```
bool isUnival(node* tree){
    switch(typeNode(tree)){
        case BOTH:
            if((tree->data == tree->lc->data)&&(tree->data == tree->rc->data)){
                return isUnival(tree->lc) && isUnival(tree->rc);
            }
            else{
                return false;
            }
            break;
        case LEFT:
            if(tree->data == tree->lc->data){
                return isUnival(tree->lc);
            }
            else{
                return false;
            }
            break;
        case RIGHT:
            if(tree->data == tree->rc->data){
                return isUnival(tree->rc);
            }
            else{
                return false;
            }
            break;
        case NONE:
            return true;
        default:
            printf("type failed");
    }
}
```

(Fig 1)

```
void countUnival(node* tree,int* count){
    if(tree->lc) countUnival(tree->lc,count);
    if(isUnival(tree)) ++(*count);
    if(tree->rc) countUnival(tree->rc,count);
}
```

(Fig 2)

```
node* findMin(node* tree){
    if(tree->lc) return findMin(tree->lc);
    return tree;
}
```

(Fig 1)

```
node* inorderSuccessor(node* tree,node* cur){
    node* succ = cur;
    if(cur->rc){
        succ = cur->rc;
        while(succ->lc) succ = succ->lc;
        return succ;
    }
    int temp = succ->data;
    while(typeChild(tree,succ->data) != HEAD){
        succ = searchParent(tree,succ->data,tree);
        if(succ->data > temp) return succ;
    }
    return searchNode(tree,temp);
}
```

(Fig 2)

```
node *searchNode(node* tree,int val){
    if(tree->data == val) return tree;
    if(tree->lc) if(val < tree->data) return searchNode(tree->lc,val);
    if(tree->rc) if(val > tree->data) return searchNode(tree->rc,val);
    return emptyNode();
}

node *searchParent(node* tree,int val,node* parent){
    if(tree->data == val) return parent;
    if(tree->lc) if(val < tree->data) return searchParent(tree->lc,val,tree);
    if(tree->rc) if(val > tree->data) return searchParent(tree->rc,val,tree);
    return emptyNode();
}
```

(Fig 3)

```
node *kthSmallest(node* tree,int k){
    node* smallest = findMin(tree);
    node* kthSmallestNode = smallest;
    for(int i = 0;i < k-1;i++){
        kthSmallestNode = inorderSuccessor(tree,kthSmallestNode);
    }
    return kthSmallestNode;
}
```

(Fig 4)

```c
int power2(int a){
    int i = 1;
    int j = 1;
    for(i;i<=a;i++){
        j *= 2;
    }
    return j;
}

int heightTree(node* tree){
    if (!tree) {
        return 0;
    }

    int leftHeight = heightTree(tree->lc);
    int rightHeight = heightTree(tree->rc);

    return 1 + max(leftHeight, rightHeight);
}
```

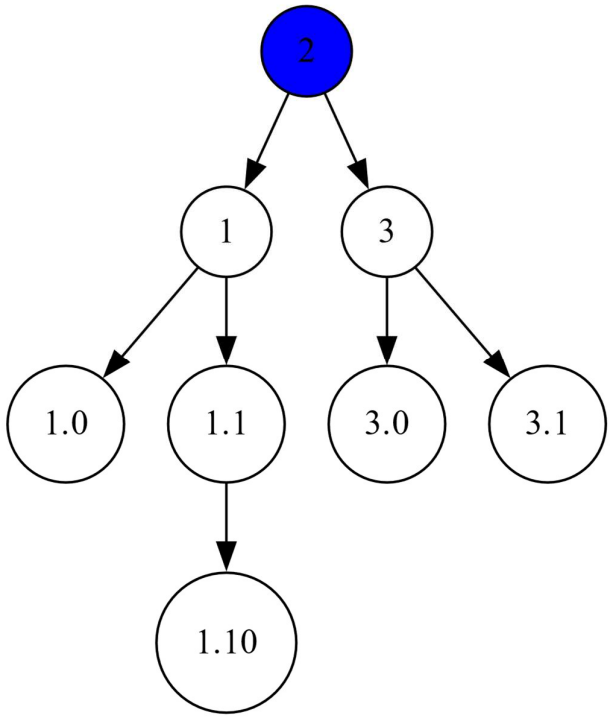
(Fig 1)

```c
void addtoArray(node* tree,int* arr,int i){
    arr[i] = tree->data;
    if(tree->lc) addtoArray(tree->lc,arr,2*i+1);
    if(tree->rc) addtoArray(tree->rc,arr,2*i+2);
}

void serializeBST(node* tree,char* s){
    int size = power2(heightTree(tree));
    int* array = (int*)calloc(size,sizeof(int));
    int i = 0;
    addtoArray(tree,array,i);

    int offset = 0;
    for (i = 0; i < size; i++) {
        offset += sprintf(s + offset, "%d,", array[i]);
    }
}
```

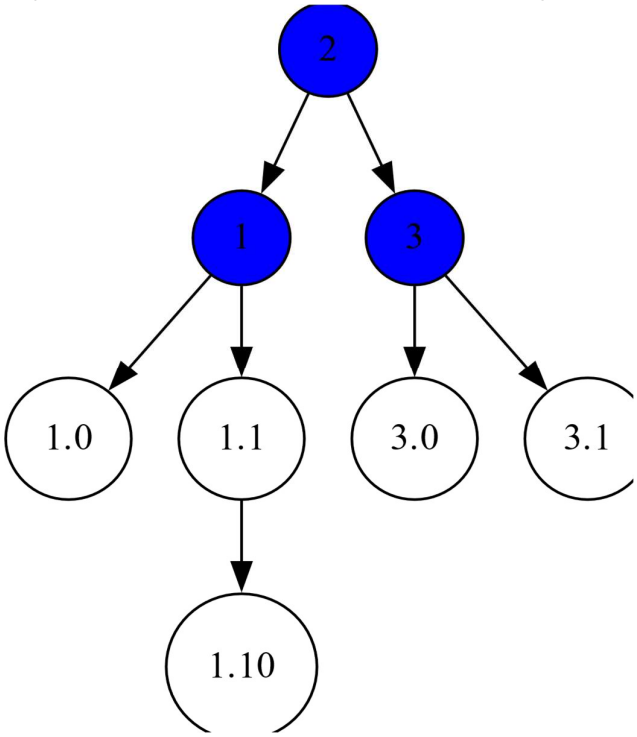
(Fig 2)

```
node* deserializeBST(char* exp){
    int i = 0;
    node* tree;
    if(exp[i] > 47 && exp[i] < 58){
        int num = 0;
        while(exp[i] != ','){
            num *= 10;
            num += exp[i]-48;
            i++;
        }
        tree = createNode(num);
    }

    while(exp[i] != '\0'){
        if(exp[i] > 47 && exp[i] < 58){
            int num = 0;
            while(exp[i] != ','){
                num *= 10;
                num += exp[i]-48;
                i++;
            }
            if(num) insertElement(tree,num);
        }
        else i++;
    }

    return tree;
}
```

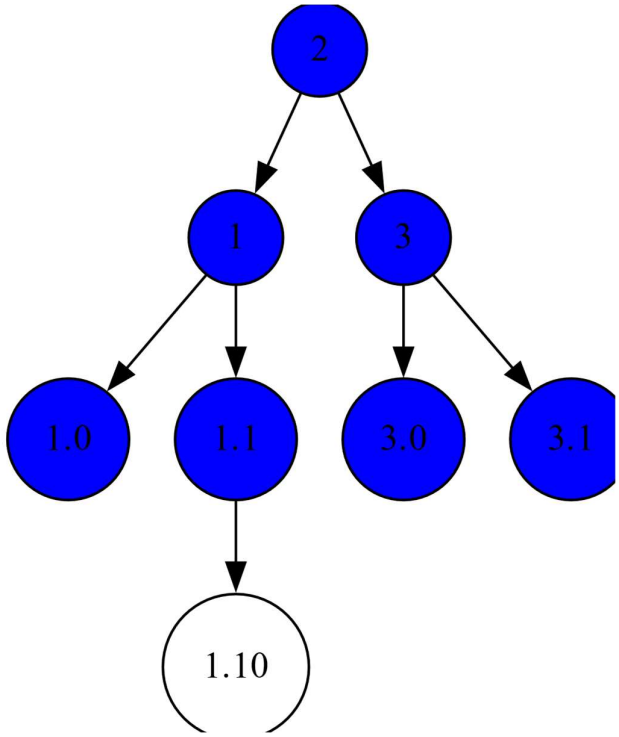
(Fig 3)



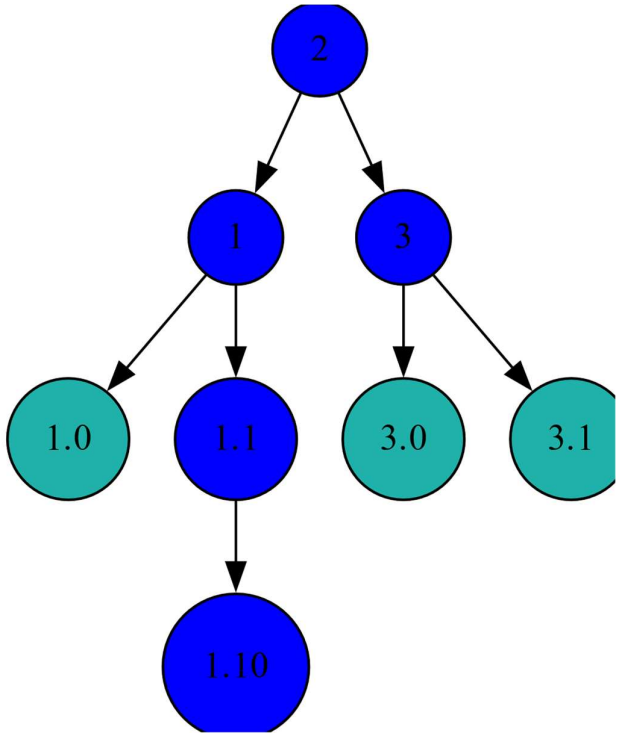
Only consider the whole number sections of nodes



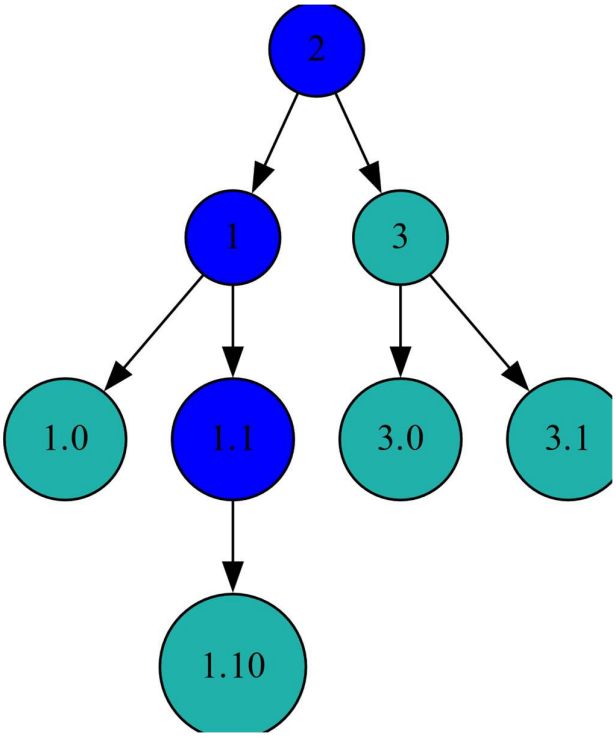
The blue nodes are currently running the isUnival function



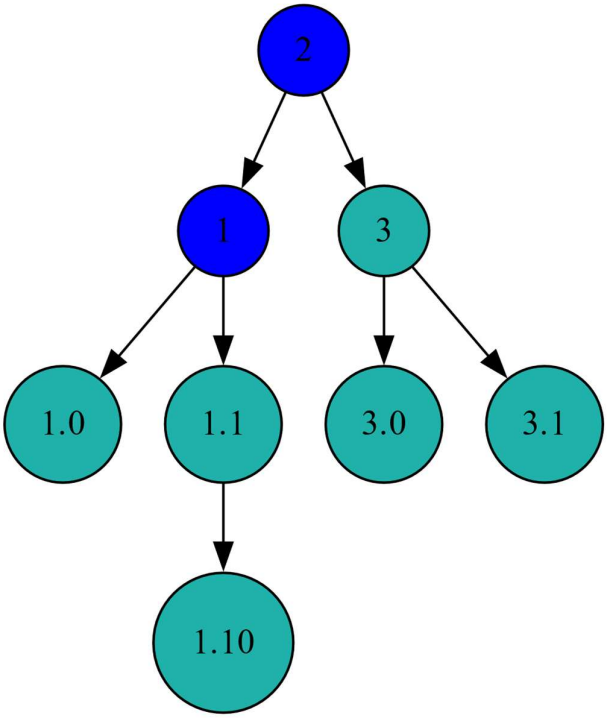
The isUnival function calls the function on every subtree



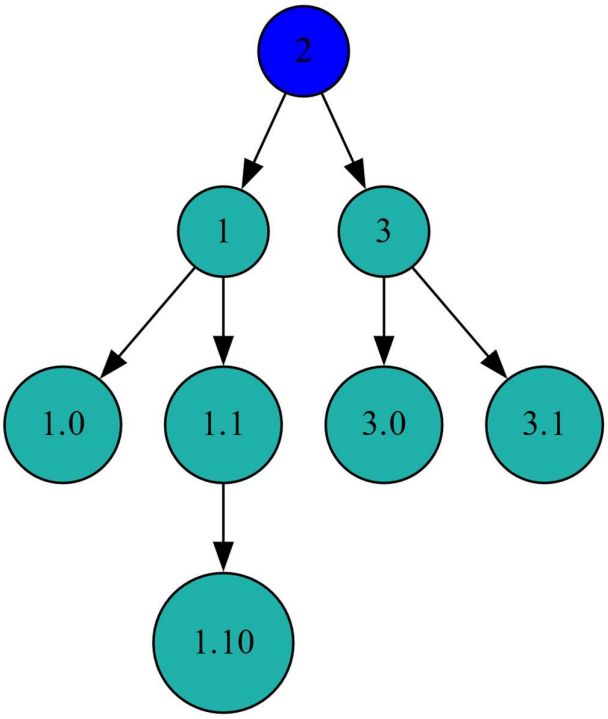
The leaf nodes return a true value



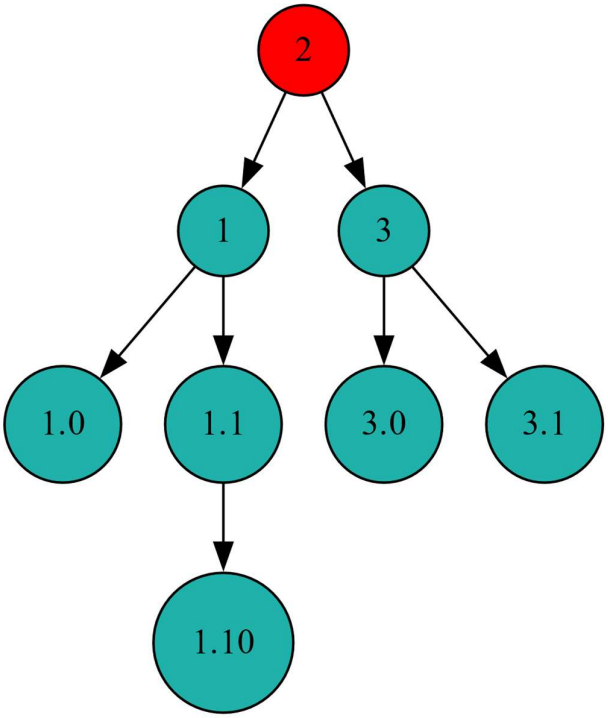
The 3 subtree returns true when its children both return true and are equal to the parent



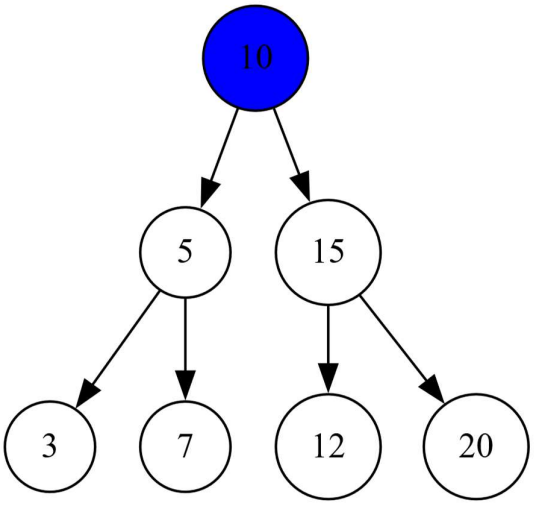
The 1.1 subtree returns true when its only child returns true



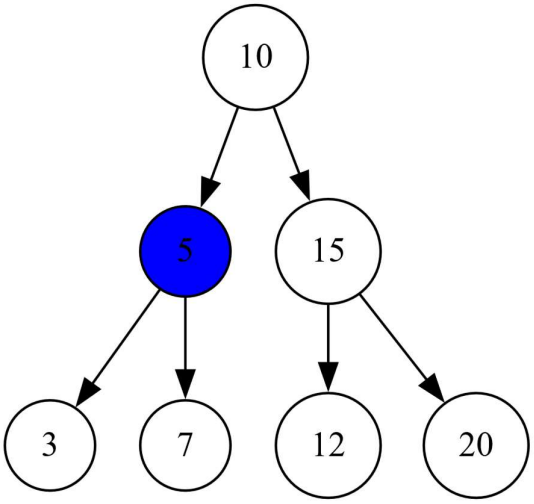
Each node that returns true increments the counter variable



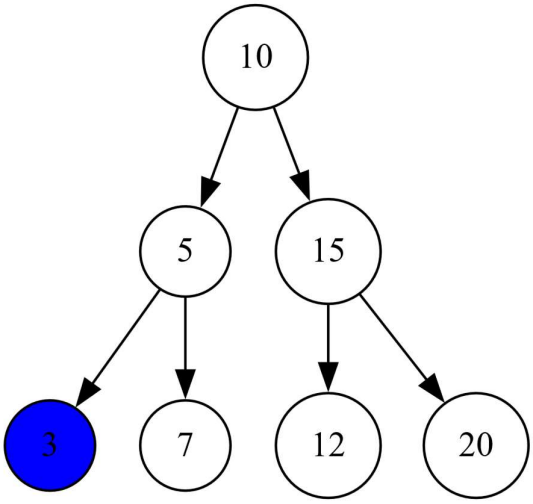
The root node returns false as its not equal to its children, though they returned true



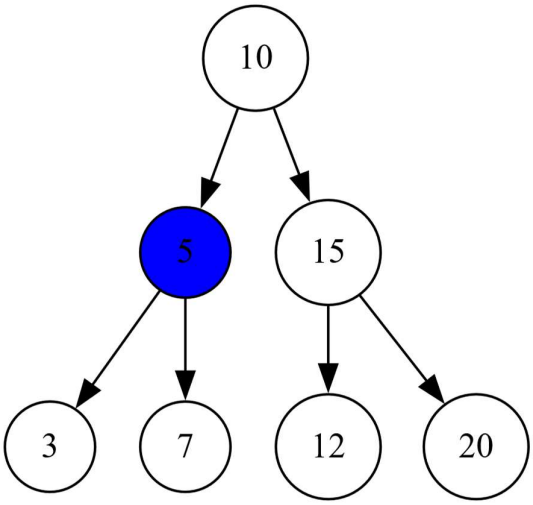
The coloured nodes are running the findMin function



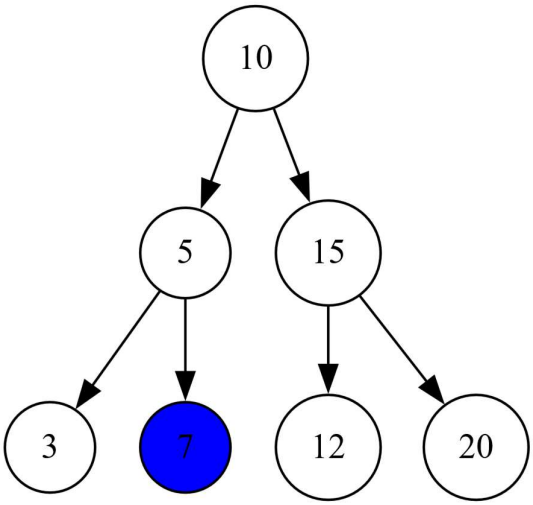
This function simply moves to the left



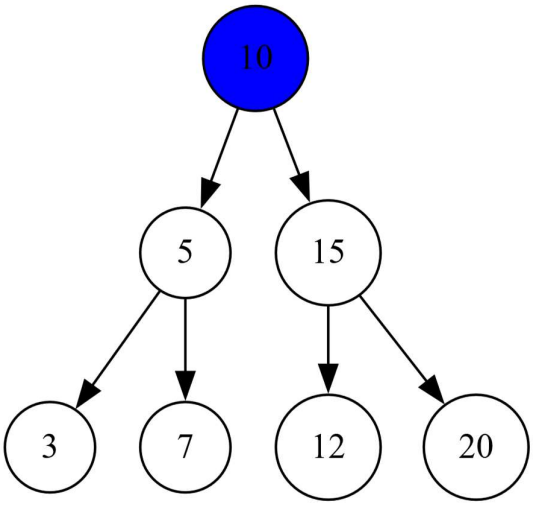
When it reaches a leaf node, it returns it as the smallest node

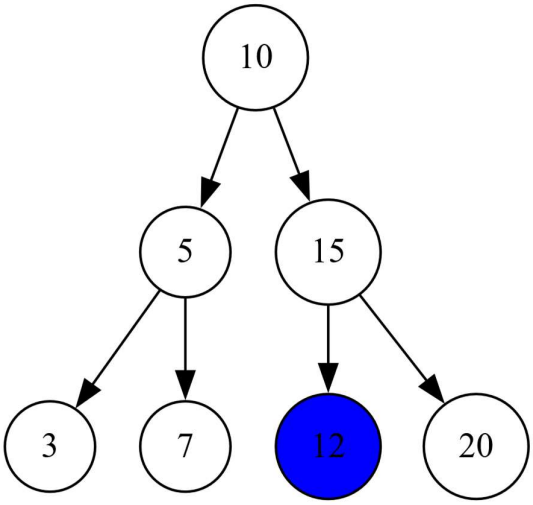


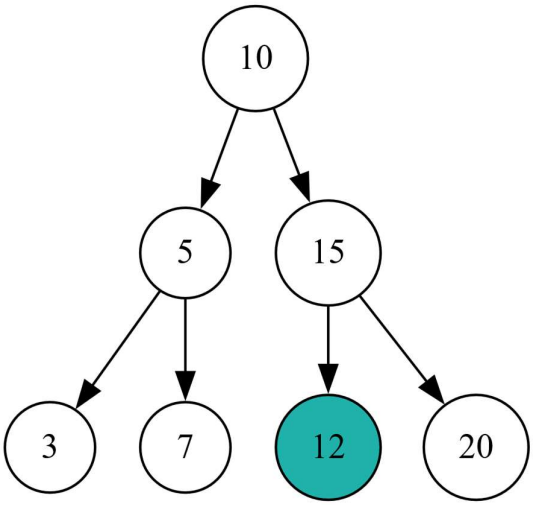
Here the coloured nodes are the inorder successors of the node in the previous image



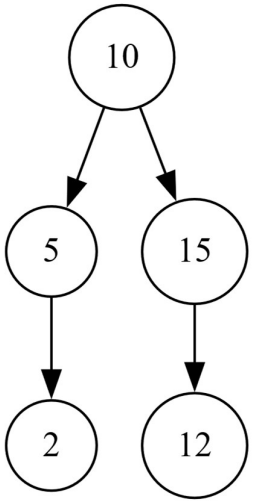
The inorderSuccessor function is called k-1 times(4) and returns that node



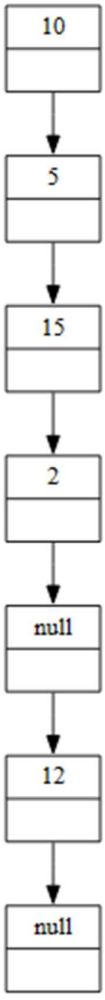




This node is returned

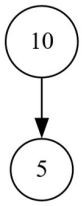


This is the original tree

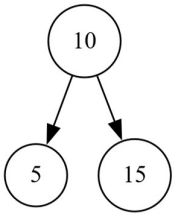


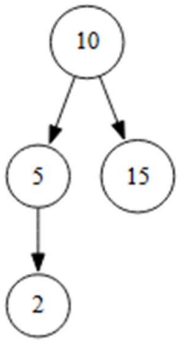
This is the serialized version with null elements

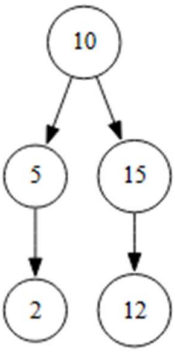
The deserializeBST function first creates a node with the first element



It then iterates through the elements and inserts them into the tree







This is the tree returned by the deserializeBST function