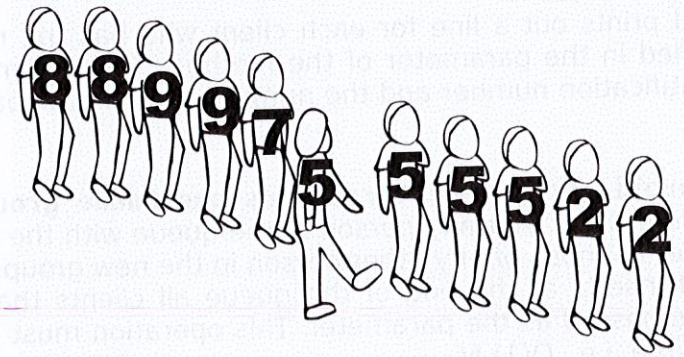
Data Structures and Algorithms
Exam Midterm I (7th-October-2014)

The service staff of a restaurant has successfully specialized in serving groups of clients. When a group of clients come to the restaurant, each of its members receives a T-shirt with a number written on it, clearly visible to any other client. That number is the same for all members of a group, and it is a different number from other groups. For the service payment, a customer must get in line at checkout, based on the following rules:

1. if there is no person of the same group, he/she gets to the end of the queue
2. if there are people in the same group, he/she joins that group

The following figure shows the state of the queue at a specific point in time, when a new member of group 5 is arriving to the queue:



The software system for managing the restaurant reservations uses several Java classes for representing the queue at checkout. The client is represented by the class **Client**:

```
public class Client {
    private int ntshirt; //number on the T-shirt
    private String taxnum; // Tax identification number
    private double bill; //amount to pay

    public Client(int tshirt, String tin, double pounds){
        ntshirt = tshirt;
        taxnum = tin;
        bill = pounds;
    }
    public int getTshirt(){return ntshirt;}
    public String getTin(){return taxnum;};
    public double getBill(){return bill;}
```

[Part A] Your task is to include in the class **CustomerQueue** the full implementation of the following methods, mentioning the order of growth O() of their running time. You may define other helper methods. Note: "*preconditions*" are conditions that are given and that you do not need to check them again.

1. **[1 point] public void print()**
   /* *Precondition: none*
   Description: This method prints out, in the standard output, a line for each person in the queue, with tax number, group number and amount to pay, separated by " " */

2. **[1 point] public void addClient2rear(Client clientdata)**
   /* *Precondition*: there isn't any person in the queue with the same T-shirt or Tax number.
   Description: This method inserts a client at the end of the queue */

3. **[1 point] public void print(int tshirt)**
   /* *Precondition: none*
   This method prints out a line for each client who has the number of T-shirt specified in the parameter of the method. The line must contain the Tax identification number and the number of T-shirt, separated by " " */

4. **[1 point] public void insertGroup(CustomerQueue groupclients)**
   /* *Precondition*: there isn't any person in the queue with the same T-shirt or Tax number as those of any other person in the new group of clients.
   This method inserts at the end of the queue all clients that are in the queue that is passed as the parameter. This operation must be executed in constant time, i.e., O(1) */

5. **[2 points] public void invoiceGroups()**
   /* *Precondition: none*
   This method prints out in standard output a line for each group in the queue. The line states the sum of the bills of every person in the group */

6. **[2 points] public void insertClient(Client clientdata)**
   /* *Precondition*: there isn't any person in the queue with the same Tax number.
   This method inserts a Client in the queue, according to the rules of the restaurant */

[Part B]

1. **[2 points]** The tables in the restaurant have seats for **four** people and there is room for accomodating as many people as we wish (that is, unlimited space for tables). The restaurant management is committed to foster intergroup relationships and dialog. Consequently, they arrange the clients in the following way:
   - a table will never have two clients of the same group

The queue **CustomerQueue** is implemented by means of a series of **linked nodes NodeQ**, as described below:

```
public class CustomerQueue {
    * This is a queue of groups of customers

    private NodeQ first;
    private NodeQ last;
    private int count;

    public CustomerQueue() {
        first = null;
        last = null;
    }
```

The class **NodeQ** has available the usual getters and setters.

```
public class NodeQ {
    private Client aclient;
    private NodeQ previous;
    private NodeQ next;

    public NodeQ(){
    public NodeQ(Client data) {
        aclient = data;
        previous = null;
        next=null;
    }
    public Client getAclient() {
    public void setAclient(Client aclient) {
    public NodeQ getPrevious() {
    public void setPrevious(NodeQ previous) {
    public NodeQ getNext() {
    public void setNext(NodeQ next) {
```

Software analysts have come up with a design that includes the following methods of the class **CustomerQueue.** Those classes still need to be implemented with the description found in the next pages:

```
public void print() {

public void addClient2rear(Client clientdata) {

public void print(int tshirt) {

public void insertGroup(CustomerQueue groupclients) {

public void invoiceGroups() {

public void insertClient(Client clientdata) {

public double removeGroup(int tshirt) {

public Client removeFirstinLine() {
```

- management will try to minimize the number of tables, by filling all seats of every table.

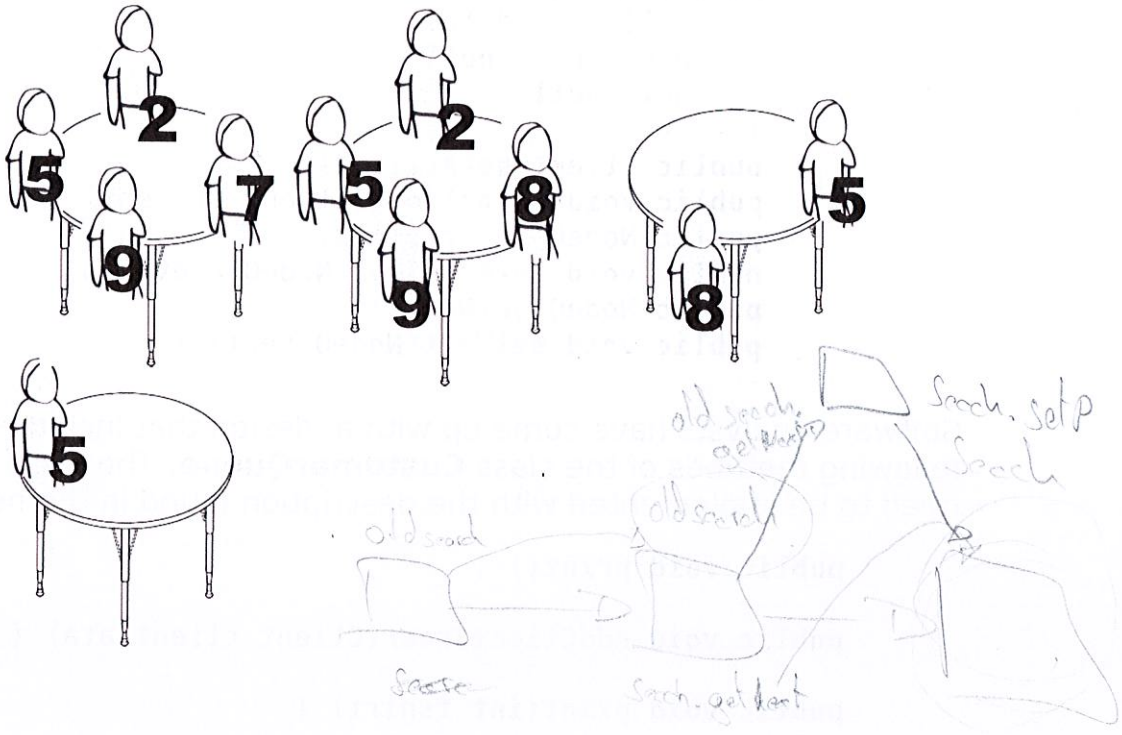
There is a queue of clients already waiting to be seated.
You must implement an executable Java class (i.e. a **public static void main"** class), different from CustomerQueue, that will be in charge of the process. You may use additional data structures.

You may use the methods previously described, jointly with the next one, which is assumed to be fully implemented:

`public Client removeFirstinLine()`

***precondition**: none.*
***description**: this method removes the first person in the queue*
returns: null if the queue is empty. Otherwise it returns the data of the person removed from the queue.

The standard output is a line for each table used, printing out the Tax number and T-shirt of each client in the table.



Linked List → Is a sequence of ~~struc~~ items

One node reference other node

add → first

```
Node first = new Node();
first.item = " ";
~~first~~ = last = first;
```

add other →

```
Node oldlast = new Node();
oldLast = last
Node last = new Node();
last.item = " ";
oldLast.next = last;
```

add ~~mid~~ remaining

```
oldnext = current.next;
current.next = newNode;
NewNode.next = oldnext;
```

Insert Node in the middle

```
if(count == 0){
    first = newNode;
else {Node current = first;
    while( i <= count && !found) {
        if (
            current.tshirt.equals(tshirt) && current.next!=
            Current=Current.next;
        } else {
            found = true;
    }
    if (found){
        Newnode.next(current.next);
        Current.next(newNode);
    }else {
        Current.next(
```

Linear Collections: Stacks and Queues

Stacks: the first in last out

Queue: the first in first out

A Abstract Data type, whose values and operations are not inherently defined, ~~whe~~ we have to define it.

Searching and sorting:

Linear Search:

```
public static <T extends comparable<? super T>>
        boolean linear search (T[] data, int min, int max, T target)
{
  int index = min;
  boolean found = false;
  while (!found && index <= max){
    if (data[index].Compare to[target] == 0) {
        found = true;
    } index ++;
  } return found;
```

Binary search:

```
    boolean binarySearchNonRecursive (T []data, int min, int max, T target
  boolean found = false;
  int index; ~~...~~
  low = min;
  high = max;
  while(!found & low <= hig)
```

```
midpoint = (low + high)/2;
if (data[midpoint].compareTo[target] == 0) { found = true;
else if (data[midpoint].compareTo[target] < 0) { low = midpoint+1;
else high = midpoint - 1;
```

Linked structure:

```
private class Node {Item item;
                    Node next;
public class Person {
    private String name;
    private String address;
    private Person next
    private Person first;
    private Person Last;
Public void addPerson (Person p) {
    if (count == 0) {
        list.name(p.name);
        first.address(p.address);
    else {
        Person person = new Person();
        person.Previous(Last);
        Last.next(person);
```

```
public class ListInt{
    private Node first;
    private Node last;
    private int size;

    static private class Node{
        public int value;
        public Node previous;
        public Node next;

        public Node(Node previous, int v, Node next){
            this.value = v;
            previous = previous;
            next = next;

        public void sort(){ Node current = new Node(first, list.value, next);
            current = first
            while (current.next != null){ current1 = current.next
                while (current1.next != null){
                    if (first.value > current1.value){
                        int v = current1.value;
                        current1.value = first.value;
                        first.value = v;
                    }
                    current1 = current1.next;
                }
                current = current.next
            }
        }
```

```
public void purge(){
  int count = 1; previous = first
                 current = first.next
  while (previous.next != null){
      count++;
      if (count % 2 ≠ 0){
          if (count == 1){
              first = current;
          else{
              previous.next = current.next
              }
          previous = previous.next;
          current = current.next;
      }
```

Estructuras de datos y algoritmos

Primer examen parcial (22-10-2012)

Considera la clase `LinearNode<T>` que aparece a continuación:

```
public class LinearNode<T> {
        private LinearNode<T> next;
        private T elem;

        public LinearNode(){...}
        public LinearNode(T element){...}
        public LinearNode<T> getNext() {...}
        public void setNext(LinearNode<T> next) {...}
        public T getElem() {...}
        public void setElem(T elem) {...}
}
```

y una implementación del TAD Listas desordenadas:

```
public class LinearList<T> {
        private LinearNode<T> first; //apunta al primer nodo de la lista (si hay)

        public LinearList(){...} //Construye una lista vacía
        public T removeFirst(){...} //Elimina el primer elemento de la lista y
                                    //devuelve una referencia al mismo
        public T removeLast(){...} //Elimina el último elemento de la lista y
                                   //devuelve una referencia al mismo
        public T remove(T elem){...} //Elimina la primera aparición de elem
                                     //en la lista y devuelve una referencia al mismo
        public T first(){...} //Da acceso al primer elemento de la lista
        public T last(){...} //Da acceso al último elemento de la lista
        public boolean contains(T elem){...} //Determina si la lista contiene a elem
        public boolean isEmpty(){...} //Determina si la lista está vacía
        public int size(){...} //Determina el número de elementos de la lista
        public void addToFront(T elem){...} //Añade elem al principio de la lista
        public void addToRear(T elem){...} //Añade elem al final de la lista
        public void addAfter(T elem, T target){...} //Añade elem detrás de otro
        //elemento concreto, target, que ya se encuentra en la lista
        public Iterator<T> iterator(){...} //devuelve un iterador para esta colección
}
```

Puedes usar directamente los métodos de estas clases para tus implementaciones. Si quieres usar otros métodos tendrás que implementarlos completamente.

Consideremos la clase:

```
public class Ciclista {
        private int dorsal;     // número de dorsal del ciclista.
        private String nombre; // nombre del ciclista.
        private int tiempoAcumulado; // número de segundos empleados por el ciclista.
        private String categoria; // nombre de la categoría del ciclista.
        //Puede haber varias categorías: infantil, juvenil, aficionado, semiprof, etc.

        public Ciclista (int numDorsal, String mombre, int numSegundos, String cat) {
                dorsal = numDorsal;
                texto = nombre;
                tiempoAcumulado = numSegundos;
                categoria = cat;
        }
        //Con sus métodos pertinentes:
        // int getDorsal(), String getNombre(), int getTiempoAculumado(),
        //String getCategoria().
}
```

1. Implementa el método `public T remove (T elem)` de la clase `LinearList<T>`.

   Analiza la complejidad de tu implementación. La valoración de la implementación puede disminuir si la complejidad es indebidamente ineficiente.

2. Define los atributos necesarios e implementa el método constructor de objetos de una clase `Pelotón` que sirva para representar una lista de objetos de clase `Ciclista`.

3. Asumimos que la precondición del siguiente método es que la lista de ciclistas está ordenada de **mayor a menor** valor de `tiempoAcumulado` por cada ciclista. Implementa el método

   `public LinearList<Ciclista>clasificacion(String cat)`,

   para tu clase `Pelotón`, que elimine de la lista los ciclistas de la categoria indicada en el parámetro `cat` y los devuelva en una lista ordenada por tiempos de **menor a mayor**.

   Analiza la complejidad de tu implementación. La valoración de la implementación puede disminuir si la complejidad es indebidamente ineficiente.

4. Asumimos que la precondición del siguiente método es que la lista de ciclistas está ordenada de **mayor a menor** valor de `tiempoAcumulado` por cada ciclista y que el entero `k` representa el número de categorías que aparecen en la lista. Implementa el método

   `public Ciclista[] mejorDeCadaCategoria (int k)`,

   para tu clase `Pelotón`, que debe obtener un array con el ciclista de cada categoría que menor tiempo ha acumulado.

   Analiza la complejidad de tu implementación. La valoración de la implementación puede disminuir si la complejidad es indebidamente ineficiente.

Valoración: 1. (3 puntos) 2. (1 punto) 3. (3 puntos) 4. (3 puntos)

PROBLEM 1. (5 points. 2.5+2.5)

The implementation of a list of integers is provided by the following Java code

```
public class ListInt {
	private Node first;
	private Node last;
	private int size;

	//TODO: the methods specified below

	static private class Node {
		public int value;
		public Node previous;
		public Node next;

		public Node(Node previous, int v, Node next){
			this.value = v;
			this.previous = previous;
			this.next = next;
		}
	}
}
```

- Your task is to include in the class `ListInt` the definition and Java code of the following methods:

```
public void sort(){
	// this method sorts the (nodes of the) list in ascending order
	// it is not allowed to use arrays or auxiliary lists
	// hint: adapt one of the algorithms already studied
	}

public void purge(){
	// this method deletes the odd nodes 1st, 3rd, 5th, ...,
}
```

- The next task is to analyze the time complexity of both methods.
- What would be the complexity of the *sort* method if instead of using a linked list we would have used an array implementation?

PROBLEM 2. (5 points)

A robot arm is in charge of the control of a scale (a balance). The robot tries to keep as balanced as possible the two pans of the scale while it is attending the elements that come sequentally by a conveyor belt. On the conveyor belt there are three different types of elements going to the robot: cylinders, cubes and baskets. Each cylinder must be stacked on the right pan of the scale and each cube must be stacked on the left pan. When the robot picks up a basket, it will remove elements from the pan that has more weight until the weight is less than or equal to the opposite pan.

Part of he behaviour of the robot is controlled by the following code

```
Robot myRobot = new Robot();
While (conveyorbelt.hasNext()) {
     myRobot.processElement (conveyorbelt.next ());
}
```

The program that simulates the robot uses integers to represent the elements going in the conveyor belt. A zero represents a basket. A positive number represents a cylinder whose weight is the integer value. A negative number represents a cube whose weight is the absolute value of the number.

Your tasks are:

to define the attributes in the class Robot for simulating its behaviour and to implement the method

```
public void processElement(int num) {}
```

This method should report the movements made by the robot, printing out the corresponding message to the standard output. That message shall also include a number indicating the element being processed. Below is a possible result of program execution:

```
placing cylinder weigth 17
placing cube weight 12
placing cylinder weight 21
removing cylinder weight 21
removing cylinder weight 17
```

Recursion

Infinite Recursion

This type of Recursion contains one option that is recursive, and one option that is not, this part is called the base case.

If we don't use a base case, the program will never end, so we call infinite recursion.

```
public int sum (int num)
{
  int result;
    if (num == 1)
        return num;
    else
    {
      result = num + Sum(num-1);
    }
  return result;
```

Recursion and the Call Stack

the call stack is the parameter that we pass to the recursion.

tail Recursion

A recursive function is a tail recursive when recursive call is the last thing executed by the function.

We can convert a recursive method to a tail recursive, for that we use an acumulator parameter.

A example of a tail recursive:

```
public static int sumInts(int n){
    return SumIntsWork(n, 0); //0 is the value that increments
}                             // the sum
private static int sumIntsWork(int n, int accum){
    if(n==0){ return accum;}
    if(n==1){ return accum+1;}
    accum = accum+n;
    return (SumIntsWork(n-1, accum);
}
```

Direct versus Indirect Recursion

Direct recursion is that a function call itself to continue the execution.

Indirect recursion occurs when a method invokes another method, to continue the execution.

Sometime, Indirect recursion can be more difficult to trace becaus of the intervening method calls.

Towers of Hanoi

By Recursion:

```
private void movetower(int numDisk, int start, int end, int temp){
    if(numDisks==1)
        Move oneDisk(start, end);
    }else{
        move tower(numDisk-1, start, temp, start);
        Move One Disk(start, end);
        move tower(numDisk-1, temp, end, start);
}
```

Comparable and Comparitors

If we want to sort any custom object array/list, we need to implement the comparable to provide default sorting and implement the Comparator to provide especific sorting.

falta, impago

We use the comperable we want to sort objects based on natural order, and the comparator when we want to sort on some other attribute of object.

Comparable

The comparable interface has a Single method, int compareto(To).

The comparable is used to ensure the natural ordering of objects in a collection.

If we want to use Arrays or Collections sorting methods we have to implement the method compareto (T obj)

Comparcto

This method compares this object with the object that we pass.

int Compareto (Object O_1)

And returned a int value. If return positive value this object is greater than O_1, if return negative value, this object is smaller than O_1, and if returns zero, there are equals.

In this function we do this equation:

return ($O - O_1$);

Comparator

Our purpose is to sort elements based on different parameters.

A comparator object is capable of comparing two different objects.

Comparators can be passed to a sort method to allow precise control over the sort order.

Compare

int compare (Object o_1, Object o_2)

This method compares o_1 and o_2 objects. Returns a positive value if o_1 is greater than o_2, a negative value if o_1 is smaller than o_2 and zero if o_1 and o_2 are equals.

Sorting

Sorting is the process of arranging a group of items into a defined order, either ascending or descending, based on some criteria.

There are two categories based on efficiency; Sequential sort, and logarithmic sort.

Sequential sort: selection sort, insertion sort, and bubble sort.

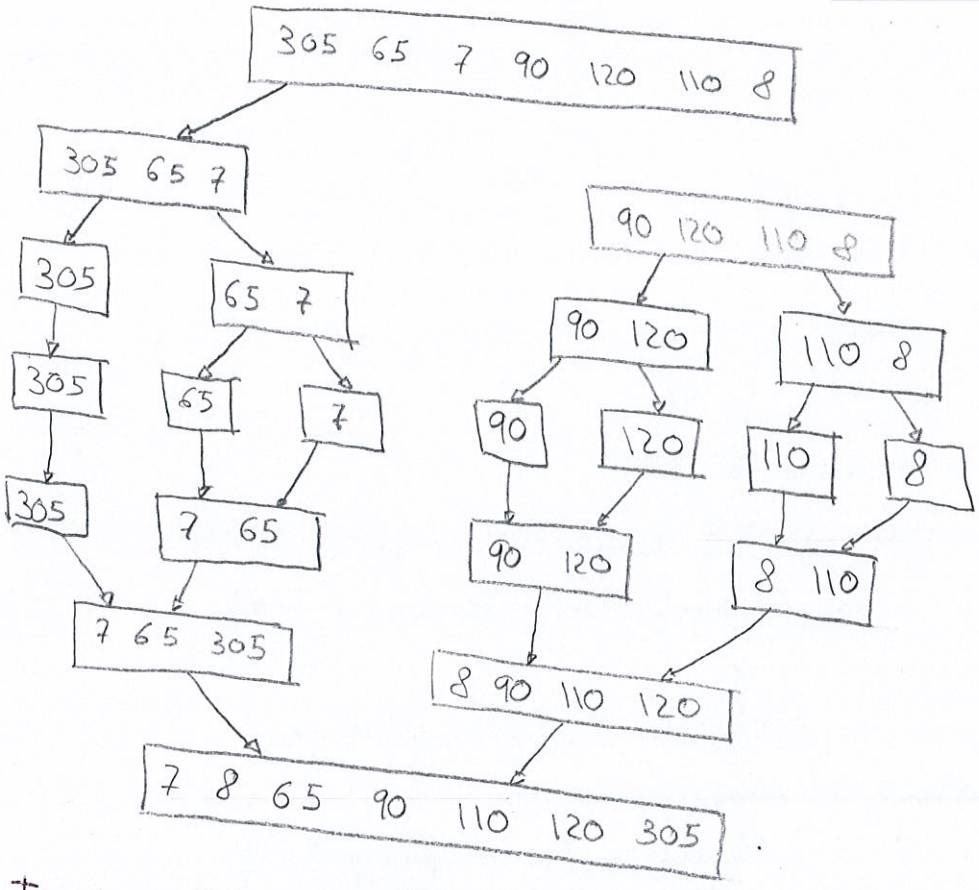
Logarithmic Sort: merge sort and quick sort.

Merge Sort

The merge sort alghorithm, sorts a list by recursively dividing the list in half, until each sublist has one element and then recombining these sublists in order.

The general strategy is to divide the list until we have lists of only one element, and then recombining this elements

with other element until we have again a only one list. this list will be sorted.



Implementation of the merge algorithm

```
public static void <T extends Comparable<? super T>> void
    mergeSort (T[] data, int min, int max)
        T[] temp;
        int index1, left, right;
        if (min == max)
            return;
        int size = max - min + 1;
        int pivot = (min + max)/2;
        temp = (T[]) (new Comparable[size]);
        mergeSort (data, min, pivot);
        mergeSort (data, pivot+1, max);
```

```
// copy sorted data into workspace
    for (index1 = 0; index1 < size; index1++)
        temp(index1) = data[min + index1];

// merge the two sorted lists
    left = 0;
    right = pivot - min + 1;
    for (index1 = 0; index1 < size; index1++) {
        if (left <= max - min)
            if (left <= pivot - min)
                if (temp[left].compareTo(temp[right]) > 0)
                    data[index1 + min] = temp[right++];
                else
                    data[index1 + min] = temp[left++];
            else
                data[index1 + min] = temp[right++];
        else
            data[index1 + min] = temp[left++];
    }
}
```

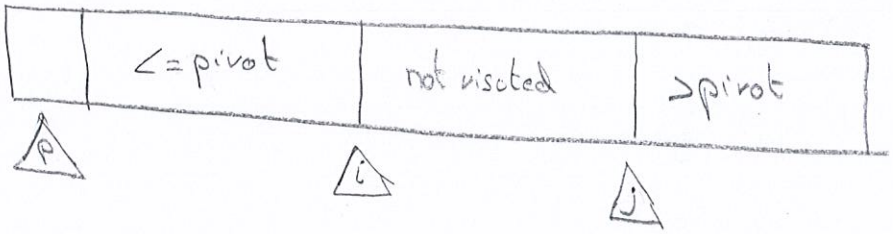
Quicksort

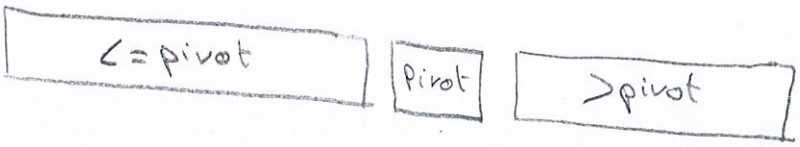
The Quicksort algorithm sorts a list by partitioning the list using an arbitrarily chosen partition element and then recursively sorting the sublists on either side of the partition element.

↳ Objects be sorted arbitrarily

the general estrategy of this algorithm is: get a element to act as a partition element. then partition the list to sat the elements that are smaller and greater than the partition element, and then apply this partition to both partitions.

At each step the pivot goes to its correct place:





Implementation of the Quick sort algorithm

```
public class QuicSort {
    private static <t extends Comparable<t>>
        int split(T[] list, int lo, int hi){
            int left = lo+1;
            int right = hi;
            T pivot = list[lo];
            while (true){
                while(left<=right) {
                    if (list[left].compareto(pivot)<0)
                        left++;
                    else{ break; }}
                while (right>left){
                    if (List[right].compareto(pivot)>0)
                        right--;
                    else { break; }}
                if(left >= right { break;}
                t temp = list[left];          // Swap left and right items
                list[left] = list[right];
                list[right] = temp; left++; right--;}
            list[lo] = list[left-1];
            list[left-1] = pivot
            return left-1;
}
```

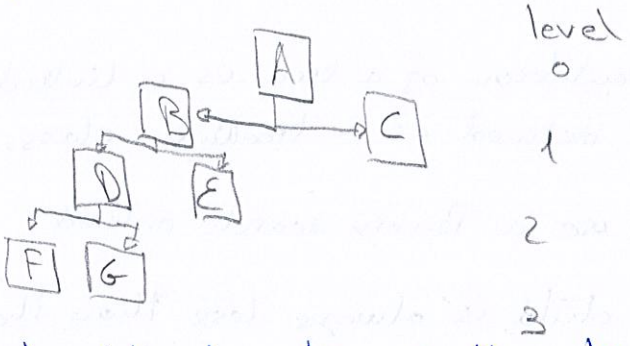
```
private static <T extends Comparable<T>>
  void Sort(T[] list, int lo, int hi){
      if((hi-lo)<=0)
          return;
      int splitPoint = split(list, lo, hi);
      Sort(List, lo, SplitPoint-1);
      Sort(List, splitPoint+1, hi);
  }

public static <T extends Comparable<T>>
    void sort(T[] list){
      if(list.length<=1)
          return;
      sort(list, 0, list.length-1);
  }
}
```

Trees

A tree is a nonlinear structure in which elements are organized into a hierarchy.

Conceptually, a tree is a group of nodes there are conected by edges, and are organized in a levels, the root of the tree is the only node at the top of the tree.

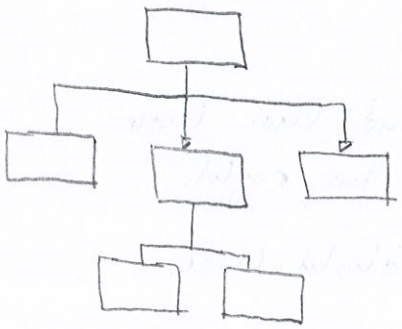
The nodes of the lower level of the tree are the children of the nodes at the previous level. this nodes are the parents. And if one node does not have any children is called leaf.



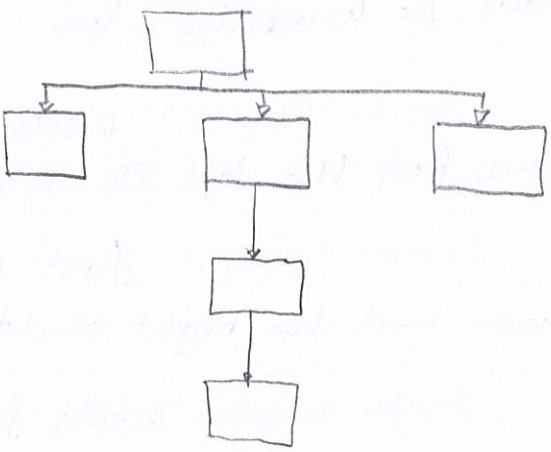
The height of a tree is the length of the longest path from the root to a leaf.

Balanced and Unbalanced trees

A tree is considered balanced if all of the leaves of the tree are on the same level or at least o within one level of each other.



Balanced



Unbalanced

Complete trees

A tree is considered complete if it is balanced and all of the leaves at the bottom level are on the left side of the tree.

We can say that a tree is a toll complete tree if all of the leave are in the same level and every node is eather a leaf or has n children.

Strategies of Implementing trees

The most obvious implementation of a tree is a linked structure. Each node could be defined as a treeNode class.

To analysi trees we can use a Binary search ordered list. In this tree the left child is always less than the parent and this is always less or equal to the right child. Then we could improve the efficiency of the find operation to $O(a \log n)$.

Trees traversal

If we want to travers a tree we have to go visiting all the nodes of the tree. So there are different methods for traverssing a tree.

- Preorder traversal visits the node, and then their childrens, first the left ~~the~~ children and then the right.

- Inorder traversal first visits the left child then the node and the right child.

- Postorder traversal visits first the children and then the node

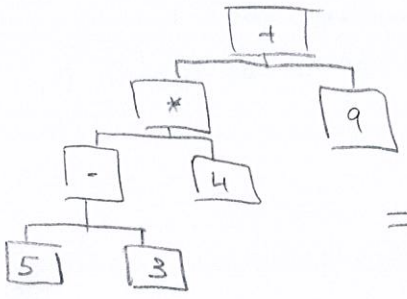
Level-order traversal visits all the nodes of each level, traversing all the nodes

A Binary tree ADT

Operation	Description
getRoot	Returns a reference of the root of the binary tree
isEmpty	Determines if the tree is empty
Size	Return the number of elements in the tree
conteins	Determines if the espesified target is in the tree
find	Return a refence to specified target if is found
toString	Returns a String representation of the tree.
iteratorInOrder	Returns an iterator for inorder traversal of the tree
iteratorPreOrder	Returns an iterator for preorder traversal of the tree
iteratorPostOrder	Returns an iterator for postorder traversal of the tree
iteratorLevelOrder	Returns an iterator for leverorder traversal of the tree

Using Binary trees: Expression trees

We use a postfix expression to contruct an expression tree using a Expressiontree class that extends our definition of binary tree. In this class the root and the internal nodes of an expression tree conteins operations and that all of the leaves contains operands.



$\Rightarrow (5-3)*4+9$

```
public class Expressiontree extends LinkedBinarytree<ExpressiontreeObj>
{
    public Expressiontree() {
        super();
    }
    public Expresiontree(ExpressiontreeObj element,
                Expressiontree leftSubtree, Expressiontree rightSubtree)
    {
      root = new BinaryTreeNode<T>(element);
      count = 1;
      if (leftSubtree != null) {
          count = count + leftSubtree.Size();
          root.left = leftSubtree.root;
      }
      else
          root.left = null;
     if ( leftSubtree != null){
          count = count + RightSubtree.Size;
          root.right = rightSubtree.root;
     }else
          root.right = null;
}

public int evaluatetree()
{
    return evaluateNode(root);
}
public int evaluateNode(BinaryTreeNode root) {
    int result, operand1, operand2; ExpressiontreeObj temp;
    if (root == null)
        result = 0;
    else
        temp = (ExpressiontreeObje) root.element;
```

```
if (temp.isOperator()) {
    operand1 = evaluateNode(root.left);
    operand2 = evaluateNode(root.right);
    result = computeTerm(temp.getOperator, operand1, operand2);
} else
    result = temp.getValue();
}

return result;
}

private static int computeTerm(char operator, int operand1,
                               int operand2){
    int result = 0;
    if (operator == '+')
        result operand1 + operand2;
    else if (operator == '-')
        result operand1 - operand2;
    else if (operator == '*')
        result operand1 * operand2;
    else
        result operand / operand;
    return result;
}
}
```

Implementing Binary trees with Links

The LinkedBinaryTree class implementing the BinaryTreeADT interface will need to keep track of (Pista jarraitu) the node that is at the root of the tree and the number of elements in the tree.

The constructors for the LinkedBinaryTree class should handle two cases:

we want to create a LinkedBinaryTree that is empty and a binary tree with a single element but not children.

```
/** Creates an empty binary tree */
public LinkedBinaryTree() {
    count = 0;
    root = null;
}

/** Creates a binary tree with one element that
 * is the root */
public LinkedBinaryTree(T element) {
    count = 1;
    root = new BinaryTreeNode<T>(element);
}
```

The find Method

If the target element is not found, ~~it~~ this method throws a exception.

If we write this method recursively, require a private support because the signature and/or the behavior of the first call and each successive call may not the same.

Implementation of the method find

```
public T find (T targetElement)
        throws ElementNotFoundException
    BinaryTreeNode<T> current = findAgain (targetElement, root);
    if (current == null)
        throw new ElementNotFoundException ("Binary tree");
    return (current.element);

public BinaryTreeNode findAgain (T targetElement,
                                 BinaryTreeNode next){
    if (next == null)
        return null;
    if (next.element.equals (targetElement)
        return next;
    BinaryTreeNode temp = findAgain (targetElement, next.left);
    if (temp == null)
        temp = findAgain (targetElement, next.right);
    return temp;
}
```

Binary Search tree

A binary search tree is a binary tree but, it is ordered, so the left child of a node is less than the node but that is less or equal of the right hand child.

Implementing binary search tree: with Links

Operation	Description
addElement	Add an element to the tree
removeElement	Remove an element from the tree
removeAllOccurrences	Removes all occurrences of element from the tree
removeMin	Removes the 'smallest' element from the tree
removeMax	Removes the largest element from the tree
findMin	Returns a reference to the minimum element in the tree
findMax	Returns a reference to the maximum element in the tree

The LinkedBinarySearchtree class offers two constructors: One create an empty LinkedBinarySeach tree and the other a LinkedBinarySearch tree that has a particular element at the root.

```
/** Creates an empty binary search tree */
public LinkedBinarySearchtree() {
    super();
}

/** Creates a binary search tree with the specified
 * element as its root */
public LinkedBinarySearchtree(T element) {
    super(element);
}
```

Implementing Binary Search trees with Ordered List

One of the uses of trees is to provide efficient implementations of other collections.

Using a binary seach tree, we can create an implementation called BinarySearchTreeList that is a more efficient implementation than those previously discussed.

Operation	Description
removeFirst	Removes the first element from the tree
removeLast	Removes the last element from the tree
remove	Removes a particular element from the tree
first	Examines the element at front of the tree
last	Examines the element at rear of the tree
contains	Determines if the trees contains a particular element
isEmpty	Determines if the list is empty
size	Determines the number of elements on the list
add	calls to the method addElement()

Graphs

Undirected Graphs

A graph is a made up of ~~vertices~~ nodes and the connections between those nodes, like a tree. In the graphs, we refer to the nodes as vertices. and refer to the connections among them as edges.

A graph is an undirected graph if the pairings representing the edges are unordered. So listing an edges as (A, B) means the same thing as listing the edges as (B, A) so they can be traversed in either directions.

Two vertices in a graph are adjacent if there is a connection, like a edge.

An undirected graph can be complete or uncomplete, is considered complete if it has the maximun connections between edges.

A path is a sequence of edges that connect two vertices. And the length of a path is the number of the vertices in that path.

If there is a edge between any vertices in a graph, we can say that the graph is connected.

Directed Graph

A directed graph or digraph, is a graph where the edges are ordered pairs of vertices. A path in this type of graphs is a sequence of directed edges that connect two edges.

Graph Algorithms

traversal

We can divide graph traversal into two categories: a breadth-first and depth-first traversal.

Breadth-traversal: we can use a queue to manage the traversal an unordered list to buil our result.

```
public Iterator<t> iteratorBFS(int startIndex){

    Integer x;
    LinkedQueue<Integer> traversalQueue = new LinkedQueue<Integer>();
    ArrayUnorderedList<t> resultList = new ArrayUnorderedList<t>();
    If(IndexIsValid(startIndex)
        return resultlist.Iterator();
    Boolean[] visited = new Boolean[numVertices];
    for(int i; i< numVertices; i++){
        visited[i]=false;
    traversalQueue.enqueue( new Integer(startIndex);
    visited[startIndex] = true;
    while (!traversalQueue.isEmpty()){
        x= traversetQueue.dequeue();
        result.addtoRear(vertices[x.intValue()];
//find all vertices adjacent to x that have not bean visited
//and enqueue
    for(inti=0; i< numVertices; i++){
        if(adjMatrix[x.intValue()][i] && !visited[i]){
            traversalQueue.enqueue(new Integer(i));
            visited(i) = true;
        }
    }
    return resultList.iterator();
}
```

Depth-first traversal: We can use a traversalstack to manage the traversal. And the vertex is not marked as visited until it has been added to the result List.

```
public IteratorT> iteratorDFS(int startIndex) {

    Integer X;
    LinkedStack<Integer> traversalStack = new LinkedStack<Integer>();
    ArrayUnorderedList<T> resultList = new ArrayUnorderedList<T>();
    Boolean[] visited = new Boolean[numVertices];
    if (!IndexIsValid(startIndex))
            return resultListIterator();
    for (int i=0; i< numVertices; i++)
            visited[i]=false;
// Depth-first traversal part
    TraversalStack.push(new Integer(startIndex));
    result.addToRear(vertices[startIndex]);
    visited[startIndex] = true
    while (!traversalStack.isEmpty()) {
        X= traversalStack.peek();
        found=false;
    //find a vertex adjacent to x that has not been visited and
    // push it on the stack
        for (int i=0; (i< NumVertices) && !found; i++) {
            if (adjMatrix[x.intValue()][i] && !visited[i]) {
                traversalStack.push(new Integer(i));
                resultList.addToRear(vertices[i]);
                visited[i]= true;
                found = true;
            }
        }
```

```
        if(!found && !traversalStack.isEmpty())
            traversalStack.pop();
    }
    return resultList.iterator();
}
```

Directed graph

In this type of graph the edges are ordered pairs of vertices. This means that the edges (A,B) and (B,A) are separate, directional edges in a direct graph

Constructor

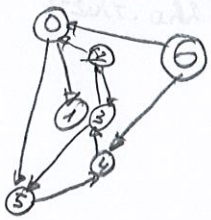
```
public Digraph(Digraph G){
    this(G.V());
    this.E = G.E();
    for(int v=0; v< G.V(); v++){
        Stack<Integer> reverse = new Stack<Integer>();
        for(int w : G.adj[v]){
            reverse.push(w);
        }
        for(int w : reverse){
            adj[v].add(w);
        }
    }
}
public addEdge(int v, int w){
    if(v<0 || v>=V) throw new IndexOutOfBoundsException;
    if(w<0 || w>=W) throw new IndexOutOfBoundsException;
    adj[v].add(w);
    E++;
```

```
public Digraph reverse(){
    Digraph R = new Digraph V();
    for(int v=0; v<V; v++){
        for(int w : adj(v)){
            R.addEdge(w,v);
        }
    }
}
```

the BFS and DFS traversal are identical to indirect graphs.

Cycles

DirectedCycle is a cycle formed by 3 or more edges, with this class we can find this type of cycles in our Graph.




```
public class DirectedCycle {
    private boolean[] marked;
    private int[] edgeTo;
    private Stack<Integer> cycle;
    private boolean[] onStack;
    public DirectedCycle(Digraph G){
        marked = new boolean[G.V()];
        onStack = new boolean[G.V()];
        edgeTo = new int[G.V()];
        for(int v=0; v<G.V(); v++)
            if(!marked[v]) dfs(G,v);
    }
```

```
//check that algorithm computes either the topological order
private void dfs(Digraph G, int v){
    onStack[v] = true;
    marked[v] = true;
    for(int w: G.adj(v)){
      //short circuit if directed cycle found
      if(cycle != null) return;
      //found new vertex, so recur
      else if(!marked[w]){
          edgeTo[w]= v;
          dfs(G, w);
      }
      //trace back directed cycle
      else if (onStack[w]){
        cycle = new Stack<Integer>();
        for( int x = v; x!=w; x= edgeTo[x]){
            cycle.push(x);
        }
        cycle.push(w);
        cycle.push(v);
      }
    }
    onStack[v] = false;
}
```

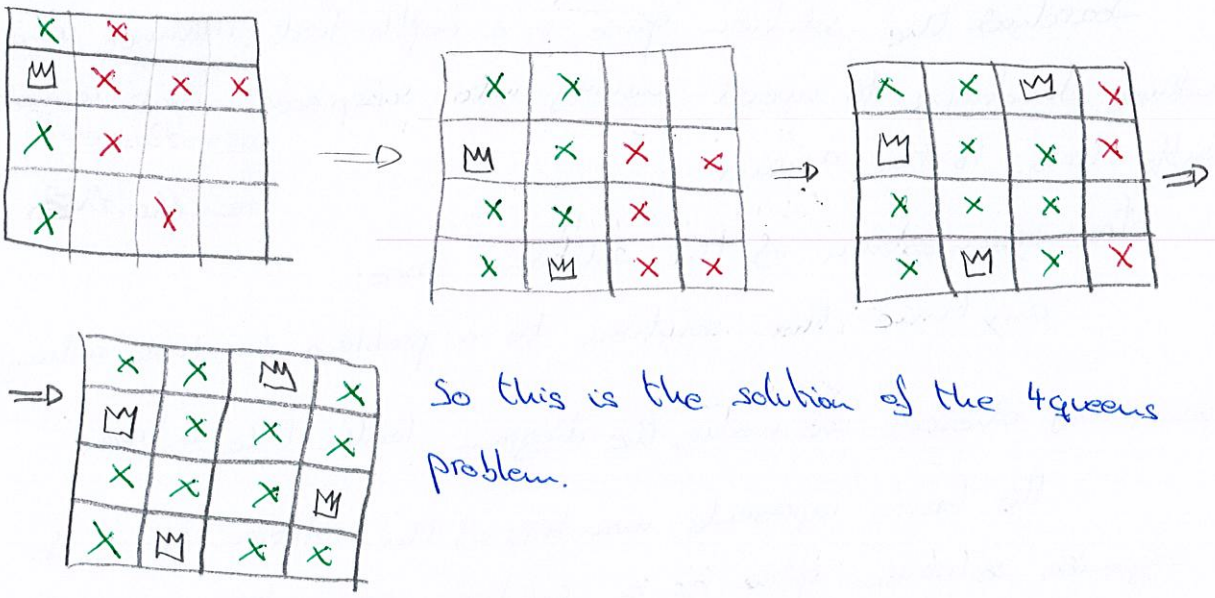
Backtracking

Backtracking is a systematic way to iterate through all the posible configurations of a search space.

These configurations may represent all posible arrangements of objects (permutations) or all posible ways of building a collection of them (subsets).

N-Queens

Is a classic combinatorial problem. The goal is to place eight queens on an 8x8 chessboard so that no two attack each other. So no two of them are on the same row, column or diagonal.



So this is the solution of the 4queens problem.

If we diagram the sequence of choice we make, the diagram looks like a tree. The edges are labeled by posible values x_i.

For the case of 4x4 chessboard there are $4! = 24$ leaf nodes in the tree

Backtracking is a refinement of the brute force approach, which systematically searches for a solution to a problem among all avalible choices.

We build a vector $(x_1, x_2, \ldots, x_i)$ from a partial solution and try to extending it we will test whether what we have so far is still possible as a partial solution.

Steps of the backtracking method

Define a solution space of the problem. if S_i is the domain of x_i then $S_1 \times S_2 \times \ldots \times S_n$ is the solution space of the problem

Organize the solution space so that it can be searched easily.

Searched the solution space in a depth-first manner using bounding functions to avoid moving into subspaces that cannot possibly lead to the answer

tree organization of the solution space;

anytime the solution to a problem involves making sequence of choices we make, the diagram looks like a tree

the leaves represents members of the solution space. So we organize solution space as a solution space tree.

Structure of backtracking

In a combinatorial search, we represent our configurations by a vector $A = (a_1, \ldots, a_n)$: where each element a_i is selected from an ordered set of possible candidates S_i for position i.

The search procedure works by growing solutions one element at a time

At each step a partial solution $(a_1, \ldots, a_k)$ is constructed.

A candidate set S_{k+1} for position $(k+1)$ is defined

try to extend the partial solution by adding the next element from S_{k+1}.

So long as the extension yields a longer partial solution we continue to try to extend it.

At some point, $S_{k+1} = \emptyset$, if so, we must backtrack and replace a_k, the last item in the solution value, with the next candidate is S_k.

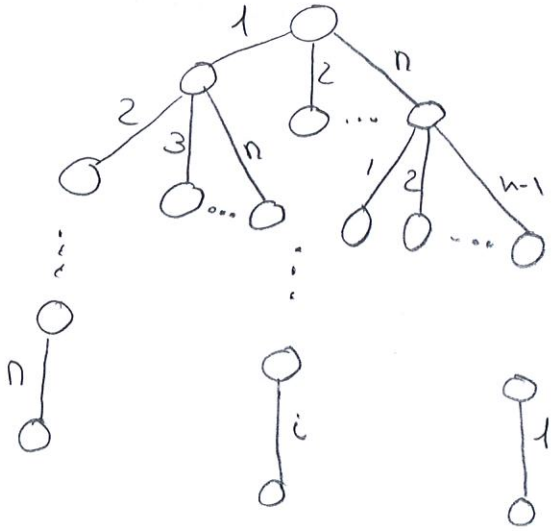
Recursive Backtracking scheme

```
Backtrack-DFS(a, k)
    if A = (a1, a2, ..., ak) is a solution, report it.
    else
        k = k+1;
        compute Sk.
        while Sk ≠ ∅ do
            ak = an element in Sk
            Sk = Sk - ak
            Backtrack-DFS(A, k)
```

Constructing the Permutation tree

When the problem asks for an n-element permutation that optimizes some function, the solution space tree is a permutation tree

Set up an array of n elements, where the value of x_i is an integer from 1 to n which has not appeared thus far in the array, corresponding to the ith element of the permutation. There are n! permutations.



Improving Backtracking by pruning

Make use of the bounding function and the constraint function to prune invalid branches and to focus the search on branches that appear most promising

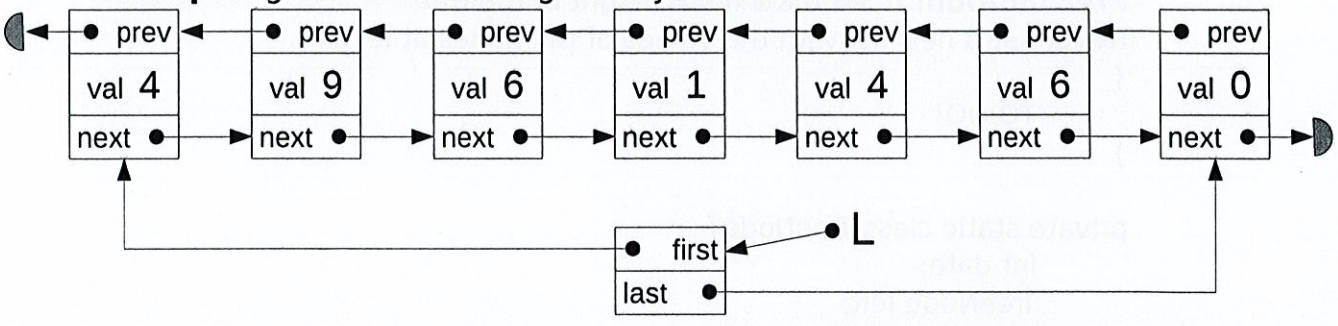
Data Structures and Algorithms

Final Exam (18th January 2016)

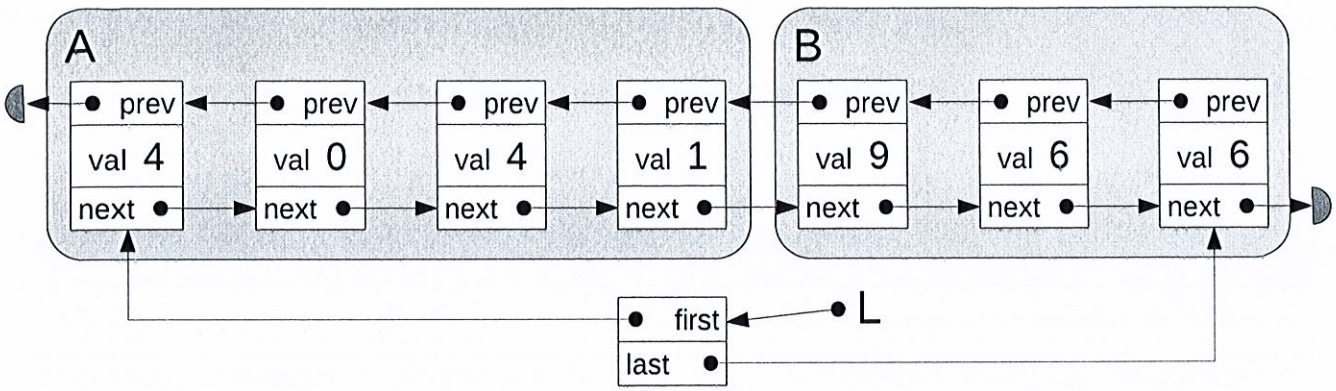
1. **[3 points]** Your task is to implement the method "*partition*" in the following implementation of doubly linked lists of integers, in such a way that the running time in the worst case be O(n) (n being the number of elements of the list). Explain briefly your answer.

```
class LinkedListOfInt{
        private Node first;
        private Node last;

        public void partition(int v)
        Precondition: none (the integer v can be in the list or not)
        Postcondition:
            1. the content of the list is a permutation of the original list
            2. the values lower than v are located before than the values greater
               or equal to v (see examples below).
        Note: it is not allowed to create new nodes, nor arrays or other objects.
        {
                TO DO!
        }

        private static class Node{
                int data;
                Node previous;
                Node next;
        }
}
```

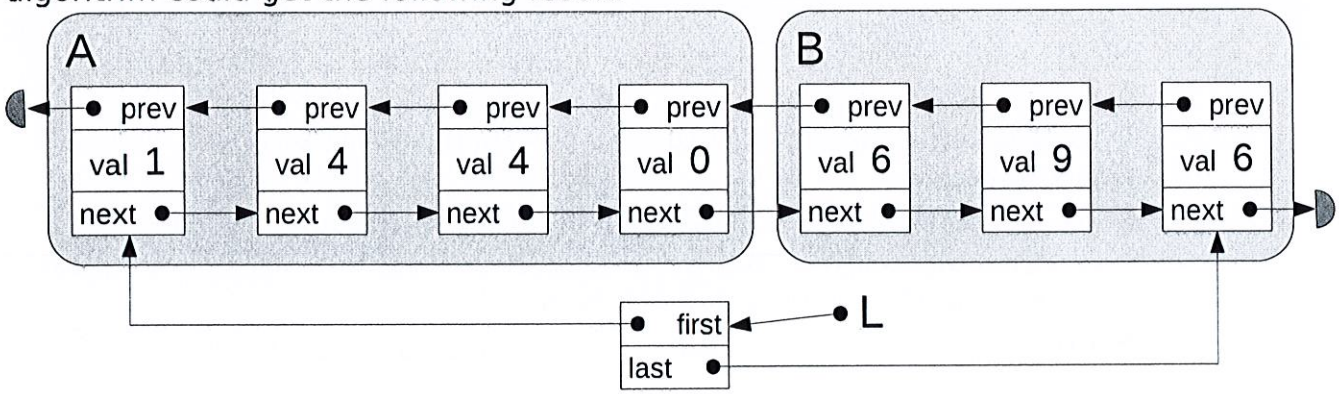
Example: given the following doubly linked list **L**:



after calling *L.partition(6)* the list **L** could be like the figure below, with all elements lower than 6 in the zone A, and those greater or equal to 6 in zone B.



It doesn't matter the internal order in zones A and B. Another different algorithm could get the following result:



2. **[3 points]** Your task is to implement the method *valuesAtLevel* in the following implementation of binary trees. You should also analyze the running time in the worst case (explain briefly your answer) .

```
class BinaryTree{
        private TreeNode root;

        public List<Integer> valuesAtLevel(int n)
        Precondition: n >= 0 && n <= height of the tree
        Returns: a new list with the values of the nodes at level n.
        {
                TO DO!
        }

        private static class TreeNode{
                int data;
                TreeNode left;
                TreeNode right;
        }
}
```

3. **[2 points]** An undirected graph G=(N,A) is called red-black if each node n is painted red or black but with the condition that no two adjacent nodes are painted in the same color. Your task is to write an algorithm with running time O(n+a) (being n the number of nodes and a the number of edges) to determine if a graph is red-black. If a graph is red-black then the algorithm should output the color of each node.

4. **[2 points]**

   A pastry workshop has a number of master bakers, numbered 0, 1, 2 ... n. Each baker (pastry cook) is dedicated to the development of a different cake.

   There are also a number of tools (numbered 0, 1, 2 ...) each of which can be used by a baker or another, depending on each tool.

   Over a working day, each baker should make a single pie, for which employs an amount of time that depends on what tool he/she uses. As each tool can only be used once per day, you need to decide which tool uses each baker.

   Write a program that, given the tools used by each baker, and the time taken by each baker with each tool, *outputs a mapping from tools to bakers such the sum of the times taken by the bakers is minimum.*

Implementing trees

Tree traversal refers to the process of visiting each node, once.

There are four basic methods:

- Preorder traversal, visit each node followed by its children.

- In Order traversal, visit the left child of the node, then the node, and finally the right child.

- Post Order traversal, visit the children and then the node.

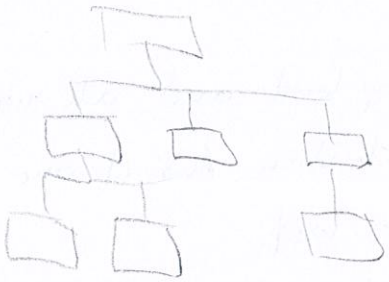
- Level-order traversal, visit all of the nodes at each level, and the continue with the other levels.

trees

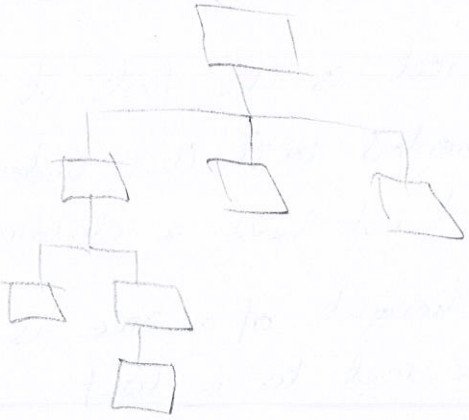
A tree is a nonlinear structure in which elements are organized into a hierarchy. And is composed of a set of nodes. Each node is a particular level in the tree hirachy.

The nodes in a lower level are childrens of nodes at the previous level.

A tree is a balanced tree if all of the leaves, are in the same level or at least within one level of each other.

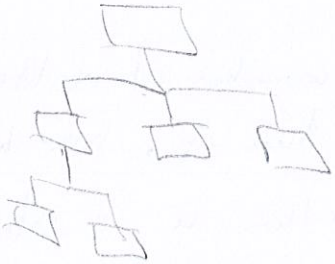


Balanced tree

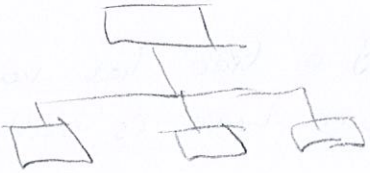


Complete trees

A tree is considered complete if it is balanced and all of the leaves at the bottom level are on the left-slide. And full if is complete and the leaves of the tree are at the same level and every node is either a leaf or has exactly n children.



Complete tree



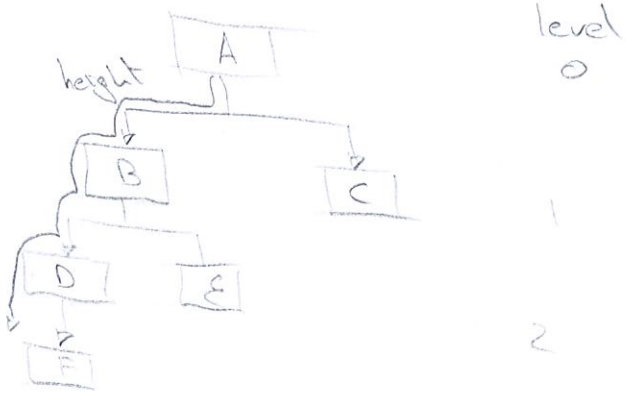
Complete, full tree

trees

A tree is a nonlinear structure in which the nodes are organized into a hierarchy. Is composed with different nodes that are connected, and there are different levels.

the root is the first it is in the top level and all nodes are connected to it. this nodes are called children. And if one node does not have a children, is called a leaf.

the height of a tree is the lenght of the longest path from the root to a leaf.



Balanced and Unbalanced trees

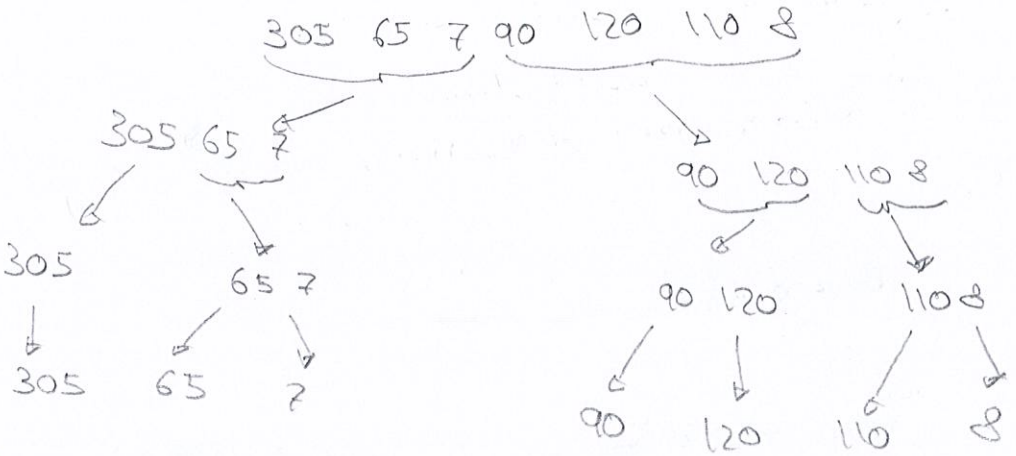
If a tree has no limit to the number of children a node may have, is called general tree. And if a tree has limited each node children to no more than n is referred to as an n-ary tree.

A tree is a binary tree if nodes have at most two children

Sorting (Part 2)

Merge Sort

This algorithm, divide recursively a list in half until each sublist has only one element, and then recombine this list in order.

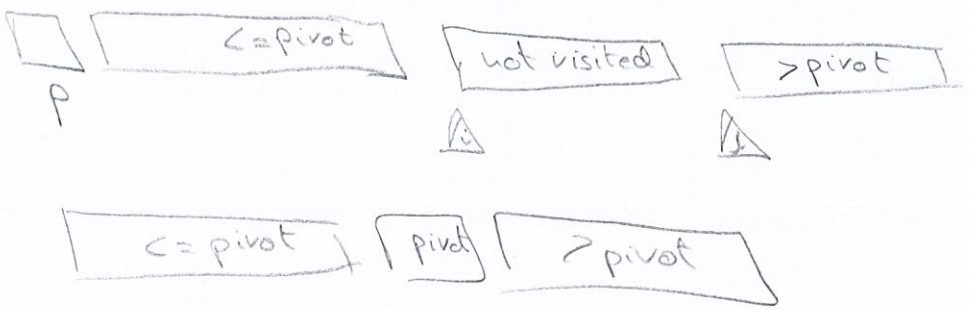


Then back up the recursive calling, and we compare the list with the list we are going to link, until we have only one list.

Quicle Sort

This algorithm sorts a list by partitioning the list using an arbitrary chosen partition element and then ~~sublist~~ sort the sublist recursively.

We have a pivot



Direct and Indirect Recursion

Direct recursion occurs when a method calls itself.

Indirect recursion occurs when a method calls other method.

Towers of Hanoi

```
private void moveTower(int numDisks, int start, int end, int temp){
    if (numDisks == 1)
        moveOneDisk(start, end);
    else{
        moveTower(numDisks-1, start, temp, end);
        moveOneDisk(start, end);
        moveTower(numDisks-1, temp, end, start);
    }
}

private void moveOneDisk(int start, int end){
    System.out.println("Move one disk from "+start+" to "+end);
}

public void solve()
    moveTower(totalDisks, 1, 3, 2);
```

```
public void toString( Expressiontree tree){
    {
    if( tree.isEmpty ){
        System.out.println("the tree is empty ");

    }else{                                    evalueNode(root.left)
                                                          evalueNod.
                                                          (root.right)
        System.out.println(" ("+ tree.root.getOperator + ")"

    }
```

```
public String evalueNode(BinaryTreeNode<T> current){
```

(B)

```
public Integer floor(int S){
    if (root == null){
        return null;
    else if (root.value.equal(S))
        return (Integer) root.value;
    else if (root.value > S)
        return findFloor(root.left, S);
    else
        if (findFloor(root.Right, S) == null)
            return root;
        else return findFloor(root.Right, S);
    return null;
}

private int findFloor (Node current, int S);
    if (current == null){
        return null;
    else if (current.value.equals(S)){
        return current.value;
    else if (current.value > S){
        return (findFloor(current.left, S)){
    else
        if (findFloor(current.Right, S) == null)
            return current.value
        else return findFloor(current.Right);
```

Exam 2014

```
public int leaves (int level) {
    int n = 0;
    int m = 0;
    if (root == null) {
        return 0;
    } else if (root.left == null && root.right == null) {
        if (level == 0)
            return 1;
        else
            return 0;
    } else
        n = countLeaves (root.left, level-1);
        m = countLeves (root.right, level-1);
        return n+m;
}

private int countLeves (Node current, int level) {
    if (current == null) {
        return 0;
    } else if (current.left == null && current.right == null) {
        if (level == 0) {
            return 1;
        } else {
            return 0;
        }
    } else {
        int n = countLeaves (current.left, Level-1);
        int m = countLeves (current.right, Level-1);
        return n+m;
    }
```

```
private Boolean childS (BTNode current) {
    BtNode current1 = null; Current2 = null;
    if ( current.left == null && current.right == null) {
        if ( current.content == 0)
            return true;
        else
            return false;
    }
    if (current.left != null && current.right == null) {
        current1 = current.left;
        if (current1.content.equals(current.content)){
            return childS (current1);
        else
            return false
    }
    if ( current.right != null && current.left == null) {
        Current1 = current.right;
        if ( current1.content.equals(current.content)
            return childS ( current1);
        else
            return false;
    }
    if ( current.right != null && current.left != null ){
        current1 = current.left;
        current 2 = current.right;
        int sum = current1.content + current2.content;
        if( current.content.equals(sum))
            if( current.childS(current1) && current.childS(current2)
                return true;
            else
                return false
    }
}
return false;
```

Problem 2

```
public boolean childSum() {
    if (root.isEmpty && root.content != 0) {
        return false;
    if (root.left != null && root.right == null) {
        BtNode current1 = root.left;
        if (current1.content.equals(root.content)) {
            return childS(current1);
        } else {
            return false;
        }
    }
    if (root.left == null && root.right != null) {
        BtNode current2 = root.right;
        if (current2.content.equals(root.content)) {
            return childS(current2);
        } else {
            return false
        }
    }
    if (root.left != null && root.right != null) {
        BtNode current3 = root.left;
        BtNode current4 = root.right;
        int sum = current3.content + current4.content;
        if (root.content.equals(sum)) {
            if (childS(current3) && childS(current4)) {
                return true;
            } else {
                return false;
            }
        return false
    }
return false;
```

Exam 2013

Problem 1

4 + 3 + 10

```
List<Integer[]>
public static ~~Integer[]~~ getList() {
    List<Integer[]> treeList = new ~~List~~ ArrayList<Integer[]>();
    if(root.IsEmpty)
        return treeList;
    if(root.left != null)
        getElements(root.left, treeList);
    treeList.add(root.content);
    if(root.right != null)
        getElements(root.right, treeList);
}
return treeList
}

private void getElements(BTNode current, List<Integer[]> list)
    if(current.left != null)
        getElements(current.left, List);
    List.add(current.content);
    if(current.right != null)
        getElements(current.right, List);
}
```

Binary trees

(1)

```
/** Returns true if an element elem is in the tree, and
 * appears exactly n times. */

public boolean appears (T elem, int n) {
    int m = 0;
    if ( root.left == null && root.right == null)
        return false;
    m = appearesN(elem, root.left);
    if ( root.content == elem.content) {
        m = m + 1;
    }
    m = m + appearesN(elem, root.right);
    if( m == n)
        return true;
    return false;
}

private int appearsN( T elem, BinaryTreeNode<T> current);
    int m = 0;
    if (current.content == elem.content)
        m = m+1;
    if (current.left == null && current.right == null){
        return m;
    if (current.left != null)
        m = m + appearsN (elem, current.left);
    if (current.right != null)
        m = m + appearsN ( elem, current.right);
    return m;
```

Data Structures and Algorithms
Exam Midterm II (18th-November-2015)

Problem 1 [10 points; 2.5 points each method]

An implementation of Binary Search Trees is based on the concept of "*lazy deletion*", and it is handy when the number of deletions is small. To delete an element, we merely mark it "deleted" instead of removing the node. The node that is being deleted remains in the tree, but it is marked as "deleted". In the same way, when an element is added to the tree, we check wether the element was previously marked as "deleted". In that case, the mark "deleted" is changed to "undeleted", without creating a new node.
Your task is to complete the methods of the following implementation of LazyBST with lazy deletion. You must indicate the running time of your implementation in terms of O().

```
public class LazyBST<T> {
    /*The instances of this class are BSTs with lazy deletion*/
    private final Comparator<? super T> comparator;
    private Node root=null ; /*null if the tree is empty*/
                /*do NOT ADD more instance variables*/
    public LazyBST(Comparator<? super T> comparator){
        this.comparator = comparator;
    }
    public boolean add(T value){} /* The specified "value"
    is added to the BST. If it was marked as "deleted" it is
    marked as "undeleted", without creating a new node. This method
    returns True if the value was not present in the tree or if
    was marked as "deleted". Otherwise, this method returns False
    */

    public boolean remove(T value){} /* the value that is
    passed in as a parameter is removed from the tree. If the
    node was marked as "undeleted", it is marked as "deleted"
    without removing the node. This method returns True if the
    value was in the tree and if it was marked as "undeleted";
    otherwise, it returns False. */

    public int size() {} /* this method returns the number
    of elements in the tree without including the nodes marked
    as deleted. This method must be recursive */

    public List<T> getLeaves() {} /* this method returns the
    list with all the leaves of the tree without including the nodes
    marked as deleted */

     static private class Node<T>{
         public T value;
         public Node left; /*lower*/
         public Node right; /*greater*/
         public boolean deleted; /* if True,
                 the node is marked as "deleted"*/
         public Node (Node left, T value, Node right){
             this.value = value;
             this.left = left; this.right=right; deleted= false;
         }
     }}
```

```
public List<T> getLeaves(){
    List<T> leavesList = new ArrayList<T>();
    if(!(root == null)){
        return getLeavesWork(root, leavesList);
    }
    return leavesList;
}

protected List<T> getLeavesWork(Node<T> current, List<T> leavesList){
    if(current == null){
        return leavesList;
    else if(current.left == null && current.right == null){
        if(current.deleted == false){
            leavesList.add(current.value);
        return leavesList;
    else{
        leavesList = getLeavesWork(current.left, leavesList);
        return getLeavesWork(current.right, leavesList);
    }
}
```

```
public int size () {
        if (root == null) {
            return sizeWork (root);
        }else{
            return 0;
        }
}
```

```
protected int sizeWork (Node<T> current) {
        if (current == null) {
            return 0;
        }else{
            if (current.deleted == false) {
                return (1+ sizeWork(current.left) + sizeWork(current.right));
            }else{
                return (sizeWork(current.left) + sizeWork(current.right));
            }
        }
}
```

```
public boolean remove(T value) {
        if (!(root == null)) {
            return removeWork(root, value);
        }
        return false;
}
```

```
protected boolean removeWork(Node<T> current, T value) {
        if (current == null) {
            return false;
        } else if (((Comparable) current.value).compareTo((Comparable) value) > 0) {
            return removeWork(current.left, value);
        } else if (((Comparable) current.value).compareTo((Comparable) value) < 0) {
            return removeWork(current.right, value);
        } else {
            if (current.deleted == true) {
                return false;
            } else {
                current.deleted = true;
                return true;
            }
        }
}
```

Aitzol Elu Etxano

```
public boolean add(T value){

    if(!(root == null)){
        return addWork(root, value);// Call this method to traverse
    }                                // the tree
    root = new Node(null, value, null);

/** traverse the tree until find the value, if it does not find the value
in the tree, return true, else if it finds but is marked as "deleted"; will return true
and will mark as "undeleted", else will return false
public boolean addWork(Node current, T value){

    if( current == null)
        current = new Node (null, value, null);
        return true;
    }else if(((Comparable) current.value).compareTo((Comparable) value)>0){
        return addWork(current.left, value);
    }else if(((Comparable) current.value).compareTo((Comparable) value)<0){
        return addWork((current.right, value);
    }else{
        if( current.deleted == true){
            current.deleted = false;
            return true;
        }else{
            return false;
        }
    }
```