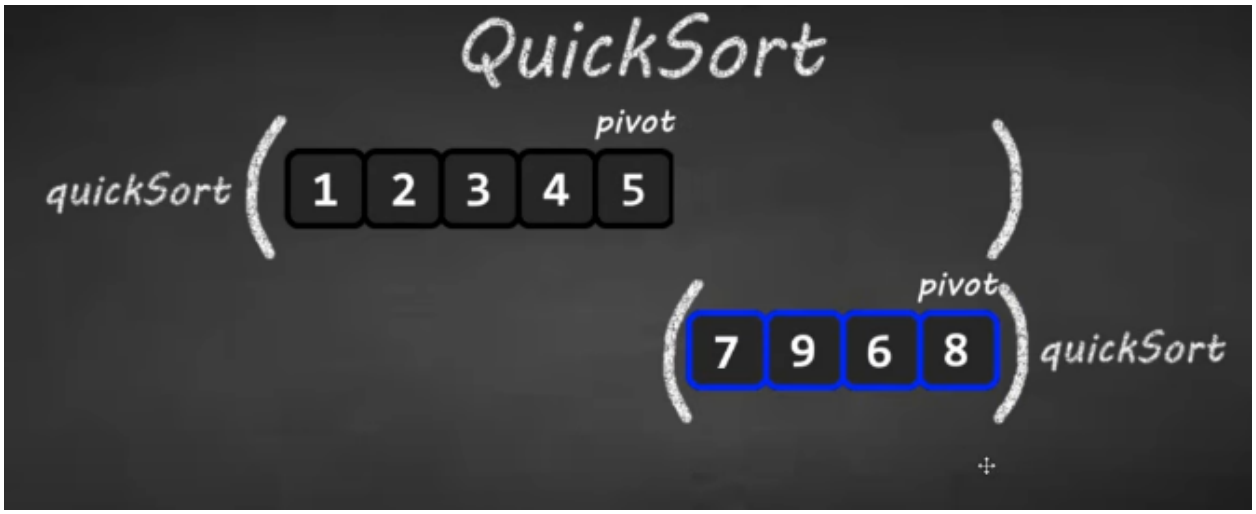

QuickSort
pivot
2 4 1 3 5 7 9 6 8

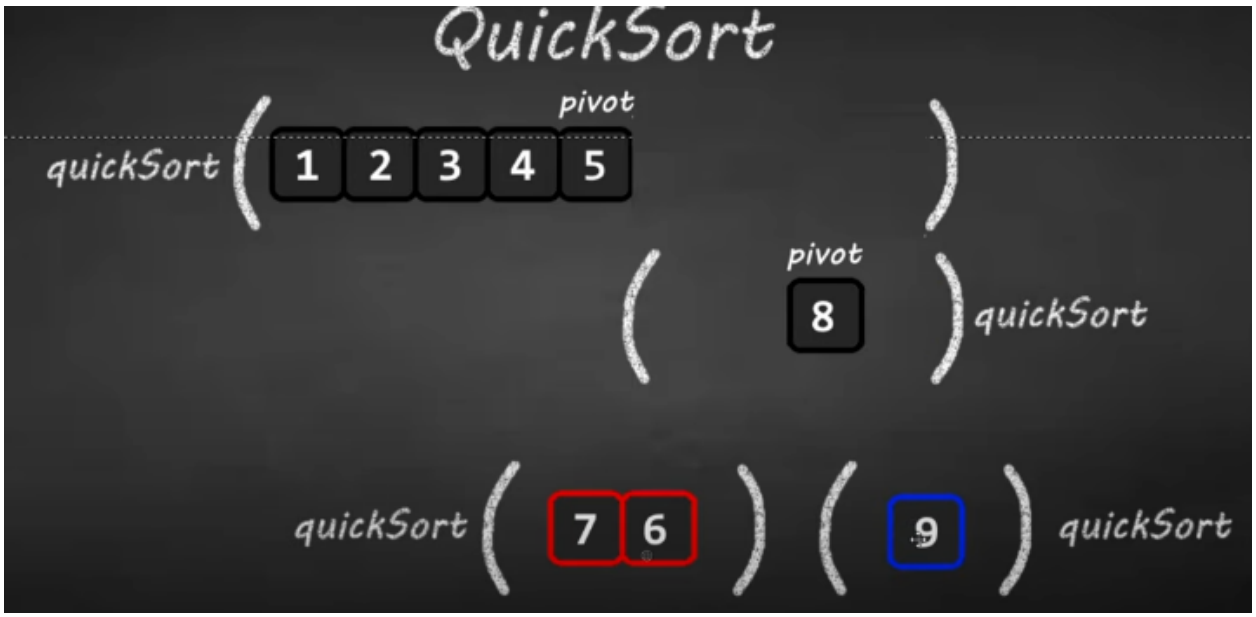

quickSort
0
3
2 4 1 3
5
8
7 9 6 8
quickSort

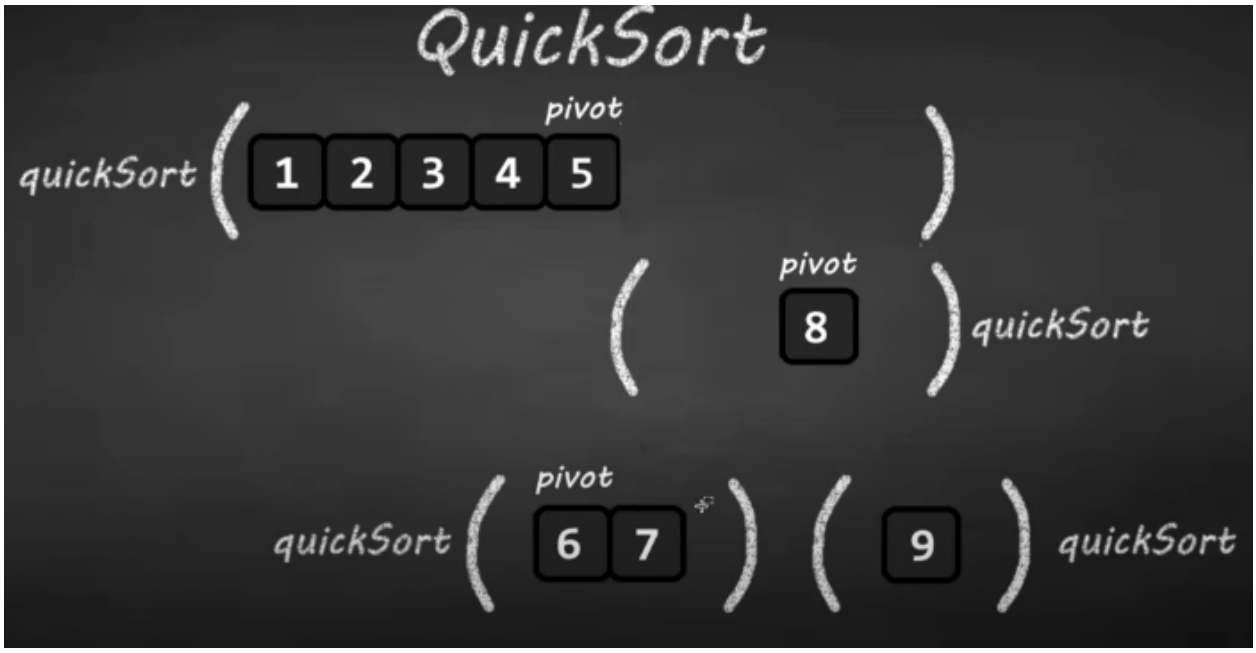

quickSort
pivot
3
7 9 6 8
quickSort
quickSort
2 1
quickSort
4

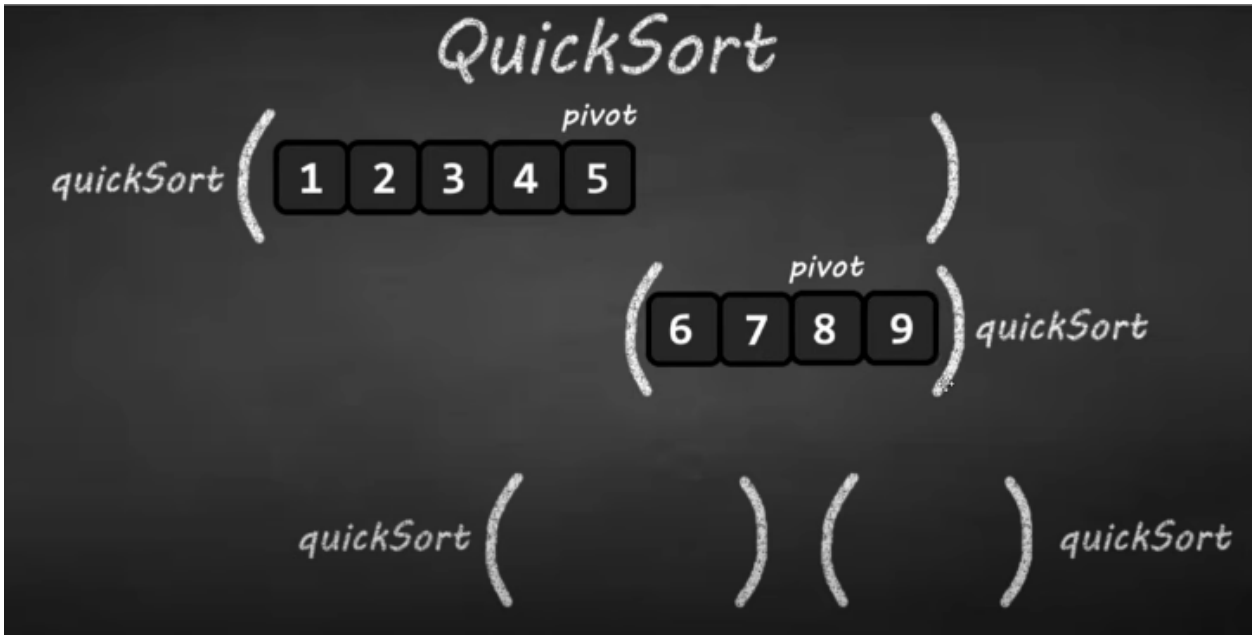

QuickSort
pivot
quickSort
5
pivot
quickSort
3
7 9 6 8
quickSort
pivot
quickSort
1 2
quickSort
4


QuickSort
pivot
quickSort
5
pivot
quickSort
1 2 3 4
7 9 6 8
quickSort
quickSort
quickSort


QuickSort
pivot
quickSort
1 2 3 4 5
pivot
7 9 6 8
quickSort


QuickSort
pivot
quickSort
1 2 3 4 5
pivot
8
quickSort
quickSort
7 6
9
quickSort


QuickSort
pivot
quickSort
1 2 3 4 5
pivot
8
quickSort
pivot
quickSort
6 7
9
quickSort

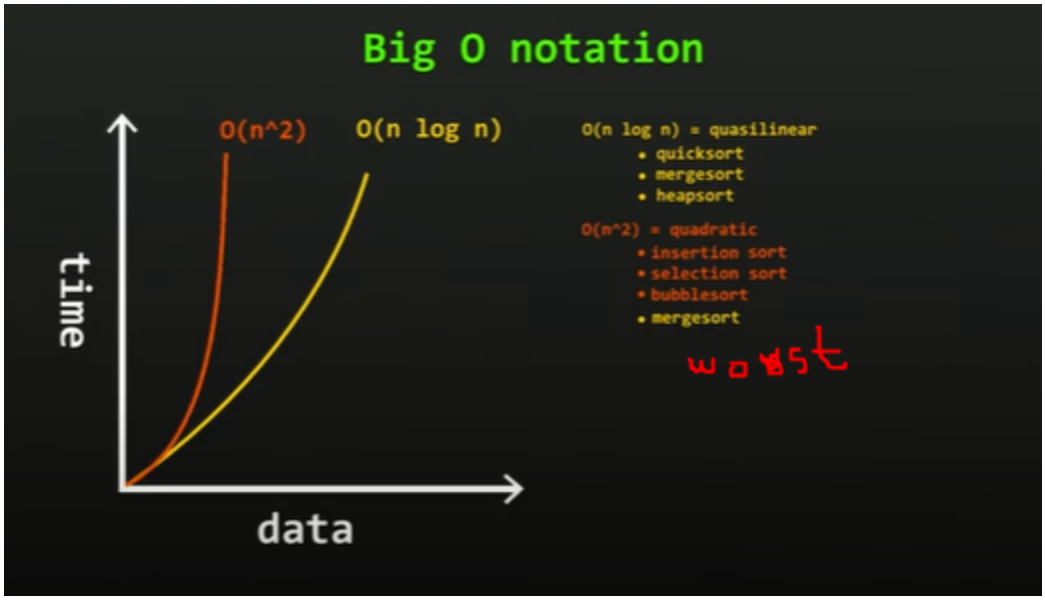

QuickSort
pivot
quickSort
1 2 3 4 5
pivot
6 7 8 9
quickSort
quickSort
quickSort

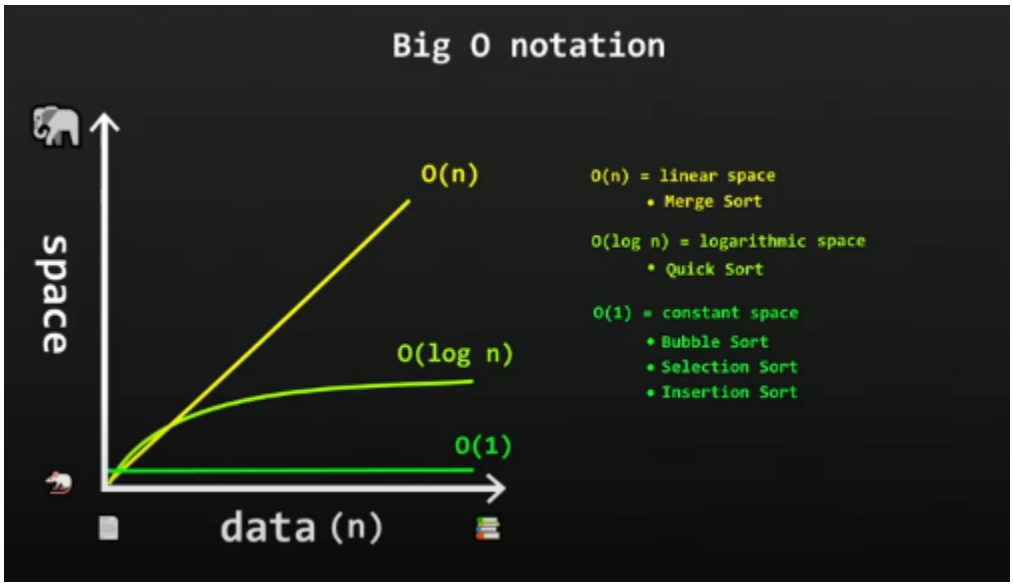
Algorithm 1 Quicksort(A, p, r)

```
1: if p < r then
2:     q ← Partition(A, p, r)
3:     Quicksort(A, p, q − 1)
4:     Quicksort(A, q + 1, r)
```

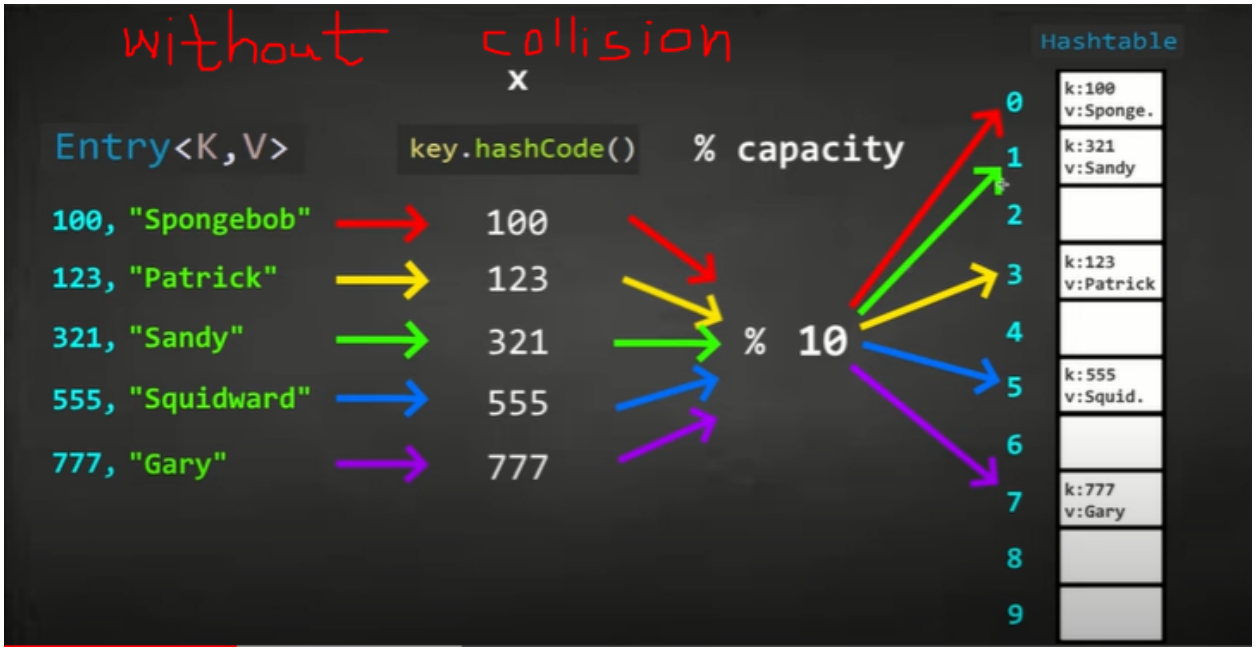
Algorithm 2 Partition(A, p, r)

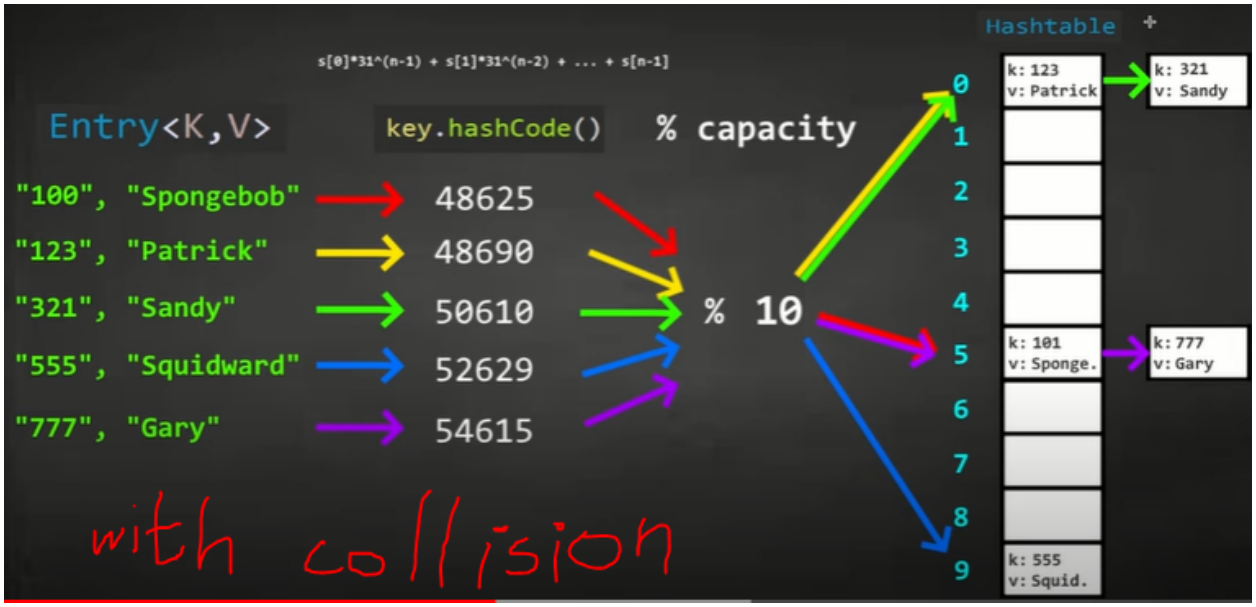
```
1: x ← A[r]
2: i ← p − 1
3: for j ← p to r − 1 do
4:     if A[j] ≤ x then
5:         i ← i + 1
6:         exchange A[i] with A[j]
7: exchange A[i + 1] with A[r]
8: return i + 1
```


Big O notation
O(n^2)
O(n log n)
time
data
O(n log n) = quasilinear
• quicksort
• mergesort
• heapsort
O(n^2) = quadratic
• insertion sort
• selection sort
• bubblesort
• mergesort
worst


Big O notation
O(n)
O(log n)
O(1)
space
data (n)
O(n) = linear space
• Merge Sort
O(log n) = logarithmic space
• Quick Sort
O(1) = constant space
• Bubble Sort
• Selection Sort
• Insertion Sort

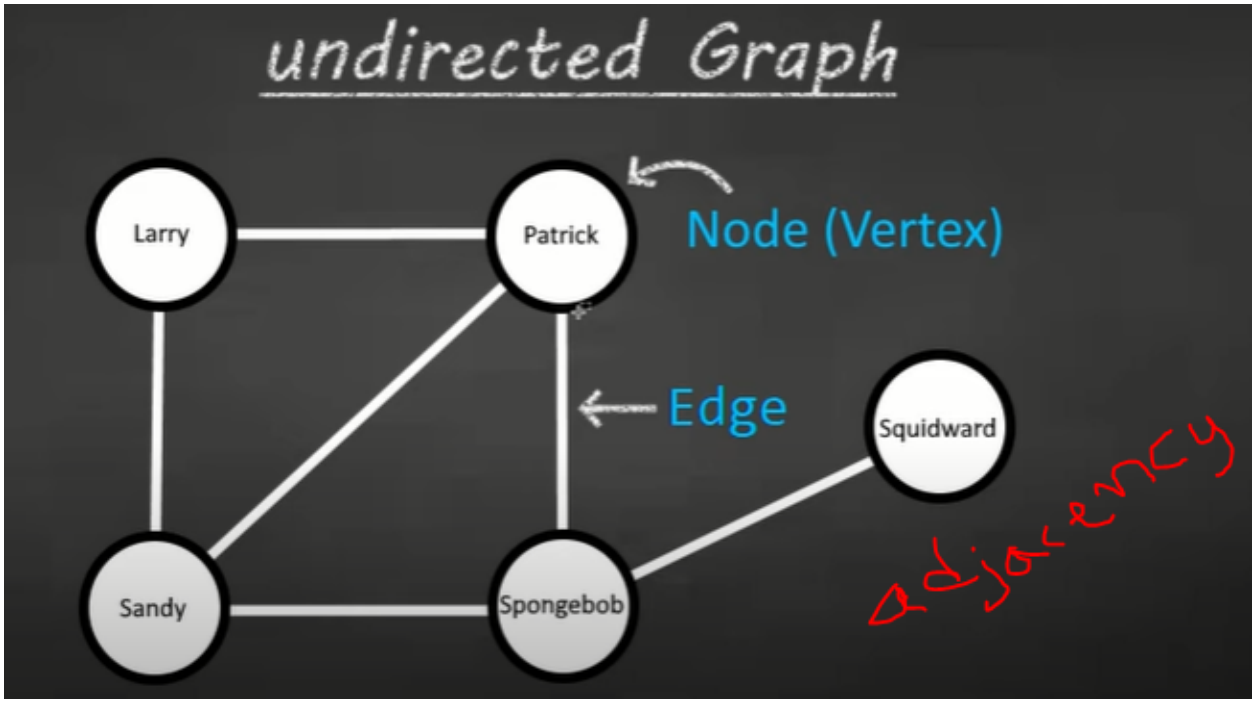
Hash Trees


without collision
Hashtable
Entry<K,V>
key.hashCode()
% capacity
100, "Spongebob"
123, "Patrick"
321, "Sandy"
555, "Squidward"
777, "Gary"
100
123
321
555
777
% 10
0
1
2
3
4
5
6
7
8
9
k:100
v:Sponge.
k:321
v:Sandy
k:123
v:Patrick
k:555
v:Squid.
k:777
v:Gary

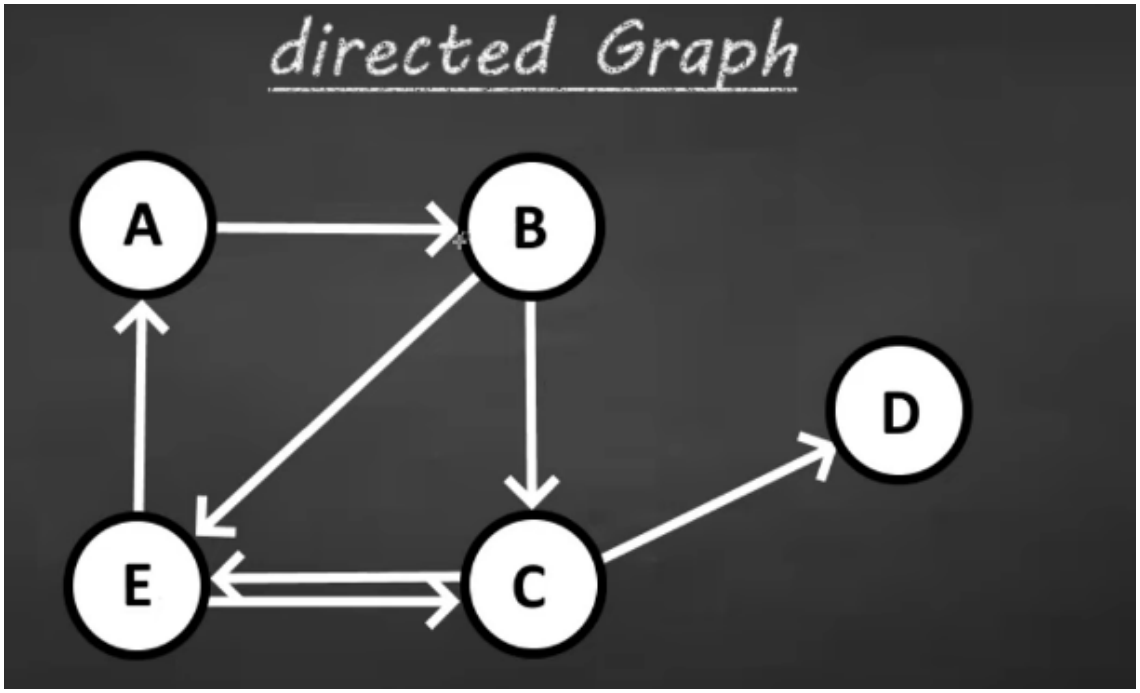

s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
Hashtable
Entry<K,V>
key.hashCode()
% capacity
"100", "Spongebob"
"123", "Patrick"
"321", "Sandy"
"555", "Squidward"
"777", "Gary"
48625
48690
50610
52629
54615
% 10
with collision
k: 123
v: Patrick
k: 321
v: Sandy
k: 101
v: Sponge.
k: 777
v: Gary
k: 555
v: Squid.

Graphs

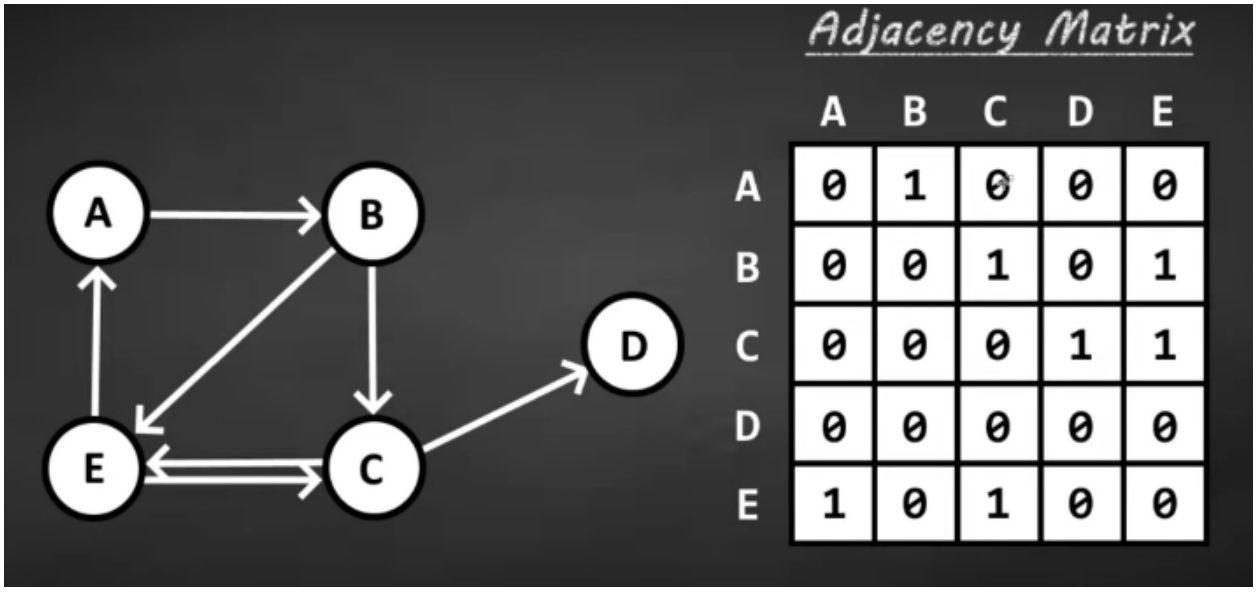
1) Undirected Graph



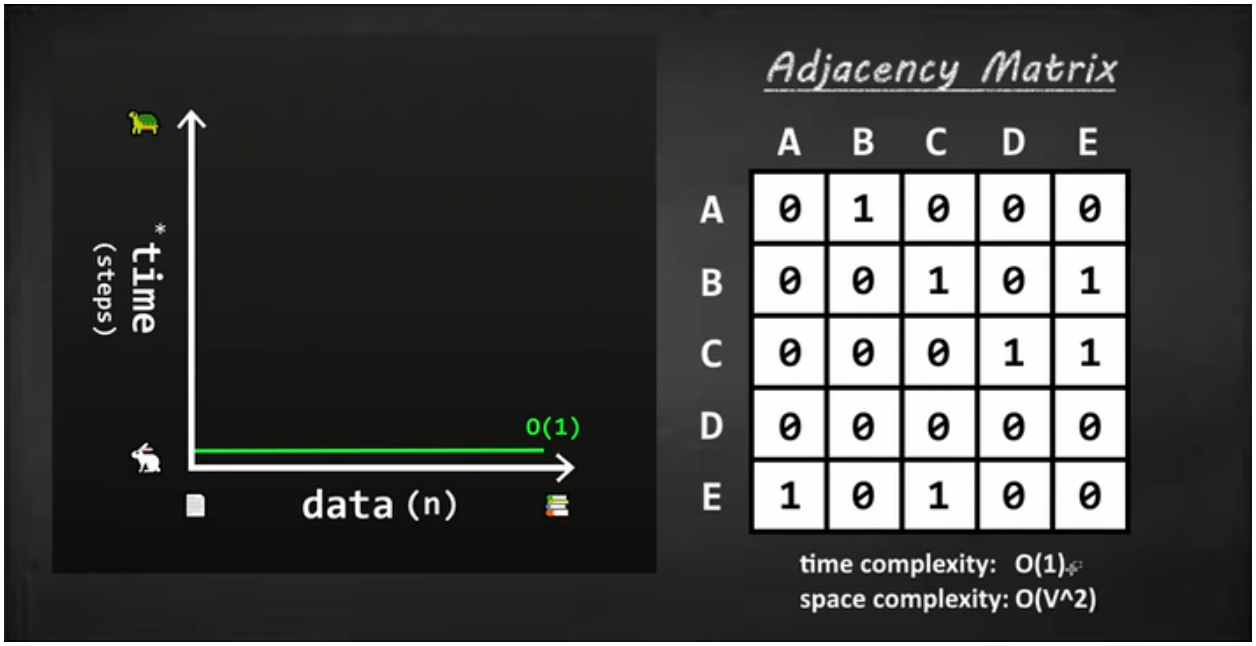
2) Directed Graph



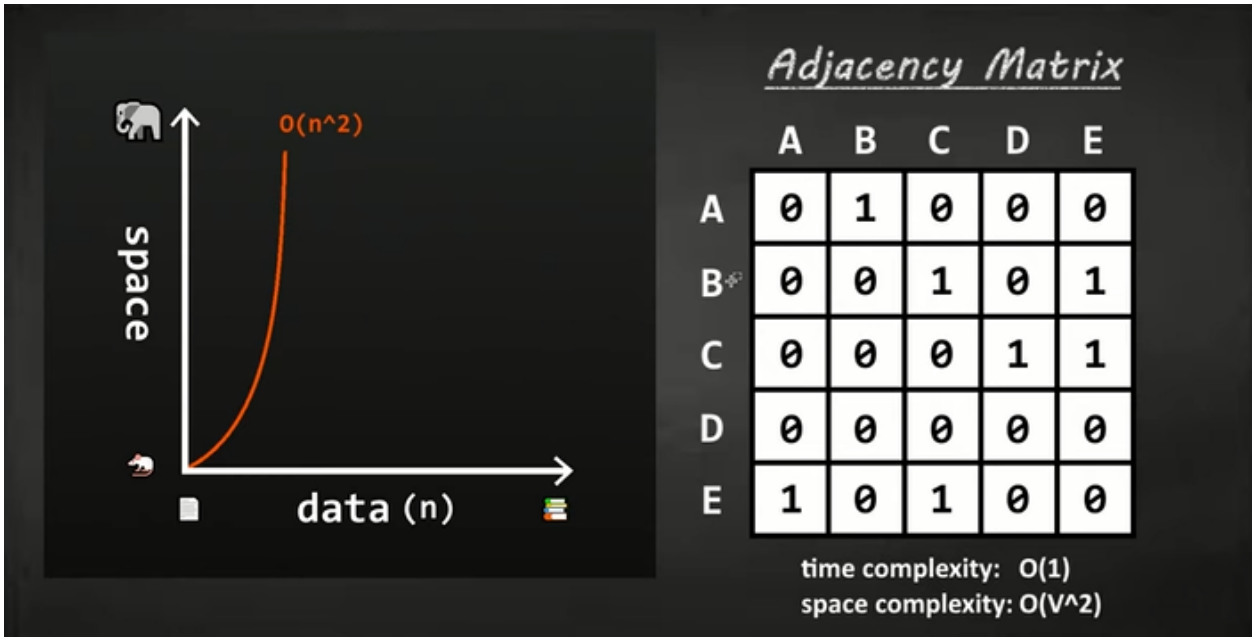
3) Methods to represent a graph



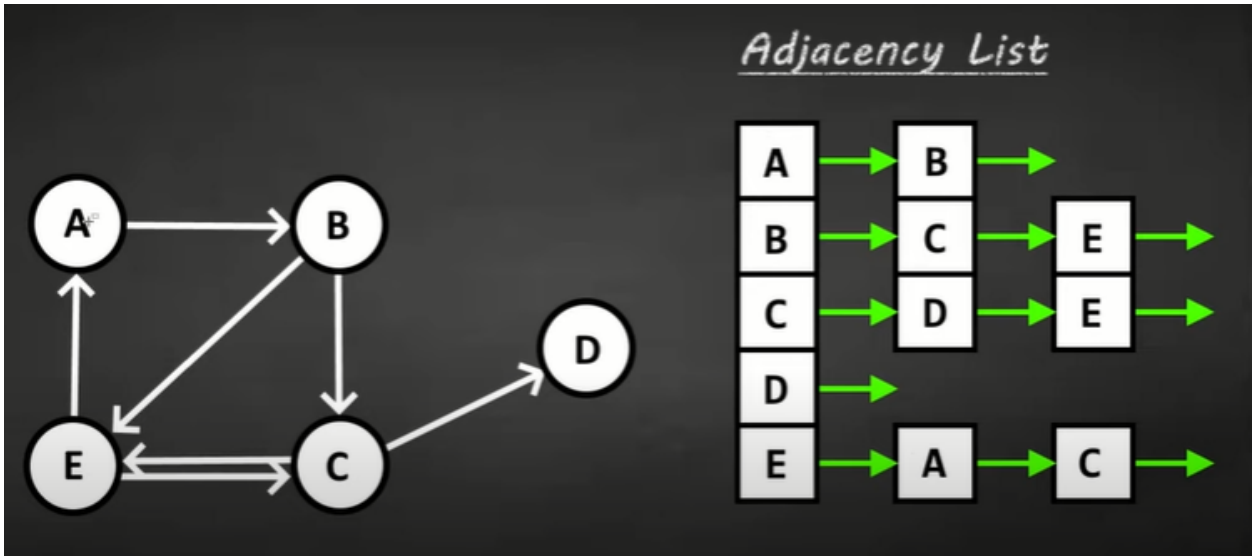
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	0	1
C	0	0	0	1	1
D	0	0	0	0	0
E	1	0	1	0	0



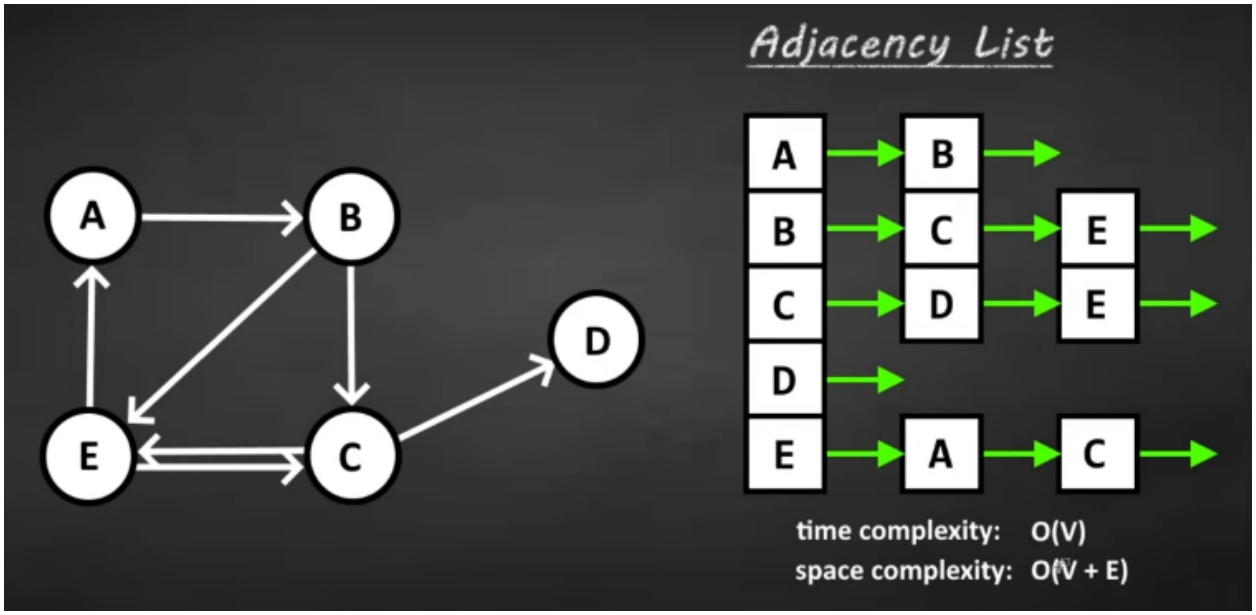
Time complexity =O(1)



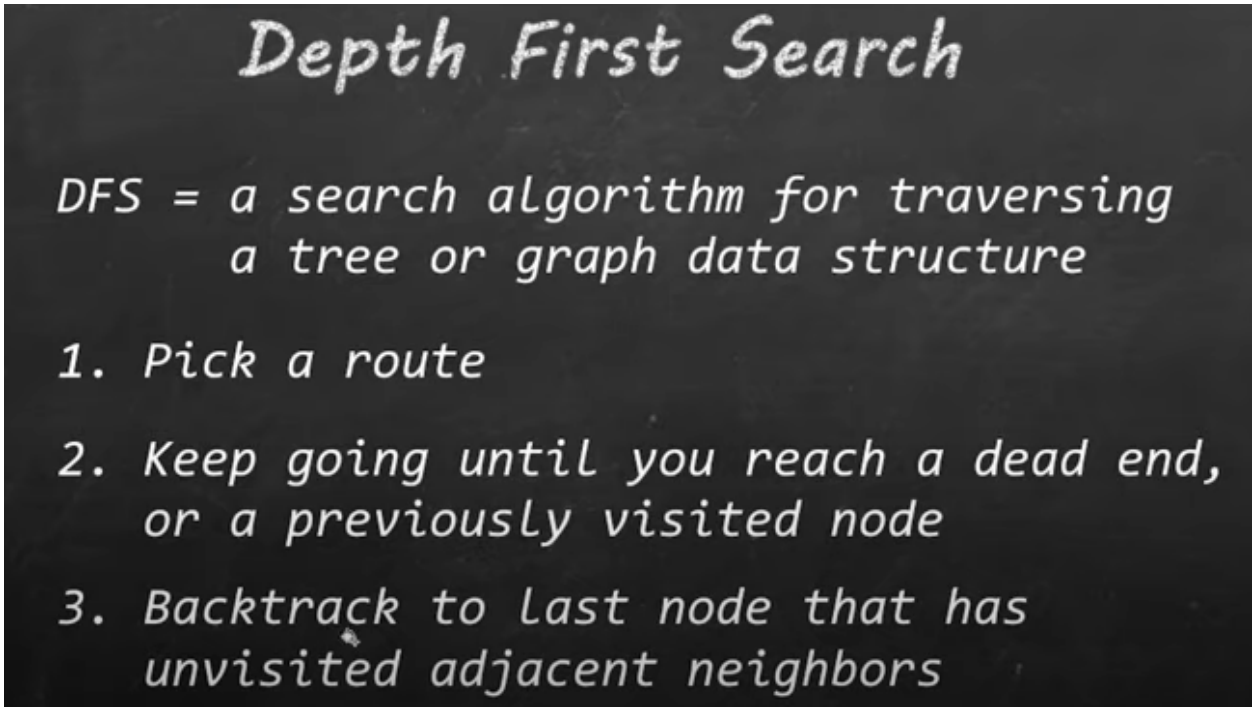
Space complexity = O(N^2)
N=5 Space complexity =25

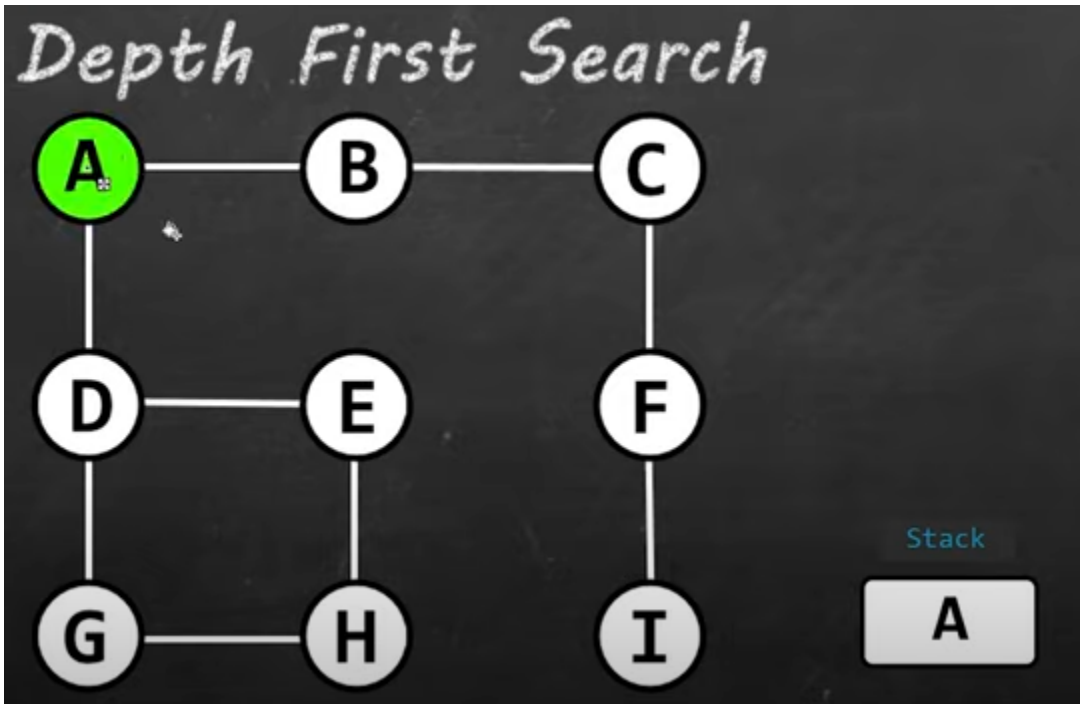


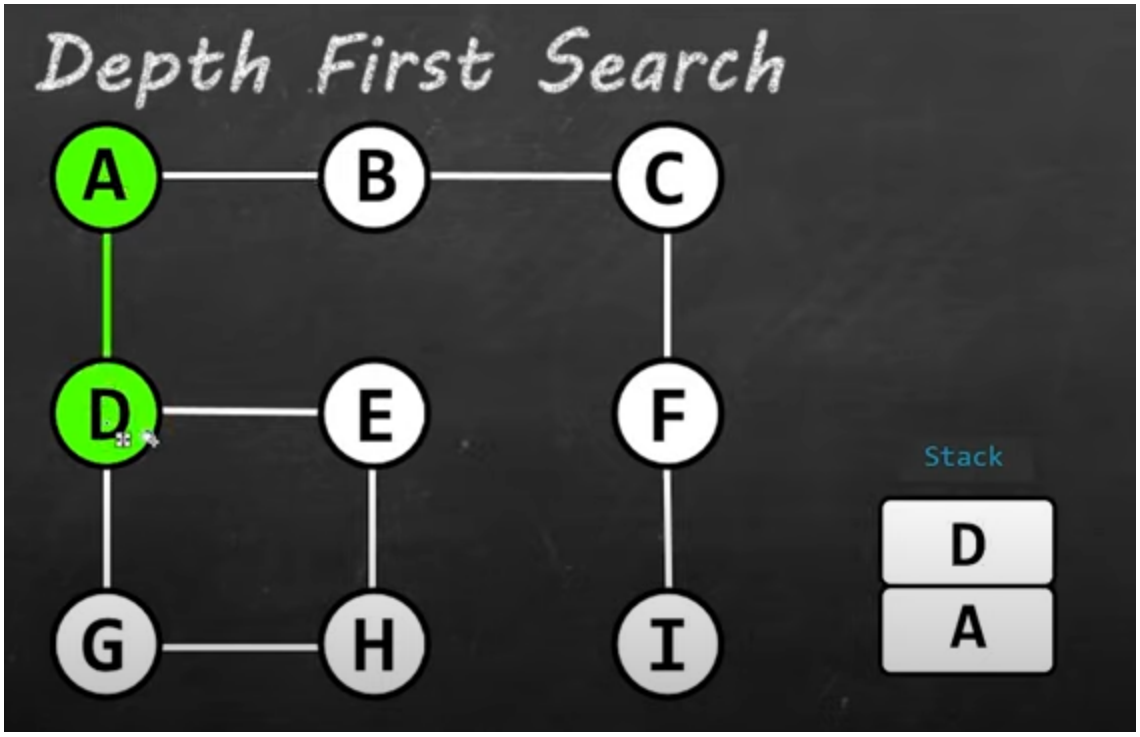
It’s an ArrayList of LinkedLists.

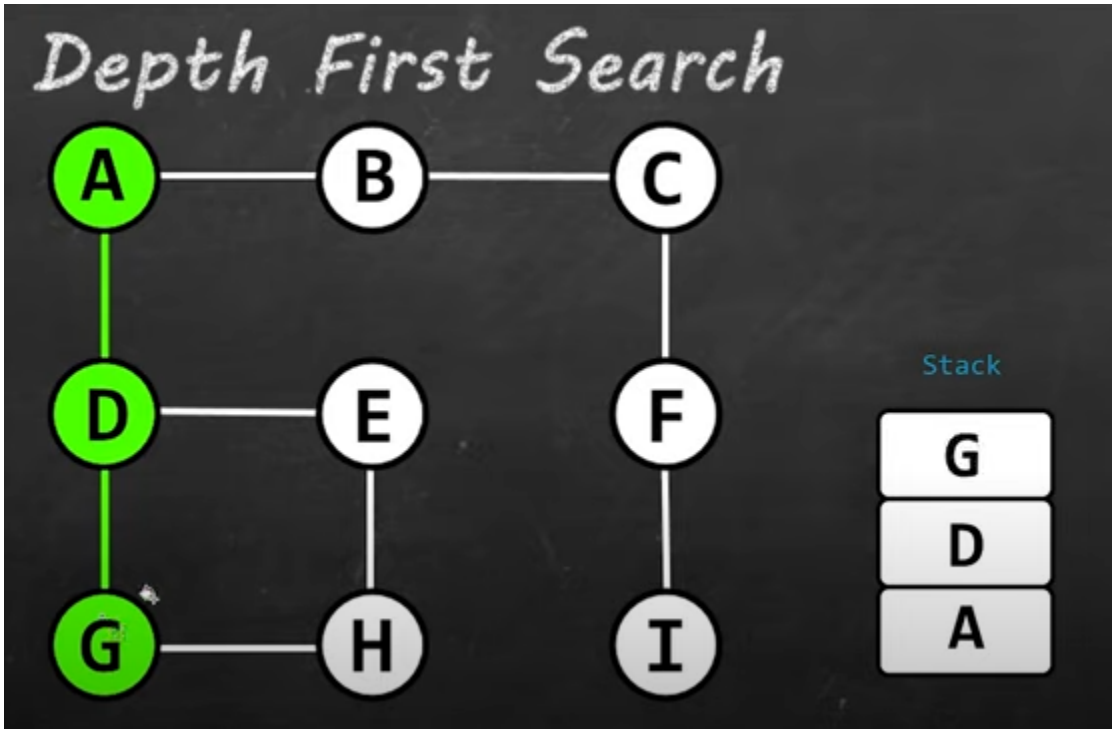


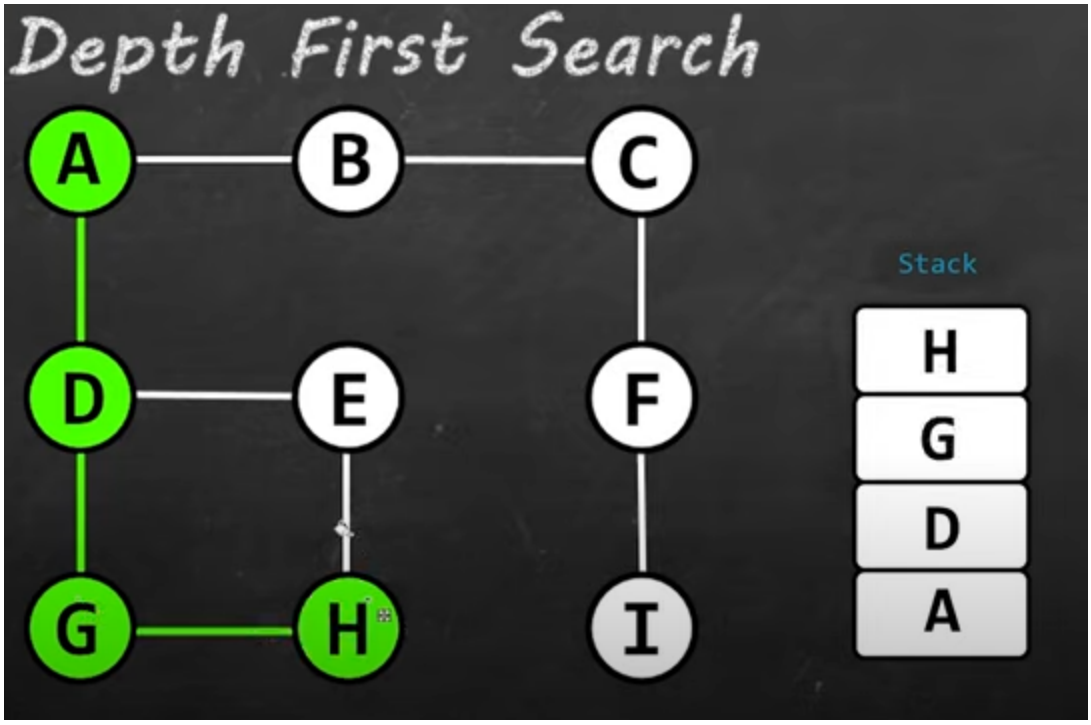
DFS

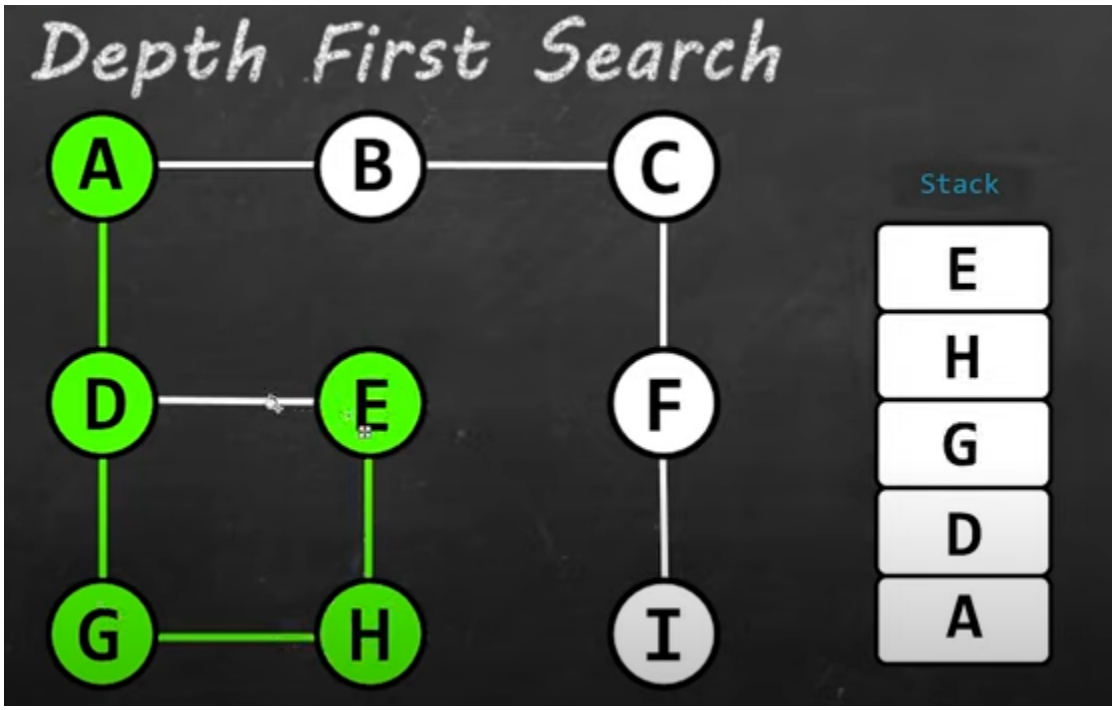

Depth First Search
DFS = a search algorithm for traversing a tree or graph data structure
1. Pick a route
2. Keep going until you reach a dead end, or a previously visited node
3. Backtrack to last node that has unvisited adjacent neighbors

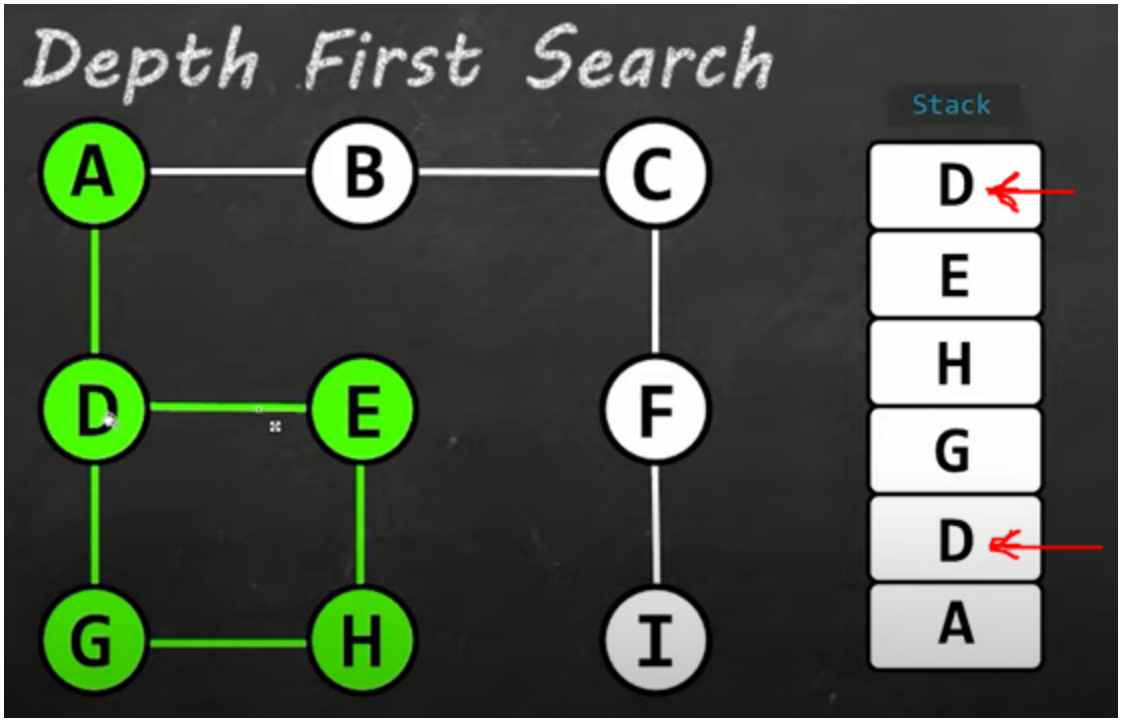

Depth First Search
A
B
C
D
E
F
G
H
I
Stack
A

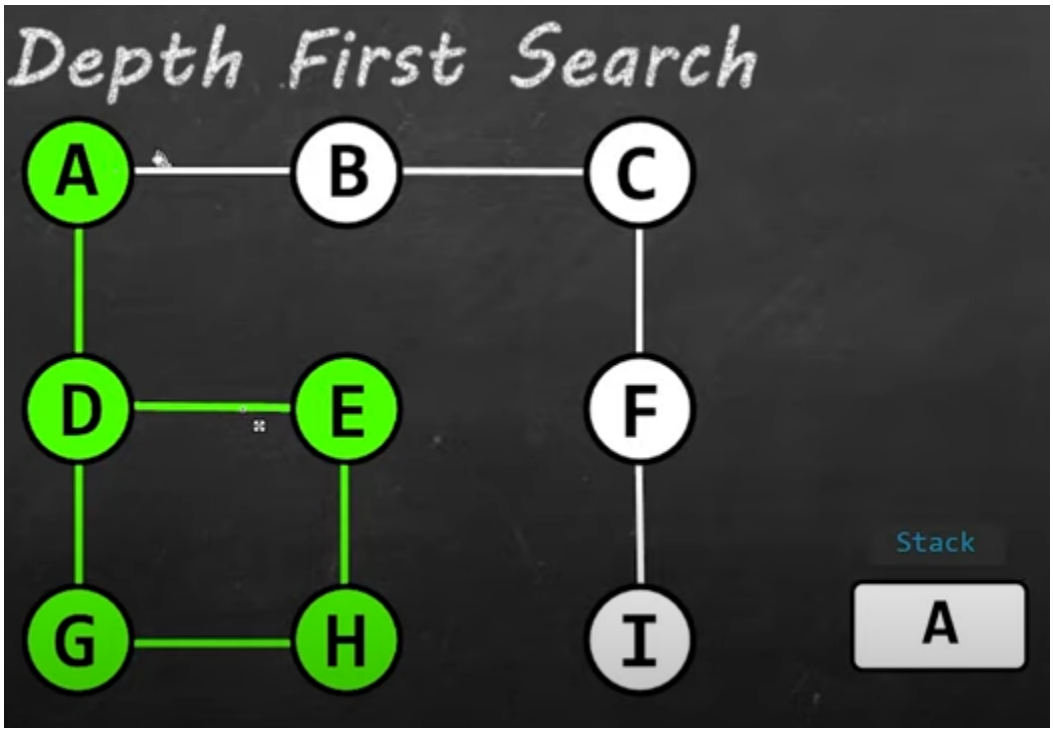

Depth First Search
A
B
C
D
E
F
G
H
I
Stack
D
A

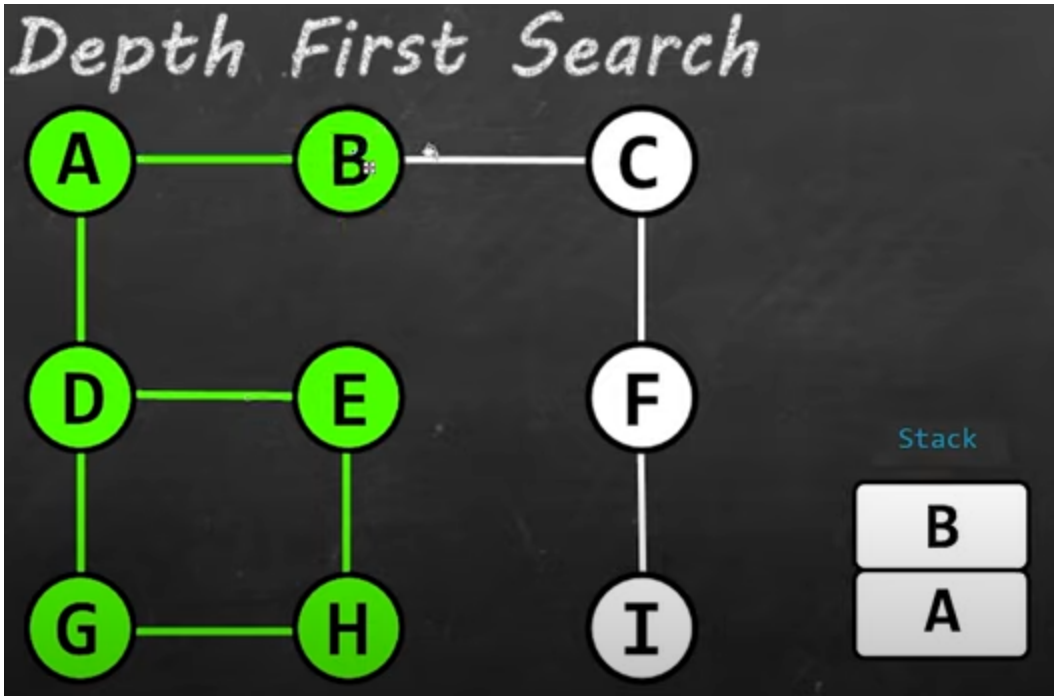

Depth First Search
A
B
C
D
E
F
G
H
I
Stack
G
D
A

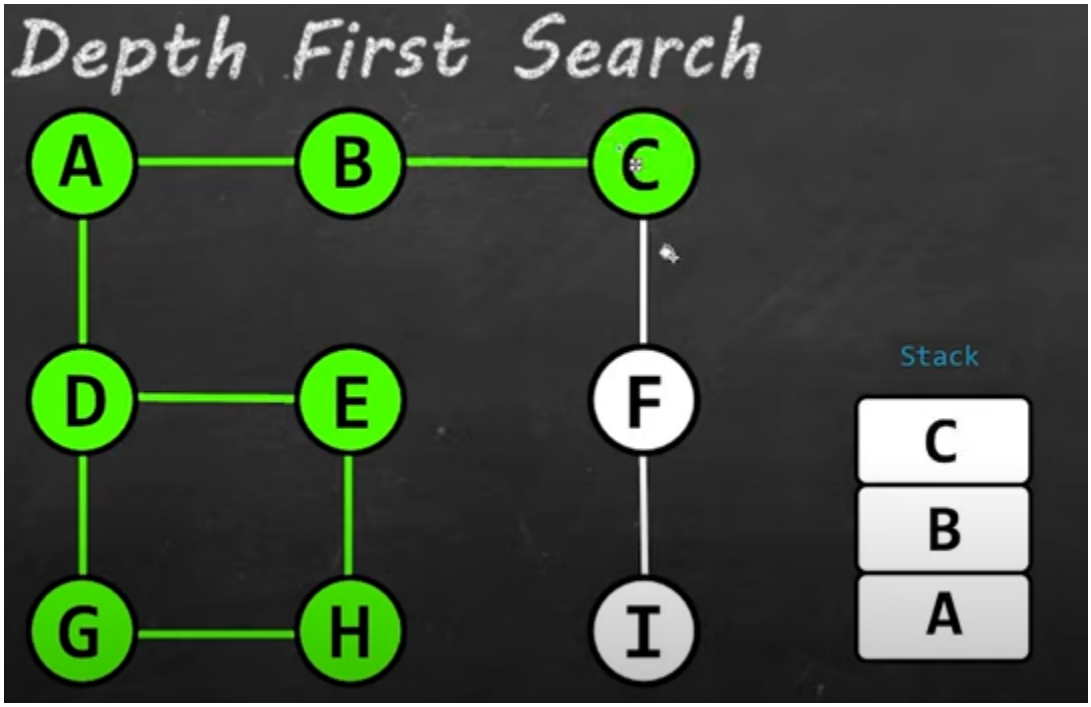

Depth First Search
A
B
C
D
E
F
G
H
I
Stack
H
G
D
A

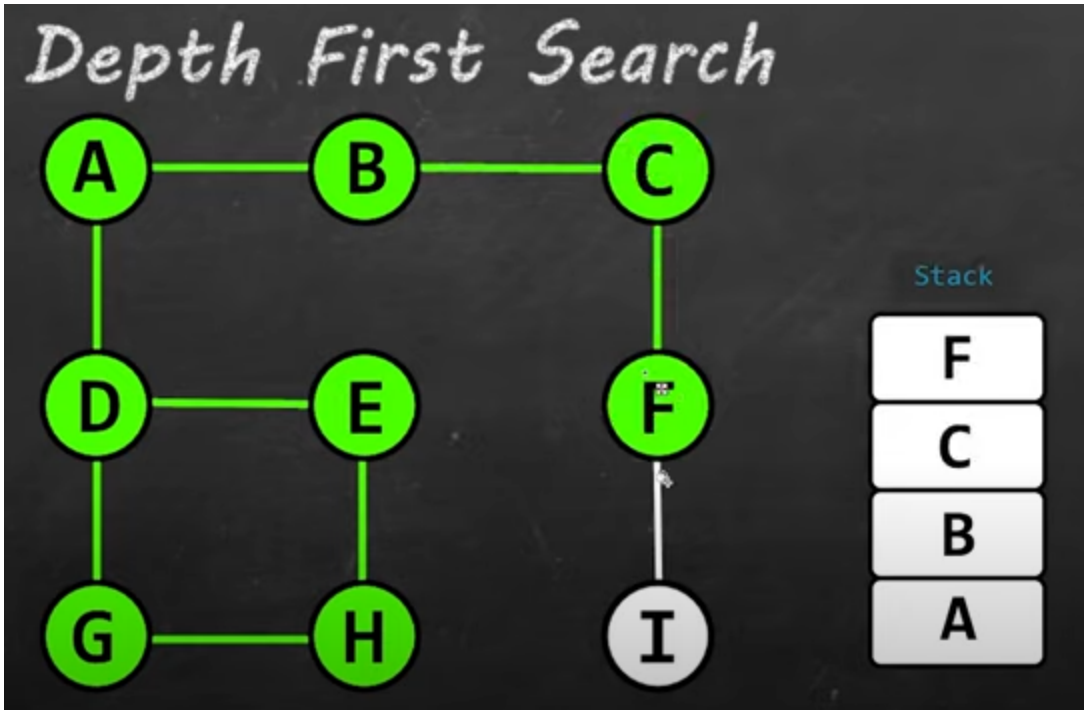

Depth First Search
A
B
C
D
E
F
G
H
I
Stack
E
H
G
D
A

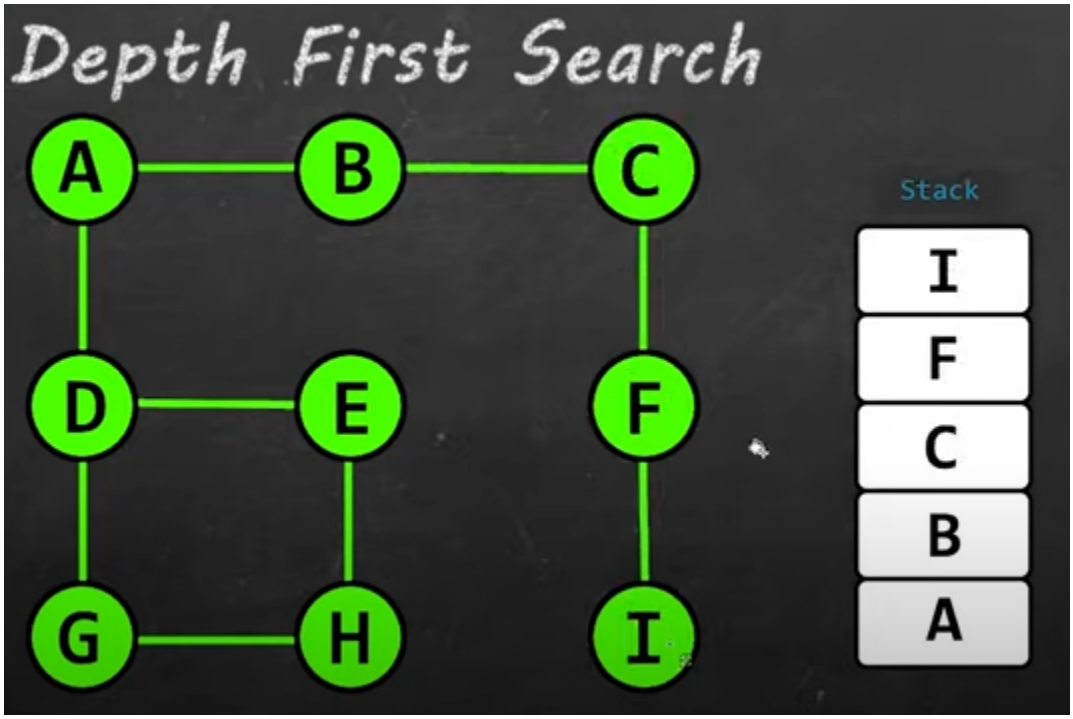

Depth First Search
A
B
C
D
E
F
G
H
I
Stack
D
E
H
G
D
A


Depth First Search
A
B
C
D
E
F
G
H
I
Stack
A


Depth First Search
A
B
C
D
E
F
G
H
I
Stack
B
A

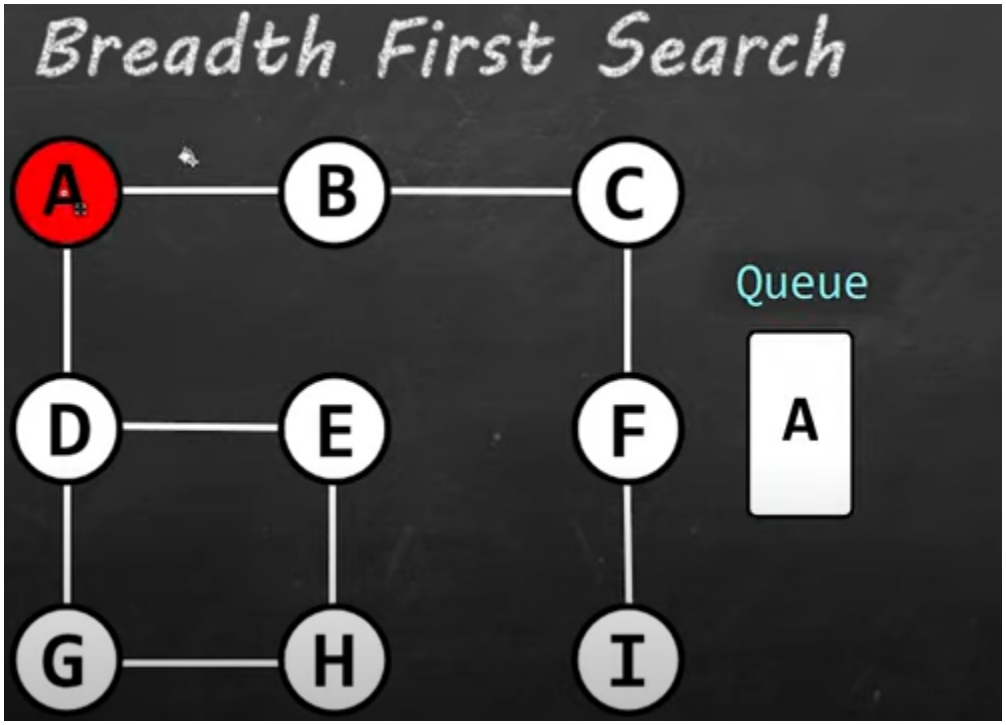

Depth First Search
A
B
C
D
E
F
G
H
I
Stack
C
B
A

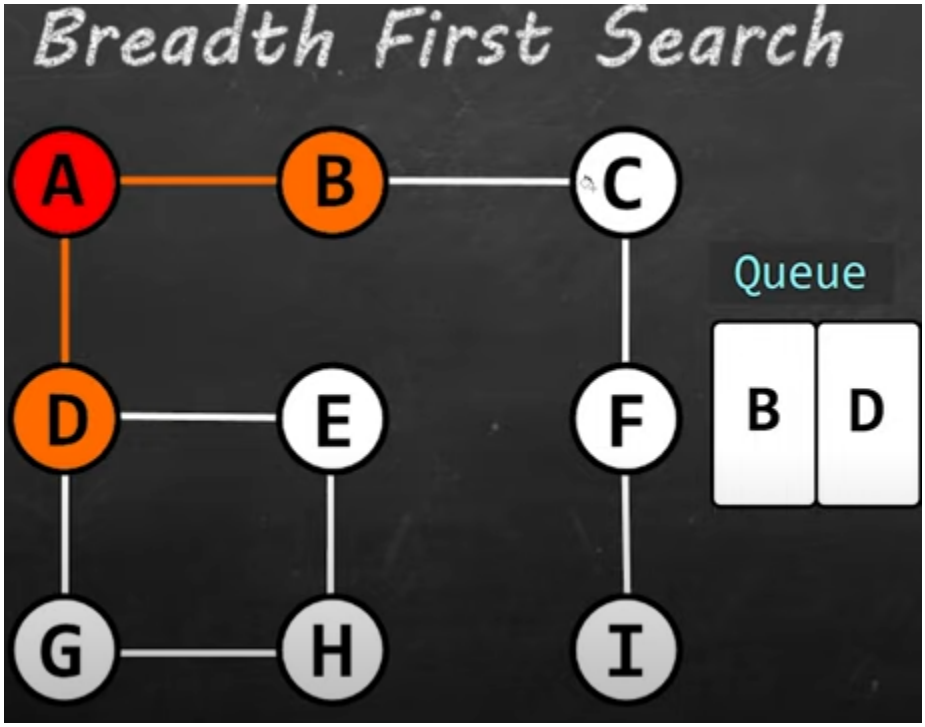


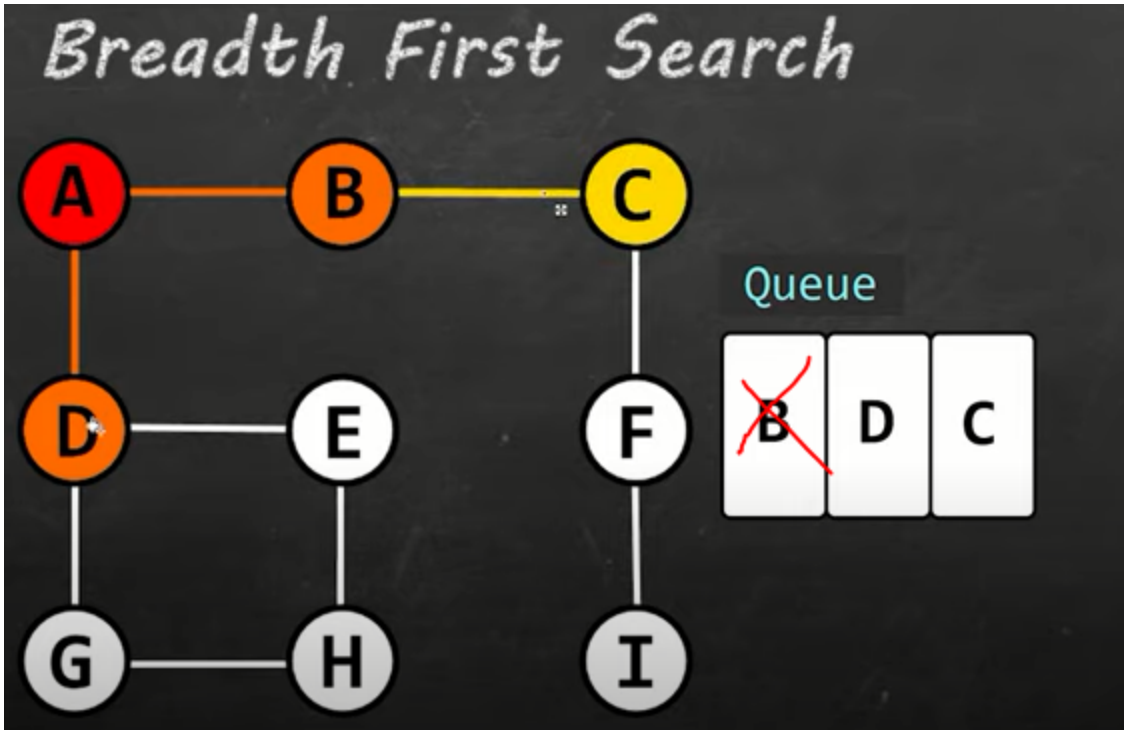


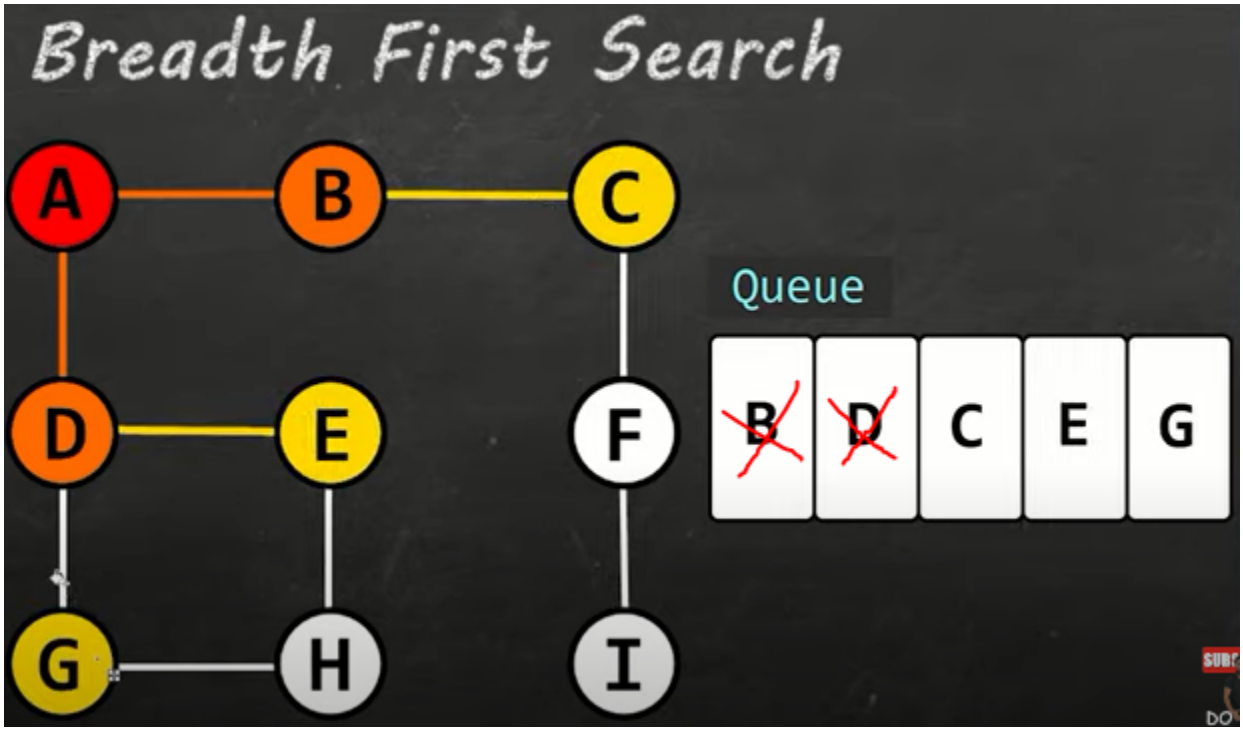
We got ‘I’ in the end the node we wanted.

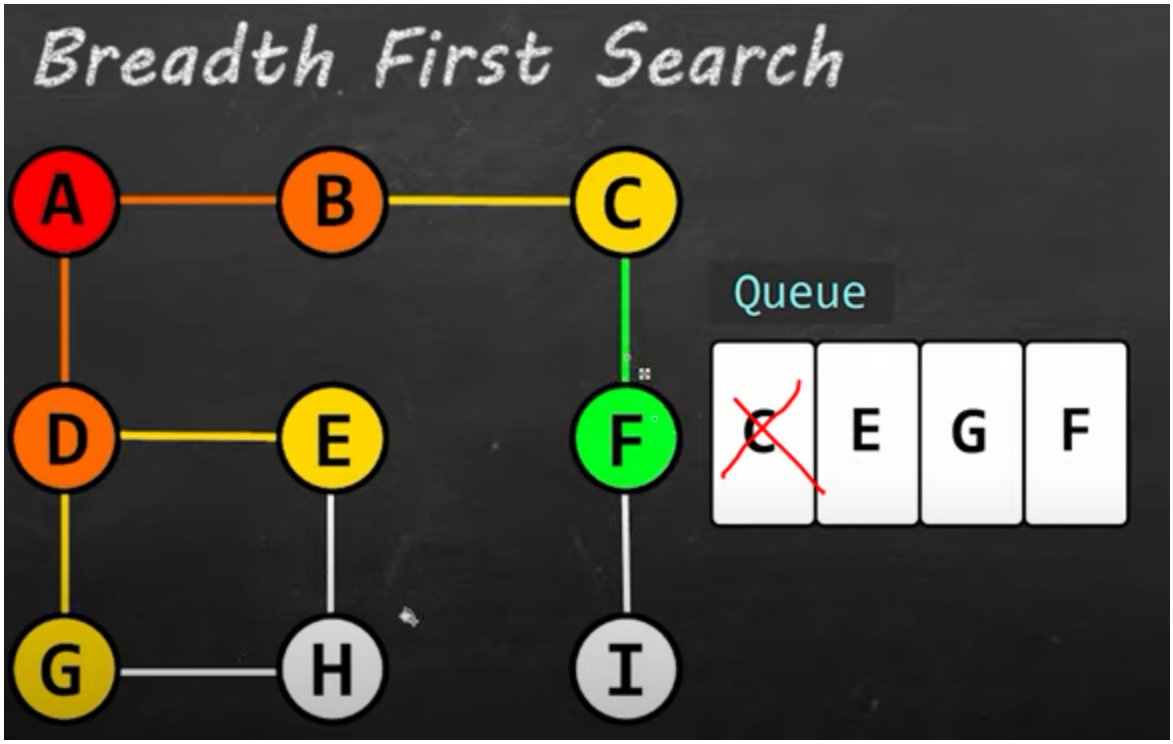
BFS

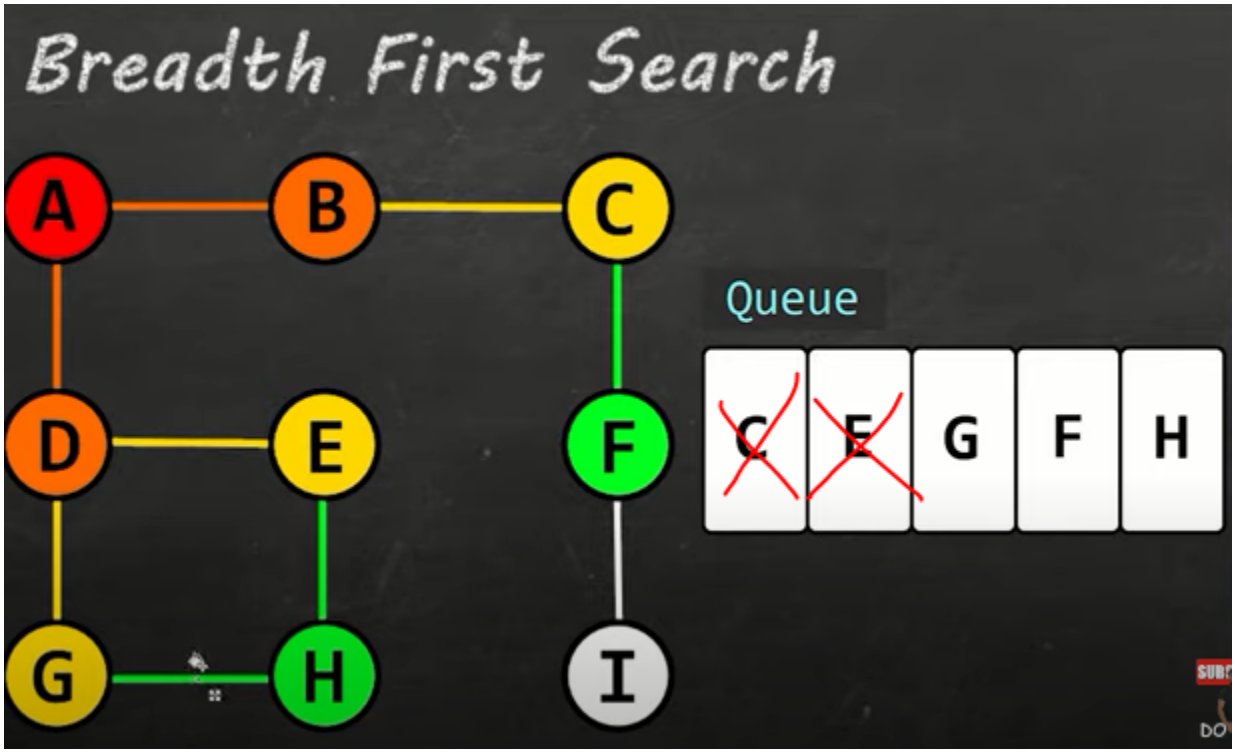

Breadth First Search
A
B
C
D
E
F
G
H
I
Queue
A

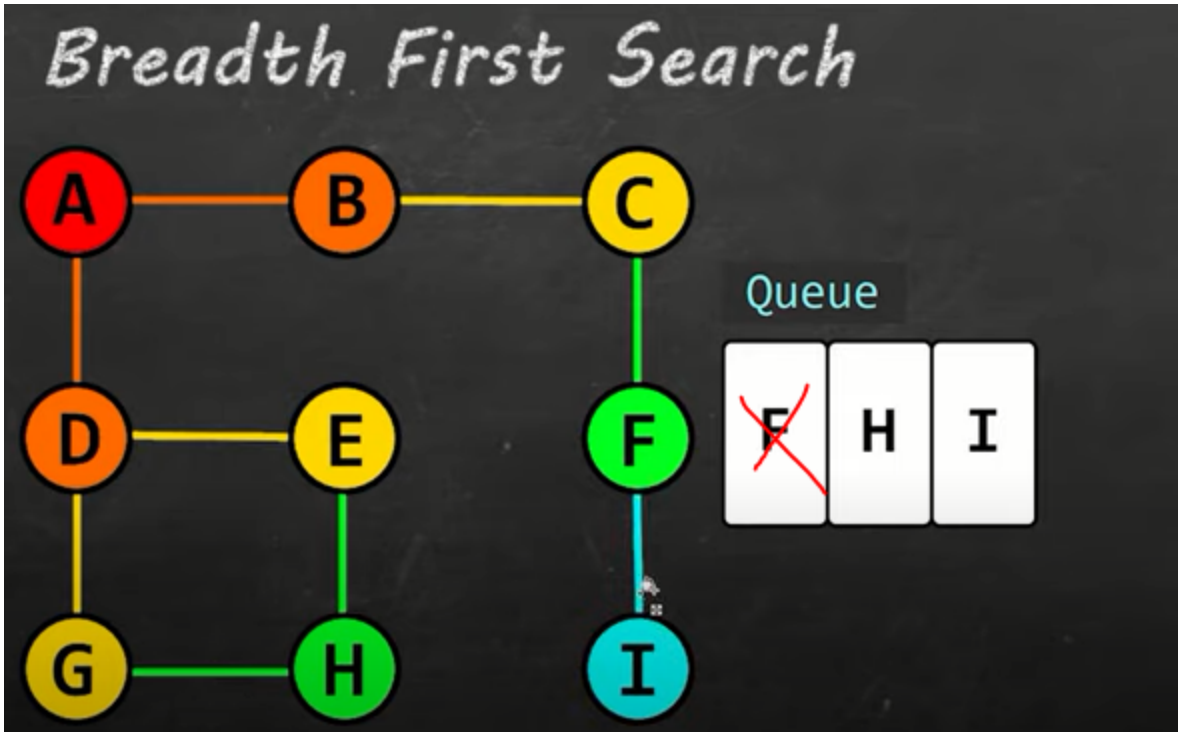

Breadth First Search
A
B
C
D
E
F
G
H
I
Queue
B
D

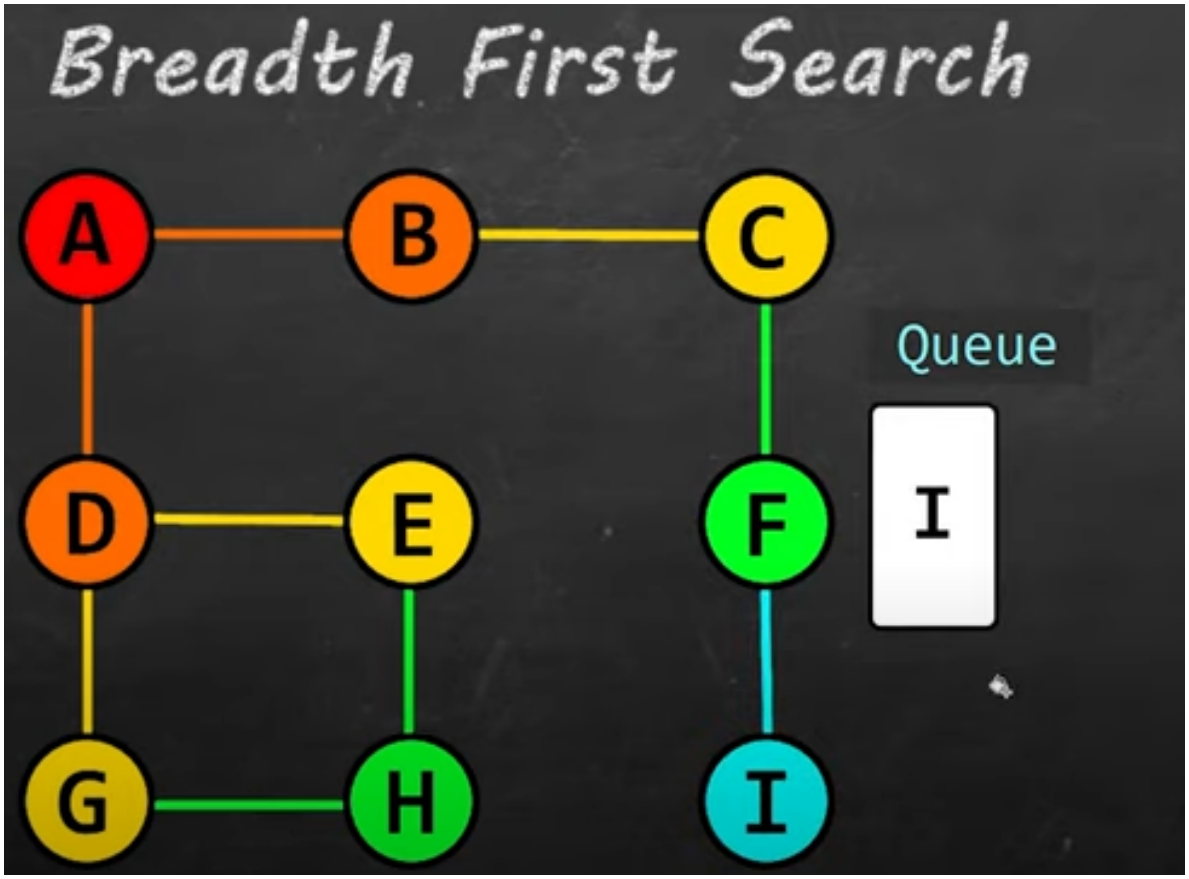

Breadth First Search
A
B
C
D
E
F
G
H
I
Queue
B
D
C


Breadth First Search
A
B
C
D
E
F
G
H
I
Queue
B
D
C
E
G

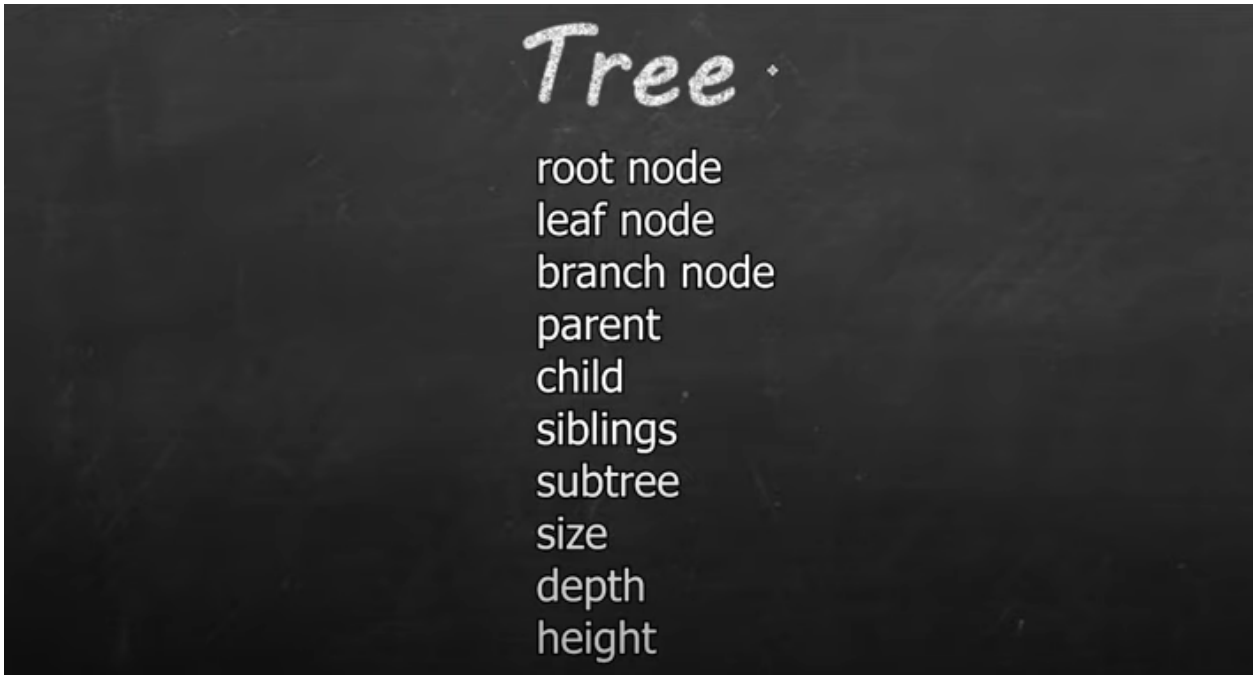

Breadth First Search
A
B
C
D
E
F
G
H
I
Queue
C
E
G
F

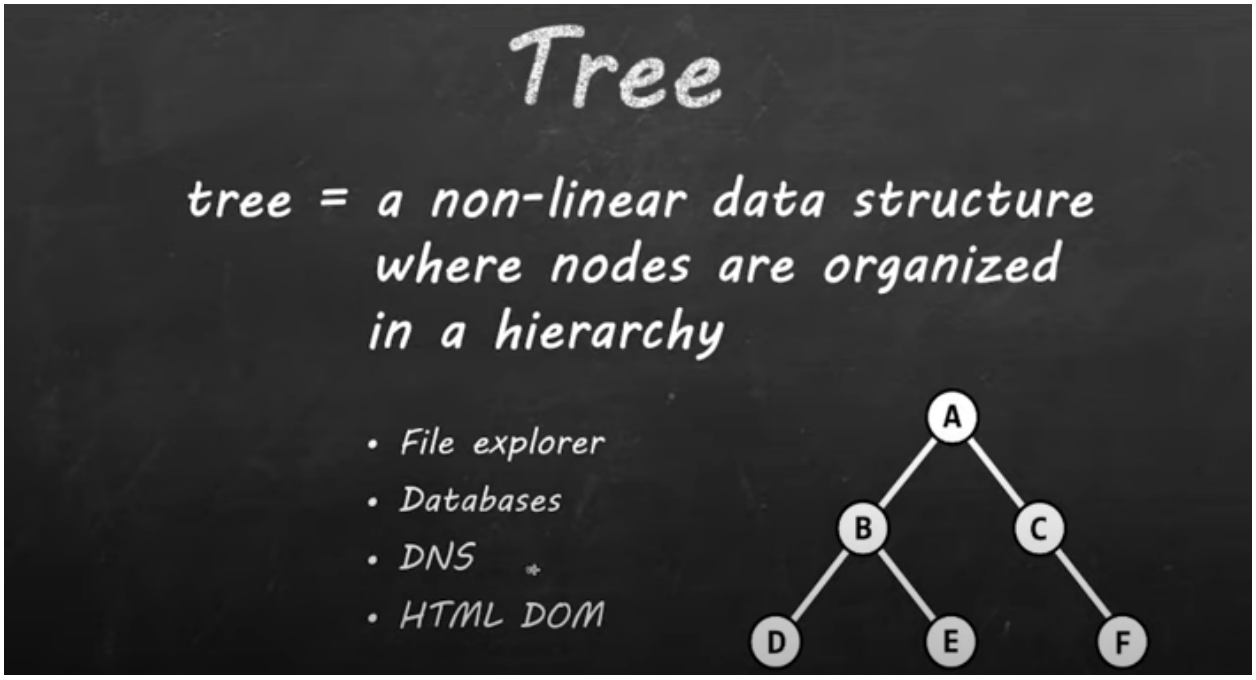

Breadth First Search
A
B
C
D
E
F
G
H
I
Queue
C
E
G
F
H

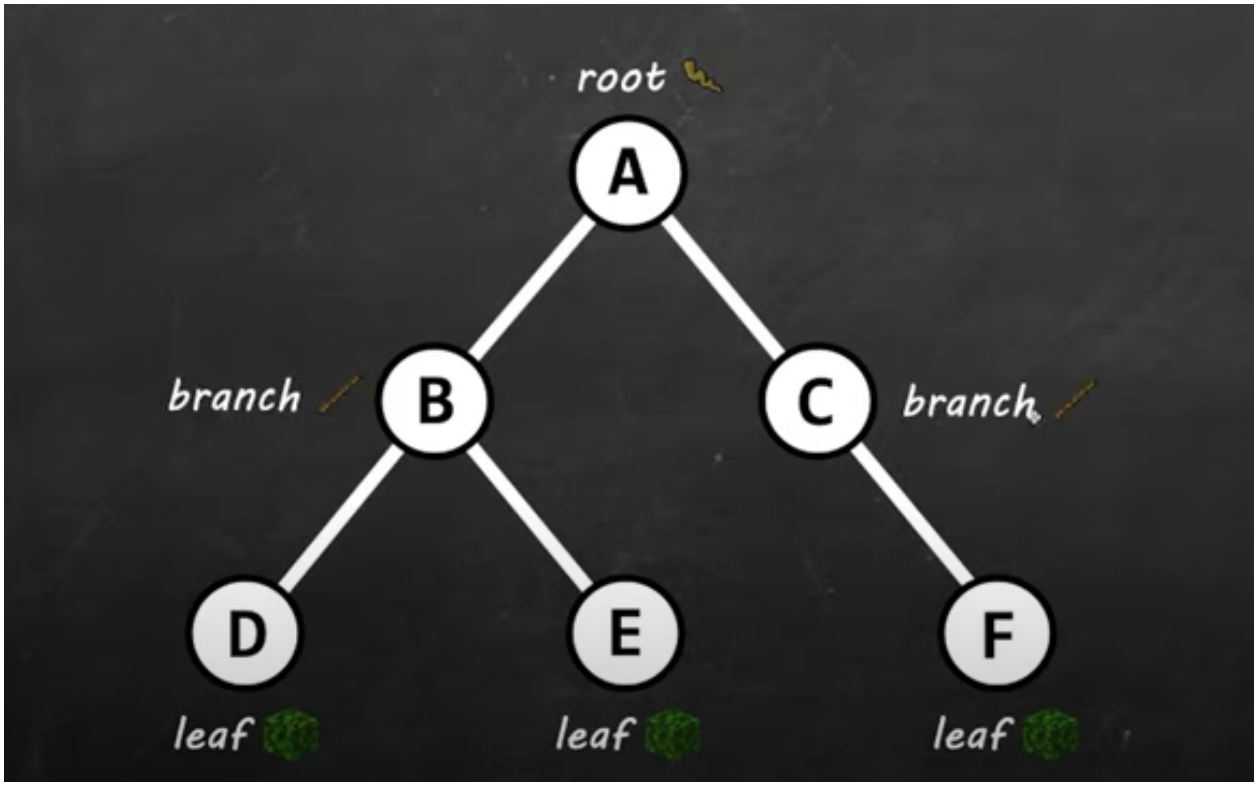

Breadth First Search
A
B
C
D
E
F
G
H
I
Queue
F
H
I

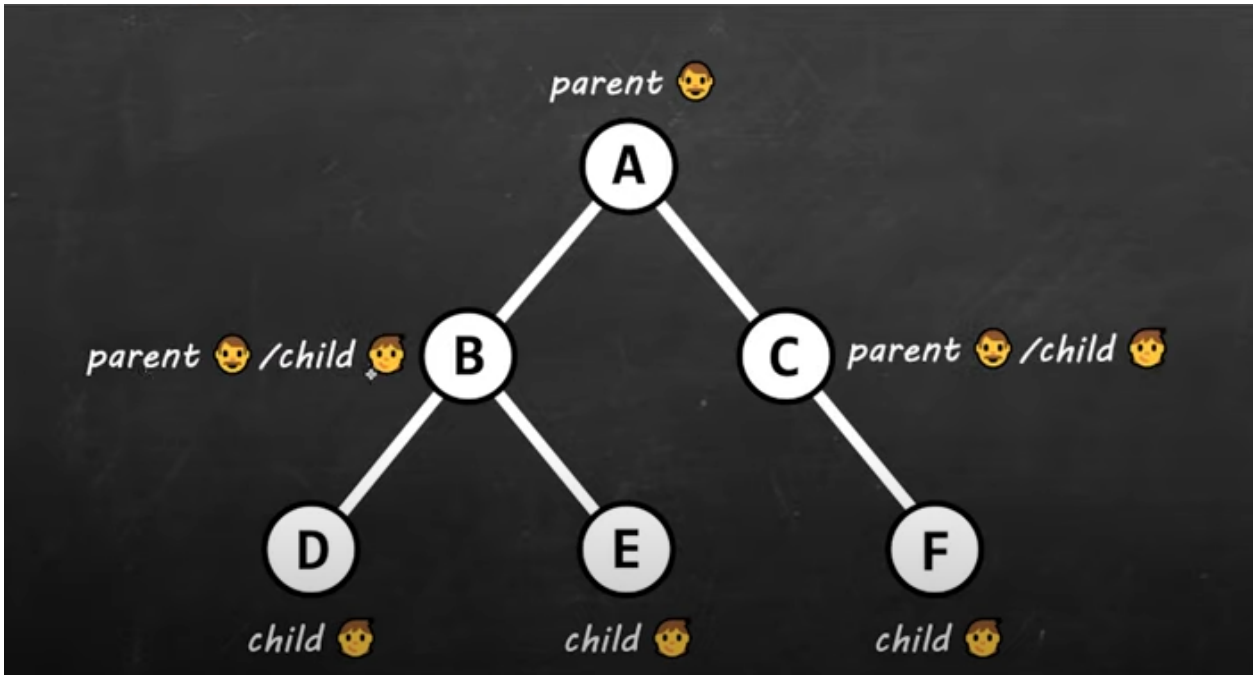

Breadth First Search
A
B
C
D
E
F
G
H
I
Queue
I

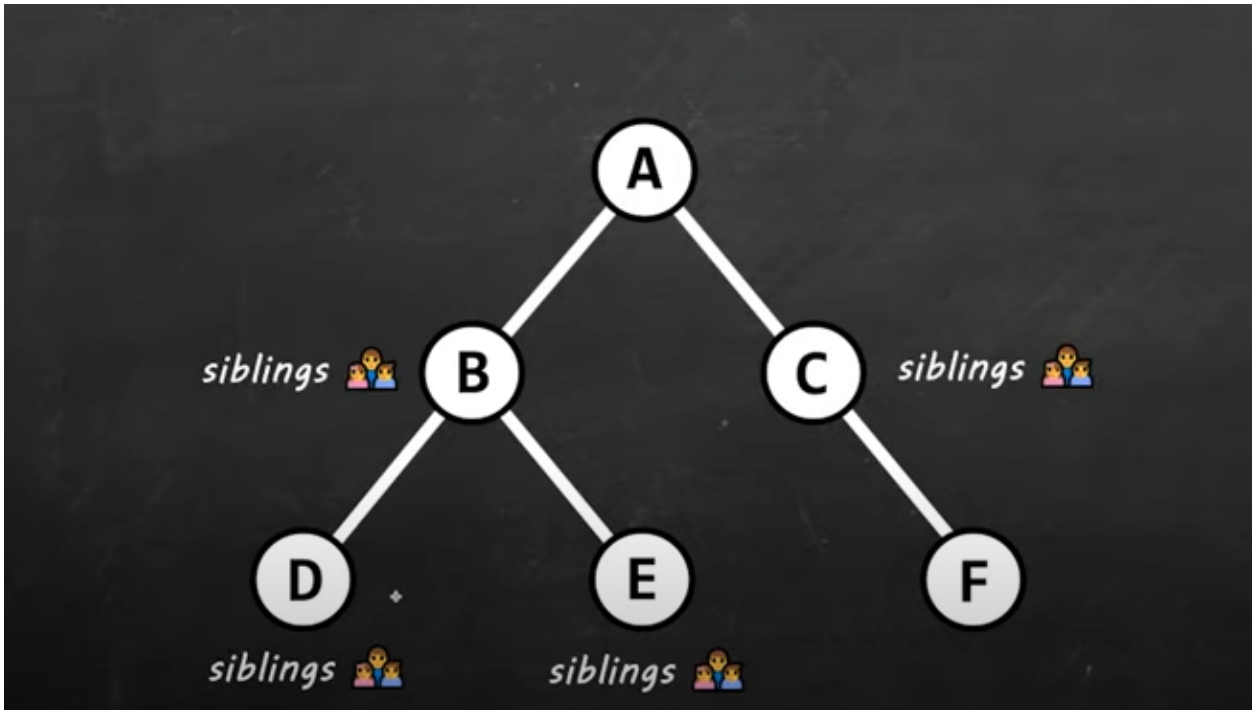
Tree Data Structure

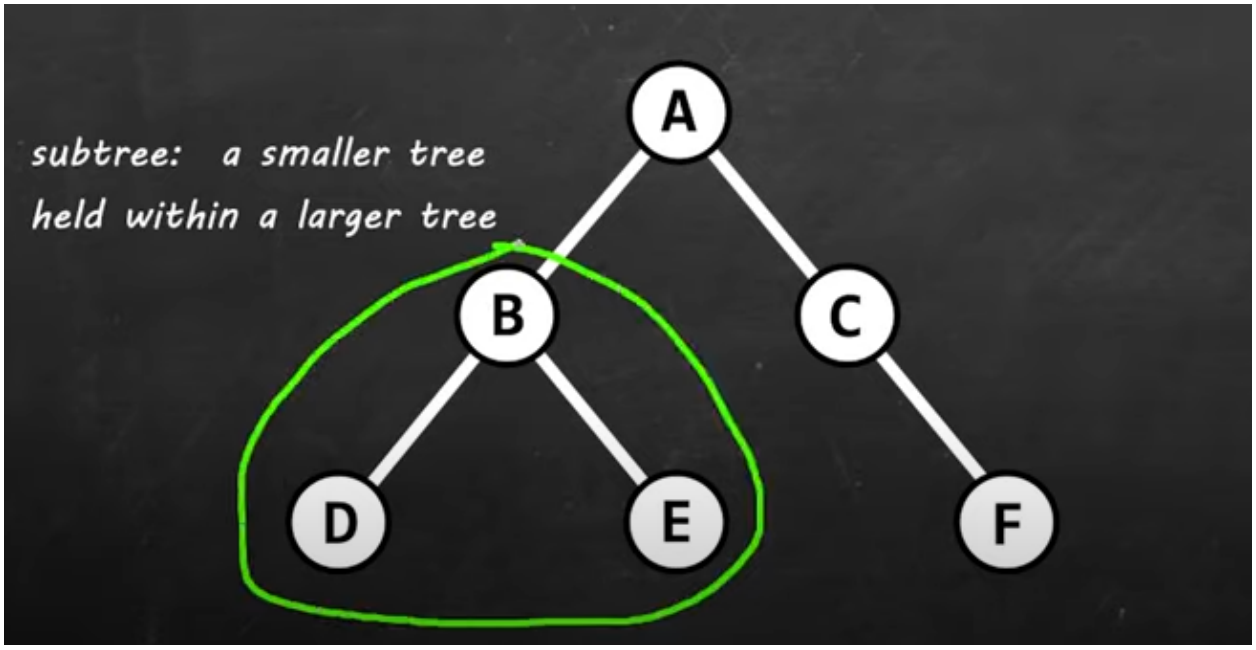


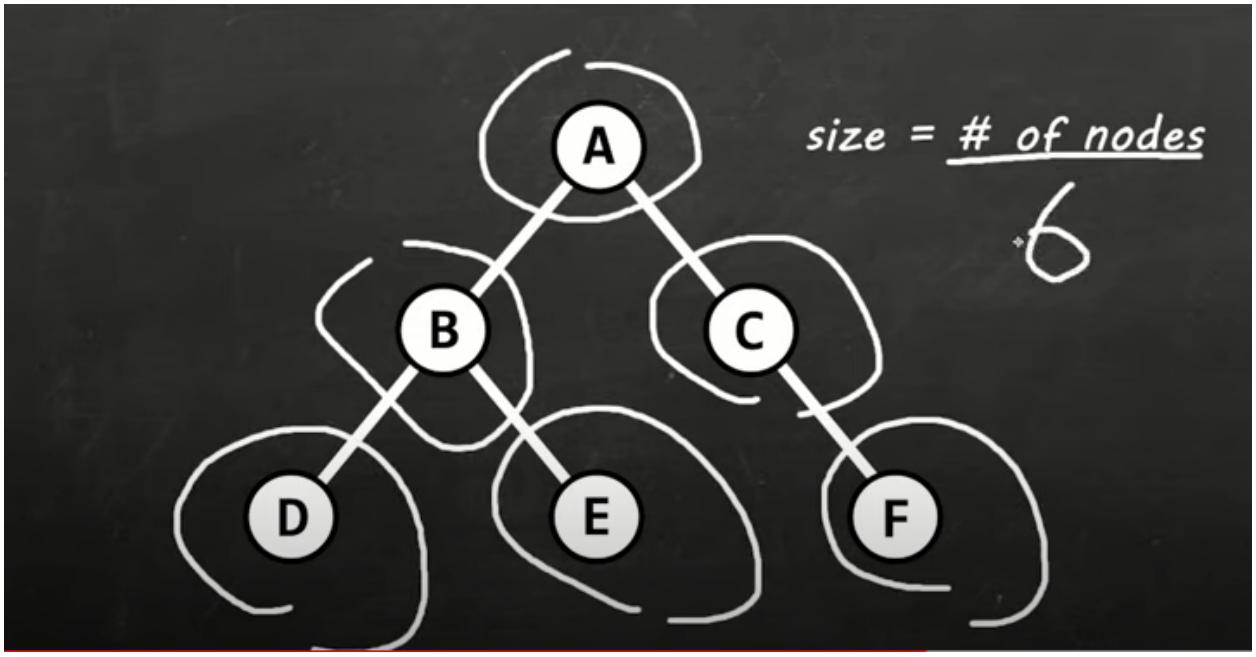


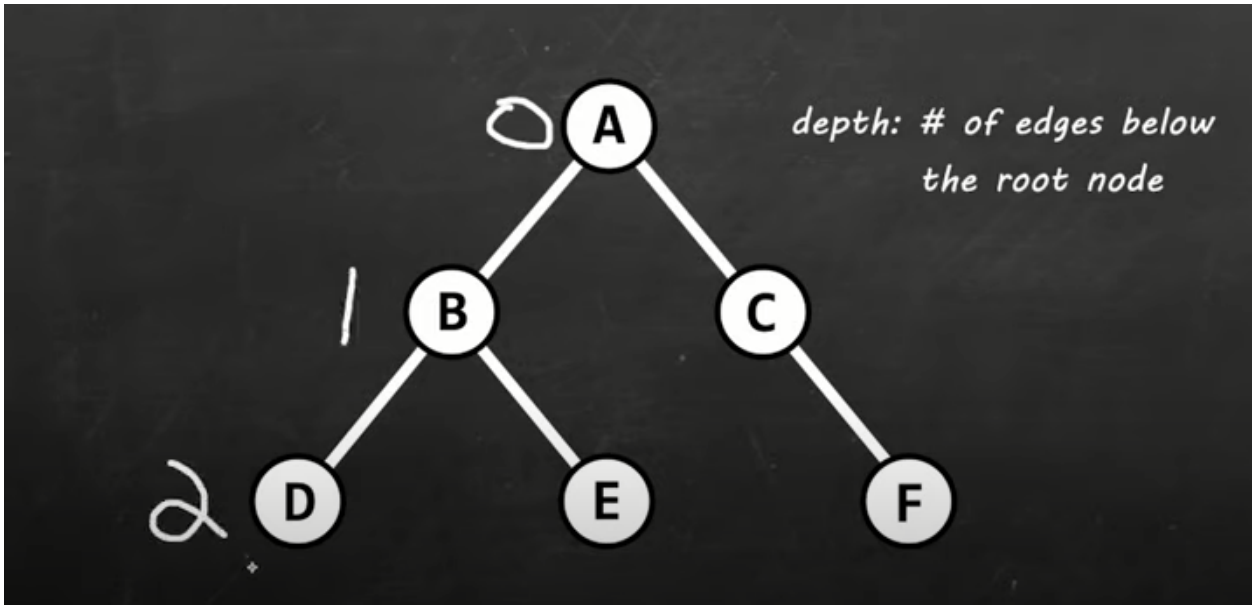

root
A
branch
B
C
branch
D
E
F
leaf
leaf
leaf

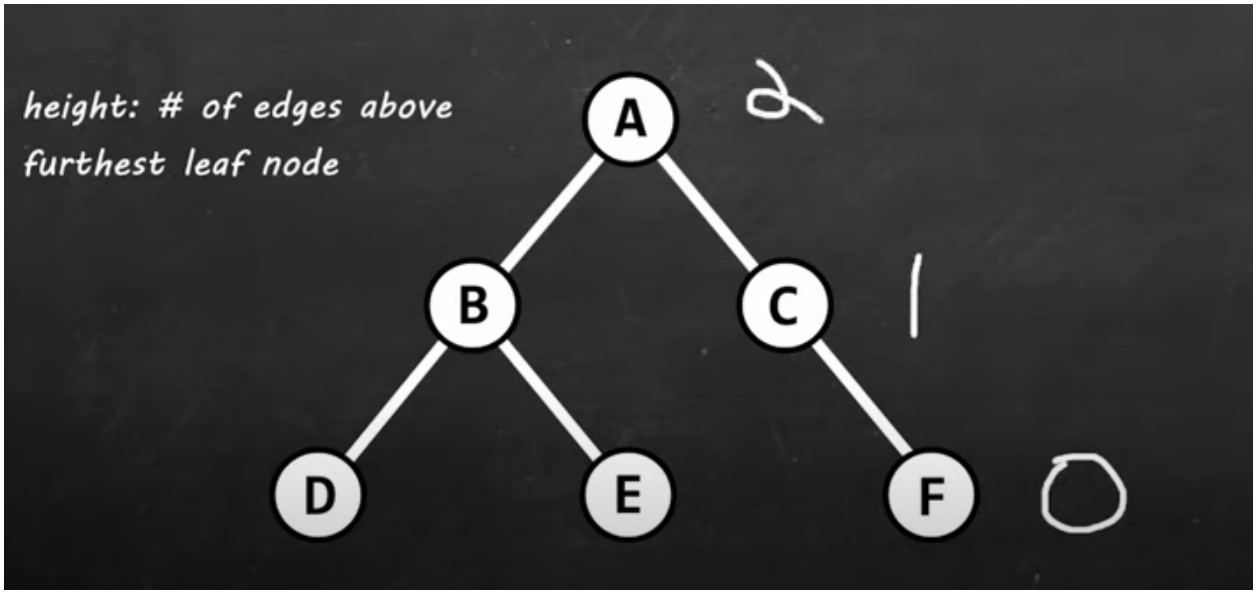

parent
A
parent /child
B
C
parent /child
D
E
F
child
child
child

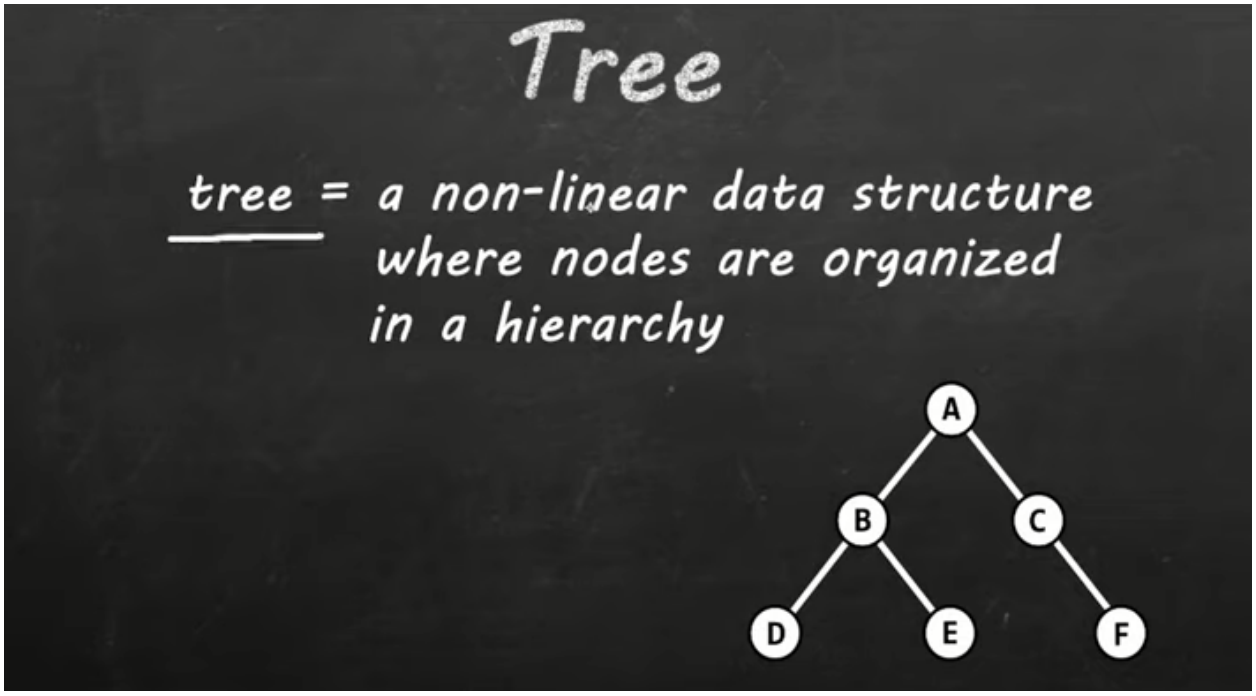

A
siblings
B
C
siblings
D
E
F
siblings
siblings


subtree: a smaller tree
held within a larger tree
A
B
C
D
E
F


A
B
C
D
E
F
size = # of nodes
6

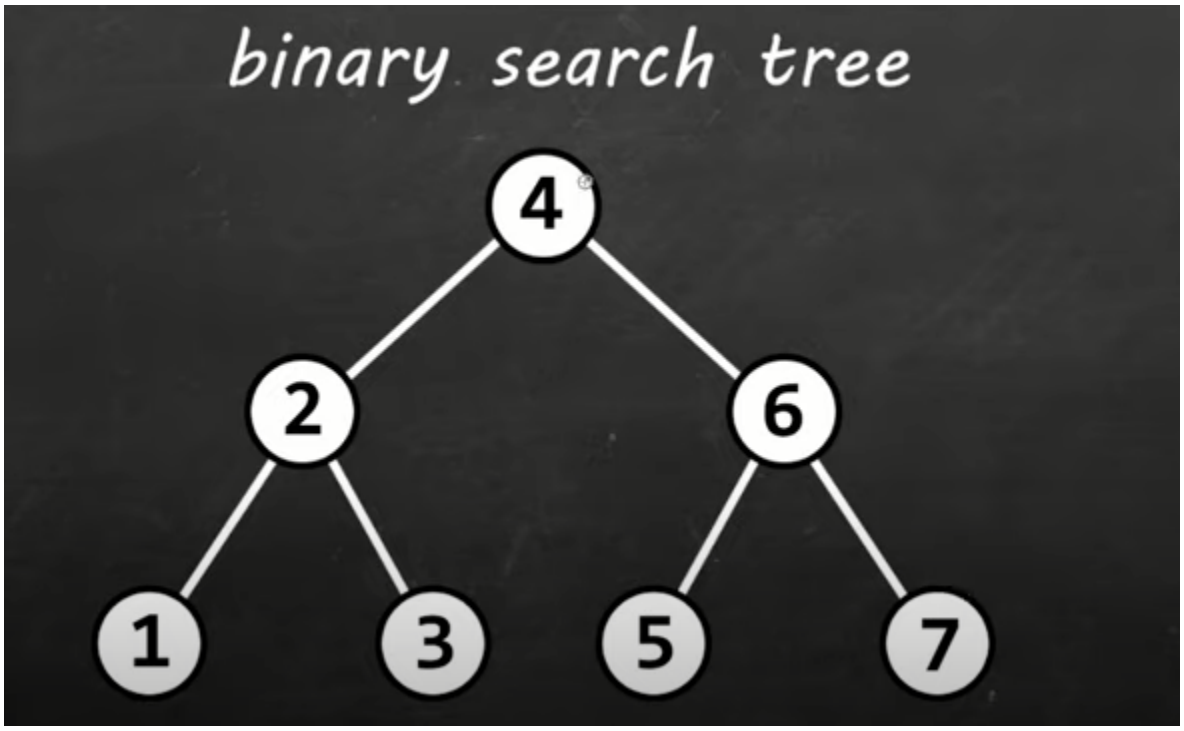

0
A
1
B
C
2
D
E
F
depth: # of edges below
the root node

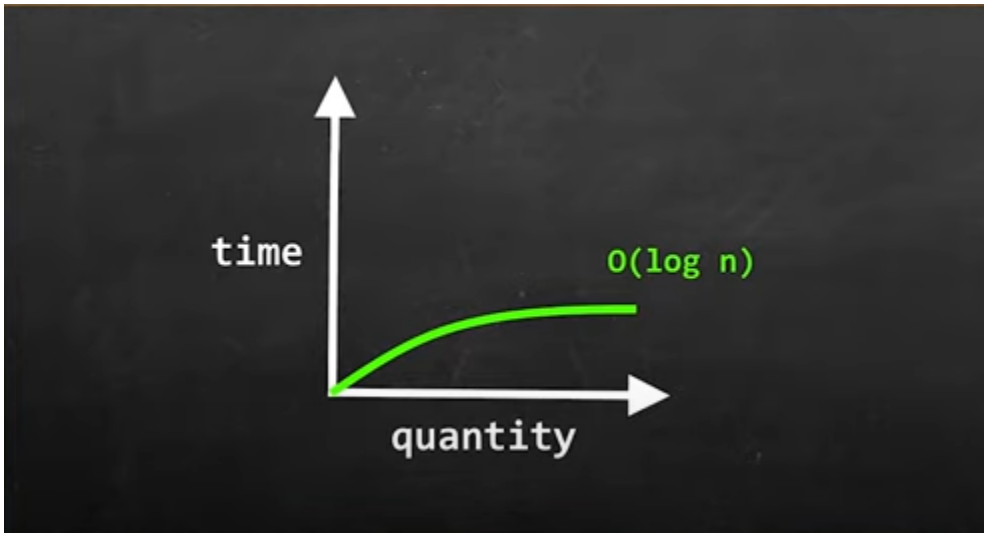




Binary Search Tree

- In a binary search tree a node can contain only two maximum nodes.
- Right childs always contain larger values as compared to the value of the parent node. Left childs always contain lesser values as compared to the value of the parent node.
- Left most child is the least value and the right most child is the greatest value.
- Run time complexity to find(search) a value is O(log n)(Best case) and O(n)(worst case)
- Space complexity O(n)
- The left child of the node i is 2i. The right child of the node i is 2i+1.

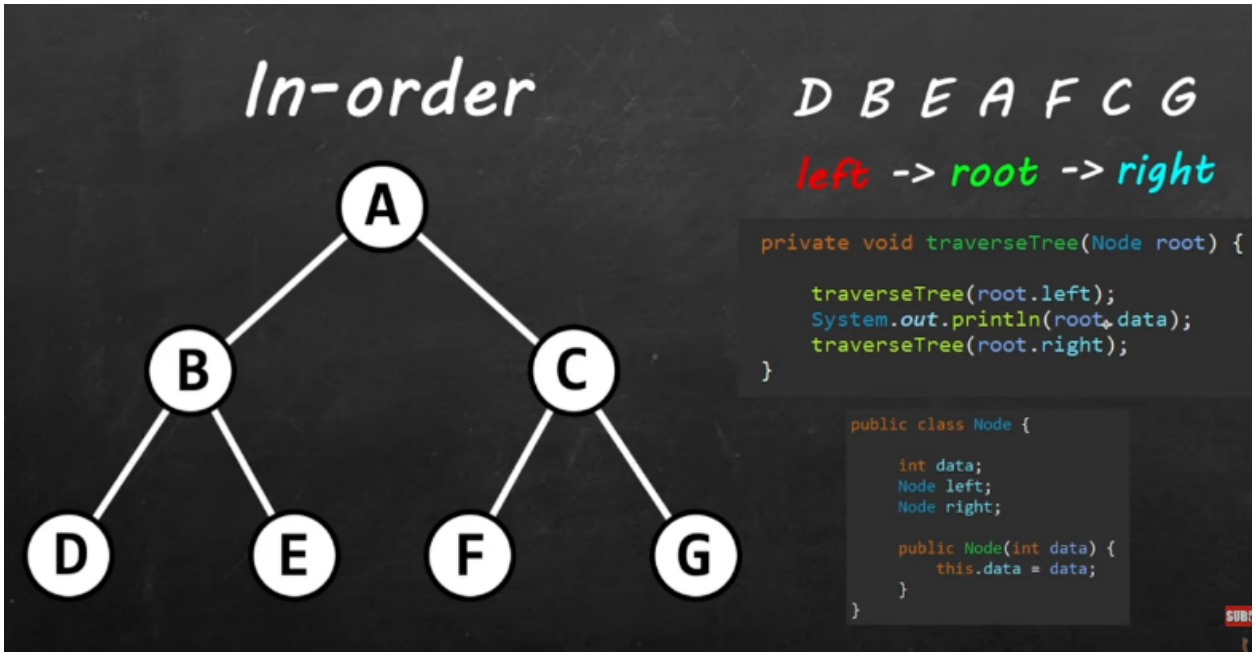


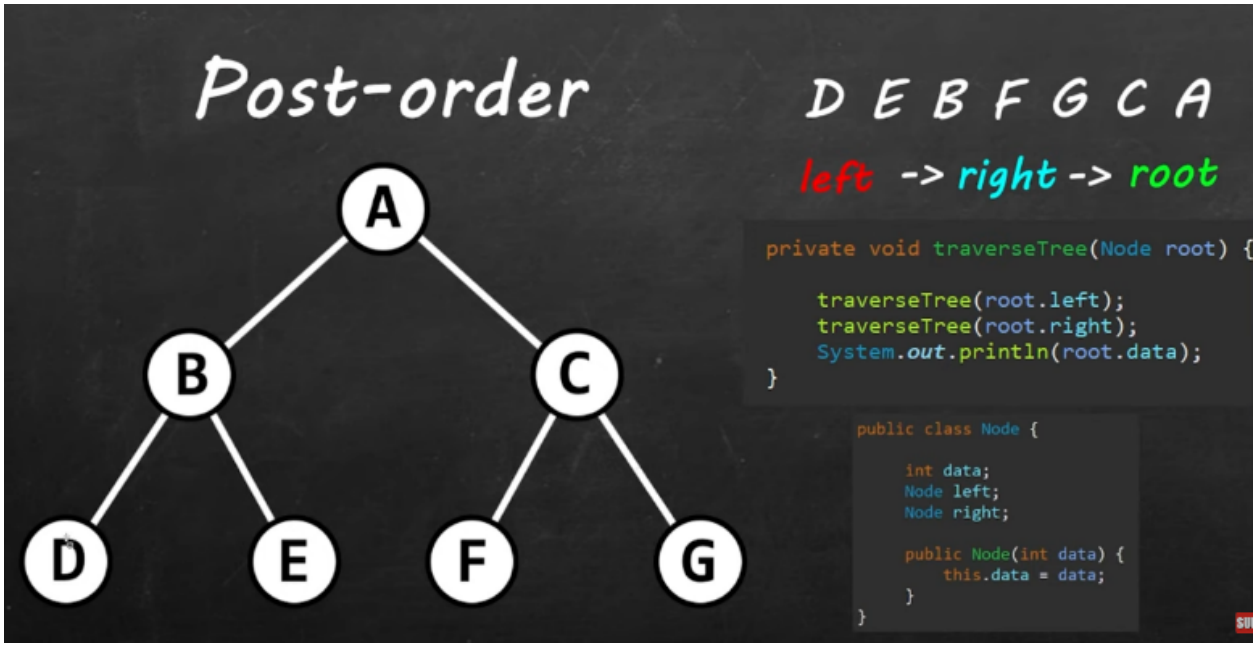


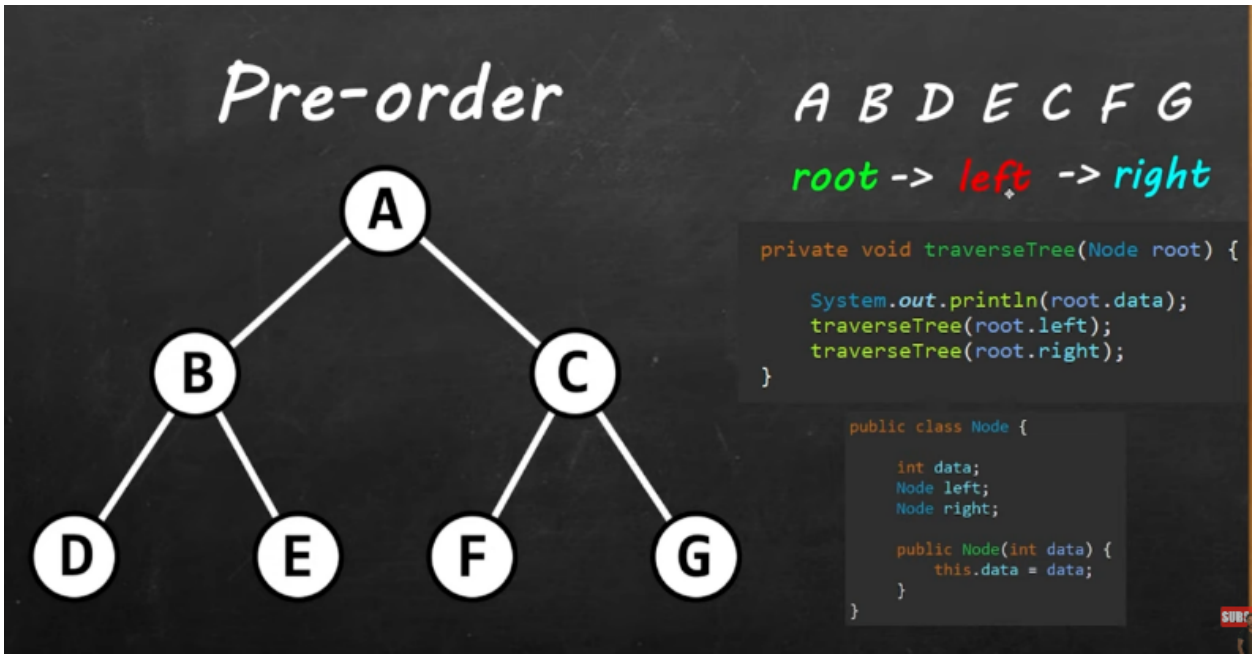
Tree Traversal

There are three types of tree traversals:

1) **In**order traversal (Left, **Root**, Right)
2) **Pre**order traversal (**Root**, Left, Right)
3) **Post**order traversal (Left, Right, **Root**)

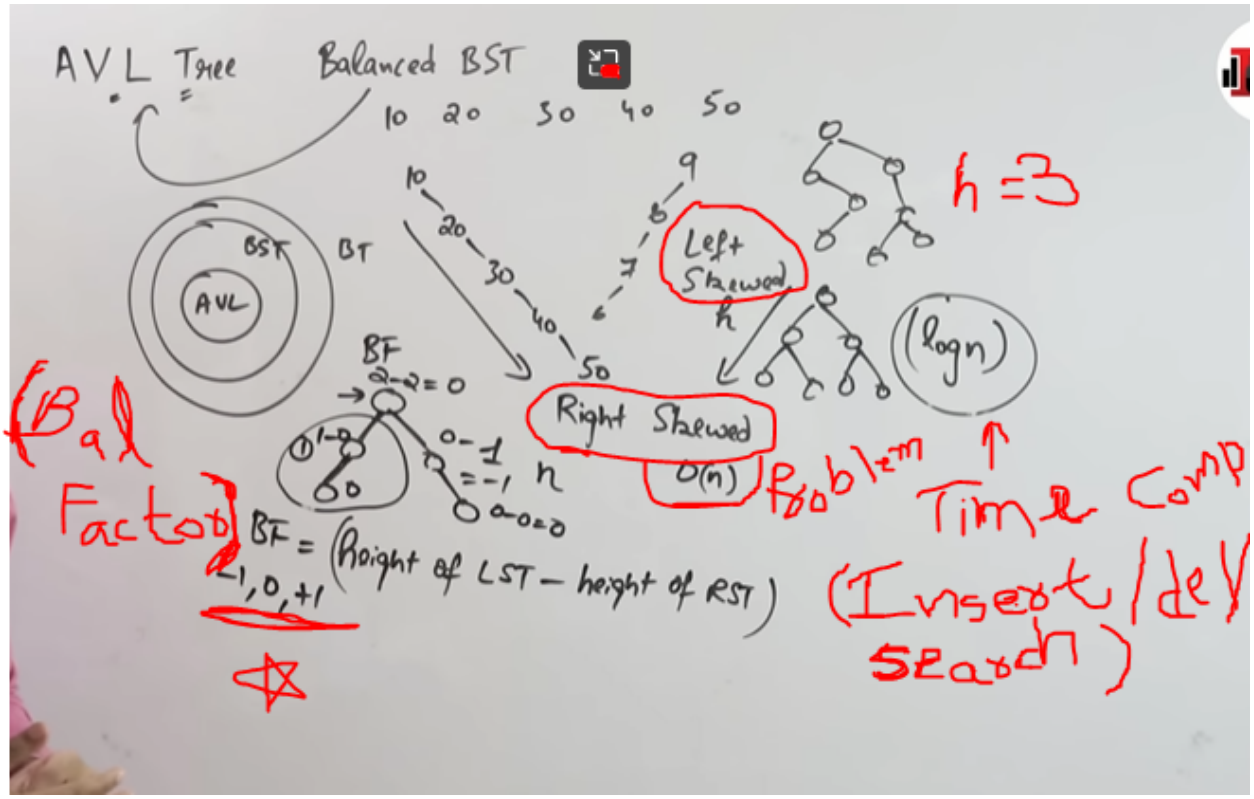

In-order
D B E A F C G
left -> root -> right
A
B
C
D
E
F
G
private void traverseTree(Node root) {
traverseTree(root.left);
System.out.println(root.data);
traverseTree(root.right);
}
public class Node {
int data;
Node left;
Node right;
public Node(int data) {
this.data = data;
}
}


Post-order
D E B F G C A
left -> right -> root
A
B
C
D
E
F
G
private void traverseTree(Node root) {
traverseTree(root.left);
traverseTree(root.right);
System.out.println(root.data);
}
public class Node {
int data;
Node left;
Node right;
public Node(int data) {
this.data = data;
}
}



AVL Tree

Check out from: AVL Tree



Red black tree

Red Black Tree(All Cases With Example) In Hindi

Big(O), Big (Ω) and Big (Θ)