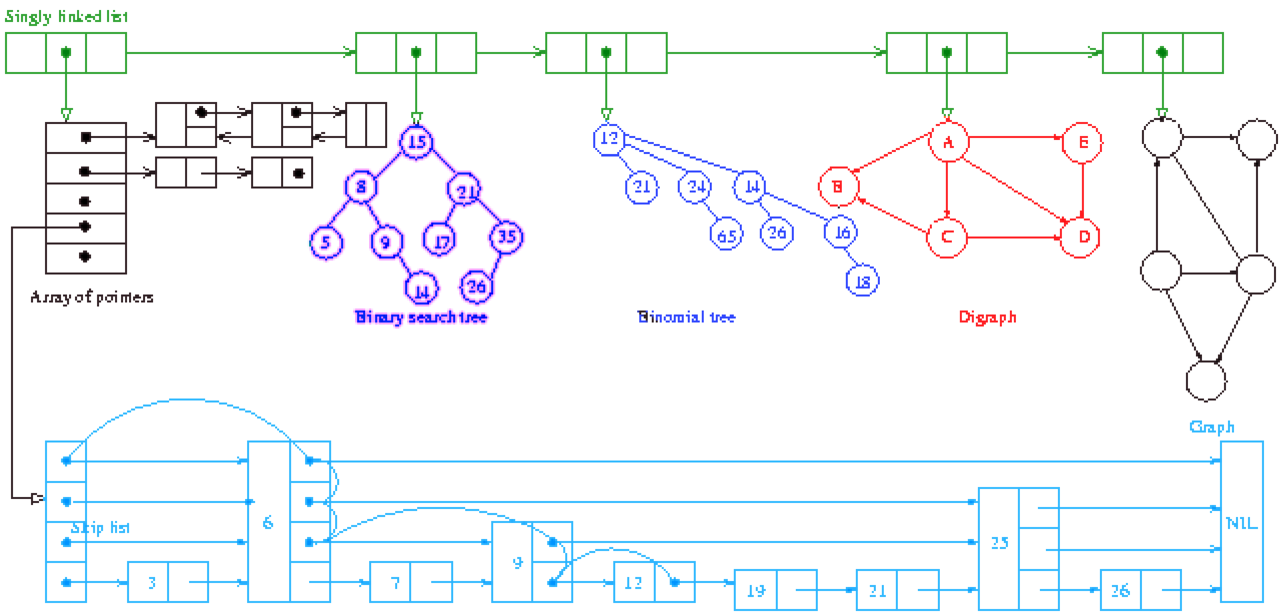
ESINF – Estruturas de Informação



Emanuel Cunha Silva
ecs@isep.ipp.pt

"Recursion is a technique by which a method makes one or more calls to itself during execution,…" [Goodrich, et al.2014]

- The Russian Matryoshka dolls is a physical example of recursion;

- The *factorial function* (commonly denoted as *n*!) has a recursive definition.

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

Consider the following code:

```
public static void process (int a[], int liminf, int limsup) {
        int i=liminf ;
        int j=limsup-1 ;
        while (i < j) {
                int temp=a[i] ;
                a[i]=a[j] ;
                a[j]=temp ;
                i++ ; j-- ;
        }
}
public static void example (int[] a, int li, int ls) {
        if (li < ls) {
                process (a,li,ls);
                ls=ls/2;
                example (a,li,ls);
        }
}
```

a[8]={ 6,1,4,2,7,3,1,5}, li=0, ls=8

example (a, 0, 8)

6	1	4	2	7	3	1	5
li							ls

5	1	3	7	2	4	1	6
li				ls			

7	3	1	5	2	4	1	6
li		ls					

3	7	1	5	2	4	1	6
li	ls						

Calculate the sum of two positive integers numbers

Iterative method

```
public int soma (int x, int y){
  int res=x;
  while( y != 0){
    res += 1;
    y--;
  }
  return(res);
}
```

Recursive method

```
public int soma (int x, int y){
   if(y==0)
      return x;
   else
      return (1 + soma(x, y-1));
}
```

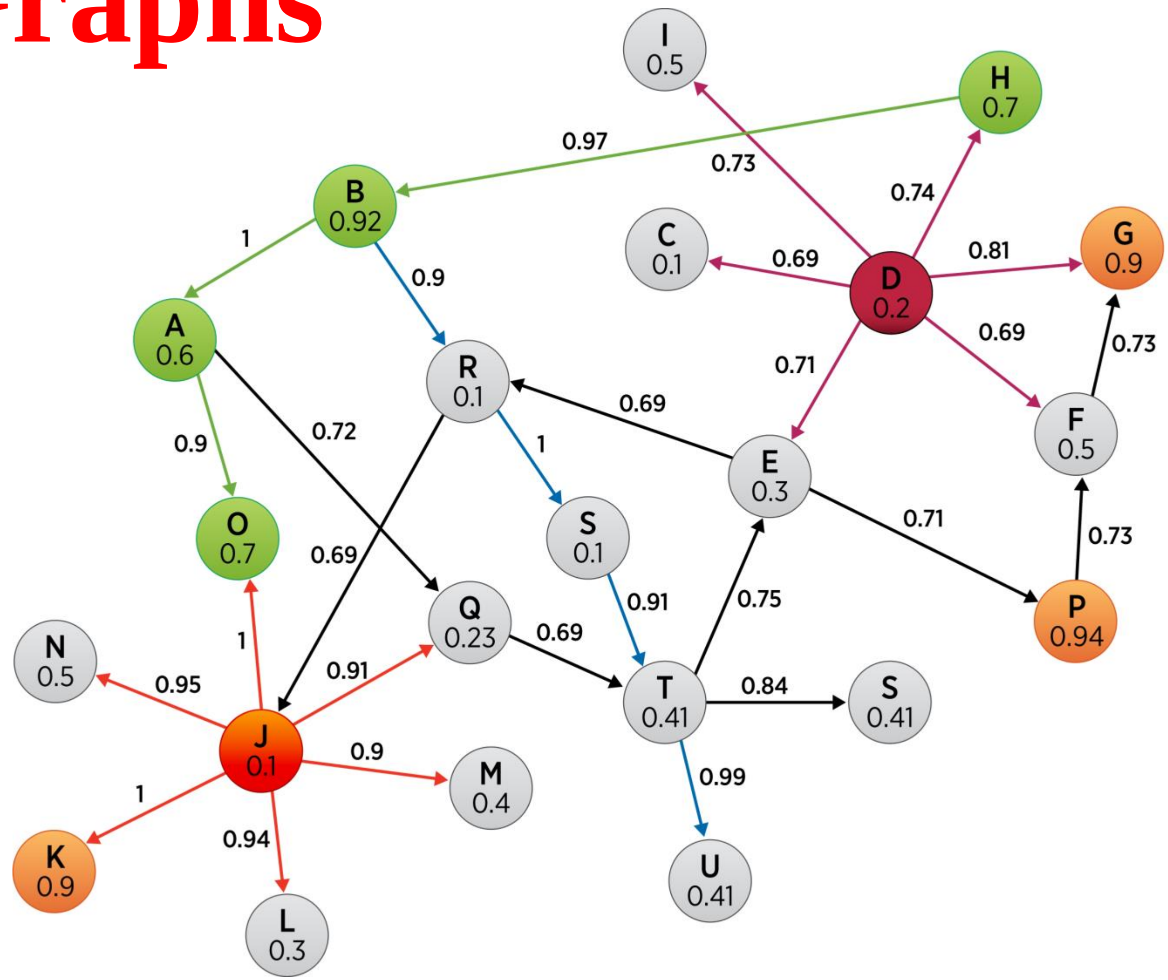
Print a decimal integer *n* to its binary representation

```
public static void dec_2_bin (int valor)
{
int quociente, resto;
    if(valor != 0){
        quociente = valor/2;
        resto = valor - quociente*2;
        dec_2_bin (quociente);
        System.out.print(resto);
    }
}
```

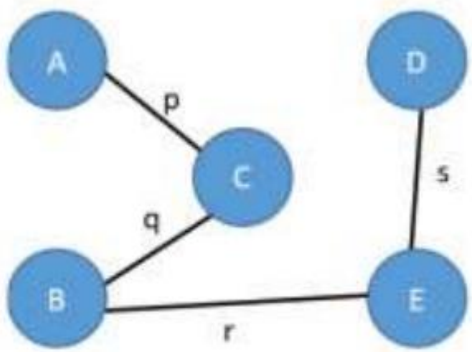
Verify if a positive integer is prime

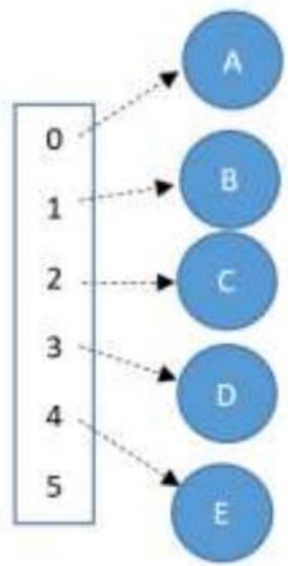
```
public boolean isPrime (int num)
{
    return prime (num, 2);
}

public boolean prime (int num, int i)
{
     if (i > Math.sqrt(num))
          return true;
     if (num % i == 0)
          return false;

     return prime (num, i+1);
}
```

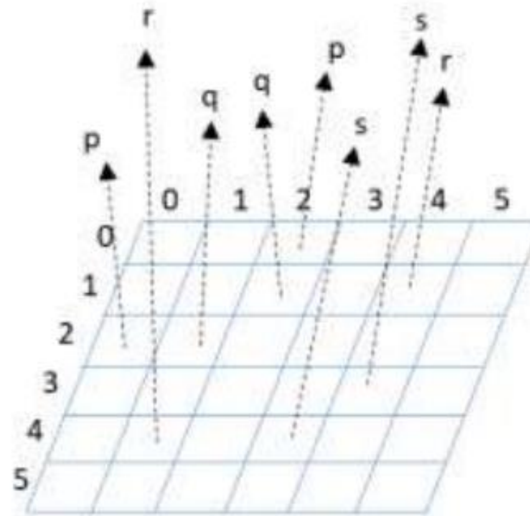
Graphs


I
0.5
H
0.7
0.97
0.73
B
0.92
0.74
C
0.1
G
0.9
1
0.69
D
0.2
0.81
0.9
A
0.6
0.69
0.73
R
0.1
0.71
0.72
0.69
F
0.5
0.9
1
E
0.3
O
0.7
S
0.1
0.71
0.73
0.69
0.91
0.75
Q
0.23
P
0.94
N
0.5
1
0.69
0.95
0.91
T
0.41
0.84
S
0.41
J
0.1
0.9
M
0.4
1
0.99
0.94
K
0.9
L
0.3
U
0.41

Graphs: Adjacency Matrix Structure







undirected Graph **Vertices' ArrayList** **Edges' Matrix**

Graph Traversal Techniques

- There are two standard graph traversal techniques that provide an efficient way to “visit” each vertex and edge exactly once:

  - **Breadth-First Search (BFS)**
  - **Depth-First Search (DFS)**

- Starts at some source vertex ***S***

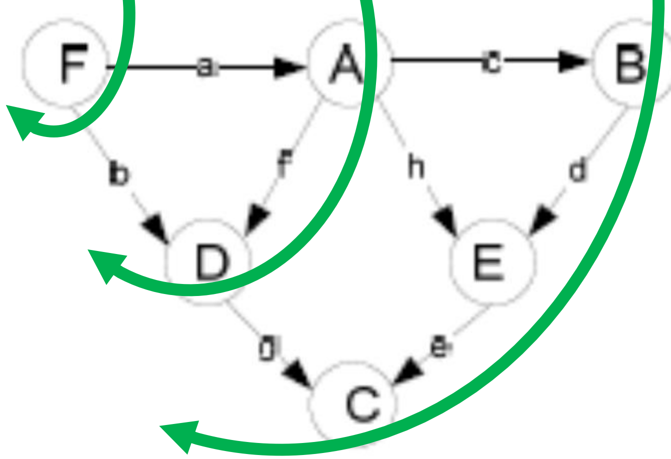
- Find all vertex that are reachable from initial ***S***

BFS (Breadth-First Search)

Steps:

- For a given vertex (origin) cycle through all its adjacent vertices.
- if they have not yet been visited, they are placed in an auxiliary queue.
- The node that is at the beginning of the queue is removed and becomes the next source node.
- the adjacent nodes (of the node origin) are inserted into the queue, if they have not yet been visited.

The BFS processes the vertices by levels, starting with the closest vertices of the initial vertex.

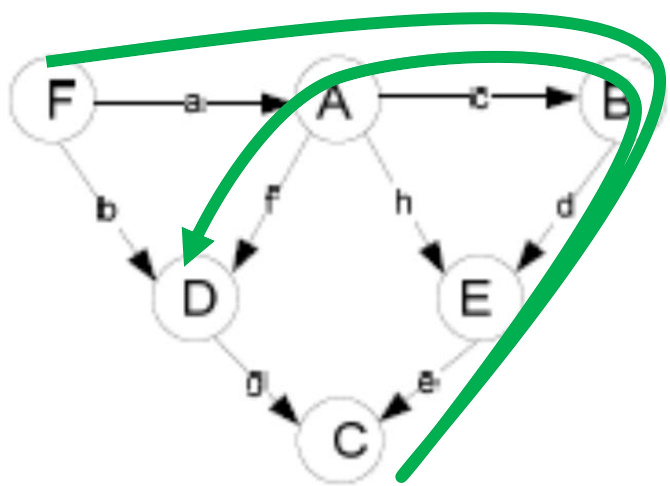


BFS: **F A D B E C**

DFS (Depth-First Search)

Steps:

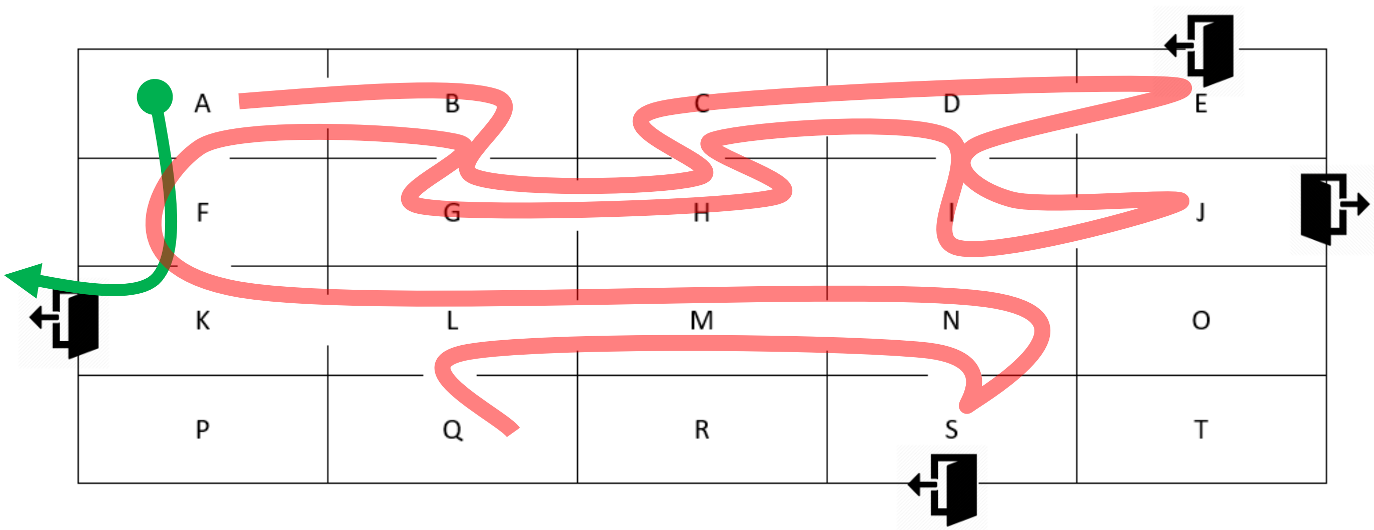
- For a given vertex (origin) cycle through all its adjacent vertices.
- move to an adjacent node, if it was not yet visited
- from the latter, which becomes the origin, move to its adjacent unvisited node, and so on, traversing a path.
- when you can not move on, returns to the last node considered (backtraking) that still has adjacent nodes to visit
- moves to the next adjacent and so on



DBS: **F A B E C D**

LabyrinthCheater class

Implement a LabyrinthCheater class which is intended to represent a labyrinth through a map of **rooms** and **doors**. Each room has a name and potentially provides an exit. Doors represent connections between rooms.



Rooms which are reachable from A: **A, B, G, H, C, D, E, I, J, F, K, L, M, N, S, Q** (DFS)

Nearest exit from A: **K**

Path to nearest exit from A: **A, F, K**

What will be the sequence for BFS? **A, B, F, G, K, H, L, C, M, Q, D, N, E, I, S, J**

Graphs: Adjacency Matrix Structure

Not directed Graph (bidirectional)

```
public class LabyrinthCheater {

        private class Room{
        }

        private class Door{
        }

        AdjacencyMatrixGraph<Room, Door> map;

        public LabyrinthCheater(){
                map = new AdjacencyMatrixGraph<Room, Door>();
        }

        public boolean insertRoom(String name, boolean hasExit);
        public boolean insertDoor(String from, String to);
        public Iterable<String> roomsInReach(String room);
        public String nearestExit(String room);
        public LinkedList<String> pathToExit(String from);
}
```

```
public class LabyrinthCheater {

        private class Room{
                public String name;
                public boolean hasExit;
                public Room(String n, boolean exit){
                        name = n;
                        hasExit = exit;
                }

                /*
                 * Implementation of equals
                 * Comparison of rooms is by name only
                 */

                public boolean equals(Object other){
                        if(!(other instanceof Room)) return false;
                        return name.equals(((Room)other).name);
                }
        }
}
```

Graphs: Adjacency Matrix Structure

```
public class LabyrinthCheater {

public boolean insertRoom(String name, boolean hasExit){
        return map.insertVertex(new Room(name, hasExit));
}

public boolean insertDoor(String from, String to){
        return map.insertEdge(new Room(from,false),
                              new Room(to, false),
                              new Door());
}

}
```

Inserts in the list of vertexes

Inserts an edge, fills positions in the adjacency matrix

Graphs: Adjacency Matrix Structure

```
public class LabyrinthCheater {

public Iterable<String> roomsInReach(String room){
  if(!map.checkVertex(new Room(room, false)))
             return null;

  LinkedList<Room> rooms;
  rooms = GraphAlgorithms.DFS(map, new Room(room, false));

  LinkedList<String> names = new LinkedList<String>();
  for(Room r : rooms)
             names.add(r.name);
  return names;
}

}
```

Graphs: Adjacency Matrix Structure

```
public class LabyrinthCheater {

public String nearestExit(String room){

   if(!map.checkVertex(new Room(room, false)))
               return null;

   LinkedList<Room> rooms;
   rooms = GraphAlgorithms.BFS(map, new Room(room, false));

   for(Room r : rooms)
               if(r.hasExit) return r.name;

   return null;
  }
}
```

Why BFS?

Graphs: Adjacency Matrix Structure

```
public class LabyrinthCheater {

public LinkedList<String> pathToExit(String from){

(1)   if(!map.checkVertex(new Room(from, false))) return null;

(2)   String exitName = nearestExit(from);
      if(exitName == null) return null;

(3)   Room exit = new Room(exitName, true);

(4)   LinkedList<LinkedList<Room>> paths = new LinkedList<LinkedList<Room>>();
      boolean result = GraphAlgorithms.allPaths(map, new Room(from, false), exit, paths);
      if(result == false) return null;
      if(paths.isEmpty()) return null;

(5)   Iterator<LinkedList<Room>> it = paths.listIterator();
      LinkedList<Room> min = it.next();
      while(it.hasNext()){
           LinkedList<Room> cur = it.next();
           if(cur.size() < min.size())
                min = cur;
      }

(6)   LinkedList<String> names = new LinkedList<String>();
      for(Room r : min)
           names.add(r.name);

(7)   return names;
}
}
```

Validate origin

Seek nearest exit

Set the exit room

Calculate all paths

Select path with min size

Build result

Return result

The Map interface

- **MAP**: an unordered collection that associates a collection of element values with a set of keys so that elements they can be found very quickly (O(1))
  - Each key can appear at most once (no duplicate keys)
  - A key maps to at most one value
  - the main operations:
    - **put**(*key*, *value*)
      "Map the *key* to that *value*."
    - **get**(*key*)
      "get the value associated to this *key*."
  - Examples: dictionary, phone book, etc.

The Map interface

```
public interface Map {
   Object put(Object key, Object value);
   Object get(Object key);
   Object remove(Object key);
   boolean containsKey(Object key);
   boolean containsValue(Object value);
   int size();
   boolean isEmpty();

   void putAll(Map map);
   void clear();

   Set keySet();
   Collection values();
}
```

The Map interface

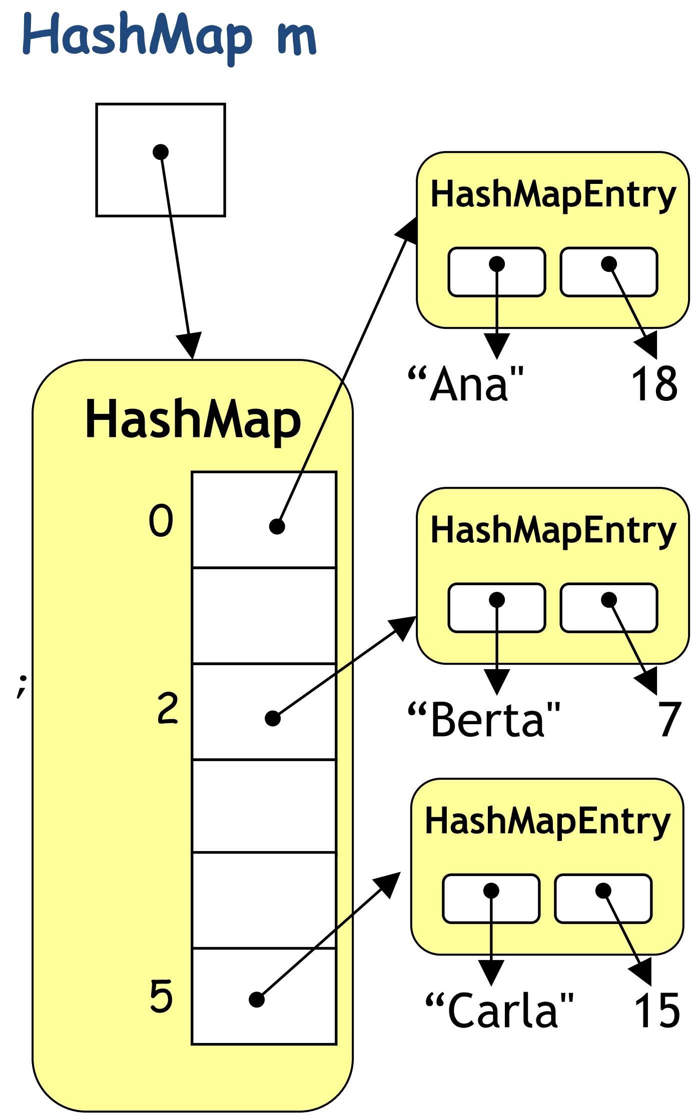
- `Map` is an interface; you can't write `new Map()`
- We will use the `HashMap` implementation

- Preferred:
  - `Map m = new HashMap();`
- Not:
  - `HashMap m = new HashMap();`

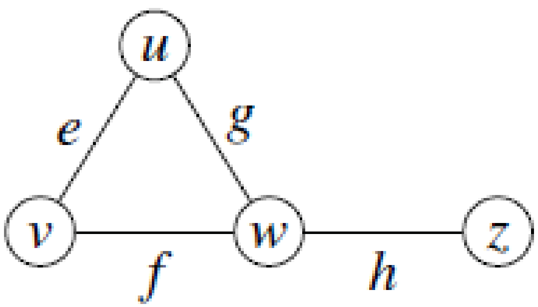
HashMap example

```
Map m = new HashMap();
m.put("Ana", 18);
m.put("Berta", 7);
m.put("Carla", 15);

System.out.println(m.get("Ana"));
System.out.println(m.get("Berta"));

m.put("Ana", 20);
m.remove("Carla");

for (String key : m.keySet()) {
   System.out.println(key +":"+ m.get(key));
}
```

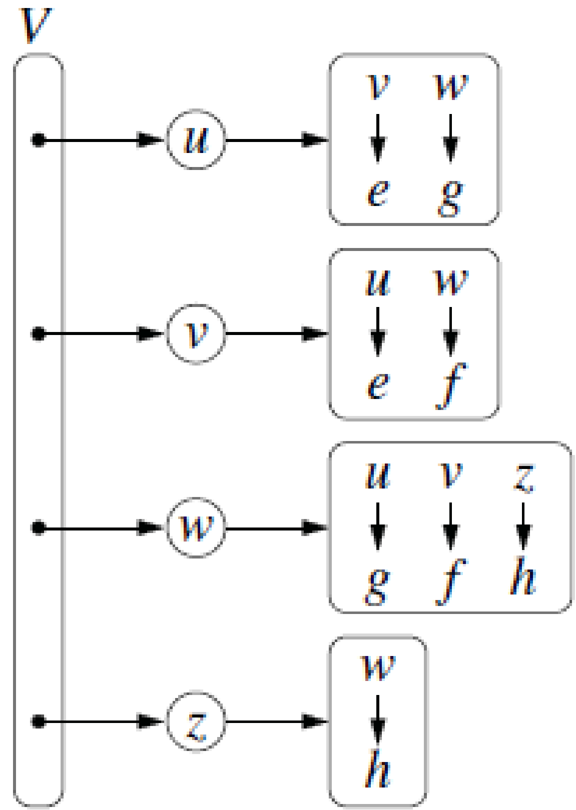
```
Output:
18
7
Ana:20
Berta:7
```



Graphs: Adjacency Map Structure



(a)

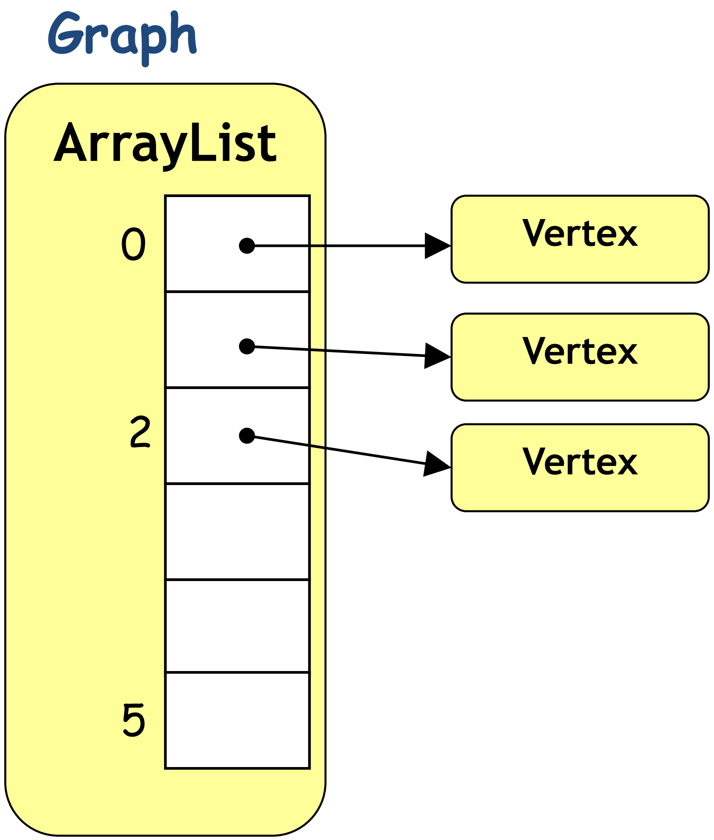


(b)

undirected Graph

Vertices' ArrayList **Edges' Map**

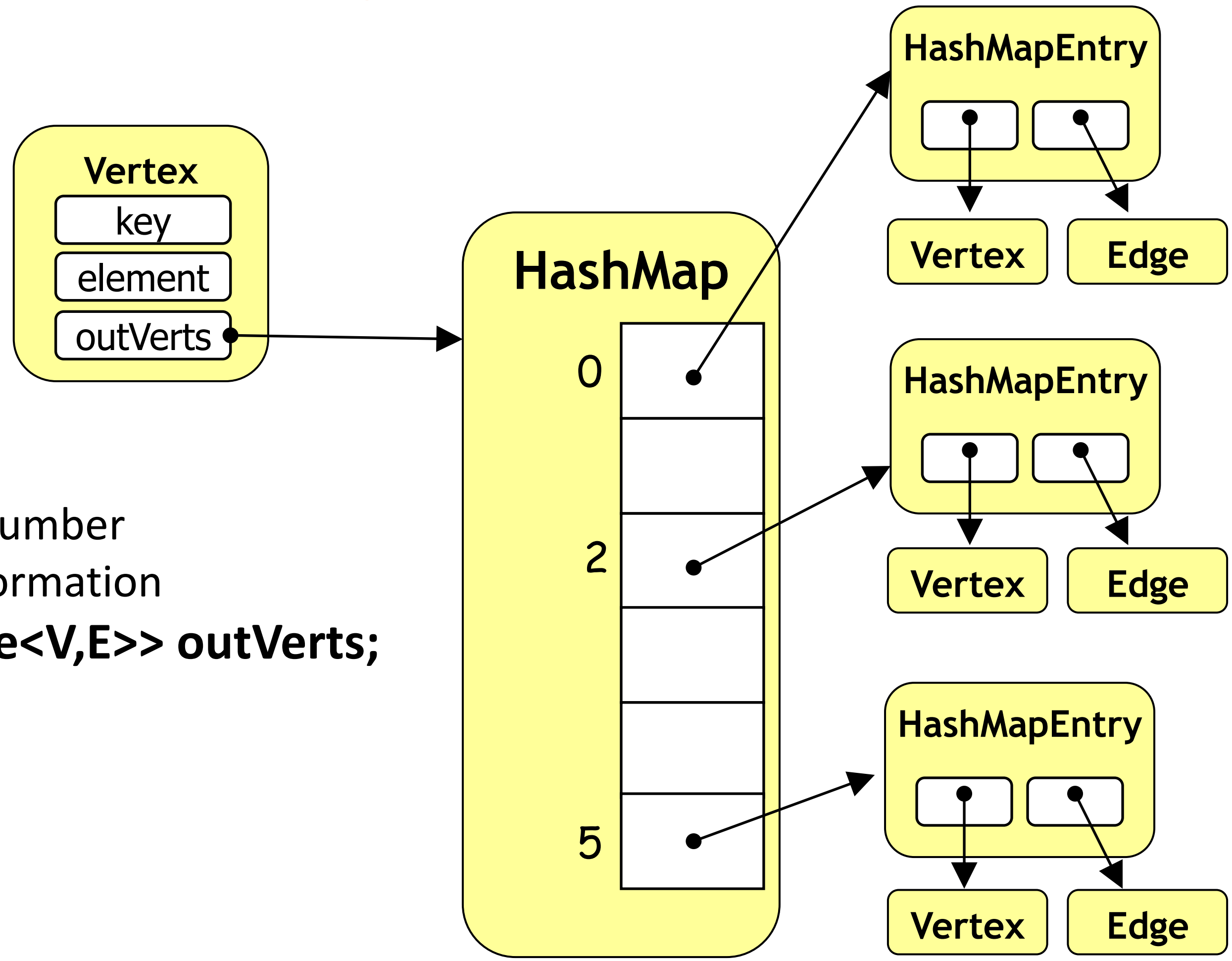
Graph




```
class Graph<V,E> {

   int numVert;
   int numEdge;
   boolean isDirected;
   ArrayList<Vertex<V,E>> listVert;
...
}
```

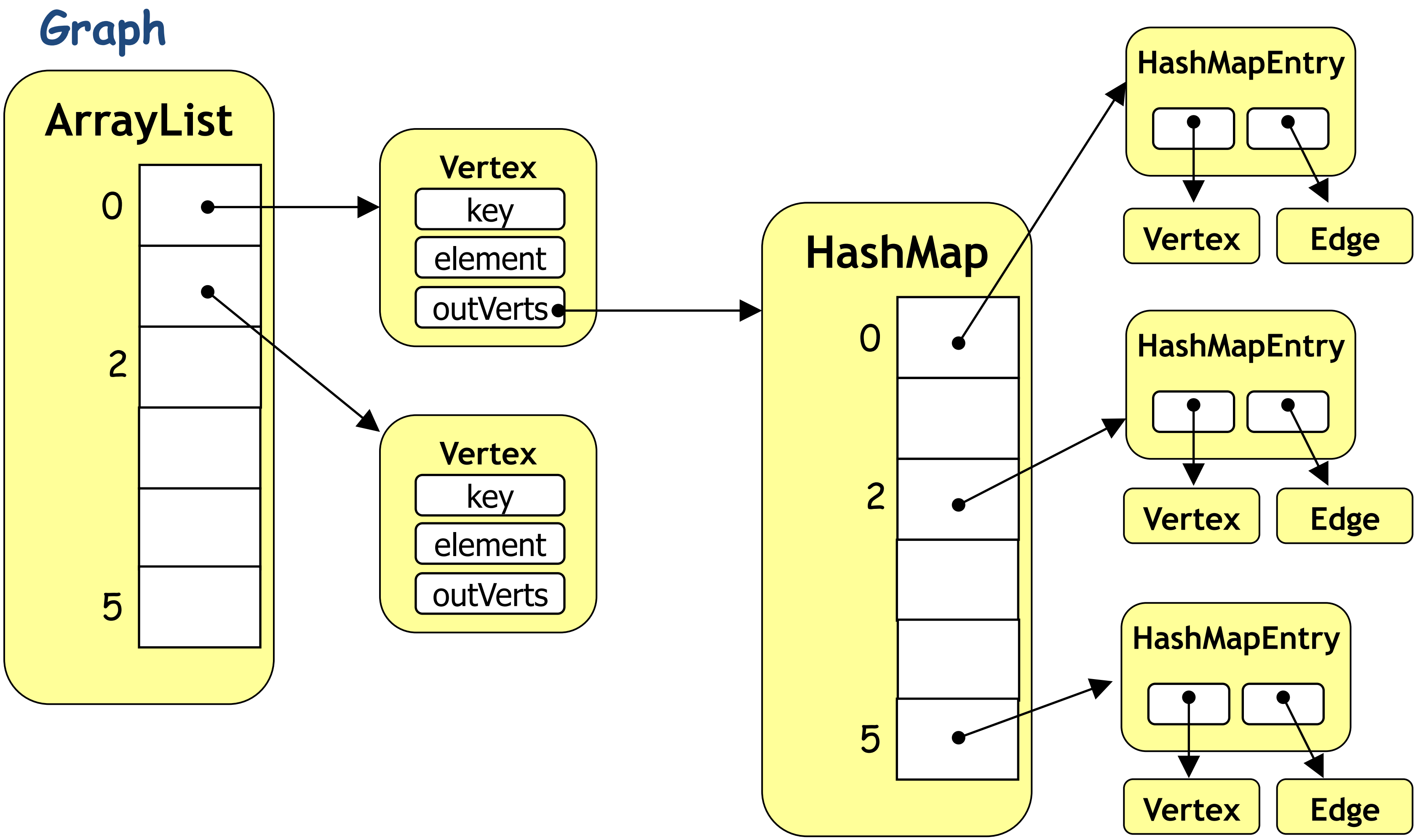
Graphs: Adjacency Map Structure

Graph

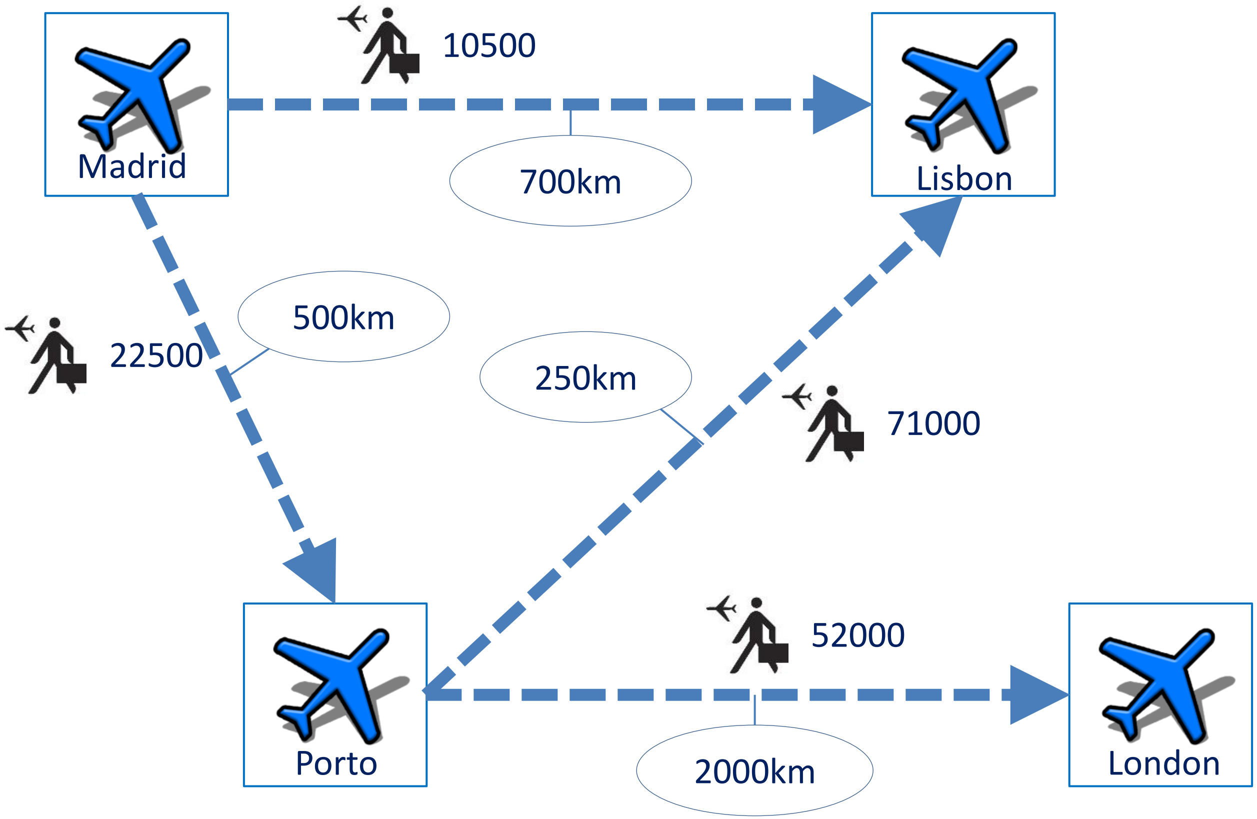



```
class Vertex<V,E> {

  int key ;        //Vertex key number
  V  element ;   //Vertex information
  Map<Vertex<V,E>, Edge<V,E>> outVerts;
 ...
}
```

Graph


Graph
ArrayList
0
2
5
Vertex
key
element
outVerts
Vertex
key
element
outVerts
HashMap
0
2
5
HashMapEntry
Vertex
Edge
HashMapEntry
Vertex
Edge
HashMapEntry
Vertex
Edge

The Airports' Graph



a. Make the declaration of the class AirportNet

```
public class AirportNet {

    private Graph<String, Integer> gAirports;
...
}
```

Air traffic

Airport

Add the following methods to the class AirportNet:

b. Return the numeric key of a given airport

```
public int keyAirport (String airp){
    return gAirports.getVertex(airp).getKey();
}
```

c. Return the airport with a specific numeric key

```
public String airportbyKey (int key){
    Vertex<String, Integer> vairp = gAirports.getVertex(key);
    return vairp.getElement();
}
```

Add the following methods to the class AirportNet:

d. Return the traffic between two airports

```
public Integer trafficAirports (String airp1, String airp2){

    Vertex<String, Integer> vairp1=gAirports.getVertex(airp1);
    Vertex<String, Integer> vairp2=gAirports.getVertex(airp2);

    if (vairp1 == null || vairp2 == null)
       return null;

    Edge<String, Integer> edge = gAirports.getEdge(vairp1, vairp2);

    if (edge == null)
       return null;

    return edge.getElement();
}
```

Add the following methods to the class AirportNet:

e. Return the miles between two directly linked airports

```
public Double milesAirports (String airp1, String airp2){

    Vertex<String, Integer> vairp1=gAirports.getVertex(airp1);
    Vertex<String, Integer> vairp2=gAirports.getVertex(airp2);

    if (vairp1 == null || vairp2 == null)
       return null;

    Edge<String, Integer> edge = gAirports.getEdge(vairp1, vairp2);

    if (edge == null)
      return null;

    return edge.getWeight();
  }
```

Add the following methods to the class AirportNet:

f. Return the number of routes (origin + destination) for all airports

```
public String nroutesAirport (){

    String routesAirp = "";

    for (Vertex<String, Integer> vertex : gAirports.vertices()){

        int grau = gAirports.outDegree(vertex) + gAirports.inDegree(vertex);
        routesAirp += vertex.getElement() + “ “ + grau + "\n";
    }
    return routesAirp;
}
```

Add the following methods to the class AirportNet:

g. Return the airports with the greatest connection (more miles)

public String AirpMaxMiles ( ){

(1) define data structures

- result to return
- greatest connection value (global)
- greatest connection for each vertex (local)
- arrays to store for each vertex its greatest connection and associated airports

(2) for each vertex

search all connections

find the greatest connection

store connection information

(3) Return from the arrays the airports' connection which distance is equals to the greatest connection found

}

Add the following methods to the class AirportNet:

g. Return the airports with the greatest connection (more miles)

```
public String AirpMaxMiles ( ){

(1)
    int nverts = gAirports.numVertices();
    String airpMaxmiles="";                      //result to return
    double[] milesbetairps = new double[nverts]; //greatest conn. each airport
    String[] airports = new String[nverts];      //orig. and dest. each airport
    double maxmiles=0;                           //longest connection
    int i=0;                                     //index for array

(2)…
(3)…

}
```

Add the following methods to the class AirportNet:

g. Return the airports with the greatest connection (more miles)

```
 public String AirpMaxMiles ( ){
(1)…
(2)
for (Vertex<String, Integer> vertex : gAirports.vertices()){ //for each airport
        double maxmilesAirp=0;                      //its longest connection
        String airpsmaxdist="";                     //orig. and dest. of its longest connection
        for (Edge<String, Integer> edge : gAirports.outgoingEdges(vertex)){
           if (maxmiles < edge.getWeight())         //found a greater connection (global)
              maxmiles = edge.getWeight();
           if (maxmilesAirp < edge.getWeight()){    //found a greater connection (local)
              maxmilesAirp = edge.getWeight();
              Vertex<String, Integer> vDest=airport.opposite(vertex, edge);
              airpsmaxdist = vertex.getElement()+" "+vDest.getElement();
           }
        }
        milesbetairps[i]=maxmilesAirp;              //vertex's greatest connection
        airports[i]=airpsmaxdist;                   //vertex connection orig. and dest.
        i++;
      }
}
(3)…
}
```

Add the following methods to the class AirportNet:

g. Return the airports with the greatest connection (more miles)

```
 public String AirpMaxMiles ( ){

(1)…
(2)…
(3)
    for (i=0; i < nverts; i++)
          if (milesbetairps[i] == maxmiles)
             airpMaxmiles += airports[i]+" "+maxmiles+"\n";

    return airpMaxmiles;

}
```

And if an airport has more than one connection whose distance is equals to the greatest value?

Add the following methods to the class AirportNet:

h. Check whether two airports are reachable

```
public Boolean ConnectAirports (String airp1, String airp2){

      Queue<String> qairps = DepthFirstSearch(gAirports, airp1);

      for (String airp : qairps){
         if (airp.equals(airp2))
             return true;
      }
      return false;
}
```

```
return DepthFirstSearch(gAirports, airp1).contains(airp2);
```

The Museum's Graph

a. Make the declaration of the class Museum

```
public class Museum {
   private Graph<Room,Integer> exhibition ;
...

 //----------- nested Room class ------
   public class Room {

      private Integer number;
      private Double time;

      public Room (Integer n, Double t) {
         number=n;
         time=t;
   }

}
```

Nothing to declare !

b. Determine the time to visit all the rooms of the exhibition.

```
public double timevisitAllrooms (){

    double tottime=0;

    for (Vertex<Room,Integer> vertex : exhibition.vertices()){
       tottime += vertex.getElement().getTime();
    }

    return tottime ;
 }
```

c. Provide all possible routes between two exhibition rooms, whose time of visit is less than a certain value.

```
public ArrayList<Deque<Room>> visitwithLimitedtime (Room r1, Room r2, double time){

    ArrayList<Deque<Room>> allvisits = allPaths (exhibition, r1, r2);
    ArrayList<Deque<Room>> visits = new ArrayList<>() ;

    for (Deque<Room> pathExhib : allvisits){

        double timevisit = timeOnevisit (pathExhib);
        if (timevisit <= time)
            visits.add(pathExhib);
    }
    return visits;
  }
```

```
public double timeOnevisit(Deque<Room> visit){
    double timevisit=0;
    for (Room r : visit){
        timevisit += r.getTime();
    }
    return timevisit;
  }
```

d. Return a full path that involves all the exhibition rooms, starting in a given room.

Starting room

```
public Deque<Room> visitwithAllrooms (Room r){

    for (Vertex<Room,Integer> vertex : exhibition.vertices()){

        Room rdst = vertex.getElement();
        if (rdst.getNumber() != r.getNumber()){

            ArrayList<Deque<Room>> allvisits = allPaths (exhibition, r, rdst);

            for (Deque<Room> pathExhib : allvisits)
                if (pathExhib.size() == exhibition.numVertices())
                    return pathExhib;
        }
    }
    return null;
}
```

The Entertainment Park's Graph

a. Define the graph class Park and its V and E classes

V

E

```
public class Park{
    private Graph<Activity,Double> entertainments;

    public Park (){
        entertainments = new Graph<>(true) ;
    }
...
 //----------- nested Activity class ------
 public class Activity {

        private String code;
        private Double time;

        public Activity (String c, Double t) {
            code=c;
            time=t;
        }
}
```

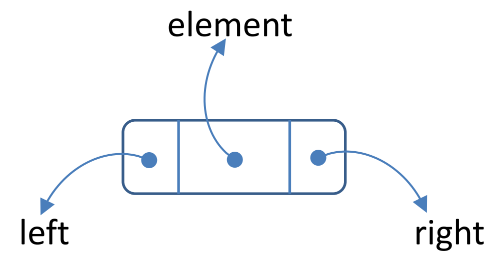
b. Return the shortest path between two activities and present its total time (time of all activities and travel time)

total time   path

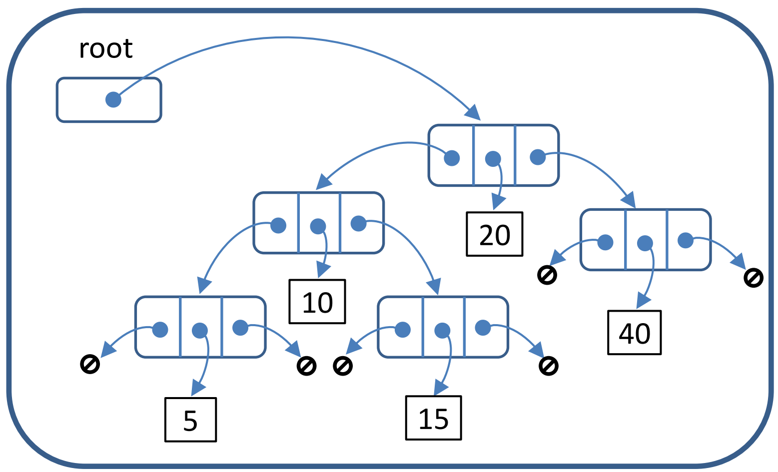
```
public double shortPath (Activity a1, Activity a2, Deque<Activity> shortpath){

    double pathtime = shortestPath (entertainments, a1, a2, shortpath);

    double actstime = 0;                     //activities time
    for (Activity a : shortpath)             //sum the time of all activities
       actstime += a.getTime();

    return (actstime + pathtime);            // return the sum of both times

}
```

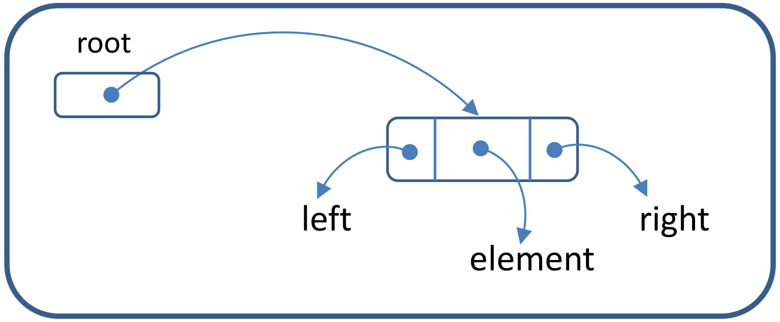
Binary Search Tree

- A BST is a data structure with a set of nodes
- Each node contains an element, a left link, and a right link.



- The left link points to a BST having elements with lesser value
- The right link points to a BST having elemets with greater values





elements must allow comparison between them

```
public class BST <E extends Comparable<E>>{
    protected Node<E> root = null;     // root of the tree
...
//-----------------------------------------------------
    protected class Node<E> {
        private E element;          // an element stored at this node
        private Node<E> left;       // a reference to the left child (if any)
        private Node<E> right;      // a reference to the right child (if any)
    ...
    }
//-----------------------------------------------------
}
```

1. Implement the class TREE by inheriting from the BST

```
public class TREE<E extends Comparable<E>> extends BST<E>{


}
```

a) Implement a method that returns an iterable list with the left tree of the main elements of the root in increasing order and the elements of the right tree in descending order.

```
public Iterable<E> ascdes(){
   List<E> result = new ArrayList<>();
   if(root!=null){

      ascSubtree(root().getLeft(), result);

      result.add(root().getElement());

      desSubtree(root().getRight(), result);
   }

   return result;
}
```

Left subtree in ascending order

The root element

Right subtree in decending order

a) Implement a method that returns an iterable list with the left tree of the main elements of the root in increasing order and the elements of the right tree in descending order.

```
private void ascSubtree(Node<E> node, List<E> snapshot) {
    if(node == null)
        return;
    ascSubtree(node.getLeft(), snapshot);
    snapshot.add(node.getElement());
    ascSubtree(node.getRight(), snapshot);
}
```

inorder

```
private void desSubtree(Node<E> node, List<E> snapshot) {
    if(node == null)
        return;
    desSubtree(node.getRight(), snapshot);
    snapshot.add(node.getElement());
    desSubtree(node.getLeft(), snapshot);
}
```

inorder inverted

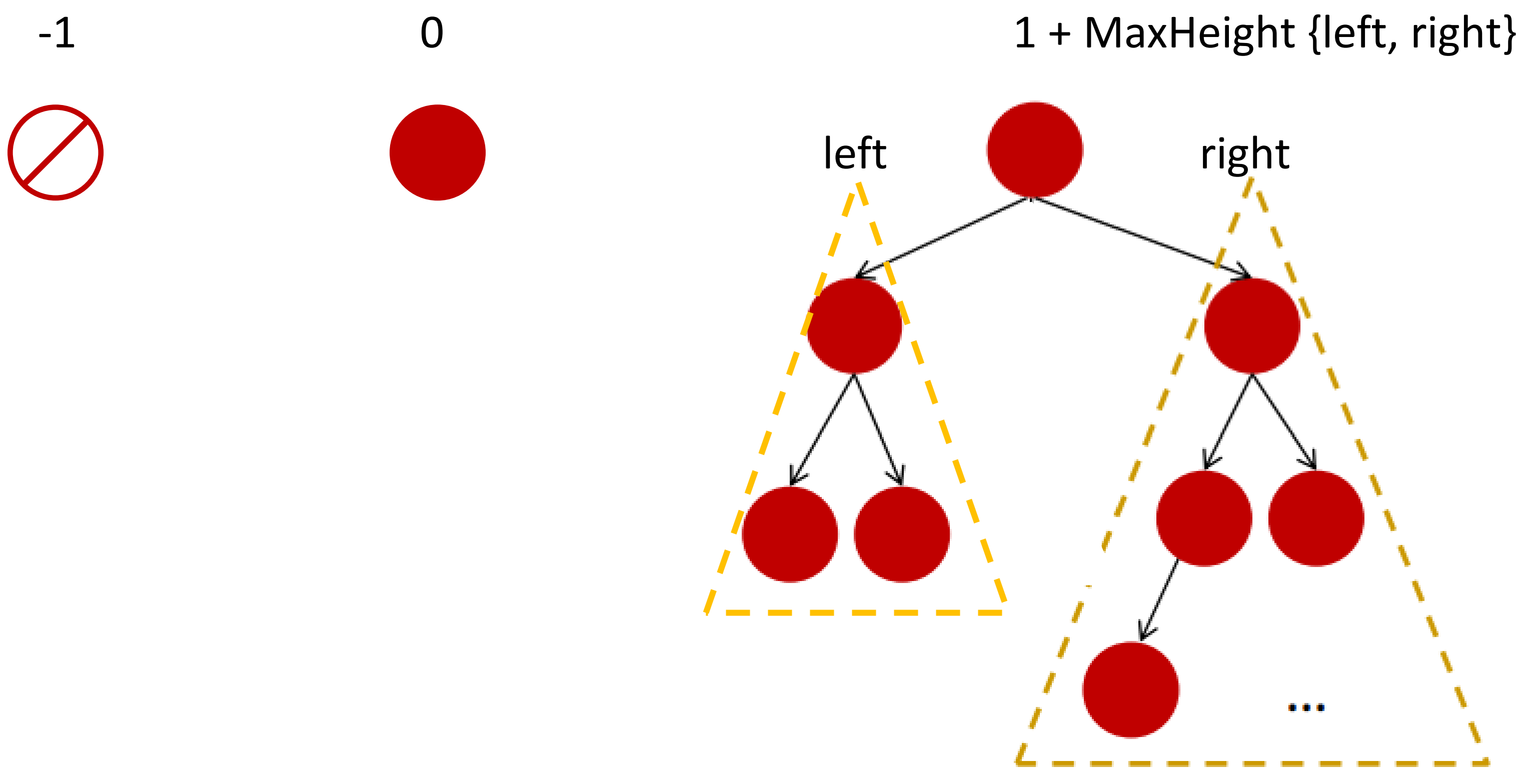
b) Implement a method that returns a new binary search tree, identical to the original, but without leaves.

```
public BST<E> autumnTree() {
    BST<E> newTree=new TREE();
    newTree.root = copyRec(root);
    return newTree;
  }

private Node<E> copyRec(Node<E> node){
   if(node==null)
      return node;

   if(node.getLeft()!=null || node.getRight()!=null)
      return (new Node(node.getElement(), copyRec(node.getLeft()), copyRec(node.getRight())));

   return null;
}
```

Is NOT a leaf

Is a leaf

Implement a method that returns the height of a binary search tree.



Implement a method that returns the height of a binary search tree.

```
 public int height (){
   return height (root);
}

protected int height (Node<E> node){
    if (node == null)
        return -1;

    return 1 + Math.max( height (node.getRight()) , height (node.getLeft()) );
}
```

The height of any tree/subtree

Left subtree

Right subtree

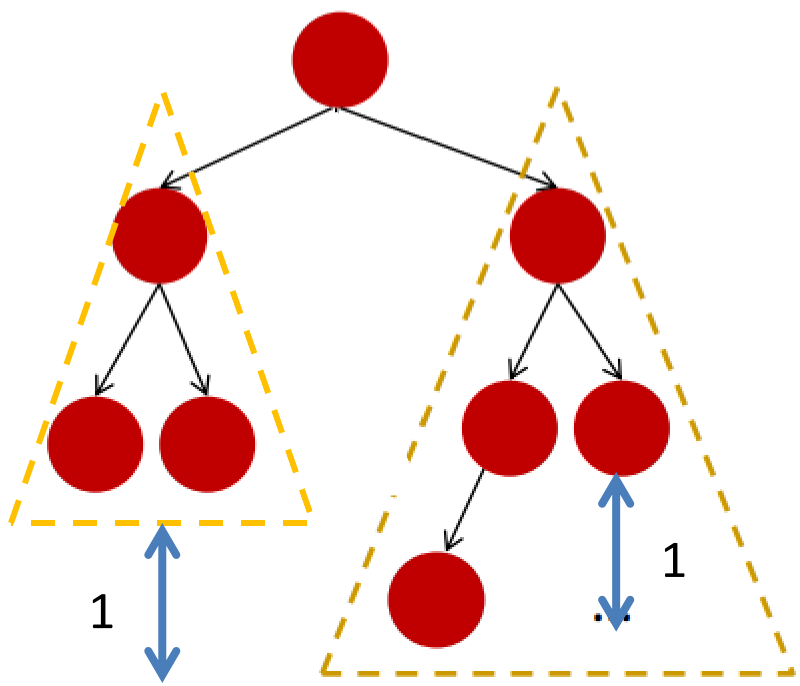
This node + greatest height of its subtrees

AVL Tree

AVL - Georgy **A**delson-**V**elsky and Evgenii **L**andis' tree

An AVL tree is a binary search tree which has the following properties:

- The sub-trees of every node differ in height by at most one.
- Every sub-tree is an AVL tree.

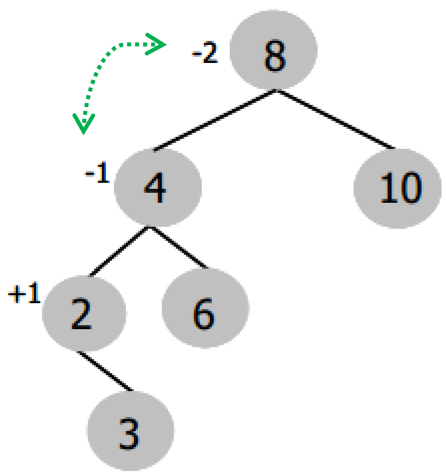


Each node has a Balance Factor: **BF = height (right subtree) - height (left subtree)**

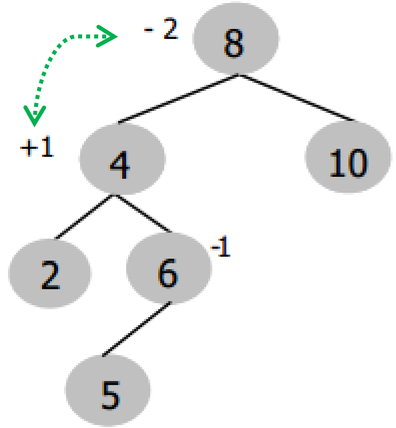
Balancing is required when inserting and removing a new node violates the balancing property: nodes in the tree with BF ∊ [-1,..,1]

The tree balancing is achieved by Rotations:

- **Simple** - when the unbalanced node presents the same BF signal as its child's root node of unbalanced subtree
- **Double** - when the unbalanced node presents a BF signal contrary to its child's root node unbalanced subtree



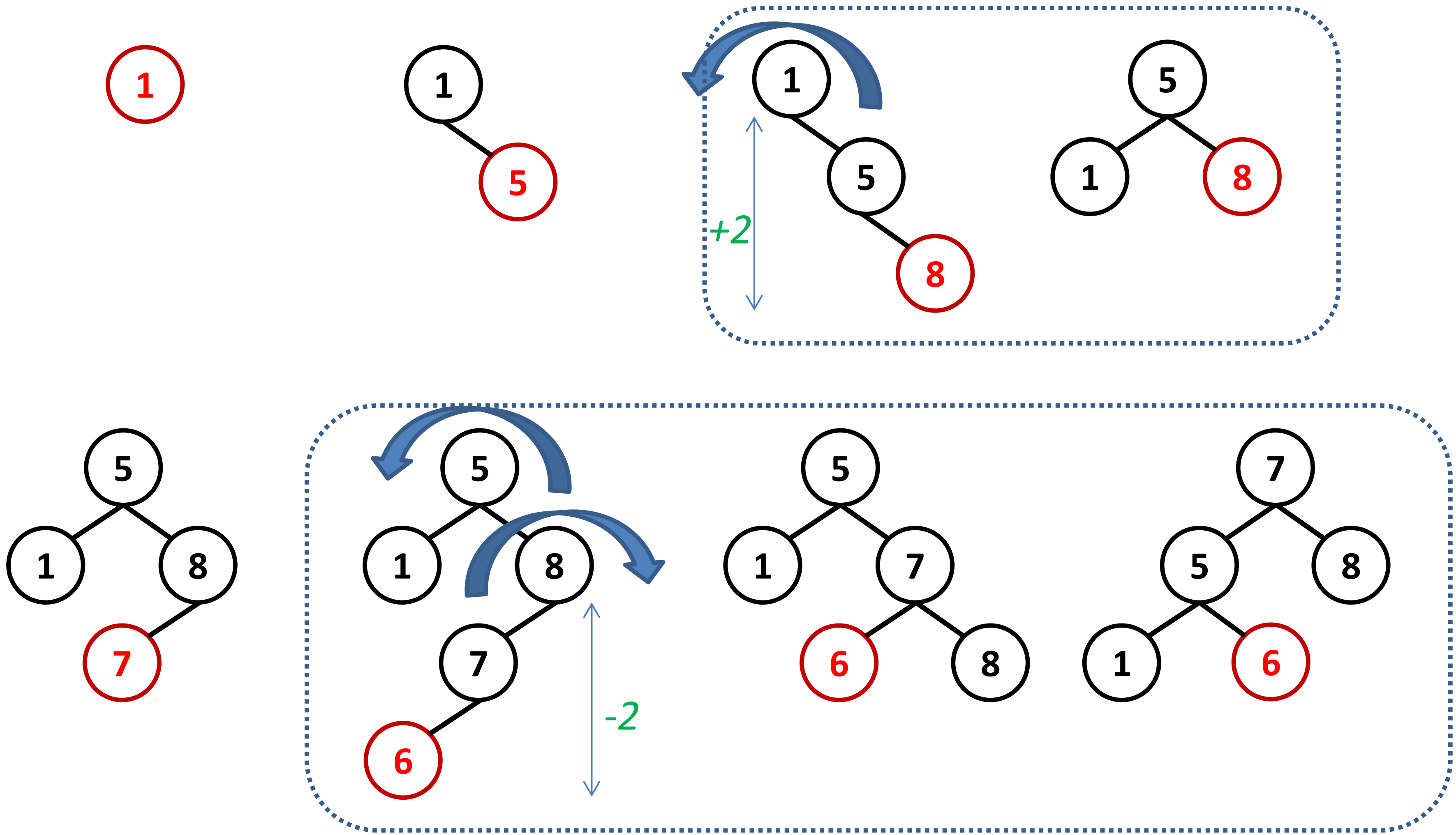
Simple rotation



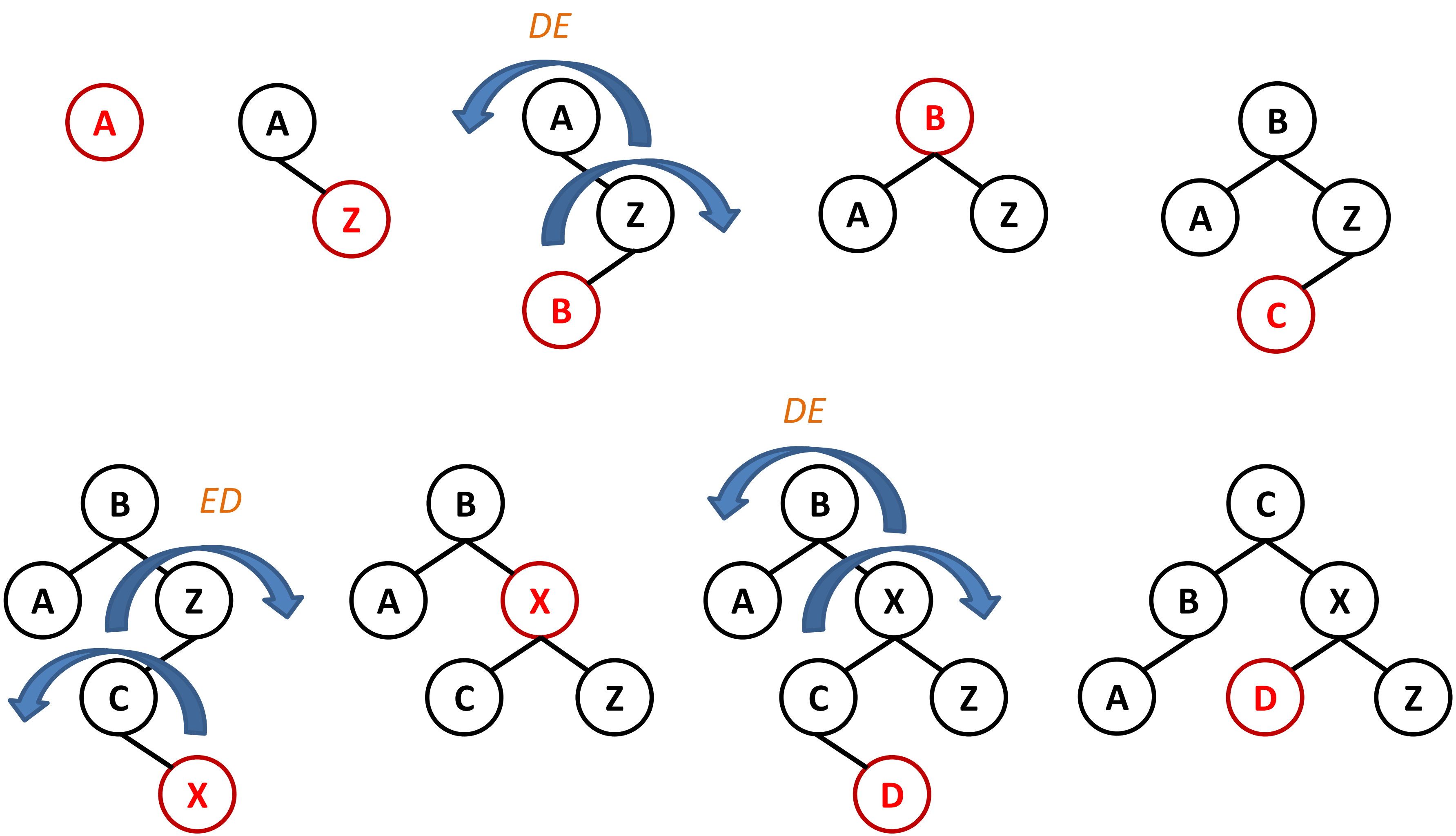
Double rotation

Draw the tree after each operation and presenting the respective rotations.

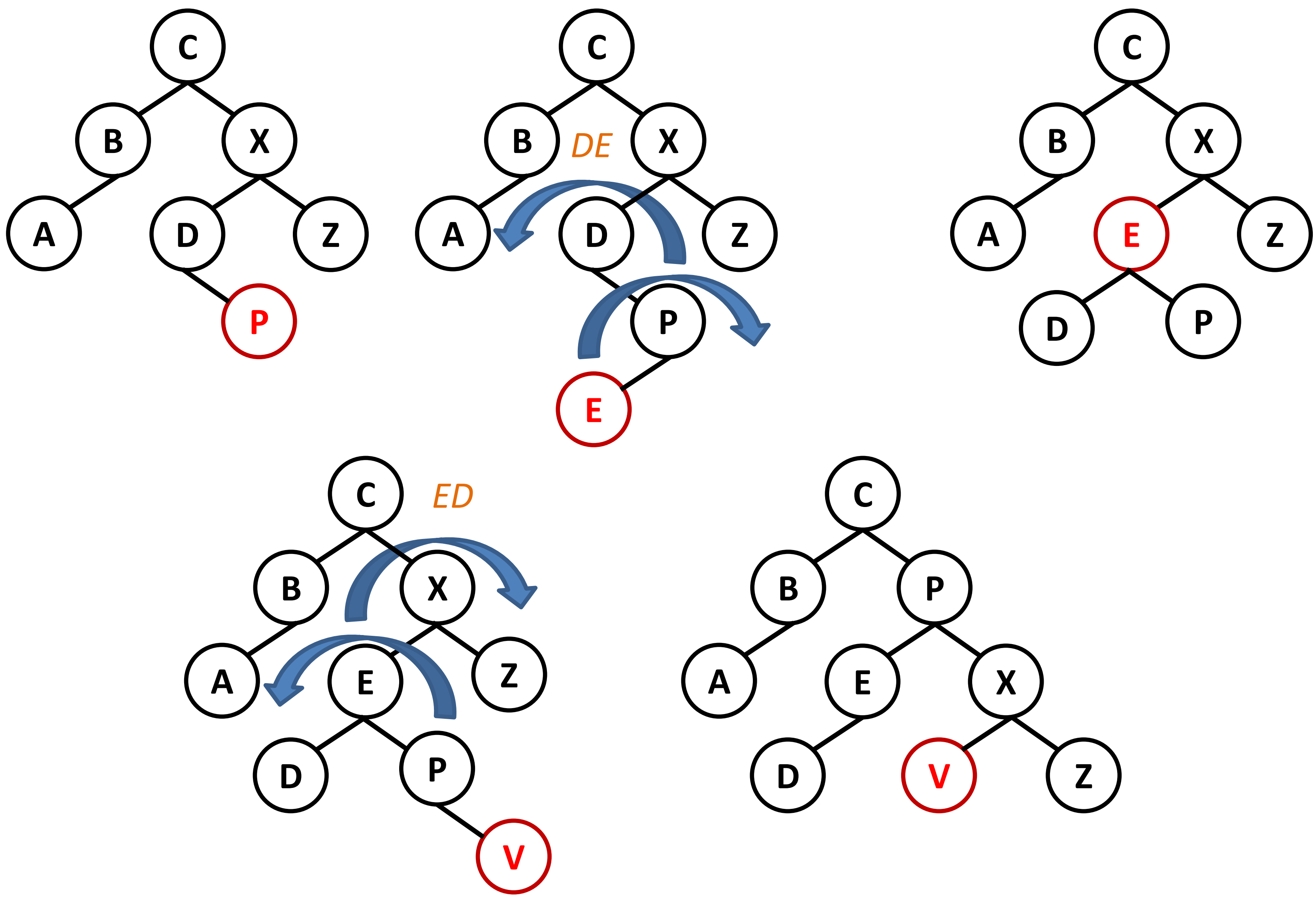
a) Insert the following elements: 1,5,8,7,6



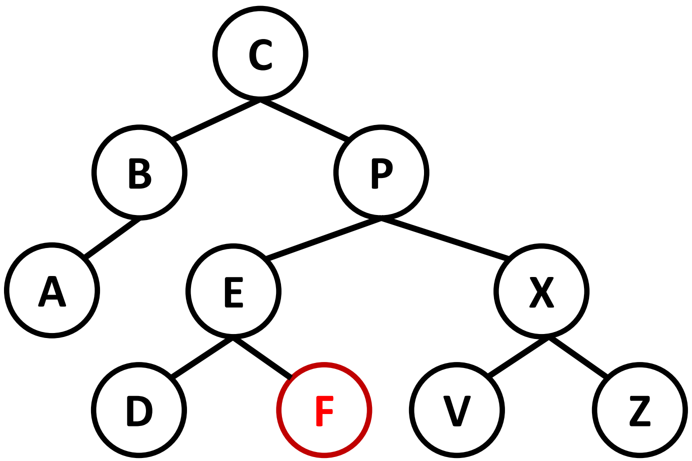
b) Insert the following elements: A, Z, B, C, X, D, P, E, V, F, Q

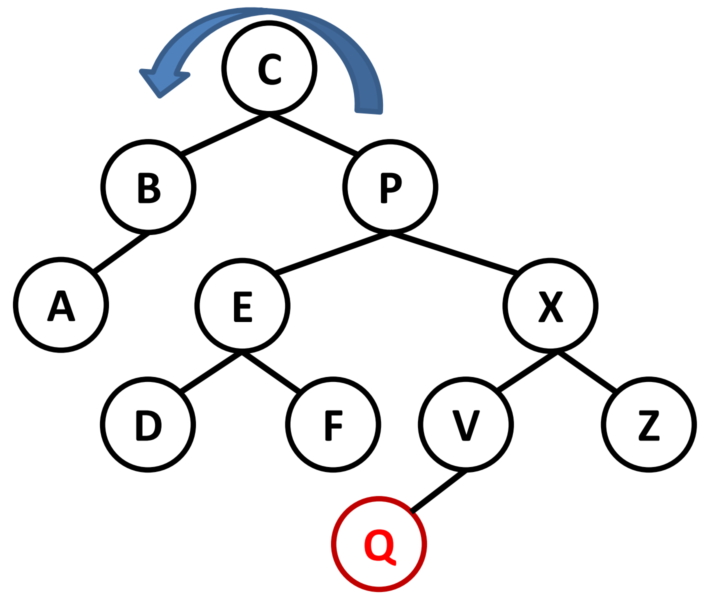


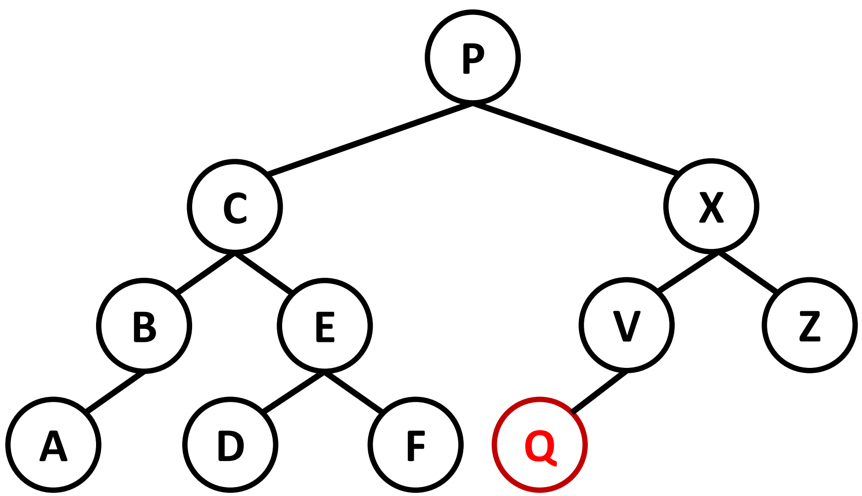
b) Insert the following elements: A, Z, B, C, X, D, P, E, V, F, Q



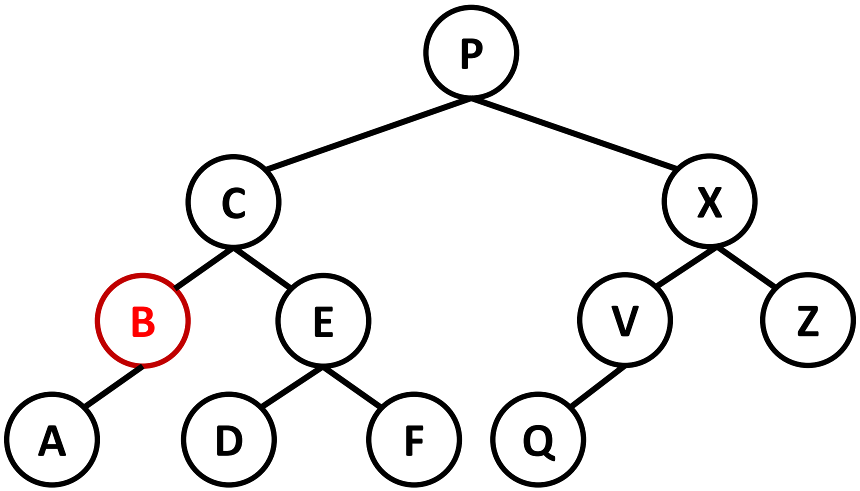
b) Insert the following elements: A, Z, B, C, X, D, P, E, V, F, Q

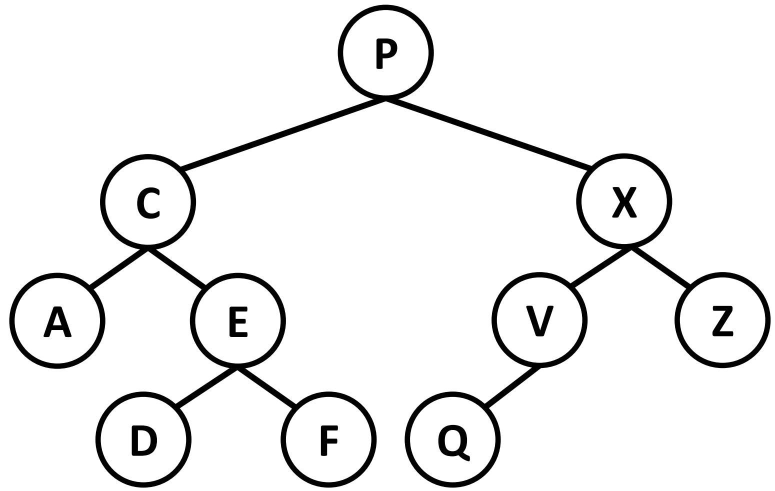




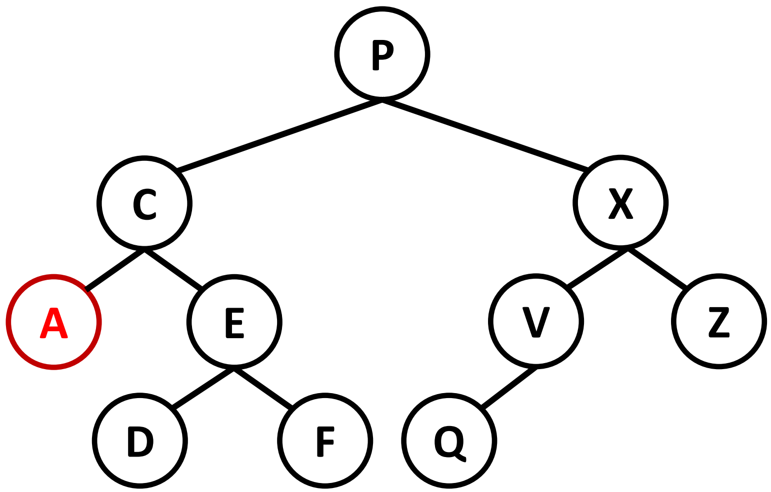


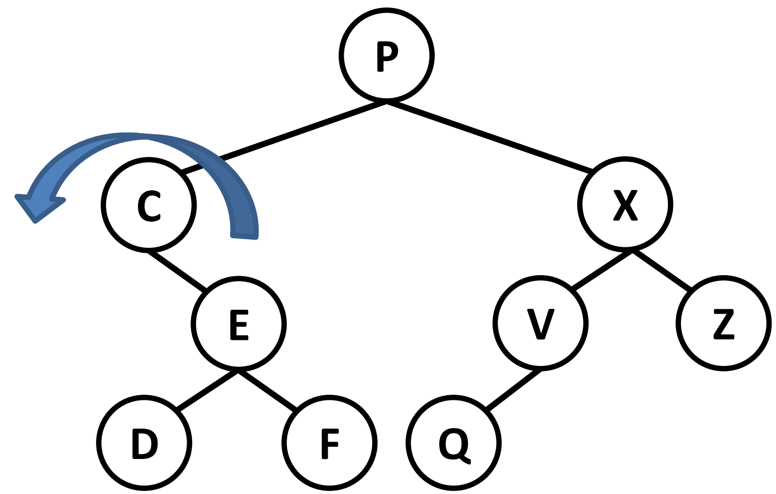
c) Remove the following elements: ***B****, A, F, P, C*

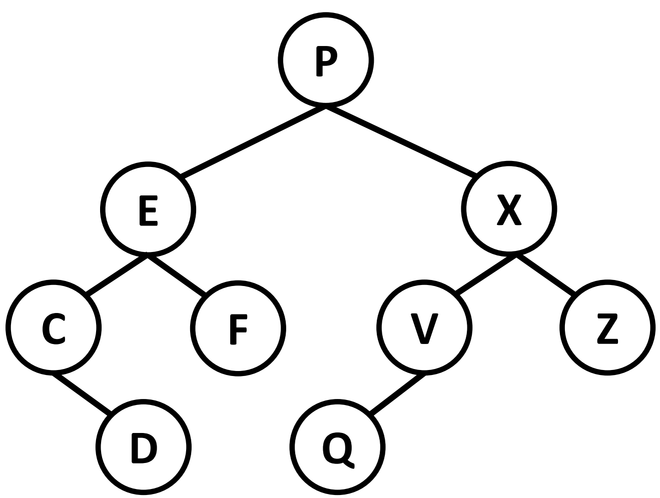




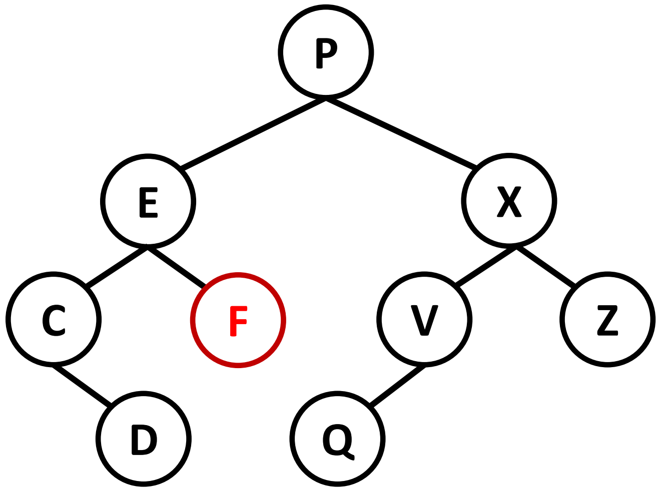
c) Remove the following elements: B, ***A****, F, P, C*

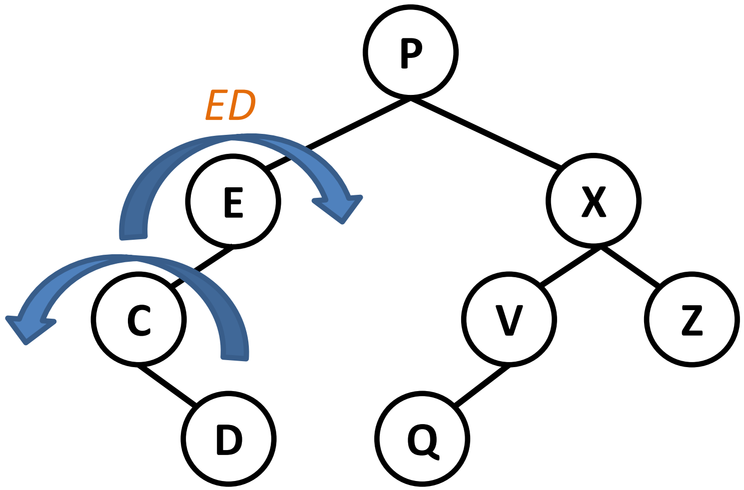


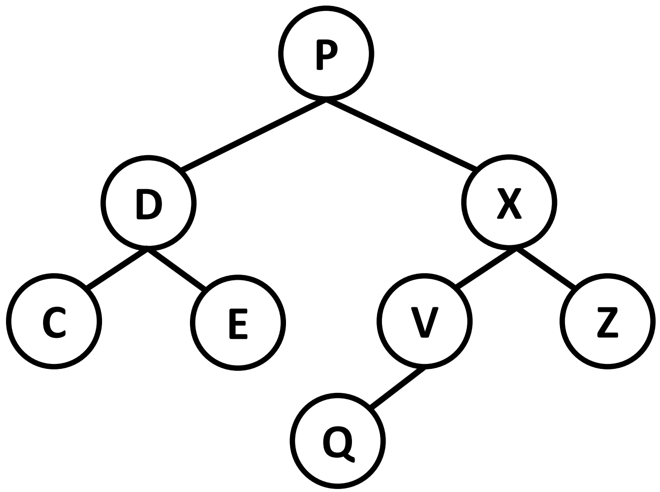




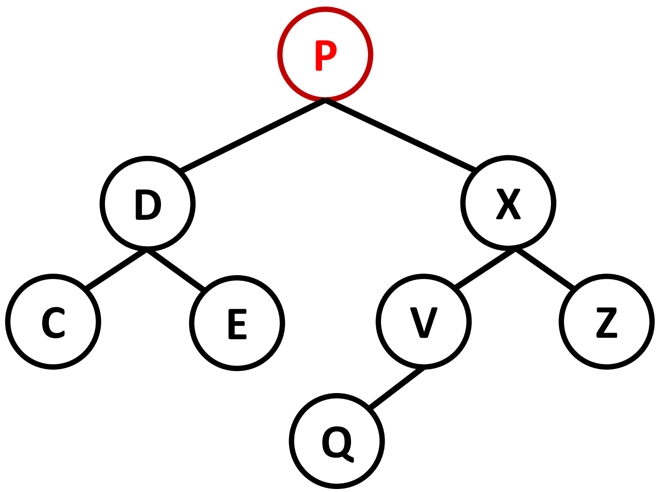
*c) Remove the following elements: B, A, **F**, P, C*

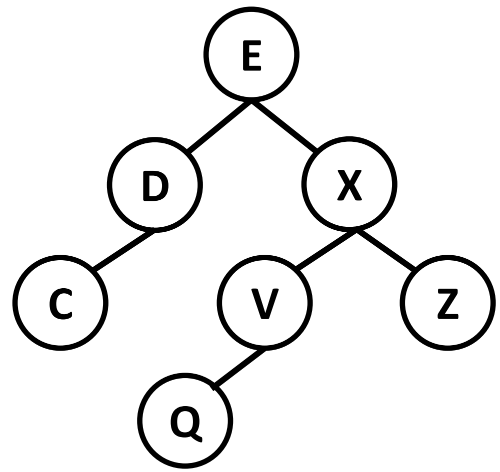




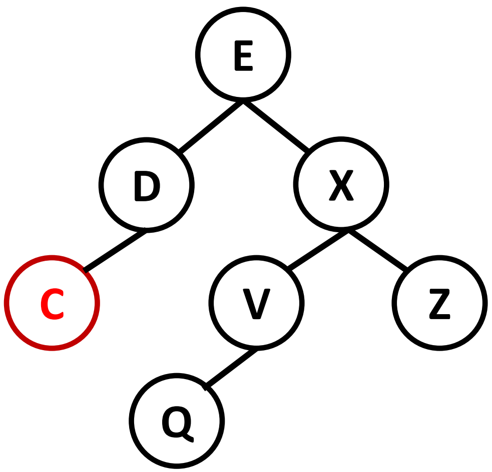


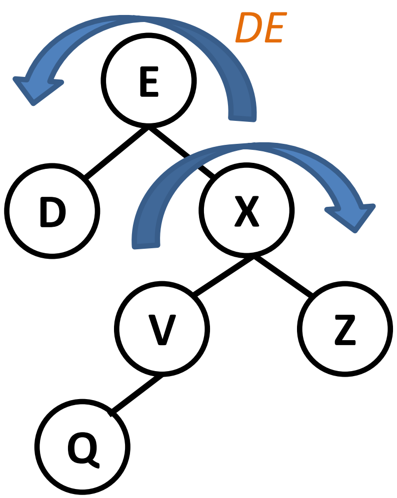
c) Remove the following elements: B, A, F, ***P****, C*

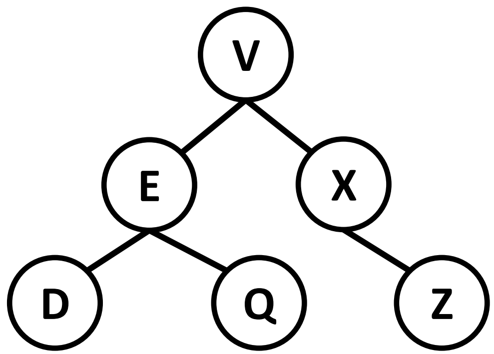




c) Remove the following elements: B, A, F, P, ***C***





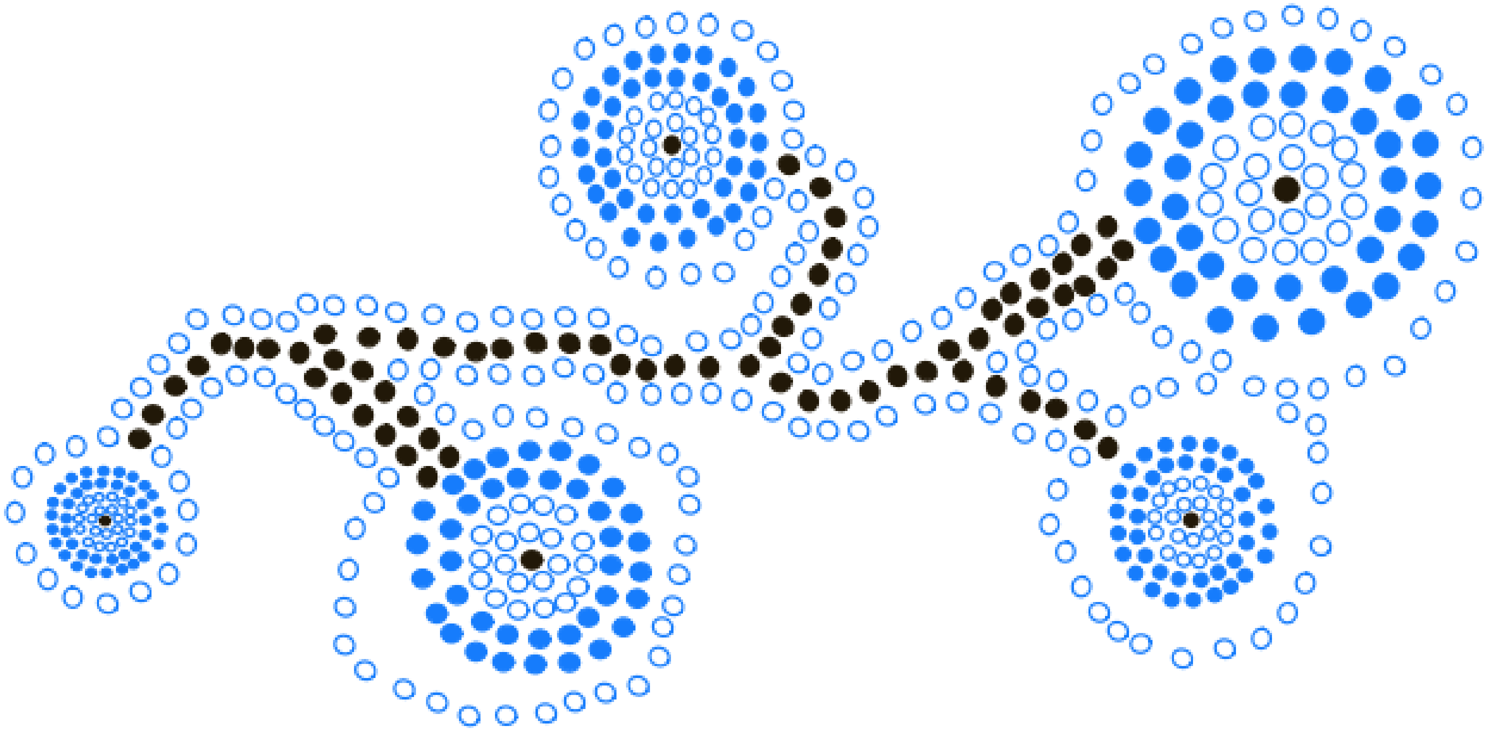


2. Develop a method to find if an AVL tree is perfectly balanced, that is, a tree where all nodes have an equilibrium factor (eqf) of 0.

```
public boolean perfectlyBalanced (){
        return perfectlyBalanced (root);
}

private boolean perfectlyBalanced (Node<E> node){
        if (node==null)
            return true;

        if(balanceFactor(node) !=0 )
            return false;

        return perfectlyBalanced(node.getLeft()) && perfectlyBalanced(node.getRight());
}
```

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
Instituto Superior de Engenharia do Porto

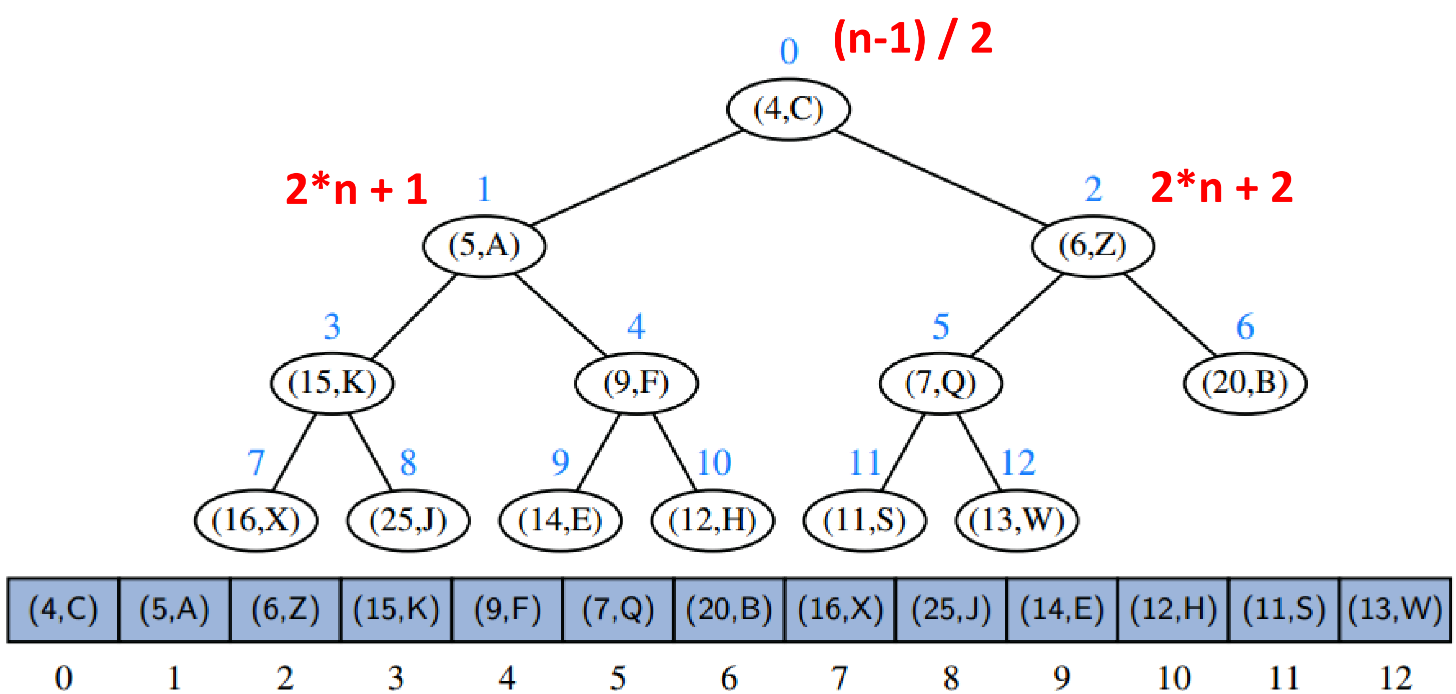
Priority Queues / Heap



An efficient way of implementing priority queues is using a **binary heap.**

A **Heap** is a complete binary tree in which every node's value is less or equal to its children values.

Heap-order implies that each path in the tree is sorted



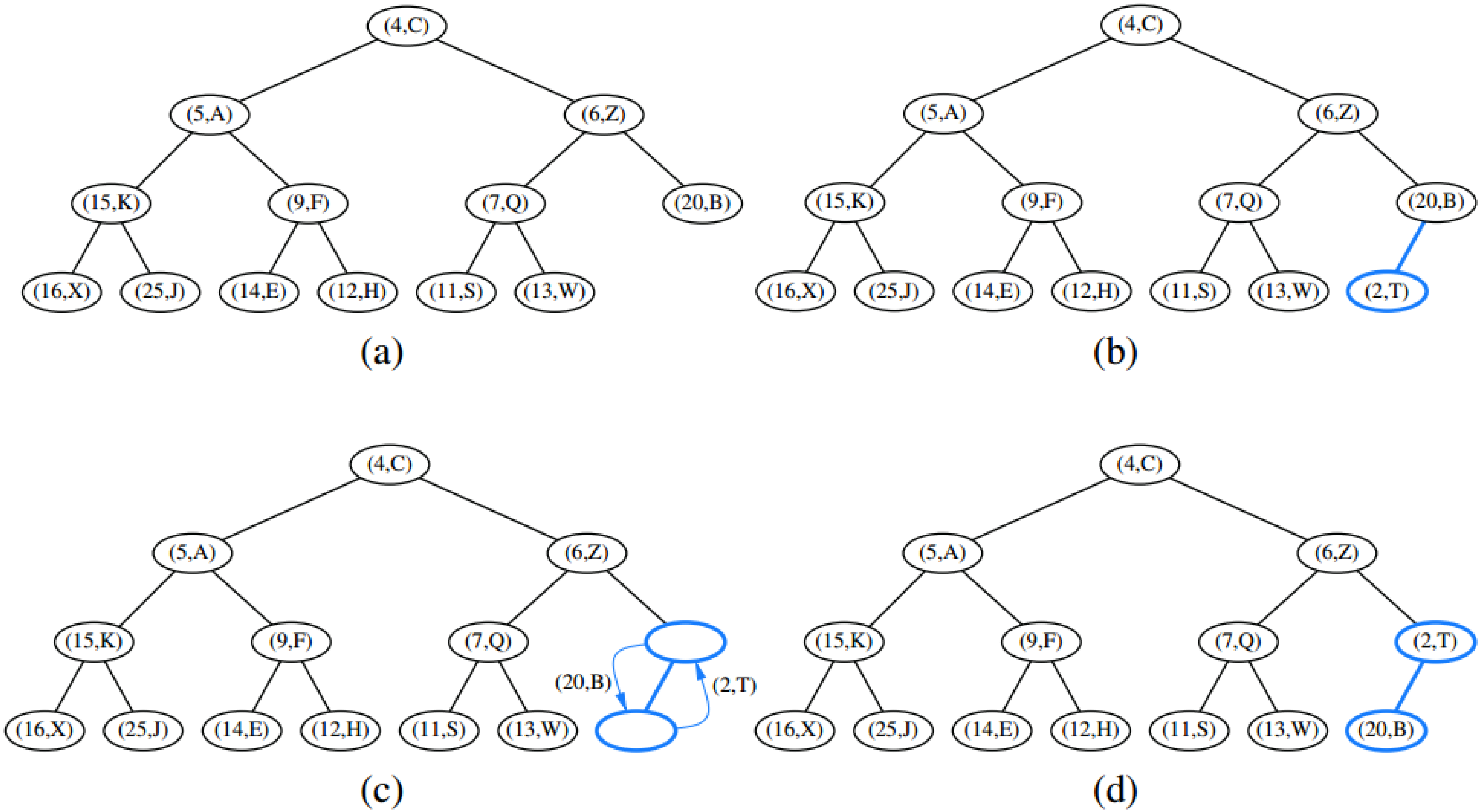
The insertion of an element in a heap must guarantee two properties:

1. The tree remains complete - the new element is inserted on the last level of the tree the rightmost possible

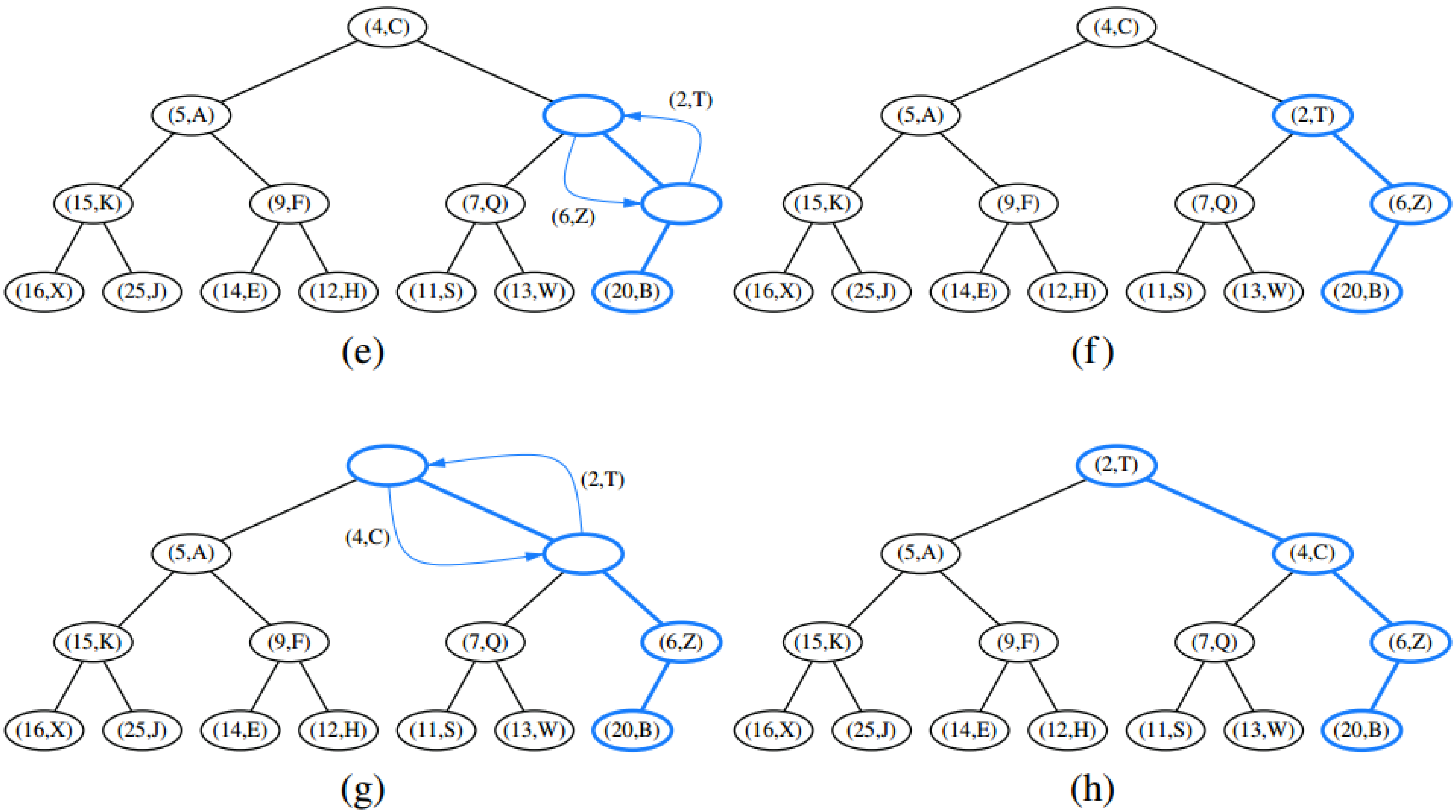
2. 2. the tree remains orderly - Fix it by percolating up

Example of the insertion of the element **(2,T)** in the Heap

Insert element (2,T) in the Heap



Insert element (2,T) in the Heap



The HeapPriorityQueue data structure.

```
public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {

        protected ArrayList<Entry<K,V>> heap = new ArrayList<>();
                                  Key         Value
}
```

```
public abstract class AbstractPriorityQueue<K,V> implements PriorityQueue<K,V> {
 //---------------- nested PQEntry class ----------------
 // A concrete implementation of the Entry interface to be used within a PriorityQueue implementation.
protected static class PQEntry<K,V> implements Entry<K,V> {
   private K k;                    // key
   private V v;                    // value

   public PQEntry(K key, V value) {
     k = key;
     v = value;
   }
```

Add the two methods percolateUp() and percolateDown() to the class HeapPriorityQueue, to perform the bubbling up and down of an entry, by means of successive swaps, in order to restore the heap-order property.

```
protected void percolateUp (int j) {
    while (j > 0) {                                    // continue until root (or break)
        int p = (j-1) / 2                              // parent(j);
        if (compare(heap.get(j), heap.get(p)) >= 0)
            break;                                     // heap property verified
        swap (j, p);
        j = p;                                         // continue from the parent's location
    }
}

protected void swap (int i, int j) {
   Entry<K,V> temp = heap.get(i);
   heap.set(i, heap.get(j));
   heap.set(j, temp);
}
```

```
protected void percolateDown (int j) {
   while (hasLeft (j)) {                          // continue to bottom (or break)
     int leftIndex = left (j);
     int smallChildIndex = leftIndex;              // although right may be smaller
     if (hasRight (j)) {
          int rightIndex = right (j);
          if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
               smallChildIndex = rightIndex;       // right child is smaller
     }
     if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0)
          break;                                   // heap property has been restored
     swap (j, smallChildIndex);
     j = smallChildIndex;                          // continue at position of the child
   }
 }

boolean hasLeft   (int j) {   return left(j) < heap.size();   }
boolean hasRight (int j) {   return right(j) < heap.size();   }
int left   (int j) {    return 2*j + 1;    }
int right (int j) {    return 2*j + 2;    }
```

```
public Entry<K,V> insert (K key, V value) {
        Entry<K,V> newest = new PQEntry<>(key, value);
        heap.add(newest);                // add to the end of the list
        percolateUp (heap.size() - 1);   // percolateUp newly added entry
        return newest;
}


public Entry<K,V> removeMin () {
   if (heap.isEmpty())
        return null;

   Entry<K,V> answer = heap.get(0);
   swap (0, heap.size() - 1);            // put minimum item at the end
   heap.remove(heap.size() - 1);         // and remove it from the list;
   percolateDown (0);                    // then fix new root
   return answer;
}
```


That's all Folks!