Notes on CS fundamentals

Sandesh Paudel

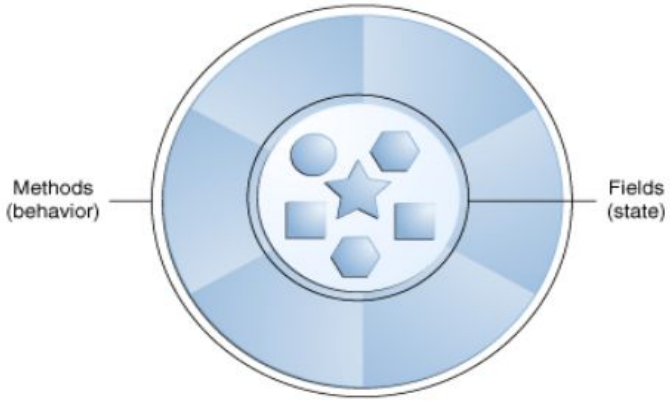
Outline:

1) BASICS
   - ❖ Principles of Object Oriented Programming
   - ❖ Big Oh Notation
   - ❖ Data Structures
     - ➢ Arrays and Strings
     - ➢ Linked Lists
     - ➢ Stacks and Queues
     - ➢ Hashing
     - ➢ Trees and Graphs
   - ❖ Algorithms
     - ➢ QuickSort
     - ➢ MergeSort
     - ➢ BubbleSort
   - ❖ Recursion
   - ❖ Search
     - ➢ Breadth First Search
     - ➢ Depth First Search
     - ➢ Dijkstra's Algorithm

Principles of Object Oriented Programming

❖ Objects

Objects are abstract models that have a **state** and a **behavior.**



By attributing **state** (current speed, current gear) and **providing methods for changing that state,** the object can be in control of how it should be used.

- the code for the object can be written and maintained **independently** of other code.
- the details of the code for the object is **hidden** from outside world.
- the code can be **reused and implemented** as its own **separate component.**

- **Constructors: constructs an instance of the class as an object**
  [invoked as: **Human tom = new Human("M");]**
  Private- only the class that declared it can see it.
  Protected- subclasses and the original class can see it.
  Public- everybody can see it.

```
public class Human {

    private String gender;
    private int age;

    public Human() {
        gender = "male";
    }

    public Human(String g) {
        gender = g;
    }
```

-

❖ Class

a class is a **blueprint** from which individual objects are created / **instantiated.**

Class/Blueprint:

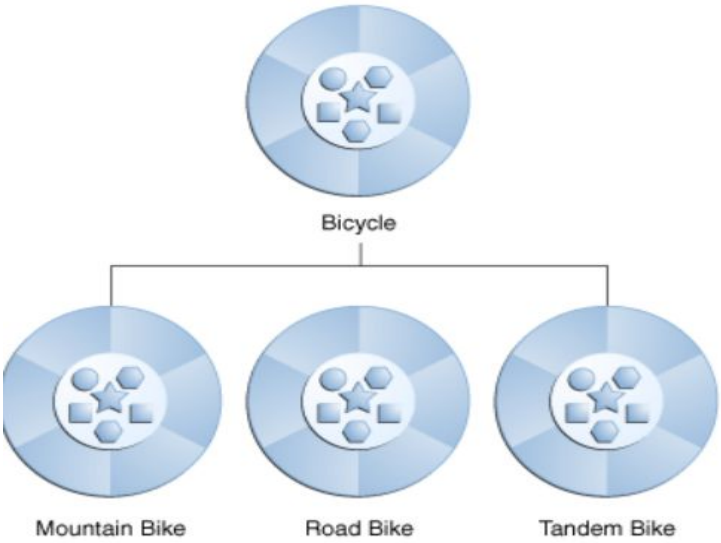
```
class Bicycle {

    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
         cadence = newValue;
    }
```

Instantiation:

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

        // Create two different
        // Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on
        // those objects
        bike1.changeCadence(50);
```

❖ Inheritance

Allows for a hierarchy of classes and subclasses. Subclasses **extend** the superclass. Subclasses **inherit the fields and the methods** of the superclass. But you can code to focus exclusively on the subclass to make it unique.




```
class MountainBike extends Bicycle {

    // new fields and methods defining
    // a mountain bike would go here

}
```

❖ **Interface**

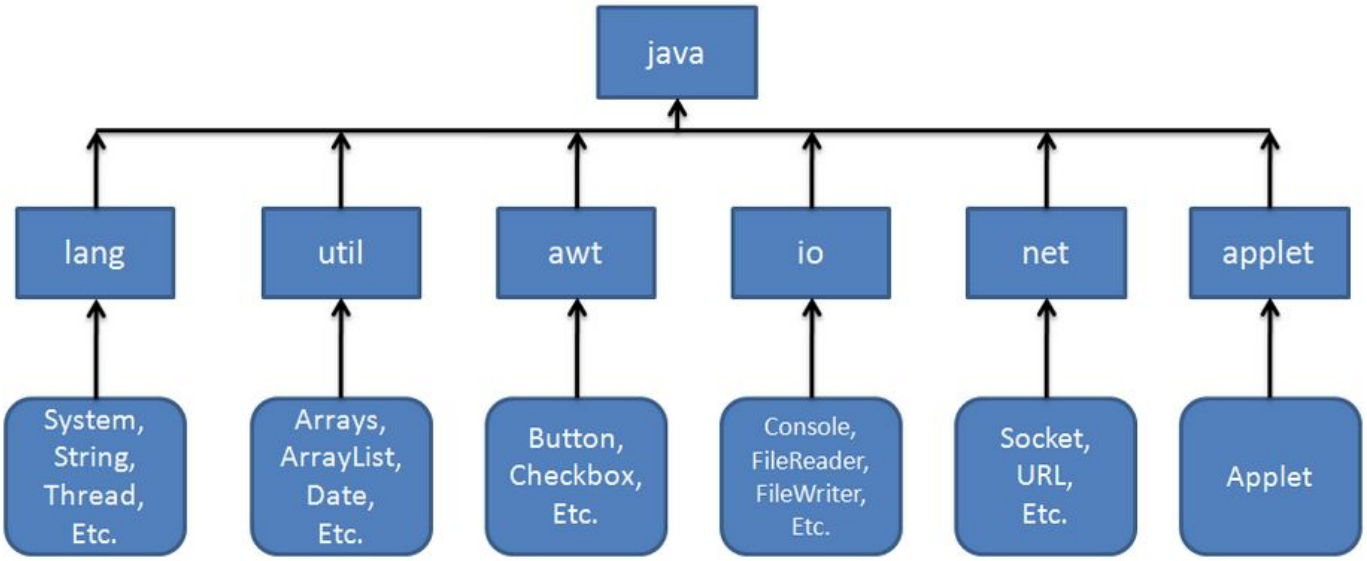
A group of related methods with **empty bodies/no implementation.** Classes that implement interfaces **implement all methods defined by the interface.** It allows classes to be more formal about the behavior it promises to provide. In a way, it **separates behaviors/actions from classes.** Often named with adjectives like drivable/playable.

```
interface Bicycle {

    //  wheel revolutions per minute
    void changeCadence(int newValue);

    void changeGear(int newValue);

    void speedUp(int increment);

    void applyBrakes(int decrement);
}
```

```
class ACMEBicycle implements Bicycle {

    int cadence = 0;
    int speed = 0;
    int gear = 1;

   // The compiler will now require that methods
   // changeCadence, changeGear, speedUp, and applyBrakes
   // all be implemented. Compilation will fail if those
   // methods are missing from this class.

    void changeCadence(int newValue) {
         cadence = newValue;
    }

    void changeGear(int newValue) {
         gear = newValue;
    }
```

❖ **Packages** - a set of related classes and interfaces. Java provides an enormous class library (set of packages) that can be used by the programmer (Application Programming Interface).

```
import java.io.File;
```



❖ **Access Modifiers**

- **public:** visible to all classes.
- **private:** visible only to the class that they belong to.
- **protected:** visible only to the class they belong to, and any subclasses.
- **static:** variables or methods that belong to the class and not the object instance. A single copy of static variable is shared by all instances.

❖ **OOP Features**

- **Encapsulation:** hide internal representations and unnecessary details from the outside world. [using **private/protected** modifiers, and **getters/setters**]

```
public class Category {
    private int id;
    private String name;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

- **Inheritance:** class hierarchy using classes and subclasses. Subclasses inherit the fields and methods of the parent class.

- **Abstraction:** the process of hiding the implementation details and showing only functionality to the user.

An abstract class contains the **abstract** keyword, and **may or may not contain abstract methods** [methods without a body]. Abstract classes cannot be instantiated, you have to **inherit from another class [extends]** and provide implementations to the abstract methods in it.

```java
public abstract class Employee {
   private String name;
   private String address;
   private int number;

   public abstract double computePay();
   // Remainder of class definition
}
```

```java
/* File name : Salary.java */
public class Salary extends Employee {
   private double salary;   // Annual salary

   public double computePay() {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }
   // Remainder of class definition
}
```

- **Polymorphism:** the ability of an object to take on may forms. [Deer is an Animal, a Vegetarian, a Deer and an Object]

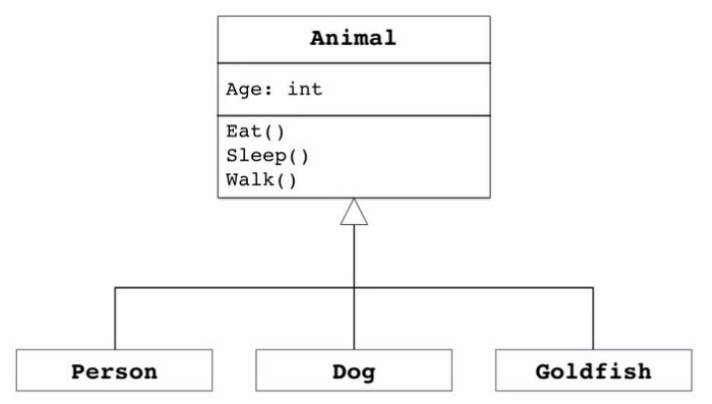
```java
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

```java
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

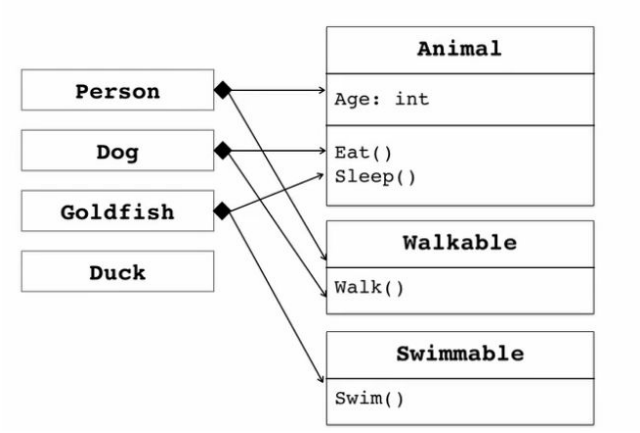
❖ **Ideals of OOP:**

- objects must have **high cohesion** (single responsibility) and **low coupling** (relations) with other objects.
- **Composition [“has a” / “is composed of”]** over **Inheritance [“is a”]**
- The problem with inheritance: inheritance is very rarely clean. It traps you into a certain complicated structure/hierarchy. It comes with **tight coupling**. Changing a parent attribute will change all children classes.

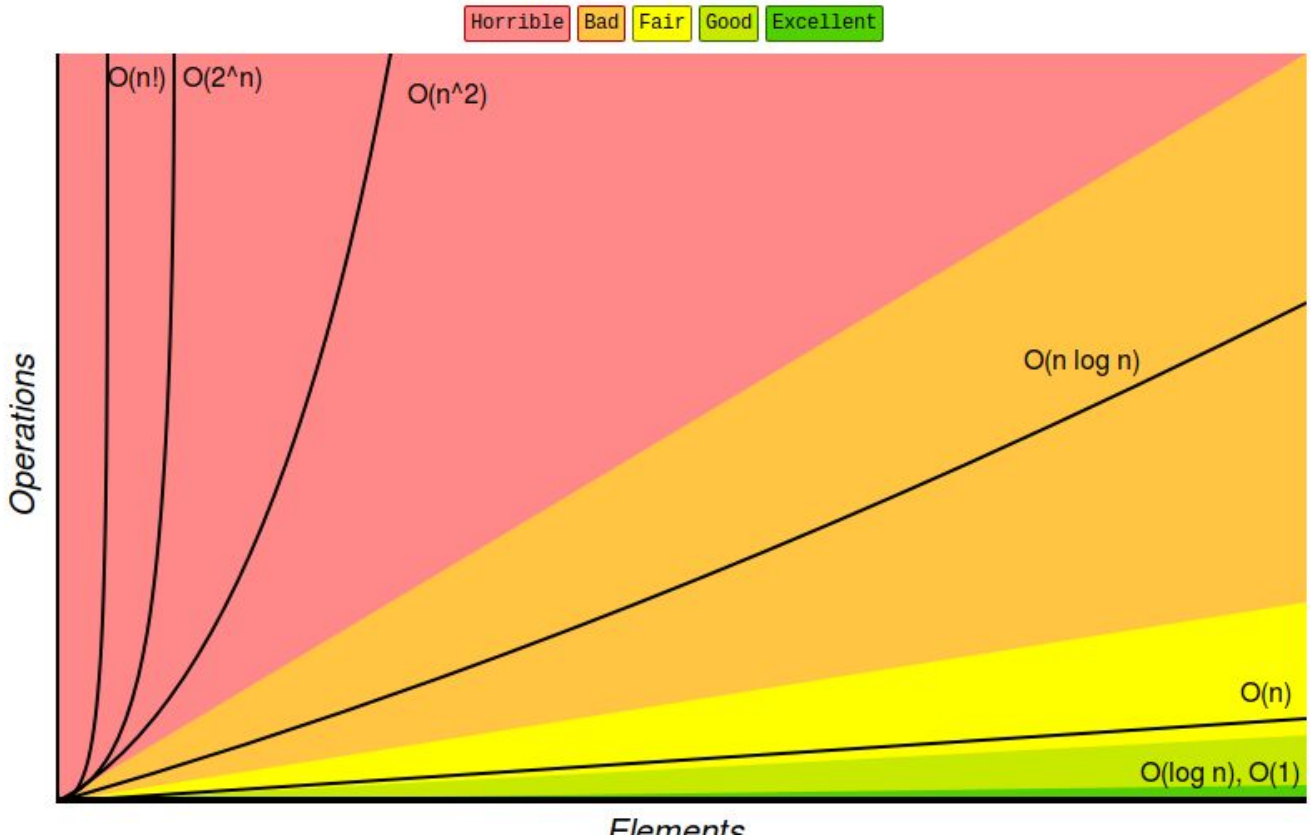
Inheritance:



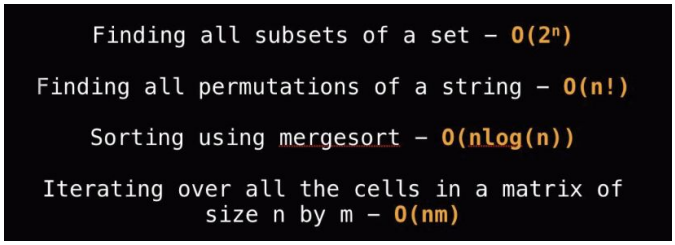
Coupling [with interface/can declare default methods to eliminate repeating code]**:**



Big Oh Notation



Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	O(1)	O(1)	O(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	O(log(n))	O(log(n))	O(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)



> Space complexity (memory cost): If a function A uses M units of its own space (local variables and parameters), and it calls a function B that needs N units of local space, then A overall needs M + N units of temporary workspace. What if A calls B 3 times? When a function finishes, its space can be reused, so if A calls B 3 times, it still only needs M + N units of workspace.

What if A calls itself recursively N times? Then its space can't be reused because every call is still in progress, so it needs O(N^2) units of workspace.

But be careful here. If things are passed by pointer or reference, then space is shared. If A passes a C-style array to B, there is no new space allocated. If A passes a C++ object by reference to B, there is no new space. What if A passes a vector or string by value? Most likely, new space will be allocated. Some C++ compilers try to avoid this for strings, using a technique called copy-on-write, but this is no longer a common thing to do.

> For space complexity, we simply look at the total size of **any new variables we are allocating** (as the input size grows).

O(1) example: (no new variables are declared)

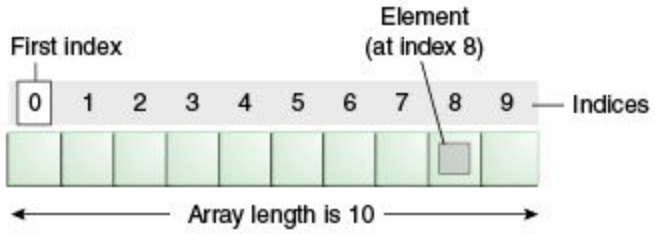
```
public static void sayHiNTimes(int n) {
    for (int i = 0; i < n; i++) {
        System.out.println("hi");
    }
}
```

O(n) example: (the array grows in size linearly to input size)

```
public static String[] arrayOfHiNTimes(int n) {
    String[] hiArray = new String[n];
    for (int i = 0; i < n; i++) {
        hiArray[i] = "hi";
    }
    return hiArray;
}
```

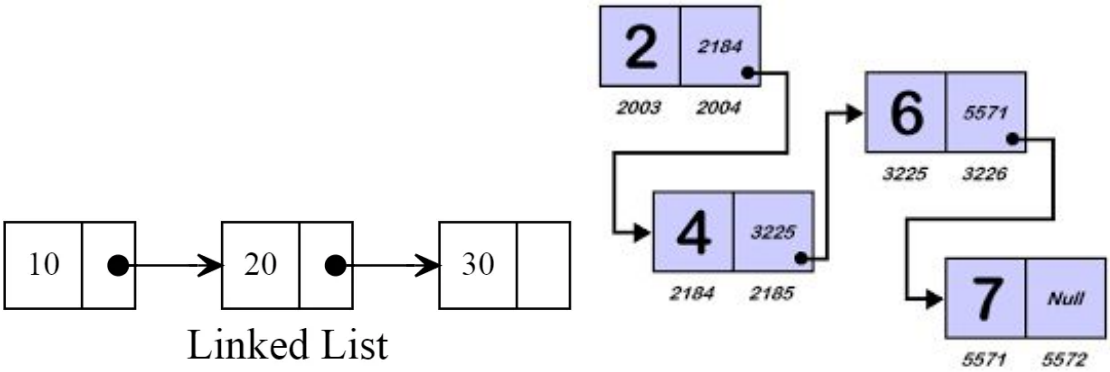
Data Structures

> Arrays

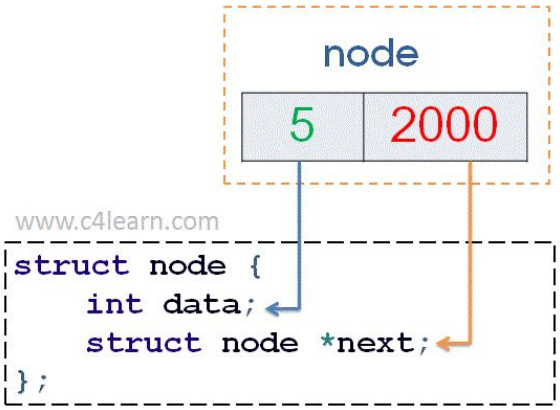


- Instantaneous data access O(1) : a[n]
- Search Iteration O(n)
- Insertion O(n) [high cost - must shift the remaining data to index+1]
- Deletion O(n) [high cost- must shift the remaining data to index-1]

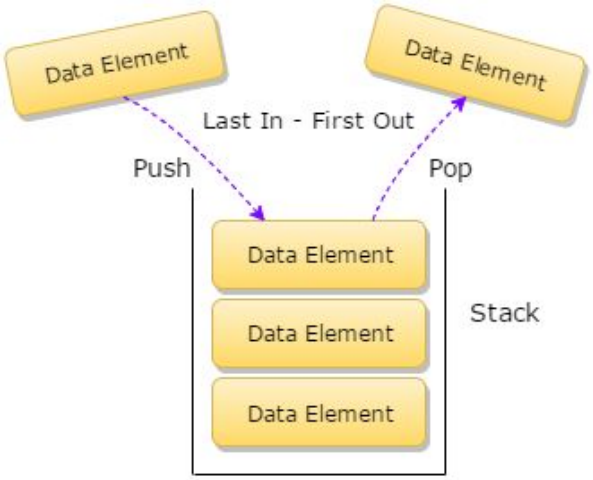
> Lists / Linked lists



- Pair data with pointers, that point to another data location in memory
- Access O(n), must traverse through the list
- Search O(n)
- Insertion O(1), just modify the pointer values to point to the new data
- Deletion O(1)

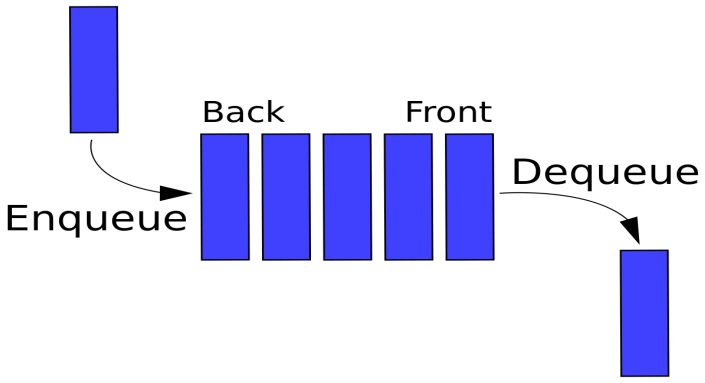


> Stacks (LIFO)



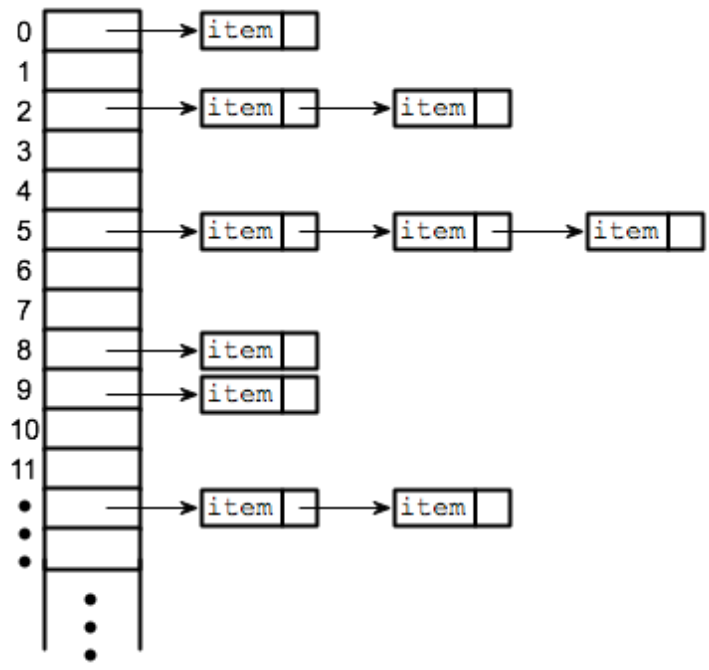
- Access: O(n)
- Search: O(n)
- Insertion/push: O(1)
- Deletion/pop: O(1)

> Queue (FIFO)



- Access: O(n)
- Search: O(n)
- Insertion/Enqueue: O(1)
- Deletion/Dequeue: O(1)

> Hash Tables: obtaining values quickly in an array through keys and the hash function



- Access: O(1 + the size of the chain)
- Search: O(n)
- Insert: O(n)
- Deletion: O(n)

> Data is stored as key:value pairs. Each key is run through the hash function: **key mod memsize = index.**

This hash function outputs unique values that are 0 <= index <= memsize.

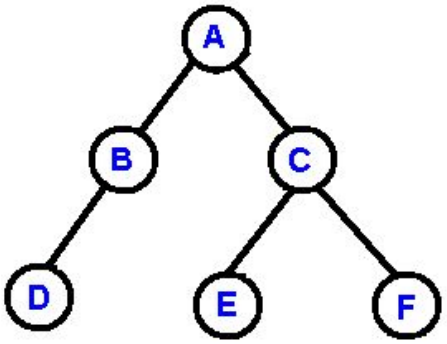
> During insertion if index is already taken, the data is connected through a linked list at the end of the index.

> During access, the index is looked and linear search is performed on the chain.

> **HashMap:** Same idea as Hash Tables, but is faster because it is not synchronized. Use this for single-threaded application. Uses the **Map** interface.

"In many situations, hash tables turn out to be on average more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets."

> **Binary Trees:**

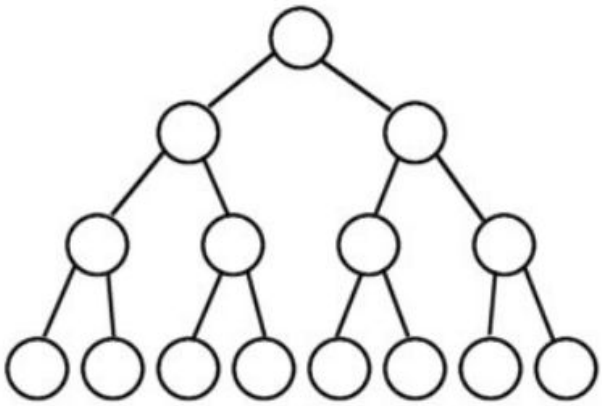


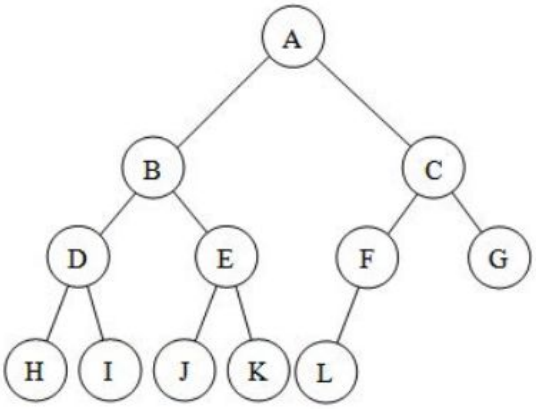

```
struct BstNode {
    int data;
    BstNode* left;
    BstNode* right
};

BstNode* root;

root = NULL;
```

> **Leaf:** a node with no children.
> **Internal Node: != Leaf**

More tree terminology:

- The **depth** of a node is the number of edges from the root to the node.
- The **height** of a node is the number of edges from the node to the deepest leaf.
- A **full binary tree** is a binary tree in which each internal node has exactly zero or two children.
- A **complete binary tree** is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.





> **a complete tree is a very special tree** in that it provides the best ratio between the number of nodes and the height. Proven using the follows: n being the number of nodes and h height of the tree.

$$n = 1 + 2 + 4 + \ldots + 2^{h-1} + 2^h = 2^{h+1} - 1$$

$$h = O(\log n)$$

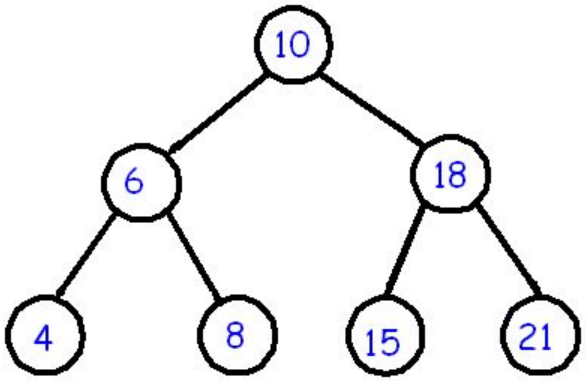
Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move subtrees around with minumum effort

Binary Search Trees

>provides the efficient way of data **sorting, searching** and **retrieving.**



A BST is a binary tree where nodes are ordered in the following way:

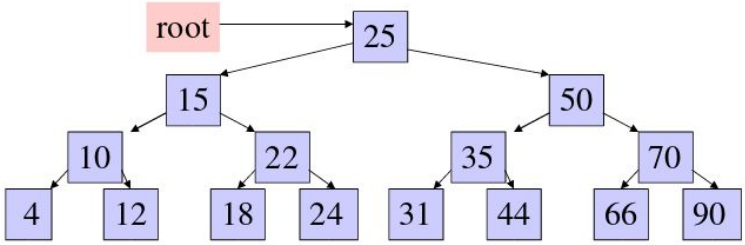
- each node contains one key (also known as data)
- the keys in the left subtree are less then the key in its parent node, in short L < P;
- the keys in the right subtree are greater the key in its parent node, in short P < R;
- duplicate keys are not allowed.

-Depth first Traversals

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



DFS Recursive traversals:

```c
/* Given a binary tree, print its nodes according to th
  "bottom-up" postorder traversal. */
void printPostorder(struct node* node)
{
     if (node == NULL)
        return;

     // first recur on left subtree
     printPostorder(node->left);

     // then recur on right subtree
     printPostorder(node->right);

     // now deal with the node
     printf("%d ", node->data);
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct node* node)
{
     if (node == NULL)
          return;

     /* first recur on left child */
     printInorder(node->left);

     /* then print the data of node */
     printf("%d ", node->data);

     /* now recur on right child */
     printInorder(node->right);
}

/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct node* node)
{
     if (node == NULL)
          return;

     /* first print data of node */
     printf("%d ", node->data);

     /* then recur on left sutree */
     printPreorder(node->left);

     /* now recur on right subtree */
     printPreorder(node->right);
}
```

```java
public void depthFirstSearch(Node root) {
        if(root == null) {
                return;
        }

        Stack<Node> nodeStack = new Stack<Node>();
        nodeStack.push(root);

        while(!nodeStack.isEmpty()) {
                Node node = nodeStack.pop();
                System.out.print(node.data + " ");

                if(node.right != null) {
                        nodeStack.push(node.right);
                }
                if(node.left != null) {
                        nodeStack.push(node.left);
                }
        }
}
```

```java
public void levelOrderQueue(Node root) {
        Queue<Node> q = new LinkedList<Node>();
        if (root == null)
                return;
        q.add(root);
        while (!q.isEmpty()) {
                Node n = (Node) q.remove();
                System.out.print(" " + n.data);
                if (n.left != null)
                        q.add(n.left);
                if (n.right != null)
                        q.add(n.right);
        }
```

> Searching:

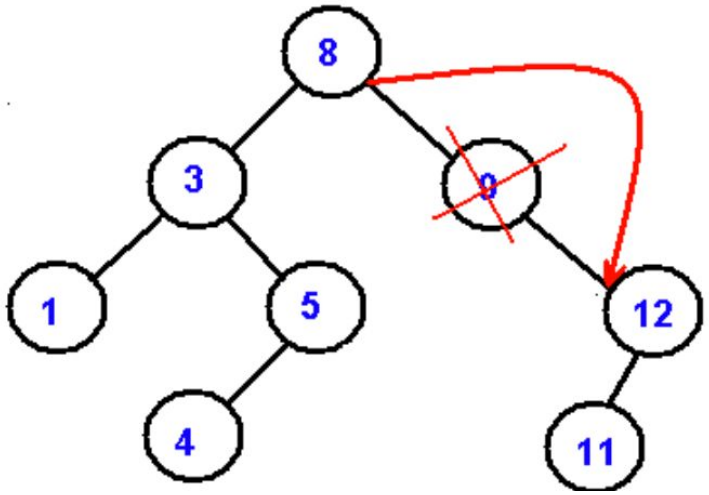
- Start at root. If val > root, go right. If val < root, go left.
- BST has a minimum of O(logn) levels.
- Unfortunately, BST can collapse to linked list so the worst-case complexity is still O(n)

> Insertion:

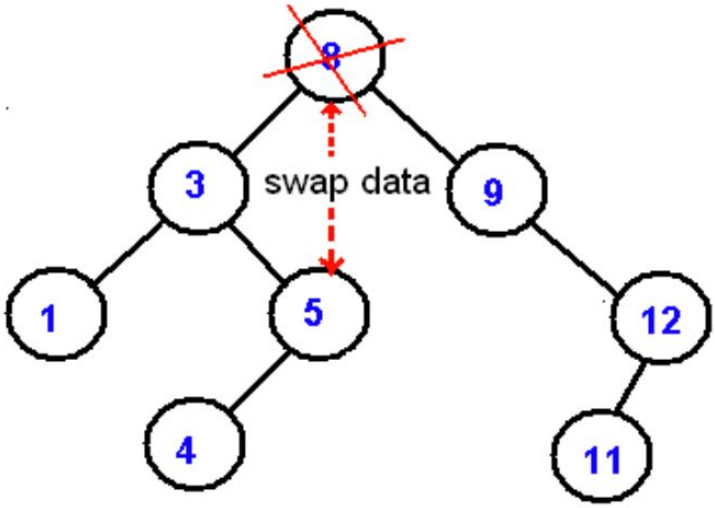
- Similar to searching. Create a new node at the appropriate place.

> Deletion:

-Case 1: Node isn't in the tree, no deletion necessary.
-Case 1.5: Node is a leaf. Just delete it.
-Case 2: Node has One child. Point the parent to the child.



-Case 3: Node has two children. Replace the Node with the biggest node in the left subtree. [pointer goes straight through the node that replaced root [5]]

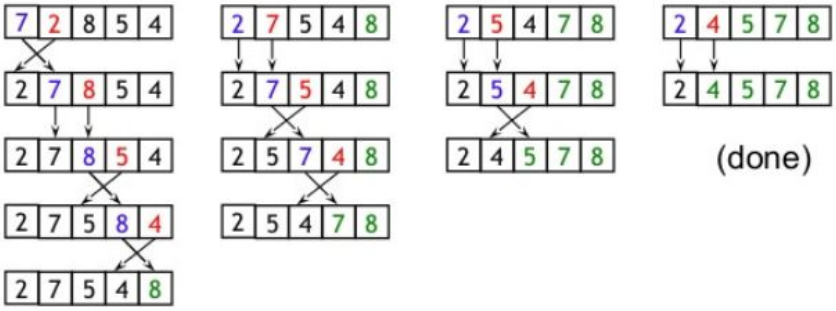


Data Structures Comparison

Data Structures	Average Case			Worst Case		
	Search	Insert	Delete	Search	Insert	Delete
Array	O(n)	N/A	N/A	O(n)	N/A	N/A
Sorted Array	O(log n)	O(n)	O(n)	O(log n)	O(n)	O(n)
Linked List	O(n)	O(1)	O(1)	O(n)	O(1)	O(1)
Doubly Linked List	O(n)	O(1)	O(1)	O(n)	O(1)	O(1)
Stack	O(n)	O(1)	O(1)	O(n)	O(1)	O(1)
Hash table	O(1)	O(1)	O(1)	O(n)	O(n)	O(n)
Binary Search Tree	O(log n)	O(log n)	O(log n)	O(n)	O(n)	O(n)
B-Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Red-Black tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
AVL Tree	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)

Algorithms- Sorting

> Bubble Sort *O(n^2)*

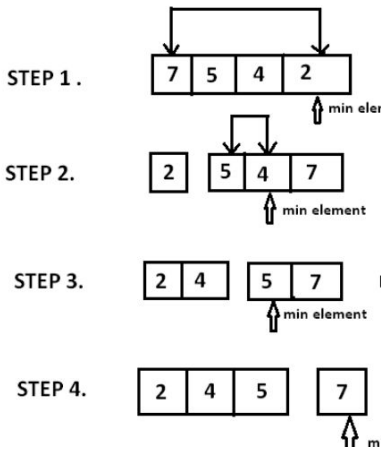



```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

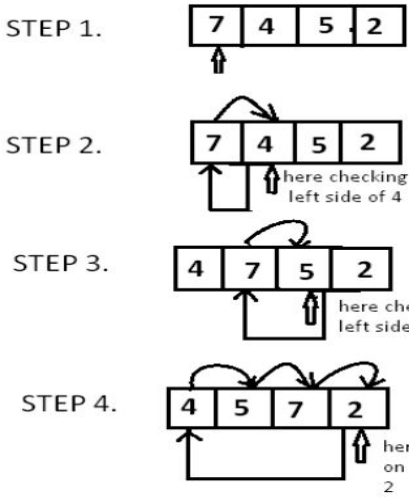
-Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort. Bubble sort can be practical if the input is in mostly sorted order with some out-of-order elements nearly in position.

> Selection Sort *O(n^2)* ~ *outperforms bubble sort*

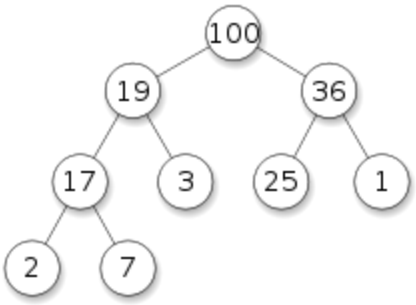
Select a min and swap to the leftmost until the end of the list traversal



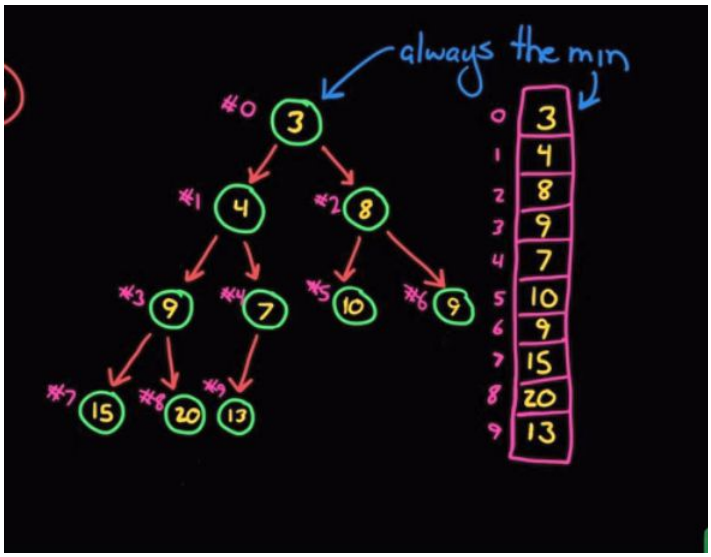
> Insertion Sort ***O(n^2)*** *~ outperforms bubble and selection sorts*
Starts with "already-sorted" LHS and takes a RHS value smaller to LHS and inserts to LHS using linear search so that LHS is sorted

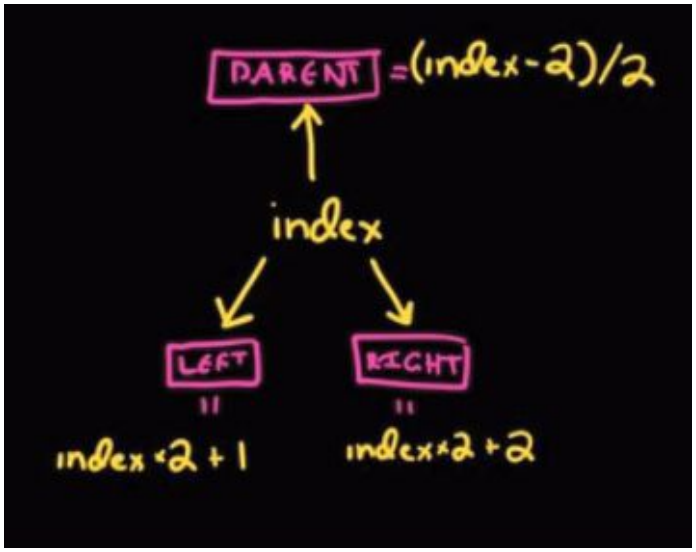


> Heaps: **tree-based** data structure that satisfies the **heap property**: if P is a parent node of C, then the value of P is either greater than or equal to (in a max heap) or less than or equal to (in a min heap) the key of C.



- Used for heap sorting algorithm. Also loosely referred to as priority queues.
- In a heap, the highest (or lowest) priority element is always stored at the root.
- A heap is a useful data structure when you need to remove the object with the highest (or lowest) priority.
- **Heaps are managed in array!!** [Where **parentIndex= (index-1)/2****]
-





- **Insertion:** Add as a leaf node at the bottom of the tree, and **heapify-up** (bubble up)

```
public void heapifyUp() {
    int index = size - 1;
    while (hasParent(index) && parent(index) > items[index]) {
        swap(getParentIndex(index), index);
        index = getParentIndex(index);
    }
}
```

- **Deletion:** remove the root element, take the last leaf node and swap it as the root, and **heapify down.**

```
public void heapifyDown() {
    int index = 0;
    while (hasLeftChild(index)) {
        int smallerChildIndex = getLeftChildIndex(index);
        if (hasRightChild(index) && rightChild(index) < leftChild(index)) {
            smallerChildIndex = getRightChildIndex(index);
        }

        if (items[index] < items[smallerChildIndex]) {
            break;
        } else {
            swap(index, smallerChildIndex);
        }
        index = smallerChildIndex;
    }
}
```

-**Heapsort:** extract the root element, heapify. Repeat until heap is empty.

```
Heapsort (A as array)
    BuildMaxHeap(A)
    for i = n to 1
      swap (A[1], A[i])
      n = n - 1
      Heapify (A, 1)

BuildMaxHeap (A as array)
    n = elements_in(A)
    for i = floor (n/2) to 1
      Heapify (A,i)

Heapify (A as array, i as int)
    left = 2i
    right = 2i+1

    if (left <= n) and (A[left] > A[i])
      max = left
    else
      max = i

    if (right<=n) and (A[right] > A[max])
      max = right

    if (max != i)
      swap (A[i], A[max])
      Heapify (A, max)
```

O(nlogn)

build-max-heap : O(n)

heapify : O(logn), called n–1 times

> **Recursion:**

- Recursive functions may also be executed more slowly (depending on your environment). Each time a function is called, certain values are placed onto the stack - this not only takes time, but can eat away at your resources if your function calls itself many times. In extreme examples you could run out of stack space. Functions that just iterate make no such demands on stack space, and may be more efficient where memory is limited.

- **Factorial:**

```
    1. if n is 0, return 1
    2. otherwise, return [ n × factorial(n-1) ]

end factorial
```

- **Tower of Hanoi:**

```
    1. if n is 1 then return 1

    2. return [2 * [call hanoi(n-1)] + 1]

end hanoi
```

Practice Algorithms

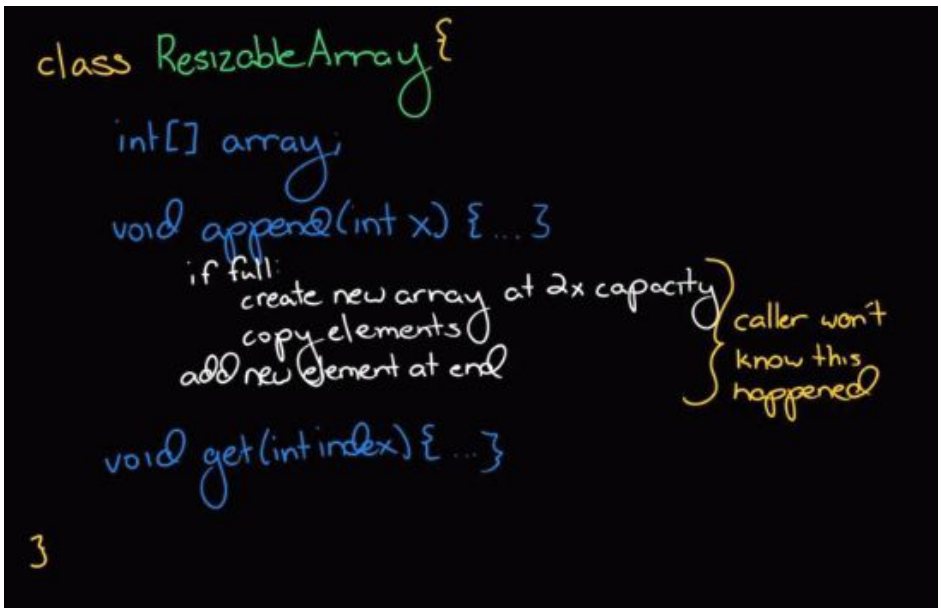
1) Left Rotation: 12345, 2 -> 34512

- N is the size of the array and K is the number of rotations

```
for(int j=0; j<k; j++){

    //one left shift operation
    int initial = a[0];
    for(int i = 1; i < n; i++){
        a[i-1] = a[i];
    }

    a[n-1] = initial;
}
```

```
#!/usr/bin/env/python

def leftrotation():

    stringa = input("Enter the string to rotate");
    numrotate = int(input("Enter a number to rotate"));

    chararray = list(stringa);
    chararraySize = len(chararray);

    outputlist = [];


    for x in range(numrotate, chararraySize):
        outputlist.append(chararray[x]);

    for y in range(0, numrotate):
        outputlist.append(chararray[y]);

    #joining a list in python
    returnString = ''.join(outputlist);

    print(returnString);

leftrotation();
```

2) Dynamic Array List



3) Algorithm for anagram checker, Java:

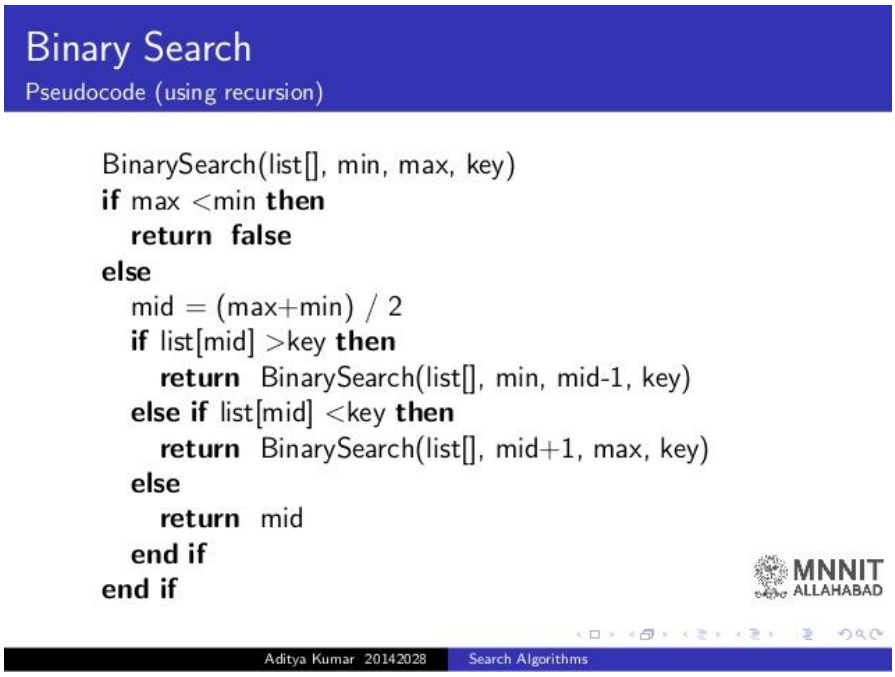
```
    char[] achars = a.toCharArray();
    char[] bchars = b.toCharArray();
    int match = 0;
    //iterate through achars
    for(int i = 0; i < achars.length; i++){
        char current = achars[i]; //current at string1

        //now iterate bchar array to look for current
        for( int j = 0; j < bchars.length; j++){
            if(bchars[j]==current){
                match++;
                bchars[j] = '/';
                break;
            }
    }
    }
    //-------
    //++++
    //main formula for the number of deletions
    int deletions = (achars.length + bchars.length) - 2*match;
```

Python:

```
#!/usr/bin/env python

#anagram checker in python


def anagramchecker():


    stringa = input('Welcome to the Python anagram checker! Enter a string \n');
    stringb = input('Now enter another string! \n');

    print('String 1: ', stringa);
    print('String 2: ', stringb);

    #string to char conversion in python is really easy
    chara = list(stringa);
    charb = list(stringb);
    charasize = len(chara);
    charbsize = len(charb);

    match = 0;
    #nested forloops here we go!!
    #iterating through achars
    for charina in chara:

        for charinb in charb:

            if(charina == charinb):
                match = match+1;
                charinb = '/';
                break;

    result = (charasize+charbsize) - 2*match;

    print('You need to delete', result, " characters for the strings to be an anagram.")



anagramchecker();
```

4) Binary Search Example O(log n)

-Recursive:



Iterative:

```
public static boolean binarySearchIterative(int[] array, int x) {
    int left = 0;
    int right = array.length - 1;
    while (left <= right) {
        int mid = left + ((right - left) / 2);
        if (array[mid] == x) {
            return true;
        } else if (x < array[mid]) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    return false;
```

5) Example of O(nlogn) algorithm:

```
int n = 100
for(int i = 0; i < n; i++) //this loop is executed n times, so O(n)
{
    for(int j = n; j > 0; j/=2) //this loop is executed O(log n) times
    {

    }
}
```