# Self Balancing Binary Search Tree

## Problem ID: selfbalancingbst

Write your own implementation of a self balancing binary search tree, specialized to store 32-bit signed integer key-value pairs.

Track a cursor (or current node) for each tree which we will use as a reference point for certain operations. The cursor should initially be unset, meaning it is a null value, sentinel value or something similar, depending on your implementation. The cursor node is the one provided to operations which take a node as input.

The data structure should support the following operations:

- Default construction - initializes an empty instance with a reasonable capacity. This is not tested explicitly here!
- Assignment and copy construction - must properly copy the contents of another instance of the data structure. Ensure the two instances do not share memory afterwards!
- Find - return the node with the given key. If such a node does not exist, returns a sentinel value.
- Lower Bound - return the node with the smallest key which is greater than or equal to the given key, If such a node does not exist, returns a sentinel value.
- Upper Bound - return the node with the smallest key which is strictly greater than the given key. If such a node does not exist, returns sentinel value.
- Insert - must insert a node with the given key-value pair, returning the newly inserted node. If the key exists already, nothing is inserted and the existing node is returned.
- Erase - must remove the given node. If given node is sentinel value, does nothing.
- Front - return the node with the minimum key in the tree. If no such node exists, returns a sentinel value.
- Back - return the node with the maximum key in the tree. If no such node exists, returns a sentinel value.
- Predecessor - return the node before the given node, that is, the node with the largest key which is lower than the given node's key. If such a node does not exist, or if given node is sentinel value, returns sentinel value.
- Successor - return the node after the given node, that is, the node with the smallest key which is higher than the given node's key. If such a node does not exist, or if given node is sentinel value, returns sentinel value.
- Rank - return the index of the node within the tree's left-to-right order, or in other words, how many nodes have smaller keys than the given node.
- $k$-th Element - return the node with the given index within the tree's left-to-right order.
- Element Access - provide access to reading and writing the value associated with a key, usually directly through the public interface of a node instead of the tree
- Size - return the size of the instance

The cursor should **NOT** be a member of the instance, as in real world applications you may have more than one per tree. After assignment and erase, the cursor should be reset to the special unset state. After insert, lower bound, upper bound, front, back, predecessor, successor, and $k$-th node operations, the cursor is set to the return value of the operation in question.

Note, in node based structures, some operations may be better suited as members of the node object, rather than the data structure object.

You must avoid any memory leaks or other memory errors in your implementation.

## Input

The input starts with a line containing an integer $q$, representing the number of lines that follow. Each line will represent an operation that is run. The lines have the following format: `<id> <operation> [arg]`

The instance ID will be an integer from 1 to $1,000$, representing an instance of the data structure. Each instance should be default initialized at the start as empty.

The operations are the following:

- `a` - construct a copy of the array, takes integer argument for the instance ID of the binary search tree to copy
- `?` - find, one integer argument, the key for which is searched
- `l` - lower bound, one integer argument, the key for which is searched
- `u` - upper bound, one integer argument, the key for which is searched
- `i` - insert, takes two integer arguments, the key to insert and the associated value
- `e` - erase, no additional arguments, uses cursor node
- `f` - front, no additional arguments, uses cursor node
- `b` - back, no additional arguments, uses cursor node
- `>` - successor, no additional arguments, uses cursor node
- `<` - predecessor, no additional arguments, uses cursor node
- `r` - rank, no additional arguments, uses cursor node, see output section
- `k` - $k$-th element, one integer argument, the value of $k$
- `g` - element access, read, see output section
- `s` - element access, write, takes integer argument, the element to write into the cursor node
- `z` - size, no additional arguments, see output section

You may assume requested operations will not cause an error in a correctly implemented data structure. For example, an erase operation on an unset/null/sentinel node will not occur in the input.

## Output

For each rank operation, output a line with the rank of the cursor node.

For each get operation, output a line with the value of the cursor node.

For each size operation, output a line with the size of the instance.

| Sample Input 1 | Sample Output 1 |
|---|---|
| 14<br>1 i -8 -2<br>1 i 4 5<br>1 ><br>1 i 2 3<br>1 e<br>1 f<br>1 i -7 0<br>1 ? 10<br>1 s 2<br>1 z<br>1 i 4 5<br>1 i -3 4<br>1 g<br>1 < | 3<br>4 |

**Sample Input 2**

```
22
1 g
1 i 96 83
2 ? 76
1 z
1 ? 81
2 z
1 s -35
1 g
1 k 0
2 a 2
1 >
2 >
1 k 0
2 f
1 ? -89
1 k 0
1 <
2 <
1 e
1 u -96
1 g
1 r
```

**Sample Output 2**

```
-
1
0
-
83
0
```

**Sample Input 3**

```
26
1 ? -47
2 l 0
1 a 2
1 u 78
2 e
1 >
1 r
1 g
1 ? -77
1 >
1 >
2 g
2 u 79
2 i 17 -29
2 r
2 a 2
1 l -127
2 f
2 ? -69
2 r
2 u 9
1 u -58
2 a 2
1 i 92 -46
2 b
2 u -110
```

**Sample Output 3**

```
-
-
-
0
-
```