CS 284: Homework Assignment 4

1 Assignment Policies

Collaboration Policy. Homework will be done individually: each student must hand in their own answers. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

Your code must include a comment with your name and section .

[Problem 1- Treap] A *treap* is a binary search tree (BST) which additionally maintains heap priorities. An example is given in Figure 1. A node consists of

- A key k (given by the letter in the example),
- A random heap priority p (given by the number in the example). The heap priority p is assigned at random upon insertion of a node. It should be unique in the treap.
- A pointer to the left child and to the right child node,

For example, the root node in the treap in Figure 1 has "h" as key and 9 as heap priority.

Note that if you insert just the keys of the treap in Figure 1 into an empty BST, selecting each node based on the priority (from max to min), you get back a BST of the exact same form as the treap. For example, if you insert the keys [h;j;c;a;e] (in that order) into an empty BST, then you get back a tree just like that in Figure 1. Since the priorities are chosen at random, treaps may be understood as *random BSTs*.

The good thing about random BSTs is that they are known to have logarithmic height with high probability. Thus the same is true for treaps. Indeed, on average (the run time depends on the randomly chosen heap priorities), add, delete, and find operations take $\mathcal{O}(\log(n))$ time where n is the number of nodes in the treap.

In this homework, you will implement a treap. In the following, we discuss the components of the data structure and its operations in detail.

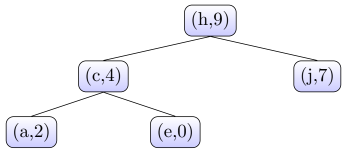


Figure 1: Example of a Treap

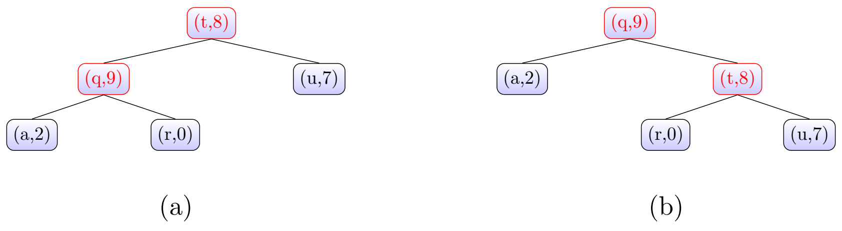


Figure 2: Right rotation (a→b) and left rotation (b→a)

1.1 The Node Class

Create a private static inner class `Node<E>` of the Treap class (described in the next subsection) with the following attributes and constructors:

- Data fields:

```java
E data; // key for the search
int priority; // random heap priority
Node<E> left;
Node<E> right;
```

- Constructors:

```java
public Node(E data, int priority)
```

Creates a new node with the given data and priority. The pointers to child nodes are null. Throw exceptions if data is null.

- Methods:

```java
Node<E> rotateRight();
Node<E> rotateLeft();
```

`rotateRight()` performs a right rotation according to Figure 2, returning a reference to the root of the result. The root node in the figure corresponds to this node. Update the data and priority attributes as well as the left and right pointers of the involved nodes accordingly.

`rotateLeft()` performs a left rotation according to Figure 2 returning a reference to the root of the result. The root node in the figure corresponds to this node. Update the attributes of the nodes accordingly.

Why rotation? Rotations preserve the ordering of the BST, but allows one to restore the heap invariant. Indeed, after adding a node to a treap or removing a node from a treap, the node may violate the heap property considering the priorities. In this case it is necessary to perform one or more of these rotations to restore the heap property. Further details shall be supplied below.

1.2 The Treap Class

- Data fields:

```java
private Random priorityGenerator;
private Node<E> root;
```

`E` must be `Comparable`.

- Constructors:

```java
public Treap();
public Treap(long seed);
```

`Treap()` creates an empty treap. Initialize `priorityGenerator` using `new Random()`. See http://docs.oracle.com/javase/8/docs/
for more information regarding Java's pseudo-random number generator.

`Treap(long seed)` creates an empty treap and initializes `priorityGenerator` using `new Random(seed)`.

- Methods:

```java
boolean add(E key);
boolean add(E key, int priority);
boolean delete(E key);
private boolean find(Node<E> root, E key);
public boolean find(E key);
public String toString();
```

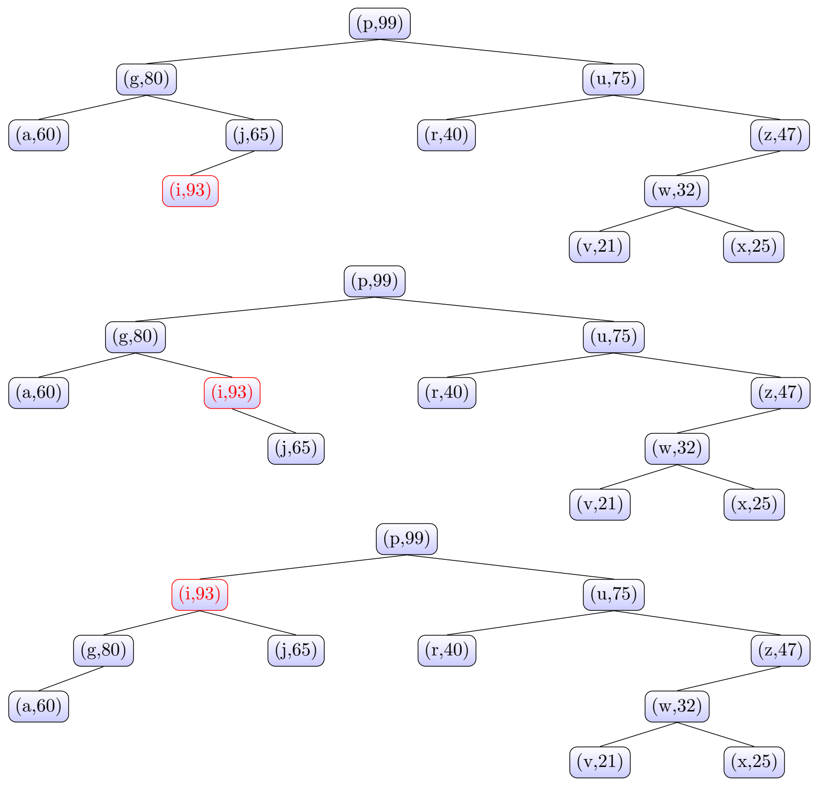
We next describe each of these methods.

1.2.1 Add operation

To insert the given element into the tree, create a new node containing `key` as its data and a random priority generated by `priorityGenerator`. The method returns **`true`**, if a node with the key was successfully added to the treap. If there is already a node containing the given key, the method returns **`false`** and does not modify the treap.

- Insert the new node as a leaf of the tree at the appropriate position according to the ordering on `E`, just like in any BST.
- If the priority of the parent node is less than the priority of the new node, bubble up the new node in the tree towards the root such that the treap is a heap according to the priorities of each node (the heap is a max-heap, i.e., the root contains the highest priority). To bubble up the node, you must implement the rotation operations mentioned above.

Here is an example in which the node (i,93) is added to the treap.



Hint: For the `add` method you should proceed as in the addition for BSTs. Except that I would suggest

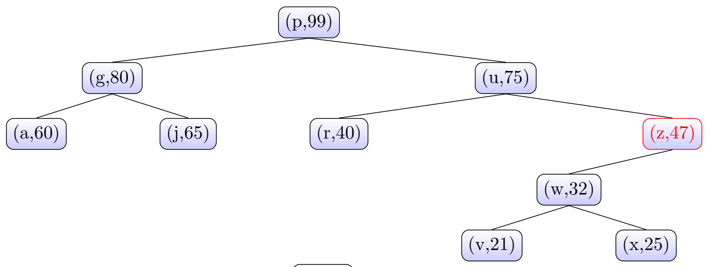
- adapting it to an iterative version (rather than using recursion).
- storing each node in the path from the root until the spot where the new node will be inserted, in a stack
- after performing the insertion, use a helper function `reheap` (with appropriate parameters that should include the stack) to restore the heap invariant. Note that if our nodes had pointer to their parents, then we would not need a stack.
- have `add(E key)` call the `add(E key, int priority)` method once it has generated the random priority. Thus all the "work" is performed by the latter method.

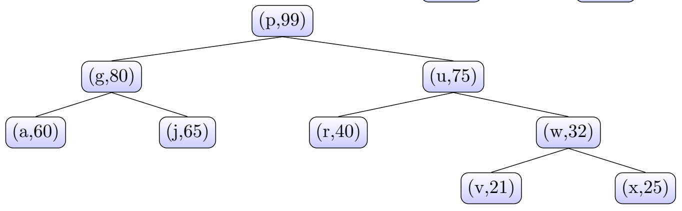
1.2.2 Delete operation

`boolean delete(E key)` deletes the node with the given key from the treap and returns `true`. If the key was not found, the method does not modify the treap and returns `false`. In order to remove a node trickle it down using rotation until it becomes a leaf and then remove it. When trickling down sometimes you will have to rotate left and sometimes right. To decide which direction to rotate: (1) if ~~there is no right subtree, rotate right (an example of this case is shown below);~~ if either the left or right child does not exist, replace the target node with the non-null child; (2) if both left and right subtree exists, rotate in the direction of the smaller child.

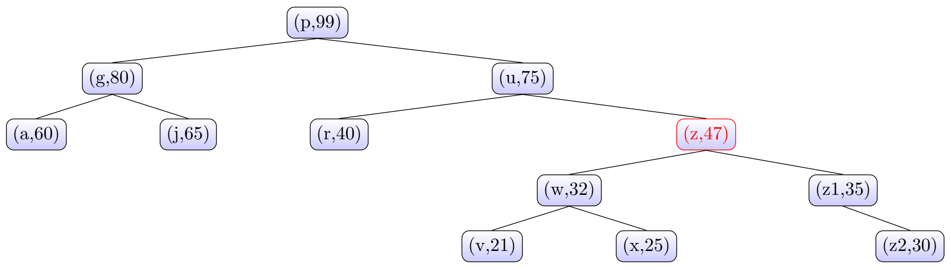
Here are two examples of deletion of the node (z,47):

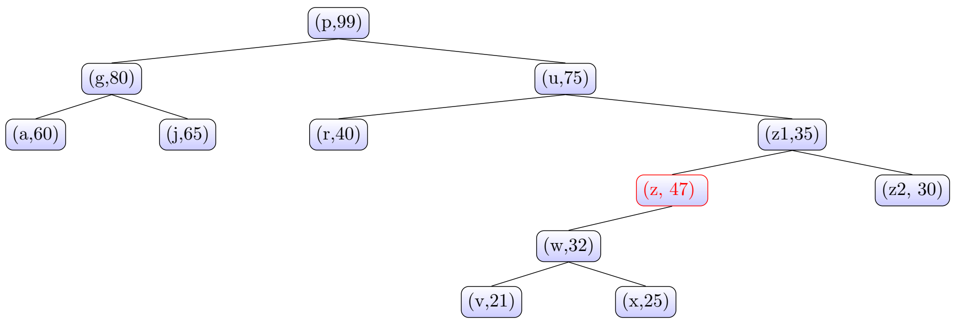
(1) In the first example, first, observe that (z, 47) has no right subtree, so replace it with the non-null child (w, 32):



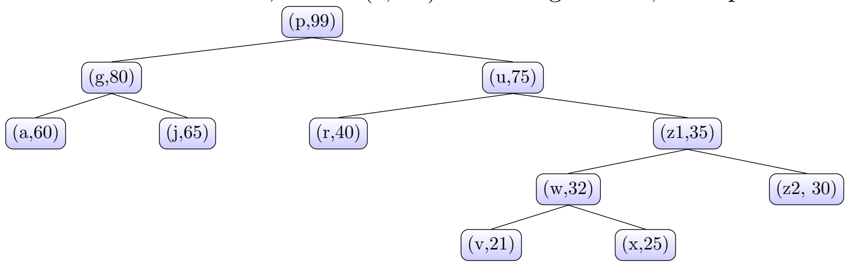


(2) In the second example below, both the left and right child of (z, 47) are not null. When trickling down, we rotate to the direction of the smaller child. Notice that the smaller child of (z, 47) is the left child, so rotate left:





After the first rotation, notice (z, 47) has no right child, we replace it with the non-null child:



1.2.3 Find operation

`private boolean find(Node<E> root, E key)`: Finds a node with the given key in the treap rooted at `root` and returns true if it finds it and false otherwise.

`boolean find(E key)`: Finds a node with the given key in the treap and returns true if it finds it and false otherwise.

1.2.4 `toString` operation

`public String toString()`: Carries out a preorder traversal of the tree and returns a representation of the nodes as a string. Each node with key k and priority p, left child l, and right child r is represented as the string (l) $[k,p](r)$. If the left child does not exist, the string representation is $[k,p]$ (r). Analogously, if there is no right child, the string representation of the tree is (l) $[k,p]$. Variables l, k, and p must be replaced by its corresponding string representation, as defined by the `toString()` method of the corresponding object.

Hint: You can reuse the exact same method in the binary tree class we saw in class. You will have to add a `toString` method to the `Node` class so that it prints a pair consisting of the key and its priority.

1.3 An Example Test

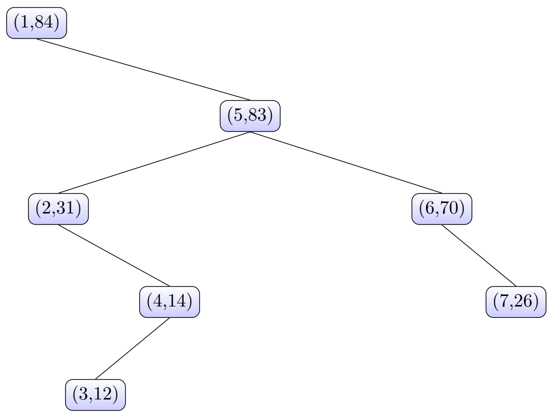
For testing purposes you might consider creating a Treap by inserting this list of pairs (key,priority) using the method `boolean add(E key, int priority)`:

(4,19);(2,31);(6,70);(1,84);(3,12);(5,83);(7,26)

The code for building this Treap is:

```java
testTree = new Treap<Integer>();
testTree.add(4,19);
testTree.add(2,31);
testTree.add(6,70);
testTree.add(1,84);
testTree.add(3,12);
testTree.add(5,83);
testTree.add(7,26);
```

The resulting Treap should look like this:



The output using `toString()` of the above would be

```
(key=1, priority=84)
  null
  (key=5, priority=83)
    (key=2, priority=31)
      null
      (key=4, priority=19)
        (key=3, priority=12)
          null
          null
        null
    (key=6, priority=70)
      null
      (key=7, priority=26)
        null
        null
```