DSC 395T Algorithms and Data Structures	Individual Project
Treaps	Programming Assignment #4

In this assignment you will implement a map (associative lookup) using a data structure called a treap, which is a combination of a tree and a heap. Your key challenge in this assignment will be to carefully and thoroughly test your data structure, so you should also design a testing program for your code. For this assignment, you should not discuss testing strategies with other teams.

1 Treaps

A *treap* is a binary search tree that uses randomization to produce balanced trees. In addition to holding a key-value pair (a map entry), each node of a treap holds a randomly chosen priority value, such that the priority values satisfy the *heap property*: Each node has a priority that is at least as large as the priorities of its two children. An example treap is shown in Figure **??**, where the keys are shown at the top of each node and the priorities are shown at the bottom of each node. Notice that the keys obey the binary search tree (BST) property and the priorities obey the heap property. Because the keys obey the BST property, a lookup operation can be performed just as with any BST. However, the insert and remove operations are slightly more complex.

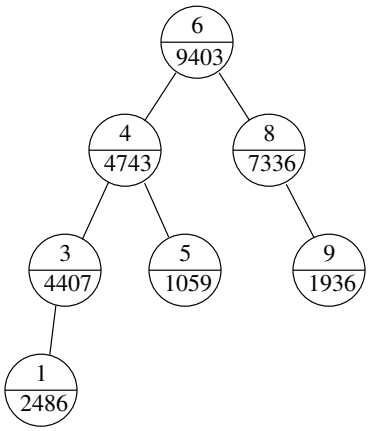

Figure 1: A treap for a map with key set $\{1, 3, 4, 5, 6, 8, 9\}$. For each node, the key is shown in the top half, while the priority is shown in the bottom half. The priority values are chosen at random, with larger numbers representing higher priorities. The values are not shown.

To insert a new node x with key k, we first perform the insertion at the appropriate leaf position according to the BST property, exactly as in a binary search tree (See Figure **??**). The node is assigned a randomly chosen priority p, and because x's parent y may have priority less than p, the heap property may be violated. To restore the heap property, we perform a rotation, making x the parent of y, as shown in Figure **??**(b). Specifically, if x is the left child of y, then we rotate right around y, and if x is the right child of y, then we rotate left around y. Node x now has a new parent, and the heap property may still be violated, requiring another rotation. In general, the heap property is restored by rotating the new node x up the treap as long as it has a parent with a lower priority. Figure **??** shows an insertion requiring 2 rotations.

To remove a node x, we "reverse" the insertion. We rotate x down the treap until it becomes a leaf, and then we simply clip it off. At each step, the decision to rotate left or right is governed by the relative priorities of the children. The child with the higher priority should become the new parent. Thus, if x's left child has higher priority than x's right child, then we rotate right around x. Conversely, if x's right child has higher priority than x's left child, then we rotate left around x. Figure **??** illustrates a removal requiring 2 rotations. This removal reverses the insertion of Figure **??**.

All three map operations—lookup, insert, and remove—run in time $O(h)$, where h is the height of the treap. It is not hard to show that a treap with n nodes has expected height $\Theta(\log n)$. Note that the root of a treap is determined by the randomly chosen priorities. The node with the highest priority is the root. Thus, the root node is equally likely to contain any of the map entries, regardless of the order in which the entries are inserted or removed. Consequently, we

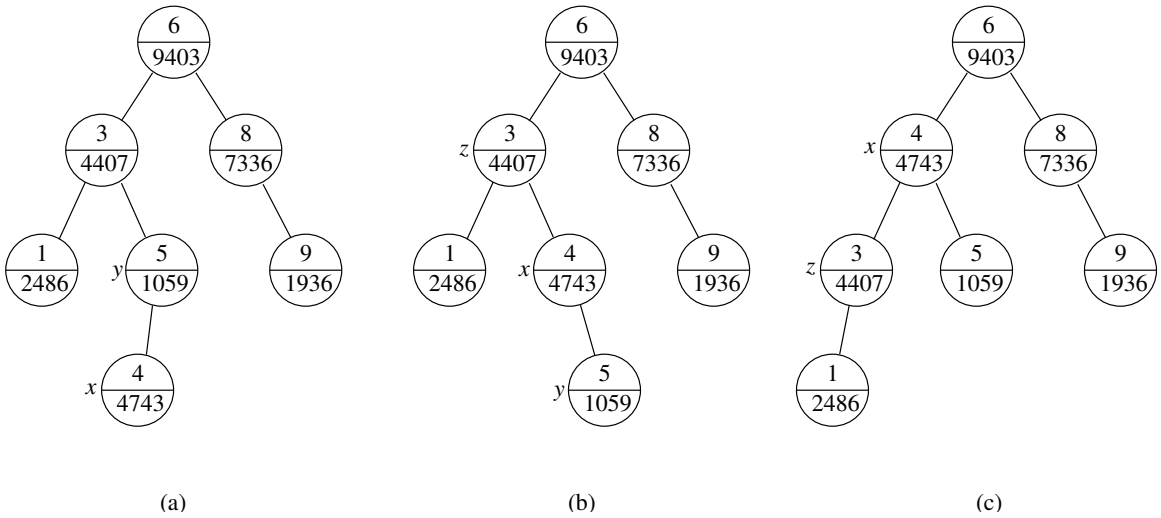

Figure 2: Inserting new node x into a treap. (a) The new node x, with key *k=4* and priority *p=4743*, is added as a leaf according to the BST property. The heap property with respect to x's parent y is violated. (b) The situation after a right rotation around y; the heap property with respect to x's new parent z is violated. (c) After a left rotation around z, the heap property is restored.

expect that half of the entries will be in the left subtreap and the other half in the right subtreap. The analysis of treap height is therefore similar to the analysis of recursion depth in quicksort.

2 Your Assignment

Implement a map using a treap. In particular, you should implement the following abstract base class. Your treap should store entries with keys that are `Comparable` objects and values that are any object. In both cases the Treap is generic on the exact type; keys have type `KT` and values have type `VT`. The `lookup(k)` method should return None if no entry with key *k* is in the map.

```
class Treap(ABC, Generic[KT, VT], Iterable):
  def get_root_node(self) -> Optional[TreapNode]: ...
  def lookup(self, key: KT) -> Optional[VT]: ...
  def insert(self, key: KT, value: VT) -> None: ...
  def remove(self, key: KT) -> Optional[VT]: ...
  def split(self, threshold: KT) -> List[Treap[KT, VT]]: ...
  def join(self, other: Treap[KT, VT]) -> None: ...
  def meld(self, other: Treap[KT, VT]) -> None: ...
  def difference(self, other: Treap[KT, VT]) -> None: ...
  def balance_factor(self) -> float: ...
  def __str__(self) -> str: ...
  def __iter__(self) -> Iterator: ...
```

A more detailed description of the interface is in `treap.py`. Implement your treap-based map as `TreapMap` in `treap_map.py` and make sure your implementation inherits from `Treap[KT, VT]`. Please use the TreapNode defined in `treap_node.py` and do not modify or rename the six existing fields in TreapNode. You only need to modify `treap_map.py` for this assignment, but you are welcome to modify `treap_node.py` as well.

The insert method. Insertion into the treap should be implemented as outlined in Figure **??**. You can assume that two nodes will never have an equal priority, which is enforced by the starter code.

The remove method. Removal from the treap should be implemented as outlined in Figure **??**.

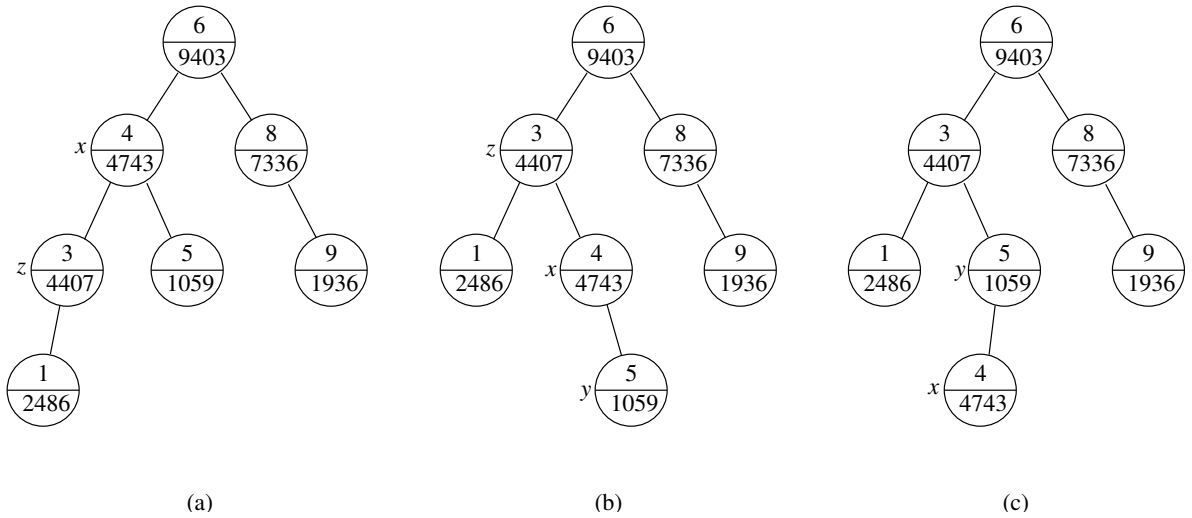

Figure 3: Removing a node *x* from a treap. (a) Node *x* has two children, of which the left child *z* has higher priority. (b) After a right rotation around *x*, node *x* now has only one child, *y*. (c) After a left rotation around *x*, node *x* is now a leaf and can be clipped off like an excessively long toenail.

The split method. A treap *T* can be *split*, using a key *k*, to produce two treaps, T_1 and T_2, such that T_1 contains all of the entries in T with key less than k, and T_2 contains all of the entries in T with key greater than or equal to k. To perform the split, we insert into T a new entry x with key k and priority $p =$`MAX_PRIORITY`, forming a new treap T'. (`MAX_PRIORITY` is defined by the `Treap` abstract base class.) Because x has the highest possible priority, x is the root of T', so the split has been accomplished with T_1 being the left subtreap and T_2 being the right subtreap. You should not "lose" any value associated with k if k is already in the treap, although it is ok if you destroy the old treap. Existing TreapNode priorities should remain unchanged.

The join method. The inverse of a split is *join*, in which two treaps, T_1 and T_2, with all keys in T_1 being smaller than all keys in T_2, are merged to form a new treap T. To perform the join, we create a new treap T' with an arbitrary new root node x and with T_1 and T_2 as the left and right subtreaps. We then remove x from T' to form the joined result T. You can assume that two nodes will never have the same priority.

Split and join both take time $O(h)$, where h is the height of the T (the input to split or the result of join). The expected height is $\Theta(\log n)$, where n is the size of T, so split and join both run in $O(\log n)$ expected time. More interestingly, split and join can be used as subroutines to $meld$ two treaps or take the *difference* between two treaps. Those functions are described in the Karma section.

3 Testing

Since the treap in this assignment is not part of a larger application, you will not be able to use or test your treap without writing your own test program. Write a test suite to test your treap for correctness. You may use pytest or just write a program that manually runs the appropriate tests. We have provided a few starter test cases inside of the `test/test_treaps.py` file, which you can run with the Pytest run configuration.

4 Karma

For those of you looking for an additional challenge, implement any of the three optional operations defined in the interface, namely, `balance_factor()`, `meld()` and `difference()`. If you do not implement them, raise an `AttributeError`.

4.1 Meld

A *meld* takes two treaps, T_1 and T_2 and merges them into a new treap T, much like the Vulcan mind meld for which it is named[1]. Unlike a join, a meld does not require any relationship between the keys in T_1 and T_2. Meld is a naturally recursive procedure and should be able to meld two treaps of size n and m ($m \leq n$) in $O(m \log(n/m))$ time. Describe how you meld treaps and how your algorithm meets the specified asymptotic time bound.

4.2 Difference

The *difference* between two treaps, T_1 and T_2, is a treap T containing the keys of T_1 with any keys in T_2 removed. The difference can also be computed recursively and also runs in $O(m \log(n/m))$ time. Describe how you take a difference and how your algorithm satisfies this time bound.

4.3 Diagnosing Problems Through Testing

Typically, the goal of a test program is to identify bugs. With some additional work, you can attempt to diagnose common problems by observing the behavior of the program. For example, if the iterator misses one key, it is likely that the missing key is the first or last key added. A test program can attempt to verify this hypothesis and provide a suggestion to the user. Can you use your test program to assist in finding common mistakes?

4.4 Balance Statistics

It would be useful to know how balanced or imbalanced your treap is. The balance factor is the ratio between the height of the treap and the minimum possible height. A perfectly balanced treap will have a balance factor of 1.0. Include observations on how well the treap seems to keep itself balanced in your report.

5 What to Turn in

Submit your code in the same manner as the previous projects, on Gradescope. Be sure to commit and push your code before submitting.

Note: This is an individual project. You are not allowed to work in a group. We will check your code for collaboration. Please check the syllabus for more clarification on what is and isn't allowed.

Acknowledgments. We thank Bobby Blumofe for the original version of this assignment, and we thank Walter Chang, Arthur Peters, Ashlie Martinez, Elvin Yang, and Matthew Giordano for their subsequent modifications.

[1] Not really.