ENS Lyon Camp. Day 2. Basic group. Cartesian Tree.

26 October

Contents

1 Cartesian Tree. Definition.

`Max Heap` is a binary tree that satisfies the heap property: if v is a parent node of, u then the key of node v is greater than key of u. From definition follows that the root has the maximal key. The similar definition is for `Min heap`. The `Binary Search Tree (BST)` property states that the key in node must be greater than all keys stored in the left subtree, and smaller than all keys in right subtree.

`Cartesian Tree` is a binary tree such that every node stores two values x and y. Value x is named key and satisfies BST property; value y is named priority and satisfies heap property.

2 Cartesian Tree. Construction

How to construct cartesian tree given the pairs (x_i, y_i)? Firstly, the algorithm sorts all pairs by their keys: $x_1 < x_2 < \cdots < x_n$. Then it goes from left to right, if the pair (x_k, y_k) was added on the last step than node stores this pair in the rightest node in the tree. (Because all keys x satisfy the BST property). Algorithm tries to add a new pair (x_{k+1}, y_{k+1}) as a right child of (x_k, y_k) (it is possible that $y_k > y_{k+1}$). Otherwise, algorithm goes up to the parents in the tree until priority

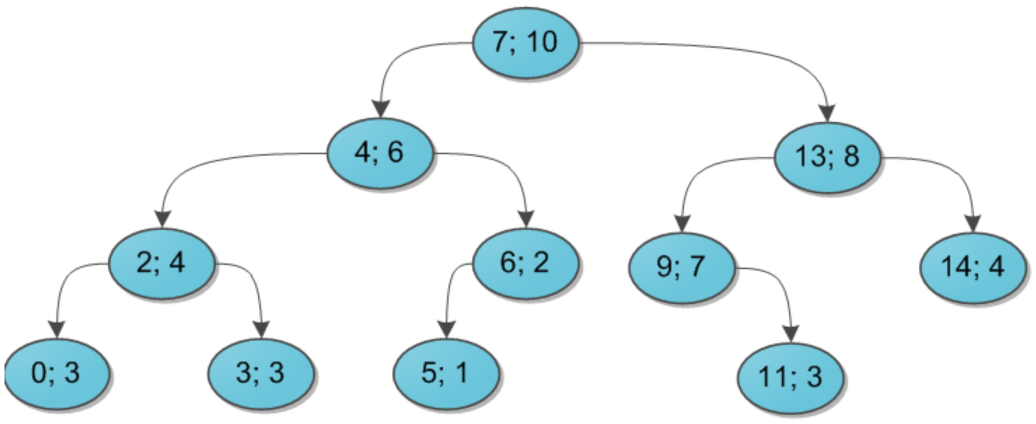

Figure 1: Cartesian Tree.

y_{k+1} is less than priority in current vertex. When the algorithm finds suitable vertex v, the right child of v becomes the left child of new node (x_{k+1}, y_{k+1}), and the new node becomes the new right child of v.

The time complexity of this algorithm is $O(n)$ (without time for sort).

The cartesian tree is stored in this way:

```
struct Node {
    Node* left, right; // left and right subtrees
    int x, y; // key and priority

    static Node* null; // empty tree
}

typedef Node* pNode;
```

3 Cartesian Tree. Operations.

3.1 Split.

This operation allows splitting cartesian tree T by key k into two cartesian trees T_1 and T_2. T_1 contains all keys less than or equal to k and T_2 contains all keys greater than k.

For the sake of brevity $T.L$ and $T.R$ will be the left and the right subtrees of T, orrespondingly.

Imagine the case when the key k is greater than or equal to the key stored in the root:

- Left subtree of T_1 is equal to $T.L$. To find the right subtree of T_1, algorithm splits $T.R$ recursively into T_1' and T_2'; so T_1' is the desired subtree.

- T_2 is equal to T_2'.

The opposite case is processed similarly.

```
void split(pNode T, pNode &T1, pNode &T2, int x) {
    // empty tree
    if (T == Node::null) {
        l = Node::null;
        r = Node::null
    }
    else if (T.x <= k) {
        split(T.right, T.right, T2, x);
        T1 = T;
    }
    else {
        split(T.left, T1, T.left, x);
        T2 = T;
    }
}
```

The time complexity of this operation is $\mathcal{O}(h)$ where h is the height of T.

3.2 Merge.

This operation allows to merge two cartesian trees T_1 and T_2 into the one, with the condition that the biggest key in T_1 is less than or equal to the smallest key in T_2. The resulting tree T contains all keys from T_1 and T_2 trees.

The root of the resulting tree T should have maximal priority y among all other vertices. Of course, it is either root of T_1 or root of T_2 with maximal y. To be specific, we consider the case when root of T_1 has the greater priority than root of T_2. The left subtree of T is left subtree of T_1, and the right subtree is equal to the merge of right subtree of T_1 and T_2.

```
merge(pNode T1, pNode T2, pNode &T) {
    // emtpy T1
    if (T1 == Node::null) {
        T = T2;
        return;
    }
    // empty T2
    if (T2 == Node::null) {
        T = T1;
        return;
    }

    if (T1->y > T2->y) {
        merge(T1->right, T2, T1->right);
        T = T1;
    }
    else {
        merge(T2->left, T1, T2->left);
        T = T2;
        }
    }
```

The time complexity of this operation is $\mathcal{O}(h)$ where h is the maximal height of T_1 and T_2.

3.3 Insert.

This operation insert element *newNode* into the tree T. The new element is the tree, which consists of only one node and tree T with inserting element is the merge of T and new element. But T might contain keys less and greater than key of the new element $newNode.x$. Therefore algorithm should split the tree T by $newNode.x$ before merge.

```
    void insert(pNode &T, pNode newNode) {
        pNode T1, T2;
        split(T, newNode->x, T1, T2);
        merge(T1, newNode, T1);
        merge(T1, T2, T);
    }
```

3.4 Delete.

This operation remove element *node* from the tree T. Algorithm splits the tree T two times to pick out the required element and merges remaining trees.

```
    void delete(pNode &T, pNode node) {
        pNode T1, T2, trash;
        split(T, node->x, T1, T2);
        split(T1, node->x - eps, T1, trash);
        merge(T1, T2, T);
    }
```

4 Treap with implicit key

4.1 Problem statement.

Consider an array with n elements. The queries might be very different. For example, insert element in arbitrary position, remove element in arbitary position or even remove the range of elements. Other queries are reverse or rearrange a range in the array. Of course, this queries should be procedeed effectively. The data structure, which solves this kinds of problems, is named `Treap with implicit key`.

4.2 Definition and Construction

The treap with implicit key is cartesian tree, but each node of it stores *the order number of node* instead of x. It means, if someone arrange nodes in order of BST, x — is the order number of node in this arrangment. Of course this modification maintain the invariant of binary search tree. But there is one problem: insert and delete could change order, so it needs $O(n)$ time to recalculate all keys in the worst case.

Actually the key x is not stored in the tree. Instead of this, each node stores auxiliary value *size*, This value equals to the size of tree in the node, i.e. the total number of vertices in subtree of node including the node. The important observation is that the sum of sizes of all not-visited left subtrees on the path from the root to the node v plus the number of nodes in the left subtree of v equals to the key x of node.

```
struct Node {
    Node* left, right; // left and right subtrees
    int y;
    int size; // size of the tree

    static Node* null; // empty tree

    void recalc() {
        size = 1 + l->size + r->size;
    }
}

typedef Node* pNode;
```

Array is implemented with a treap with implicit key, index of element is a key in treap and the value of element is a priority. Like in array algorithm does not store index, but the order of elements is known.

4.3 Split and Merge.

Now merge operation allows to merge two arbitrary trees, it corresponds to concatenation of arrays. Implementations of merge for cartesian tree does not use the x key. So the implementation in this case is exactly the same, except for recalculation of the sizes.

Split operation divides tree T by key k into two trees T_1 and T_2 in the way that T_1 contains exactly k nodes. It corresponds to dividing array into two parts such that fist part contains exactly k elements.

```
void split(pNode T, pNode &T1, pNode &T2, int k) {
    // empty tree case
    if (T == Node::null) {
        T1 = Node::null;
        T2 = Node::null;
    }

    // left subtree corresponds to prefix of the array with
    // length equals to size of this subtree
    int leftSize = T->left->size;

    // required prefix is shorter or equal
    if (leftSize >= k) {
        split(T->left, k, T1, T->left);
        T->recalc();
```

```
            T2 = T;
        } else { // required prefix is longer
            // algorithm already cuts left subtree and root, so
            // the remaining number of elements to cut is k - leftSize - 1
            split(T->right, k - leftSize - 1, T->right, T2);
            T->recalc();
            T1 = T;
        }
    }
```

To add element into the position i of array, algorithm splits the correspondig tree by the key i into T_1, T_2 and merge three trees: T_1, new element and T_2. To remove element on the position i algorithm finds the corresponding vertex v, and replace v with merge of its children.

4.4 Application

. It is possible to use treap with implicit key to solve RMQ/RSQ problem with the range update. The algorithm keeps two additional fields in node: one for the result of operation and one for inconsistency.

Operation is calculated over all nodes in subtree of vertex, including vertex. It keeps up-to-date in the similar way as *size*, it updates this field using the *size* fields of children. To get result of operation in some range, algorithm cuts the subtree corresponding given range with two splits, return the value from the root and merge all trees back.

For the range update algorithm uses the same invariant as segment tree. Algorithm push inconsistency to children for each visit of the vertex and keeps up-to-date value for vertex that have inconsistensy.

For example, this is the algorithm for reverse problem. The main query is to reverse elements in the given range. We will keep boolean field for inconsistency. The value 1 shows that the corresponding segment should be reversed. And push swaps the left and right subtree and flip boolean flag (xor flag with true).

```
    // call this function at the beginning of split and merge calls
    void push(pNode v) {
        if (!isReversed)
            return;

        swap(v->left, v->right);
        if (v->left != Node::null) // if there exist left subtree
            v->left->isReversed ^= true; // flip inconsistency
        if (v->right != Node::null) // if there exist left subtree
            v->right->isReversed ^= true; // flip inconsistency

        isReversed = false;
    }

    void reverse(pNode &T, int l, int r) {
        pNode T1, T2, R; // R is the range from l to r

        split(T, T1, T2, l - 1);
        split(T2, R, T2, r);

        R->isReversed = true;

        merge(T1, R, T1);
        merge(T1, T2, T);
    }
```