CS2700 : PROGRAMMING AND DATA STRUCTURES

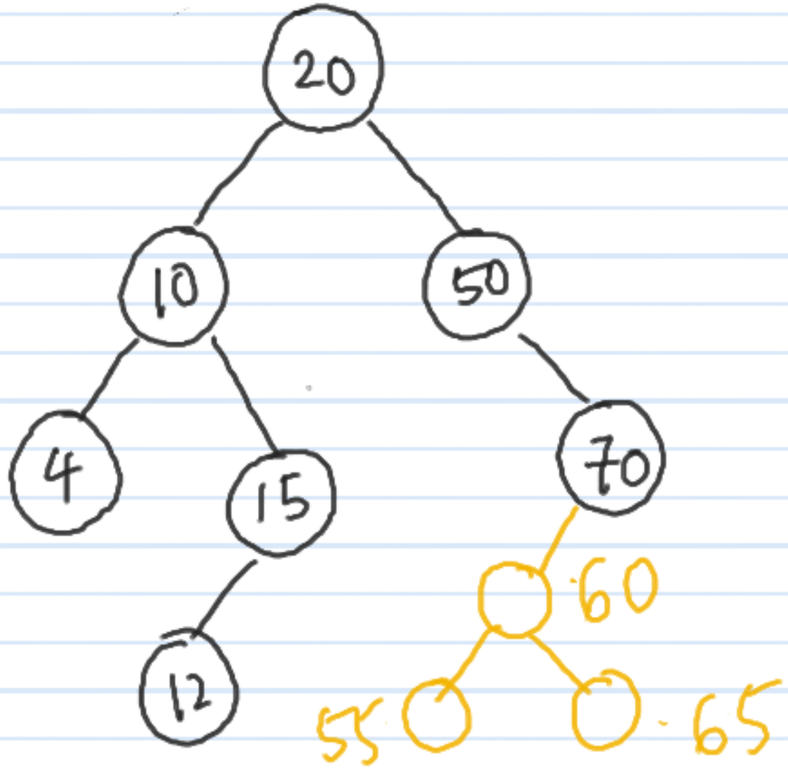
WEEK 8 : BINARY SEARCH TREES

GOALS :

- BST ADT
  - INSERT
  - DELETE
  - SEARCH

BST DEFINITION

RESTRICTED BINARY TREE



Assumption : all keys are unique

A non empty BST has

- a root with a key
- left subtree is a BST with all keys smaller than root key
- Right subtree is a BST with all keys > root key

BSTs vs ARRAYS and LINKED LISTs

STORING n key valued items

- search
- insert / delete

} Goal is to provide efficient operations

SORTED ARRAY : search is $O(\log n)$

: insert and delete are $O(n)$

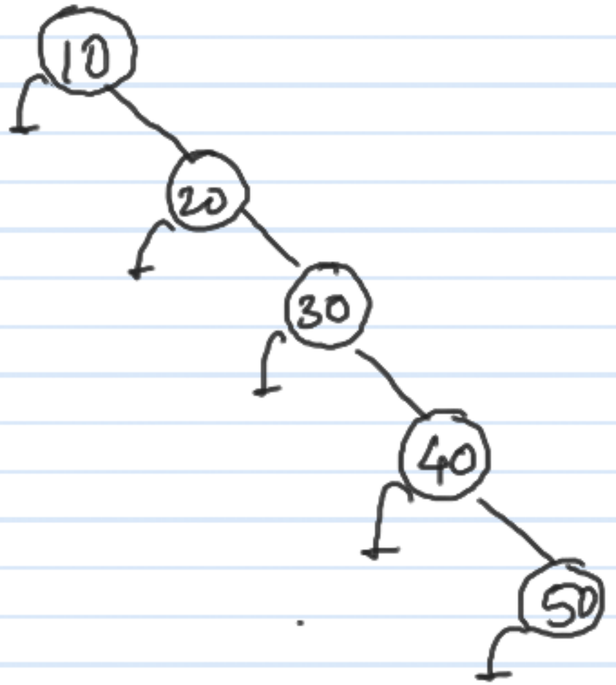
LINKED LISTS : insert $O(1)$
delete $O(n)$
search $O(n)$

BST : some more examples

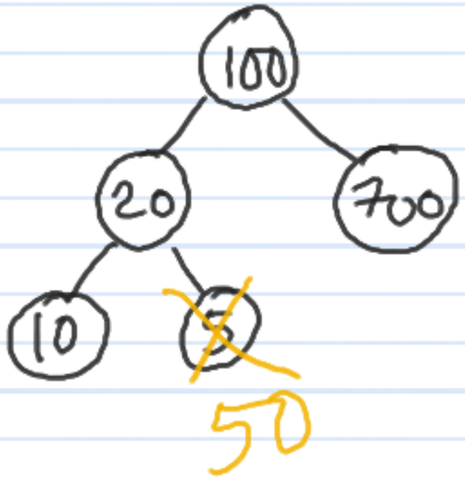
1

EMPTY TREE

SINGLE NODE



SKEWED TREE



ARBITRARY BST

```
struct TreeNode {
    Datatype data;
    struct TreeNode * left;
    struct TreeNode * right;
};
```

```
class BST {
private:
    struct TreeNode *root;

public:
    find
    findMax
    findMin
    insert
    delete
```

```
struct TreeNode * find ( DataType e) {
        return recfind (root , e);
}

struct TreeNode * recfind ( struct TreeNode * ptr, DataType e)
{                                  note use of short circuiting
    if (ptr==NULL || ptr→data ==e) return ptr;
    if (ptr→data >e) recfind (ptr→left , e);
    if (ptr→data <e) recfind ( ptr→right, e);

}
```

findMax

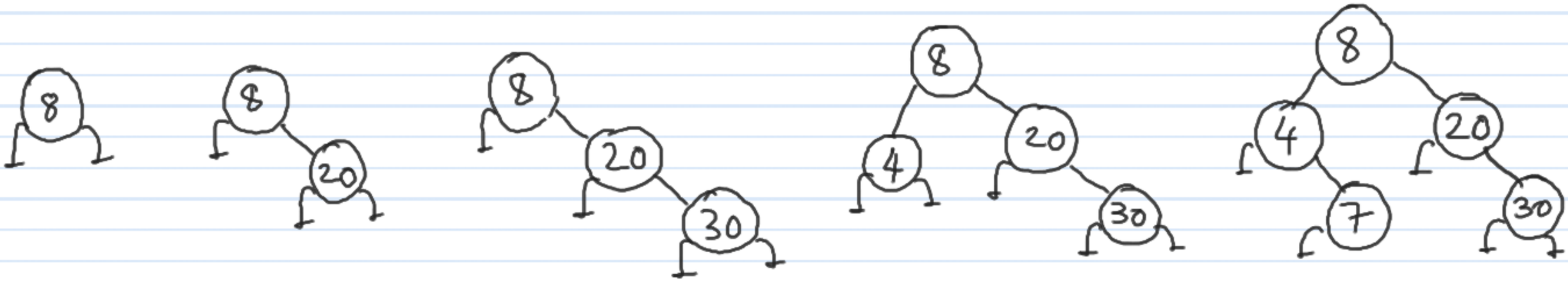
- keep traversing the tree towards the right till we encounter a node with an empty right child.
- the max node need not be a leaf node

findMin

- keep traversing the tree towards the left till we encounter a node with an empty left child.
- The min node need not be a leaf node.

Insert into BST

8, 20, 30, 4, 7

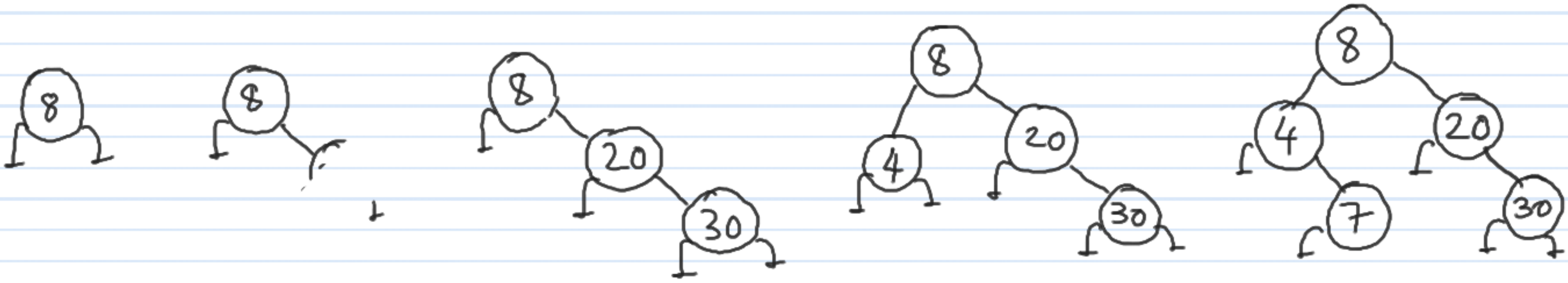


- A given sequence of inserts creates a unique tree
- The same tree can be created via different sequences

  8, 4, 20, 30, 7 OR 8, 20, 4, 7, 30
- Insert always creates a leaf node

Insert into BST

8, 20, 30, 4, 7

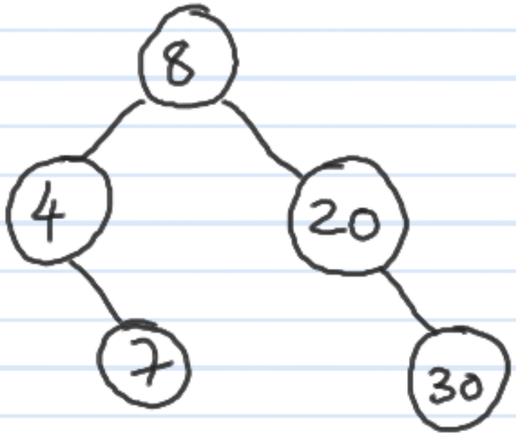


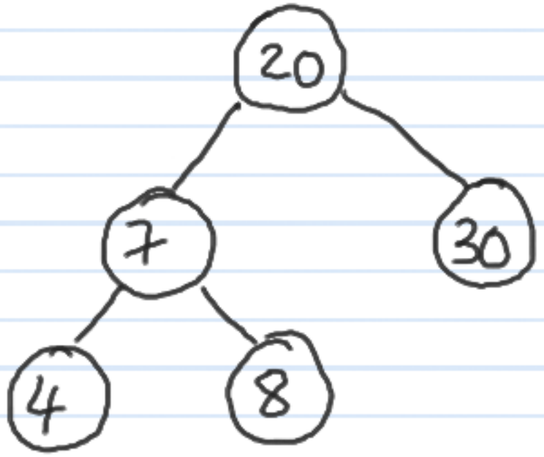
- Order of insertions determines the height of the tree

  4, 7, 8, 20, 30

- Complexity of operations is dependent on height of tree. (will be good to have low height)

TREE TRAVERSALS.





INORDER	4, 7, 8, 20, 30	4, 7, 8, 20, 30
PREORDER	8, 4, 7, 20, 30	20, 7, 4, 8, 30
POSTORDER	7, 4, 30, 20, 8	4, 8, 7, 30, 20
LEVEL ORDER	8, 4, 20, 7, 30	20, 7, 30, 4, 8

DELETION OF A NODE (BST)

- Delete (e) // assume unique values.

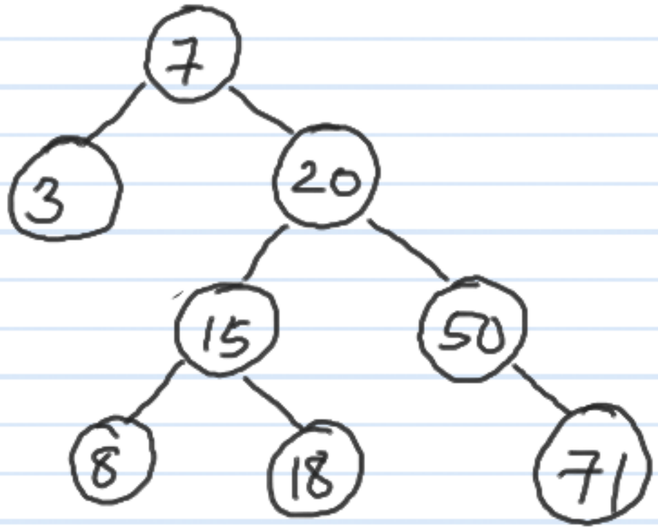
CASES:

(1) e is a leaf node

(2) e has one child

(3) e has 2 children

(4) e is a root.



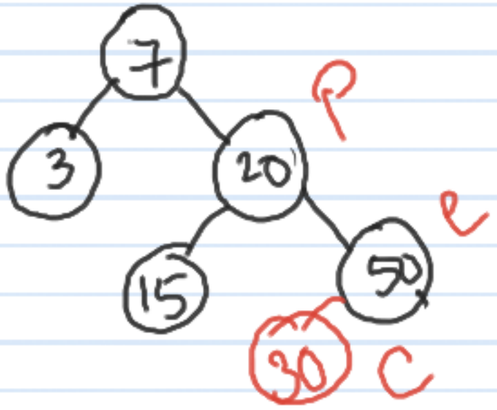
DELETION OF A NODE (BST)

- Delete (e) // assume unique values.

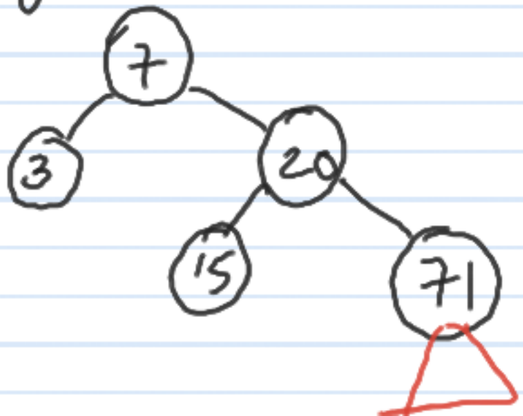
CASES:

(1) e is a leaf node : delete the node and adjust pointers of parent

(2) e has only 1 child c:



delete 50 ⟹

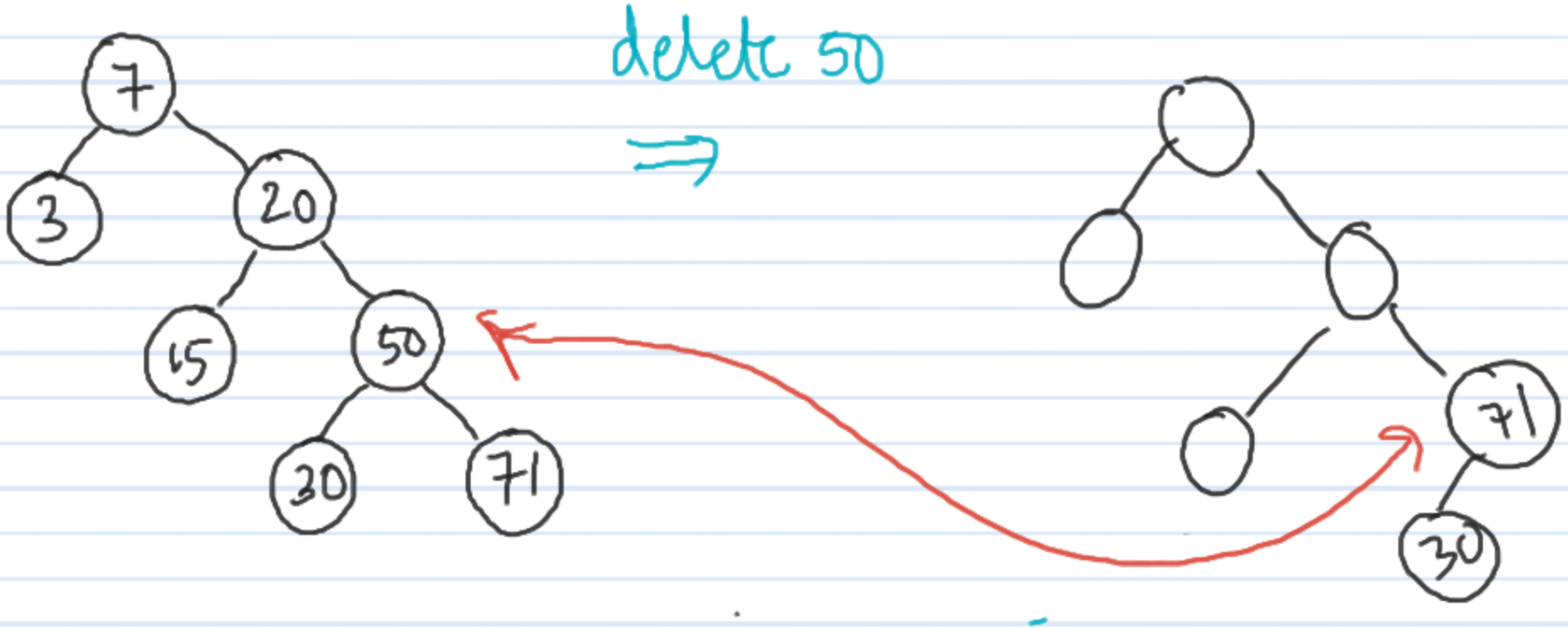


DELETION OF A NODE (BST)

- Delete (e) // assume unique values.

CASES:

(3) e has 2 children

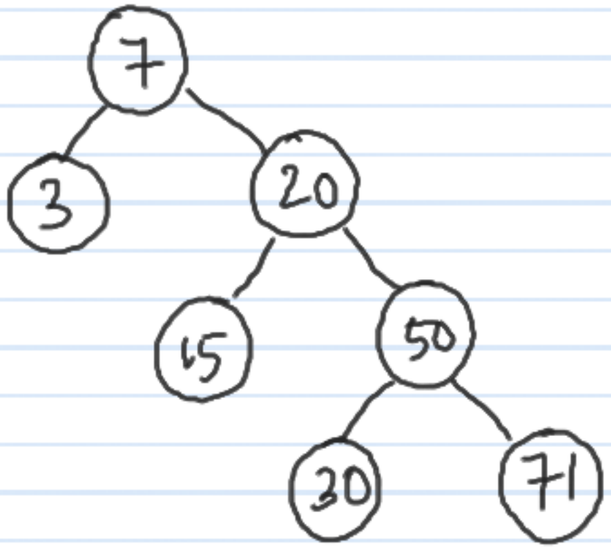


DELETION OF A NODE (BST)

- Delete (e) // assume unique values.

CASES:

(3) e has 2 children : reduce this complicated case to case 1 or case 2.



delete 20

⇒

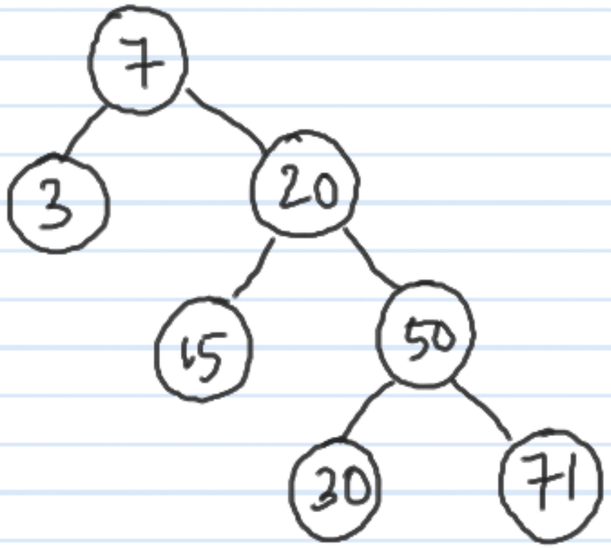
how??

DELETION OF A NODE (BST)

- Delete (e) // assume unique values.

CASES:

(3) e has 2 children : reduce this complicated case to case 1 or case 2.



delete 20

⇒

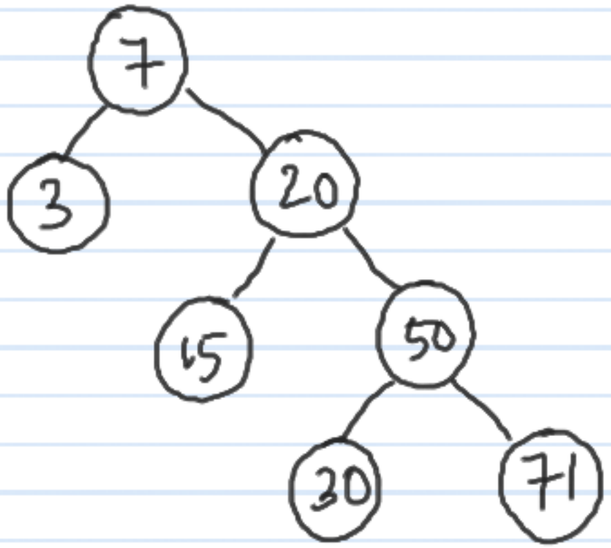
- copy the smallest value from the right subtree (RST) to the current node and delete the smallest value in (RST)

DELETION OF A NODE (BST)

- Delete (e) // assume unique values.

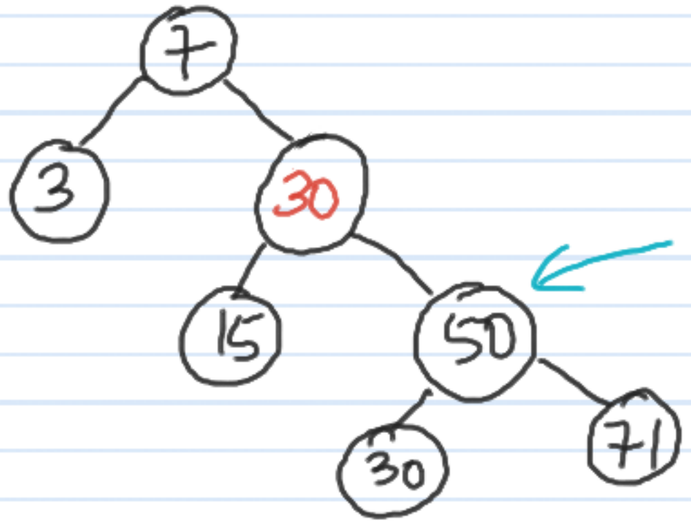
CASES:

(3) e has 2 children : reduce this complicated case to case 1 or case 2.



delete 20

⇒



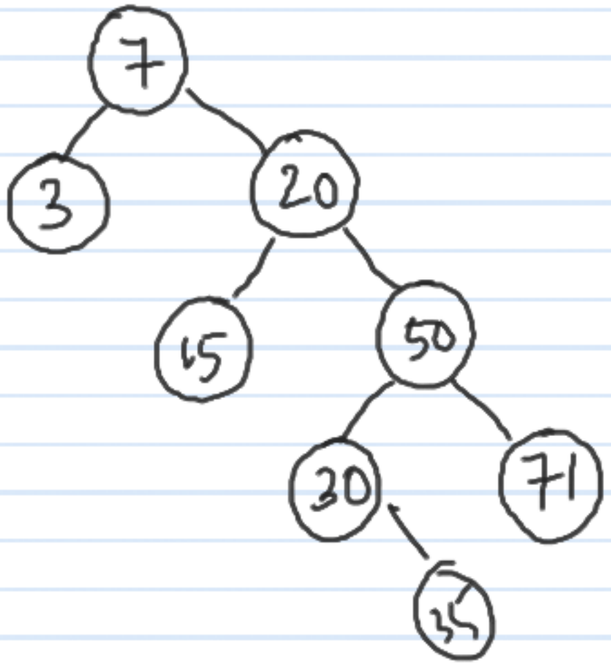
delete 30

DELETION OF A NODE (BST)

- Delete (e) // assume unique values.

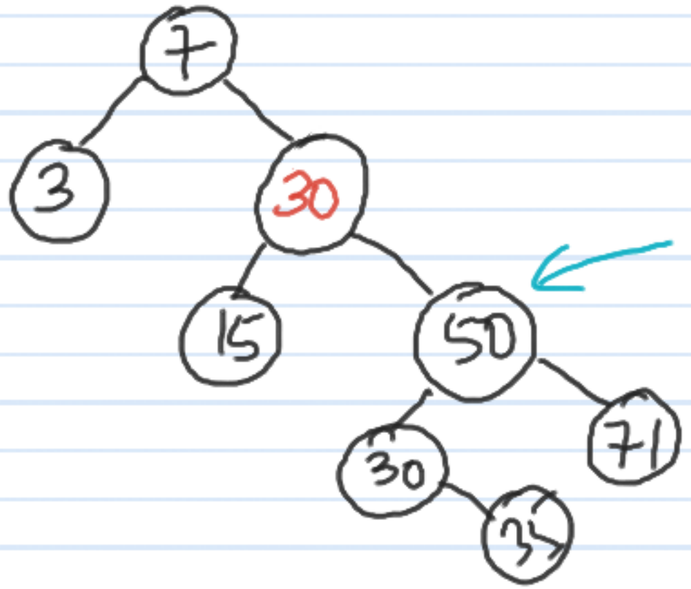
CASES:

(3) e has 2 children : reduce this complicated case to case 1 or case 2.



delete 20

⇒



delete 30

DELETION OF A NODE (BST)

- Delete (e) // assume unique values.

CASES:

(3) e has 2 children : copy the smallest value (e') from right subtree into the node and delete smallest value from RST.

- What is the property of the node e' ?
- Is there another way to do it ?

- complexity ?

insert 20, 7, 30, 35, 6

delete 35, 7.