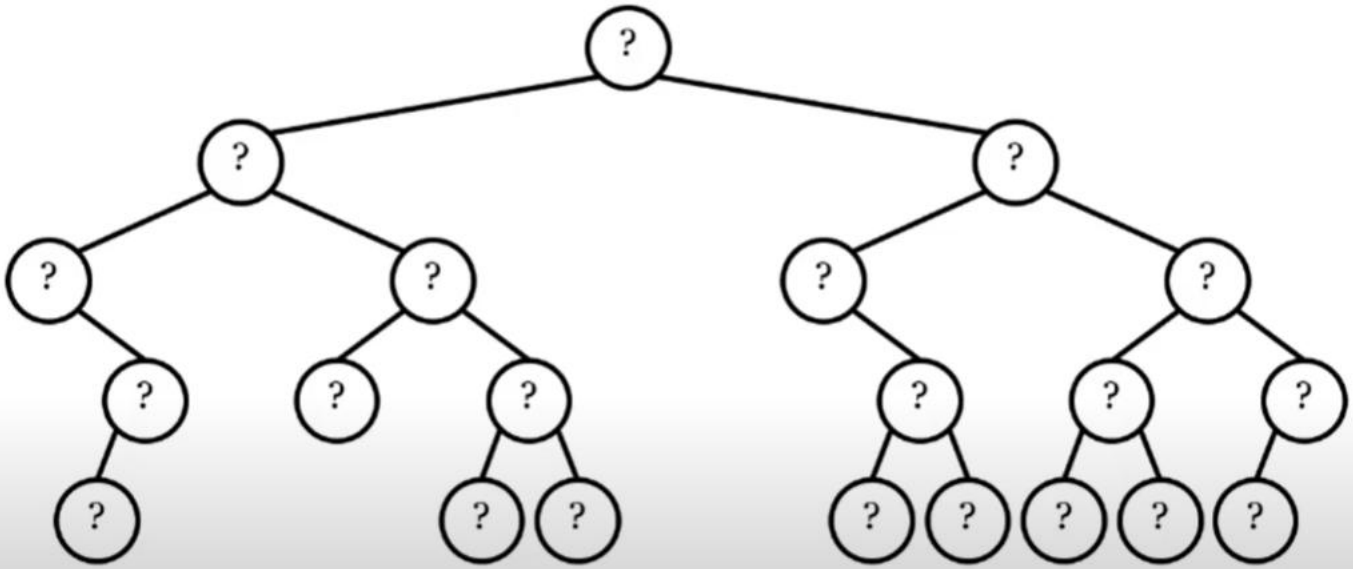
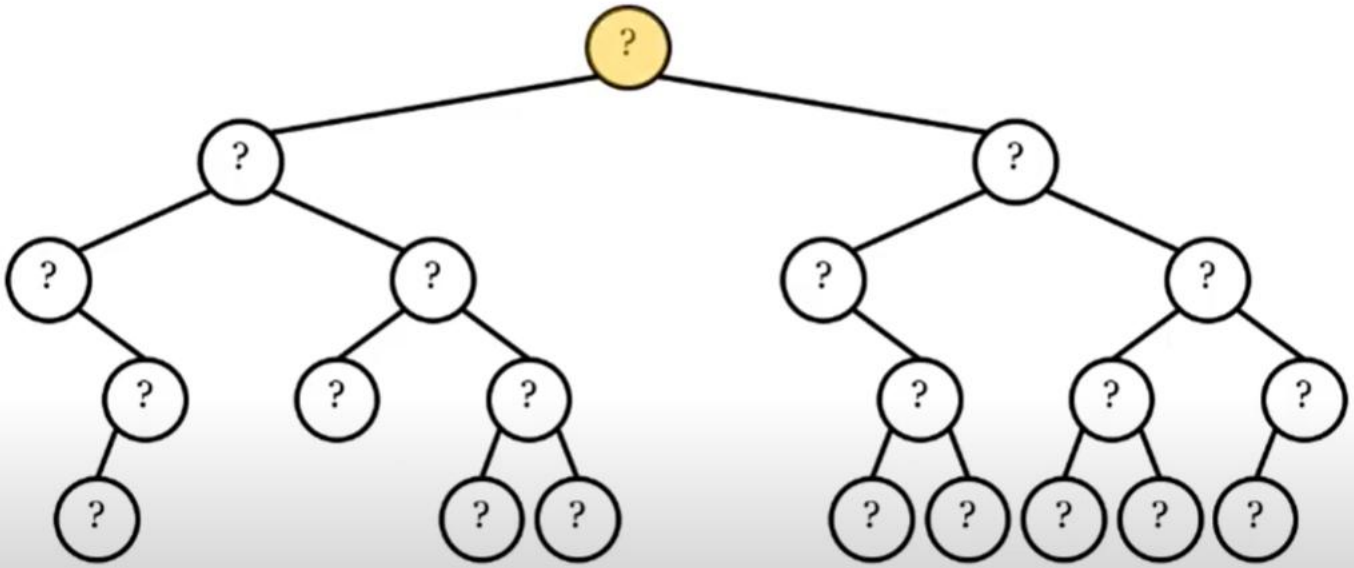
Декартово дерево по неявному ключу

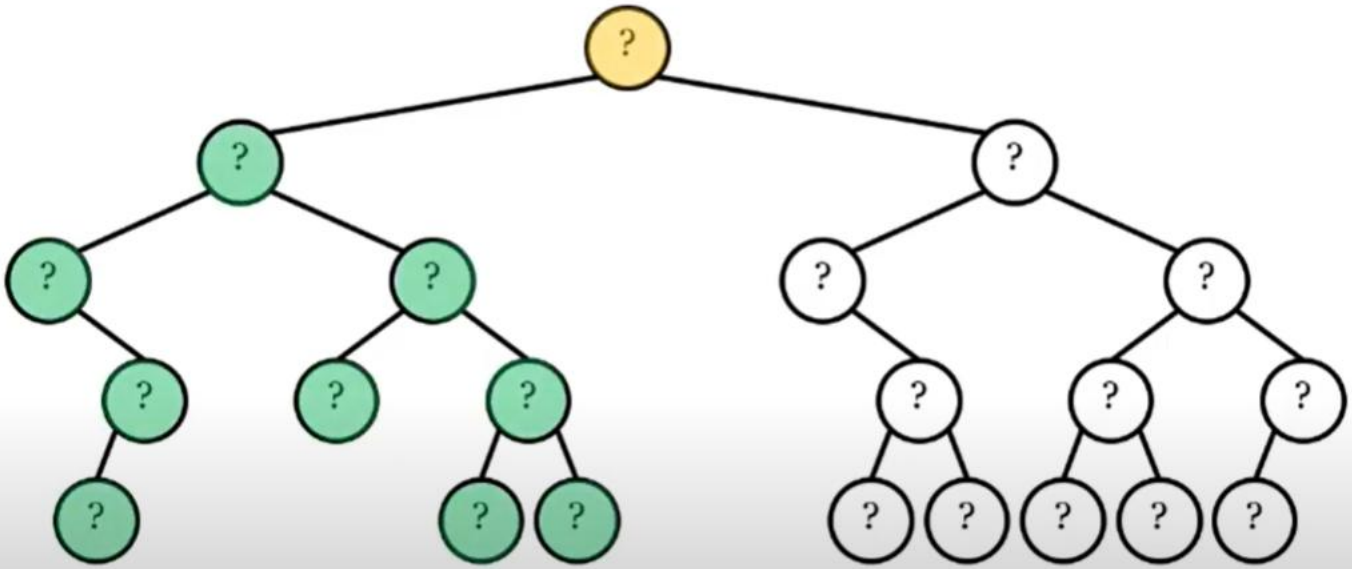
Ключами были числа от 0 до 20. Сможем ли мы их восстановить?



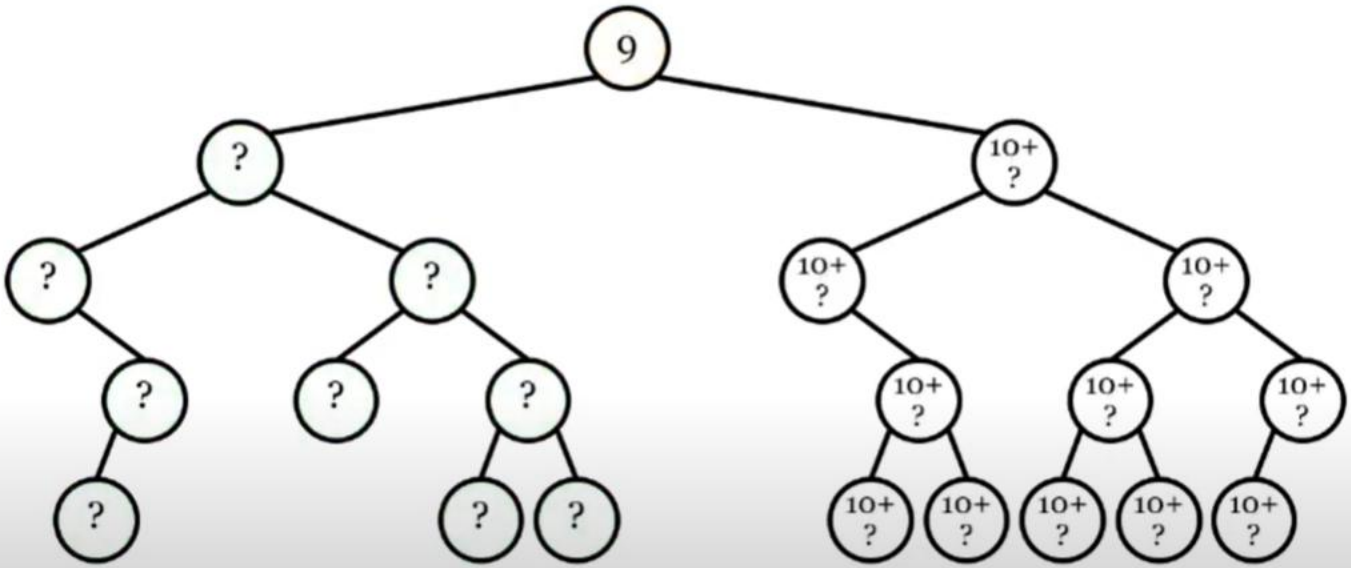
Ключами были числа от 0 до 20. Сможем ли мы их восстановить?



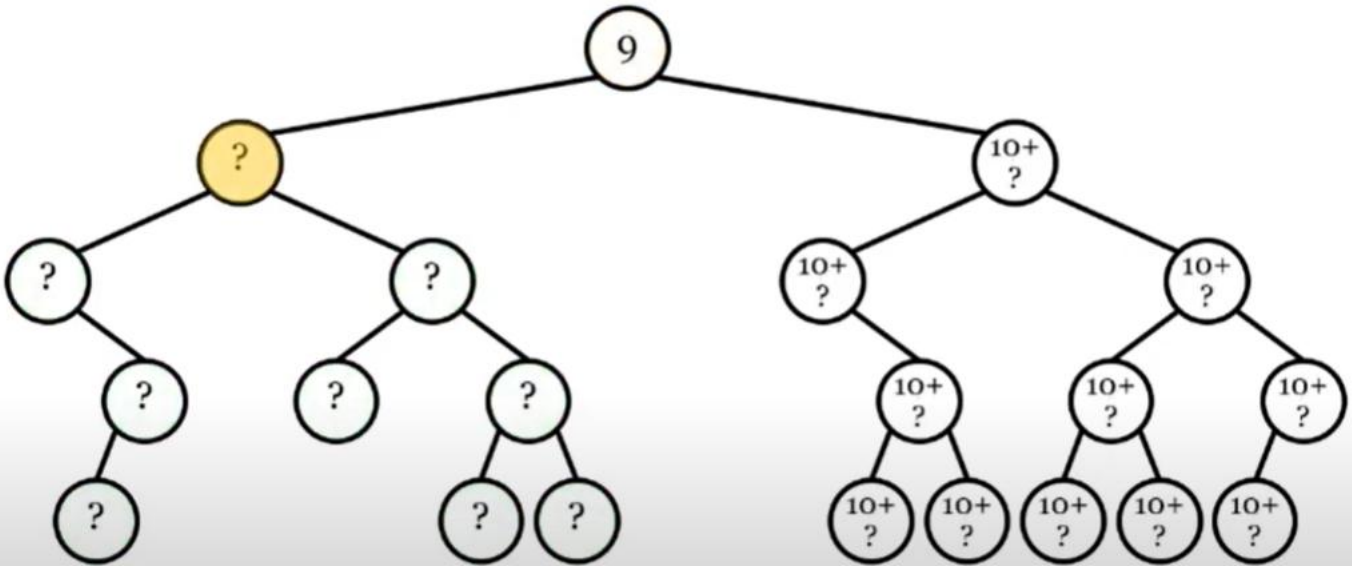
Ключами были числа от 0 до 20. Сможем ли мы их восстановить?



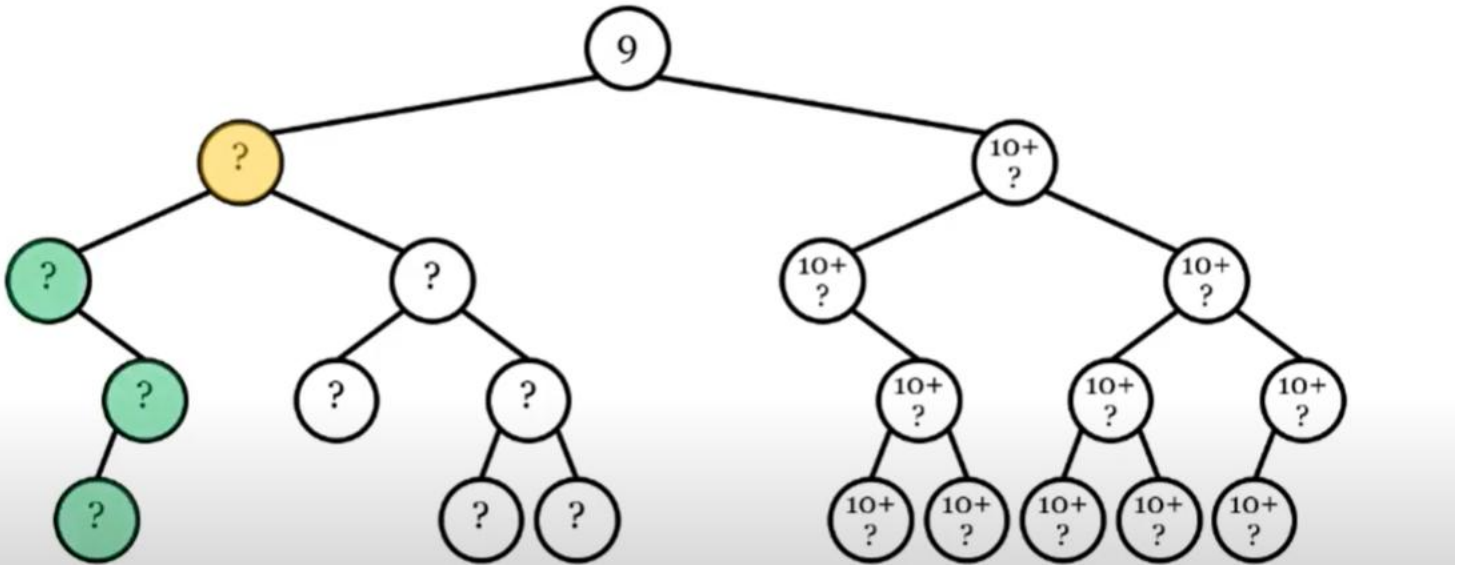
Ключами были числа от 0 до 20. Сможем ли мы их восстановить?



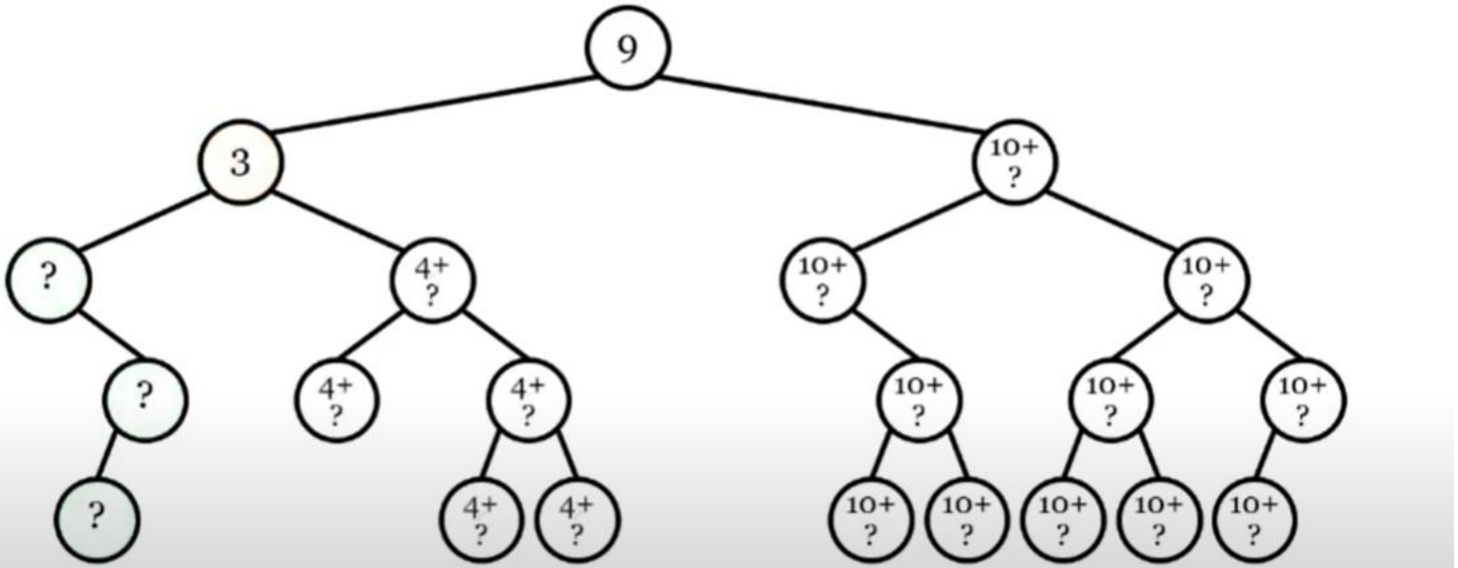
Ключами были числа от 0 до 20. Сможем ли мы их восстановить?



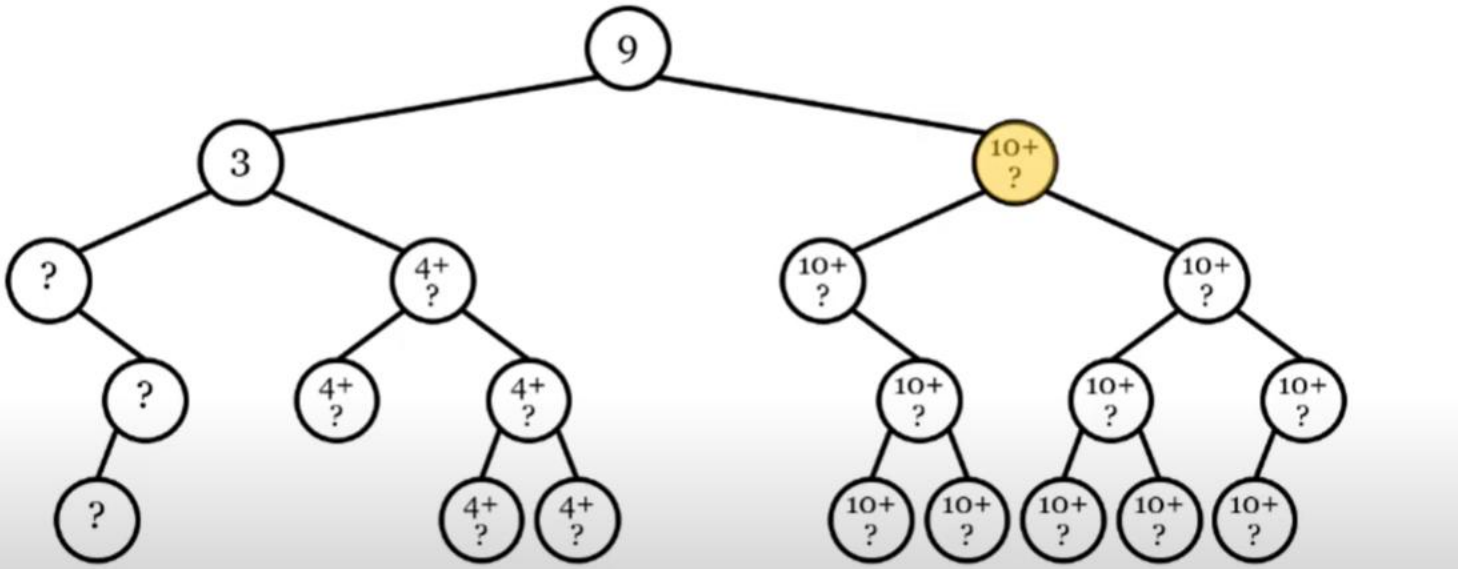
Ключами были числа от 0 до 20. Сможем ли мы их восстановить?



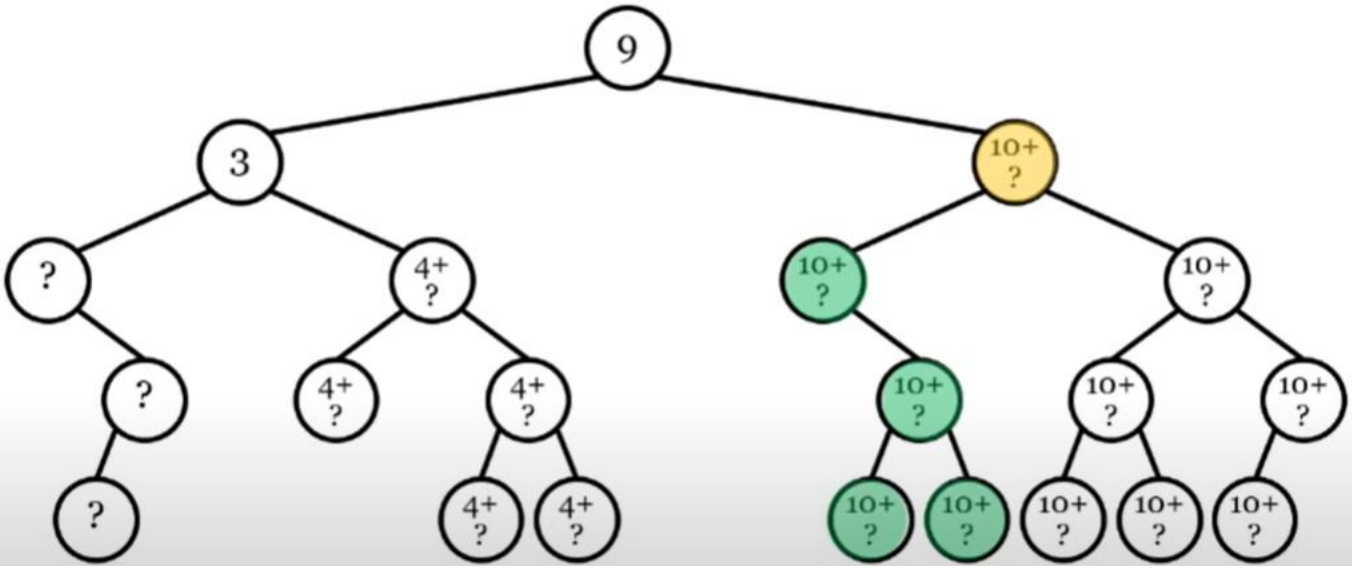
Ключами были числа от 0 до 20. Сможем ли мы их восстановить?



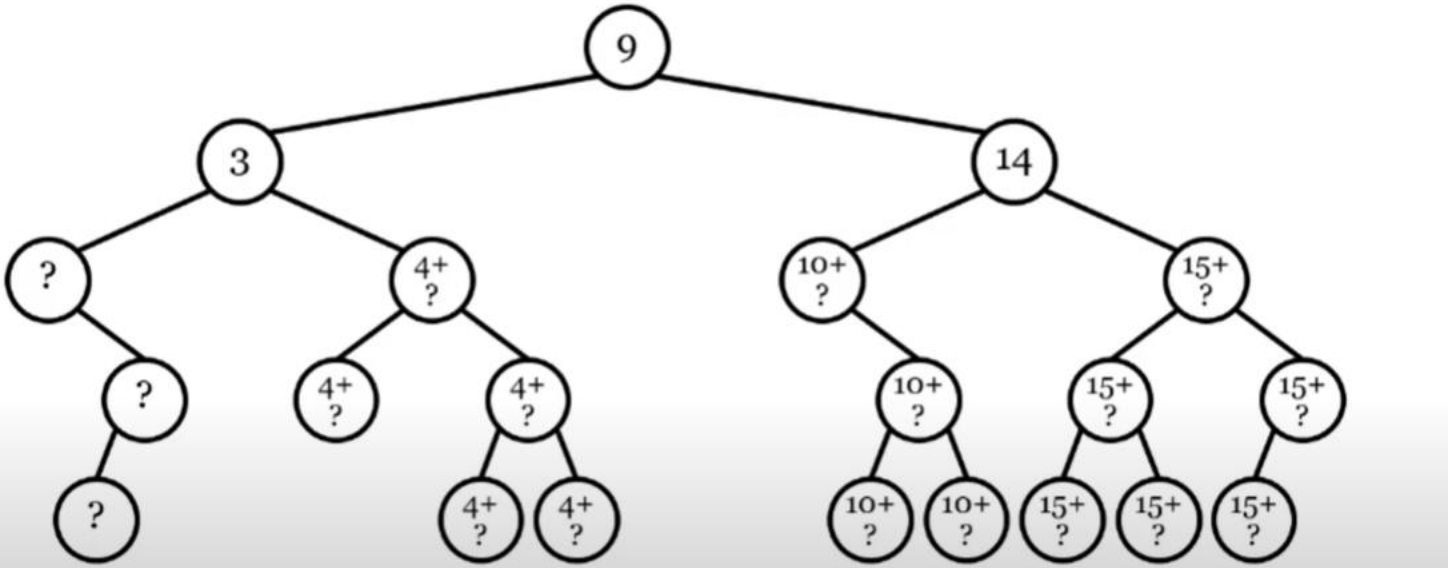
Ключами были числа от 0 до 20. Сможем ли мы их восстановить?



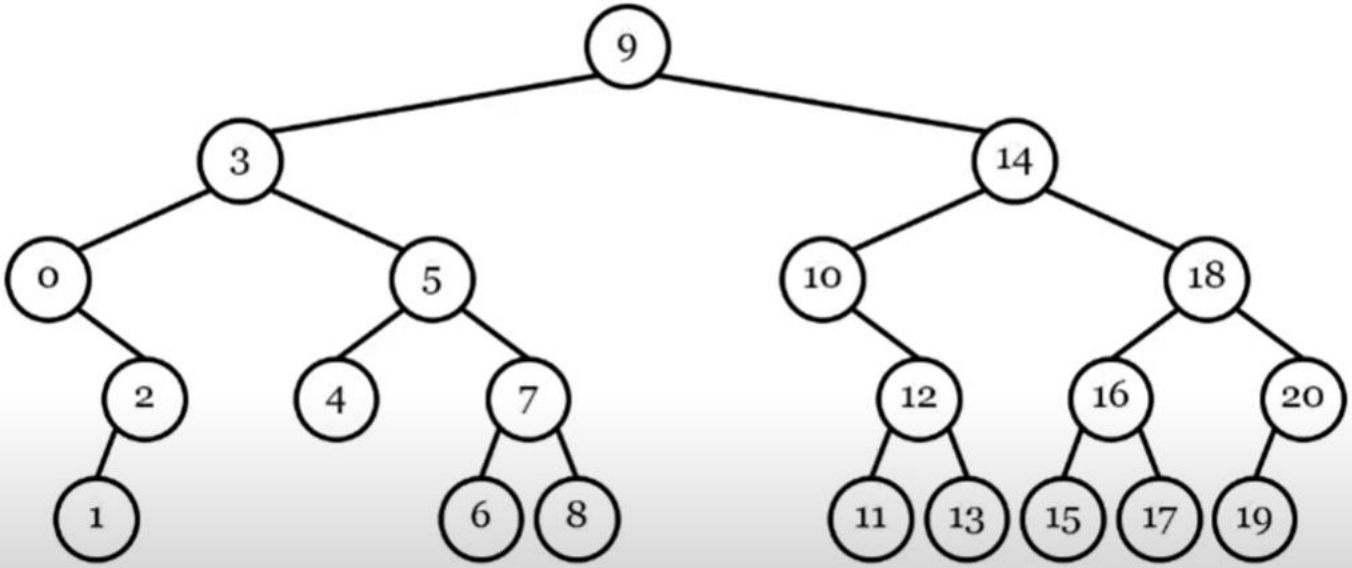
Ключами были числа от 0 до 20. Сможем ли мы их восстановить?



Ключами были числа от 0 до 20. Сможем ли мы их восстановить?



Ключами были числа от 0 до 20. Сможем ли мы их восстановить?



```
struct Node {
    int priority, size, value;
    Node *l = nullptr, *r = nullptr;
    Node (int value): value(value), priority(generator()), size(1) {}
} *root = nullptr;
```

```
struct Node {
    int priority, size, value;
    Node *l = nullptr, *r = nullptr;
    Node (int value): value(value), priority(generator()), size(1) {}
} *root = nullptr;
```

```
static Node *merge(Node *a, Node *b){
    if (!a || !b){
        return a ? a : b;
    }
    if (a->priority > b->priority){
        a->r = merge(a->r, b);
        update(a);
        return a;
    }
    else {
        b->l = merge(a, b->l);
        update(b);
        return b;
    }
}
```

```
static void split(Node *n, int key, Node *&a, Node *&b){
    if (!n){
        a = b = nullptr;
        return ;
    }
    if (n -> key < key){
        split(n->r, key, n->r, b);
        a = n;
    }
    else {
        split(n->l, key, a, n->l);
        b = n;
    }
    update(a);
    update(b);
}
```

```
struct Node {
    int priority, size, value;
    Node *l = nullptr, *r = nullptr;
    Node (int value): value(value), priority(generator()), size(1) {}
} *root = nullptr;
```

```
static Node *merge(Node *a, Node *b){
    if (!a || !b){
        return a ? a : b;
    }
    if (a->priority > b->priority){
        a->r = merge(a->r, b);
        update(a);
        return a;
    }
    else {
        b->l = merge(a, b->l);
        update(b);
        return b;
    }
}
```

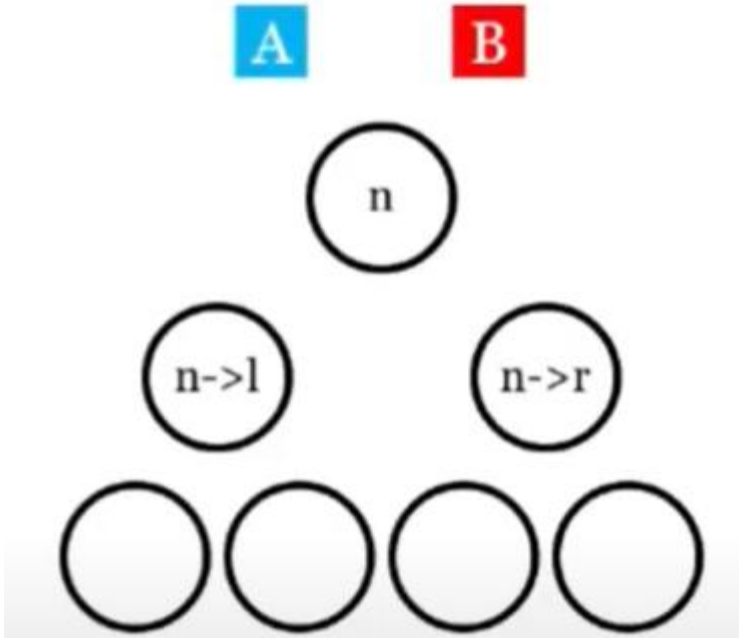
```
static void split(Node *n, int k, Node *&a, Node *&b){
    if (!n){
        a = b = nullptr;
        return ;
    }
    if (n -> key < key){
        split(n->r, key, n->r, b);
        a = n;
    }
    else {
        split(n->l, key, a, n->l);
        b = n;
    }
    update(a);
    update(b);
}
```

```
struct Node {
    int priority, size, value;
    Node *l = nullptr, *r = nullptr;
    Node (int value): value(value), priority(generator()), size(1) {}
} *root = nullptr;
```

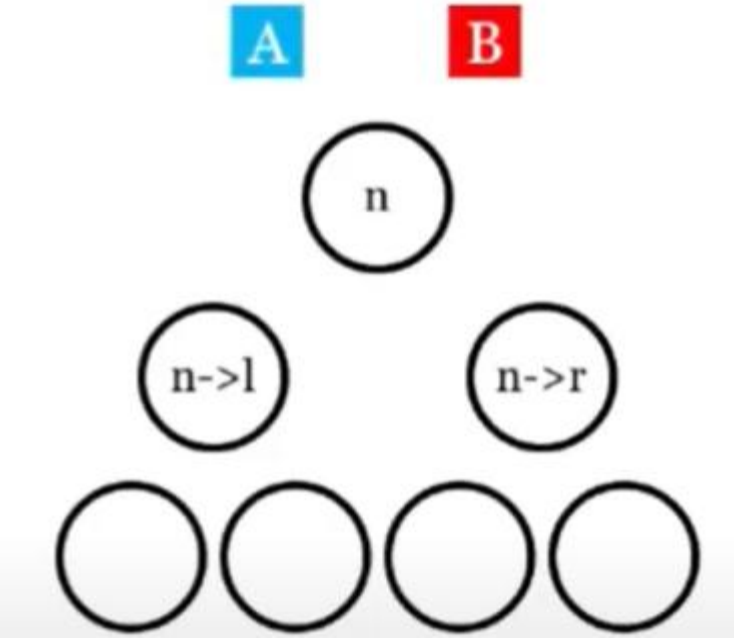
```
static Node *merge(Node *a, Node *b){
    if (!a || !b){
        return a ? a : b;
    }
    if (a->priority > b->priority){
        a->r = merge(a->r, b);
        update(a);
        return a;
    }
    else {
        b->l = merge(a, b->l);
        update(b);
        return b;
    }
}
```

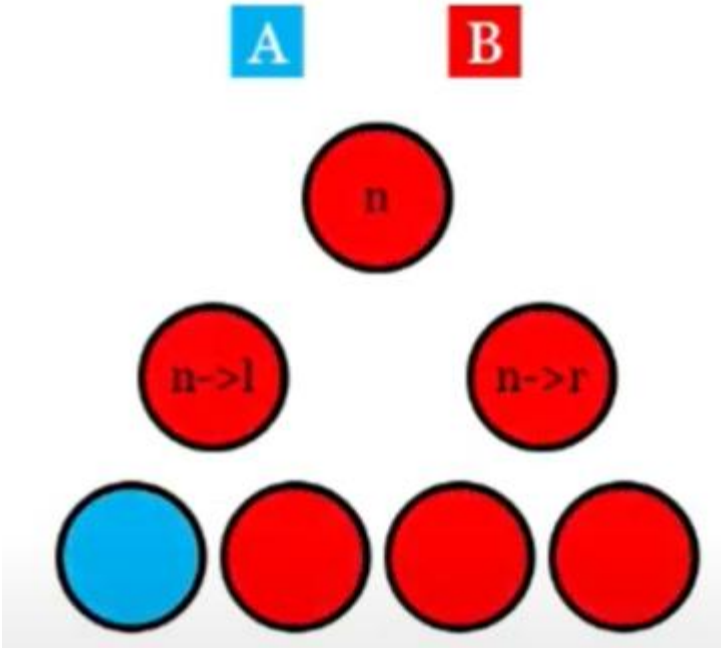
```
static void split(Node *n, int k, Node *&a, Node *&b){
    if (!n){
        a = b = nullptr;
        return ;
    }
    if (n -> key < key){
        split(n->r, key, n->r, b);
        a = n;
    }
    else {
        split(n->l, key, a, n->l);
        b = n;
    }
    update(a);
    update(b);
}
```

Новый смысл split: в дерево a попадают первые k вершин, в дерево b – все остальные

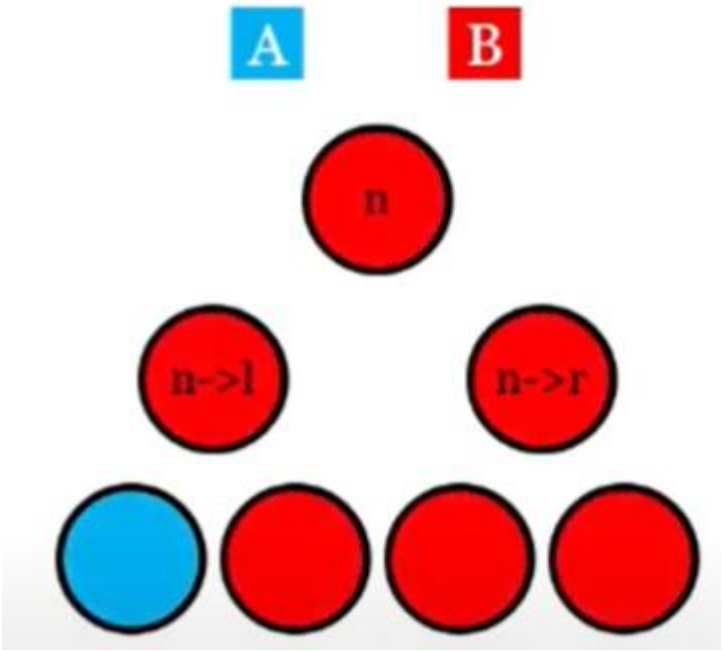

A
B
n
n->l
n->r

split(n, 1, a, b)

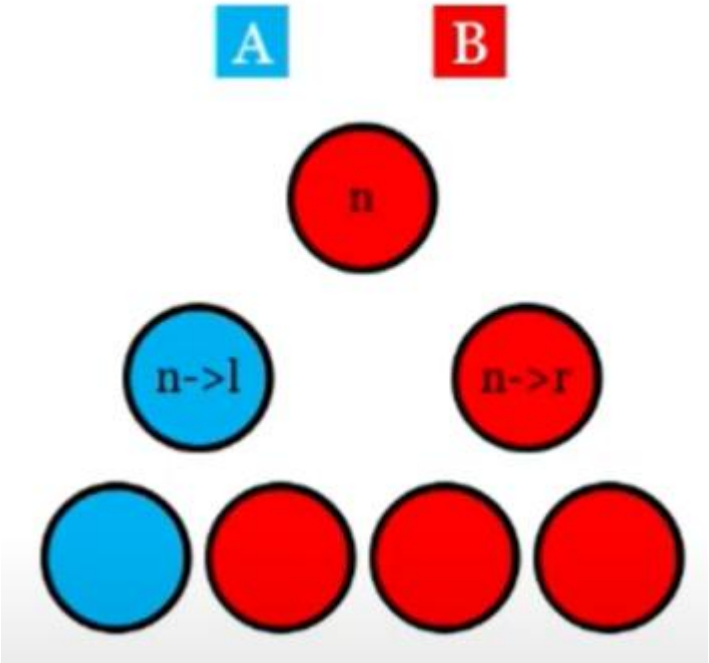




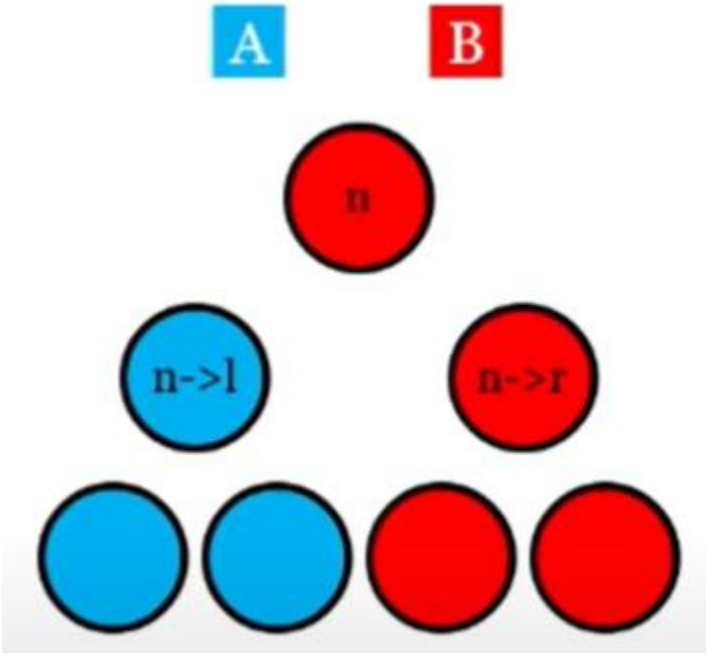
split(n, 1, a, b)



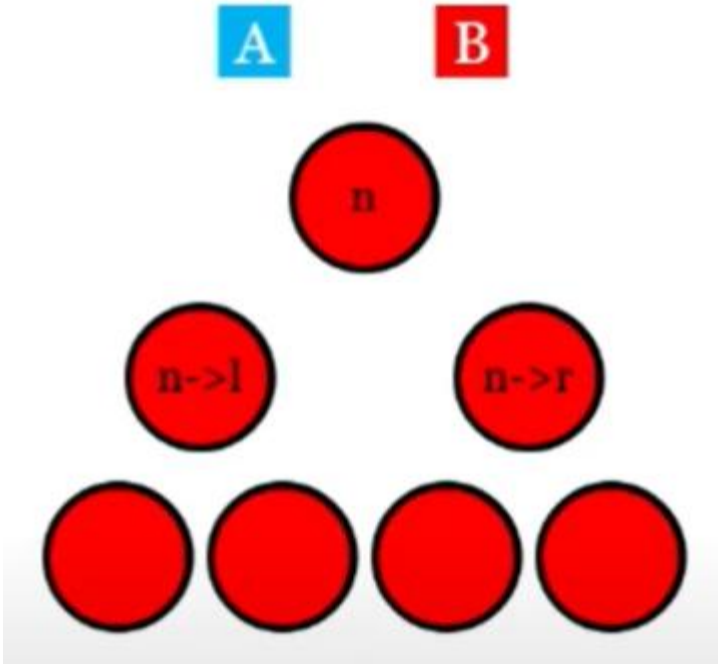
split(n, 1, a, b) -> split(n->l, 1, a, n->l), b = n



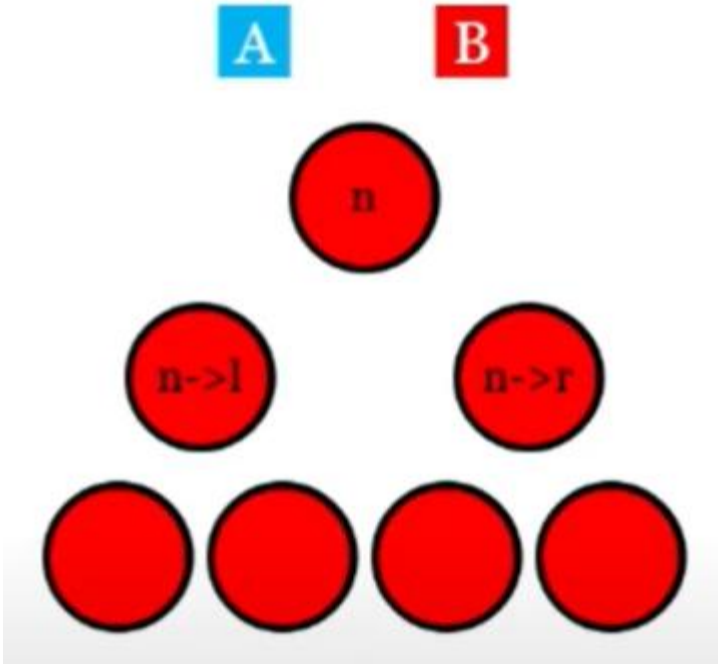
split(n, 1, a, b) -> split(n->l, 1, a, n->l), b = n
split(n, 2, a, b) -> split(n->l, 2, a, n->l), b = n



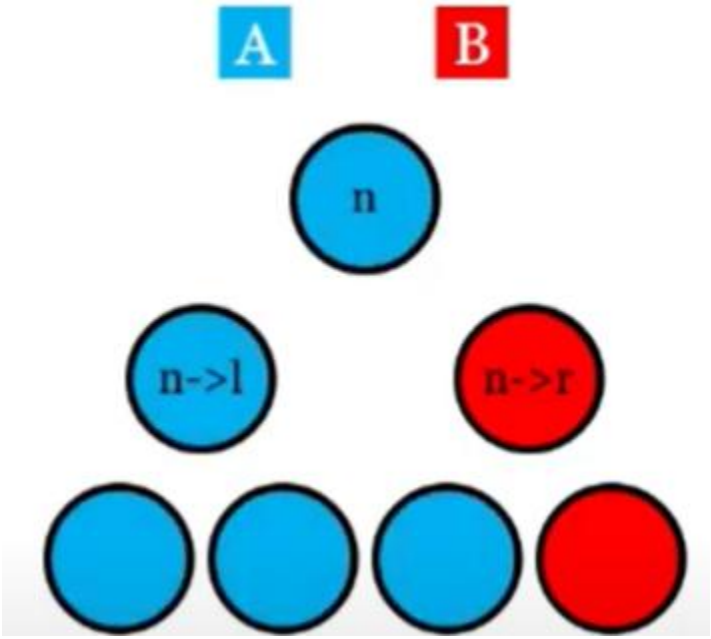
split(n, 1, a, b) -> split(n->l, 1, a, n->l), b = n
split(n, 2, a, b) -> split(n->l, 2, a, n->l), b = n
split(n, 3, a, b) -> split(n->l, 3, a, n->l), b = n



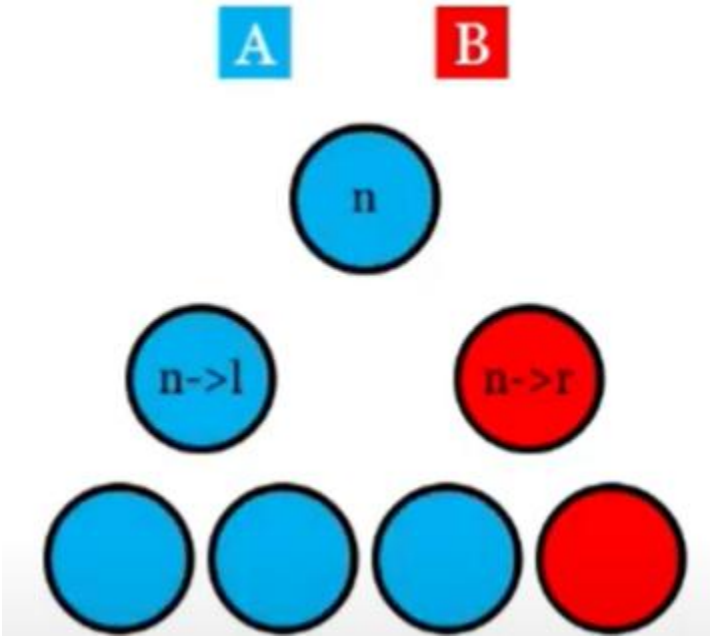
split(n, 1, a, b) -> split(n->l, 1, a, n->l), b = n
split(n, 2, a, b) -> split(n->l, 2, a, n->l), b = n
split(n, 3, a, b) -> split(n->l, 3, a, n->l), b = n
split(n, 0, a, b) -> split(n->l, 0, a, n->l), b = n



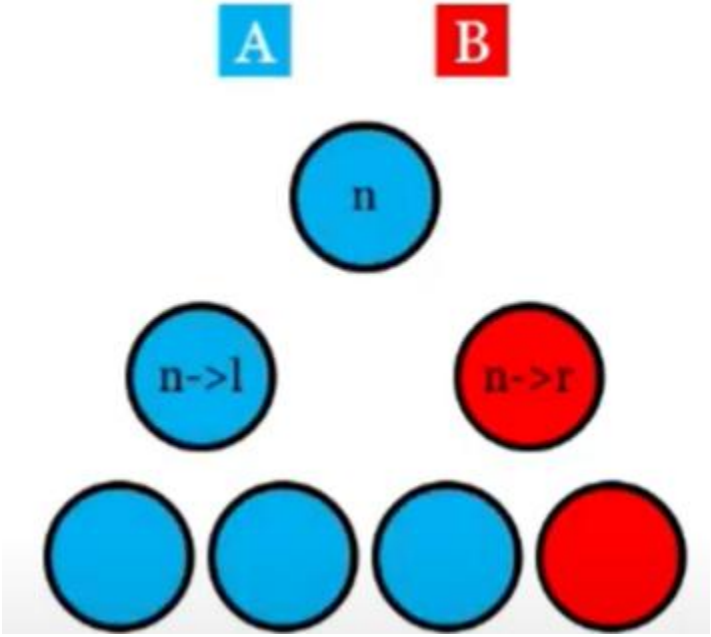
split(n, 1, a, b) -> split(n->l, 1, a, n->l), b = n
split(n, 2, a, b) -> split(n->l, 2, a, n->l), b = n
split(n, 3, a, b) -> split(n->l, 3, a, n->l), b = n
split(n, 0, a, b) -> split(n->l, 0, a, n->l), b = n
split(n, 5, a, b)



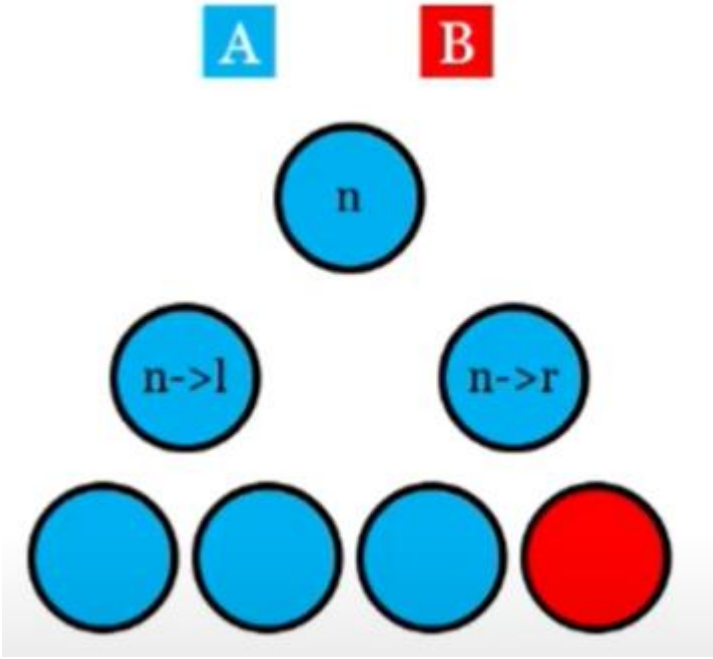
split(n, 1, a, b) -> split(n->l, 1, a, n->l), b = n
split(n, 2, a, b) -> split(n->l, 2, a, n->l), b = n
split(n, 3, a, b) -> split(n->l, 3, a, n->l), b = n
split(n, 0, a, b) -> split(n->l, 0, a, n->l), b = n
split(n, 5, a, b)



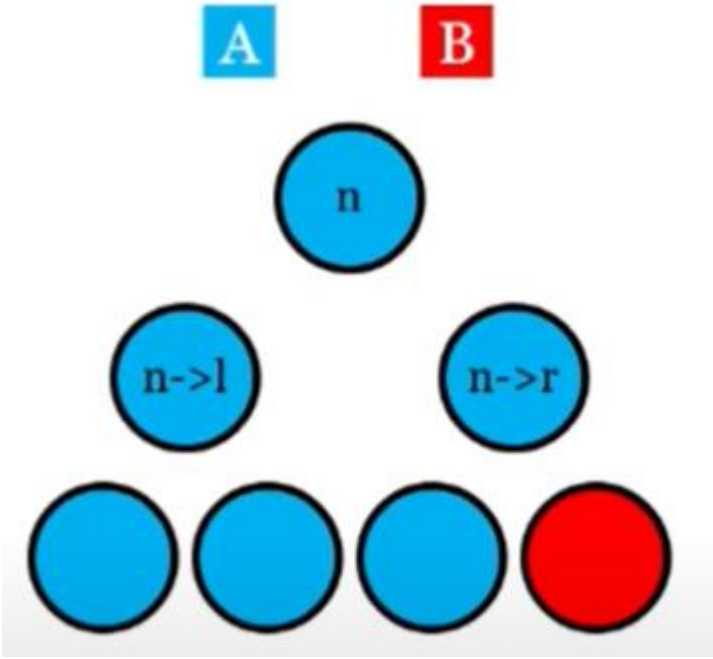
split(n, 1, a, b) -> split(n->l, 1, a, n->l), b = n
split(n, 2, a, b) -> split(n->l, 2, a, n->l), b = n
split(n, 3, a, b) -> split(n->l, 3, a, n->l), b = n
split(n, 0, a, b) -> split(n->l, 0, a, n->l), b = n
split(n, 5, a, b) -> split(n -> r, 5, n -> r, b), a = n



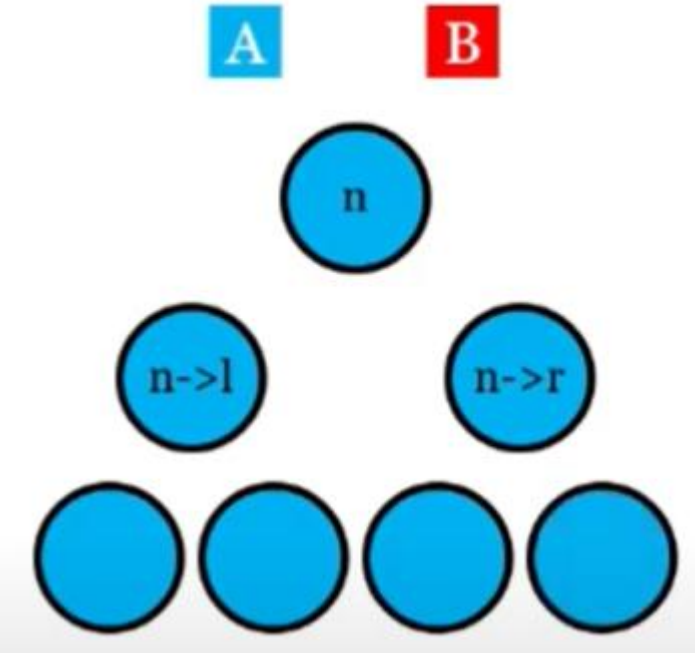
split(n, 1, a, b) -> split(n->l, 1, a, n->l), b = n
split(n, 2, a, b) -> split(n->l, 2, a, n->l), b = n
split(n, 3, a, b) -> split(n->l, 3, a, n->l), b = n
split(n, 0, a, b) -> split(n->l, 0, a, n->l), b = n
split(n, 5, a, b) -> split(n -> r, 1, n -> r, b), a = n



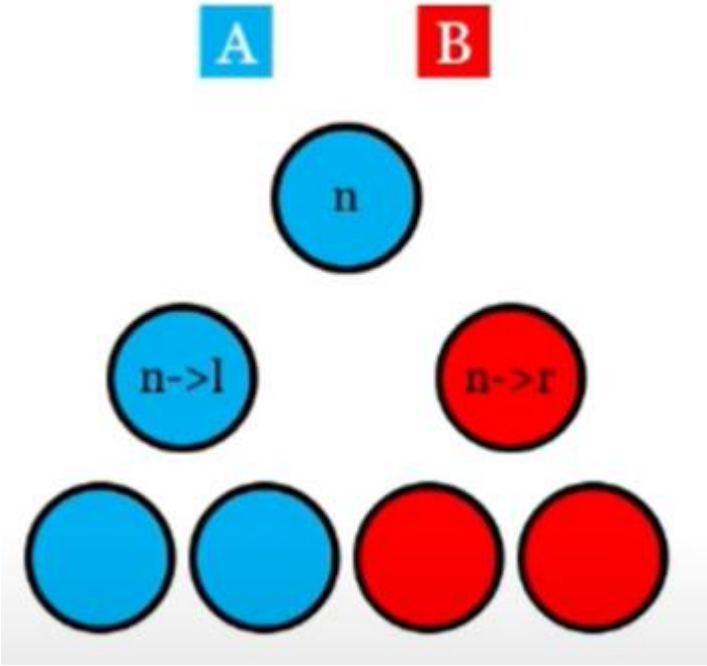
split(n, 1, a, b) -> split(n->l, 1, a, n->l), b = n
split(n, 2, a, b) -> split(n->l, 2, a, n->l), b = n
split(n, 3, a, b) -> split(n->l, 3, a, n->l), b = n
split(n, 0, a, b) -> split(n->l, 0, a, n->l), b = n
split(n, 5, a, b) -> split(n -> r, 1, n -> r, b), a = n
split(n, 6, a, b) -> split(n -> r, 2, n -> r, b), a = n



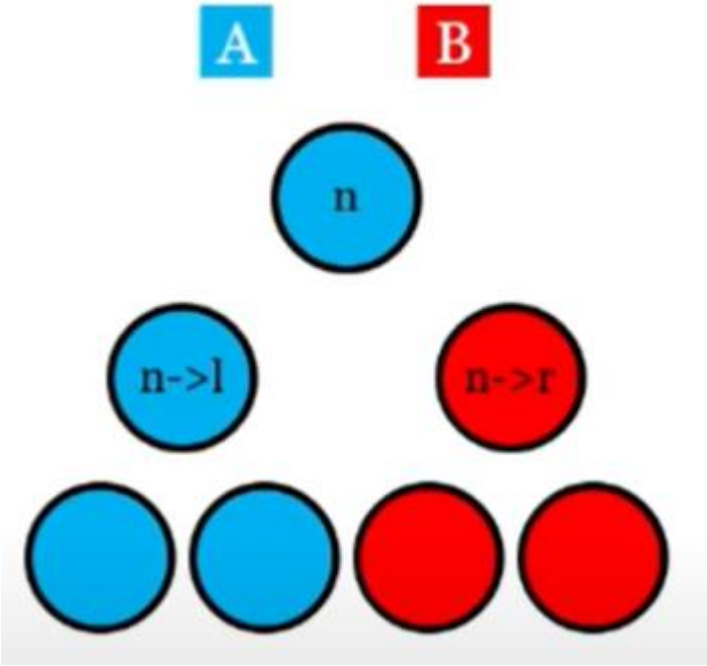
split(n, 1, a, b) -> split(n->l, 1, a, n->l), b = n
split(n, 2, a, b) -> split(n->l, 2, a, n->l), b = n
split(n, 3, a, b) -> split(n->l, 3, a, n->l), b = n
split(n, 0, a, b) -> split(n->l, 0, a, n->l), b = n
split(n, 5, a, b) -> split(n -> r, 1, n -> r, b), a = n
split(n, 6, a, b) -> split(n -> r, 2, n -> r, b), a = n



split(n, 1, a, b) -> split(n->l, 1, a, n->l), b = n
split(n, 2, a, b) -> split(n->l, 2, a, n->l), b = n
split(n, 3, a, b) -> split(n->l, 3, a, n->l), b = n
split(n, 0, a, b) -> split(n->l, 0, a, n->l), b = n
split(n, 5, a, b) -> split(n -> r, 1, n -> r, b), a = n
split(n, 6, a, b) -> split(n -> r, 2, n -> r, b), a = n
split(n, 7, a, b) -> split(n -> r, 3, n -> r, b), a = n



split(n, 1, a, b) -> split(n->l, 1, a, n->l), b = n
split(n, 2, a, b) -> split(n->l, 2, a, n->l), b = n
split(n, 3, a, b) -> split(n->l, 3, a, n->l), b = n
split(n, 0, a, b) -> split(n->l, 0, a, n->l), b = n
split(n, 5, a, b) -> split(n -> r, 1, n -> r, b), a = n
split(n, 6, a, b) -> split(n -> r, 2, n -> r, b), a = n
split(n, 7, a, b) -> split(n -> r, 3, n -> r, b), a = n
split(n, 4, a, b)



split(n, 1, a, b) -> split(n->l, 1, a, n->l), b = n
split(n, 2, a, b) -> split(n->l, 2, a, n->l), b = n
split(n, 3, a, b) -> split(n->l, 3, a, n->l), b = n
split(n, 0, a, b) -> split(n->l, 0, a, n->l), b = n
split(n, 5, a, b) -> split(n -> r, 1, n -> r, b), a = n
split(n, 6, a, b) -> split(n -> r, 2, n -> r, b), a = n
split(n, 7, a, b) -> split(n -> r, 3, n -> r, b), a = n
split(n, 4, a, b) -> split(n -> r, 0, n -> r, b), a = n

split(n, 1, a, b) -> split(n->l, 1, a, n->l), b = n
split(n, 2, a, b) -> split(n->l, 2, a, n->l), b = n
split(n, 3, a, b) -> split(n->l, 3, a, n->l), b = n
split(n, 0, a, b) -> split(n->l, 0, a, n->l), b = n
split(n, 5, a, b) -> split(n -> r, 1, n -> r, b), a = n
split(n, 6, a, b) -> split(n -> r, 2, n -> r, b), a = n
split(n, 7, a, b) -> split(n -> r, 3, n -> r, b), a = n
split(n, 4, a, b) -> split(n -> r, 0, n -> r, b), a = n

```
split(n, k, a, b){
    if (n -> key < key){
        split(n->r, key, n->r, b);
        a = n;
    }
    else {
        split(n->l, key, a, n->l);
        b = n;
    }
}
```

```
split(n, k, a, b){
    if (get_size(n -> l) < k){
        split(n -> r, k - get_size(n -> l) - 1, n -> r, b);
        a = n;
    }
    else {
        split(n -> l, k, a, n -> l);
        b = n;
    }
}
```

```
static Node *merge(Node *a, Node *b){
    if (!a || !b){
        return a ? a : b;
    }
    if (a->priority > b->priority){
        a->r = merge(a->r, b);
        update(a);
        return a;
    }
    else {
        b->l = merge(a, b->l);
        update(b);
        return b;
    }
}
```

```
static void split(Node *n, int k, Node *&a, Node *&b){
    if (!n){
        a = b = nullptr;
        return ;
    }
    if (get_size(n -> l) < k){
        split(n->r, k - get_size(n -> l) - 1, n->r, b);
        a = n;
    }
    else {
        split(n->l, k, a, n->l);
        b = n;
    }
    update(a);
    update(b);
}
```

```
static Node *merge(Node *a, Node *b){
    if (!a || !b){
        return a ? a : b;
    }
    if (a->priority > b->priority){
        a->r = merge(a->r, b);
        update(a);
        return a;
    }
    else {
        b->l = merge(a, b->l);
        update(b);
        return b;
    }
}
```

```
static void split(Node *n, int k, Node *&a, Node *&b){
    if (!n){
        a = b = nullptr;
        return ;
    }
    if (get_size(n -> l) < k){
        split(n->r, k - get_size(n -> l) - 1, n->r, b);
        a = n;
    }
    else {
        split(n->l, k, a, n->l);
        b = n;
    }
    update(a);
    update(b);
}
```

```
bool find(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key + 1, equal, greater);
    bool result = equal;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
void insert(int key, int value){
    Node *greater, *less;
    split(root, key, less, greater);
    less = merge(less, new Node(key, value));
    root = merge(less, greater);
}

void erase(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key, equal, greater);
    root = merge(less, greater);
}
```

```
bool find(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key + 1, equal, greater);
    bool result = equal;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
void insert(int key, int value){
    Node *greater, *less;
    split(root, key, less, greater);
    less = merge(less, new Node(key, value));
    root = merge(less, greater);
}

void erase(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key, equal, greater);
    root = merge(less, greater);
}
```

```
bool find(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key + 1, equal, greater);
    bool result = equal;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
void insert(int key, int value){
    Node *greater, *less;
    split(root, key, less, greater);
    less = merge(less, new Node(key, value));
    root = merge(less, greater);
}
```

```
void erase(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key, equal, greater);
    root = merge(less, greater);
}
```

```
int get(int index){
    Node *greater, *equal, *less;
    split(root, index, less, greater);
    split(greater, 1, equal, greater);
    int result = equal -> value;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
bool find(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key + 1, equal, greater);
    bool result = equal;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
void insert(int key, int value){
    Node *greater, *less;
    split(root, key, less, greater);
    less = merge(less, new Node(key, value));
    root = merge(less, greater);
}
```

```
void erase(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key, equal, greater);
    root = merge(less, greater);
}
```

```
int get(int index){
    Node *greater, *equal, *less;
    split(root, index, less, greater);
    split(greater, 1, equal, greater);
    int result = equal -> value;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
void push_back(int value){
    root = merge(root, new Node(value));
}
```

```
bool find(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key + 1, equal, greater);
    bool result = equal;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
void insert(int key, int value){
    Node *greater, *less;
    split(root, key, less, greater);
    less = merge(less, new Node(key, value));
    root = merge(less, greater);
}
```

```
void erase(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key, equal, greater);
    root = merge(less, greater);
}
```

```
int get(int index){
    Node *greater, *equal, *less;
    split(root, index, less, greater);
    split(greater, 1, equal, greater);
    int result = equal -> value;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
void push_back(int value){
    root = merge(root, new Node(value));
}
```

```
void push_front(int value){
    root = merge(new Node(value), root);
}
```

```
bool find(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key + 1, equal, greater);
    bool result = equal;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
void insert(int key, int value){
    Node *greater, *less;
    split(root, key, less, greater);
    less = merge(less, new Node(key, value));
    root = merge(less, greater);
}
```

```
void erase(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key, equal, greater);
    root = merge(less, greater);
}
```

```
int get(int index){
    Node *greater, *equal, *less;
    split(root, index, less, greater);
    split(greater, 1, equal, greater);
    int result = equal -> value;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
void push_back(int value){
    root = merge(root, new Node(value));
}
```

```
void push_front(int value){
    root = merge(new Node(value), root);
}
```

```
void insert(int value, int index){
    Node *greater, *less;
    split(root, index, less, greater);
    root = merge(merge(less, new Node(value)), greater);
}
```

```
bool find(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key + 1, equal, greater);
    bool result = equal;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
void insert(int key, int value){
    Node *greater, *less;
    split(root, key, less, greater);
    less = merge(less, new Node(key, value));
    root = merge(less, greater);
}

void erase(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key, equal, greater);
    root = merge(less, greater);
}
```

```
int get(int index){
    Node *greater, *equal, *less;
    split(root, index, less, greater);
    split(greater, 1, equal, greater);
    int result = equal -> value;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
void push_back(int value){
    root = merge(root, new Node(value));
}

void push_front(int value){
    root = merge(new Node(value), root);
}
```

```
void insert(int value, int index){
    Node *greater, *less;
    split(root, index, less, greater);
    root = merge(merge(less, new Node(value)), greater);
}

void erase(int index){
    Node *greater, *equal, *less;
    split(root, index, less, greater);
    split(greater, 1, equal, greater);
    root = merge(less, greater);
}
```

```
bool find(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key + 1, equal, greater);
    bool result = equal;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
void insert(int key, int value){
    Node *greater, *less;
    split(root, key, less, greater);
    less = merge(less, new Node(key, value));
    root = merge(less, greater);
}

void erase(int key){
    Node *greater, *equal, *less;
    split(root, key, less, greater);
    split(greater, key, equal, greater);
    root = merge(less, greater);
}
```

```
int get(int index){
    Node *greater, *equal, *less;
    split(root, index, less, greater);
    split(greater, 1, equal, greater);
    int result = equal -> value;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
void push_back(int value){
    root = merge(root, new Node(value));
}

void push_front(int value){
    root = merge(new Node(value), root);
}
```

```
void insert(int value, int index){
    Node *greater, *less;
    split(root, index, less, greater);
    root = merge(merge(less, new Node(value)), greater);
}
```

```
void erase(int index){
    Node *greater, *equal, *less;
    split(root, index, less, greater);
    split(greater, 1, equal, greater);
    root = merge(less, greater);
}
```

```
void erase(int l, int r){
    Node *greater, *equal, *less;
    split(root, l, less, greater);
    split(greater, r - l + 1, equal, greater);
    root = merge(less, greater);
}
```

```
int get(int index){
    Node *greater, *equal, *less;
    split(root, index, less, greater);
    split(greater, 1, equal, greater);
    int result = equal -> value;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
void push_back(int value){
    root = merge(root, new Node(value));
}

void push_front(int value){
    root = merge(new Node(value), root);
}
```

```
void insert(int value, int index){
    Node *greater, *less;
    split(root, index, less, greater);
    root = merge(merge(less, new Node(value)), greater);
}
```

```
void erase(int index){
    Node *greater, *equal, *less;
    split(root, index, less, greater);
    split(greater, 1, equal, greater);
    root = merge(less, greater);
}
```

```
void erase(int l, int r){
    Node *greater, *equal, *less;
    split(root, l, less, greater);
    split(greater, r - l + 1, equal, greater);
    root = merge(less, greater);
}
```

```
int get(int index){
    Node *greater, *equal, *less;
    split(root, index, less, greater);
    split(greater, 1, equal, greater);
    int result = equal -> value;
    root = merge(merge(less, equal), greater);
    return result;
}
```

```
void push_back(int value){
    root = merge(root, new Node(value));
}

void push_front(int value){
    root = merge(new Node(value), root);
}
```

```
void insert(int value, int index){
    Node *greater, *less;
    split(root, index, less, greater);
    root = merge(merge(less, new Node(value)), greater);
}
```

```
int size(){
    return get_size(root);
}
```

```
void erase(int index){
    Node *greater, *equal, *less;
    split(root, index, less, greater);
    split(greater, 1, equal, greater);
    root = merge(less, greater);
}
```

```
void erase(int l, int r){
    Node *greater, *equal, *less;
    split(root, l, less, greater);
    split(greater, r - l + 1, equal, greater);
    root = merge(less, greater);
}
```

Задача A. В начало строя!

Имя входного файла:	`movetofront.in`
Имя выходного файла:	`movetofront.out`
Ограничение по времени:	4 секунды
Ограничение по памяти:	256 мегабайт

Капрал Питуца любит командовать своим отрядом. Его любимый приказ «в начало строя». Он выстраивает свой отряд в шеренгу и оглашает последовательность приказов. Каждая приказ имеет вид «Солдаты с l_i по r_i — в начало строя!»

Пронумеруем солдат в начальном положении с 1 до n, слева направо. Приказ «Солдаты с l_i по r_i — в начало строя!» означает, что солдаты, стоящие с l_i по r_i включительно, перемещаются в начало строя, сохраняя относительный порядок.

Например, если в некоторый момент солдаты стоят в порядке $2, 3, 6, 1, 5, 4$, после приказа: «Солдаты с 2 по 4 — в начало строя!» порядок будет $3, 6, 1, 2, 5, 4$.

По данной последовательности приказов найти конечный порядок солдат в строю.

Формат входного файла

В первой строке два целых числа n and m ($2 \leqslant n \leqslant 100\,000$, $1 \leqslant m \leqslant 100\,000$) — количество солдат и количество приказов. Следующие m строк содержат по два целых числа l_i и r_i ($1 \leqslant l_i \leqslant r_i \leqslant n$).

Формат выходного файла

Выведите n целых чисел — порядок солдат в конечном положении после выполнения всех приказов.

Пример

movetofront.in	movetofront.out
6 3 2 4 3 5 2 2	1 4 5 2 3 6

```
void movetofront(int l, int r){
    Node *greater, *equal, *less;
    split(root, l, less, greater);
    split(greater, r - l + 1, equal, greater);
    root = merge(merge(equal, less), greater);
}
```

Новые операции (должны выполняться одновременно):

1. Прибавление на отрезке [l; r] числа x
2. Переворот значений на отрезке [l; r]
3. Циклический сдвиг отрезка [l; r] на x. Пример:
   [1, 2, 3, 4, 5], l = 1, r = 3, x = 2 -> [1, 3, 4, 2, 5]

4. Найти минимум на отрезке [l; r]

Новые операции (должны выполняться одновременно):

1. Прибавление на отрезке [l; r] числа x
2. Переворот значений на отрезке [l; r]
3. Циклический сдвиг отрезка [l; r] на x. Пример:
   [1, 2, 3, 4, 5], l = 1, r = 3, x = 2 -> [1, 3, 4, 2, 5]

4. Найти минимум на отрезке [l; r]

```
void revolve(int l, int r, int x){
    Node *greater, *equal, *less;
    split(root, l, less, greater);
    split(greater, r - l + 1, equal, greater);
    int len = get_size(equal);
    x %= len;
    // переставляем x последних элементов в начало
    Node *left, *right;
    split(equal, len - x, left, right);
    equal = merge(right, left);
    root = merge(merge(less, equal), greater);
}
```

Новые операции (должны выполняться одновременно):

1. Прибавление на отрезке [l; r] числа x
2. Переворот значений на отрезке [l; r]
3. Найти минимум на отрезке [l; r]

```
struct Node {
    int priority, size, value, min_value;
    Node *l = nullptr, *r = nullptr;
    Node (int value): value(value), priority(generator()), size(1), min_value(value) {}
} *root = nullptr;
```

Новые операции (должны выполняться одновременно):

1. Прибавление на отрезке [l; r] числа x
2. Переворот значений на отрезке [l; r]
3. Найти минимум на отрезке [l; r]

```
struct Node {
    int priority, size, value, min_value;
    Node *l = nullptr, *r = nullptr;
    Node (int value): value(value), priority(generator()), size(1), min_value(value) {}
} *root = nullptr;

static int get_min_value(Node *n){
    return n ? n -> min_value : INF;
}

static void update(Node *&n){
    if (n){
        n -> size = get_size(n -> l) + 1 + get_size(n -> r);
        n -> min_value = min(get_min_value(n -> l), min(n -> value, get_min_value(n -> r)));
    }
}
```

Новые операции (должны выполняться одновременно):

1. Прибавление на отрезке [l; r] числа x
2. Переворот значений на отрезке [l; r]
3. Найти минимум на отрезке [l; r]

```
struct Node {
    int priority, size, value, min_value;
    Node *l = nullptr, *r = nullptr;
    Node (int value): value(value), priority(generator()), size(1), min_value(value) {}
} *root = nullptr;

static int get_min_value(Node *n){
    return n ? n -> min_value : INF;
}

static void update(Node *&n){
    if (n){
        n -> size = get_size(n -> l) + 1 + get_size(n -> r);
        n -> min_value = min(get_min_value(n -> l), min(n -> value, get_min_value(n -> r)));
    }
}

int get_min(int l, int r){
    Node *greater, *equal, *less;
    split(root, l, less, greater);
    split(greater, r - l + 1, equal, greater);
    int result = get_min_value(equal);
    root = merge(merge(less, equal), greater);
    return result;
}
```

Новые операции (должны выполняться одновременно):

1. Прибавление на отрезке [l; r] числа x
2. Переворот значений на отрезке [l; r]
3. Найти минимум на отрезке [l; r]

```
struct Node {
    int priority, size, value, min_value;
    Node *l = nullptr, *r = nullptr;
    Node (int value): value(value), priority(generator()), size(1), min_value(value) {}
} *root = nullptr;

static int get_min_value(Node *n){
    return n ? n -> min_value : INF;
}

static void update(Node *&n){
    if (n){
        n -> size = get_size(n -> l) + 1 + get_size(n -> r);
        n -> min_value = min(get_min_value(n -> l), min(n -> value, get_min_value(n -> r)));
    }
}

int get_min(int l, int r){
    Node *greater, *equal, *less;
    split(root, l, less, greater);
    split(greater, r - l + 1, equal, greater);
    int result = get_min_value(equal);
    root = merge(merge(less, equal), greater);
    return result;
}
```

Новые операции (должны выполняться одновременно):

1. Прибавление на отрезке [l; r] числа x
2. Переворот значений на отрезке [l; r]
3. Найти минимум на отрезке [l; r]

```
struct Node {
    int priority, size, value, min_value, add = 0;
    Node *l = nullptr, *r = nullptr;
    Node (int value): value(value), priority(generator()), size(1), min_value(value) {}
} *root = nullptr;
```

Новые операции (должны выполняться одновременно):

1. Прибавление на отрезке [l; r] числа x
2. Переворот значений на отрезке [l; r]
3. Найти минимум на отрезке [l; r]

```
struct Node {
    int priority, size, value, min_value, add = 0;
    Node *l = nullptr, *r = nullptr;
    Node (int value): value(value), priority(generator()), size(1), min_value(value) {}
} *root = nullptr;
```

```
static int get_min_value(Node *n){
    return n ? n -> min_value + n -> add : INF;
}

static void push(Node *n){
    if (n){
        if (n -> add){
            n -> value += n -> add;
            n -> min_value += n -> add;
            if (n -> l){
                n -> l -> add += n -> add;
            }
            if (n -> r){
                n -> r -> add += n -> add;
            }
            n -> add = 0;
        }
    }
}
```

```
static Node *merge(Node *a, Node *b){
    push(a);
    push(b);
    if (!a || !b){
        return a ? a : b;
    }
    if (a->priority > b->priority){
        a->r = merge(a->r, b);
        update(a);
        return a;
    }
    else {
        b->l = merge(a, b->l);
        update(b);
        return b;
    }
}
```

Новые операции (должны выполняться одновременно):

1. Прибавление на отрезке [l; r] числа x
2. Переворот значений на отрезке [l; r]
3. Найти минимум на отрезке [l; r]

```
struct Node {
    int priority, size, value, min_value, add = 0;
    Node *l = nullptr, *r = nullptr;
    Node (int value): value(value), priority(generator()), size(1), min_value(value) {}
} *root = nullptr;
```

```
static void split(Node *n, int k, Node *&a, Node *&b){
    push(n);
    if (!n){
        a = b = nullptr;
        return ;
    }
    if (get_size(n -> l) < k){
        split(n->r, k - get_size(n -> l) - 1, n->r, b);
        a = n;
    }
    else {
        split(n->l, k, a, n->l);
        b = n;
    }
    update(a);
    update(b);
}
```

```
void range_add(int l, int r, int value){
    Node *greater, *equal, *less;
    split(root, l, less, greater);
    split(greater, r - l + 1, equal, greater);
    equal -> add += value;
    root = merge(merge(less, equal), greater);
}
```

Новые операции (должны выполняться одновременно):

1. Переворот значений на отрезке [l; r]

```
struct Node {
    int priority, size, value, min_value, add = 0, rev = 0;
    Node *l = nullptr, *r = nullptr;
    Node (int value): value(value), priority(generator()), size(1), min_value(value) {}
} *root = nullptr;
```

Новые операции (должны выполняться одновременно):

1. Переворот значений на отрезке [l; r]

```
struct Node {
    int priority, size, value, min_value, add = 0, rev = 0;
    Node *l = nullptr, *r = nullptr;
    Node (int value): value(value), priority(generator()), size(1), min_value(value) {}
} *root = nullptr;
```

```
static void push(Node *n){
    if (n){
        if (n -> add){
            n -> value += n -> add;
            n -> min_value += n -> add;
            if (n -> l){
                n -> l -> add += n -> add;
            }
            if (n -> r){
                n -> r -> add += n -> add;
            }
            n -> add = 0;
        }
        if (n -> rev){
            swap(n -> l, n -> r);
            n -> l -> rev ^= 1;
            n -> r -> rev ^= 1;
            n -> rev = 0;
        }
    }
}
```

```
void reverse(int l, int r){
    Node *greater, *equal, *less;
    split(root, l, less, greater);
    split(greater, r - l + 1, equal, greater);
    equal -> rev ^= 1;
    root = merge(merge(less, equal), greater);
}
```