```
Triple isOrderedBTreeAux(btree bt){
    Triple t;
    if(bt->left == NULL && bt->right == NULL) {     // isLeaf(bt)
        t.isOrdered = true;
        t.min = t.max = bt->key;
    }else if(bt->right == NULL) {       // bt->left != NULL
        Triple tleft = isOrderedBTreeAux(bt->left);
        t.isOrdered = tleft.isOrdered && bt->key > tleft.max;
        t.min = tleft.min;
        t.max = bt->key;
    }else if(bt->left == NULL) {        // bt->right != NULL
        Triple tright = isOrderedBTreeAux(bt->right);
        t.isOrdered = tright.isOrdered && bt->key < tright.min;
        t.min = bt->key;
        t.max = tright.max;
    }else{      // bt->left != NULL && bt->right != NULL
        Triple tleft = isOrderedBTreeAux(bt->left);
        Triple tright = isOrderedBTreeAux(bt->right);
        t.isOrdered = tleft.isOrdered && tright.isOrdered &&
        bt->key > tleft.max && bt->key < tright.min;
        t.min = tleft.min;
        t.max = tright.max;
    }
    return t;
}
```

```
list DescList_aux(btree bt, list l) {
    if(bt == NULL)return l;
    else{
        l = DescList_aux(bt->left, l);
        l = Cons(bt->key, l);
        return DescList_aux(bt->right, l);
    }
}
list DescList(btree bt) {return DescList_aux(bt, NULL);}
```

```
bool isOrdered(btree bt) {
    if(bt == NULL) return true;
    else{
        Triple t= isOrderedBTreeAux(bt);
        return t.isOrdered;
    }
}
```

```
list CrescList_aux(btree bt, list l) {
    if(bt == NULL) return l;
    else{
        l = CrescList_aux(bt->right, l);
        l = Cons(bt->key, l);
        return CrescList_aux(bt->left, l);
    }
}
list CrescList(btree bt) {return CrescList_aux(bt, NULL);}
```

```
btree maxInBtree(btree bt) {
    while (bt->right != NULL) {bt=bt->right;}
    return bt;
}
btree minInBtree(btree bt) {
    if(bt->left == NULL) return bt;
    else return minInBtree(bt->left);
}

btree minInBtree(btree bt) {
    while (bt->left != NULL){bt = bt->left;}
    return bt;
}
```

```
btree rightAncestor(btree nd) {
    btree p = nd->parent;
    while (p != NULL && nd == p->right) {
        nd=p;
        p=nd->parent;
    }
    return p;
}
btree successor(btree nd) {
    if(nd->right != NULL) return minInBtree(nd->right);
    else return rightAncestor(nd);
}
```

```
int cardinality(btree t) {
    if(t==NULL) return 0;
    int l, r;
    l= cardinality(t->left);
    r= cardinality(t->right);
    return l + r + 1;
}
```

```
int height(btree node) {
    if(node->left == NULL && node->right == NULL) return 0;
    else{
        int hl, hr = 0;
        if(node->left != NULL) hl = height(node->left);
        if(node->right != NULL) hr = height(node->right);
        return max(hl, hr) +1 ;
    }
}
```

```
btree insert(int k, btree bt) {
    if(bt==NULL) return ConsTree(k,NULL,NULL);
    else if(k == bt->key) return bt;
    else if(k > bt->key){
        bt->right = insert(k, bt->right);
        return bt;
    }else{ // k < bt->key
        bt->left =insert(k , bt->left);
        return bt;
    }
}
```

```c
void bTreeCrescente(btree bt){
    if(bt == NULL) return;
    bTreeCrescente(bt->left);
    printf("%d ", bt->key);
    bTreeCrescente(bt->right);
}
void bTreeDecrescente(btree bt){
    if(bt == NULL) return;
    bTreeDecrescente(bt->right);
    printf("%d ", bt->key);
    bTreeDecrescente(bt->left);
}
```

```c
btree antenatoComune(btree bt, int a, int b) {
    if(bt->key >= a && bt->key <= b ) return bt;
    else if(bt->key < a) return antenatoComune(bt->right, a, b);
    else if(bt->key > b) return antenatoComune(bt->left, a, b);
}
```