**Exercises: Heaps and BST**

This document defines the lab for the ["Data Structures – Fundamentals (C#)" course @ Software University](https://softuni.bg/trainings/3672/data-structures-fundamentals-with-csharp-february-2022). Please submit your solutions (source code) of all below-described problems in [Judge](https://judge.softuni.org/Contests/2553/06-Heaps-and-Binary-Trees-Exercise).

Write C# code for solving the tasks on the following pages.

**Do not change the names of the provided projects, interfaces, classes, and methods. You are free to create new ones or modify the current ones if you follow the previously described rule.**

# BST Operations

You are given a skeleton, in which you have the following operations already implemented:

* void Insert(T) – Recursive implementation
* void EachInOrder(Action<T>) – In-Order traversal
* bool Contains(T) – Iterative implementation
* BST<T> Search(T) – Returns copy of the BST

You have to implement the rest of the operations:

* List<T> Range(T, T) – Returns collection with the elements found in the BST. Both borders are **inclusive**.
* DeleteMin() – Deletes the smallest element in the tree.
* DeleteMax() – Deletes the **max** **element**. The logic is like the DeleteMin() method, but you need to traverse the tree to the right.
* Delete(T) – Deletes a node with given value.
* Count() – Implement a **method** that returns the count of elements in the BST.
* Rank(T) – Implement a **method** that **returns** the **count** of elements **smaller** **than** a given **value**.
* Select(int) – Implement a **method** which accepts a number (**n**) and **returns** the first **element** which has exactly **n** elements **smaller** than it. Use the logic from Count() and Rank() to implement it.
* Ceiling(T) – Implement a **method** which **finds** (returns) the **nearest** **larger** **value** than given in the BST. This operation is similar to Floor() and DeleteMax().
* Floor(T) – Implement a **method** which **finds** (returns) the **nearest** **smaller** **value** than given in the BST. This operation is similar to DeleteMin().

The operation behaviour is defined in the following table, with more details below it:

|  |  |  |
| --- | --- | --- |
| C# Method | Return Type | Exception on empty tree |
| **DeleteMax()** | void | InvalidOperationException |
| **DeleteMin()** | void | InvalidOperationException |
| **Range(T, T)** | IEnumerable<T> | / |
| **Count()** | int | / |
| **Rank(T)** | int | / |
| **Select(int)** | T | InvalidOperationException |
| **Ceiling(T)** | T | InvalidOperationException |
| **Floor(T)** | T | InvalidOperationException |
| **Delete(T)** | void | InvalidOperationException |

You can reuse the solution from the Lab we had and only add implementations for the new methods. However it is indeed a good idea to try implementing those methods again, repetition will give you a better understanding and you can choose a different approach.

## Count Hints

If our current node is null, we will **return 0**. Otherwise, we will return the count of our **current node**:

We also have to modify our Insert() method. It will set the count of elements of our new node to the count of its children nodes plus itself:

Next, we need to find a way to update the recalculate the count for each node when DeleteMin() is invoked. One way would be to change the DeleteMin() implementation to be recursive:

What will happen if our tree is empty and we call DeleteMin()? **Fix** it. Our count is ready.

## Rank Hints

Create a new recursive method that will return 0 if the node is null. Then, we need to **compare the element** with the value of the node we are currently looking at. If the element is **smaller**, we can **go to the left**. If it is **larger**, we need to **get the count of the left** elements and **go to the right**. If we **find the element**, we will return the **count of elements**, **smaller** than it.

## Select Example

Graphical user interface, text, application, email

Description automatically generated

## Ceiling Example

Graphical user interface, text

Description automatically generated

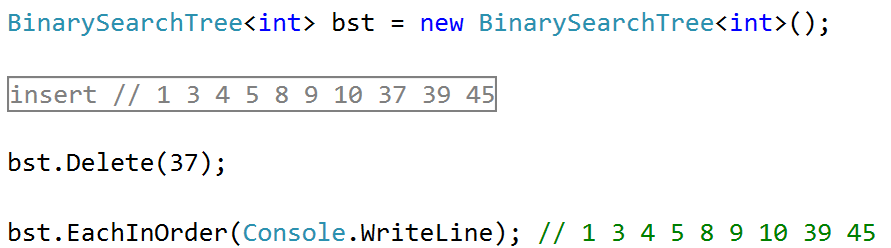
## Floor Example

Graphical user interface, text, application

Description automatically generated

## Delete Examples

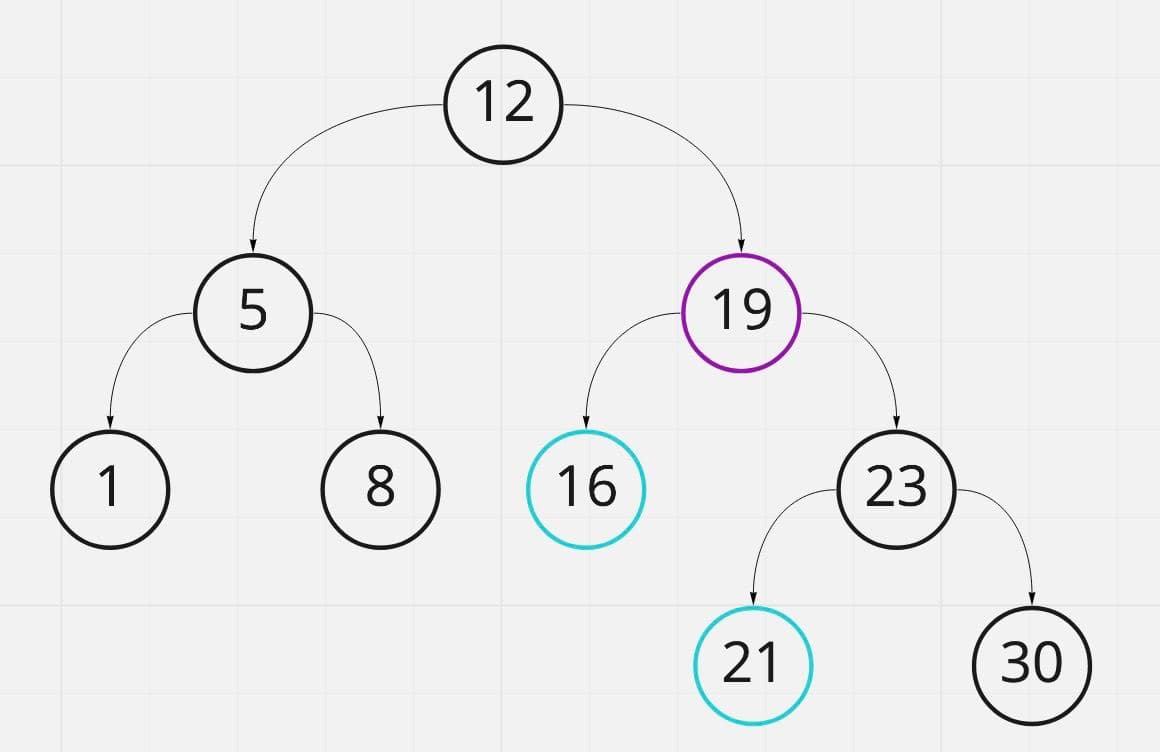
Delete is a more complex operation, which requires you to think what will happen if you delete an internal node. How are you going to preserve the BST property?



Graphical user interface, text, email

Description automatically generated

# Lowest Common Ancestor

You are given the binary tree and two values **V1** and **V2**. You need to return the lowest common ancestor (**LCA**) of **V1** and **V2** in the binary tree. Note that **lowest** here **means** in terms of **level** **distance**. The closer the node is to both values the **lower** we say it is. In other words, you can **ignore** the **value** you **should only care for the distance**.

In the given tree above the lowest common ancestor of nodes 16 and 21 is **node 19**. Node 19 is the **lowest** node which has nodes 16 and 21 **as descendants**.

# Min Heap

Based on the implementation of the **MaxHeap<T>** and ADS **Heap<T>** from the lab implement a data structure that acts and operates the same way, however this time the **heap property** you want to **preserve** is the **Min heap property**.

Remember you have to throw **InvalidOperationException** if you attempt to **Peek() or ExtractMin() on an empty heap**.

## Implement Decrease Key

In the skeleton there is a **PriorityQueue** class that extends the MinHeap to support the DecreaseKey(T element) operation, that changes the priority of a given key. In a Min Binary Heap this should increase the priority of a given key, moving it higher in the tree structure, e.g. decreasing the price of a given product, increases its priority for the customers.

# Cookies

We want the sweetness of all his cookies to be greater than value **K**. To do this, we need to repeatedly mix two cookies with the least sweetness. We create a special combined cookie with sweetness calculated by:

* (l**east sweet** cookie) + (**2** \* **2nd least sweet cookie)**.

We repeat this procedure until all the cookies in the collection have a sweetness **not less than K**.

You are given the cookies. Return the number of operations required to give the cookies a sweetness **not less than K**. Return **-1** if this isn't possible.

Implement the **Integer** solve (int **k**, int[] **cookies**) **method** inside the provided **CookiesProblem** class.

### Input

Study the tests provided. The first parameter is the **K** value and the second one is the **array** representing the **cookies**.

The first line consists of integers, the number of cookies, and the minimum required sweetness, separated by a space.

### Output

The method should **return a single integer** **-1** if there is **no solution**. **Otherwise,** return the **number of operations** required to complete the task.

|  |  |
| --- | --- |
| **Input** | **Output** |
| 7  2 3 9 10 12 | 2 |

### Hint

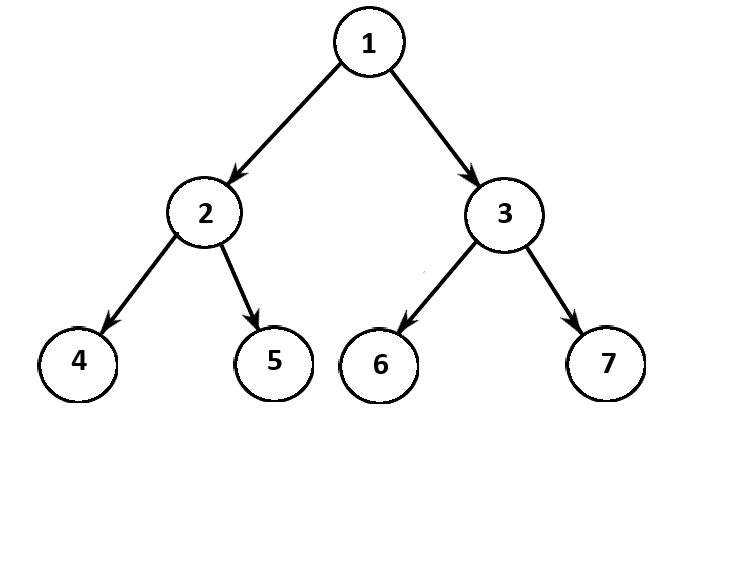
Use the **MinHeap<T>** from the previous problem or try out the **OrderedBag<T>** from **Wintellect Power Collections**. You can read more about this **nugget package** from [here](https://archive.codeplex.com/?p=PowerCollections).

# Top View

You are given the binary tree. Print the **top view** of the binary tree. More info can be found [here](https://www.techiedelight.com/print-top-view-binary-tree/).

The top view means when you look at the tree from the top of the nodes, what you will see will be called the top view of the tree. See the example below.

### Examples

****

Given the above tree, the result should be **1, 2, 4, 3, 7,** where the order of **output does not matter.**