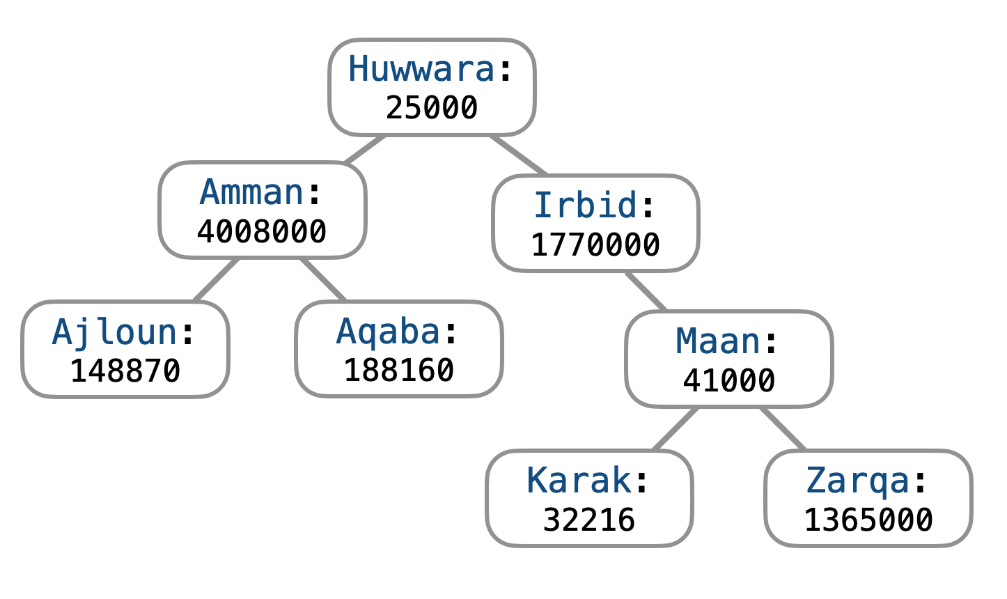
Ex1. Map it!

**Overview**

The Map class (often referred to as a "dictionary" or a "symbol table") is implemented as a binary search tree that stores in each node a **key** and a corresponding **value.** For example:

* A Map<string, string> object can map usernames (keys) to passwords (values), a word to its dictionary meaning, or a country to its capital city, etc.
* A Map<string, int> object can map student IDs (keys) to grades (values), words to the frequency of their occurrence in a book, cities to their population counts, etc.

The nodes in the BST are ordered by the key, not the value (keys can't repeat, but values can). For example, The following Map object maps cities to their population counts:



Note that Amman > Ajloun (because m > j) and Amman < Aqaba (because m < q).

You are given in this exercise a partial implementation of the class and are required to complete it.

**Things you are given**

* Class **Node**: Each node has a key, a value, and pointers to the left and right child nodes.
* Class **Map**: This is similar to the bst.h implementation with the following main differences:
  + There are two template parameters (K and V), not one.
  + Function void **insert**(const K& key, const V& val) inserts the given key-value pair. If the key is already in the tree, its value is updated to be the newly received value.
  + Function V& **value\_of**(const K& key) returns a reference to the value corresponding to the given key. If the key is not in the tree, the function throws a string exception.
  + The stream insertion operator (**<<**) is overloaded to allow for printing (in sorted order) the keys (and their values) .
  + Many functions were removed.

**Functions You Need To Implement**

Implement the following functions:

**(1)** Node<K, V>\* **search**(const K& key) (16 points)

This function returns a pointer to the node containing the given key. If the key is not in the tree, the function returns nullptr.

The function must run in *O*(height) if the previous call to the function was for a different key. If the previous call was for the same key, the function must run in*O*(1). For example:

search("A"); // O(height)

search("B"); // O(height)

search("B"); // O(1)

search("B"); // O(1)

search("A"); // O(height)

search("A"); // O(1)

search("B"); // O(height)

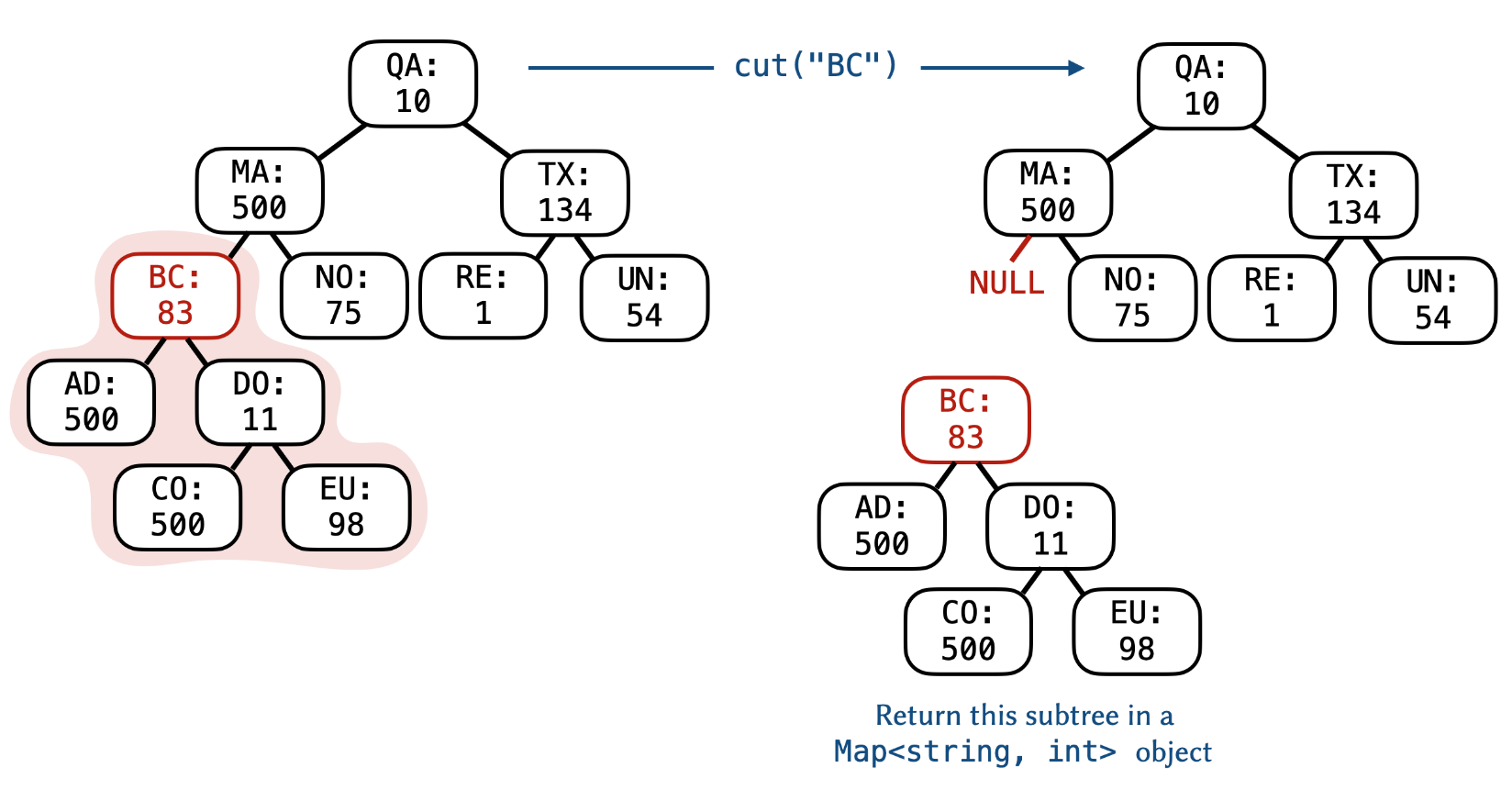
Note that if a key is searched for and then deleted from the map using one of the functions, then the next call to search should take *O*(height).

**(2)** Map **cut**(const K& key) (10 points)

This function removes the subtree rooted at key and returns a copy of it. It returns an empty Map object if key is not in the tree.

The function must run in*O*(height).

**Example.**

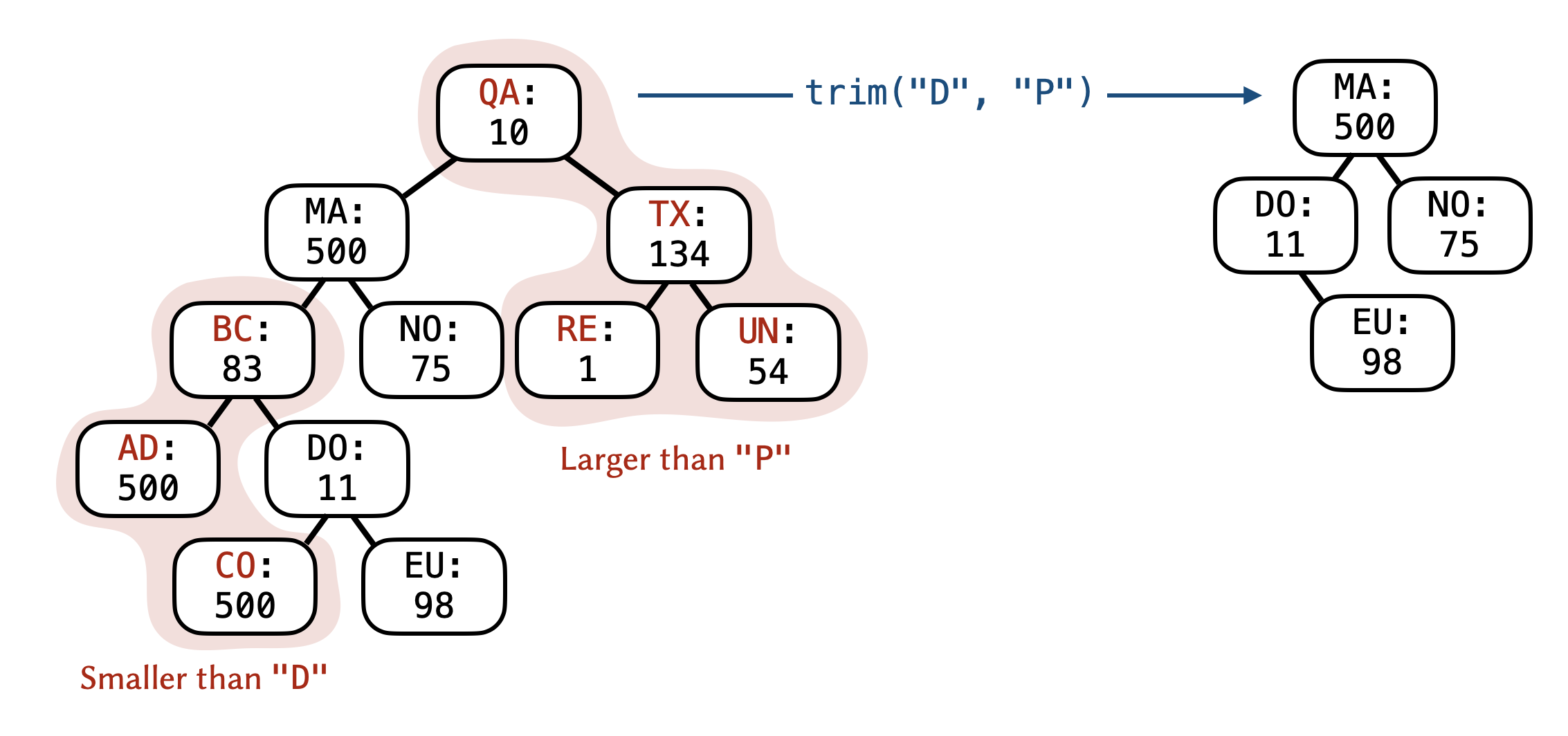


**(3)** void **trim**(const K& lo, const K& hi) (16 points)

This function removes from the tree all the keys larger than hi and all the keys less than lo.

The function must run in *O*(*n*).

**Example.**



**(4)** bool **has\_duplicate\_values**() const (12 points)

This function checks if there are any duplicate values (not keys) in the tree. The function must run in *O*(*n*×height).

**(5)** V **max\_value**() const (16 points)

This function returns the maximum value (not key) that is in the tree. It throws a string exception if the map is empty. The function must run in *O*(*n*).