inzva Algorithm Programme 2018-2019

Bundle 8

Data Structures - 2

Editor
Yasin Kaya

Reviewer
Murat Ekici

Contents

1 Introduction

The following section will introduce the basic concepts of Balanced Trees and Lowest Common Ancestor Algorithm, followed by some common problems.

2 Self Balancing Binary Trees

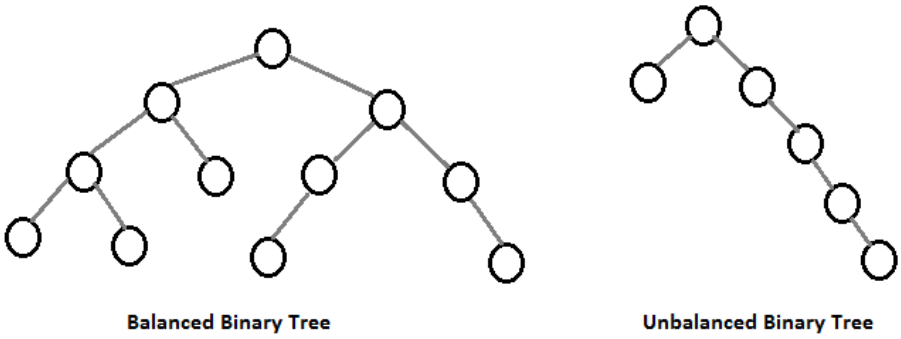

Figure 1: Balanced and Unbalanced Tree Example

Balanced Binary Tree is a Binary Tree such that every node's left and right subtrees' heights differ by 1 at most. Note that the height of a tree is the length of the path from root to the deepest leaf.

Self Balancing Binary Tree is any Binary Tree such that after an insertion or deletion that is made randomly, it keeps the tree balanced automatically.[2]

You've learned insertion and deletion algorithm for Binary Search Trees in Bundle-4.[1] You've also seen that, algorithms might be $O(N)$ in the worst case. This is mainly because those algorithms were not self balancing. After a couple of insertion made intentionally to made the tree unbalanced, the height becomes N instead of $logN$. We will see in the next section some Self Balanced Tree algorithms to overcome insertions that are made on purpose to make tree unbalanced.

3 Treap

Treap is a self balancing BST that is widely used in competitive programming as it is easy to code, other than data structures like AVL, Red-Black Tree...

Every node keeps two values, the former is called *keyvalue* and the latter is called *priorityvalue*. The main idea of treap is that it is constructed like BST according to key values while ensuring that a node's priority is larger than its children' priority(Hence its name is a composition of tree and heap - treap).

Using the advantage of the structure of treap, we can split treap into two treaps or merge two treaps in $O(log(N))$.

3.1 Node

Every node keeps two different values -key and priority- and for tree structure it keeps the left and right child of the node. So we would have a struct like the following code:

```
struct node {
    int key, priority;
    node * L, * R;
    };
```

3.2 Split

Split algorithm, given a treap T and *split key*, splits it to two new treaps called L and R in which L consists of nodes that its key value is lower than or equal to split key value.

Firstly, split operation compares the key of root with split key. If root's key is larger than split key, every node in the root's right child's subtree has a key that is larger than split key. Hence R must involve right child's subtree. On the other hand, there might be some nodes that have a larger key than split key but it is in the subtree of root's left child. Therefore, we proceed to operation by calling split function for root's left child's subtree (Similar operations would be applied if root's key is smaller than split key.). Also note that one doesn't have to check priorities since we didn't change the height order of nodes.

```
void split (node* T, int split_key, node* & L, node* & R) {
    if (!T)
        L = R = NULL;
    else if (split_key < T->key)
        split (T->L, split_key, L, T->L),  R = T;
    else
        split (T->R, split_key, T->R, R),  L = T;
}
```

3.3 Merge

Merge algorithm, given two treaps L and R, combines them into a new treap T, assuming that all nodes' key in L are smaller than minimum key value in R.

While merging two different treaps, we need to ensure that merge operation priorities are in correct order (recall that every node's priority should be smaller than its parents priority). Then, merge

operation compares priority of two child. If L's root has larger priority value than R's root, L's root should be root of combined treap. After that, we merge L's right child with R recursively. Also note that when L's root has smaller priority, R's root would be the new root of combined treap, and we continue to merge R's left child with L.

```
void merge (node* & T, node* L, node* R) {
    if (!L || !R)
        T = L ? L : R;
    else if (L->priority > R->priority)
        merge (L->R, L->R, R),  T = L;
    else
        merge (R->L, L, R->L),  T = R;
}
```

3.4 Insert

Insert operation adds a new node with p_{key} and $p_{priority}$ values to the existing treap T. There are three cases. When T is empty new node becomes root and the only node in T. If T's root's priority value is smaller than $p_{priority}$, we must change the root to new node as it has the maximum priority. Lastly, if the previous cases did not occur, insertion algorithm tries to insert new node to the right place recursively. Note that we need to check key values to find where to insert in third case.

```
void insert (node* & T, node* p) {
    if (!T)
        T = p;
    else if (p->priority > T->priority)
        split (T, it->key, p->L, p->R),  T = p;
    else
        insert (p->key < T->key ? T->L : T->R, p);
}
```

3.5 Erase

Lastly we can erase a node in the treap, by finding node using keys and merging its left and right child.

```
void erase (node* & T, int key) {
    if (T->key == key)
        merge (T, T->L, T->R);
    else
        erase (key < T->key ? T->L : T->R, key);
}
```

3.6 Complexity

Every operation that we mentioned above works in $O(log(N))$. This is mainly because treap has on average $log(N)$ height. You have learned that if BST is randomly generated it has on average $log(N)$(proof of this is out of our scope.). Then, it turns out that treap's expected height is very similar to randomly generated BST's expected height when we assign the priority values randomly. Hence treap is also called randomized binary search tree.

4 AVL Tree

4.1 Tree Rotation

Before we get started, we should know what rotation in trees is similar to what we used in AVL Tree often. Main purpose of rotation is to decrease the height difference of tree while not changing the order of keys. In other words, original tree's and rotated tree's inorder traversal are the same. There are two types of rotation called left and right rotation.

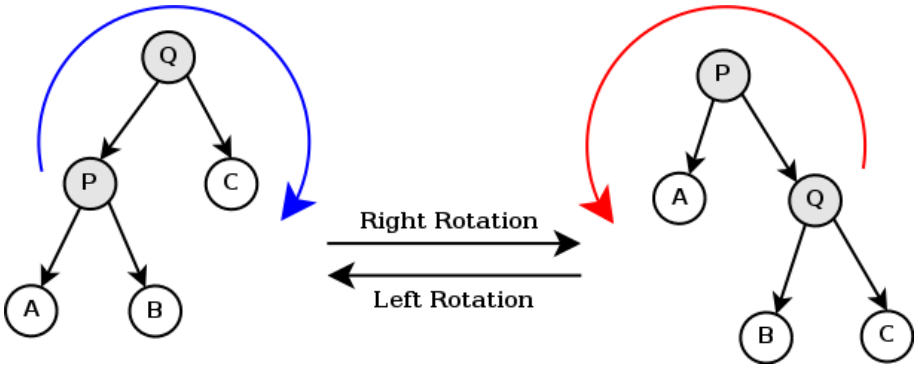

Figure 2: Left and Right Rotation[3]

Operations in Left Rotation:

- Let P be Q's left child.
- Set Q's left child to be P's right child.
- Set P's right child to be Q.

Operations in Right Rotation:

- Let Q be P's right child.
- Set P's right child to be Q's left child.
- Set Q's left child to be P.

4.2 Insertion

Insertion is similar to normal BST insertion in that we use keys to find the leaf where the new node should be. However, as we saw earlier this might disrupt the balanced structure of tree. To overcome this issue, we have to check every ancestor of a new added node and use rotations that make the tree balanced again. There occurs 4 cases when balancing a subtree.

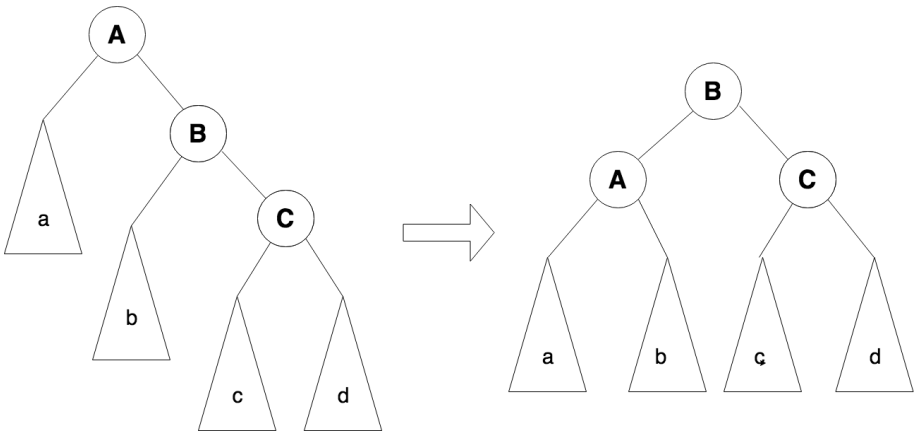

Figure 3: Left Rotation when it is right weighted

Case 1 & 2:

Let root A, root's right child B, root's right child's right child C.

When root's right child's and left child's height differ by two, it is unbalanced. At the same time, if B's height is larger than root's left child's height and right C's height is also larger than B's left child's height, it is called right weighted (left weighted is similarly defined).

If tree is right weighted, for the tree to be balanced again we can left rotate the tree (You can check 4.2) . And if tree is left weighted, we can right rotate to balance the tree again.

Case 3 & 4:

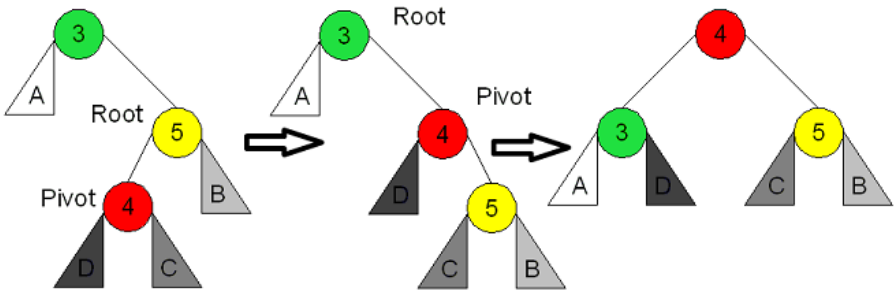

Figure 4: Left-Right Rotation

Let root 3, root's right child 5 and root's right child's left child 4.

When 5's height is larger than root's left child's height and 4's height is larger than 5's height, it is indeed unbalanced; however, it is not right or left weighted. Therefore we can firstly left rotate the tree with rooted 5. Now we have a tree that is right weighted. It is exactly the tree that we saw in case 1 & 2. Hence after right rotation is applied to the tree, it becomes a balanced tree (The same procedure would be applied in right-left rotation).

4.3 Deletion

You've learned a node deletion BST. In AVL tree, we delete the node using the same algorithm in BST. Then if the tree becomes unbalanced, we can balance the tree using left,right,left-right or right-left rotations as we did in insertion.

4.4 Complexity

After every operation, we ensured that tree is balanced. Thus, tree's height is $log(N)$ and operations insertion and deletion works in $O(log(N))$ where N is the number of nodes in the tree.

5 Red Black Tree

Red Black Tree has four fundamental rules that every RBT should follow.

1. Every node has a red or black color.
2. Root's color is always black.
3. There does not exist adjacent nodes such that they have both red color.
4. Number of black colored nodes from root to any leaf is equal.

5.1 Insertion and Deletion

Insertion and deletion operations are similar to operations in BST. If resulted tree does not follow the rules given above, we firstly check if recoloring the some nodes works. If it does not work, we rotate the tree similar to AVL tree.

We will not dive into recoloring and rotating the tree, as it is complex and hard to code.

5.2 Complexity

You might have realised that Red Black Tree does not have to be balanced. However it turns out that insertion and deletion works in $O(log(N))$. We can show that by observing that number of black nodes in a path from root does not exceed $log(N)$ thanks to rule #4; and by rule #3, the number of red nodes does not exceed $log(N)$. So height is maximum $2 \times log(N)$, hence insertion and deletion is $O(log(N))$ despite the fact that RBT is not balanced.[6]

6 Self Balancing Trees - Review

There is no precise rule specifying when to use Red Black Tree or AVL Tree. However, AVL Trees are usually preferred when there are many queries, since it is more strictly balanced. But one should not forget that constant factor from AVL Tree might result in slower results.[10]

Usually, when we don't need to modify self balancing tree, there are some pre-coded self balancing trees in C++, namely $std :: set$ and $std :: map$.[7]

7 Level Ancestor

Now, we are done with self balancing trees and we'll look into some problems about general trees. One of the common problems is level ancestor. This problem consists of queries, specifically $LA(v, d)$, and asks about v's ancestor whose depth equals d.[8]

7.1 Naive Method

Naive method is the first method that comes to mind. We will at first calculate the depths of every node. Then in queries, we'll try to find the answer by visiting the parent of current node until we reach a node whose depth is d.

Its time complexity would be $O(N)$ per query, while memory complexity would be $O(N)$.

7.2 Jump Pointer Method

At first, we will pre-calculate a node's 2^ith ancestor. So we need to keep about $log(N)$ ancestors for each node. After that, in queries, we can find the answer by jumping to the 2^ith ancestor iteratively $log(N)$ times at most. We have learned only an insight about the algorithm; we will see the implementation in LCA algorithm.

7.3 Ladder Method - Longest-Path Decomposition

Ladder Method's main idea is dividing tree into paths. It starts with finding longest path from root to a leaf. After removing the path, we recursively find another longest path from new emerging trees to leaf. After assigning unique paths to the nodes, we pre-calculate all node's depth inside the paths. In queries we first check whether the answer of query is inside the path in $O(1)$. If it is not in the same path with current node, we check if it is in the above path until we find the right answer. These operations costs $sqrt(N)$.

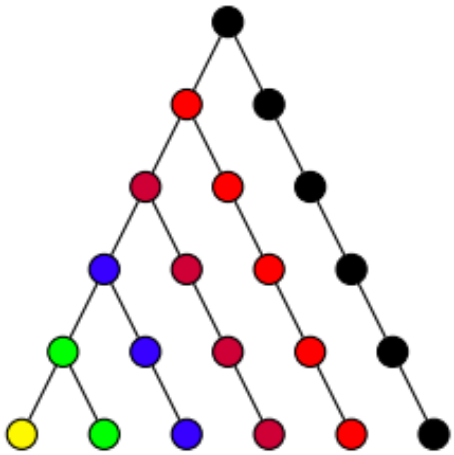

Figure 5: Worst case in Longest-Path Decomposition[9]

7.4 Ladder Method - Ladder Decomposition

It turns out that we can improve the time complexity of Ladder Method with the following trick. After we find the paths and pre-calculate the all node's depth inside the paths, we also pre-calculate next h element's depth above the path, where h is the length of a given path. Notice that, every node are still assigned to a unique path and paths are the same as the previous method, but we doubled the data about depths. Now, when we query $LA(v, d)$, we check if it is in the same path or one of the next h node above the path in $O(1)$. If it is not we check if it is in the above path until we find the result. This time when we jump to the a node that is above, current height is at least doubled, since current node's height is smaller or equal to height of path and we jump to the node where its height equals to $2 \times h + 1$ (h is height of path). Hence its complexity $O(log(N))$.

8 Lowest Common Ancestor

Another common problem is Lowest Common Ancestor problem. This problem consists of queries, $LCA(x, y)$, and asks for the ancestor of both x and y whose depth is maximum. We will use a similar algorithm to the jump pointer algorithm with implementation.

8.1 Initialization

As we did in Jump Pointer Method, we will calculate node's all 2^i. ancestors if they exist. $L[x][y]$ corresponds to x's 2^y. ancestors. Hence $L[x][0]$ is basically the parent of x.

```
void init(){
    for(int x=1 ; x<=n ; x++)
        L[x][0] = parent[x];

    for(int y=1 ; y<=logN ; y++)
        for(int x=1 ; x<=n ; x++)
            L[x][y] = L[L[x][y-1]][y-1];

}
```

Note that we have used the fact that x's 2^y. ancestor is x's 2^{y-1}. ancestor's 2^{y-1}. ancestor.

8.2 Queries-Binary Lifting

Given $LCA(x, y)$, we calculate answer by following:

Firstly, ensure that both x and y are in same depth. If it is not take the deepest one to the other one's depth. Then control whether x and y is equal. If they are equal, that means the lowest common ancestor is x. After that, from $i = log(N)$, check that if x's 2^i. ancestor is equal to y's 2^i. ancestor. If they are not equal that means LCA is somewhere above the 2^i. ancestors of x and y. Then we continue to search LCA of y and x's ancestors as $LCA(L[x][i], L[y][i])$ is the same as $LCA(x, y)$. Please notice that we have ensured that depth difference between LCA and both x and y are no longer larger than 2^i. If we apply this producure until $i = 0$, we would left with x and y such that parent of x is LCA. Of course, the parent of y would also be LCA.

```
int LCA(int x,int y){

    if(dep[x] < dep[y])
        swap(x,y);                    //Ensuring that x is the deepest node.

    int depthDif = dep[y] - dep[x];

    for(int i=0 ; i<=logN ; i++)
        if((depthDif >> i)&1)         //This line means we can jump x to its 2^i. ancestor.
            x = L[x][i];

    if(x == y)
        return x;

    // Now we have to jump x and y until they may not jump no more.

    for(int i=logN ; i>=0 ; i--)
        if(L[x][i] != L[y][i]){
            x = L[x][i];
            y = L[y][i];
        }

    return L[x][0];
}
```

References

[1] "inzva Algorithm Programme Graph1/Bundle-4". github link to pdf: github

[2] Wikipedia link for Self Balancing Binary Search Tree

[3] Link to the figure 1

[4] Link to the figure 2

[5] Link to the figure 3

[6] Red Black Tree - geeksforgeeks

[7] "inzva Algorithm Programme Graph1/Bundle-4". github link to pdf: github

[8] Level Ancestor Problem

[9] Link to the figure 4

[10] https://www.geeksforgeeks.org/red-black-tree-vs-avl-tree/