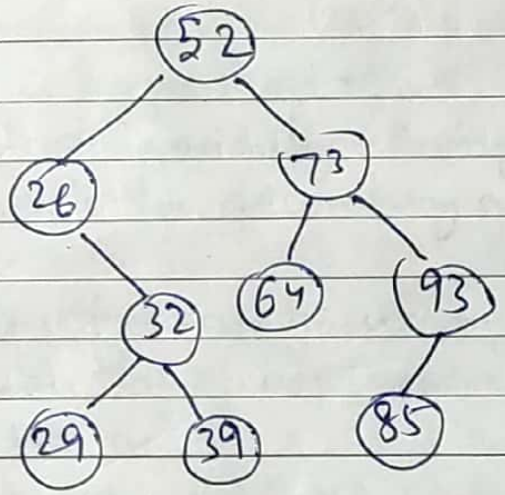
BINARY SEARCH TREE

Defn Binary search tree is a binary tree in which each node is ~~asso~~ associated with a key, at each node of the tree the following properties are true:-

i) the keys of the nodes in the left subtree are all less than the value of the key
ii) the keys of the nodes in the right subtree are all greater than the value of the key.

Ex:-

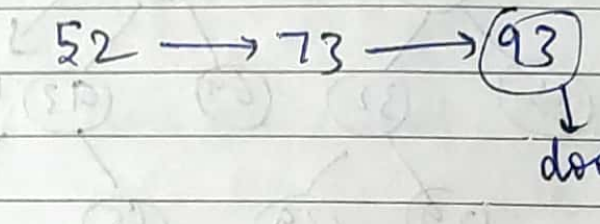


If we're given a subtree, what do we mean by given a subtree is that we're given the pointer to the root node; then find the ~~small~~ largest & the smallest no. respectively.

The claim is that if we start from root node & go on moving to the right child of each node until we reach a node which does not have a right child, then that node will contain the largest value

Similarly, for smallest we'll go on moving to the left child.

Ex:- in the previous BST
to ~~find~~ find largest



⇒ 93 is the largest value in BST

to find smallest

52 → 26 → does not have a left child

⇒ 26 is the smallest value in BST

Operations

'h' is height of the tree

1.) Search(x) ⟶ [O(h)]

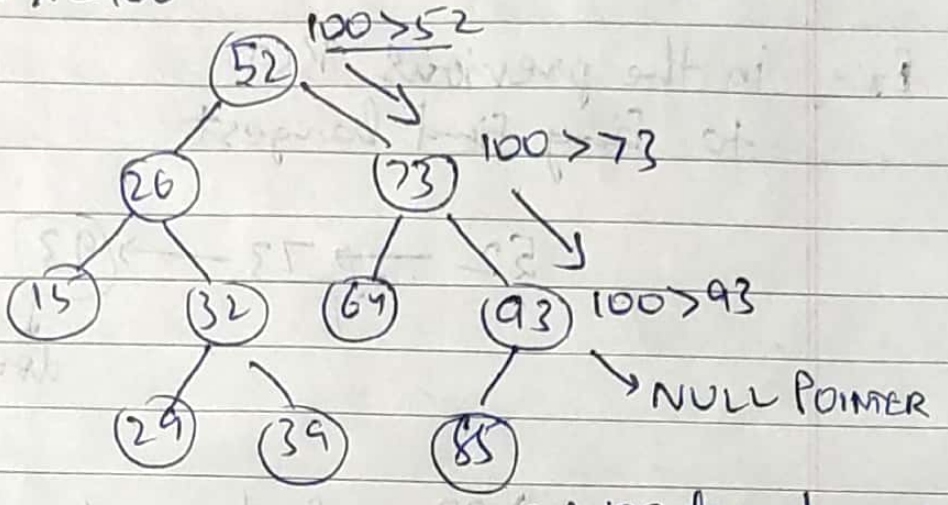
what 'search' will return depends on your need. It can return a simple boolean value or a pointer to that node.

Now, the way a BST is formed, if ~~the~~ X is greater than the key of a particular node then search in the right subtree of the ~~key~~ node; else search in the left subtree of the ~~key~~ node.

i.e. if X < node→key. < keys of right subtree
else X > node→key > keys of left subtree

This way of search will lead you to a NULL pointer if X doesn't exist in the BST.

Ex:- Suppose X = 100



∴ 100 doesn't exist

2.) INSERT(x) ⟶ [O(h)]

Search for x, until you find a NULL ptr. & insert x at that node

For ex:- in prev. ex. 100 will be inserted as the right child of 93.

3.) DELETE(x) ⟶ [O(h)]

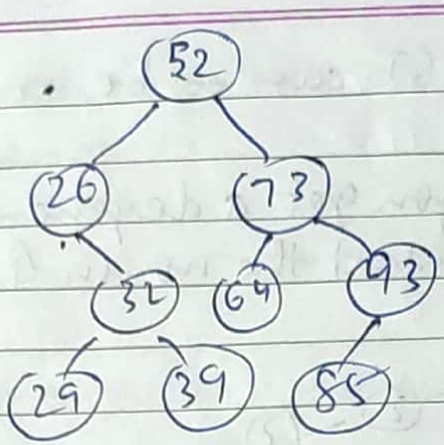
Case 0 :- x is in the leaf node ⟶ O(1)

→ Just simply delete it.

Case 1 :- X is in a node, which has a single child → O(1)

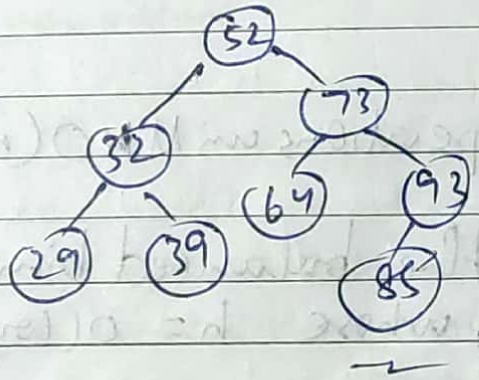
→ Replace that node with its child

Ex:-



Delete(26)

∵ 26 has one child '39', replace it with its child.



Case 2:- X is in a node, with 2 children → O(h)

Some definitions to look upon:-

a.) inorder predecessor of y is defined as the greatest no. smaller than y. That is the greatest no. in the left subtree of y.

b.) inorder successor of y is defined as the smallest no. greater than y. That is the smallest no in the right subtree of y.

—x— → will have either one or no child

To delete x, find its (inorder predecessor); replace x with it & then delete the inorder predecessor

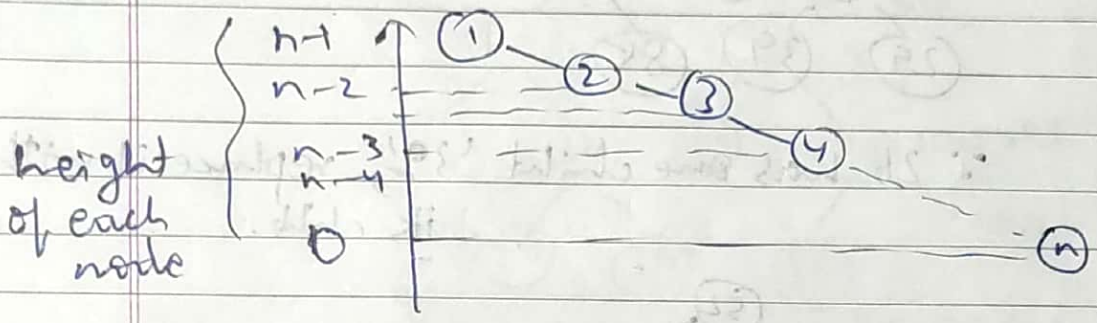
If at all the inorder predecessor has a child, it will be a left child.

n → total no. of nodes

Date: / /
Page No.:

Height of a BST can be atmost n-1

For ex:→ you get a degenerate tree if you insert the no. in ↑ing order



Hence, the operations will be O(n).

Here comes the balanced binary search tree (BBST), whose $h = O(\log n)$

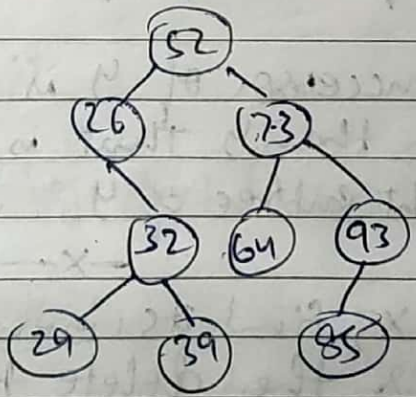
AVL is an example of BBST.

(*) Inorder traversal of BST

Q) What is an inorder traversal?

Ans You first visit the left subtree, then you visit the node & then you visit the right subtree

Ex:-



Inorder traversal gives → 26, 29, 32, 39, 52, 64, 73, 85, 93

Date: / /
Page No.:

By observing, we see that it's a sorted sequence

And Inorder traversal takes linear time

⇒ Given a BST, we can get the sorted sequence in linear time.

Q.) Given a sequence of keys, can we build a BST in linear time?

Ans → NO

Build BST + inorder traversal = Sorting

⇒ If we can build BST in linear time, we can sort in linear time, which is not possible.

(*) BST vs Binary heap

Building a binary heap by calling insert/add n time took $\Omega(n \log n)$ time, but we can build heap in linear time.

Any algo to build a binary search tree will take $\Omega(n \log n)$ time due to above average analysis

Q.) Given a heap can we build the BST in linear time?

Ans → NO Build bin heap + convert to BST + Inorder traversal = Sorting

If we can convert a bin heap into BST in linear time, we can sort in linear time.

That is BSTs are more complicated & powerful data structure than binary heaps.

(*) BST vs hashing

Search, Insert and delete

hashing: $O(1)$ but in worst case it will take linear time

BBST: $O(\log n)$ even in worst case

Moreover, some operations like Rank(x), can be done in $O(\log n)$ using BBST, but hashing takes linear time

∴ if you want to do only search, insert and delete then you might prefer hashing over BBST.

However, if operations like Rank are also there, then BBST should be your choice

Rank(x) = (no. of no. greater than x) + 1

→

```
struct bst
{
    long long int key;
    struct bst *left, *right;
};
```

→ BST without parent pointer

BST with parent pointer

→
```
struct bst
{
  long long int key;
  struct bst *left, *right, *parent;
};
```

—x—

I Implementation of BST with parent pointer

1.) ~~search(x)~~ Search

~~struct BST *search(x)~~

```
struct BST *search(struct BST *node, long long int x)
{
  while(node)
  {
    if (x == node->key) return node;
    else if (x < node->key) node = node->left;
     else node = node->right;
  }
  return NULL;
}
```

—x—

```
struct BST* newNode(long long int key)
{
  struct BST *temp=(struct BST*) malloc(sizeof(struct BST));
  temp->key = key;
  temp->left = temp->right = temp->parent = NULL;
  return temp;
}
```

2) Insert(x)

→ we have to insert key

```
void insert(struct BST **node, long long int key)
{
    if(!*node)        *node = newNode(key);
    else
    {
        bool flag = true; struct BST *temp = *node;
        while(flag)
        {
            if(key < temp->key)
            {
                if(temp->left)    temp = temp->left;
                else
                {
                    temp->left = newNode(key);
                    temp->left->parent = temp;
                    flag = false;
                }
            }
            else if(key > temp->key)
            {
                if(temp->right)  temp = temp->right;
                else
                {  temp->right = newNode(key);
                   temp->right->parent = temp;
                   flag = false;}}
            else
                flag = false; } }
    }
```

handle the case when root node itself is NULL

if key already exists in the tree

3.) Delete (x)

→
```
void Delete(struct BST **root, long long int X)
{
   struct BST *node = *root;
   while(node)
    { if (x == node->key)  DeleteNode(&*root, node);

      else if (x < node->key)  node = node->left;
      else  node = node->right;
    }
}
```

⇒ DeleteNode(node)

→
```
void DeleteNode(struct BST **root, struct BST *node)
{
  if (node->left && node->right)        // if the node has 2 children
  {
    struct BST *temp = node->left;
    while(temp->right)  temp = temp->right;

    node->key = temp->key;
    node = temp; }

  struct BST *child, *p = node->parent;
  if (node->left) child = node->left;
  else child = node->right;
  if (*root == node) *root = child;
  if (child) child->parent = p;
  if (p) { if (p->left == node) p->left = child;
    else p->right = child; } free(node);
}
```

replaces the key of root node with its predecessor

the right child may or may not be there

handles the deletion of nodes with atmost one child

handles the case when node to be deleted is root node

II BST implementation without parent pointer

1.) Search(x) → will exactly be the same as before

For Insert(x) & Delete(x), we'll write recursive functions

2.) Insert(x)

```
struct BST* insert(struct BST *node, long long int key)
{
  if(!node)  return newNode(key);

  if(key < node->key)
      node->left = insert(node->left, key);
  else if(key > node->key)
      node->right = insert(node->right, key);
  return node;
}
```

```
if(node->left && node->right){
    struct BST *temp = node->left;
    while(temp->right) temp = temp->right;
    node->key = temp->key;
    node->left = Delete(node->left, temp->key);}

else
  { struct BST *child;
    if(!node->left) child = node->right
```

3.) Delete (X)

```
struct BST* Delete(struct BST *node, long long int x)
{
    if(!node) return node;

    if(node->key > x){
        node->left = Delete(node->left, x)
        return node;}
    else if(node->key < x){
        node->right = Delete(node->right, x);
        return node;}

    if(node->left && node->right){
        struct BST *temp = node->left;
        while(temp->right) temp = temp->right;
        node->key = temp->key;
        node->left = Delete(node->left, temp->key);
        }
    else{
        struct BST *child;
        if(!node->left) child = node->right;
        else child = node->left;
        free(node);
        return child;}

}
```

if the node does not have a left child

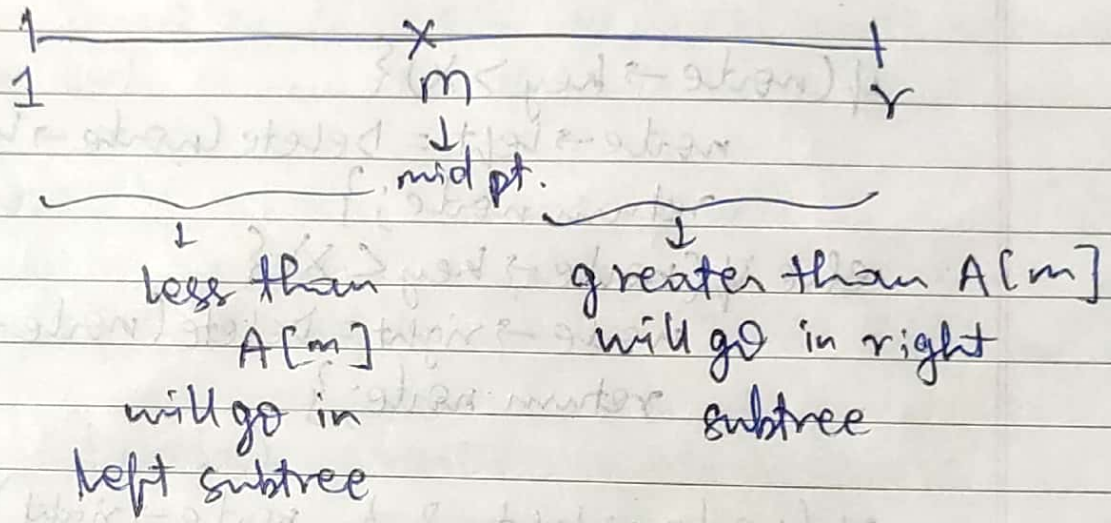
it may or may not have a right child

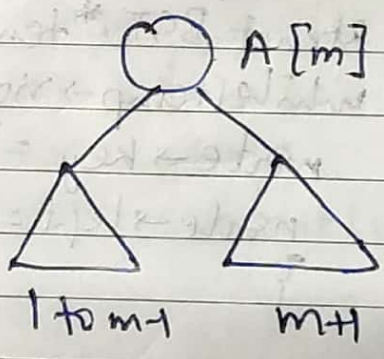
Date: / /
Page No.:

(*) Build a BST

important

Given a sorted sequence in ↑ing order, build a BST





As can be easily observed, we'll need a recursive fn to ~~do so.~~ build the tree.

Struct BST*

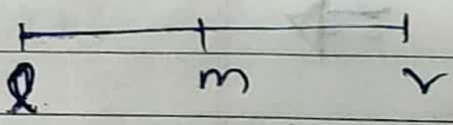
→ CreateBBST(long long int A[], long long int l, long long int r)

```
{
    struct BST *node = NULL;
    if(l<=r){
        long long int m = (l+r)/2;
        node = (struct BST *)malloc(sizeof(struct BST));
        node->key = A[m];
        node->left = CreateBBST(A, l, m-1);
        node->right = CreateBBST(A, m+1, r);
        if(node->right) node->right->parent = node;
        if(node->left) node->left->parent = node;
    }
    return node;
}
```

needed only when we're given BST with parent pointer

—x—

Complexity of above fun^n :-



** $T(n) = O(1) + T(n/2) + T(n/2)$
$= O(1) + 2T(n/2)$
$= O(n)$ (Master's Theo.)

—x—

Height of BBST thus formed

** $H(n) = 1 + \text{Max}\{H(n/2), H(n/2)\}$;
$= 1 + H(n/2)$
$= O(\log n)$

→ Given a sorted sequence of keys, a BBST can be built in linear time.

→ Given a ~~unsorted~~ sequence of keys, a BBST can ~~be~~ be built in $O(n \log n)$

—×—

✳ Predecessor & Successor → $O(H)$; H is height of the tree

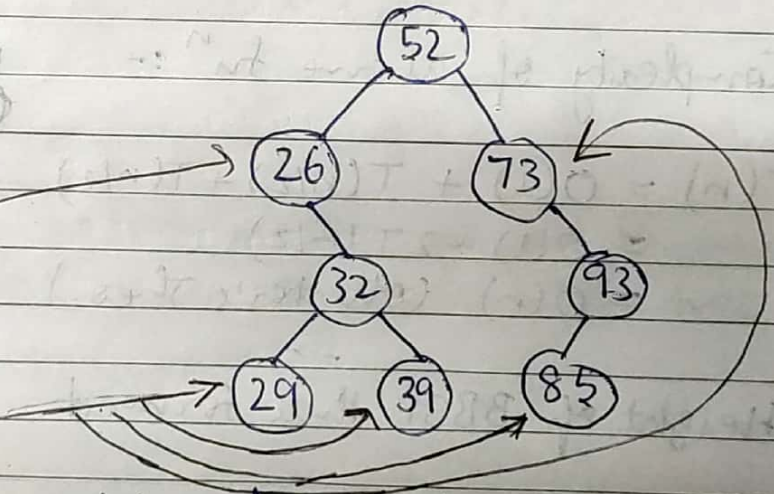
Predecessor (X) = largest no. smaller than X
Successor (X) = smallest no. larger than X

Suppose you are give a node 'N' such that N → key = X then

Predecessor (N) = node which contains predecessor of X
Successor (N) = node which contains successor of X

—×—

⇒



If there exists a left subtree for a node then we already know how to find its predecessor. However, if the node doesn't have a subtree then:-
Case-1 :- it has a predecessor

Case-2 :- it does not have a predecessor

→ when you have the parent pointer

Date: / /
Page No.:

For cases, when nodes do not have a left subtree :-

go to the parent of the node & check whether the node is a right child or not.

If yes then the parent is the predecessor.
If not then do the same thing with the parents & so on until you find one

For ex :-

1.) find predecessor of 39

39 is right child of its parent (32)
⇒ 32 is its predecessor
Similarly 52 is predecessor of 73

2.) find predecessor of 29

29 is the left child of 32.
Now we check for the parent
32 is the right child of 26

⇒ 26 is predecessor of 29

Similarly 73 is predecessor of 85.

3.) Find predecessor of 26

26 is left child of 52
Now, we check for the parent.
52 is no one's right child. In fact 52 is root
⇒ 26 does not have a predecessor

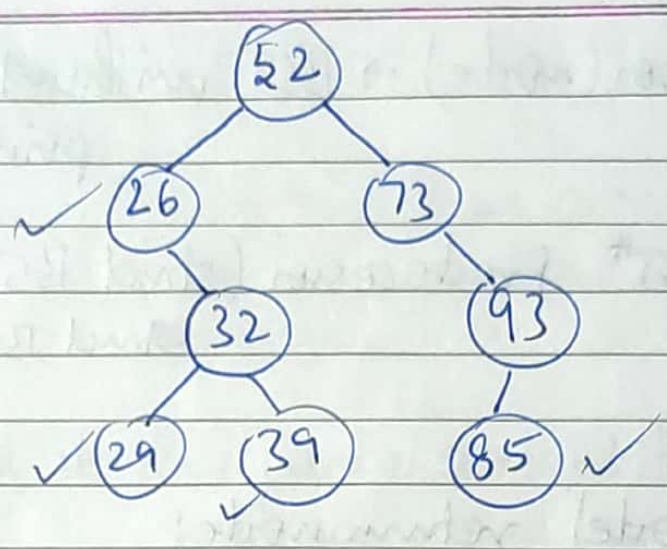
(*) Predecessor (node) → BST with parent pointer

```
struct BST* Predecessor (struct BST *node)
{
  if (!node) return node;

  if (node → left)
  {
     node = node → left;
     while (node → right) node = node → right;
     return node;
  }

  else{
     while (node → parent)
     {
        if(node → parent → right == node)
           return node → parent;
        else
           node = node → parent; }
        return node → parent;
     }
}
```

——— × ———

Now, we want to ~~see~~ find the predecessor when parent pointer is not given.

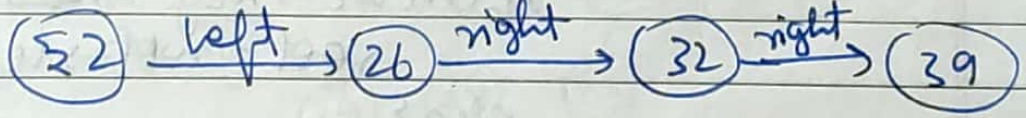
⇒ for nodes which do not have a left subtree:→ Start from the root node and search the given node, keeping in mind to take care of when you go right.



for 85, we start from 52
at 52 we ~~turned~~ right & reach 73
at 73 " again turn right & reach 93
at 93 " turn left & reach 85

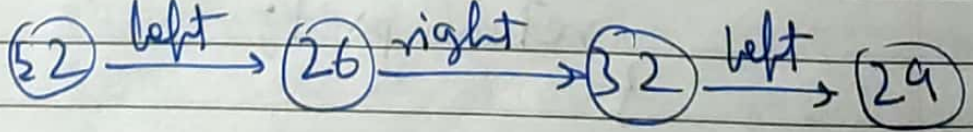
∴ the last time we turned right while searching for 85 was at 73
⇒ 73 is predecessor of 85.

Similarly for 39



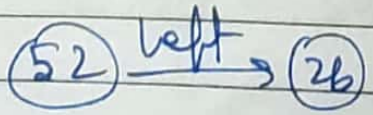
last right was at 32 ⇒ 32 is the pred.

for 29



last right was at 26 ⇒ 26 is the pre

for 26



; no right
⇒ no pred. exists.

(*) Predecessor(node) → BST without parent pointer

→
```
struct BST* Predecessor (struct BST *node,
                         struct BST *root)
{
  if (!node) return node;

  if (node → left)
  {
    node = node → left;
    while(node → right)  node = node → right;
    return node; }

  else{
    struct BST *pred = NULL;
    while(node != root){
      if (node → key < root → key)
        root = root → left;
      else {
        pred = root;
        root = root → right;
      } }

    return pred;
  }
}
```

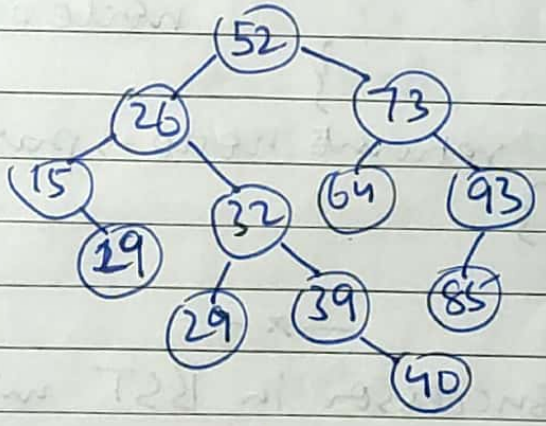
⇒ Successor (in BST with parent pointer)

Case A :- the node has a right subtree.
Find the smallest no. in the right subtree.

Case B :→ the node doesn't have a right subtree

Go to the parent of the node & check whether the node is left child or not.
If yes then the parent node is the successor.
If not then do the same thing with the parent.
& so on until you find one

Ex :-



Find successor of 40 :-
40 is right child of 39
we go to its parent
39 is right child of 32, go to the parent of 39
32 is right child of 26, go to the parent of 32
26 is left child of 52

⇒ 40 is successor of 52

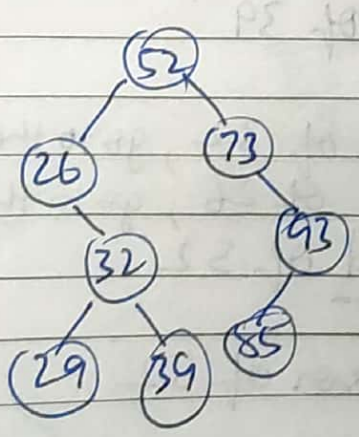
→ BST with parent pointer

Date: / /
Page No.:

→
```
struct BST* Successor (struct BST *node)
{
    if(!node) return node;
    if (node → right)
    {
        node = node → right;
        while (node → left) node = node → left;
        return node;
    }
    else {
        while (node → parent){
            if (node → parent → left == node)
                return node → parent
            else
                node = node → parent
        }
        return node → parent
    }
}
```

—×—

(*) finding successor in BST without parent pointer.



Ex:- find successor of 85
Start from root node:-

52 —right→ 73 —right→ 93 —left→ 85

last left at 93

⇒ 93 is successor of 85

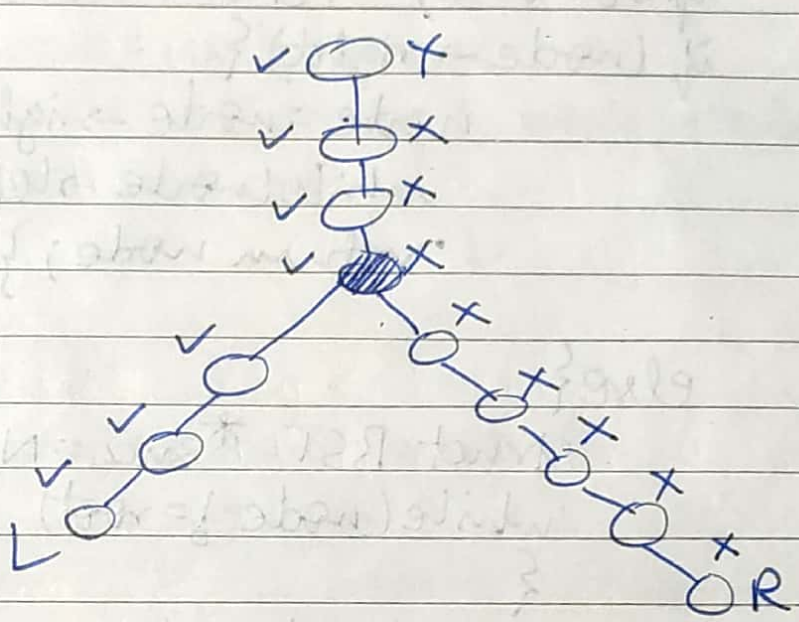
→ WITHOUT parent ptr.

Date: / /
Page No.:

→
```
struct BST* Succesor(struct BST *node,
                     struct BST *root)
{
    if(!node) return node;
    if (node->right) {
        node = node->right;
        while(node->left) node = node->left;
        return node; }

    else{
        struct BST *succ = NULL;
        while(node != root)
        {
            if(node->key < root->key)
            {
                succ = root;
                root = root->left;
            }

            else
                root = root->right;
        }
        return succ;
    }
}
```

(11) Least Common Ancestor (LCA) → O(H)

LCA(L,R) is the node of min. height which is ancestor of both L & R

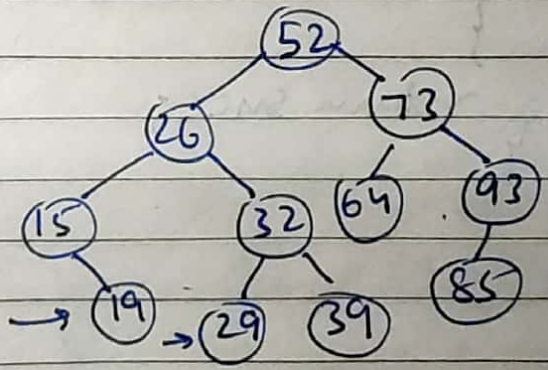


(✓) nodes are all ancestors of L
(X) nodes " " " " R

(✓) & (X) nodes are common ancestors of L & R

[(✓) & (X) & ●] node is LCA of L & R.

Ex:-



ancestors of 19 are 15, 26 & 52
" " 29 " 32, 26 & 52
common " " 19 & 29 are 26 & 52
LCA of 19 & 29 is 26

Date: / /
Page No.:

of L & R

That is LCA is the root of the smallest subtree containing L & R.

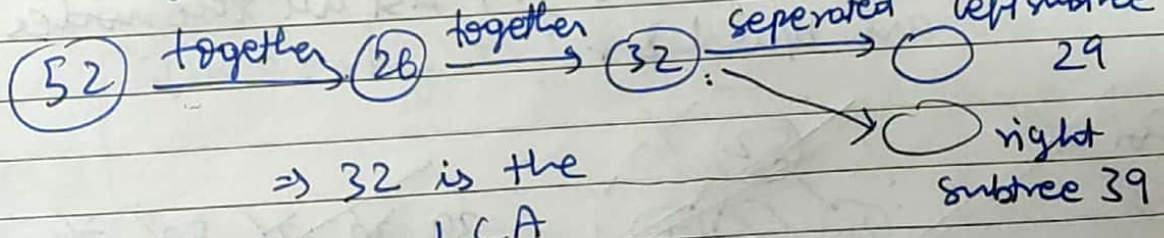
To find LCA,
~~look~~ we find the node at which L & R get seperated.

For eg:- we ~~start~~ at ~~root~~ node → 52 & we need to find LCA of 19 & 29.
both 19 & 29 lies on left subtree of 52.
then we come to 26

19 lies on left subtree of 26 while
29 lies " right " of 26

i.e 19 & 29 get ~~seperated~~ seperated
⇒ 26 is the LCA

Similarly for 29 & 39



⇒ 32 is the LCA

→ K = no. of numbers in the subtree b/w l & r

Date: / /
Page No.:

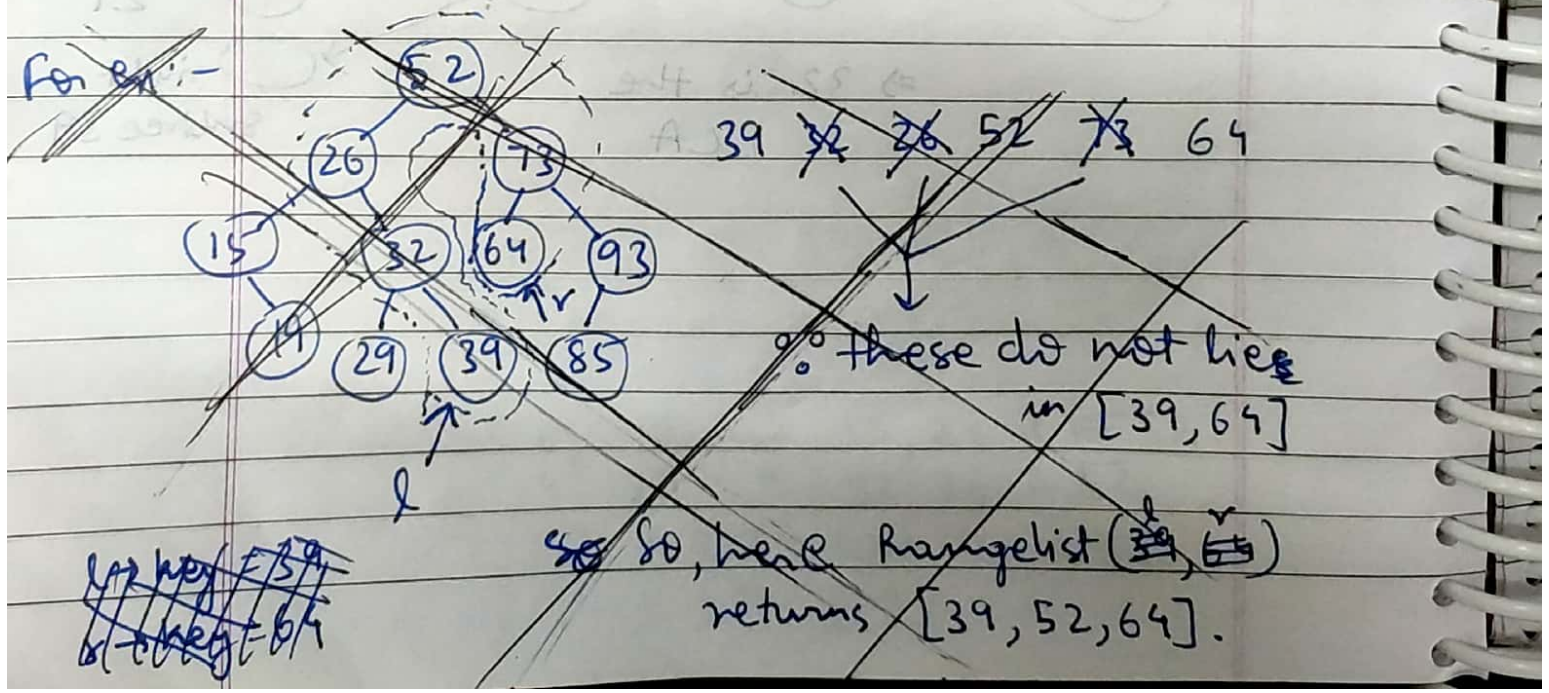
→
```
struct BST* LCA(struct BST *node,
                struct BST *l, struct BST *r)
{
    if (!node || l->key > r->key)
        return NULL;

    if (node->key > r->key)
        return LCA(node->left, l, r);

    if (node->key < l->key)
        return LCA(node->right, l, r);

    return node;
}
```

—×—

(#) Rangelist → O(H+K)

→ Rangelist(l, r) list all the nodes which are b/w l & r.

For ex:-



Date: / /
Page No.:

Rangelist will actually return a linked list whose each node will contain the pointer to the nodes which lie in smallest subtree containing l & r
such that the key value of all the nodes lie in [l->key, r->key]

—x—

→ Struct List
{
    struct BST *node;
    struct List *next;
};

—x—

→ void Rangelist(struct BST *node, long long int l, long long int r, struct List **list)

{
  if(node){
    if(node->key > r) Rangelist(node->left, l, r, &*list);
    else if(node->key < l)
      Rangelist(node->right, l, r, &*list);
    else {
      Rangelist(node->right, l, r, &*list);
      struct List *temp = (struct List*)malloc(sizeof(struct List));
      temp->node = node;
      temp->next = *list;
      *list = temp;
      Rangelist(node->left, l, r, &*list);}}}

finding root of the smallest subtree containing l & r

visiting right subtree

adding each node to the beginning of the list

visiting left subtree

(*) Rangecount → O(H+k)

Rangecount(l, r) counts the no. of numbers b/w l & r.

→
```
long long int Rangecount(struct BST *node,
                         long long int l,
                         long long int r)
{
    if (!node or l > r) return 0;

    if (node->key > r)
        return Rangecount(node->left, l, r);

    if (node->key < l)
        return Rangecount(node->right, l, r);

    return 1 + Rangecount(node->left, l, r)
             + Rangecount(node->right, l, r);
}
```