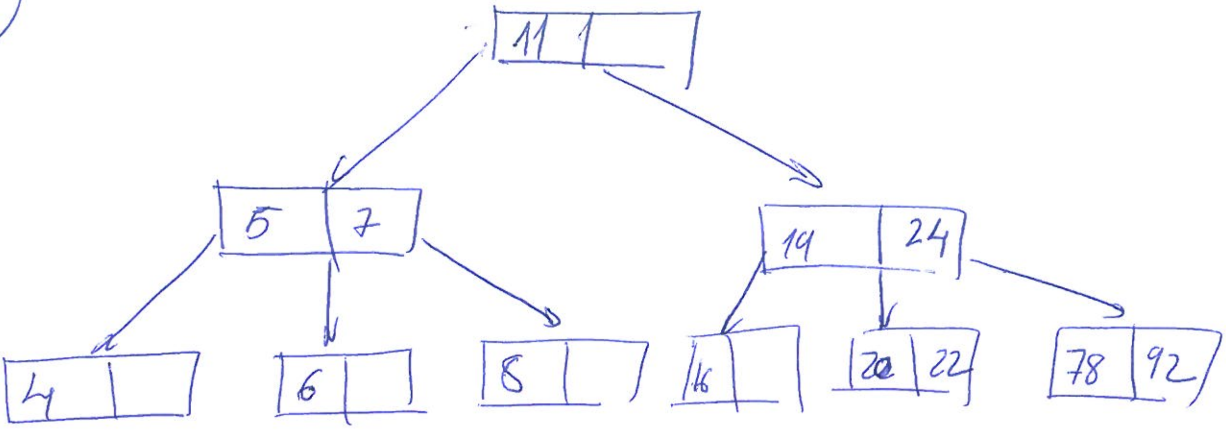
1.1



1.2

* Caso mejor O(1)

* Caso medio O(1)

* Caso peor O(n)

* No depende de la estrategia de resolución de colisiones

1.3 ~~T(79) = 1~~

~~T~~

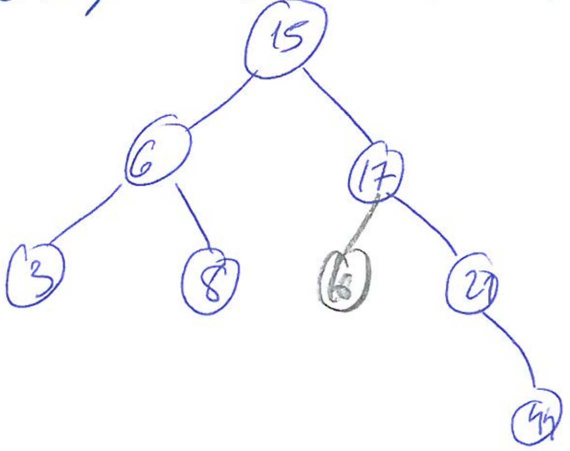
h(79, 0) = 1

h(73, 0) = 8

h(48, 0) = 7

h(11, 0) = 11

1.4 8, 3, 6, 21, 15, 17, 16, 44



1.5 Indices => ORDENADOS!!!

Estructura	Búsqueda	Inserción	Actualiz.
Vector	O(lg n)	O(n)	O(n)
Lista	O(n)	O(1)	O(n)
Hash	O(1)	O(1)	O(1)
AVL	O(lg n)	O(1)	O(lg n)

Ejercicio 2.a

```
public class ArrayListBinaryTree<E> implements BinaryTree<E> {
   private List<BTPos<E>> tree; // indexed list of tree positions
   private Set<Integer> holes;  // indexed set of holes

   /** default constructor */
   public ArrayListBinaryTree() {
      this.tree = new ArrayList<BTPos<E>>();
      this.holes = new HashSet<Integer>();
   }
...
}

public class BTPos<E> implements Position<E> {
   E element; // element stored at this position
   int index; // index of this position in the array list
   int left, right, parent;

   public BTPos(E elt, int i) {
      element = elt;
      index = i;
      left = -1;
      right = -1;
      parent = -1;
   }
...
}
```

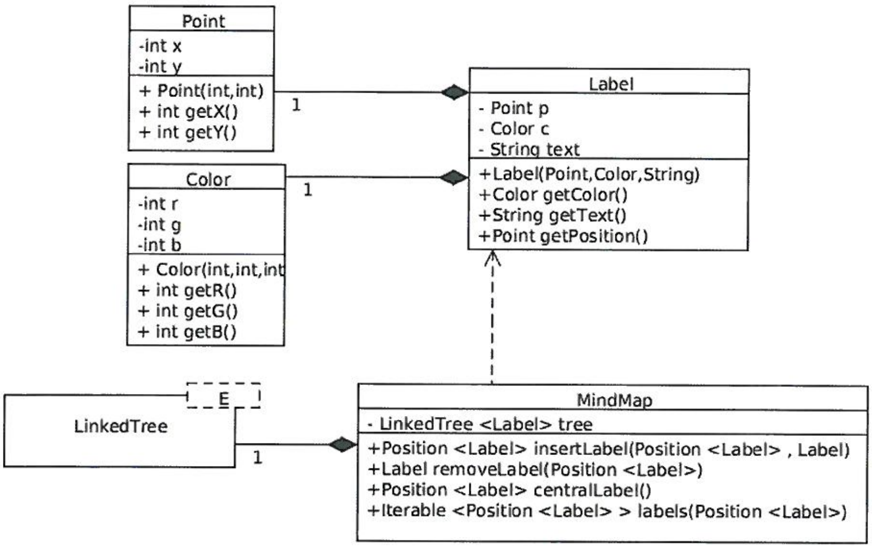
Ejercicio 2.b

```
   private void recursiveRemoval(BTPos<E> v) throws InvalidPositionException,
         BoundaryViolationException {
      this.holes.add(v.index());// lo añado al conjunto
      this.tree.add(v.index(), null); // Lo borro

      if (this.hasLeft(v)) {
         recursiveRemoval((BTPos<E>) this.left(v)); // recurse on left child
      }
      if (hasRight(v))
         recursiveRemoval((BTPos<E>) this.right(v)); // recurse on right child
   }

   /** Removes a subtree. */
   public E removeSubtree(Position<E> p) throws InvalidPositionException,
         BoundaryViolationException {

      // recopilo informacuion familiar
      BTPos<E> nodeToDelete = checkPosition(p);
      int parentIndex = nodeToDelete.getParent();
      BTPos<E> parentNode = tree.get(parentIndex);
      int indexToDelete = nodeToDelete.index;

      // Actualizo el padre
      if (parentNode.getLeft() == indexToDelete)
         parentNode.setLeft(-1);
      else
         parentNode.setRight(-1);
      this.recursiveRemoval(nodeToDelete);

      return nodeToDelete.element;
   }
```

Ejercicio 2.c

```
/** Inserts a left child at a given node. */
public Position<E> insertLeft(Position<E> p, E e)
      throws InvalidPositionException {
   BTPos<E> parentNode = checkPosition(p);
   int left = parentNode.getLeft();
   if (left != -1)
      throw new InvalidPositionException("Node already has a left child");
   int i = this.tree.size();
   if (this.holes.size() != 0) {
      i = this.holes.iterator().next();
   }

   BTPos<E> newNode = new BTPos<E>(e, i);
   newNode.setParent(parentNode.index());
   parentNode.setLeft(newNode.index);
   if (this.holes.size() == 0) {
      this.tree.add(newNode);
   } else {
      this.tree.add(i, newNode);
   }
   this.holes.remove(i);
   return newNode;
}
```

Ejercicio 3.a



Ejercicio 3.b

```
public class MindMap {

   private LinkedTree<Label> mindMap;

   public MindMap() {
      this.mindMap = new LinkedTree<Label>();
   }

   public Position<Label> insertLabel(Position<Label> p, Label l){
      return this.mindMap.add(l, p);
   }

   public void removeLabel(Position<Label> p){
      this.mindMap.removeNode(p);
   }

   public Position<Label> centralLabel() throws
ejercicio3.jun2014.EmptyTreeException{
      return this.mindMap.root();
   }

   public Iterable<Label> lablels(Position<Label> p){
      List<Label> labs = new ArrayList<Label>();
      List<Position<Label>> q = new ArrayList<Position<Label>>();
      Label l = p.element();

      labs.add(l);
      q.add(p);
      while(!q.isEmpty()){
         Iterable<? extends Position<Label>> children = this.mindMap.children(p);
         q.addAll((Collection<? extends Position<Label>>) children);
         for(Position<Label> pos : children){
            labs.add(pos.element());
         }
         q.remove(p);
      }
      return labs;
   }

}
```