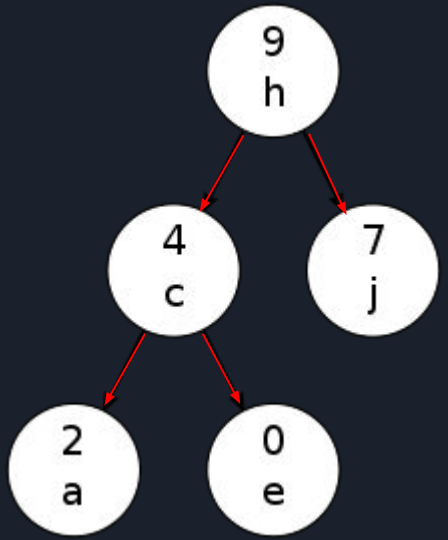
TREAP

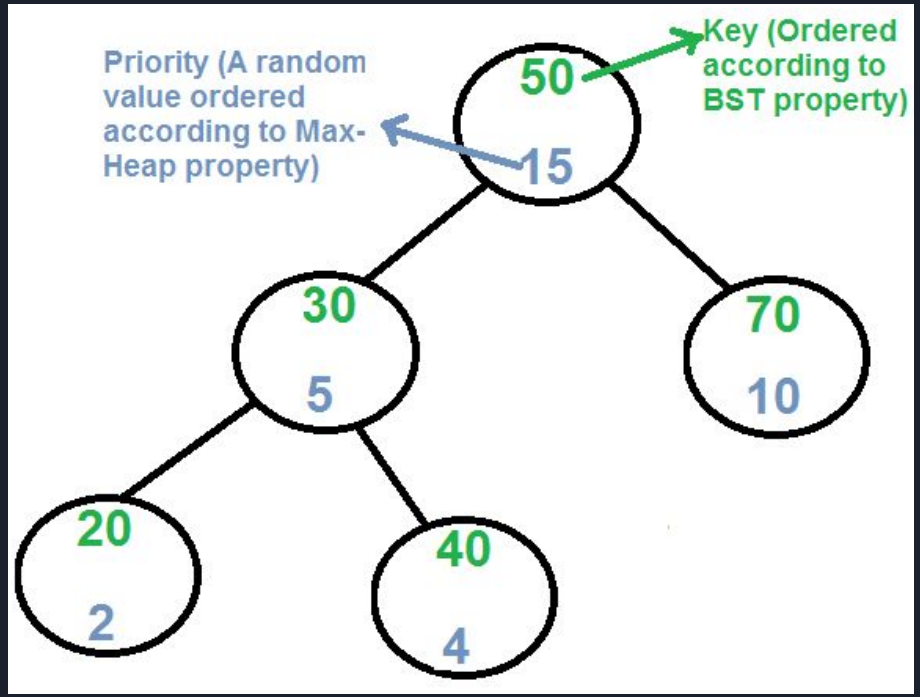


Alonso Barrios
Christian Rojas
Massimo Imparato

Structure

- Treap is a Data Structure which combine BST and Heap.

- Each node has key (BST) and priority value (Heap).
  - **Key:** follow BST ordering.
  - **Priority:** randomly assigned value that follows Heap property (child nodes can't be higher/lower than parent).

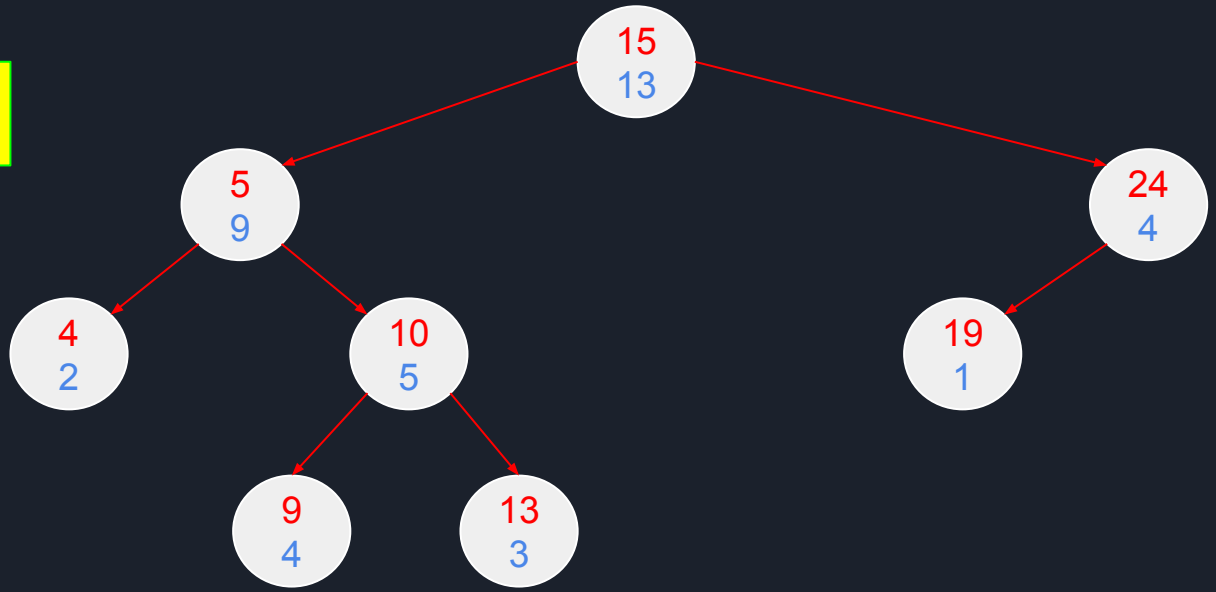


Methods and Complexity

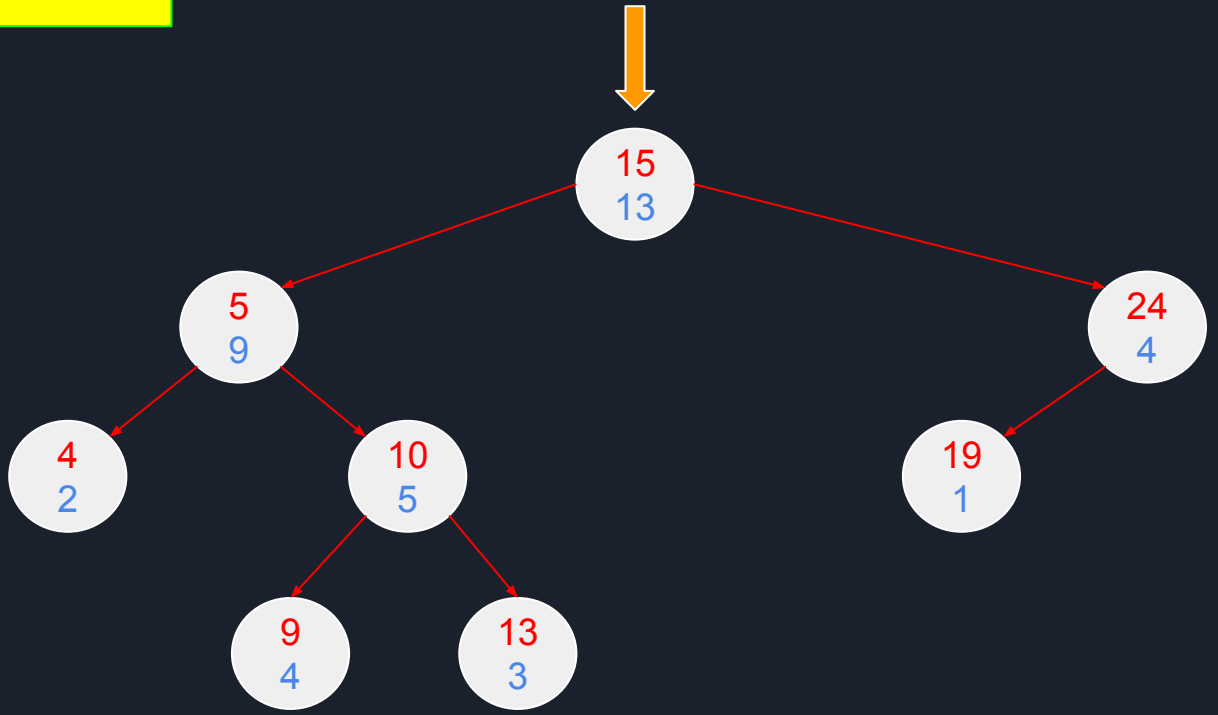
Search

- Find like BST using key.
- Complexity O(log (n))

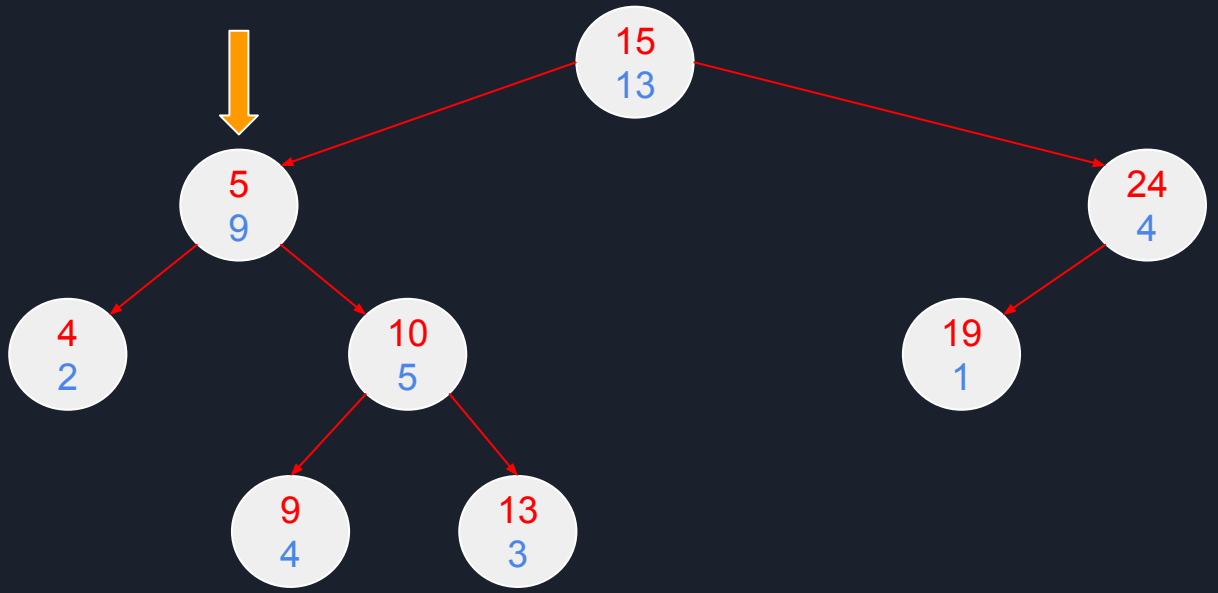
Eg. Find (10)



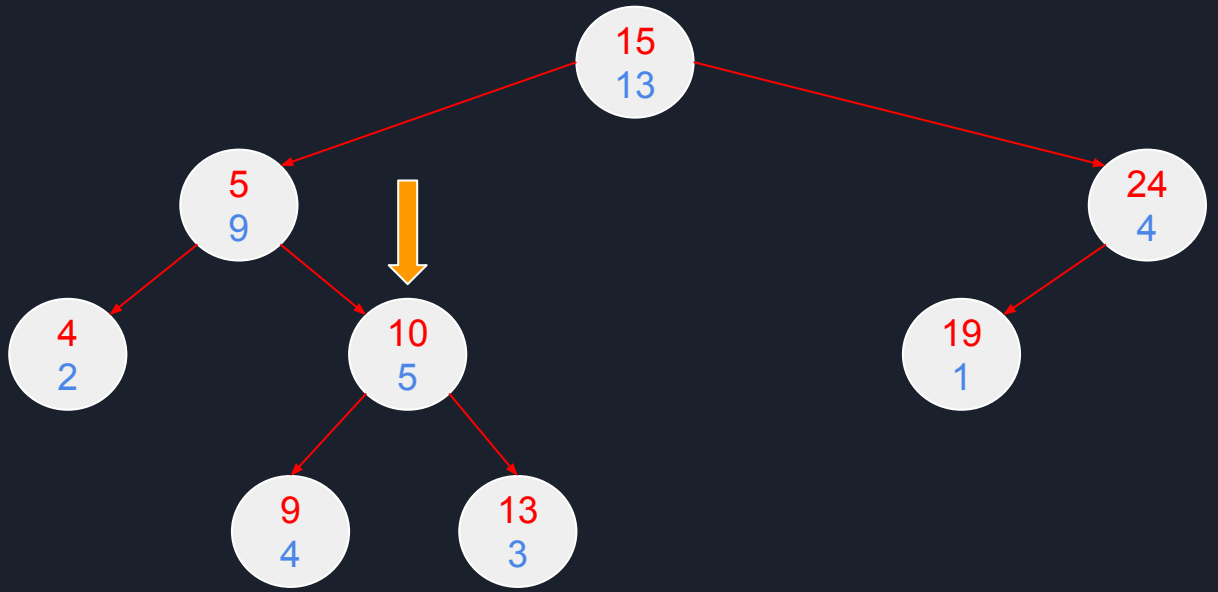
Eg. Find (10)



Eg. Find (10)



Eg. Find (10)

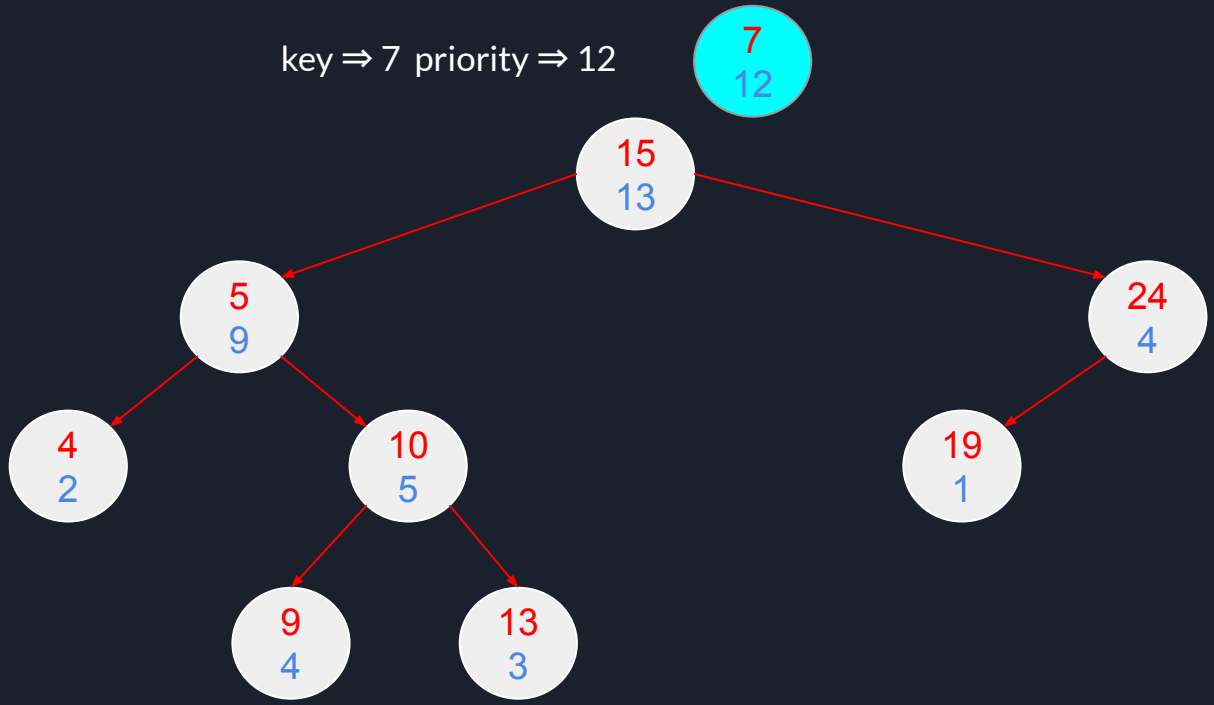


Insert

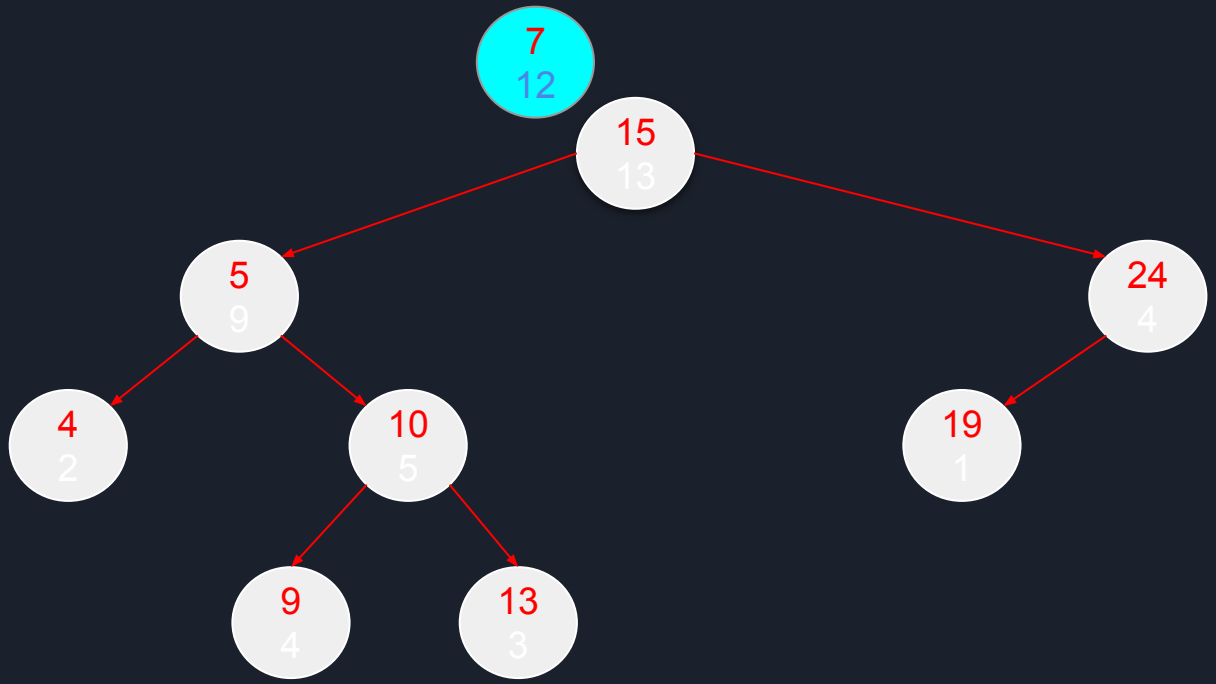
- Insert as BST ⇒ O (log (n))
- Use rotation to keep Heap property ⇒ O (1)
- Complexity O(log (n))

Insert (Eg. Insert(7, random))

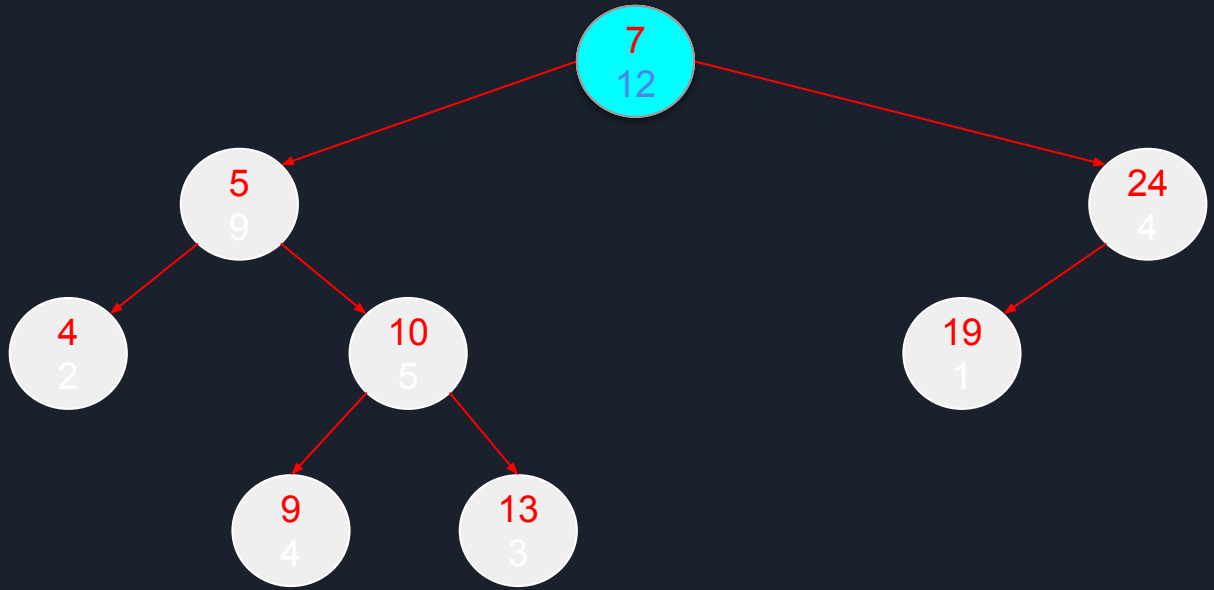
key ⇒ 7 priority ⇒ 12



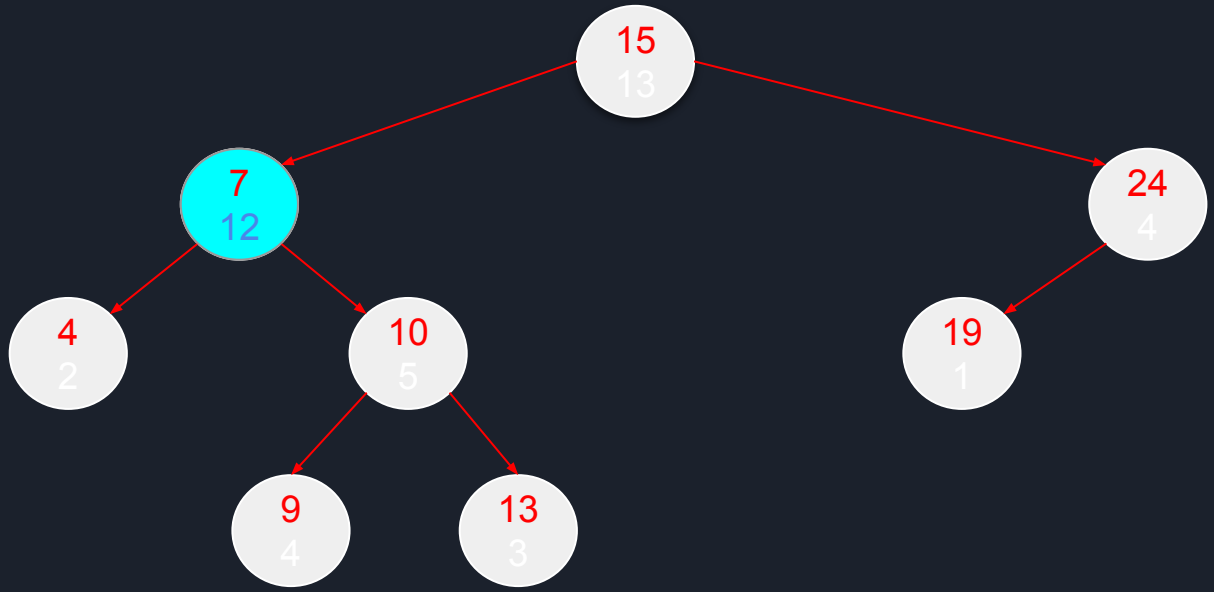
Step 1: Ignore **priority** and insert as BST



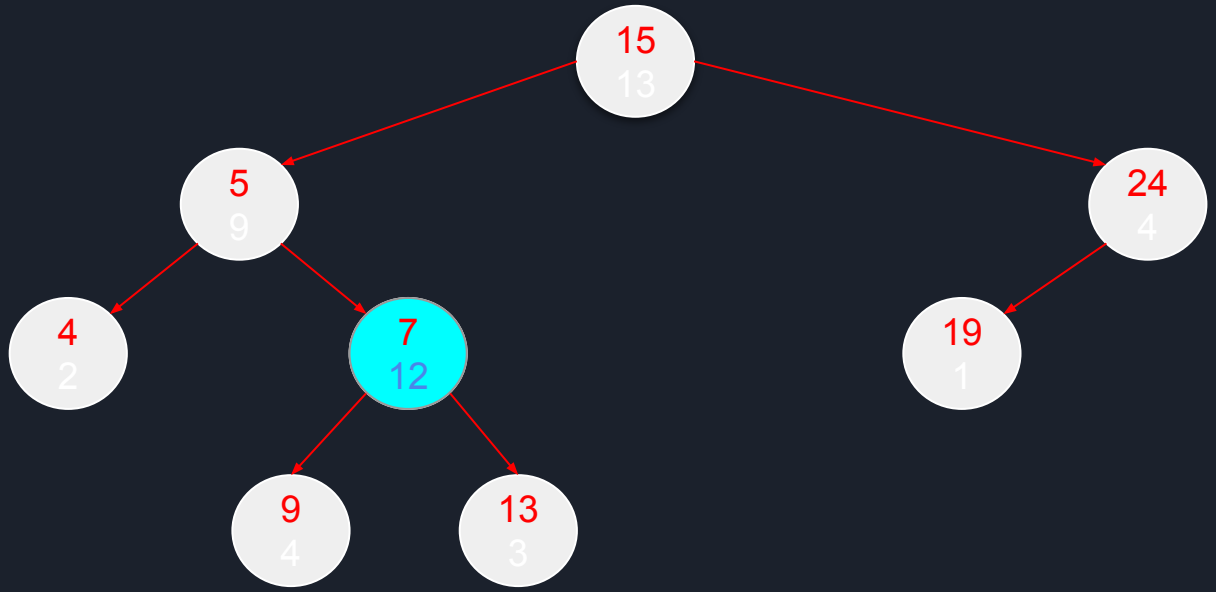
Step 1: Ignore **priority** and insert as BST



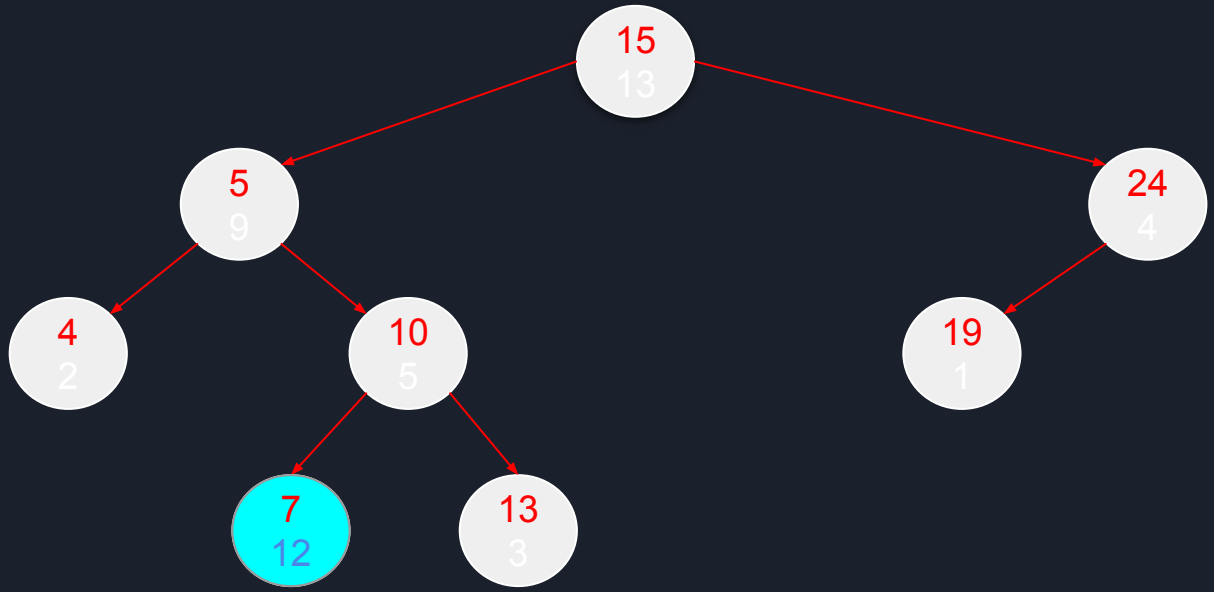
Step 1: Ignore **priority** and insert as BST



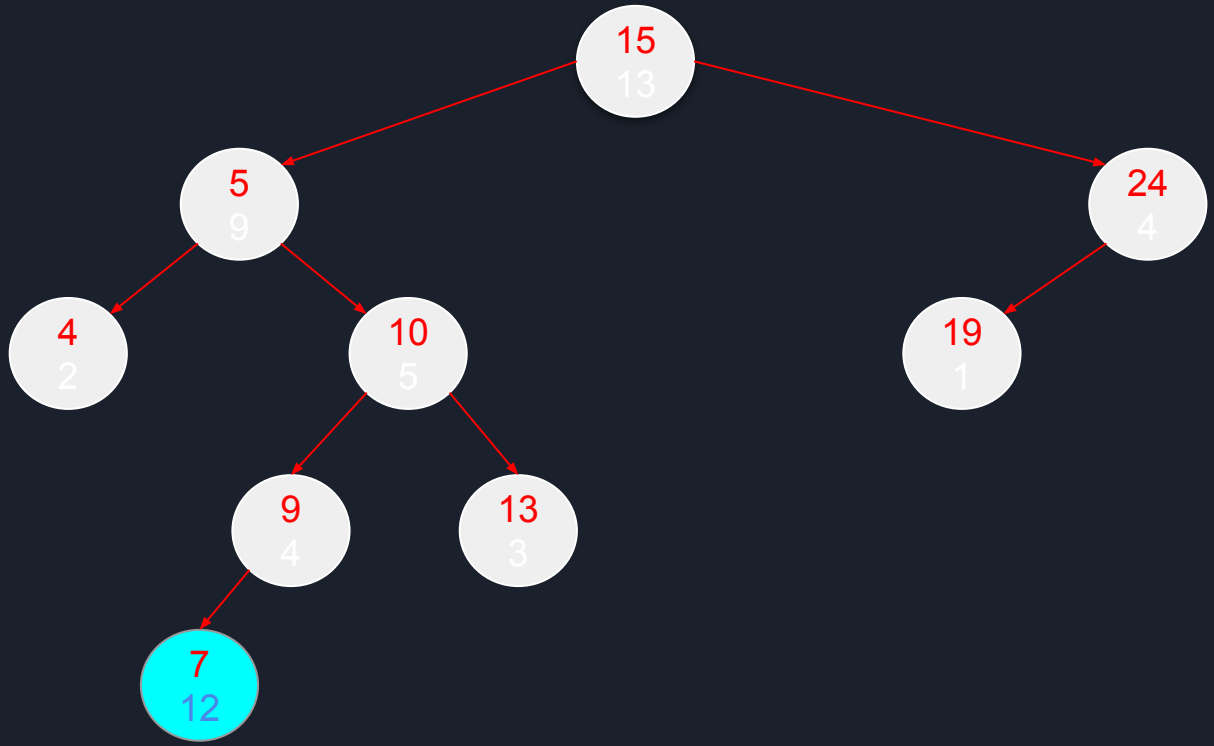
Step 1: Ignore **priority** and insert as BST



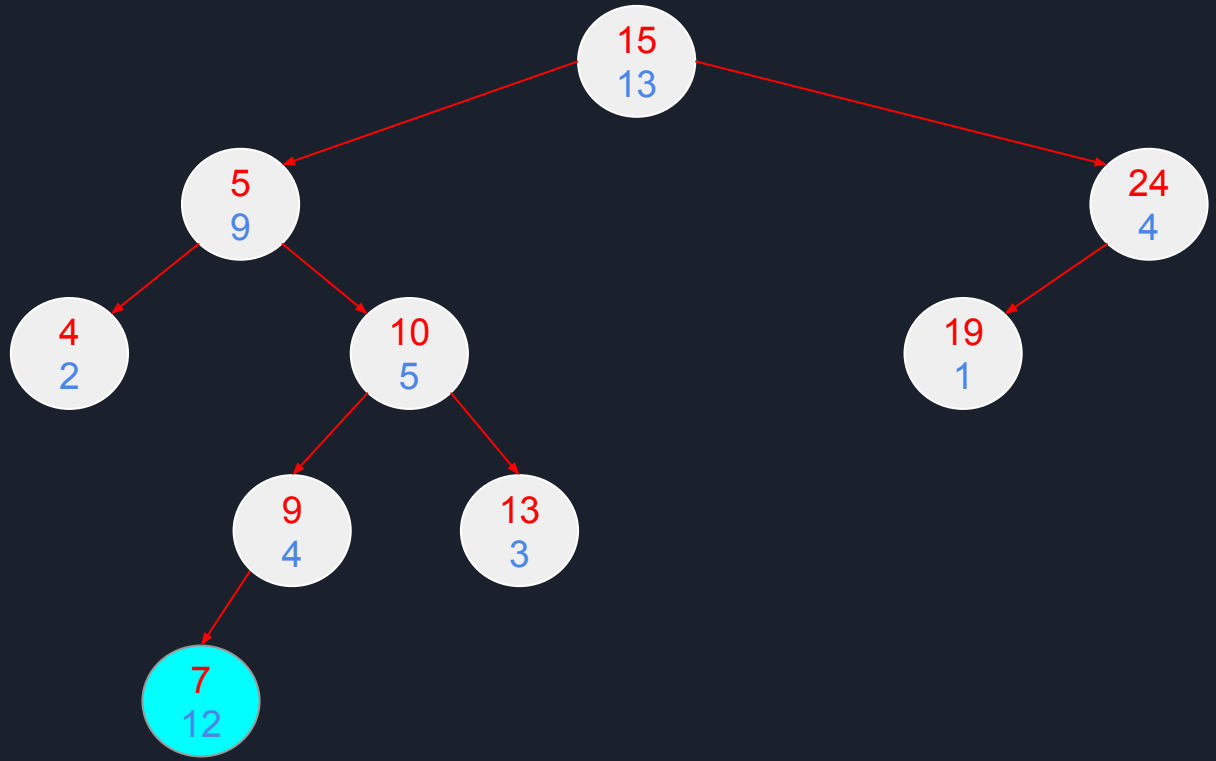
Step 1: Ignore **priority** and insert as BST



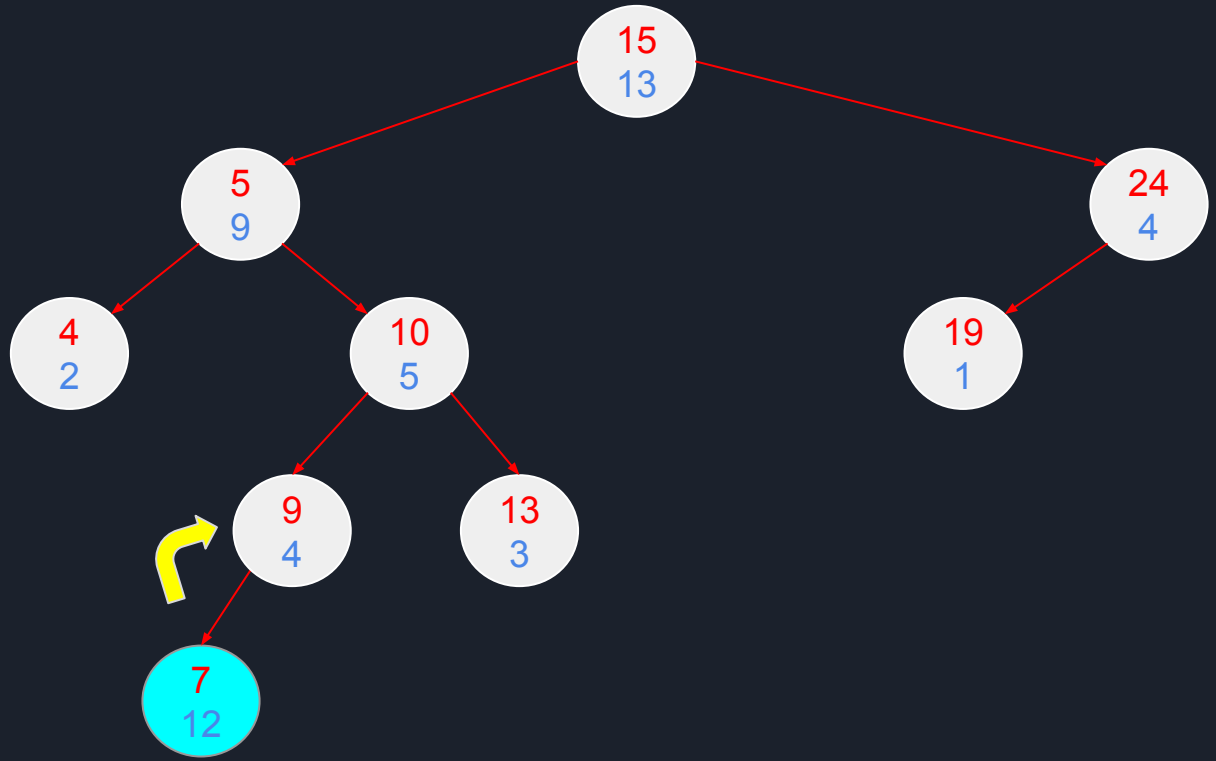
Step 1: Ignore **priority** and insert as BST



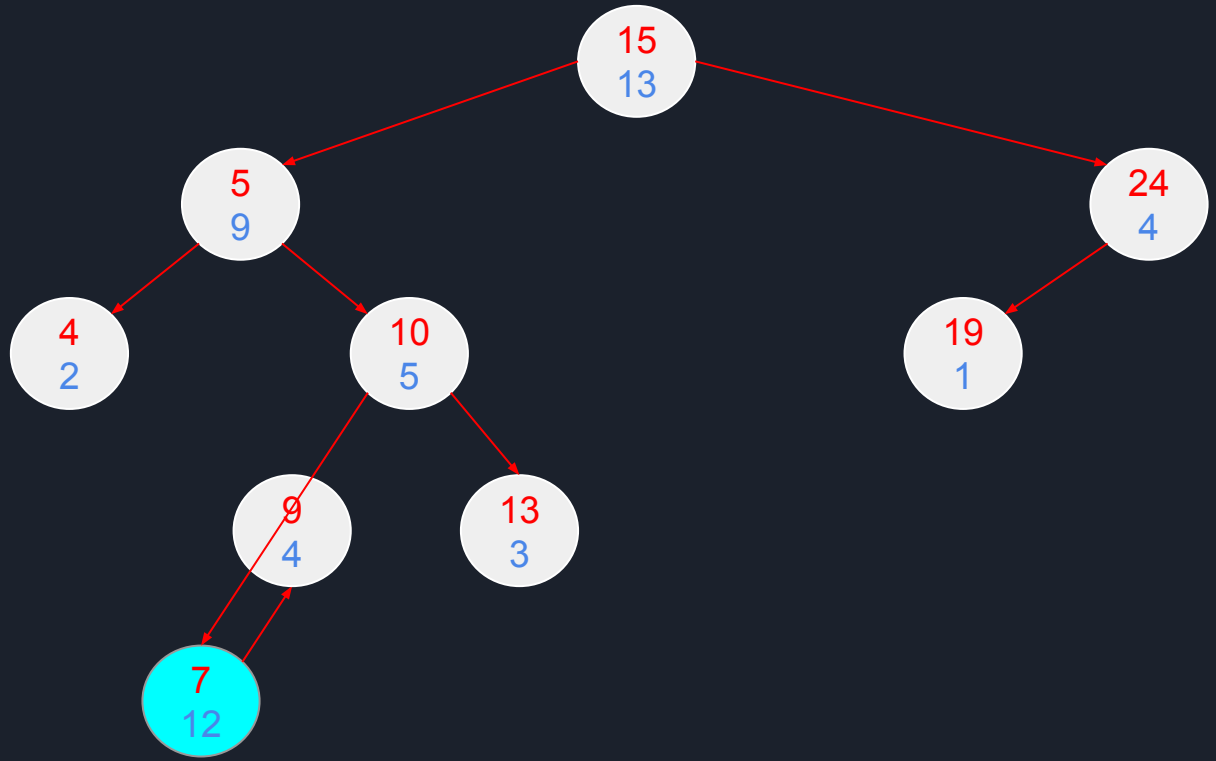
Step 2: If Heap property is not preserved, move the node using rotation



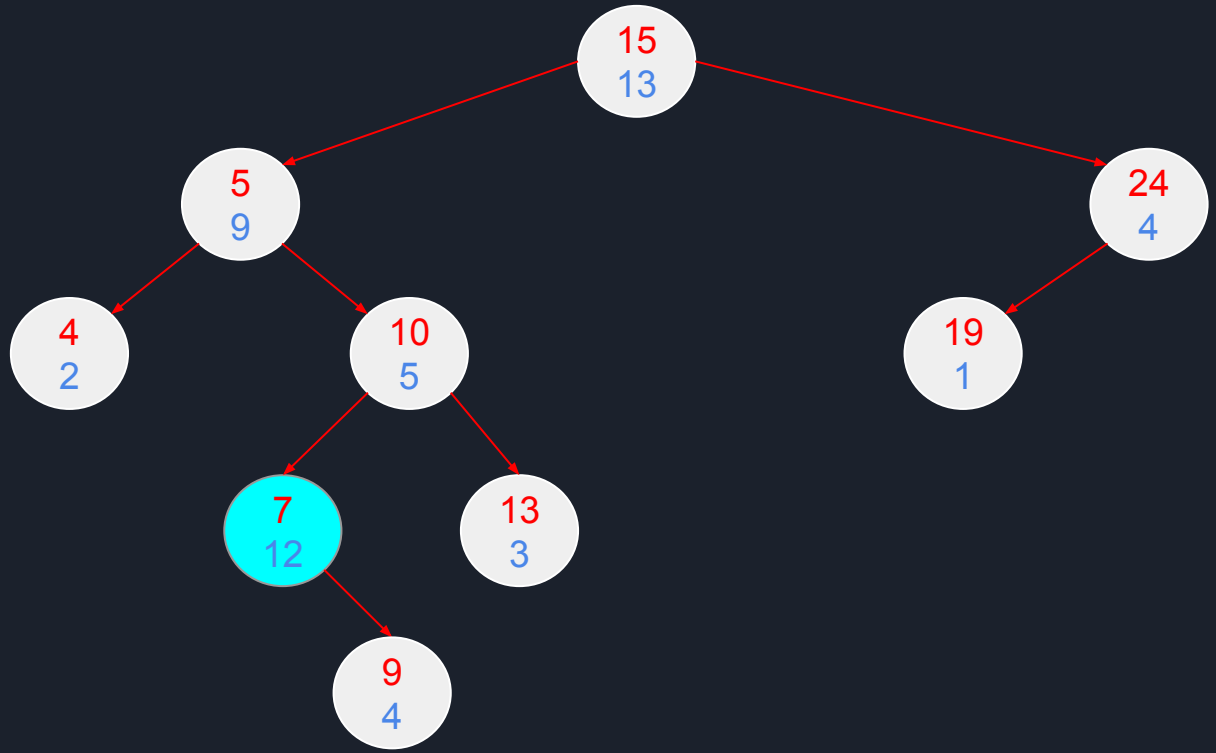
Step 2: If Heap property is not preserved, move the node using rotation and comparing with priority value



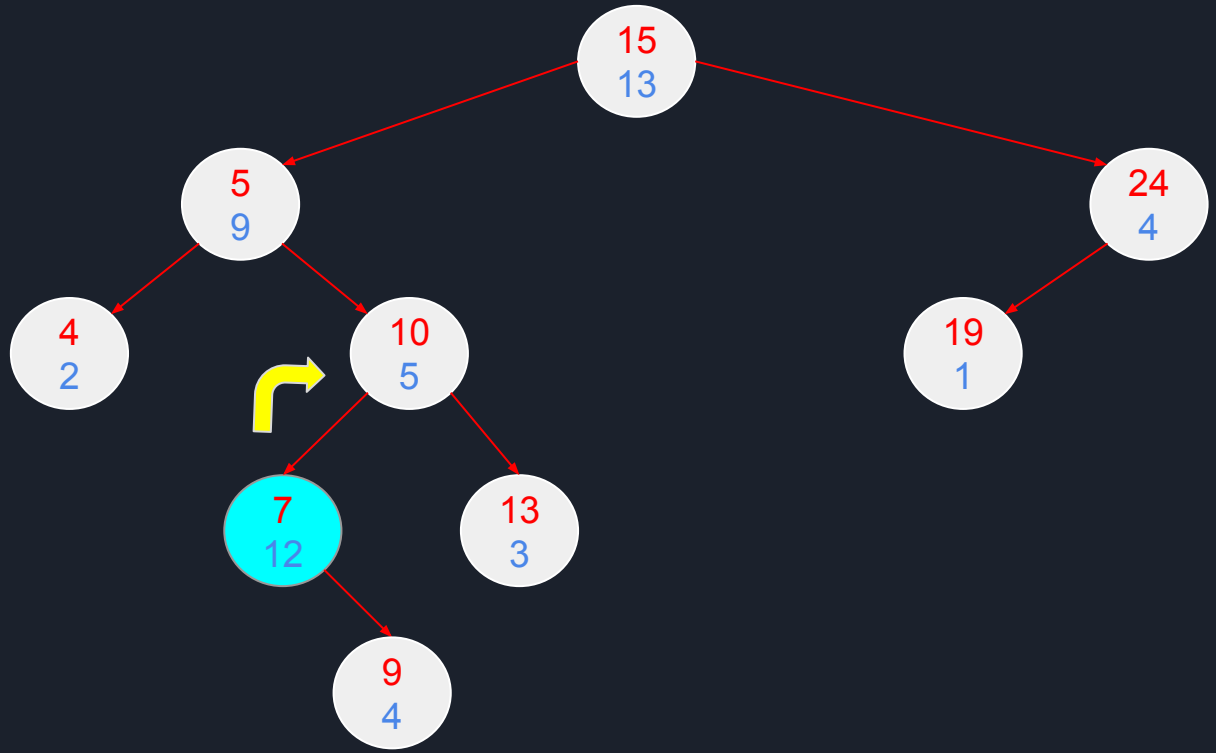
Step 2: If Heap property is not preserved, move the node using rotation and comparing with priority value



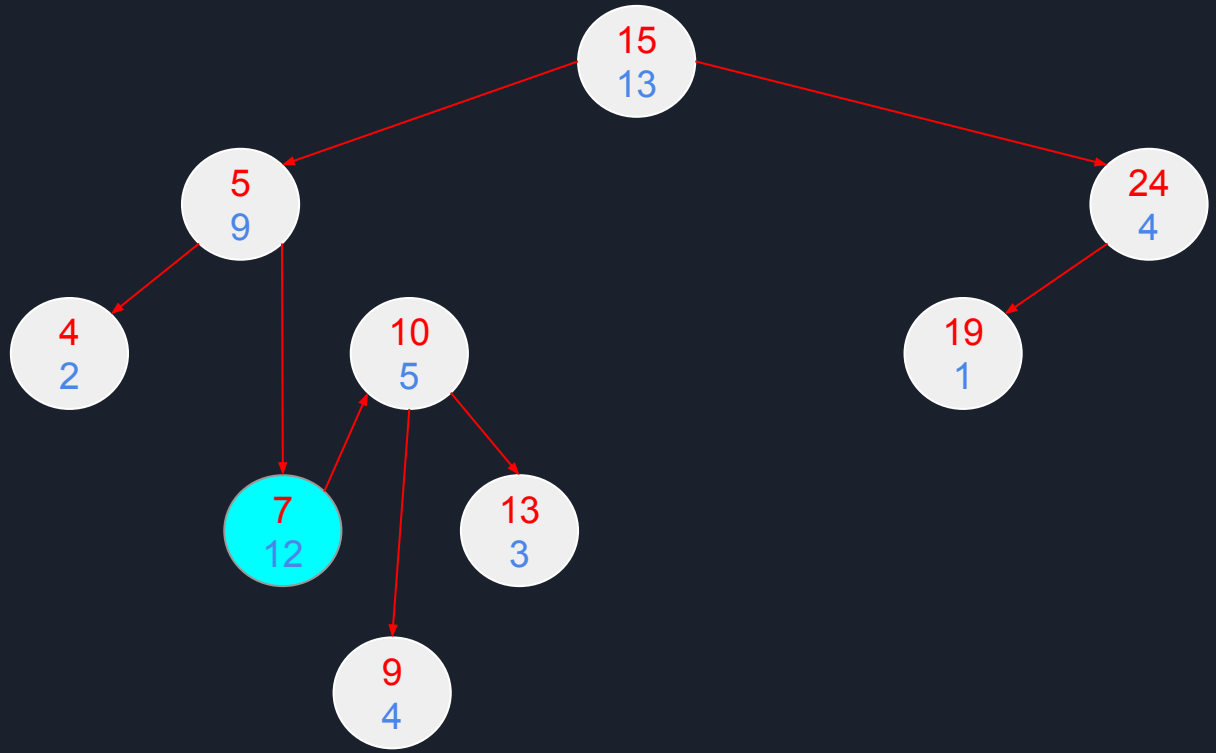
Step 2: If Heap property is not preserved, move the node using rotation and comparing with priority value



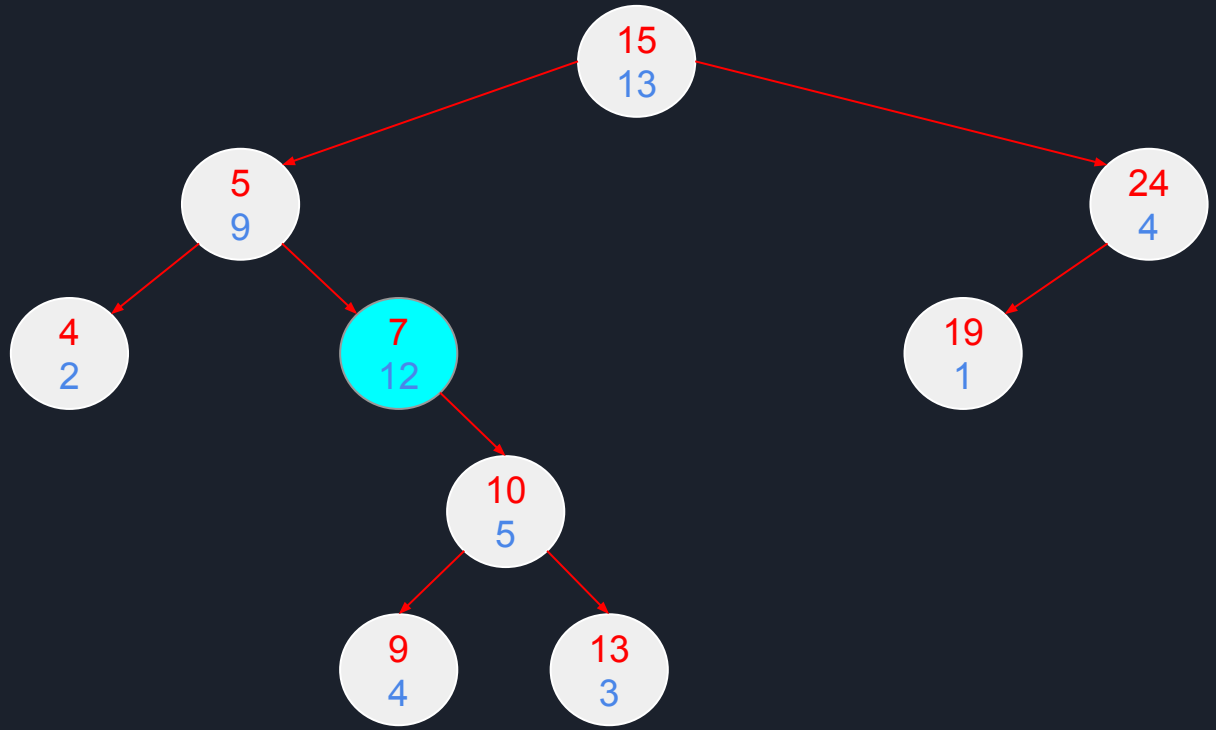
Step 2: If Heap property is not preserved, move the node using rotation and comparing with priority value



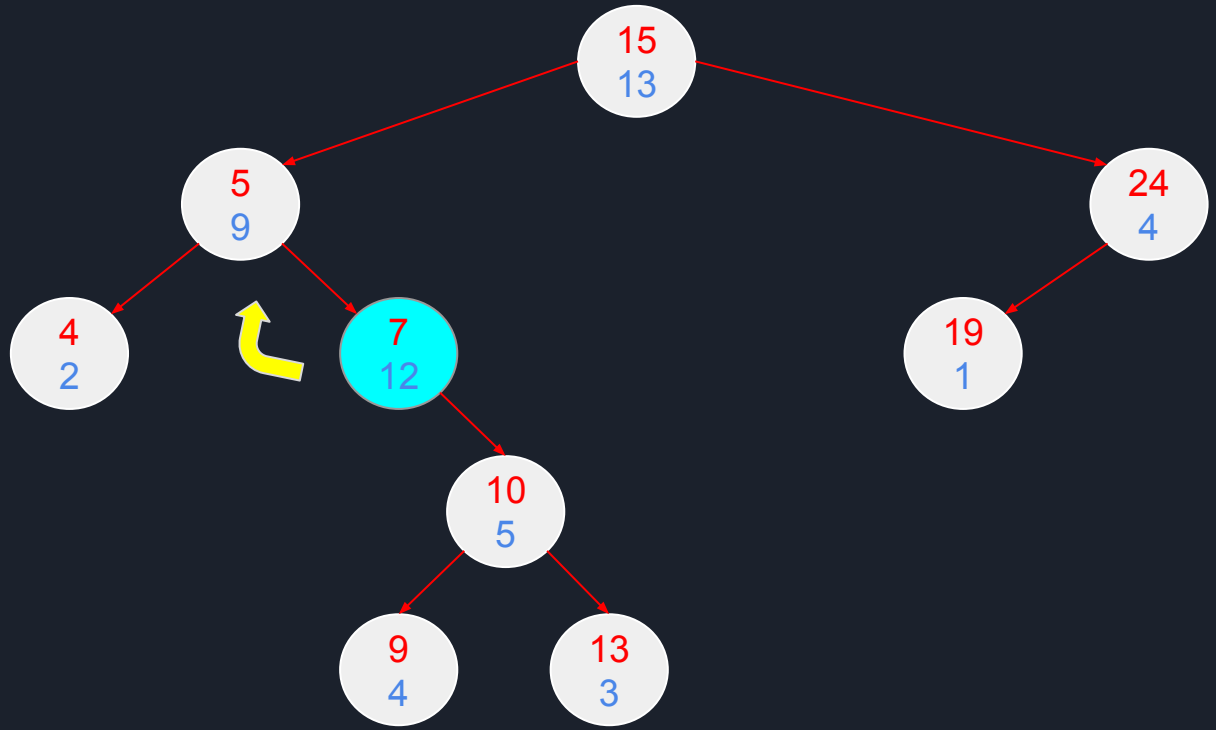
Step 2: If Heap property is not preserved, move the node using rotation and comparing with priority value



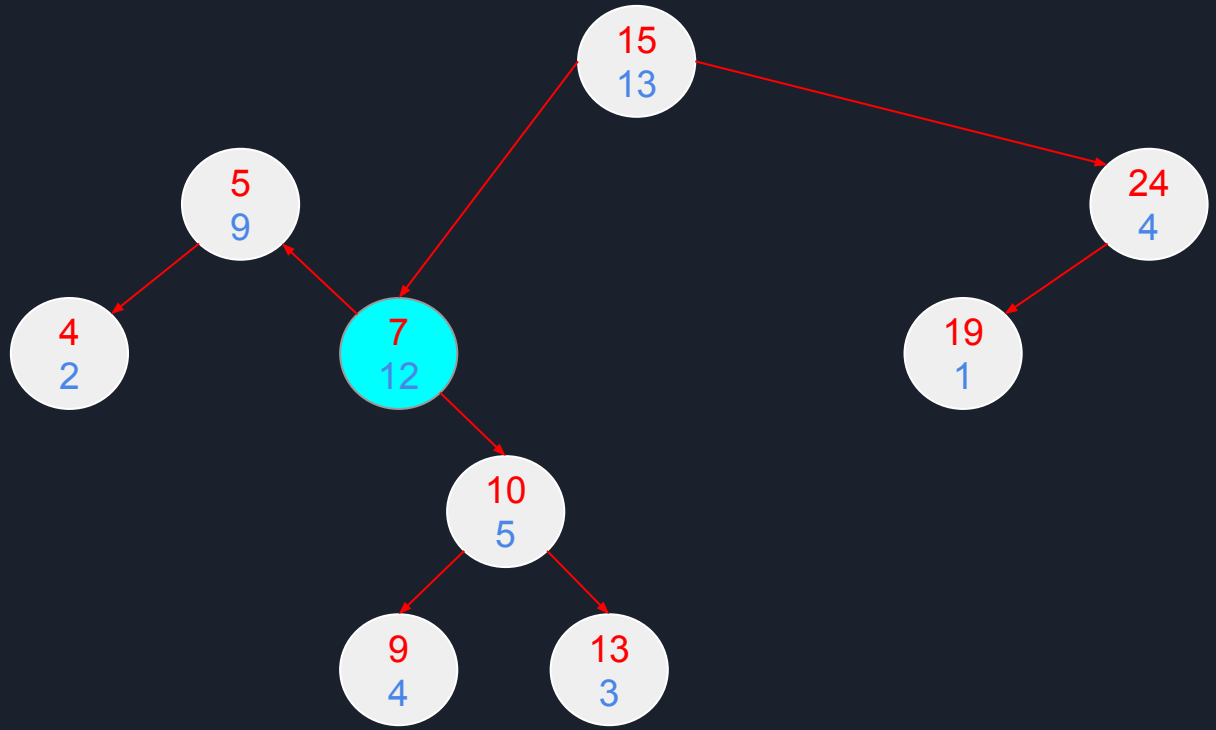
Step 2: If Heap property is not preserved, move the node using rotation and comparing with priority value



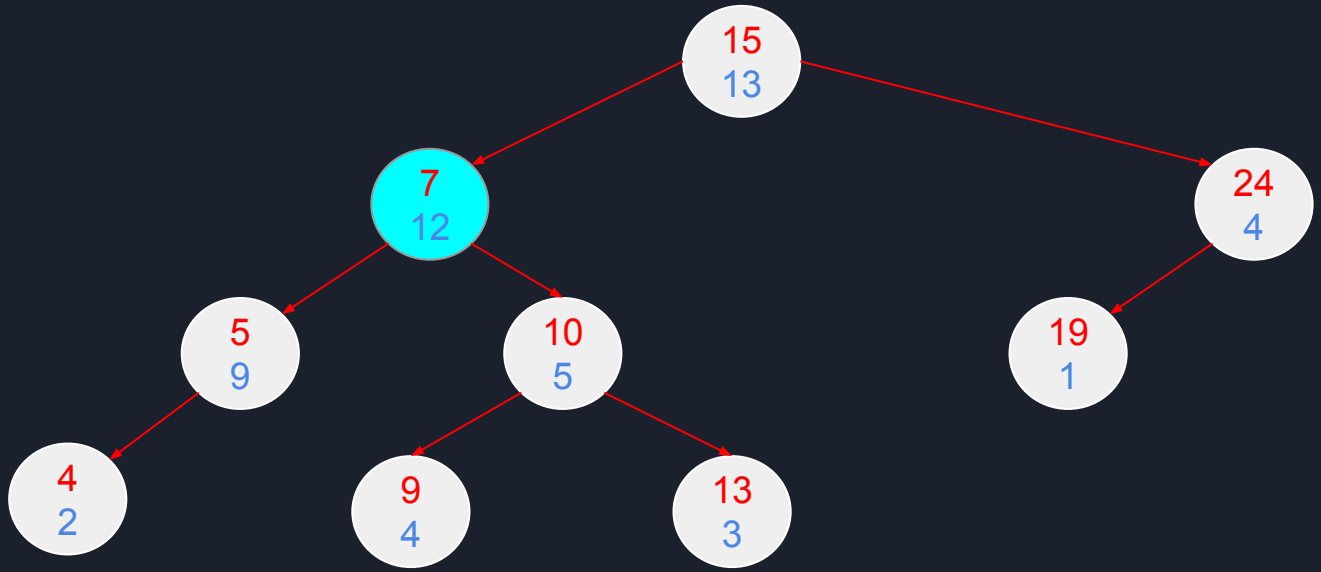
Step 2: If Heap property is not preserved, move the node using rotation and comparing with priority value



Step 2: If Heap property is not preserved, move the node using rotation and comparing with priority value



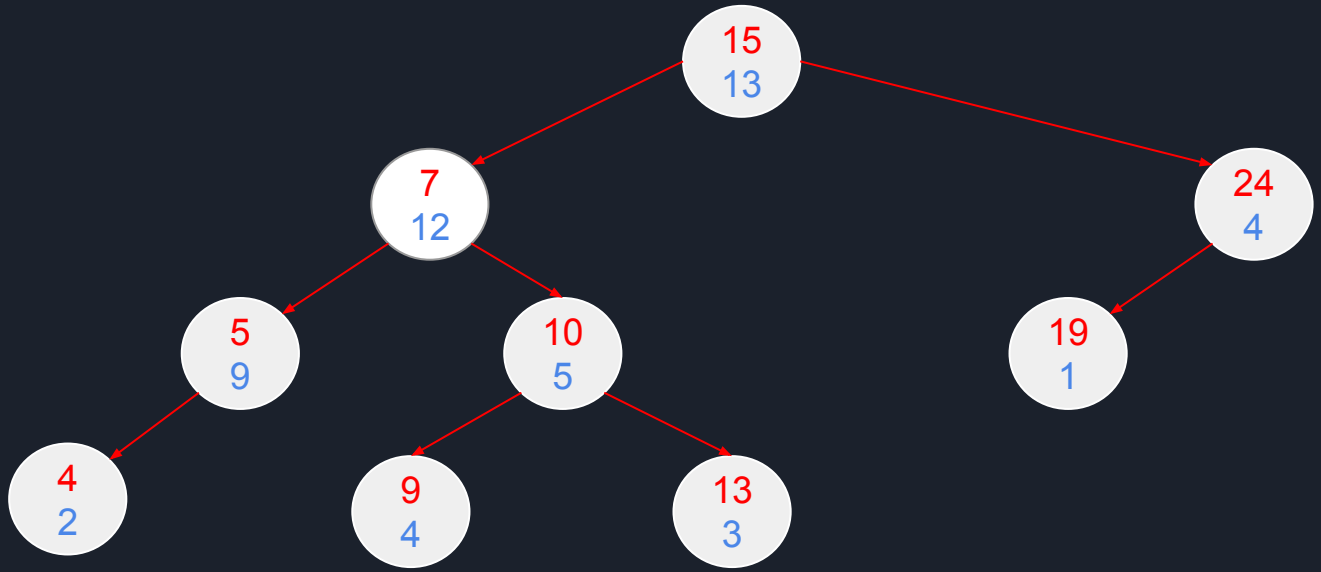
Step 2: If Heap property is not preserved, move the node using rotation and comparing with priority value



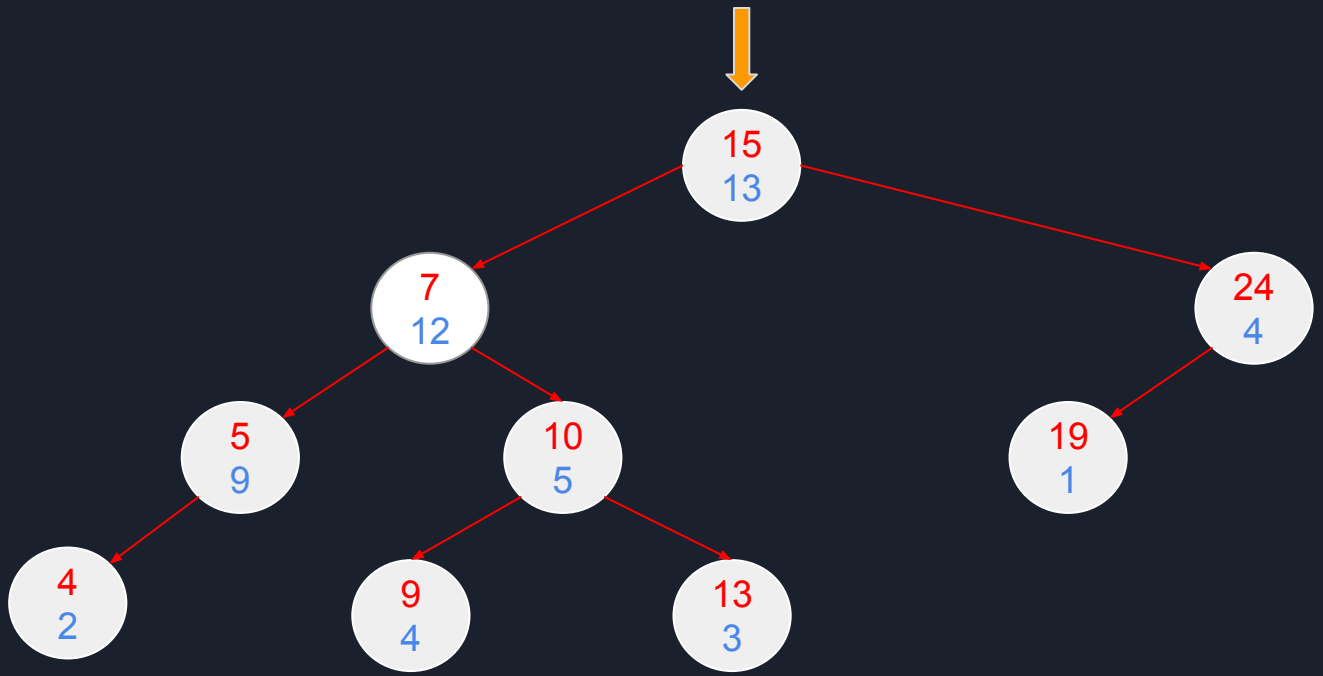
Remove

- Find node by key ⇒ O (log (n))
- Change priority to -∞ (Max Heap) or +∞ (Min Heap)
- Rotate until node is leaf ⇒ O (1)
- Complexity O (log (n))

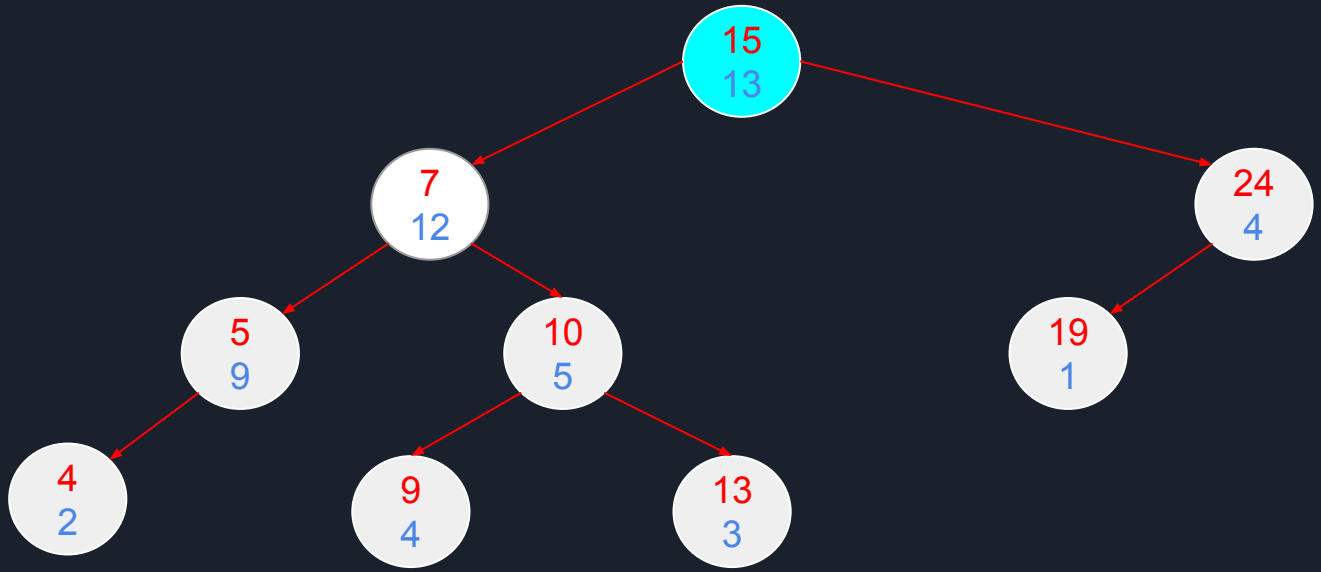
Remove (Eg. Remove(15))



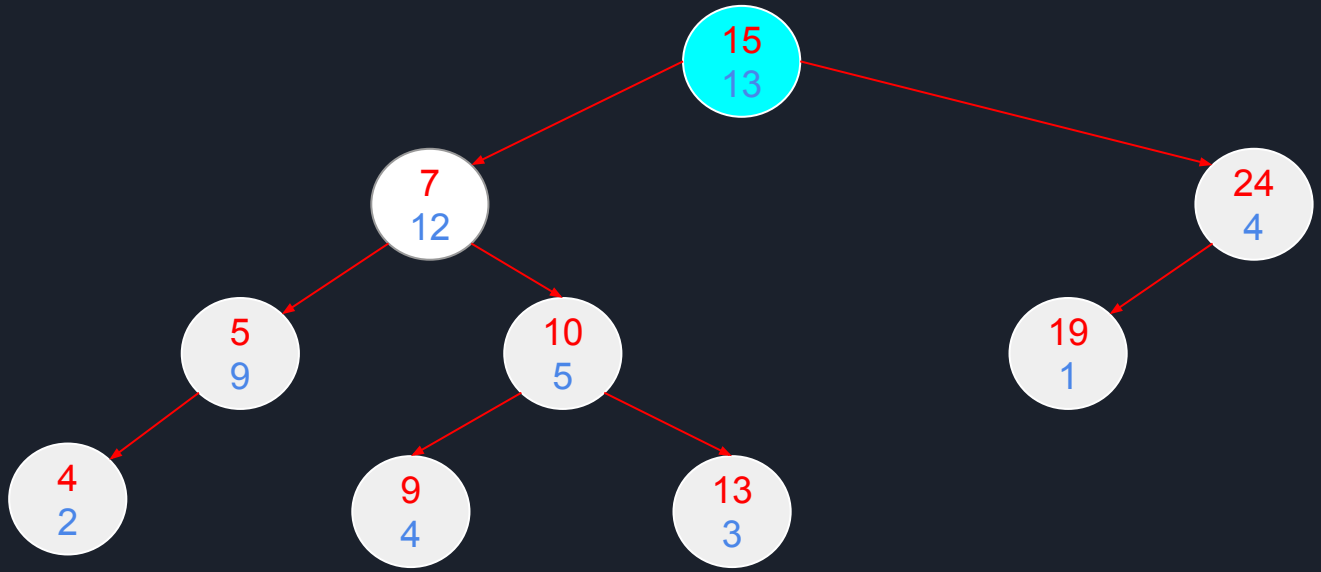
Step 1: Find node



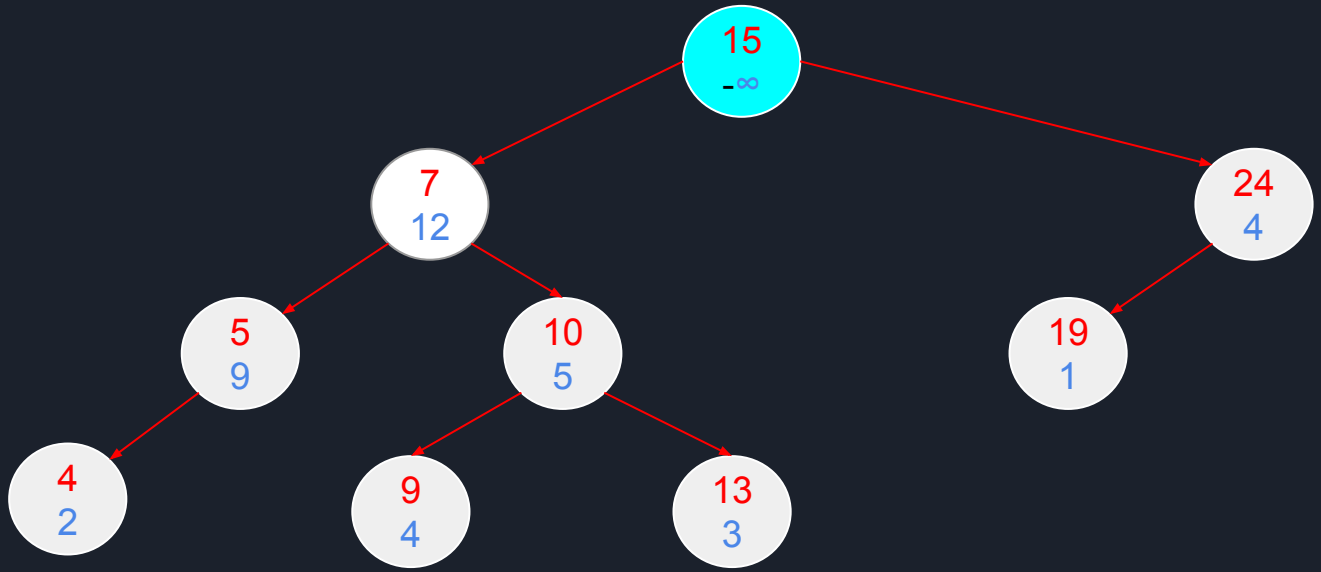
Step 1: Find node



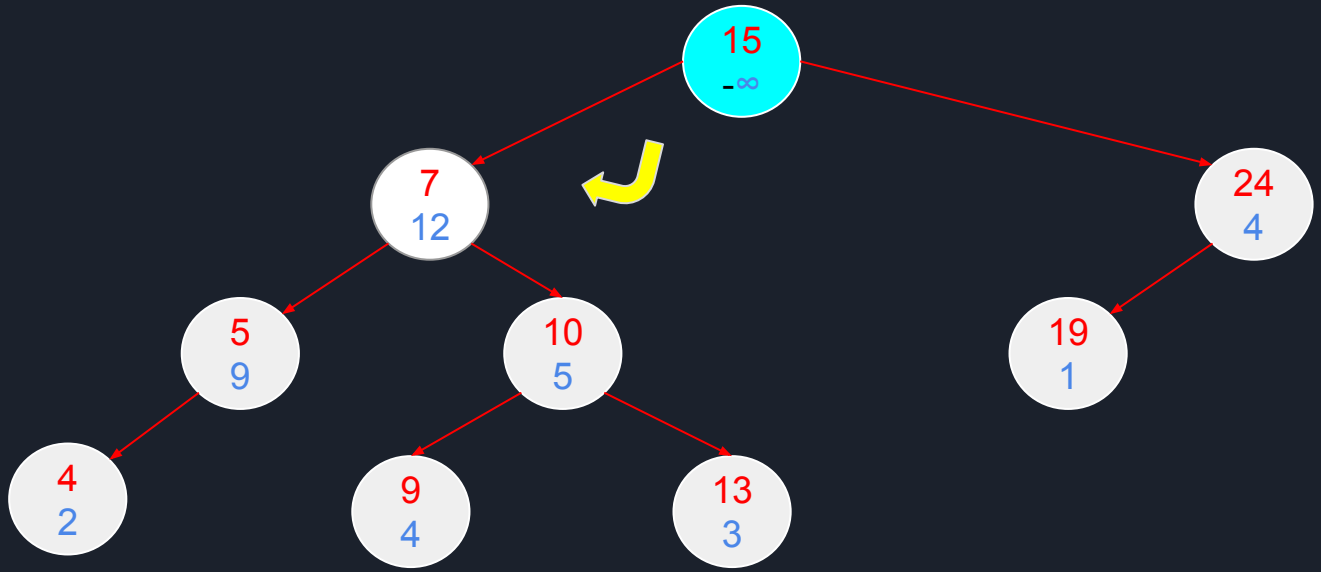
Step 2: Change priority to -INF



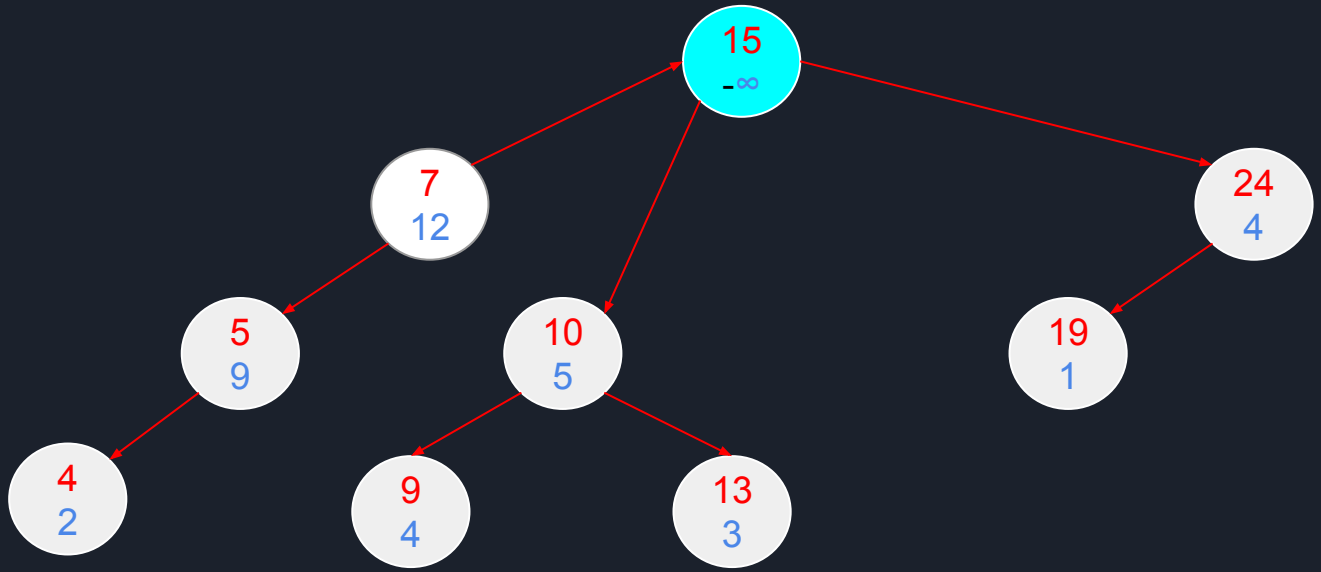
Step 2: Change priority to -INF



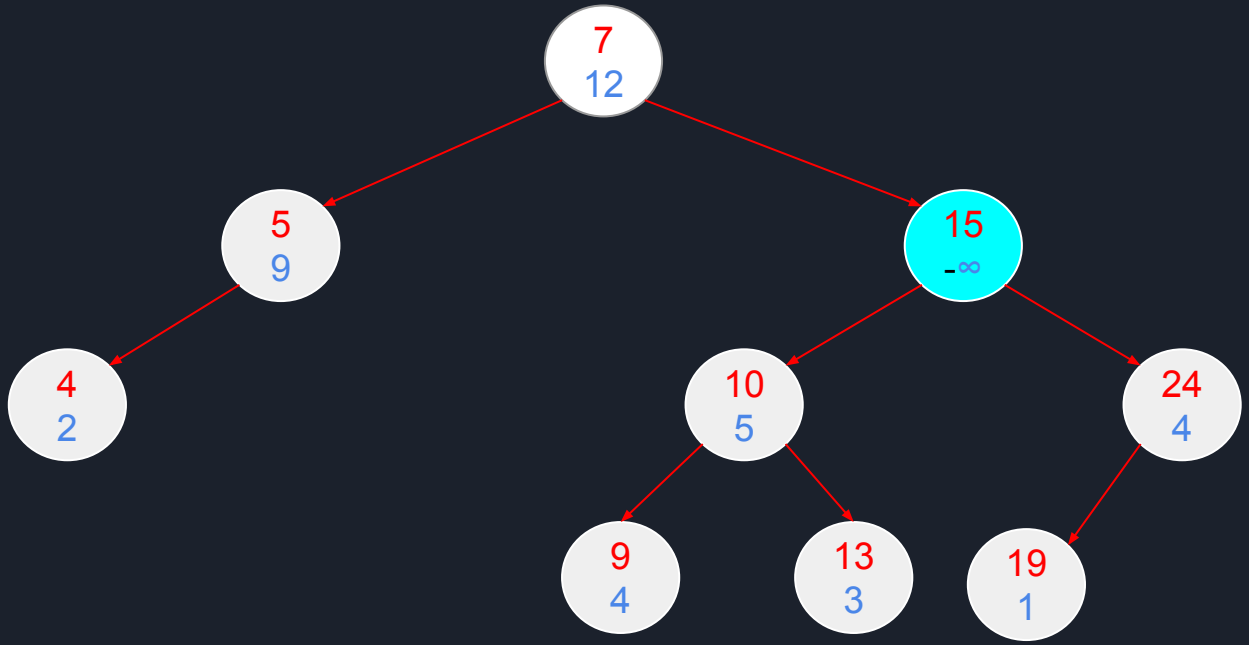
Step 3: Rotate until node is leaf with the higher priority value of its child nodes



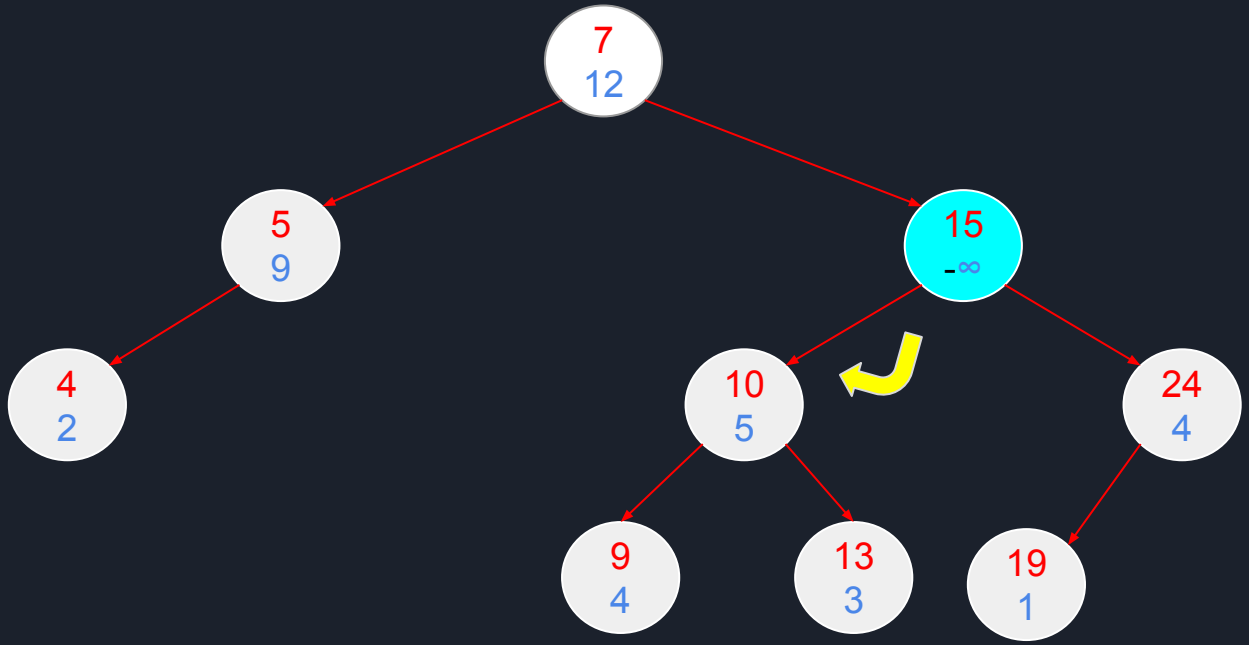
Step 3: Rotate until node is leaf with the higher priority value of its child nodes



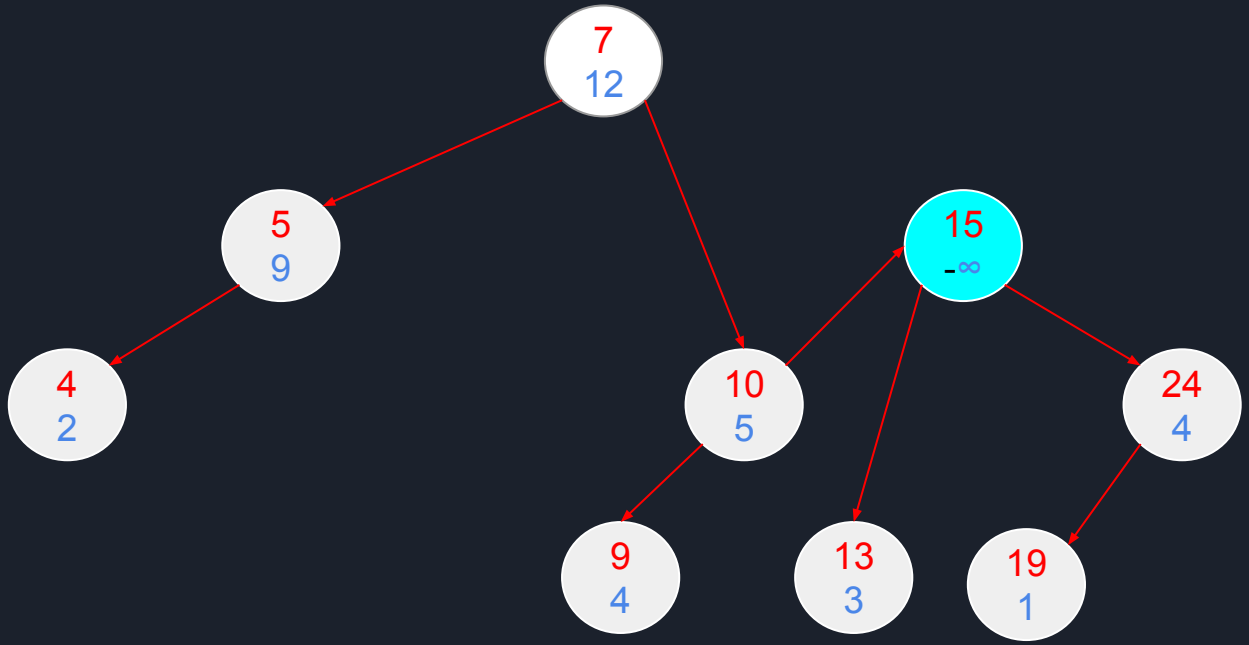
Step 3: Rotate until node is leaf with the higher priority value of its child nodes



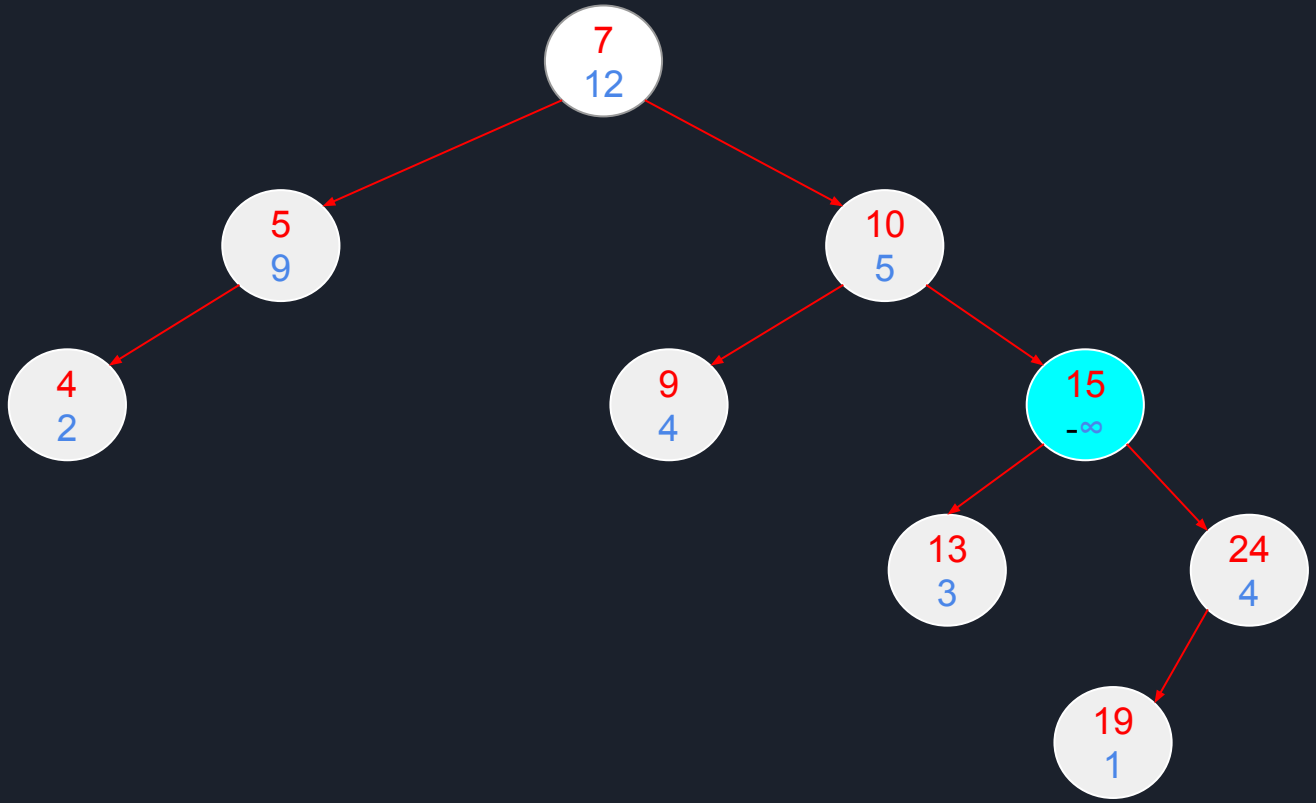
Step 3: Rotate until node is leaf with the higher priority value of its child nodes



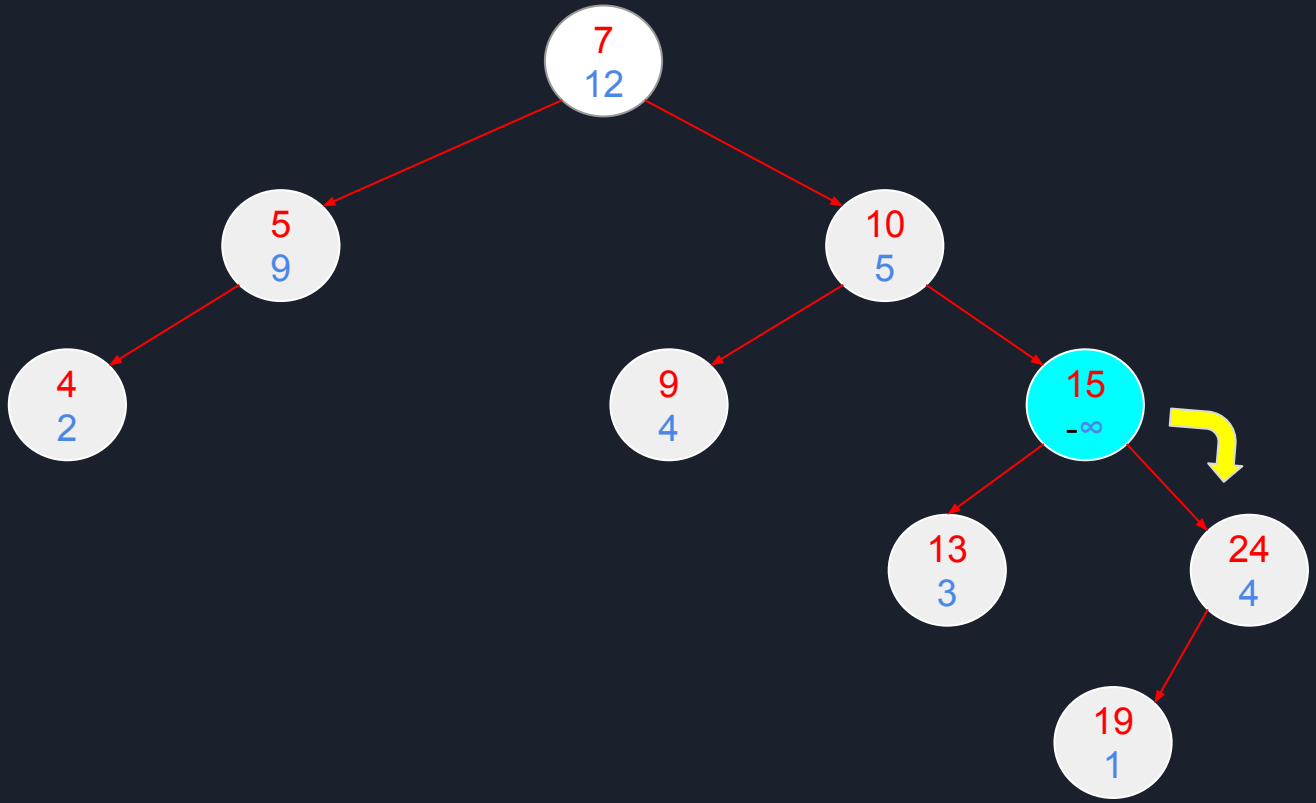
Step 3: Rotate until node is leaf with the higher priority value of its child nodes



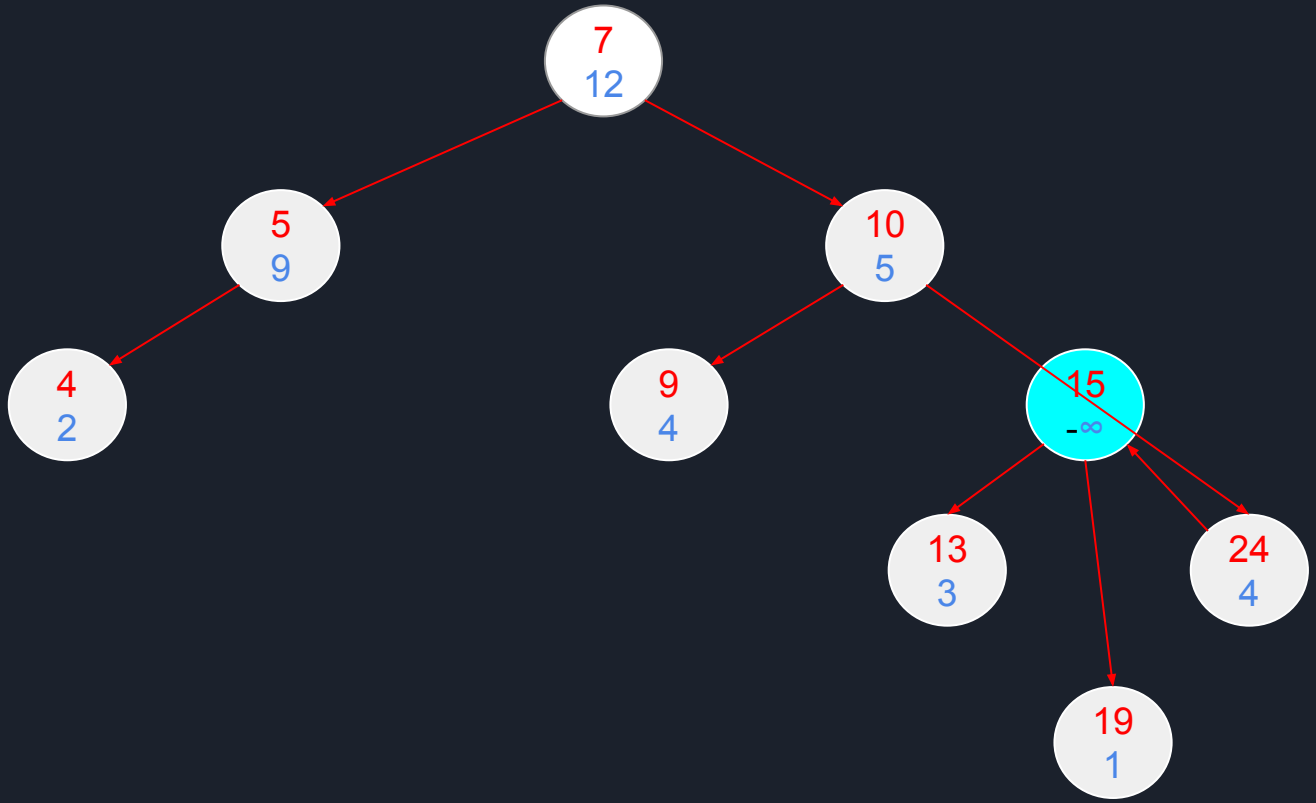
Step 3: Rotate until node is leaf with the higher priority value of its child nodes



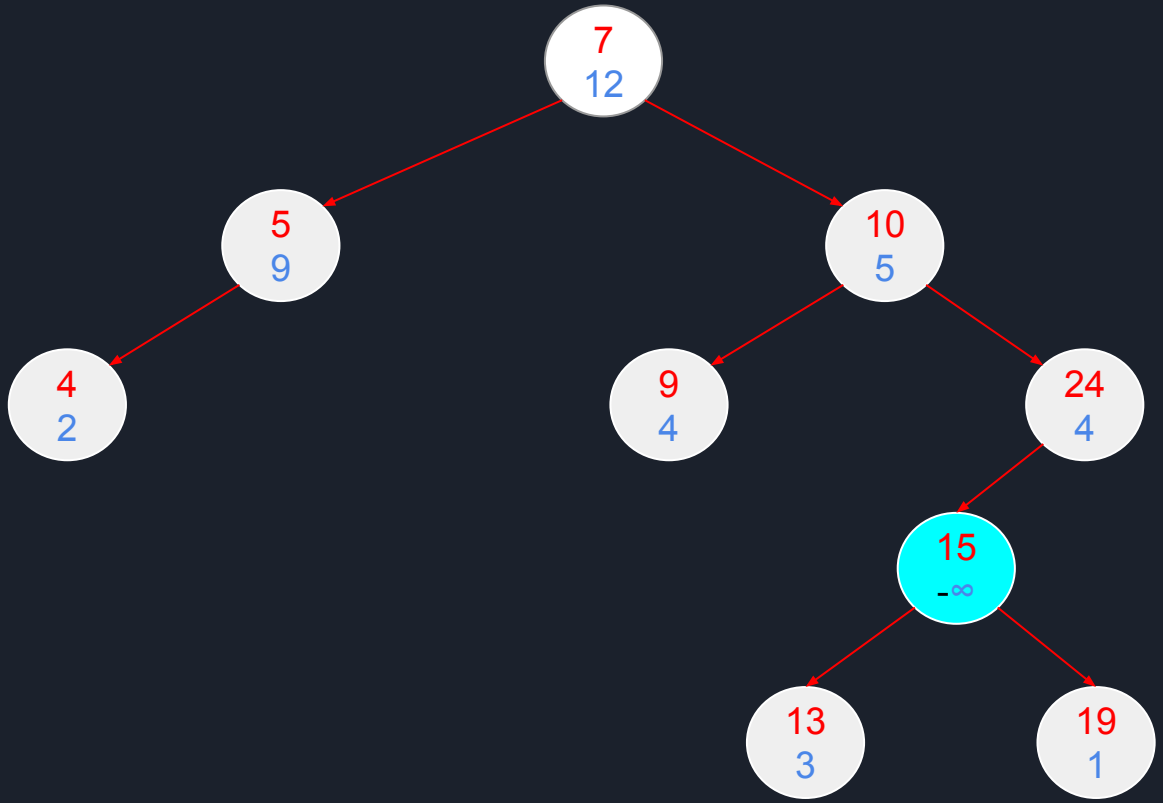
Step 3: Rotate until node is leaf with the higher priority value of its child nodes



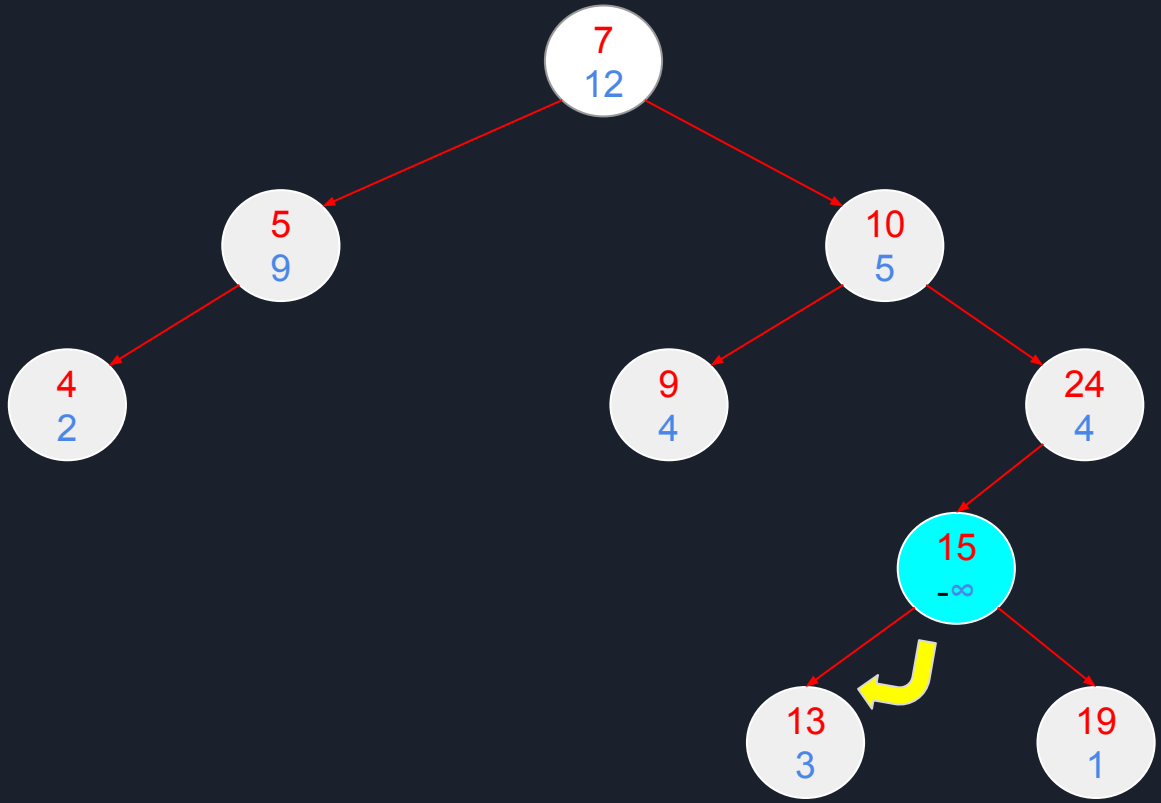
Step 3: Rotate until node is leaf with the higher priority value of its child nodes



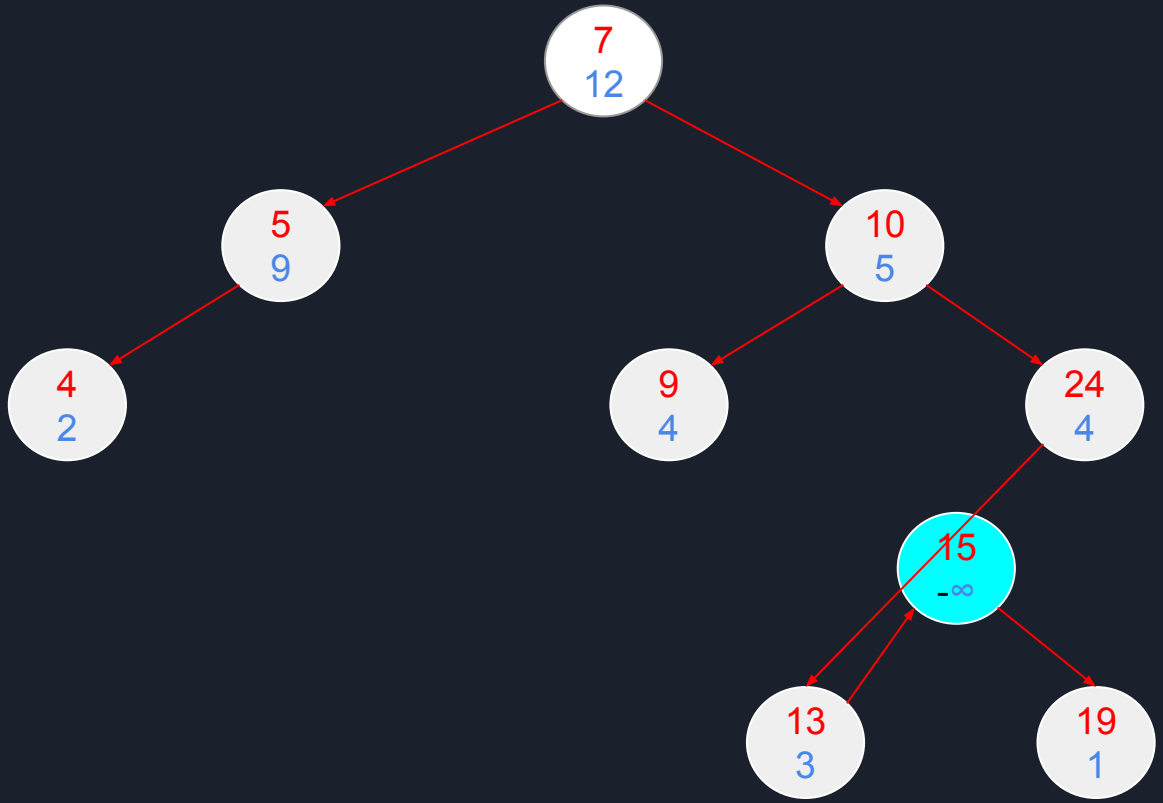
Step 3: Rotate until node is leaf with the higher priority value of its child nodes



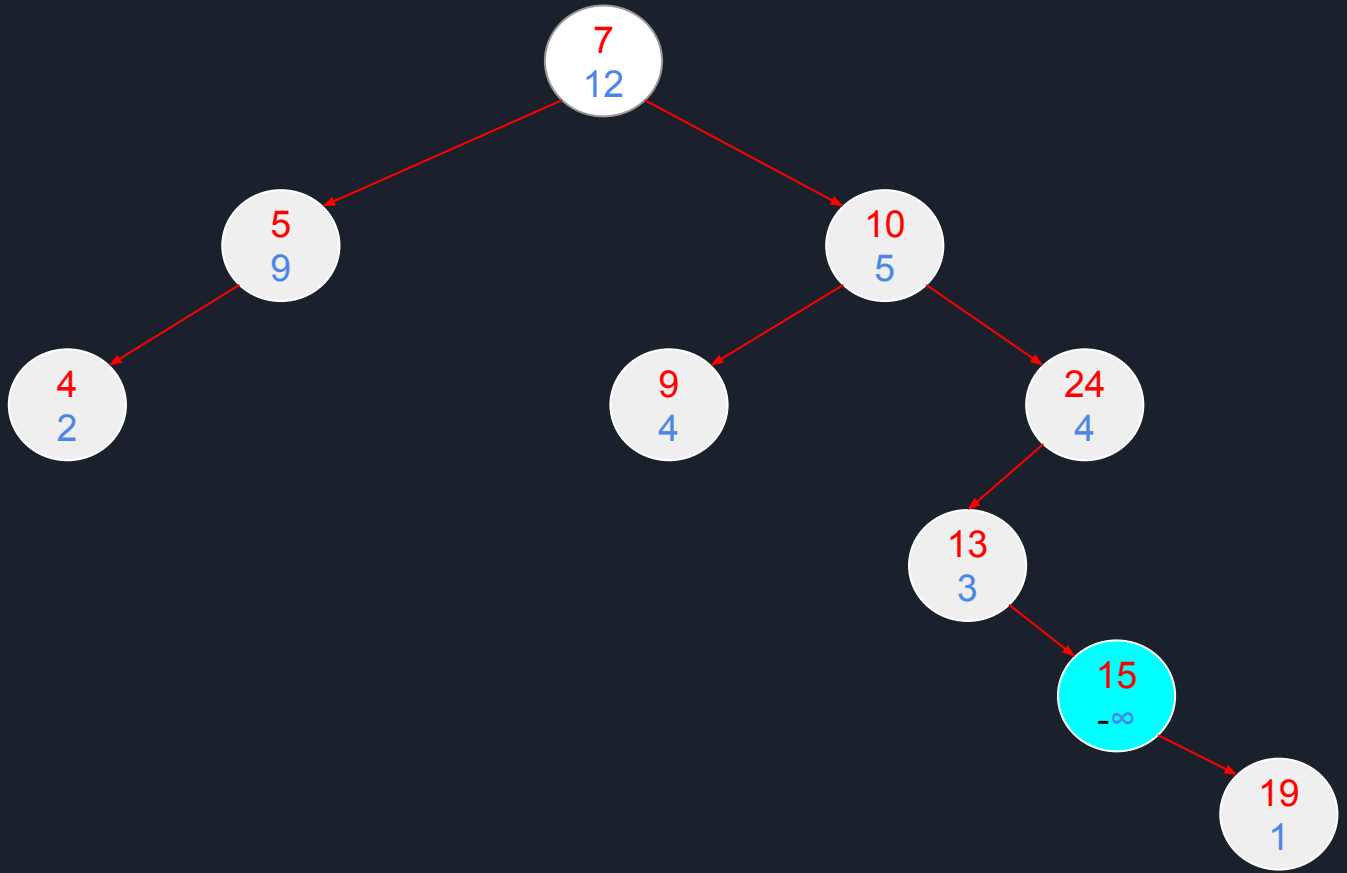
Step 3: Rotate until node is leaf with the higher priority value of its child nodes



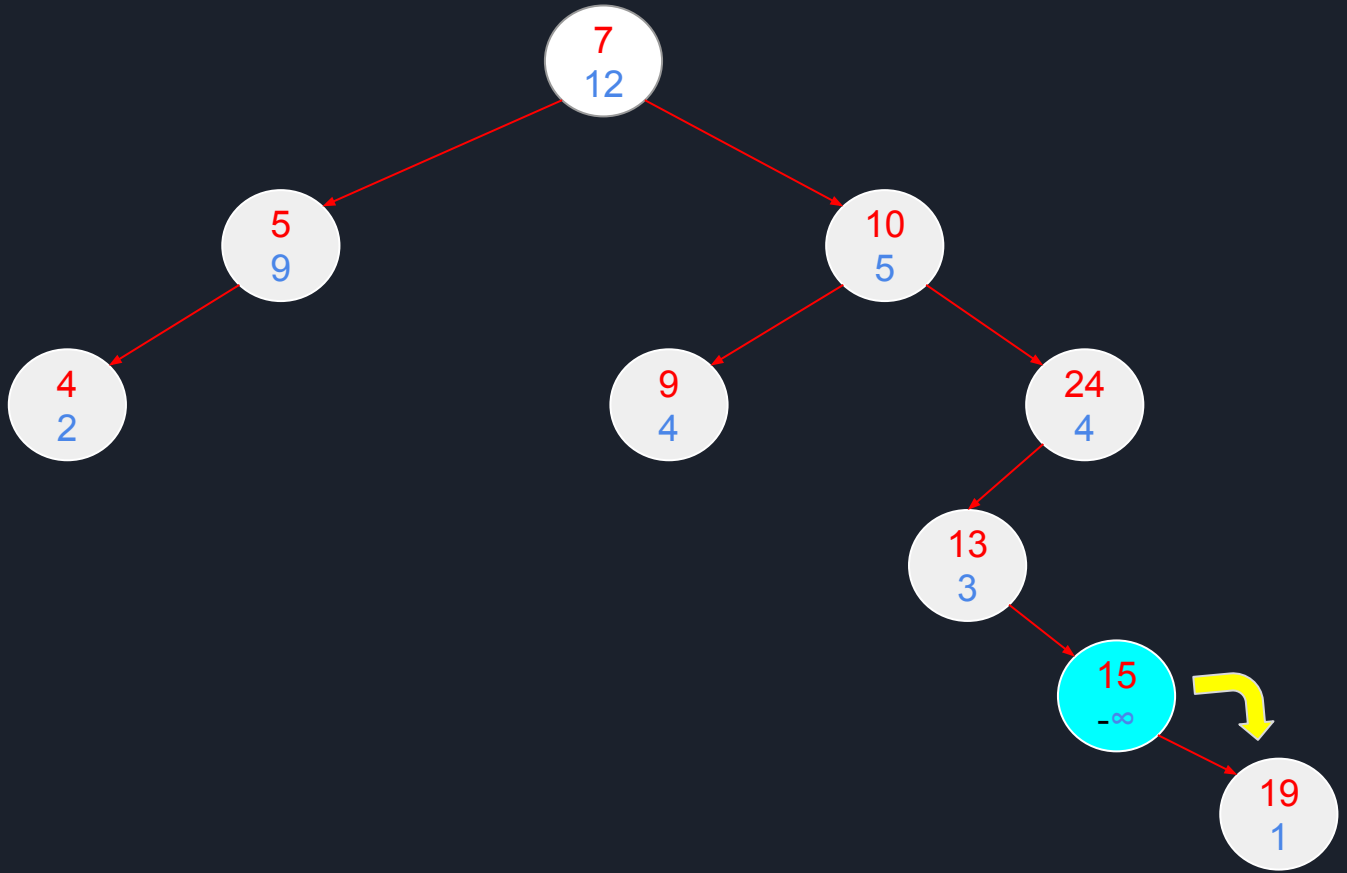
Step 3: Rotate until node is leaf with the higher priority value of its child nodes



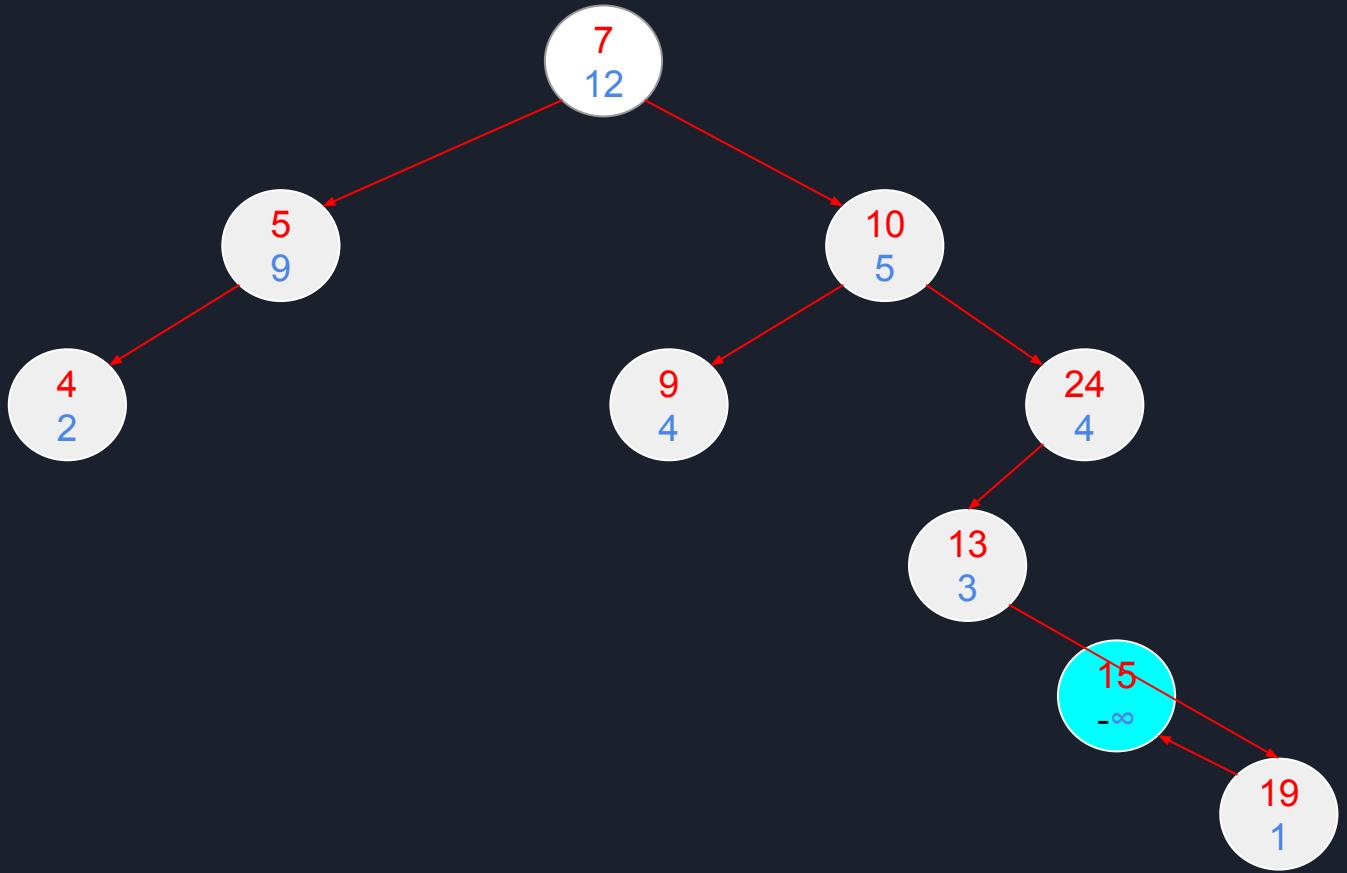
Step 3: Rotate until node is leaf with the higher priority value of its child nodes



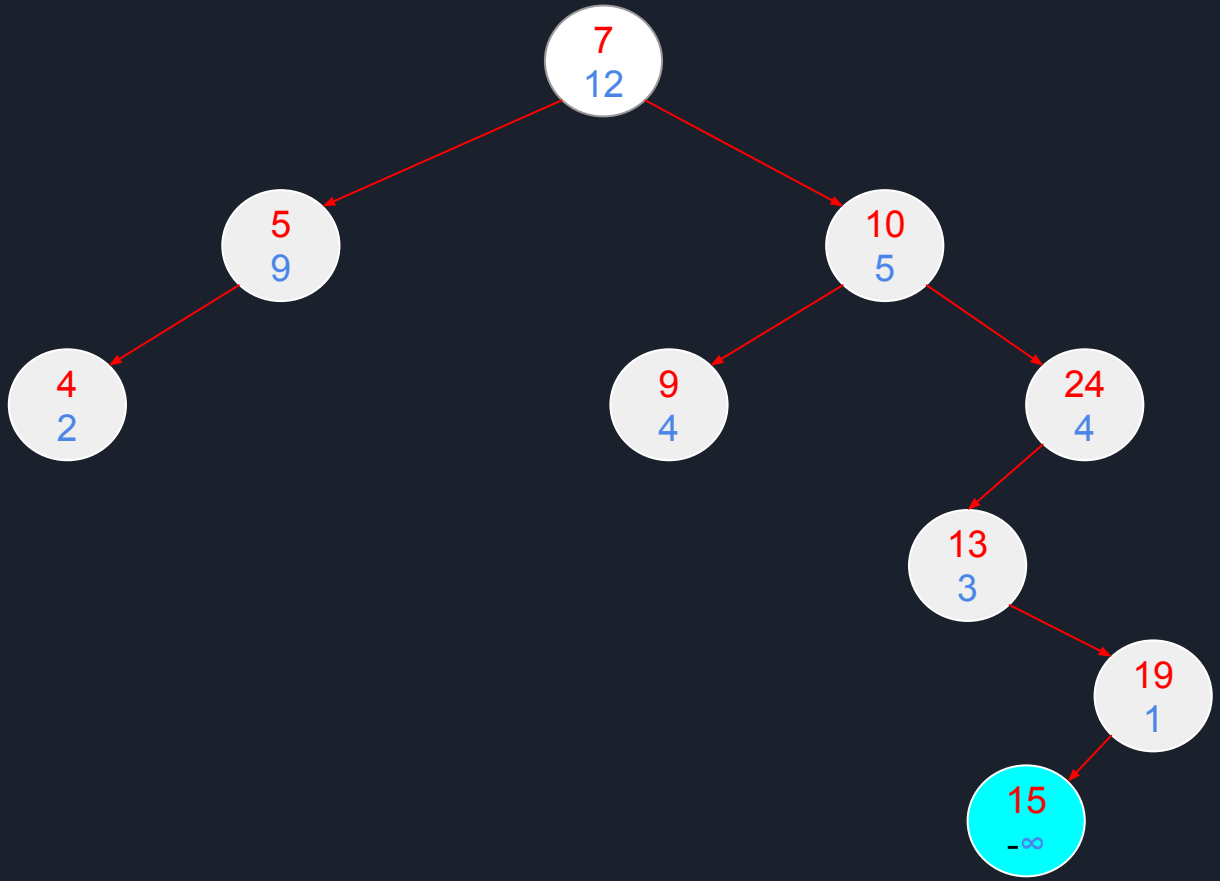
Step 3: Rotate until node is leaf with the higher priority value of its child nodes



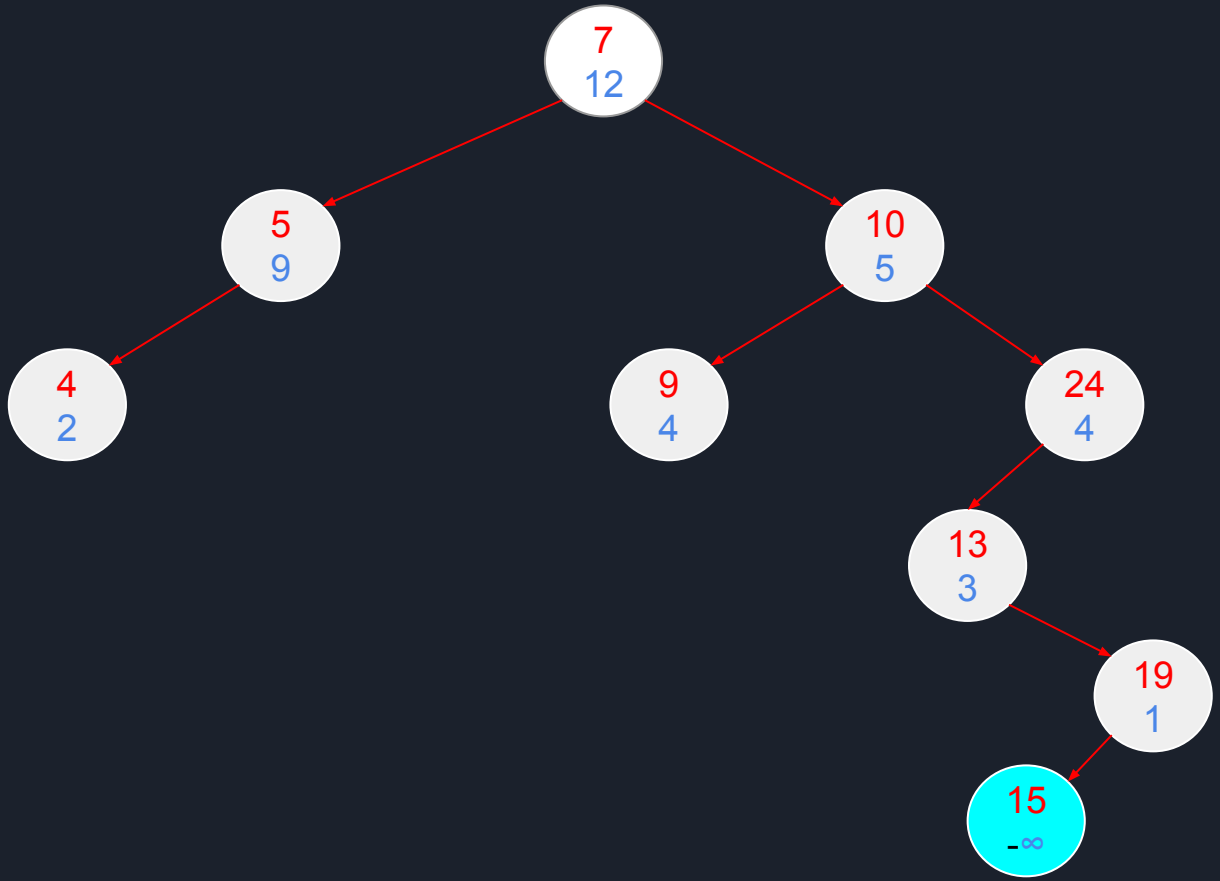
Step 3: Rotate until node is leaf with the higher priority value of its child nodes



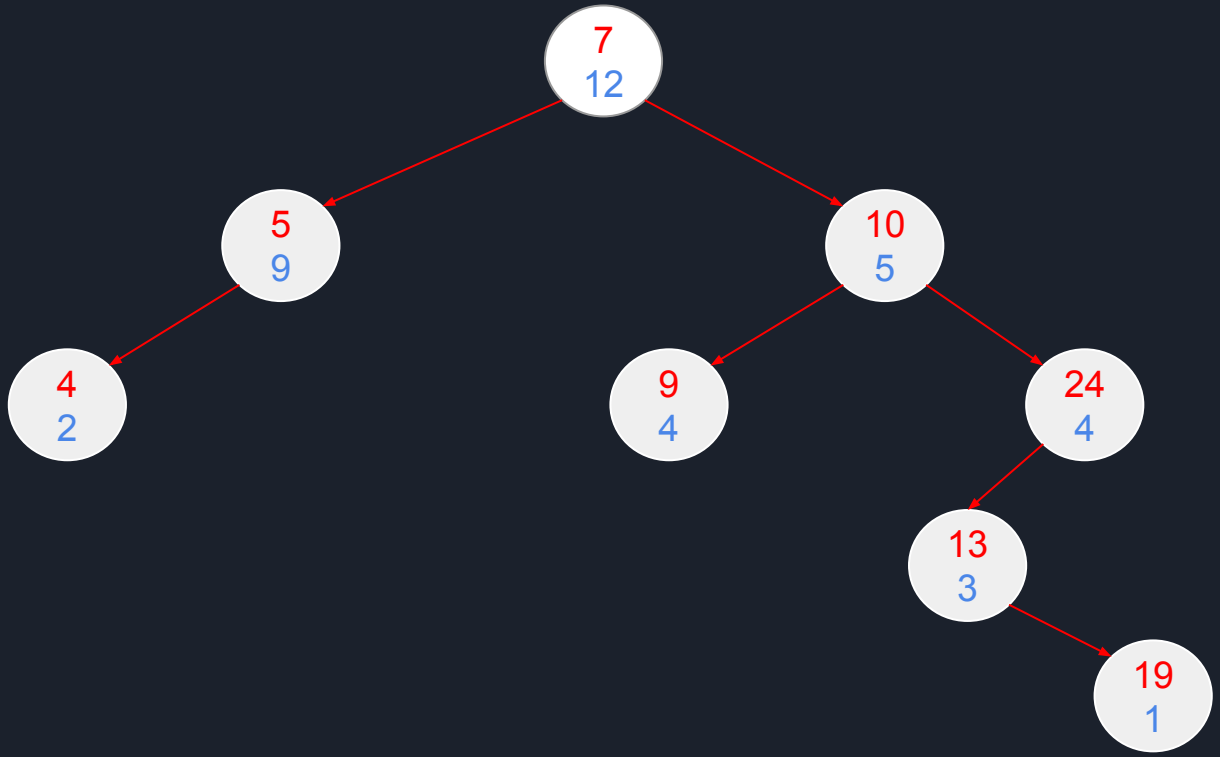
Step 3: Rotate until node is leaf with the higher priority value of its child nodes



Step 4: Delete node



Step 4: Delete node



Advantages

- Difference between Treap and BST is that this structure is dynamic and according to the inserts the tree will balance.
- It doesn’t need complex algorithms.
- Unlike balanced trees, with treaps you don’t need to handle exceptions in the code.
- Treap will be the same regardless of insertion order.
- Logarithmic height.
- No matter the order in which we add, delete, etc. Because of the randomized priorities, there is a high probability that the treap will be balanced (A random binary search tree has logarithmic height).

Disadvantages

- If the user inserts a node with both values (key and priority), Treap could be unbalanced.

- Can’t keep a strict balance condition (like AVL tree).

Application

- If you want to use a binary tree would be more efficient use a treap.

- It’s highly used in programming contests because it’s easy and efficient to code.

Code

⇐ Author

Github:
https://gist.github.com/igorcarpanese

Struct Treap Node

```
struct TreapNode {
    int key, priority;
    TreapNode *left, *right;

    TreapNode() {}

    TreapNode(int key) {
        this->key = key;
        this->priority = rand();
        this->left = NULL;
        this->right = NULL;
    }
};
```

Search method

```
TreapNode* search(TreapNode* &root, int key) {
    if (!root or root->key == key) return root;
    if (root->key < key)  return search(root->right, key);
    if (root->key > key)  return search(root->left, key);
}
```

Rotation

```
void right_rotation(TreapNode* &x) {
    TreapNode *y = x->left;

    TreapNode *r = y->left; // will not change
    TreapNode *g = y->right;
    TreapNode *b = x->right; // will not change

    x->left = g;
    y->right = x;

    x = y;
}

void left_rotation(TreapNode* &y) {
    TreapNode *x = y->right;

    TreapNode *r = y->left; // will not change
    TreapNode *g = x->left;
    TreapNode *b = x->right; // will not change

    x->left = y;
    y->right = g;

    y = x;
}
```

```
void right_rotation(TreapNode* &x) {
    TreapNode *y = x->left;

    TreapNode *r = y->left; // will not change
    TreapNode *g = y->right;
    TreapNode *b = x->right; // will not change

    x->left = g;
    y->right = x;

    x = y;
}
```

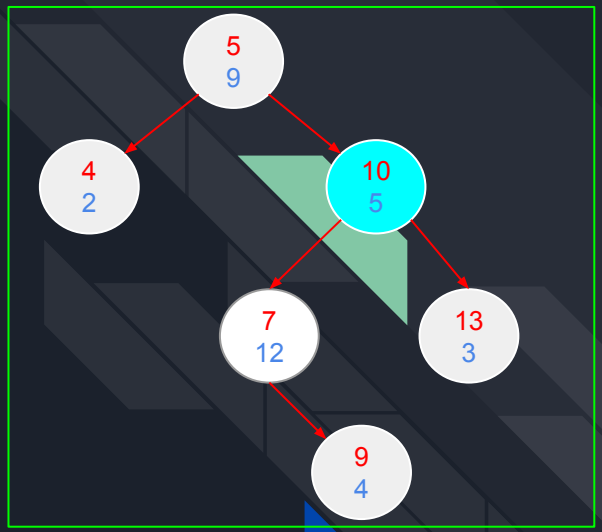
x = 10 5

y = 7 12

r = null

g = 9 4

b = 13 3



```
void right_rotation(TreapNode* &x) {
    TreapNode *y = x->left;

    TreapNode *r = y->left; // will not change
    TreapNode *g = y->right;
    TreapNode *b = x->right; // will not change

    x->left = g;
    y->right = x;

    x = y;
}
```

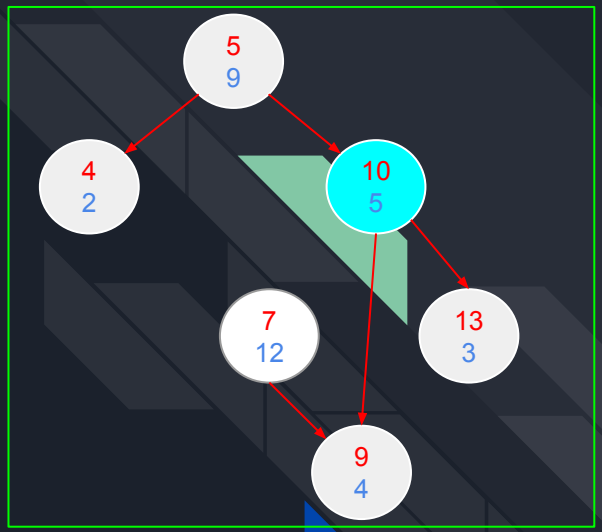
x = 10 5

y = 7 12

r = null

g = 9 4

b = 13 3



```
void right_rotation(TreapNode* &x) {
    TreapNode *y = x->left;

    TreapNode *r = y->left; // will not change
    TreapNode *g = y->right;
    TreapNode *b = x->right; // will not change

    x->left = g;
    y->right = x;

    x = y;
}
```

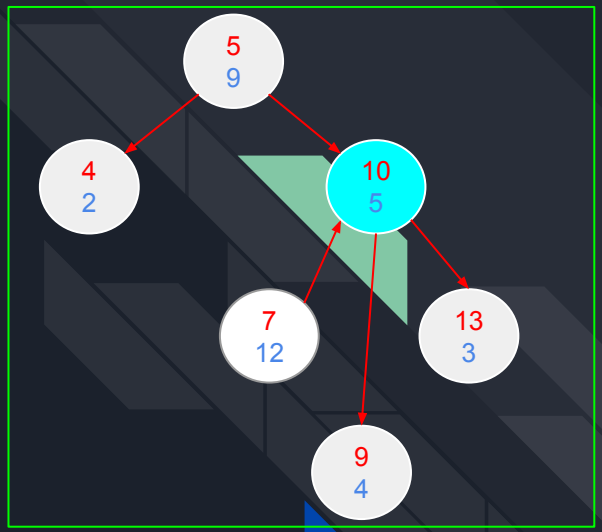
x = 10 5

y = 7 12

r = null

g = 9 4

b = 13 3



```
void right_rotation(TreapNode* &x) {
    TreapNode *y = x->left;

    TreapNode *r = y->left; // will not change
    TreapNode *g = y->right;
    TreapNode *b = x->right; // will not change

    x->left = g;
    y->right = x;

    x = y; ⟵
}
```

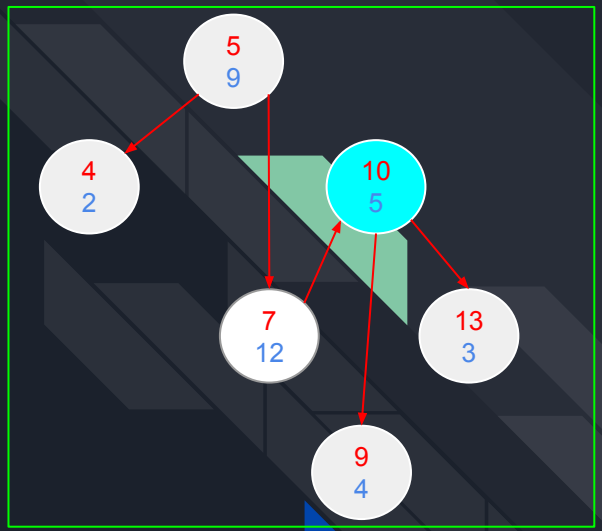
x = 10 5

y = 7 12

r = null

g = 9 4

b = 13 3



```
void right_rotation(TreapNode* &x) {
    TreapNode *y = x->left;

    TreapNode *r = y->left; // will not change
    TreapNode *g = y->right;
    TreapNode *b = x->right; // will not change

    x->left = g;
    y->right = x;

    x = y;
}
```

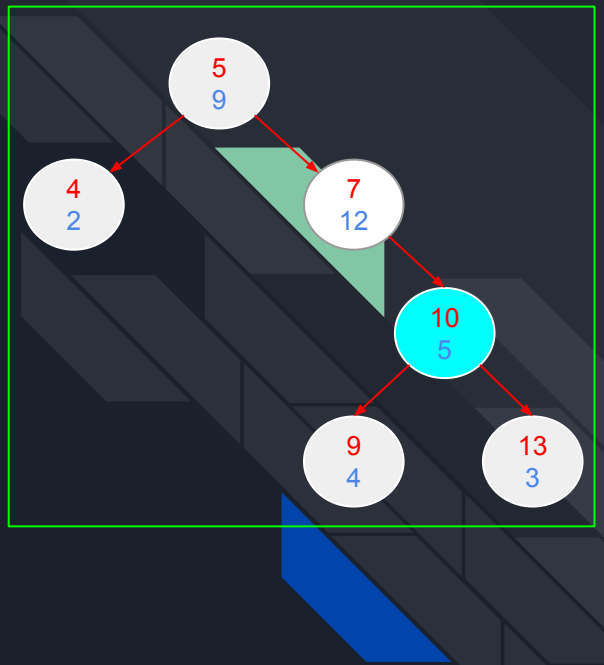
x = 10 5

y = 7 12

r = null

g = 9 4

b = 13 3



Insert

```
void insert(TreapNode* &root, int key) {
    if (!root) return void(root = new TreapNode(key));

    insert(key <= root->key ? root->left : root->right, key);

    if (root->left and root->left->priority > root->priority)
        right_rotation(root);

    if (root->right and root->right->priority > root->priority)
        left_rotation(root);
}
```

Remove

```
bool remove(TreapNode* &root, int key) {
        if (root == NULL) return false;

        if (key < root->key) return remove(root->left, key);
        if (key > root->key) return remove(root->right, key);
```

```
// Case 1: TreapNode to be deleted has no children (it is a leaf TreapNode)
if (!root->left and !root->right) {
    delete root;
    root = NULL;
}
```

```
// Case 2: TreapNode to be deleted has only one child
else if (!root->left or !root->right) {
    TreapNode* child = (root->left) ? root->left : root->right;

    TreapNode* old_root = root;
    root = child;

    delete old_root;
}
```

```
    // Case 3: TreapNode to be deleted has two children
    else {
        if (root->left->priority < root->right->priority) {
            left_rotation(root);
            remove(root->left, key);
        } else {
            right_rotation(root);
            remove(root->right, key);
        }
    }

    return true;
}
```

Thanks!