Homework Assignment #2

Due: January 30, 2020, by 5:30 pm

Jiahong Zhai 1005877561

Question 1

We can use Max Binary Heap to solve this problem.

We need four classes to implement this.

Class Node, Class Tree, Class Forest, Class Operation

Class Node:

\\ represent an order

Instance:

1. Incentive (integer) \\ how much cent customer put in this order
2. Order information \\ doesn’t affect our implement
3. Parent (node) \\ the parent of this node, maybe null
4. LeftChild (node) \\ left child of this node, maybe null
5. RightSibling (node) \\ right child of this node, maybe null

Class Tree:

\\ a binomial tree, in the which each node’s incentive is bigger or equal to its children’s

Instance:

1. Root (node) \\ the root of the tree.
2. Height (integer) \\ the height of the tree

Class Forest:

\\ the forest of some binomial trees

Instance:

1. Trees (List) \\ a list contain all binomial trees in the Forest, in increasing order of height
2. Max (integer) \\ the max incentive in this forest.

Class Operation:

\\ all methods to manipulate the data structure

Max_Incentive (f: Forest):

\\ return the max incentive of the forest

Return f.Max

```
Union_Two_forests (f1: Forest, f2: Forest):

    \\ put two forest together, when 1 warehouse is closed

    f_new = new Forest

    f.Max = Max(f1.Max, f2.Max)

    list temp = new list

    Tree carry = new Tree

    int i = 0

    int j = 0

    while(i < f1.Trees.lenght && j < f2.Trees.lenght)

        if f1.Trees[i].height < f2.Trees[j].height

            if carry = null

                f_new.Trees.add(f1.Trees[i])

                i ++

                continue

            else

                Tree new_carry = new Tree

                if carry.root.incentive>= f1.Trees[i].root.incentive:

                    f1.Trees[i].root.parent = carry.root

                    f1.Trees[i].root.sibling = carry.root.leftchild

                    new_carray.height = carry.height +1

                    carry = new_carry

                    i++
```

```
            continue
        else
            carry.root.parent = f1.Trees[i].root
            carry.root.sibling = f1.Trees[i].root.leftchild
            new_carray.height = carry.height +1
            carry = new_carry
            i++
            continue
if f1.Trees[i].height > f2.Trees[j].height
    if carry = null
        f_new.Trees.add(f2.Trees[j])
        j ++
        continue
    else
        Tree new_carry = new Tree
        if carry.root.incentive>= f2.Trees[j].root.incentive:
            f2.Trees[j].root.parent = carry.root
            f2.Trees[j].root.sibling = carry.root.leftchild
            new_carray.height = carry.height +1
            carry = new_carry
            j++
            continue
```

```
        else
            carry.root.parent = f2.Trees[j].root
            carry.root.sibling = f2.Trees[j].root.leftchild
            new_carray.height = carry.height +1
            carry = new_carry
            j++
            continue
if f1.Trees[i].height = f2.Trees[j].height
    if carry = null
        Tree new_carry = new Tree
        iff1.Trees[i].root.incentive>=f2.Trees[j].root.incentive:
            f2.Trees[j].root.parent = f1.Trees[i].root
            f2.Trees[j].root.sibling = f1.Trees[i].leftchild
            new_carray.height = f1.Trees[i].height +1
            new_carray.root = f1.Trees[i].root
            carry = new_carry
            i++
            j++
            continue
        else
            f1.Trees[i].root.parent = f2.Trees[j].root
            f1.Trees[i].root.sibling = f2.Trees[j].leftchild
```

```
                new_carray.height = f1.Trees[i].height +1
                new_carray.root = f2.Trees[j].root
                carry = new_carry
                i++
                j++
                continue
            else
                f_new.Trees.add(carry)
                carry = merged of f1.Trees[i] and f2.Trees[j]. as we
        did before
                i ++
                j ++
        return f_new


Union_Insertion (f1: Forest, n: Node):
        new_f = new Forest
        new_t = new Tree
        new_t.root = n
        new_t.height = 0
        new_f.Max = n.incentive
        new_f.Trees.add(new_t)
        return Union_Two_forests (f1, new_f)
```

```
ExtractMax (f: Forest):
    T = new Tree
    for(Tree t in f.Trees):
        if t.root.incentive = f.Max
            T = f.Trees.pop(t)
    List new_Trees = []
    new_Trees .add(T. root.left child)
    curr = T. root.left child
    while (curr.sibling exist)
        new_Trees .add(curr.sibling)
        curr = curr.sibling
    new_f = new Forest
    new_f.Trees = new_Trees
    new_f.Max = Max(t.incentive in new_Trees)
    f = Union_Two_forests (f, new_f)
```

WC MaxIncentive is O(1) because we only need to get the instance Max of the Forest

WC Union is O(logn) because each of forest have O(log n) Bk trees. We only do comparison among these trees.

WC insert is O(logn) because we use Union method to do insert, we consider the new node as a forest with one B0 tree

WC ExtractMax is O(logn) because we compaire the root of each tree to find the one with highest incentive and pop it, this takes logn time, since there is at most logn trees. Then we generate new forest with the tree we poped, this takes logn1 time(n1 is the # nodes of the tree we poped, n1 < n). Then we merge the origin tree with the new one, which takes log(n - n1)times. In total, it’ s O(logn)