CSC11F: Advanced Data Structures and Algorithms

Balanced Tree

Yutaka Watanobe (142-A, yutaka@u-aizu.ac.jp)

1 Introduction

A binary search tree can be unbalanced depending on features of data. For example, if we insert n elements in ascending order, the tree become a list, leading to long search times. One of strategies is to randomly shuffle the elements to be inserted. However, we should consider to maintain the balanced binary tree where different operations can be performed one by one depending on requirement.

2 Treap

There are number of approaches to construct balanced trees. One of simple strategies is Treap which is rather easy to implement.

We can maintain the balanced binary search tree by assigning a priority randomly selected to each node and by ordering nodes based on the following properties. Here, we assume that all priorities are distinct and also that all keys are distinct.

- **binary-search-tree property.** If v is a **left child** of u, then $v.key < u.key$ and if v is a **right child** of u, then $u.key < v.key$
- **heap property.** If v is a **child** of u, then $v.pri < u.pri$

This combination of properties is why the tree is called Treap (tree + heap).

An example of Treap is shown in Figure 1.

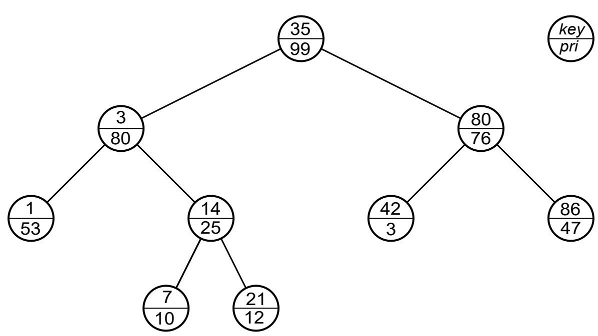

Figure 1: An example of Treap

Insert

To insert a new element into a Treap, first of all, insert a node which a randomly selected priority value is assigned, in the same way for ordinal binary search tree. For example, Figure 2 shows the Treap after a node with key = 6 and pri = 90 is inserted.

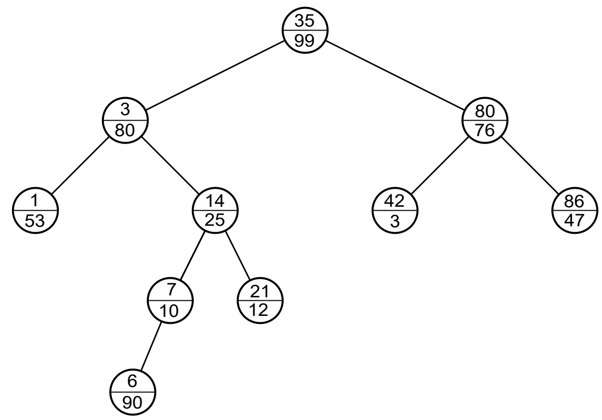

Figure 2: Insert operation to a Treap

It is clear that this Treap violates the heap property, so we need to modify the structure of the tree by **rotate** operations. The rotate operation is to change parent-child relation while maintaing the binary-search-tree property (Figure 3).

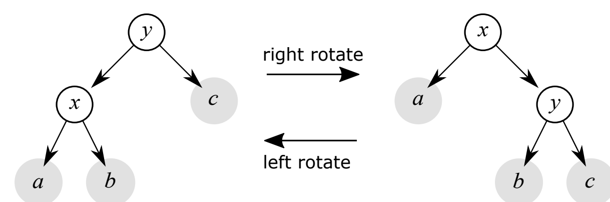

Figure 3: Rotate operations

The rotate operations can be implemented as follows.

```
rightRotate(Node t)
  Node s = t.left
  t.left = s.right
  s.right = t
  return s // the new root of subtree

leftRotate(Node t)
  Node s = t.right
  t.right = s.left
  s.left = t
  return s // the new root of subtree
```

Figure 4 shows processes of the rotate operations after the insert operation to maintain the properties.

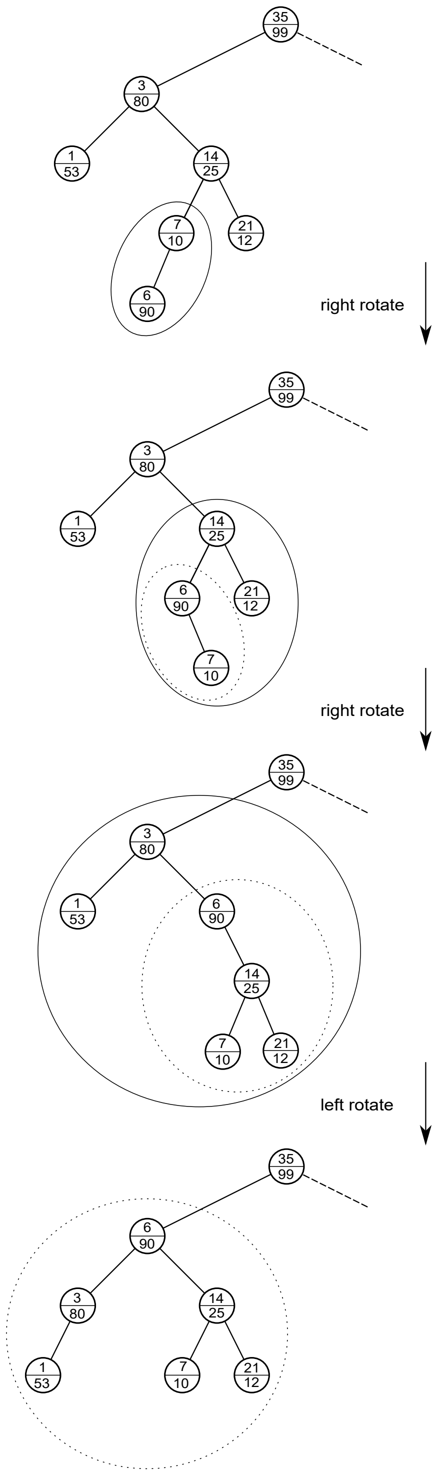

Figure 4: Processes of rotate operations after an insertion

The insert operation with rotate operations can be implemented as follows.

```
// search the corresponding place recursively
insert(Node t, int key, int pri)
  if t == NULL             // when you reach a leaf
    return Node(key, pri)  // create a new node
  if key == t.key // ignore duplicated keys
    return t

  if key < t.key  // move to the left child
    // update the pointer to the left child
    t.left = insert(t.left, key, pri)
    // if the left child has higher priority
    if t.pri < t.left.pri
      t = rrotate(t)
  else  // move to the right child
    // update the pointer to the right child
    t.right = insert(t.right, key, pri)
    // if the right child has higher priority
    if t.pri < t.right.pri
      t = lrotate(t)

  return t
```

Delete

To delete a node from the Treap, first of all, the target node should be moved until it become a leaf by rotate operations. Then, you can remove the node (the leaf). These processes can be implemented as follows.

```
// seach the target recursively
erase(Node t, int key)
  if t == NULL
    return NULL

  if key == t.key  // if t is the target node
    // if t is a leaf
    if t.left == NULL && t.right == NULL
      return NULL
    // if t has only the right child
    else if t.left == NULL
      t = lrotate(t)
    // if t has only the left child
    else if t.right == NULL
      t = rrotate(t)
    // if t has both the left and right child
    else
      // pull up the child with higher priority
      if t.left.pri > t.right.pri
        t = rrotate(t)
      else
        t = lrotate(t)
    return erase(t, key)

  // search the target recursively
  if ( key < t.key)
    t.left = erase(t.left, key)
  else
    t.right = erase(t.right, key)

  return t
```

3 References

1. Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press.

4 Assignments

Place

https://onlinejudge.u-aizu.ac.jp/beta/room.html#CSC11F_2023_Week_04

Duration

2 weeks

Problem A: Treap

Write a program which performs the following operations to a Treap T based on the above described algorithm.

- insert(k, p): Insert a node containing k as key and p as priority to T.
- find(k): Report whether T has a node containing k.
- delete(k): Delete a node containing k.
- print(): Print the keys of the binary search tree by inorder tree walk and preorder tree walk respectively.

Input

In the first line, the number of operations m is given. In the following m lines, operations represented by insert(k, p), find(k), delete(k) or print are given.

Output

For each find(k) operation, print "`yes`" if T has a node containing k, "`no`" if not.

In addition, for each print operation, print a list of keys obtained by inorder tree walk and preorder tree walk in a line respectively. Put a space character **before each key**.

Constraints

- The number of operations $\leq 500,001$
- The number of print operations ≤ 10
- $0 \leq k, p \leq 2,000,000,000$
- The height of the binary tree does not exceed 50 if you employ the above algorithm
- The keys in the binary search tree are all different
- The priorities in the binary search tree are all different

Sample Input 1

```
15
insert 35 99
insert 3 80
insert 1 53
insert 14 25
insert 80 76
insert 42 3
insert 86 47
insert 21 12
insert 7 10
insert 6 90
print
find 21
find 22
delete 35
print
```

Sample Output 1

```
 1 3 6 7 14 21 35 42 80 86
 35 6 3 1 14 7 21 80 42 86
yes
no
 1 3 6 7 14 21 42 80 86
 6 3 1 80 14 7 21 42 86
```