Campos
AVLTree<E, AVLTree<E,E>> family;
int currentSize;

Métodos
int size()
- return currentSize O(1)

boolean contains(E element)
- family.find(element.representative()) O(log n)
  + se encontrar, faz .find(element) O(log n)
    » se encontrar, retorna true O(1)
- se não encontrar, em qualquer dos finds, retorna false O(1)

void insert(E element) throws ExistingElementException
- family.find(element.representative()) O(log n)
  + se não encontrar, faz family.insert(element.representative(), new AVLTree<E,E>()) O(log n)
  + se encontrar, faz .find(element) O(log n)
    » se não encontrar, faz .insert(element, element) O(log n)
    » se encontrar, lança excepção O(1)

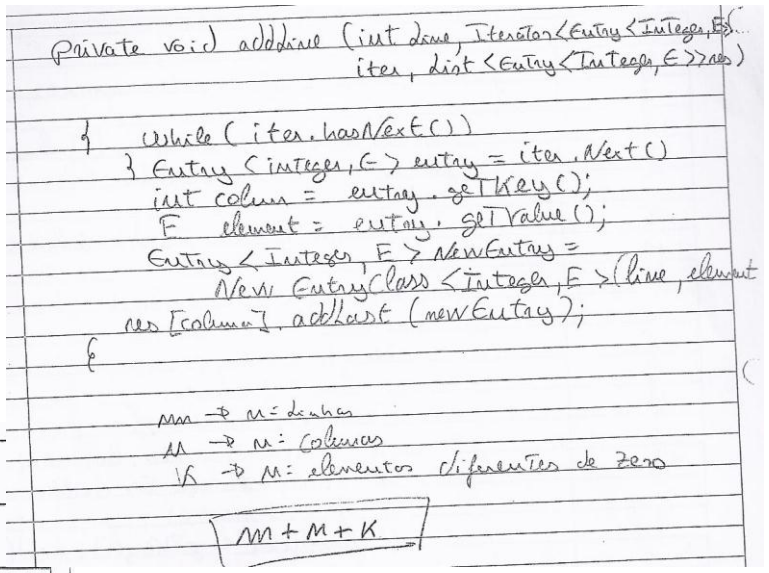
void remove(E element) throws NoSuchElementException
- se currentSize for 0, lança excepção
- family.find(element.representative()) O(log n)
  + se não encontrar, lança excepção O(1)
  + se encontrar, faz .remove(element) O(log n)
    » se não remover (devolver null), lança excepção O(1)
    » se remover, caso a família respectiva fique com 0 elementos, fazer family.remove(element.representative()) O(log n)

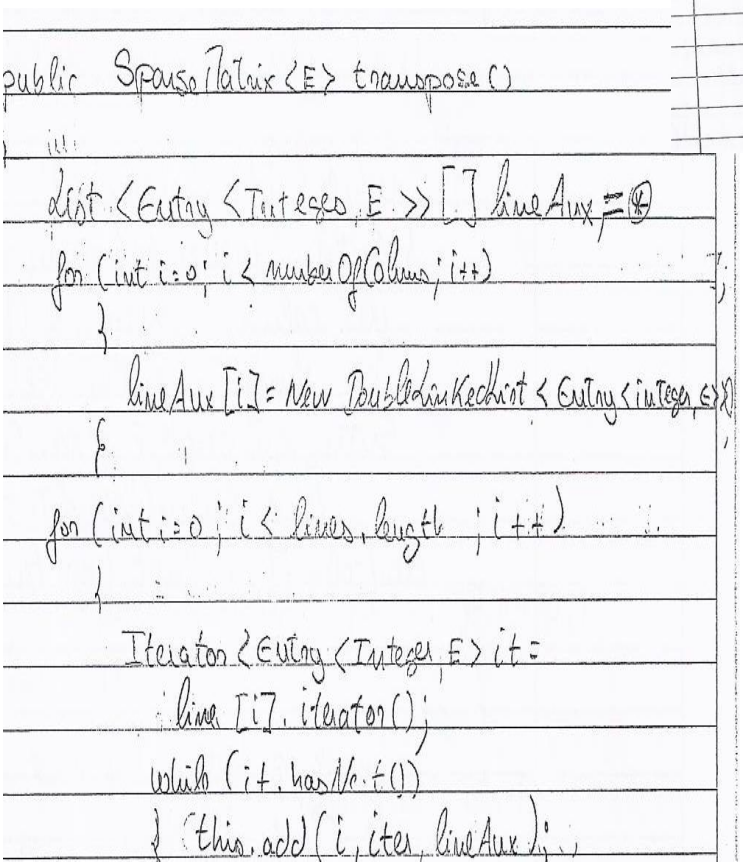
Iterator<E> sameFamily(E element)
- family.find(element.representative()) O(log n)
  + se não encontrar, retorna null O(1)
  + se encontrar, retorna .iterator() encapsulado para devolver só a chave O(log n)

Com encapsulado quero dizer um Iterator<E> que recebe esse iterador base no constructor e cujos métodos hasNext(), rewind() e next() se limitam a chamar os métodos correspondentes desse iterador, com a excepção deste último que devolve só a chave da entrada que for a seguinte no iterador base.

Complexidades
Tendo em conta as anotações que fiz nos métodos e somando tudo...
- size() » O(1)
- contains() » O(log n)
- insert() » O(log n)
- remove() » O(log n)
- sameFamily() » O(log n)

- **f(n) = O(1) <=> O(n^k) = O(1) <=> n^k = 1 <=> k = log(base N) 1 <=> k = 0**





Exercicio com iteradores e listas/DIcionarios

```
public void noNameInitMem(long[] vector){
        for(int i = 0; i < vector.lenght; i++)
            vector[i] = -1;
}
public long noNameMem(int n){
        long[] memory = new long[n+1];

        noNameInitMem(mem);
        return noNameMem(mem, n);
}
private static long noNameMem( long[] mem, int n ){
        if(mem[n] == -1)
            if ( n == 0 || n == 1 )
                mem[n] = 2 * n + 1;
            else{
                long prod = 1;
                for ( int i = 0; i < n / 2; i++ )
                mem[n] = prod * ( noNameMem(mem, n - i
- 1) % noNameMem(mem, i) + 1 );
                pro = mem;
            }
        return mem[n];
}
```

```
public static long noName( int n ) throws
NegativeNumberException{
if ( n < 0 )
throw new NegativeNumberException();
return noNameRec(n);
}
private static long noNameRec( int n ){
if ( n == 0 || n == 1 )
return 2 * n + 1;
else{
long prod = 1;
for ( int i = 0; i < n / 2; i++ )
prod = prod * ( noNameRec(n - i - 1) % noNameRec(i) + 1);
return prod;
}
}
```

Complexidades Temporais

quando o dicionário tem n entradas e

o vector tem capacidade *dim* (com $n < dim$)

	Melhor Caso	Pior Caso	Caso Esperado
Construção (vazio)	$O(dim)$	$O(dim)$	$O(dim)$
Pesquisa	$O(1)$	$O(n)$	$O(1)$
Inserção	$O(1)$	$O(n)$	$O(1)$
Remoção	$O(1)$	$O(n)$	$O(1)$
Obtenção do Iterador (1)	$O(1)$	$O(dim)$	$O(1)$
Percurso (1)	$O(dim)$	$O(dim)$	$O(dim)$

```
public void insertThVer(E element) throws RepeatedElementException {
    if(head != null) {
      SListNode<E> node = head;
      while(node.getNext() != null) {
        if ((node.getElement().compareTo(element) == 0) ||
            ((node.getNext() != null) &&
            ((node.getNext().getElement().compareTo(element)) == 0)))
          throw new RepeatedElementException();
        else if ((node.getElement().compareTo(element) < 0) &&
            ((node.getNext() != null) &&
            ((node.getNext().getElement().compareTo(element)) > 0))) {
          SListNode<E> newNode = new SListNode(element);
          newNode.setNext(node.getNext());
          node.setNext(newNode);
          break;
        }
        else if (node.getNext() == null) {
          SListNode<E> newNode = new SListNode(element);
          newNode.setNext(null);
          node.setNext(newNode);
          break;
        }
        else
          node = node.getNext();
      }
    }
```

DLL

```
protected void removeMiddleNode( DListNode<E> node ){
      DListNode<E> prevNode = node.getPrevious();
      DListNode<E> nextNode = node.getNext();
      prevNode.setNext(nextNode);
      nextNode.setPrevious(prevNode);
      currentSize--;
  }

  protected void addMiddle( int position, E element ){
      DListNode<E> prevNode = this.getNode(position - 1);
      DListNode<E> nextNode = prevNode.getNext();
      DListNode<E> newNode = new DListNode<E>(element,
prevNode, nextNode);
      prevNode.setNext(newNode);
      nextNode.setPrevious(newNode);
      currentSize++;
```