AVL Tree

Preface

- On average, a normal BST has height ***h = O(log n)***
  - ***O(log n)*** insert, delete

- However, in the worst case, a BST may degrade to a linked list
  - ***h = O(n)***
  - ***O(n)*** insert, delete

- AVL tree is a self-balancing tree that can keep it height in ***O(log n)***

Review：Basic operation of BST - Insertion

Insert node ***z*** to the subtree of ***x***.

Note that the implementation allows nodes with duplicate value.

```
void insert(Node *&x, Node *z){
    if(x == NULL){
        x = z;
        return;
    }
    if(x->key < z->key){
        if(x->r)
            insert(x->r, z);
        else
            x->r = z, z->pa = x;
    }
    else{
        if(x->l)
            insert(x->l, z);
        else
            x->l = z, z->pa = x;
    }
}
```

Review：Basic operation of BST - Transplant

Replace node ***x***'s position in the BST with ***y***.

```
void transplant(Node *x, Node *y){
    if(!x->pa)
        root = y;
    else if(x->pa->l == x)
        x->pa->l = y;
    else
        x->pa->r = y;
    if(y)
        y->pa = x->pa;
}
```

Review：Basic operation of BST - Find min

Find the node with minimum key in the subtree of ***x***.

```
Node* find_min(Node *x){
    while(x->l)
        x = x->l;
    return x;
}
```

Review：Basic operation of BST - Deletion

Delete one node with key ***key*** in the subtree of ***x***.

Think about it:
How to delete **all** nodes with key ***key***?

```
void deletion(Node *x, int key){
    if(x == NULL)
        return;
    if(key < x->key)
        deletion(x->l, key);
    else if(key > x->key)
        deletion(x->r, key);
    else{
        if(!(x->l) && !(x->r)){
            transplant(x, NULL);
            return;
        }
        else if(!(x->l))
            transplant(x, x->r);
        else if(!(x->r))
            transplant(x, x->l);
        else{
            Node* y = find_min(x->r);
            deletion(x->r, y->key);
            x->key = y->key;
        }
    }
}
```

Review：Basic operation of BST - Find lower bound

Find the node with minimum key that greater or equal to ***key*** in the subtree of ***x***.

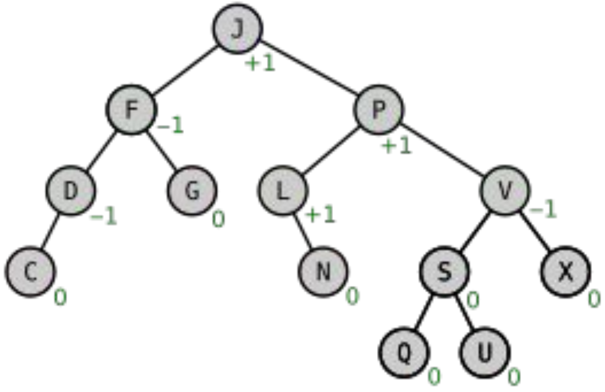
```
Node *find_lower_bound(Node *x, int key){
    if(!x)
        return x;
    if(x->key >= key){
        auto left = find_lower_bound(x->l, key);
        return (left ? left : x);
    }
    return find_lower_bound(x->r, key);
}
```

Balance Factor

- For each node *x*, we define the balance factor ***bf(x)*** to be:
  - ***bf(x) = height(x's left child) - height(x's right child)***

- The concept of AVL tree is to keep ***|bf(x)| ≤ 1***, for all the nodes ***x***.

- Theorem: The height of an AVL Tree = ***O(log n)***



The structure of AVL Tree node

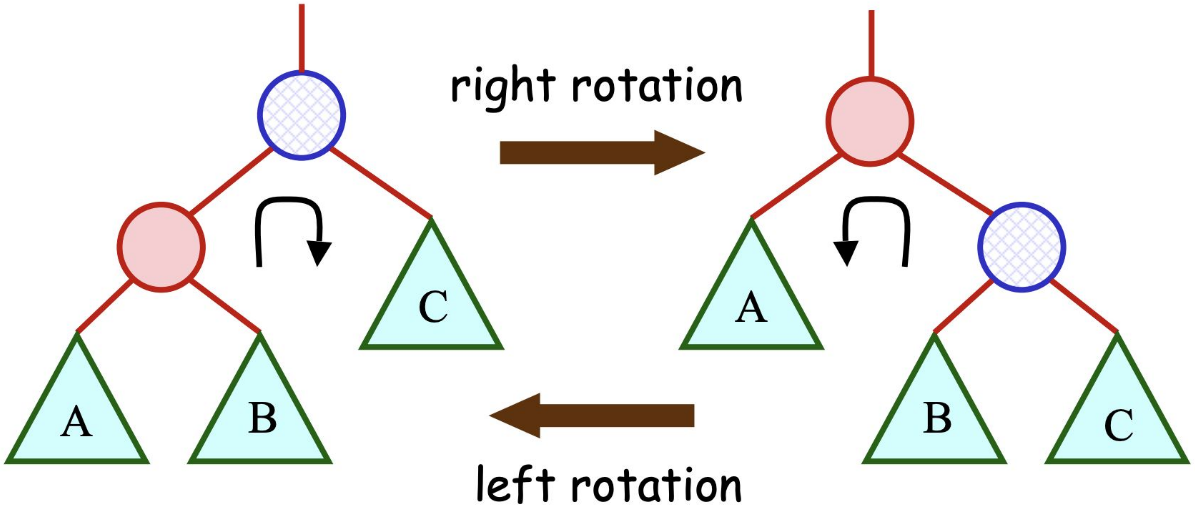
```
struct Node {
    int key, bf, h;
    Node *l, *r, *pa;

    void update(){
        int lh = (l ? l->h : -1);
        int rh = (r ? r->h : -1);
        h = max(lh, rh) + 1;
        bf = lh - rh;
    }

    Node(){}
    Node(int _key): key(_key), bf(0), h(0), pa(NULL), l(NULL), r(NULL) {}
};
```

Rotation

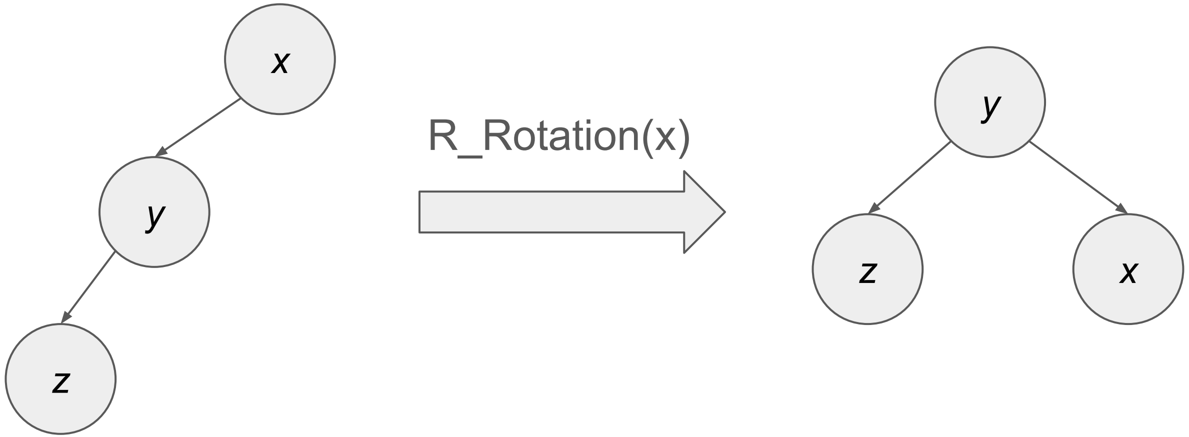
To keep the tree balanced, we introduce a operation “rotation”



Rotation

When a tree is left skewed, we can usually do right rotation ***on x*** to make it balance.

Similarly, we can do left rotation when the tree is right skewed.



Rotation

```
void right_rotate(Node *x){
    Node *left_child = x->l;

    transplant(x, left_child);

    x->l = left_child->r;
    if(left_child->r)
        left_child->r->pa = x;

    left_child->r = x;
    x->pa = left_child;

    x->update();
    left_child->update();
}
```

```
void left_rotate(Node *x){
    Node *right_child = x->r;

    transplant(x, right_child);

    x->r = right_child->l;
    if(right_child->l)
        right_child->l->pa = x;

    right_child->l = x;
    x->pa = right_child;

    x->update();
    right_child->update();
}
```

Insertion

We insert a node like a normal binary search tree.

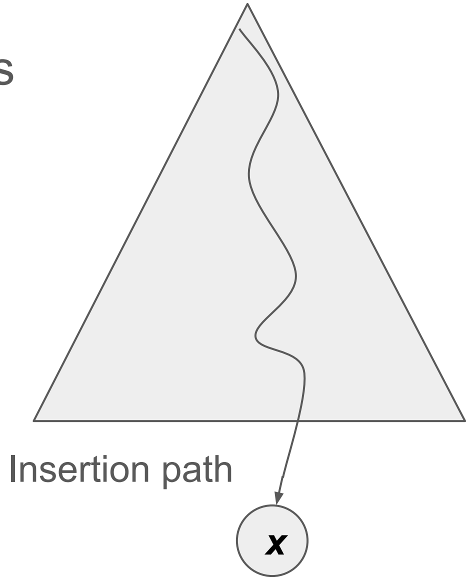
Moreover, we may need to do some rotations to keep the tree balanced.

Insertion

We do it from bottom to top.

That is, after inserting a node ***x***, we do operations to keep ***x***'s subtree balanced. Then do operations to keep ***x***'s parent ***p***'s subtree balanced, an so on.

Until we reach the root.



Insertion

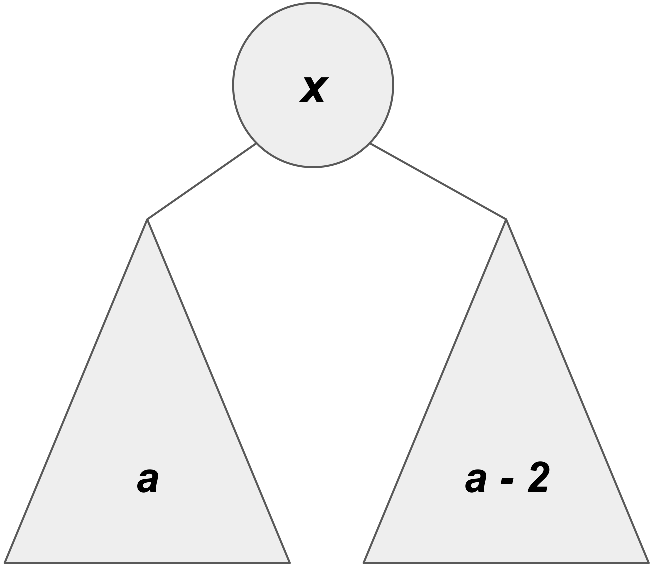
Case 1.

After insertion, ***|bf(x)| ≤ 1*** . Do nothing.

Case 2.

bf(x) = 2, we divide into several cases.

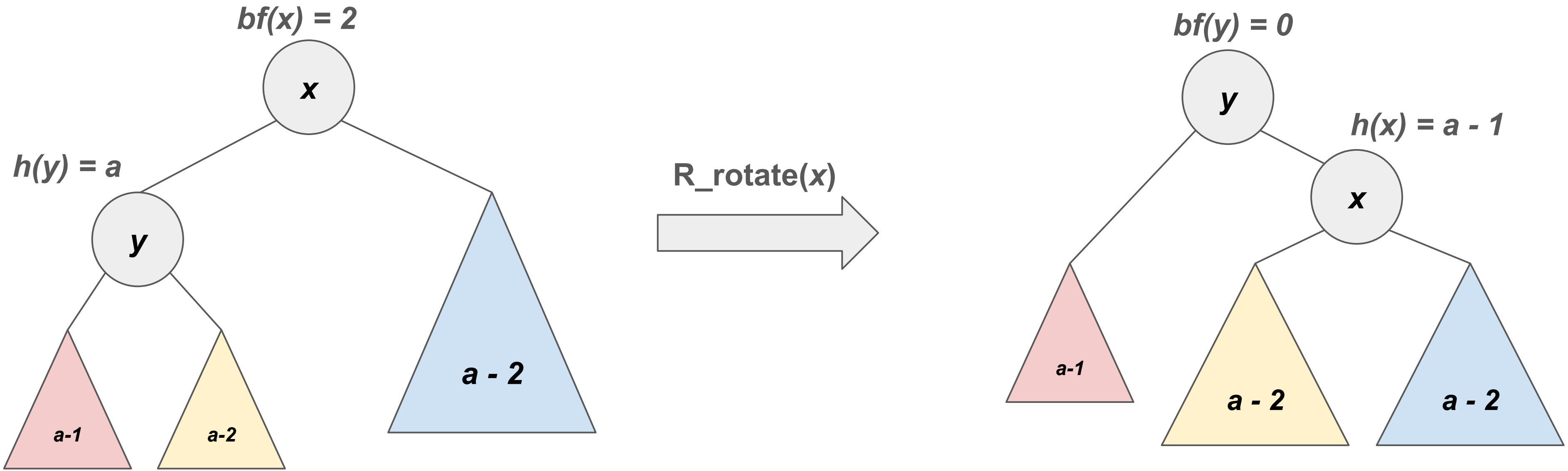
(After insertion, ***|bf(x)| ≤ 2*** must hold. Why?)



Insertion

Case 2-1.

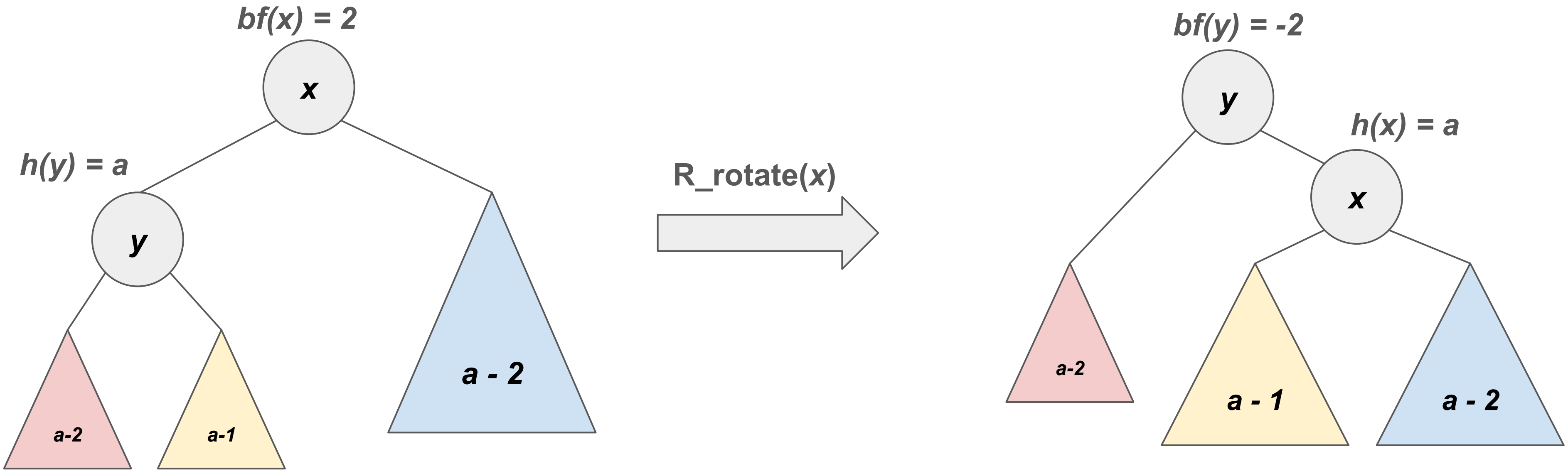
bf(x's left child) = 1, do right rotation ***on x***



Insertion

Case 2-2.

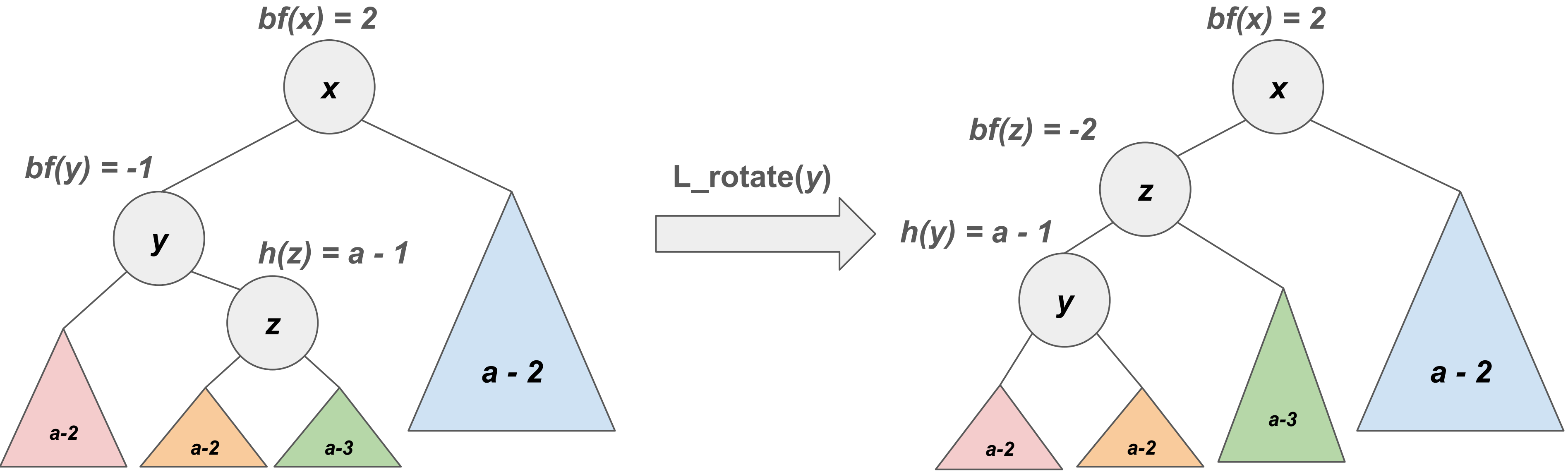
bf(x's left child) = -1, **if** we do right rotation ***on x***…



Insertion

Case 2-2.

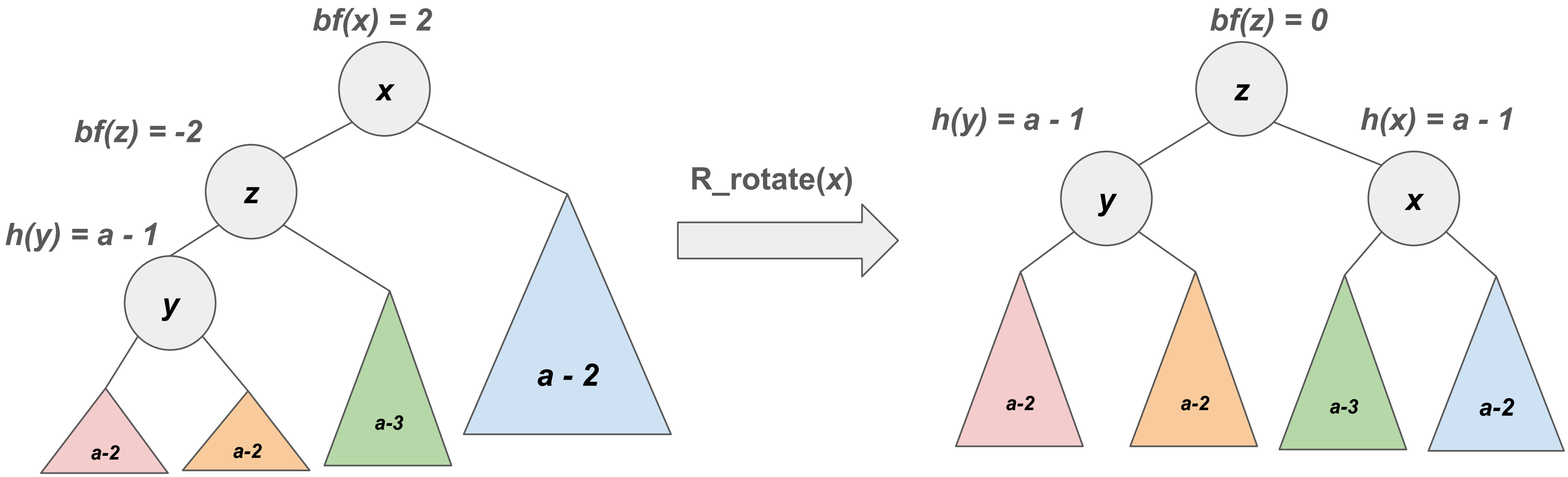
bf(x's left child) = -1, performing left rotation ***on y*** first (***bf(z) = -2***, but is ok).



Insertion

Case 2-2.

bf(x's left child) = -1, then perform right rotation ***on x***.



Insertion

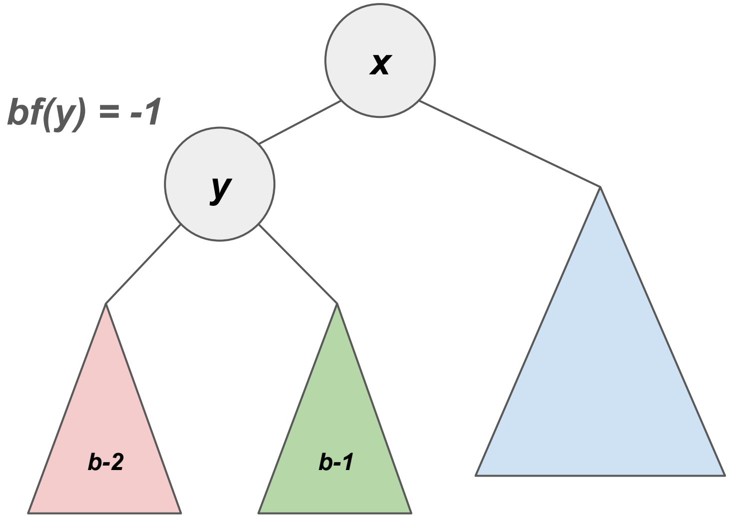
Case 2-3.

bf(x's left child) = 0, no such situation.

If the inserted node is in red subtree, then ***bf(y) = -1*** before insertion.

After insertion, ***h(y)*** doesn't change, and so does ***bf(x)*** → ***bf(x) ≠ 2***.

Same if the inserted node is in green subtree.

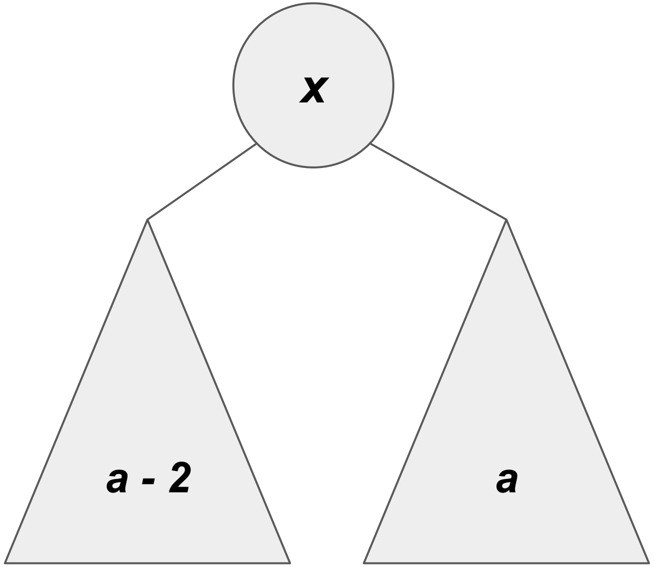


Insertion

Case 3.

bf(x)* = *-2. Symmetric to Case 2.

Modify Case 2 **left** → **right**, **right** → **left**

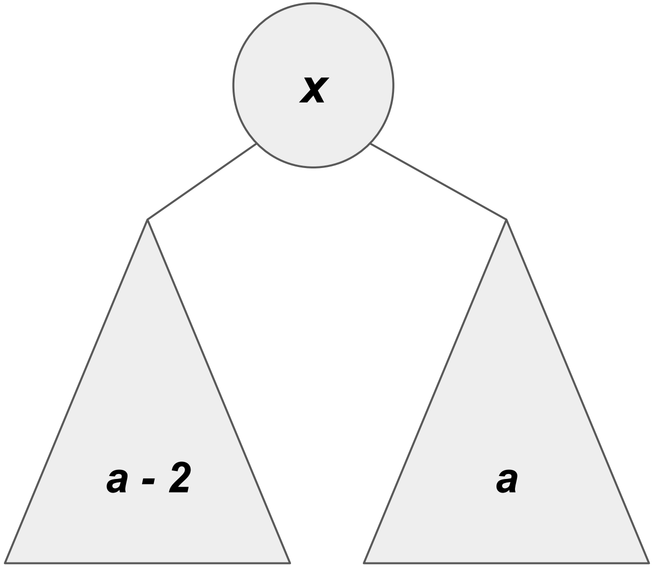


Insertion

```
void insert(Node *&x, Node *z){
    // ... same as the code above
    x->update();
    // case 2
    if(x->bf == 2){
        // case 2-1
        if(x->l->bf == 1)
            right_rotate(x);
        // case 2-2
        else if(x->l->bf == -1)
            left_rotate(x->l), right_rotate(x);
    }
    // case 3
    else if(x->bf == -2){
        // symmetric to case 2
        if(x->r->bf == -1)
            left_rotate(x);
        else if(x->r->bf == 1)
            right_rotate(x->r), left_rotate(x);
    }
    // using assert to make sure your code is correct
    assert(abs(x->bf) < 2);
}
```

Delete

Similarly, we adopt the same method of insertion to keep the tree balanced after deleting a node.



Delete

- Case 1: $|bf(x)| \leq 1$. Do nothing

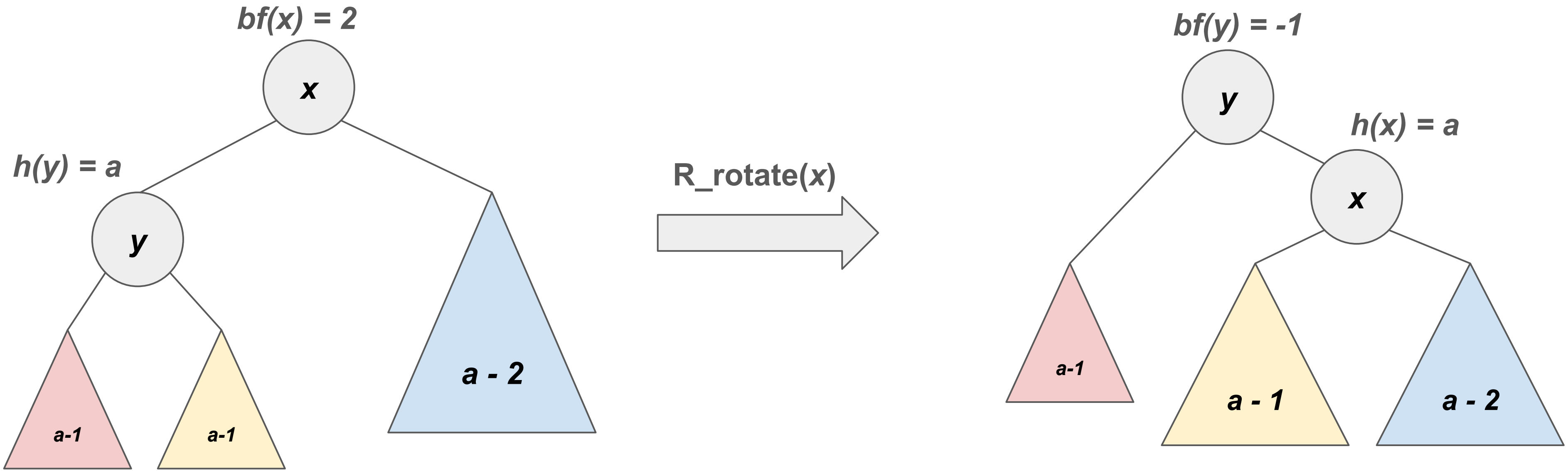
- Case 2: ***bf(x) = 2***
  - 2-1: ***bf(x's left child) = 1***, same as insertion
  - 2-2: ***bf(x's left child) = -1***, same as insertion
  - 2-3: ***bf(x's left child) = 0***, do right rotation on ***x***

- Case 3. ***bf(x) = -2***. Symmetric to case 2.

Delete

Case 2-3.

bf(x's left child) = 0, do right rotation ***on x***



Delete

```
void deletion(Node *x, int key){
    // ... same as the code above
    x->update();
    // case 2
    if(x->bf == 2){
        // case 2-1, 2-3(x->l->bf == 0)
        if(x->l->bf >= 0)
            right_rotate(x);
        // case 2-2
        else if(x->l->bf == -1)
            left_rotate(x->l), right_rotate(x);
    }
    else if(x->bf == -2){
        // symmetric to case 2
        if(x->r->bf <= 0)
            left_rotate(x);
        else if(x->r->bf == 1)
            right_rotate(x->r), left_rotate(x);
    }
    // using assert to make sure your code is corect
    assert(abs(x->bf) < 2);
}
```

Lab - Order of key

Maintain an integer set ***S***, which support the following operations:

- Insert ***k*** into ***S***

- Delete ***k*** from ***S***

- Find the number of elements in ***S*** whose value is smaller or equal to ***k***

Lab - Order of key

The first and second operations can done easily.

How about the third operation ?

Lab - Order of key

We can use the recursive function **order_of_key** to solve it!

(How to calculate the subtree size of a node ?)

```
int order_of_key(Node *x, int key){
    if(x == NULL)
        return 0;
    if(x->key <= key)
        return 1 + (x->l ? x->l->sz : 0) + order_of_key(x->r, key);
    return order_of_key(x->l, key);
}
```

Lab - Order of key

[Solution Code]