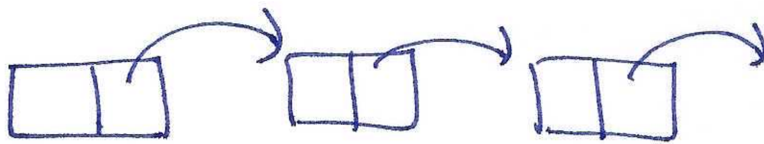
Dictionary ADT (Dictionary Data Structure)

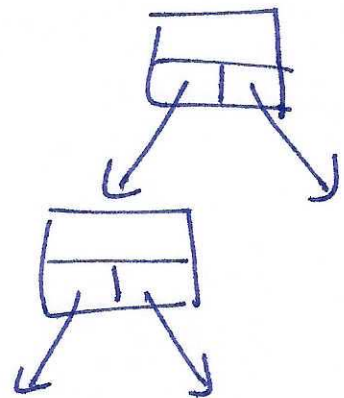
LIST [Insert, Delete, Find], FindMax, FindMin,

Pred (largest key less than a value)

Succ (smallest key more than a value)

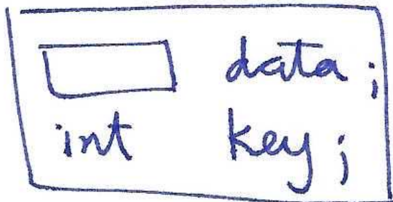
BST (Binary Search Tree)





class Node

{



Node* left; } children

Node* right; }

Node* parent; // optional

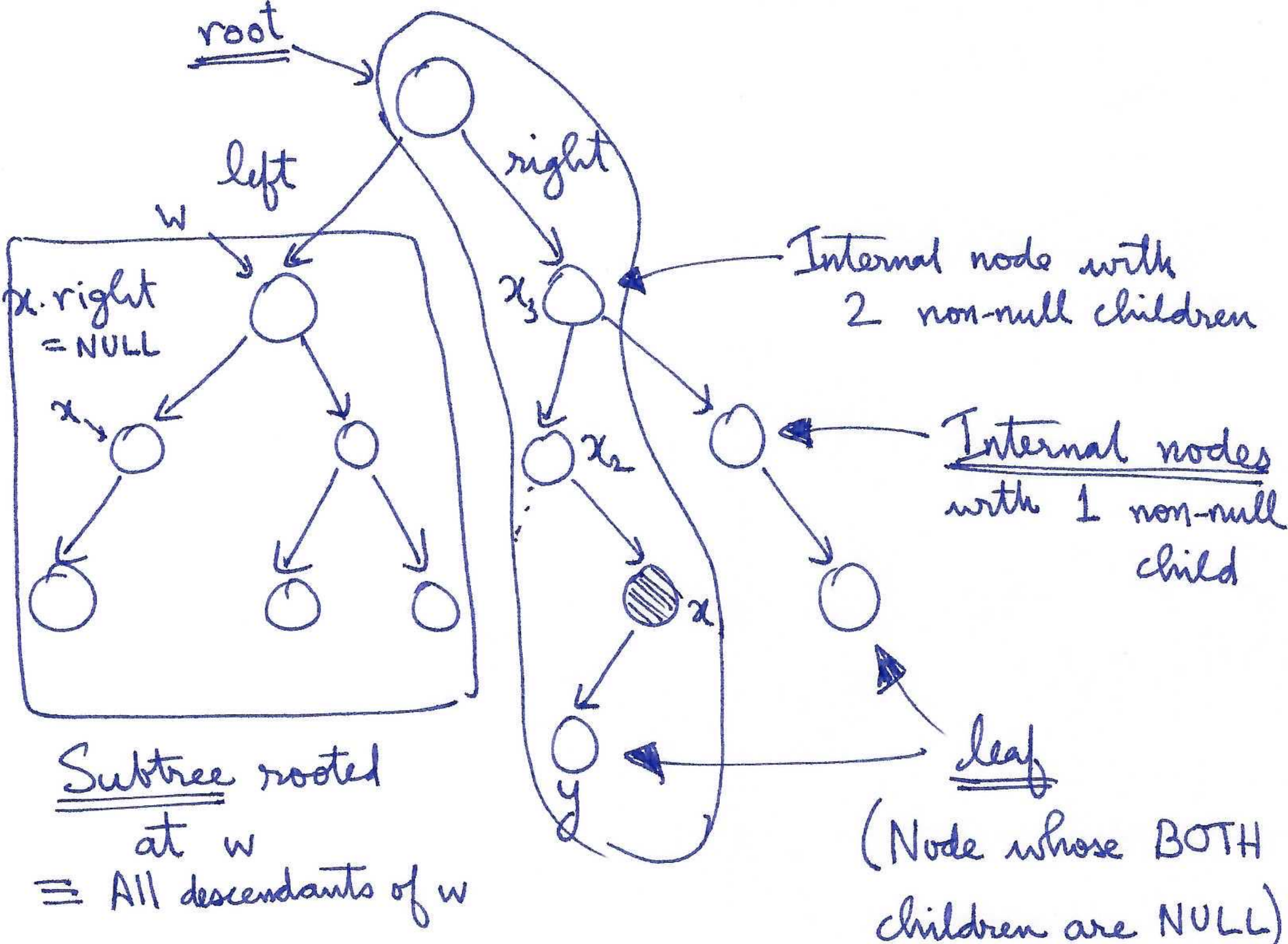
}

class BST

{

Node* root;

}



Ancestor x is an ancestor of y if:

(Base case) x is the parent of y (x is an ancestor of x)

(Induction/recursion) x is an ancestor of the parent of y.

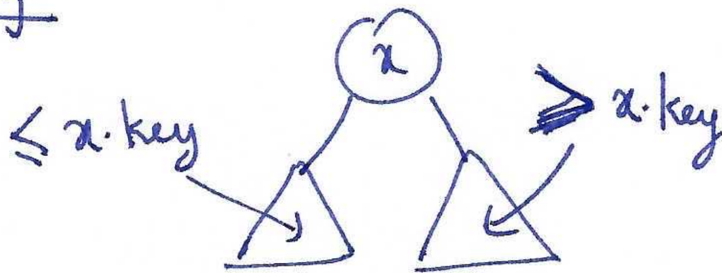
The ancestors form a path from y to the root

y, y→parent, y→parent→parent, . . .

Follow arrows backwards

Descendant: x is a descendant of y if y is an ancestor of x

BST property



For all nodes x:

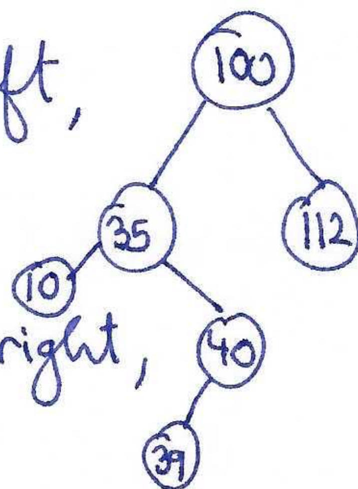
For all descendants y:

If y is descendant of $x \to left$,

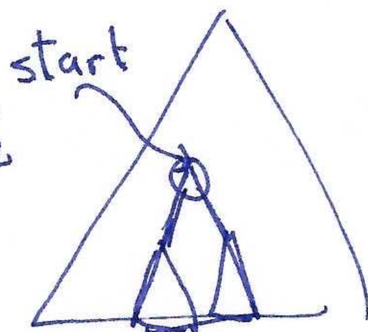
then $y.key \leq x.key$

If y is descendant of $x \to right$,

then $y.key > x.key$



Find (Node* start, int val) ← find single node with key = val in subtree rooted at start



1. If start is NULL, return NULL.
2. If start → key is val, return start.
3. If val < start → key
   return find(start → left, val)

   else return find(start → right, val)

Q. (R) return find(start → left, val)

(G) find(start → right, val)

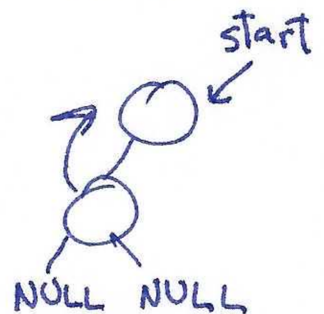
(B) find(start → parent, val)

Common corner case involves empty trees or deletion of root

insert(Node* start, Node* to_insert) to_insert.key

Insert to_insert in subtree rooted at start

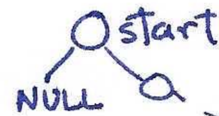
Suppose to_insert.key < start → key



If start → left == NULL

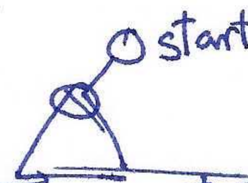
1. start → left = ~~new node~~ to_insert
2. ~~new node data = to_insert~~
3. ~~new node~~ to_insert → parent = start

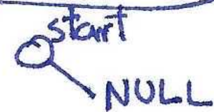
Base Case



else

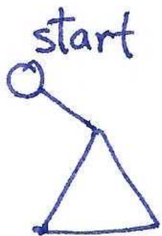
insert(start → left, to_insert)





If to_insert.key > start → key

switch all "left" to "right" in above code



Delete : Complicated (many cases)

deleteNode(Node* to_delete)

← ptr to node that needs to be deleted

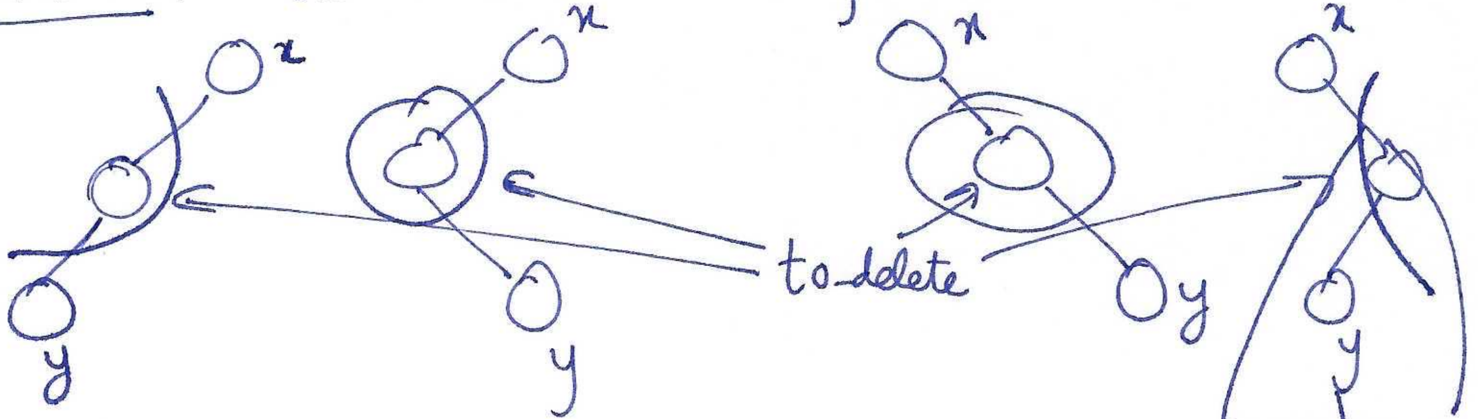
(If to_delete is the root, it's a special case. Exercise)

Case 1 : to_delete is a leaf $O(1)$

to_delete is either the left or right child of parent x.

Set appropriate child ptr of x to NULL.

Case 2 : to_delete has exactly 1 non-null child $O(1)$

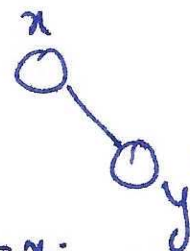


Splice : so y becomes child of x

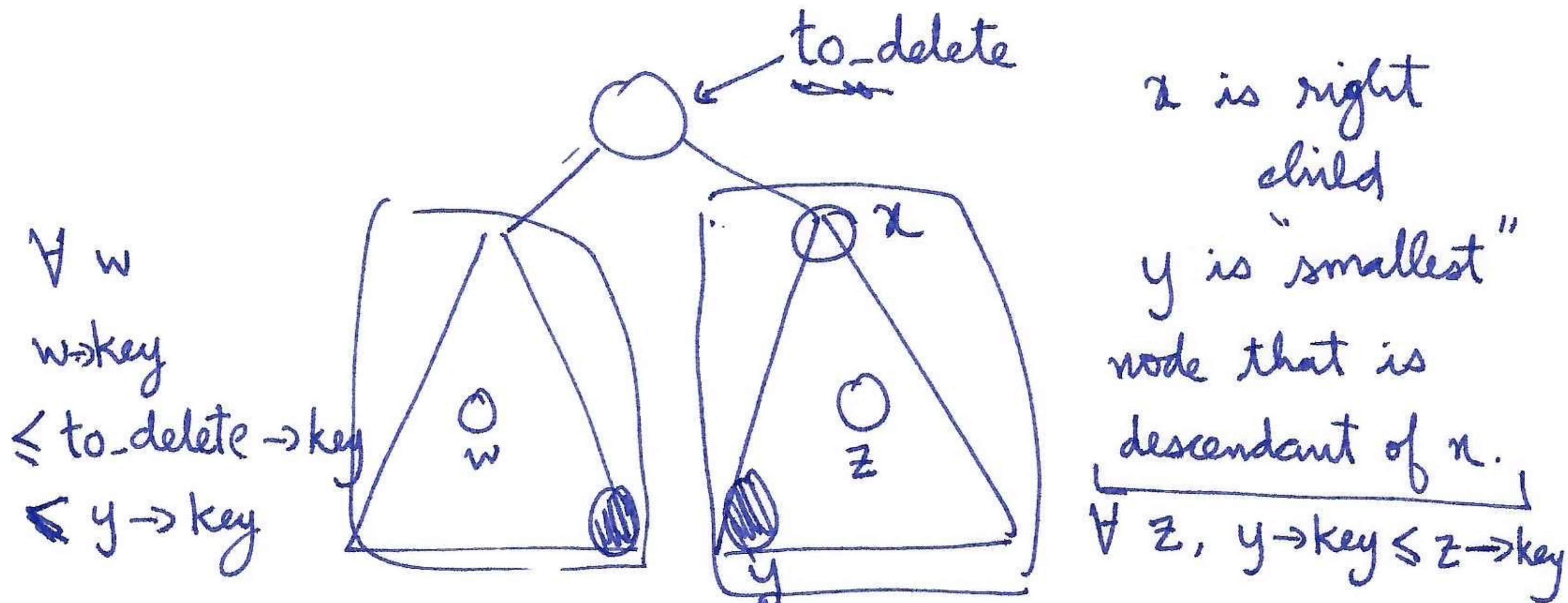
x = to_delete → parent;

y = to_delete → left;

x → ~~left~~ right = y; y → parent = x;



Case 3 : to_delete has 2 non-null children



Replace data (and key) in to_delete with data (and key) from another node. Delete the other node use Cases 1 and 2.

I want to replace to_delete→key

y is smallest node that is descendant of to_delete→right.

1) Starting from to_delete→right, keep going left until we hit NULL. Call that node y.
2) Swap content of to_delete with content of y.
3) deleteNode(y)

y has at most 1 non-null child. So deleteNode(y) is Case 1 or Case 2.