Chapter 11: Search Trees

Data Structures and Algorithms with C

Binary search tree | Insertion | Search | Depth | Deletion | AVL rotations | Red-black, B-tree

Source: Horowitz–Sahni 5.7, 10.2 · CLRS 12–13

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/dsa_c/slides/Chapter_11_Search_Trees.html

Learning Outcomes

Apply the definition

Check the four properties of a binary search tree on a given tree.

Insert and search

Write `bst_insert` and `bst_search`, both iteratively and recursively.

Compute depth

Write depth recursively, with no extra parameter and no global variable.

Delete

Compare four ways to delete a node with no child, one child or two children.

Keep balance

Measure the cost of imbalance; apply the LL, RR, LR and RL rotations of an AVL tree.

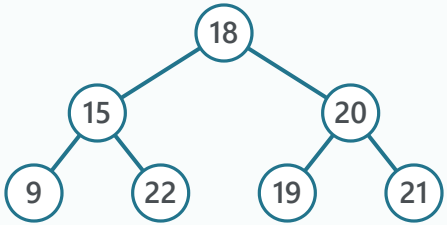
Choose a structure

Tell BST, AVL, red-black and B-tree apart by where each is used.

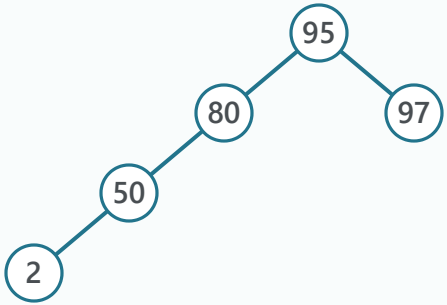
Binary Search Tree

1. All keys are distinct; every value is **unique**.
2. Keys in a nonempty **left** subtree are **smaller** than the key at the root.
3. Keys in a nonempty **right** subtree are **larger** than the key at the root.
4. The left and right subtrees are binary search trees too.

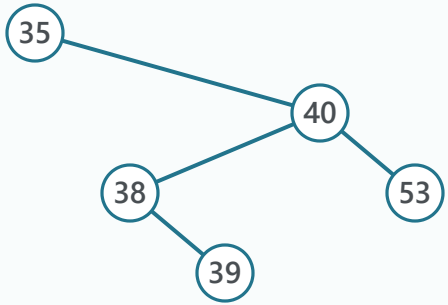
A binary search tree is a special kind of binary tree. An inorder traversal visits its keys in **sorted** order.



NOT a binary search tree

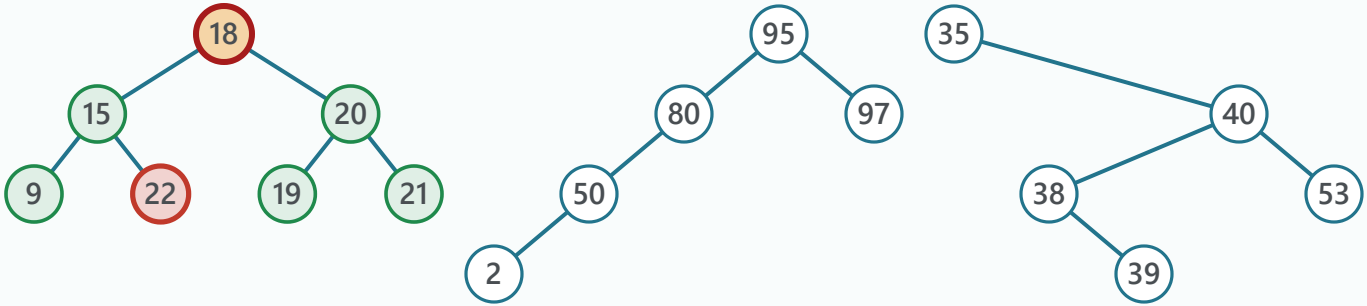


binary search tree



binary search tree

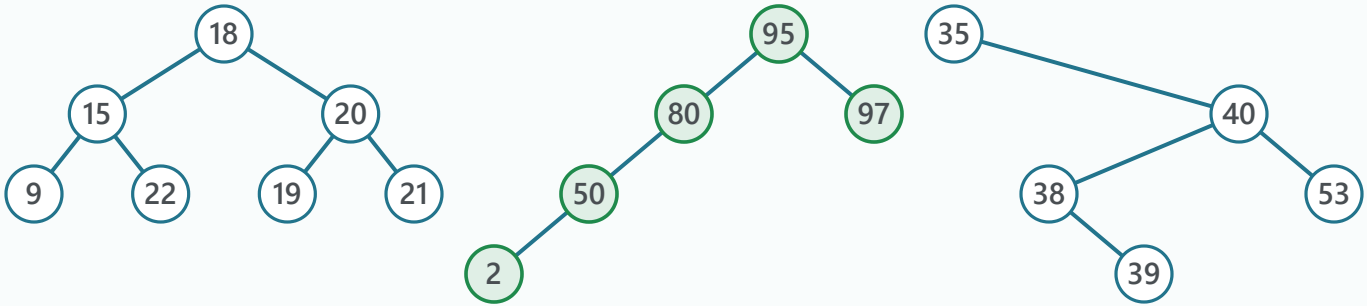
Which Ones Are Binary Search Trees?



NOT a binary search tree | **binary search tree** | **binary search tree**

22 is right of 15, so larger than 15 (fine), but it sits in the **left** subtree of 18 and is larger than 18 → violates the definition.

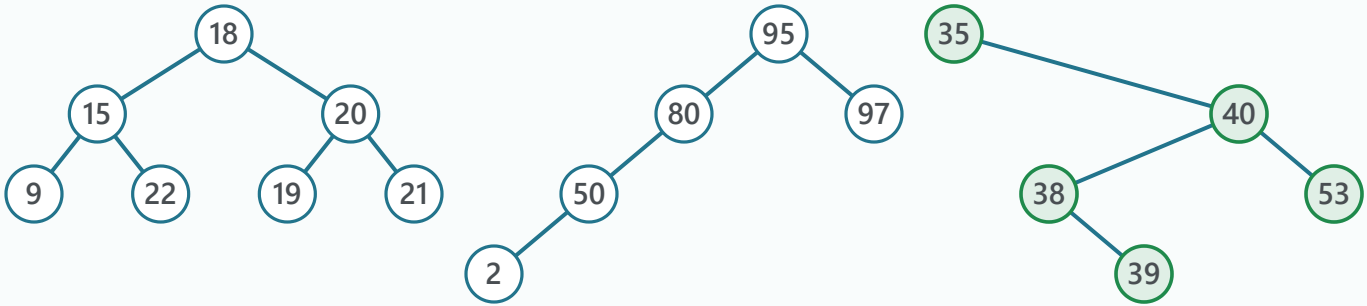
Which Ones Are Binary Search Trees?



NOT a binary search tree **binary search tree** **binary search tree**

Every key left of 95 (80, 50, 2) is smaller than 95; 80's left is 50, 50's left is 2: the order holds.

Which Ones Are Binary Search Trees?



NOT a binary search tree **binary search tree** **binary search tree**

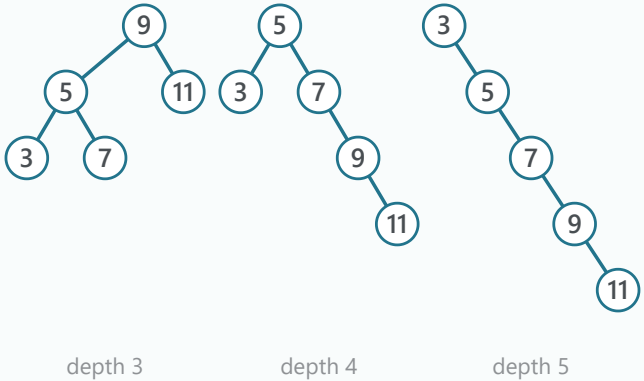
Every key right of 35 is larger; 40's left holds 38 and 39 (below 40, above 35): valid.

Insertion: The Tree Decides the Place

INSTRUCTOR "Give me a value and I'll insert it. 'Sir, put it to the right of 15.' Left or right, where it goes is my business! ☺"

QUESTION: Using the set 3, 5, 7, 9, 11, draw binary search trees of depth 3, 4 and 5. Can you draw one of depth 2?

▶ **Solution**



depth 3 depth 4 depth 5

Does the Insertion Order Matter?





47
depth 1

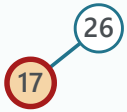
26
depth 1

Insert **47** (left) and **26** (right): start at the root, go left if smaller, right if larger, place at the first NULL.

Step 1 / 9

Does the Insertion Order Matter?



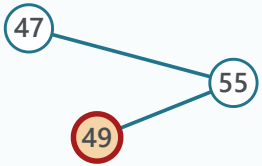


47 55
depth 2

26 17
depth 2

Insert **55** (left) and **17** (right): start at the root, go left if smaller, right if larger, place at the first NULL.

Does the Insertion Order Matter?



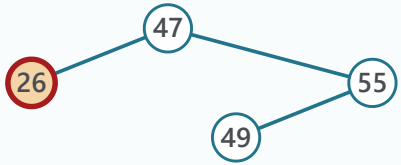
47 55 49
depth 3



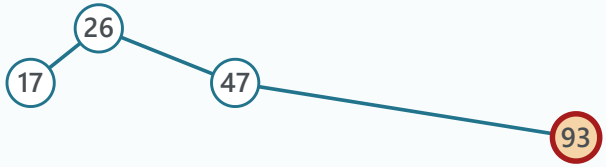
26 17 47
depth 2

Insert **49** (left) and **47** (right): start at the root, go left if smaller, right if larger, place at the first NULL.

Does the Insertion Order Matter?



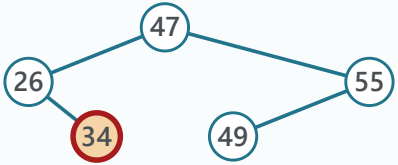
47 55 49 26
depth 3



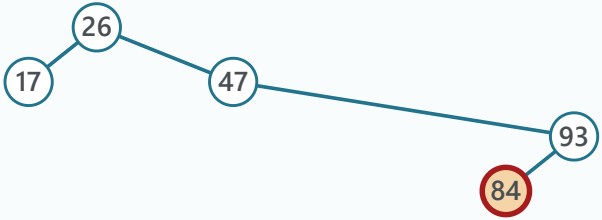
26 17 47 93
depth 3

Insert **26** (left) and **93** (right): start at the root, go left if smaller, right if larger, place at the first NULL.

Does the Insertion Order Matter?



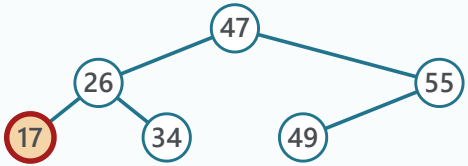
47 55 49 26 34
depth 3



26 17 47 93 84
depth 4

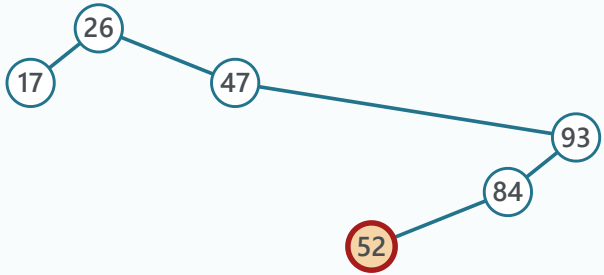
Insert **34** (left) and **84** (right): start at the root, go left if smaller, right if larger, place at the first NULL.

Does the Insertion Order Matter?



47 55 49 26 34 17

depth 3

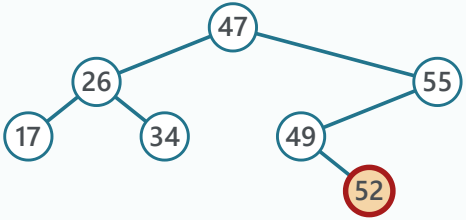


26 17 47 93 84 52

depth 5

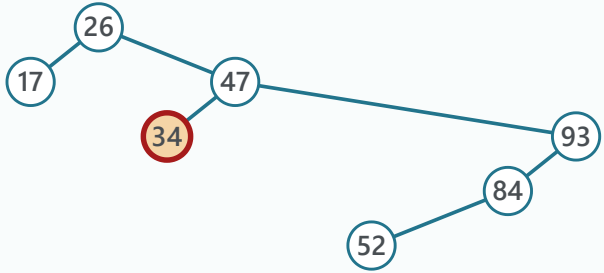
Insert **17** (left) and **52** (right): start at the root, go left if smaller, right if larger, place at the first NULL.

Does the Insertion Order Matter?



47 55 49 26 34 17 52

depth 4

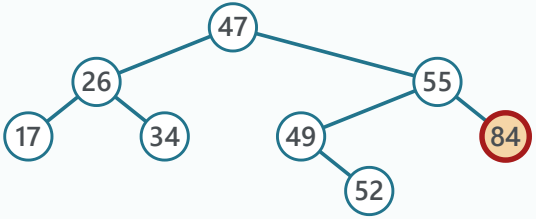


26 17 47 93 84 52 34

depth 5

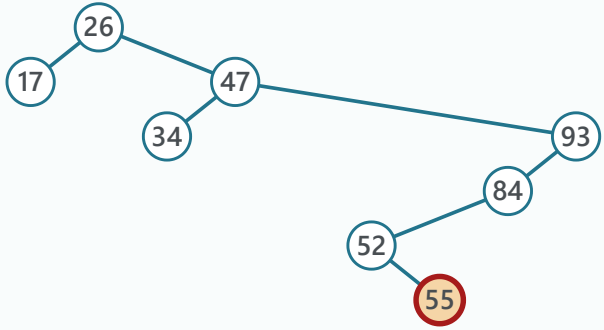
Insert **52** (left) and **34** (right): start at the root, go left if smaller, right if larger, place at the first NULL.

Does the Insertion Order Matter?



47 55 49 26 34 17 52 84

depth 4

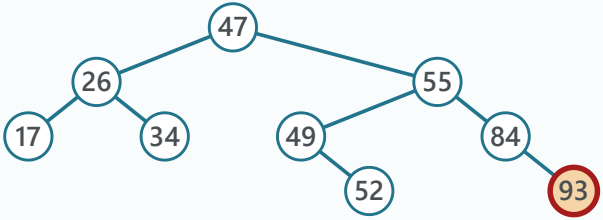


26 17 47 93 84 52 34 55

depth 6

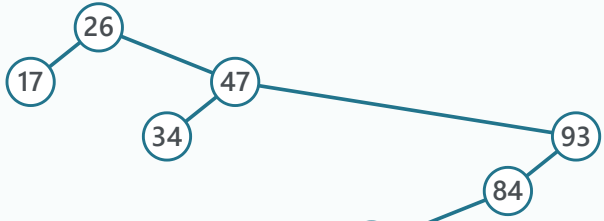
Insert **84** (left) and **55** (right): start at the root, go left if smaller, right if larger, place at the first NULL.

Does the Insertion Order Matter?



47 55 49 26 34 17 52 84 93

depth 4



26 17 47 93 84 52 34 55 49

depth 6

Result: depth **4** and **6**. The inorder traversal is the same for both: 17 26 34 47 49 52 55 84 93.

Skew, Balanced Trees and Frequently Used Nodes

- If the keys arrive **sorted**, the tree leans right or left and its depth becomes **n**: it is a list.
- If the depth is ≈ lg n, the BST is **balanced**: insert, delete and search take O(lg n).
- **Self-balancing trees**: algorithms that rebalance the tree as keys arrive — red-black, AVL.
- **Frequently used nodes**: algorithms that move often-searched nodes toward the root ("zig-zag"; known in the literature as the *splay tree*, Sleator–Tarjan).

INSTRUCTOR "If balance really matters, use algorithms that build a balanced tree. They are not that hard, they are in the books, just a bit longer. I may give one as homework."

??? — Two Questions to Think About

??? Can you recover the insertion order from the shape of the tree? Is the tree built from a given input *unique*?

▶ **Answer**

??? Can you recover the tree from its preorder, inorder or postorder sequence?

▶ **Answer**

INSTRUCTOR "Until now there was no tree behind the code, so we could not test it by running it."

Now we can build sample trees with insertion and really run the binary tree routines (traversal, copy, equality).

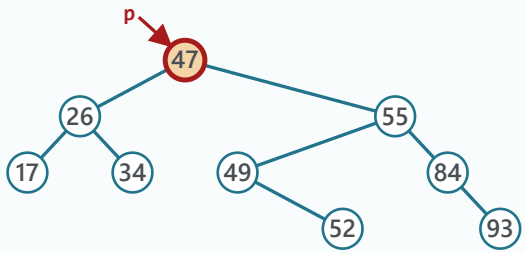
How Does Insertion Work?

- You cannot insert under a node that has both children; the new node goes under a node **missing at least one child**.
- Search for the new key from the root: go **left** at a node with a larger key, **right** at a node with a smaller key.
- If the key is already there, nothing is inserted: duplicates break the definition.
- The new node replaces the first **NULL** the search pointer reaches, so we need a pointer that **trails** one step behind (q).

```
typedef enum { FALSE, TRUE } boolean;
typedef struct node *tree_ptr;
struct node {
    tree_ptr left;
    int       data;
    tree_ptr right;
};
/* get_node(): malloc + NULL check,
   sets both children to NULL */
```

bst_insert(50, &L) Step by Step

```
1  boolean bst_insert(int x, tree_ptr *L) {
▶    tree_ptr q = NULL, p = *L, r; // q trails p
3    while (p) {                   // down to NULL
4      q = p;                      // q follows p
5      if (x < p->data) p = p->left;
6      else if (x > p->data) p = p->right;
7      else { printf("Duplicate key.\n");
8             return FALSE; }
9    }
10   r = get_node();               // children NULL
11   r->data = x;
12   if (*L == NULL) *L = r;       // empty: root
13   else if (x < q->data) q->left = r;
14   else q->right = r;
15   return TRUE;
16 }
```



x = 50

p points to the root (47); q (the trailing pointer) is NULL.

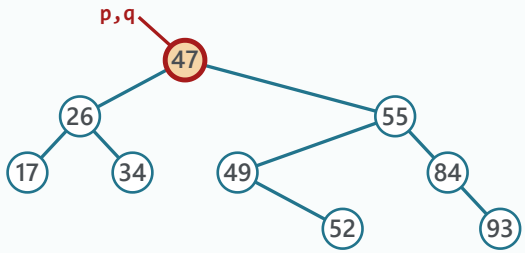
Step 1 / 13

VARIABLES

`x = 50` `p = &node(47)` `q = NULL` `r = ?`

bst_insert(50, &L) Step by Step

```
1  boolean bst_insert(int x, tree_ptr *L) {
2    tree_ptr q = NULL, p = *L, r; // q trails p
3    while (p) {                   // down to NULL
▶      q = p;                      // q follows p
5      if (x < p->data) p = p->left;
6      else if (x > p->data) p = p->right;
7      else { printf("Duplicate key.\n");
8             return FALSE; }
9    }
10   r = get_node();               // children NULL
11   r->data = x;
12   if (*L == NULL) *L = r;       // empty: root
13   else if (x < q->data) q->left = r;
14   else q->right = r;
15   return TRUE;
16 }
```



x = 50

q = p = 47: the trailing pointer stays one step behind; we will attach to q.

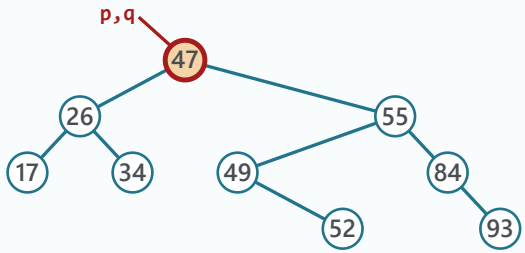
Step 2 / 13

VARIABLES

x = 50 | p = &node(47) | q = &node(47) | r = ?

bst_insert(50, &L) Step by Step

```
1  boolean bst_insert(int x, tree_ptr *L) {
2    tree_ptr q = NULL, p = *L, r; // q trails p
3    while (p) {                   // down to NULL
4      q = p;                      // q follows p
5      if (x < p->data) p = p->left;
▶      else if (x > p->data) p = p->right;
7      else { printf("Duplicate key.\n");
8             return FALSE; }
9    }
10   r = get_node();               // children NULL
11   r->data = x;
12   if (*L == NULL) *L = r;       // empty: root
13   else if (x < q->data) q->left = r;
14   else q->right = r;
15   return TRUE;
16 }
```



x = 50

50 > 47 → it belongs in the **right** subtree: p = 47->right = 55.

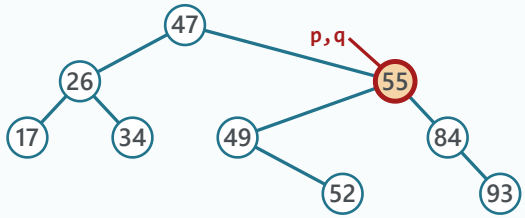
Step 3 / 13

VARIABLES

`x = 50` `p = &node(47)` `q = &node(47)` `r = ?`

bst_insert(50, &L) Step by Step

```
1  boolean bst_insert(int x, tree_ptr *L) {
2    tree_ptr q = NULL, p = *L, r; // q trails p
3    while (p) {                   // down to NULL
▶      q = p;                      // q follows p
5      if (x < p->data) p = p->left;
6      else if (x > p->data) p = p->right;
7      else { printf("Duplicate key.\n");
8             return FALSE; }
9    }
10   r = get_node();               // children NULL
11   r->data = x;
12   if (*L == NULL) *L = r;       // empty: root
13   else if (x < q->data) q->left = r;
14   else q->right = r;
15   return TRUE;
16 }
```



x = 50

q = p = 55: the trailing pointer stays one step behind; we will attach to q.

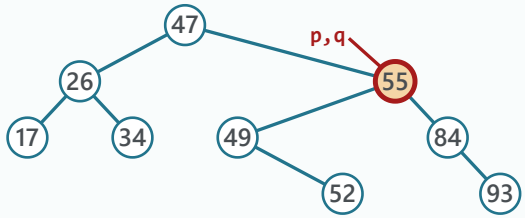
Step 4 / 13

VARIABLES

`x = 50` `p = &node(55)` `q = &node(55)` `r = ?`

bst_insert(50, &L) Step by Step

```
1  boolean bst_insert(int x, tree_ptr *L) {
2    tree_ptr q = NULL, p = *L, r; // q trails p
3    while (p) {                   // down to NULL
4      q = p;                      // q follows p
▶      if (x < p->data) p = p->left;
6      else if (x > p->data) p = p->right;
7      else { printf("Duplicate key.\n");
8             return FALSE; }
9    }
10   r = get_node();               // children NULL
11   r->data = x;
12   if (*L == NULL) *L = r;       // empty: root
13   else if (x < q->data) q->left = r;
14   else q->right = r;
15   return TRUE;
16 }
```



x = 50

50 < 55 → the new node belongs in the **left** subtree: p = 55->left = 49.

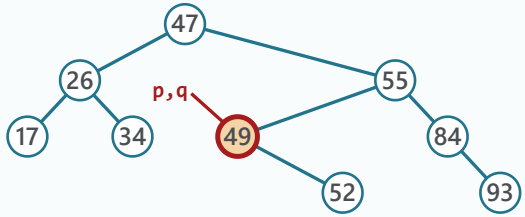
Step 5 / 13

VARIABLES

x = **50** | p = **&node(55)** | q = **&node(55)** | r = **?**

bst_insert(50, &L) Step by Step

```
1  boolean bst_insert(int x, tree_ptr *L) {
2    tree_ptr q = NULL, p = *L, r; // q trails p
3    while (p) {                   // down to NULL
▶      q = p;                      // q follows p
5      if (x < p->data) p = p->left;
6      else if (x > p->data) p = p->right;
7      else { printf("Duplicate key.\n");
8             return FALSE; }
9    }
10   r = get_node();               // children NULL
11   r->data = x;
12   if (*L == NULL) *L = r;       // empty: root
13   else if (x < q->data) q->left = r;
14   else q->right = r;
15   return TRUE;
16 }
```



x = 50

q = p = 49: the trailing pointer stays one step behind; we will attach to q.

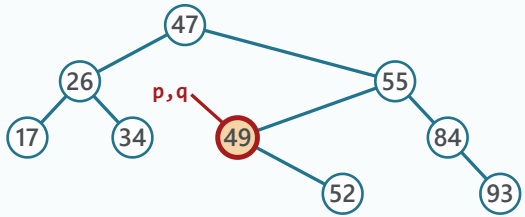
Step 6 / 13

VARIABLES

`x = 50` `p = &node(49)` `q = &node(49)` `r = ?`

bst_insert(50, &L) Step by Step

```
1  boolean bst_insert(int x, tree_ptr *L) {
2    tree_ptr q = NULL, p = *L, r; // q trails p
3    while (p) {                   // down to NULL
4      q = p;                      // q follows p
5      if (x < p->data) p = p->left;
▶      else if (x > p->data) p = p->right;
7      else { printf("Duplicate key.\n");
8             return FALSE; }
9    }
10   r = get_node();               // children NULL
11   r->data = x;
12   if (*L == NULL) *L = r;       // empty: root
13   else if (x < q->data) q->left = r;
14   else q->right = r;
15   return TRUE;
16 }
```



x = 50

50 > 49 → it belongs in the **right** subtree: p = 49->right = 52.

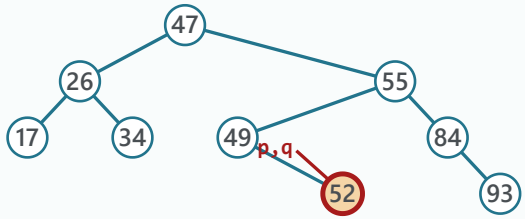
Step 7 / 13

VARIABLES

`x = 50` `p = &node(49)` `q = &node(49)` `r = ?`

bst_insert(50, &L) Step by Step

```
1  boolean bst_insert(int x, tree_ptr *L) {
2    tree_ptr q = NULL, p = *L, r; // q trails p
3    while (p) {                   // down to NULL
▶      q = p;                      // q follows p
5      if (x < p->data) p = p->left;
6      else if (x > p->data) p = p->right;
7      else { printf("Duplicate key.\n");
8             return FALSE; }
9    }
10   r = get_node();               // children NULL
11   r->data = x;
12   if (*L == NULL) *L = r;       // empty: root
13   else if (x < q->data) q->left = r;
14   else q->right = r;
15   return TRUE;
16 }
```



x = 50

q = p = 52: the trailing pointer stays one step behind; we will attach to q.

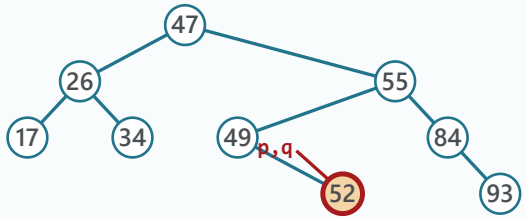
Step 8 / 13

VARIABLES

`x = 50` `p = &node(52)` `q = &node(52)` `r = ?`

bst_insert(50, &L) Step by Step

```
1  boolean bst_insert(int x, tree_ptr *L) {
2    tree_ptr q = NULL, p = *L, r; // q trails p
3    while (p) {                   // down to NULL
4      q = p;                      // q follows p
▶      if (x < p->data) p = p->left;
6      else if (x > p->data) p = p->right;
7      else { printf("Duplicate key.\n");
8             return FALSE; }
9    }
10   r = get_node();               // children NULL
11   r->data = x;
12   if (*L == NULL) *L = r;       // empty: root
13   else if (x < q->data) q->left = r;
14   else q->right = r;
15   return TRUE;
16 }
```



x = 50

50 < 52 → the new node belongs in the **left** subtree: p = 52->left = NULL.

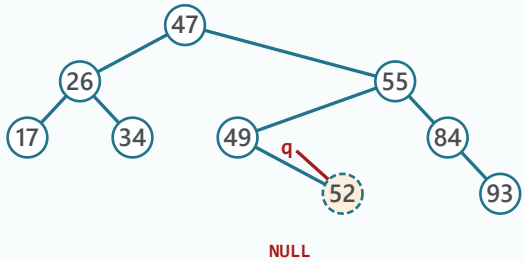
Step 9 / 13

VARIABLES

x = **50** | p = **&node(52)** | q = **&node(52)** | r = **?**

bst_insert(50, &L) Step by Step

```
1  boolean bst_insert(int x, tree_ptr *L) {
2    tree_ptr q = NULL, p = *L, r; // q trails p
▶    while (p) {                   // down to NULL
4      q = p;                      // q follows p
5      if (x < p->data) p = p->left;
6      else if (x > p->data) p = p->right;
7      else { printf("Duplicate key.\n");
8             return FALSE; }
9    }
10   r = get_node();               // children NULL
11   r->data = x;
12   if (*L == NULL) *L = r;       // empty: root
13   else if (x < q->data) q->left = r;
14   else q->right = r;
15   return TRUE;
16 }
```



x = 50

p is NULL: the loop ends. The new node takes the empty child slot of q = 52.

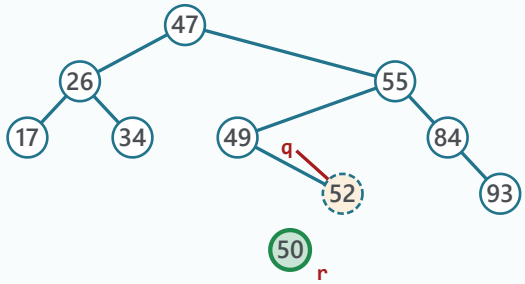
Step 10 / 13

VARIABLES

`x = 50` `p = NULL` `q = &node(52)` `r = ?`

bst_insert(50, &L) Step by Step

```
1  boolean bst_insert(int x, tree_ptr *L) {
2    tree_ptr q = NULL, p = *L, r; // q trails p
3    while (p) {                   // down to NULL
4      q = p;                      // q follows p
5      if (x < p->data) p = p->left;
6      else if (x > p->data) p = p->right;
7      else { printf("Duplicate key.\n");
8             return FALSE; }
9    }
▶    r = get_node();               // children NULL
▶    r->data = x;
12   if (*L == NULL) *L = r;       // empty: root
13   else if (x < q->data) q->left = r;
14   else q->right = r;
15   return TRUE;
16 }
```



x = 50

A new node is allocated (children NULL) and its data set to 50.

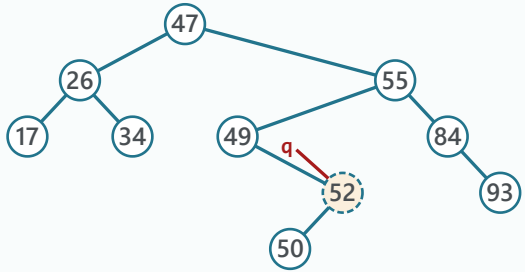
Step 11 / 13

VARIABLES

`x = 50` `p = NULL` `q = &node(52)` `r = &node(50)`

bst_insert(50, &L) Step by Step

```
1  boolean bst_insert(int x, tree_ptr *L) {
2    tree_ptr q = NULL, p = *L, r; // q trails p
3    while (p) {                   // down to NULL
4      q = p;                      // q follows p
5      if (x < p->data) p = p->left;
6      else if (x > p->data) p = p->right;
7      else { printf("Duplicate key.\n");
8             return FALSE; }
9    }
10   r = get_node();               // children NULL
11   r->data = x;
12   if (*L == NULL) *L = r;       // empty: root
▶    else if (x < q->data) q->left = r;
14   else q->right = r;
15   return TRUE;
16 }
```



x = 50

The tree is not empty; 50 < 52 → `q->left = r`. Linked.

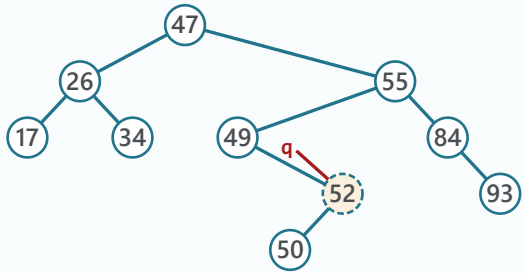
Step 12 / 13

VARIABLES

`x = 50` `p = NULL` `q = &node(52)` `r = &node(50)`

bst_insert(50, &L) Step by Step

```
1  boolean bst_insert(int x, tree_ptr *L) {
2    tree_ptr q = NULL, p = *L, r; // q trails p
3    while (p) {                   // down to NULL
4      q = p;                      // q follows p
5      if (x < p->data) p = p->left;
6      else if (x > p->data) p = p->right;
7      else { printf("Duplicate key.\n");
8             return FALSE; }
9    }
10   r = get_node();               // children NULL
11   r->data = x;
12   if (*L == NULL) *L = r;       // empty: root
13   else if (x < q->data) q->left = r;
14   else q->right = r;
▶    return TRUE;
16 }
```



x = 50

Returns TRUE. Only **one** pointer changed; the rest of the tree is untouched.

Step 13 / 13

VARIABLES

`x = 50` `p = NULL` `q = &node(52)` `r = &node(50)`

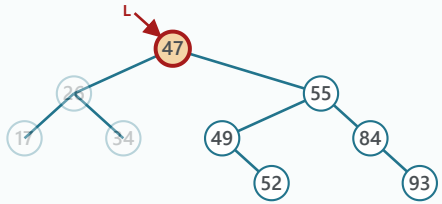
Why `tree_ptr *L`? What Does It Cost?

- If the tree is empty (`*L == NULL`), the new node becomes the **root**: the caller's root pointer must change.
- To change the pointer itself we pass its **address** (call by reference): `bst_insert(x, &root)` .
- Cost: one comparison per level → **Ω(lg n) – O(n)**.
- C99 fix: initialize `q = NULL` .

```
/* Recursive version: returns the new root */
tree_ptr insert(tree_ptr L, int x) {
  if (!L) {
    L = get_node();
    L->data = x;
  } else if (x < L->data)
    L->left = insert(L->left, x);
  else if (x > L->data)
    L->right = insert(L->right, x);
  return L;        /* root = insert(root, x); */
}
```

SEARCH · RUN THE CODE STEP BY STEP

bst_search(52, L) — Iterative



key x = 52

comparisons: 1

52 > 47 → go right. The left subtree is ruled out.

Step 1 / 4

```
1 tree_ptr bst_search(int x, tree_ptr L) {
2   while (L) {           // pick a subtree
3     if (x == L->data) return L;
4     if (x < L->data) L = L->left;
▶     else             L = L->right;
6   }
7   return NULL;          // not found
8 }
```

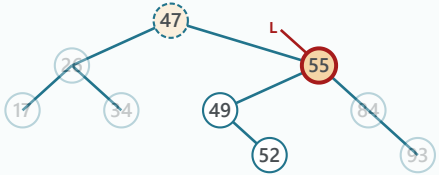
VARIABLES

`x = 52` `L = &node(47)`

“When a manager gives a clerk a task, it must be defined precisely. The clerk should never have to come back and knock on the door.”

SEARCH · RUN THE CODE STEP BY STEP

bst_search(52, L) — Iterative



key x = 52

comparisons: 2

52 < 55 → go left. The right subtree (faded) is never examined: that is where search gets its power.

Step 2 / 4

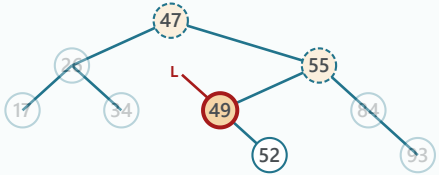
```
1 tree_ptr bst_search(int x, tree_ptr L) {
2   while (L) {           // pick a subtree
3     if (x == L->data) return L;
▶     if (x < L->data) L = L->left;
5     else             L = L->right;
6   }
7   return NULL;          // not found
8 }
```

VARIABLES

`x = 52` `L = &node(55)`

"When a manager gives a clerk a task, it must be defined precisely. The clerk should never have to come back and knock on the door."

bst_search(52, L) — Iterative



key x = 52

comparisons: 3

52 > 49 → go right. The left subtree is ruled out.

Step 3 / 4

```
1 tree_ptr bst_search(int x, tree_ptr L) {
2   while (L) {           // pick a subtree
3     if (x == L->data) return L;
4     if (x < L->data) L = L->left;
▶     else             L = L->right;
6   }
7   return NULL;          // not found
8 }
```

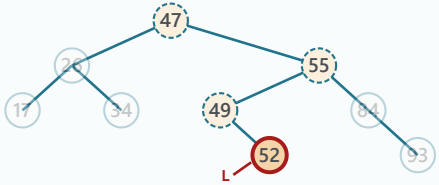
VARIABLES

`x = 52` `L = &node(49)`

"When a manager gives a clerk a task, it must be defined precisely. The clerk should never have to come back and knock on the door."

SEARCH · RUN THE CODE STEP BY STEP

bst_search(52, L) — Iterative



key x = 52

comparisons: 4

52 == 52: **found**; return the node's address. 4 comparisons in all.

Step 4 / 4

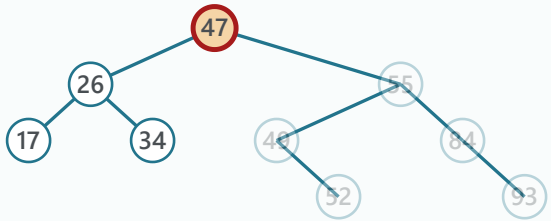
```
1 tree_ptr bst_search(int x, tree_ptr L) {
2   while (L) {           // pick a subtree
▶     if (x == L->data) return L;
4     if (x < L->data) L = L->left;
5     else             L = L->right;
6   }
7   return NULL;          // not found
8 }
```

VARIABLES

`x = 52` `L = &node(52)`

“When a manager gives a clerk a task, it must be defined precisely. The clerk should never have to come back and knock on the door.”

Recursive Search: search(34, L)



x = 34

34 < 47: `return search(x, L→left)` — whatever it returns is passed up **unchanged**.

Step 1 / 5

```
1 tree_ptr search(int x, tree_ptr L) {
2   if (!L) return NULL;          // absent
3   if (x == L->data) return L;   // found
4   if (x < L->data)
▶     return search(x, L->left);
6   return search(x, L->right);
7 }
```

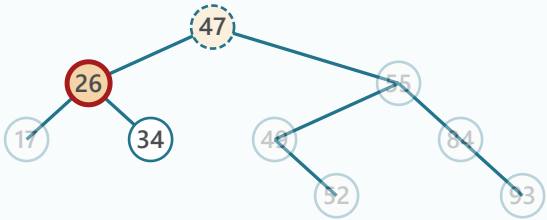
VARIABLES

x = 34 | L = &node(47)

CALL STACK

search(34, 47)

Recursive Search: search(34, L)



x = 34

34 > 26: `return search(x, L→right)`.

Step 2 / 5

```
1 tree_ptr search(int x, tree_ptr L) {
2   if (!L) return NULL;          // absent
3   if (x == L->data) return L;   // found
4   if (x < L->data)
5     return search(x, L->left);
▶   return search(x, L->right);
7 }
```

VARIABLES

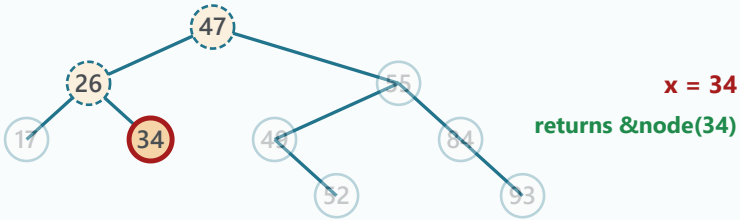
x = 34 | L = &node(26)

CALL STACK

search(34, 26)

search(34, 47)

Recursive Search: search(34, L)



34 == 34: found; this frame returns L.

Step 3 / 5

```
1 tree_ptr search(int x, tree_ptr L) {
2   if (!L) return NULL;          // absent
▶  if (x == L->data) return L;    // found
4   if (x < L->data)
5     return search(x, L->left);
6   return search(x, L->right);
7 }
```

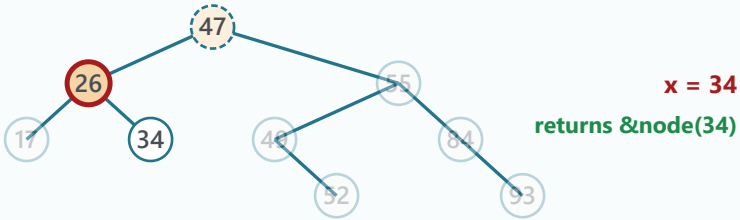
VARIABLES

x = 34 | L = &node(34)

CALL STACK

- search(34, 34)
- search(34, 26)
- search(34, 47)

Recursive Search: search(34, L)



The call returned &node(34); search(34, 26) returns it too.

Step 4 / 5

```
1 tree_ptr search(int x, tree_ptr L) {
2   if (!L) return NULL;          // absent
3   if (x == L->data) return L;   // found
4   if (x < L->data)
5     return search(x, L->left);
▶   return search(x, L->right);
7 }
```

VARIABLES

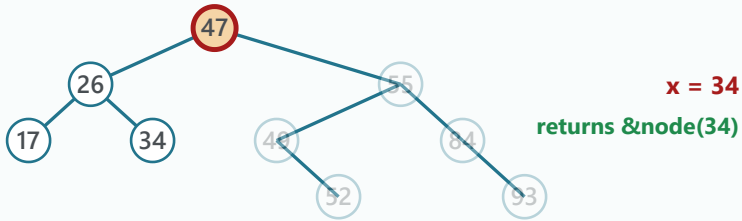
`x = 34` `L = &node(26)`

CALL STACK

`search(34, 26)`

`search(34, 47)`

Recursive Search: search(34, L)



The call returned &node(34); search(34, 47) returns it too.

Step 5 / 5

```
1 tree_ptr search(int x, tree_ptr L) {
2   if (!L) return NULL;          // absent
3   if (x == L->data) return L;   // found
4   if (x < L->data)
5     return search(x, L->left);
6   return search(x, L->right);
7 }
```

VARIABLES

x = 34 | L = &node(47)

CALL STACK

search(34, 47)

Example: “Does This Work?”

```
if (L) {
  if (x == L->data) return L;
  if (x <  L->data) return search(x, L->left);
  return search(x, L->right);
}
return NULL;    /* L == NULL: empty (sub)tree */
```

Once the `if` on line 3 is taken, the right-subtree search on line 4 **never** runs: the call returns whatever the subcall returns.

- **In a binary search tree** that is exactly what we want: if x is smaller it cannot be on the right; the code is correct and O(h).
- **In a general, unordered binary tree** x may be in the right subtree; this code misses it.
- For a general tree the fix searches both sides → O(n).

INSTRUCTOR “When writing recursive code, people put work after the return line, as in this example.”

Sample Solution: Searching Any Binary Tree

```
tree_ptr search_rec(int data, tree_ptr L) {
  if (!L) return NULL;              /* empty tree?   */
  if (L->data == data) return L;    /* key found     */
  tree_ptr T = search_rec(data, L->left);
  return T ? T : search_rec(data, L->right);
}
```

- Search the left first; if not found there (T == NULL), search the right.
- The ordering is not used → O(n) worst case; needlessly slow on a BST.
- The solution to the find-weight exam question uses this routine.

INSTRUCTOR "Everyone here should be able to write these routines with their eyes closed! Fine, one eye open ☺"

QUESTION: Write depth Iteratively

INSTRUCTOR "Iterative code is for beginners! ... If you write it this way in the exam, it gets no credit even if it works. Why? Because it uses global variables. — 'But if it works, won't you give even 1 point?' — No, not even 0.25!"

```
/* iterative, no globals:
   walk level by level, count levels */
int depth_iter(tree_ptr L) {
  tree_ptr k[MAX_SIZE];
  int front = 0, rear = 0, d = 0;
  if (L) k[rear++] = L;
  while (front < rear) {          /* one level */
    int last = rear;
    d++;
    for (; front < last; front++) {
      if (k[front]->left) k[rear++] = k[front]->left;
      if (k[front]->right) k[rear++] = k[front]->right;
    }
  }
  return d;       /* needs MAX_SIZE >= n */
}
```

DEPTH

Why No Global Variables?

INSTRUCTOR “Big companies build big systems by integrating work. A routine written by one person is used by someone at the other end. They don't know each other; he's all the way in India! He can't just call and ask ‘You have a max variable, what is it for?’”

A routine must be self-contained

Input through parameters, output through the return value; no hidden state.

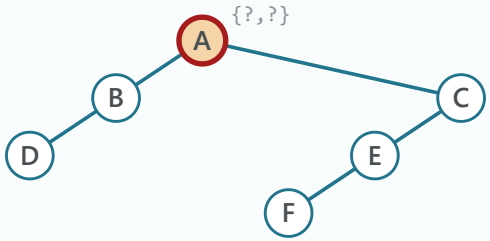
Local prototypes are fine

A routine may declare another routine's prototype locally and call it; the definition stays outside (standard C has no nested functions).

The recursive solution

Each call has its own local left/right variables; no global counter is needed.

Recursive depth: {Left, Right} Frames



running now
on the stack {left,right}
=returned

L = A: first the depth of the left subtree: call `depth(B)`.

Step 1 / 11

```
1 int depth(tree_ptr L) {
2   int left, right;
3   if (L) {                    // pointer NULL?
▶     left = depth(L->left);
5     right = depth(L->right);
6     return MAX(left, right) + 1;
7   }
8   else return 0;              // reached past a leaf
9 }
```

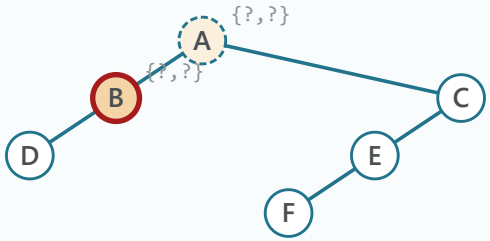
VARIABLES

`L = &node(A)` `left = ?` `right = ?`

CALL STACK

`depth(A)`

Recursive depth: {Left, Right} Frames



running now
on the stack {left,right}
=returned

L = B: first the depth of the left subtree: call `depth(D)`.

Step 2 / 11

```
1 int depth(tree_ptr L) {
2   int left, right;
3   if (L) {                    // pointer NULL?
▶     left = depth(L->left);
5     right = depth(L->right);
6     return MAX(left, right) + 1;
7   }
8   else return 0;              // reached past a leaf
9 }
```

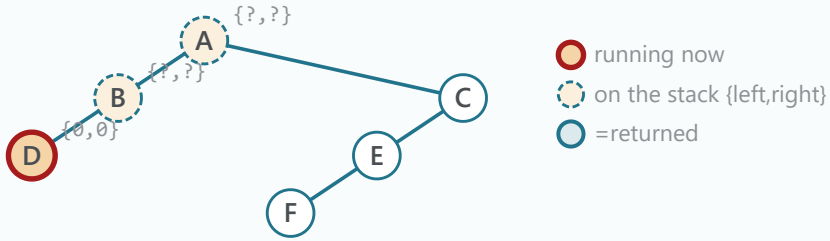
VARIABLES

`L = &node(B)` `left = ?` `right = ?`

CALL STACK

`depth(B)`
`depth(A)`

Recursive depth: {Left, Right} Frames



L = D: left = 0, right = 0 (NULL → 0) → MAX + 1 = **1** returned; locals left/right are freed.

Step 3 / 11

```
1 int depth(tree_ptr L) {
2   int left, right;
3   if (L) {                    // pointer NULL?
4     left = depth(L->left);
5     right = depth(L->right);
▶     return MAX(left, right) + 1;
7   }
8   else return 0;              // reached past a leaf
9 }
```

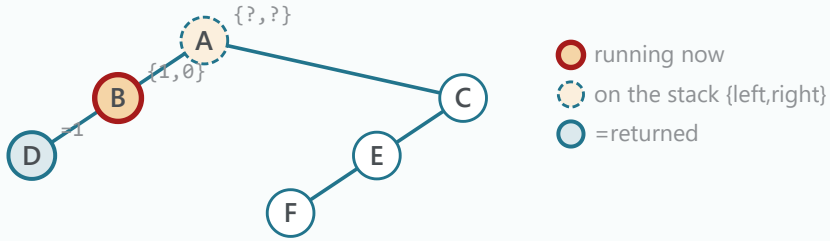
VARIABLES

`L = &node(D)` `left = 0` `right = 0`

CALL STACK

- `depth(D)`
- `depth(B)`
- `depth(A)`

Recursive depth: {Left, Right} Frames



L = B: left = 1, right = 0 (NULL → 0) → MAX + 1 = **2** returned; locals left/right are freed.

Step 4 / 11

```
1 int depth(tree_ptr L) {
2   int left, right;
3   if (L) {                    // pointer NULL?
4     left = depth(L->left);
5     right = depth(L->right);
▶     return MAX(left, right) + 1;
7   }
8   else return 0;              // reached past a leaf
9 }
```

VARIABLES

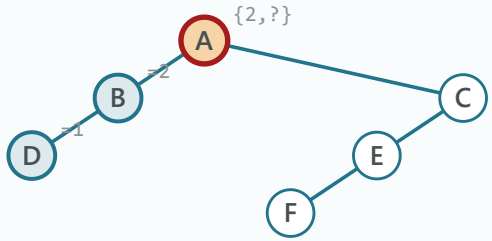
`L = &node(B)` `left = 1` `right = 0`

CALL STACK

`depth(B)`

`depth(A)`

Recursive depth: {Left, Right} Frames



running now
on the stack {left,right}
=returned

left = 2; now `depth(C)` .

Step 5 / 11

```
1 int depth(tree_ptr L) {
2   int left, right;
3   if (L) {                    // pointer NULL?
4     left = depth(L->left);
▶     right = depth(L->right);
6     return MAX(left, right) + 1;
7   }
8   else return 0;              // reached past a leaf
9 }
```

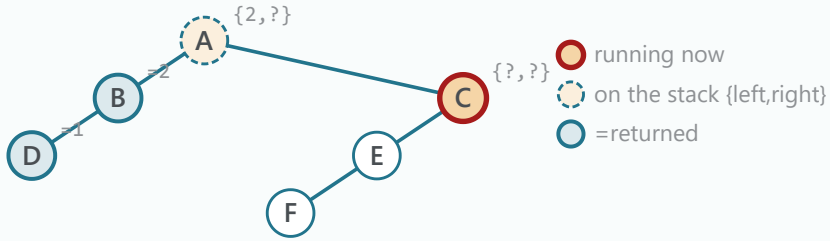
VARIABLES

`L = &node(A)` `left = 2`
`right = ?`

CALL STACK

`depth(A)`

Recursive depth: {Left, Right} Frames



L = C: first the depth of the left subtree: call `depth(E)`.

Step 6 / 11

```
1 int depth(tree_ptr L) {
2   int left, right;
3   if (L) {                    // pointer NULL?
▶     left = depth(L->left);
5     right = depth(L->right);
6     return MAX(left, right) + 1;
7   }
8   else return 0;              // reached past a leaf
9 }
```

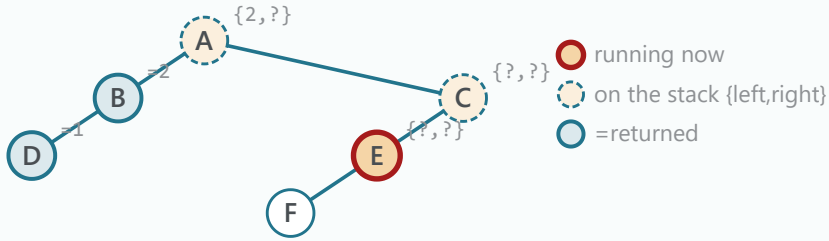
VARIABLES

`L = &node(C)` `left = ?` `right = ?`

CALL STACK

`depth(C)`
`depth(A)`

Recursive depth: {Left, Right} Frames



L = E: first the depth of the left subtree: call `depth(F)`.

Step 7 / 11

```
1 int depth(tree_ptr L) {
2   int left, right;
3   if (L) {                    // pointer NULL?
▶     left = depth(L->left);
5     right = depth(L->right);
6     return MAX(left, right) + 1;
7   }
8   else return 0;              // reached past a leaf
9 }
```

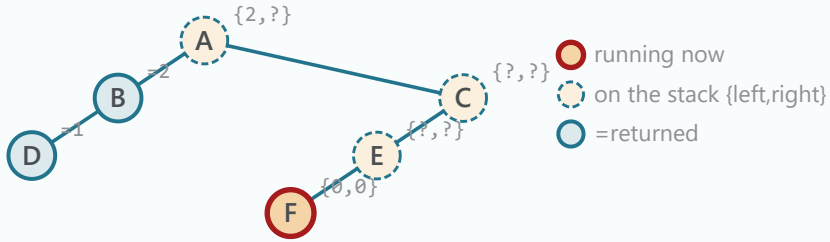
VARIABLES

`L = &node(E)` `left = ?` `right = ?`

CALL STACK

- `depth(E)`
- `depth(C)`
- `depth(A)`

Recursive depth: {Left, Right} Frames



L = F: left = 0, right = 0 (NULL → 0) → MAX + 1 = **1** returned; locals left/right are freed.

Step 8 / 11

```
1 int depth(tree_ptr L) {
2   int left, right;
3   if (L) {                    // pointer NULL?
4     left = depth(L->left);
5     right = depth(L->right);
▶     return MAX(left, right) + 1;
7   }
8   else return 0;              // reached past a leaf
9 }
```

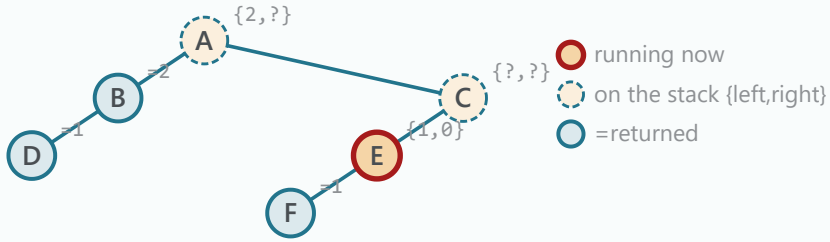
VARIABLES

`L = &node(F)` `left = 0` `right = 0`

CALL STACK

- `depth(F)`
- `depth(E)`
- `depth(C)`
- `depth(A)`

Recursive depth: {Left, Right} Frames



L = E: left = 1, right = 0 (NULL → 0) → MAX + 1 = **2** returned; locals left/right are freed.

Step 9 / 11

```
1 int depth(tree_ptr L) {
2   int left, right;
3   if (L) {                    // pointer NULL?
4     left = depth(L->left);
5     right = depth(L->right);
▶     return MAX(left, right) + 1;
7   }
8   else return 0;              // reached past a leaf
9 }
```

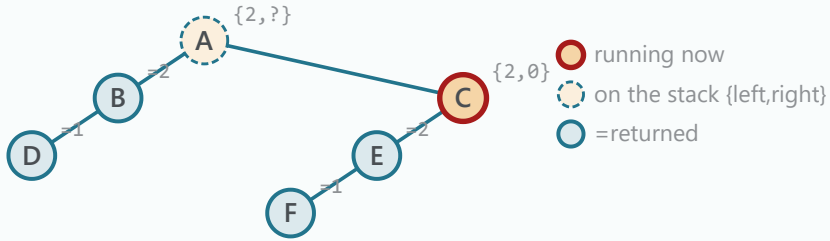
VARIABLES

`L = &node(E)` `left = 1` `right = 0`

CALL STACK

- **depth(E)**
- depth(C)
- depth(A)

Recursive depth: {Left, Right} Frames



L = C: left = 2, right = 0 (NULL → 0) → MAX + 1 = **3** returned; locals left/right are freed.

Step 10 / 11

```
1 int depth(tree_ptr L) {
2   int left, right;
3   if (L) {                    // pointer NULL?
4     left = depth(L->left);
5     right = depth(L->right);
▶     return MAX(left, right) + 1;
7   }
8   else return 0;              // reached past a leaf
9 }
```

VARIABLES

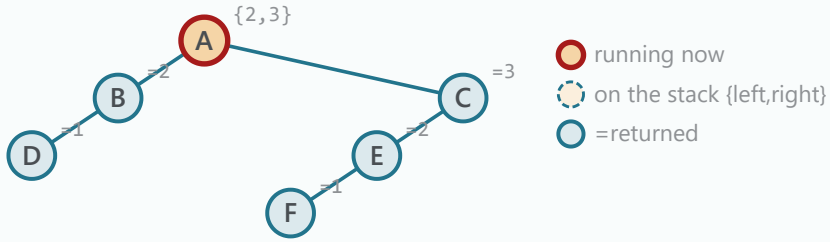
`L = &node(C)` `left = 2` `right = 0`

CALL STACK

`depth(C)`

`depth(A)`

Recursive depth: {Left, Right} Frames



L = A: left = 2, right = 3 → MAX + 1 = **4** returned; locals left/right are freed.

Step 11 / 11

```
1 int depth(tree_ptr L) {
2   int left, right;
3   if (L) {                    // pointer NULL?
4     left = depth(L->left);
5     right = depth(L->right);
▶     return MAX(left, right) + 1;
7   }
8   else return 0;              // reached past a leaf
9 }
```

VARIABLES

`L = &node(A)` `left = 2` `right = 3`

CALL STACK

`depth(A)`

Depth: Is the Problem Small Enough Yet?

- The depth of a tree is **one more** than the deeper of its left and right subtrees: `MAX(left, right) + 1`.
- Each node has its own 2 locals: the **left** and **right** depth. Calls made with NULL free their locals without using them.
- Time O(n), space O(h). `#define MAX(a,b) ((a) > (b) ? (a) : (b))`

```
int depth(tree_ptr L) {    /* one line */
  return !L ? 0 : MAX(depth(L->left),
                      depth(L->right)) + 1;
}
```

The `MAX` macro evaluates its arguments twice; in this one-liner the recursive calls run twice → exponential time! So use a `max()` function instead of the macro (as in the exam solution).

INSTRUCTOR "Is the problem small enough yet? NULL may come in right at the start!"

INSTRUCTOR "These two approaches (thinking recursively, solving recurrences) are hard to sell to our engineer friends. The man has his way of thinking, and look how nicely he codes!"

On Adding Extra Parameters

> **“If you have a procedure with 10 parameters, you probably missed some.”**
>
> — Alan J. Perlis

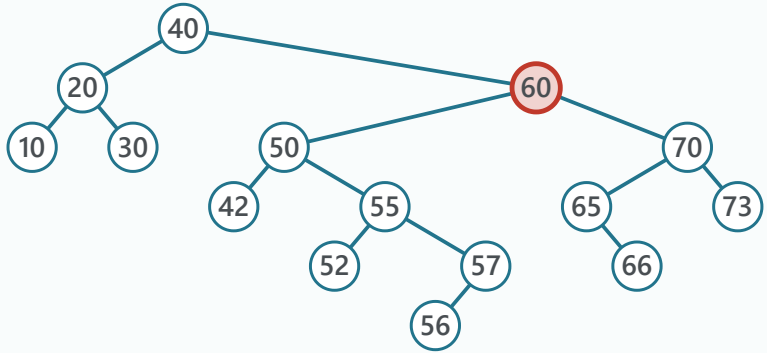
Question: “Sir, can we add an extra parameter, `int depth(tree_ptr L, int a)`, and call it as `depth(L, 0)`?”

INSTRUCTOR “Why define an extra parameter and pass zero on the first call? What do you tell the user of the routine? ‘Whatever you do, don't forget to pass zero for a!’ A sort routine that sorts when you pass 0 but fails when you pass 1… ☺ Does that make sense?”

But this is fine: a parameter that says whether to sort ascending or descending (−1/+1, '<'/'>') or gives the type of the data. A parameter should **select behavior**, not carry a hidden initial value.

Deleting from a Binary Search Tree

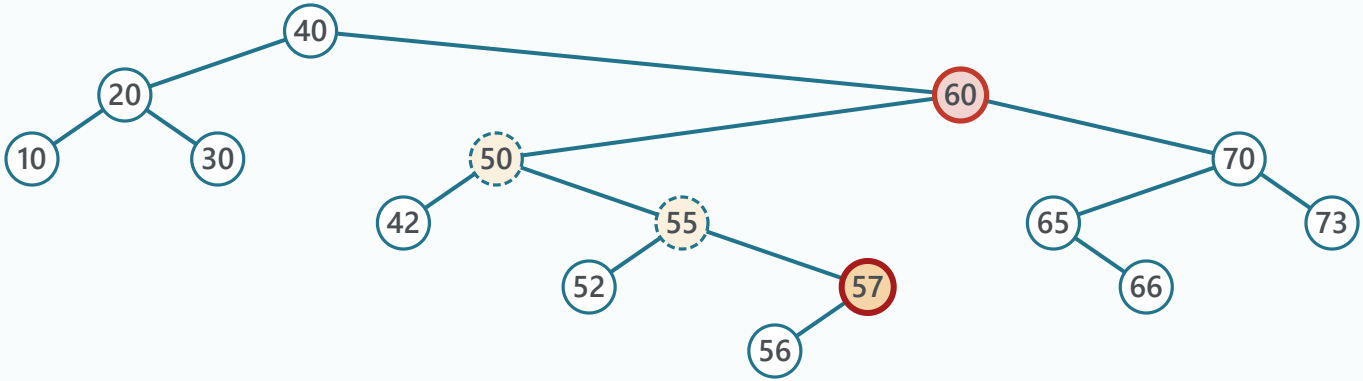
- Deletion must preserve the **binary search tree property**.
- **No children** (leaf): set the parent's pointer to NULL.
- **One child**: the parent's pointer now points to that only child.
- **Two children**: there are several methods; let's delete **60** from the sample tree.



node to delete: 60

Sample tree: built by inserting 40, 20, 60, 10, 30, 50, 70, 42, 55, 65, 73, 52, 57, 66, 56 in that order (depth 6).

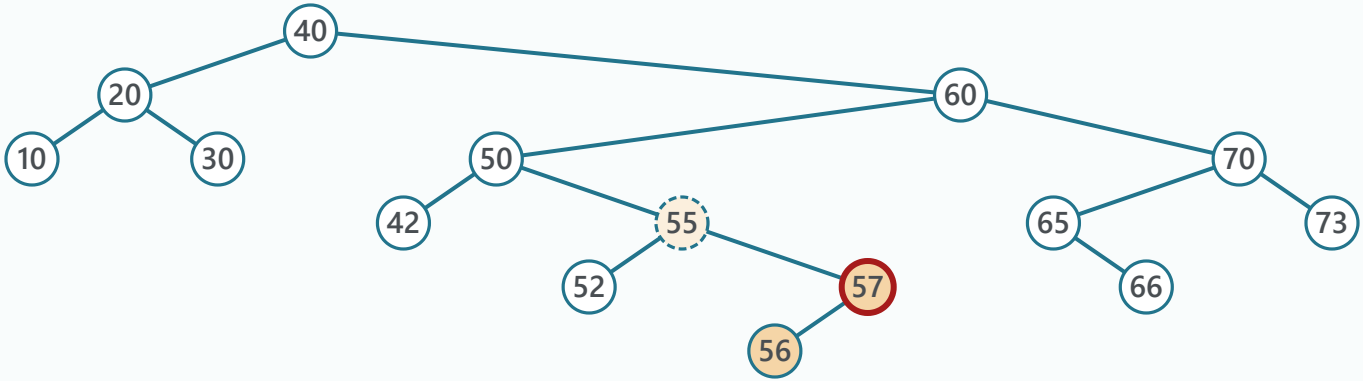
Method 1: The Largest Key in the Left Subtree



in the left subtree keep going right: 50 → 55 → 57

The largest key in the left subtree: follow the **right** pointers inside it; the node with no right child (57) is the largest.

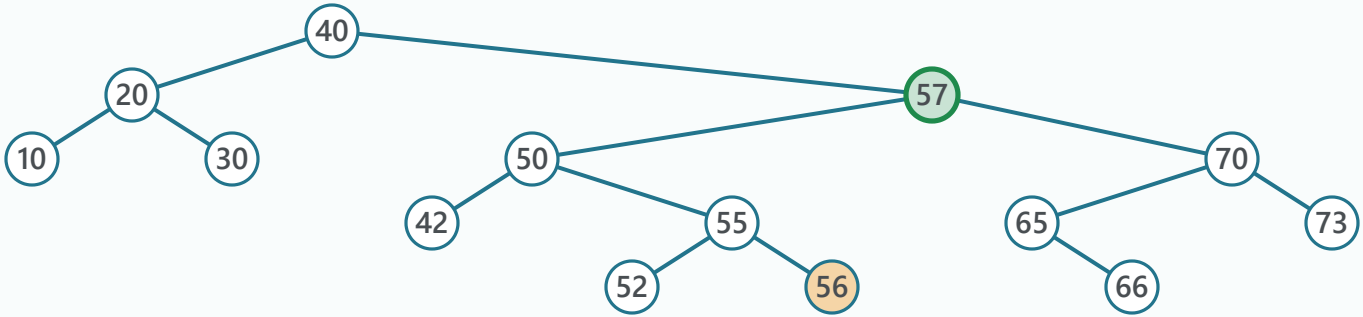
Method 1: The Largest Key in the Left Subtree



57 has a left child, 56!

That node may have a left subtree (57 has 56). This left child must move to the **right** link of the moved node's parent (55 → 56).

Method 1: The Largest Key in the Left Subtree

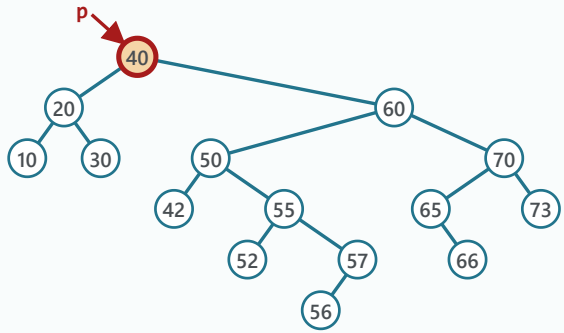


copy the data: 1 link updated · depth 6 → 5

Second way: 60 is not freed; 57 is written into its data field and only 55's right link is set to 56 → **1 link**. The **first way** (moving the node) needs 4 links: 40→57, 55→56, 57.left→50, 57.right→70.

bst_delete(60, &L): A Node with Two Children

```
1  void bst_delete(int x, tree_ptr *L) {
▶    tree_ptr p = *L, *link = L;  // link to p
3    while (p && p->data != x) {  // find x
4      link = (x < p->data) ? &p->left
5                           : &p->right;
6      p = *link;
7    }
8    if (!p) return;              // x absent
9    if (p->left && p->right) {   // 2 children
10     tree_ptr *b = &p->left;     // left subtree's
11     while ((*b)->right)           // largest
12       b = &(*b)->right;           // node
13     p->data = (*b)->data;       // move the data
14     link = b;  p = *b;           // delete it
15   }
16   *link = p->left ? p->left : p->right;
17   free(p);
18 }
```



p is the root; link holds the **address** of the root pointer, so we can change the link leading to the node we delete.

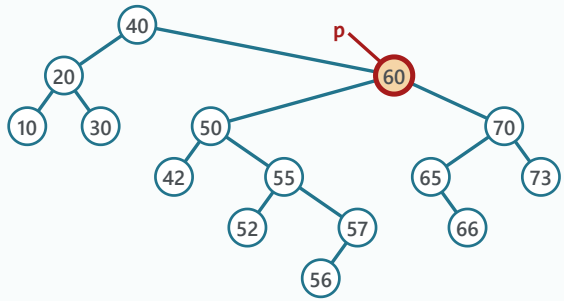
Step 1 / 11

VARIABLES

`x = 60` `p = &node(40)` `link = &L (root)`

bst_delete(60, &L): A Node with Two Children

```
1  void bst_delete(int x, tree_ptr *L) {
2    tree_ptr p = *L, *link = L;  // link to p
3    while (p && p->data != x) {  // find x
▶      link = (x < p->data) ? &p->left
▶                           : &p->right;
▶      p = *link;
7    }
8    if (!p) return;              // x absent
9    if (p->left && p->right) {  // 2 children
10     tree_ptr *b = &p->left;     // left subtree's
11     while ((*b)->right)           // largest
12       b = &(*b)->right;           // node
13     p->data = (*b)->data;       // move the data
14     link = b;  p = *b;           // delete it
15   }
16   *link = p->left ? p->left : p->right;
17   free(p);
18 }
```



60 > 40 → link = &40->right, p = 60.

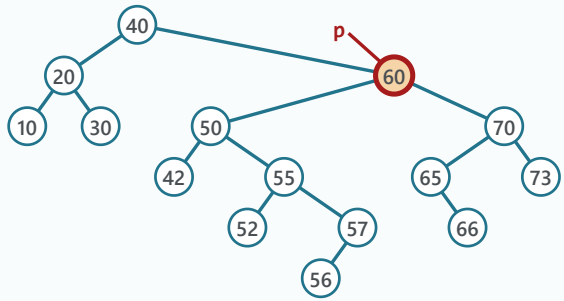
Step 2 / 11

VARIABLES

x = 60 | p = &node(60) | link = &40→right

bst_delete(60, &L): A Node with Two Children

```
1  void bst_delete(int x, tree_ptr *L) {
2    tree_ptr p = *L, *link = L;  // link to p
3    while (p && p->data != x) {  // find x
4      link = (x < p->data) ? &p->left
5                           : &p->right;
6      p = *link;
7    }
8    if (!p) return;              // x absent
▶    if (p->left && p->right) {  // 2 children
10     tree_ptr *b = &p->left;     // left subtree's
11     while ((*b)->right)           // largest
12       b = &(*b)->right;           // node
13     p->data = (*b)->data;       // move the data
14     link = b;  p = *b;           // delete it
15   }
16   *link = p->left ? p->left : p->right;
17   free(p);
18 }
```



Found 60, and it has **two children** (50, 70) → find the largest key in the left subtree.

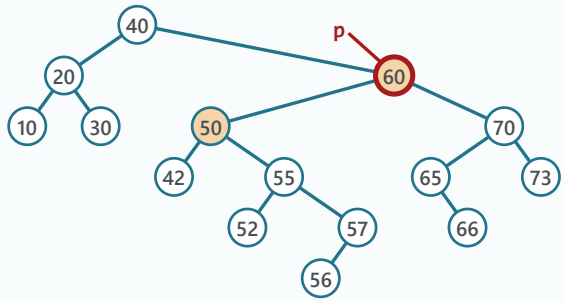
Step 3 / 11

VARIABLES

`x = 60` `p = &node(60)` `link = &40→right`

bst_delete(60, &L): A Node with Two Children

```
1  void bst_delete(int x, tree_ptr *L) {
2    tree_ptr p = *L, *link = L;  // link to p
3    while (p && p->data != x) {  // find x
4      link = (x < p->data) ? &p->left
5                           : &p->right;
6      p = *link;
7    }
8    if (!p) return;              // x absent
9    if (p->left && p->right) {   // 2 children
▶      tree_ptr *b = &p->left;     // left subtree's
11     while ((*b)->right)           // largest
12       b = &(*b)->right;           // node
13     p->data = (*b)->data;       // move the data
14     link = b;  p = *b;           // delete it
15   }
16   *link = p->left ? p->left : p->right;
17   free(p);
18 }
```



b = &60->left → *b = 50.

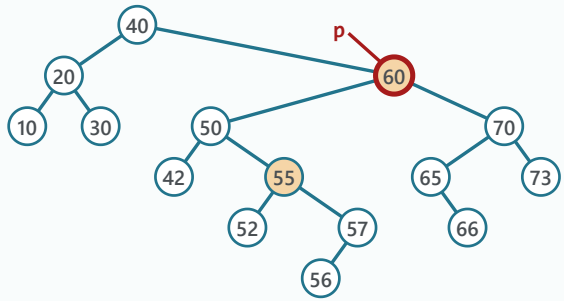
Step 4 / 11

VARIABLES

`x = 60` `p = &node(60)` `link = &40→right`

bst_delete(60, &L): A Node with Two Children

```
1  void bst_delete(int x, tree_ptr *L) {
2    tree_ptr p = *L, *link = L;  // link to p
3    while (p && p->data != x) {  // find x
4      link = (x < p->data) ? &p->left
5                           : &p->right;
6      p = *link;
7    }
8    if (!p) return;              // x absent
9    if (p->left && p->right) {   // 2 children
10     tree_ptr *b = &p->left;     // left subtree's
▶      while ((*b)->right)          // largest
▶        b = &(*b)->right;          // node
13     p->data = (*b)->data;       // move the data
14     link = b;  p = *b;          // delete it
15   }
16   *link = p->left ? p->left : p->right;
17   free(p);
18 }
```



*b has a right child → b = &50->right, *b = 55.

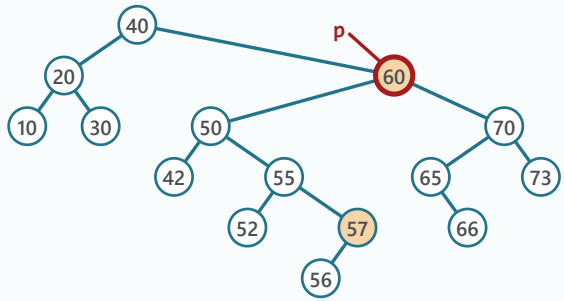
Step 5 / 11

VARIABLES

x = 60 | p = &node(60) | link = &40→right

bst_delete(60, &L): A Node with Two Children

```
1  void bst_delete(int x, tree_ptr *L) {
2    tree_ptr p = *L, *link = L;  // link to p
3    while (p && p->data != x) {  // find x
4      link = (x < p->data) ? &p->left
5                           : &p->right;
6      p = *link;
7    }
8    if (!p) return;              // x absent
9    if (p->left && p->right) {   // 2 children
10     tree_ptr *b = &p->left;     // left subtree's
▶      while ((*b)->right)          // largest
▶        b = &(*b)->right;          // node
13     p->data = (*b)->data;       // move the data
14     link = b;  p = *b;          // delete it
15   }
16   *link = p->left ? p->left : p->right;
17   free(p);
18 }
```



*b has a right child → b = &55->right, *b = 57.

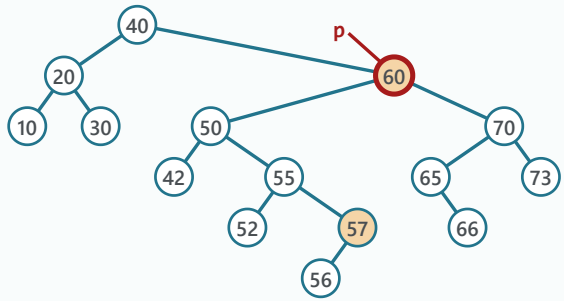
Step 6 / 11

VARIABLES

x = 60 | p = &node(60) | link = &40→right

bst_delete(60, &L): A Node with Two Children

```
1  void bst_delete(int x, tree_ptr *L) {
2    tree_ptr p = *L, *link = L;  // link to p
3    while (p && p->data != x) {  // find x
4      link = (x < p->data) ? &p->left
5                           : &p->right;
6      p = *link;
7    }
8    if (!p) return;              // x absent
9    if (p->left && p->right) {   // 2 children
10     tree_ptr *b = &p->left;     // left subtree's
▶      while ((*b)->right)          // largest
12       b = &(*b)->right;          // node
13     p->data = (*b)->data;       // move the data
14     link = b;  p = *b;          // delete it
15   }
16   *link = p->left ? p->left : p->right;
17   free(p);
18 }
```



57 has no right child: the largest key in the left subtree is **57**.

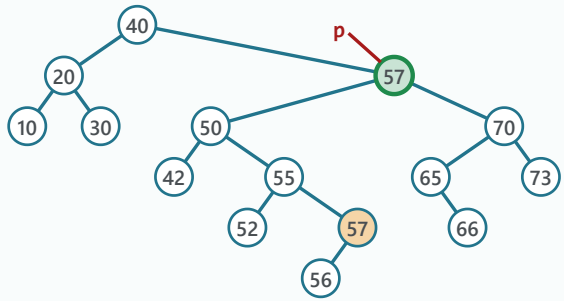
Step 7 / 11

VARIABLES

`x = 60` `p = &node(60)` `link = &40→right`

bst_delete(60, &L): A Node with Two Children

```
1  void bst_delete(int x, tree_ptr *L) {
2    tree_ptr p = *L, *link = L;  // link to p
3    while (p && p->data != x) {  // find x
4      link = (x < p->data) ? &p->left
5                           : &p->right;
6      p = *link;
7    }
8    if (!p) return;              // x absent
9    if (p->left && p->right) {   // 2 children
10     tree_ptr *b = &p->left;     // left subtree's
11     while ((*b)->right)           // largest
12       b = &(*b)->right;           // node
▶      p->data = (*b)->data;       // move the data
14     link = b;  p = *b;          // delete it
15   }
16   *link = p->left ? p->left : p->right;
17   free(p);
18 }
```



57 is written into 60's data field (the node does not move: method 1, second way).

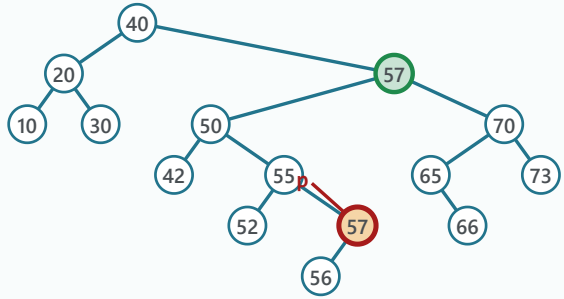
Step 8 / 11

VARIABLES

x = 60 | p = &node(57) | link = &40→right

bst_delete(60, &L): A Node with Two Children

```
1  void bst_delete(int x, tree_ptr *L) {
2    tree_ptr p = *L, *link = L;  // link to p
3    while (p && p->data != x) {  // find x
4      link = (x < p->data) ? &p->left
5                           : &p->right;
6      p = *link;
7    }
8    if (!p) return;              // x absent
9    if (p->left && p->right) {   // 2 children
10     tree_ptr *b = &p->left;     // left subtree's
11     while ((*b)->right)           // largest
12       b = &(*b)->right;           // node
13     p->data = (*b)->data;       // move the data
▶      link = b;  p = *b;          // delete it
15   }
16   *link = p->left ? p->left : p->right;
17   free(p);
18 }
```



Now the old 57 node is the one to delete; link = &55->right.

Step 9 / 11

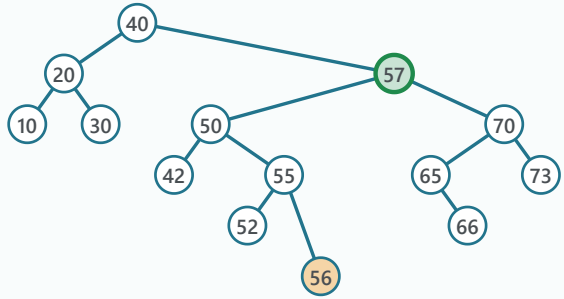
VARIABLES

`x = 60` `p = &node(57)`

`link = &55→right`

bst_delete(60, &L): A Node with Two Children

```
1  void bst_delete(int x, tree_ptr *L) {
2    tree_ptr p = *L, *link = L;  // link to p
3    while (p && p->data != x) {  // find x
4      link = (x < p->data) ? &p->left
5                           : &p->right;
6      p = *link;
7    }
8    if (!p) return;              // x absent
9    if (p->left && p->right) {   // 2 children
10     tree_ptr *b = &p->left;     // left subtree's
11     while ((*b)->right)           // largest
12       b = &(*b)->right;           // node
13     p->data = (*b)->data;       // move the data
14     link = b;  p = *b;           // delete it
15   }
▶  *link = p->left ? p->left : p->right;
17   free(p);
18 }
```



*link = 57's only child (56): 55's right link now points to 56.

Step 10 / 11

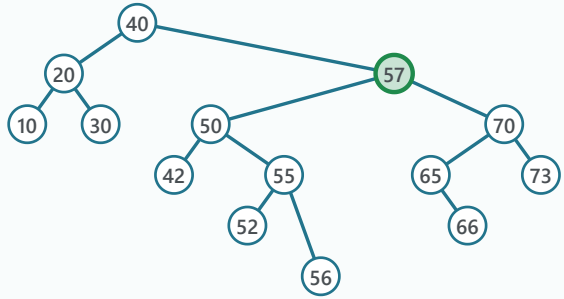
VARIABLES

x = 60 | p = NULL

link = &55→right

bst_delete(60, &L): A Node with Two Children

```
1  void bst_delete(int x, tree_ptr *L) {
2    tree_ptr p = *L, *link = L;  // link to p
3    while (p && p->data != x) {  // find x
4      link = (x < p->data) ? &p->left
5                           : &p->right;
6      p = *link;
7    }
8    if (!p) return;              // x absent
9    if (p->left && p->right) {   // 2 children
10     tree_ptr *b = &p->left;     // left subtree's
11     while ((*b)->right)           // largest
12       b = &(*b)->right;           // node
13     p->data = (*b)->data;       // move the data
14     link = b;  p = *b;           // delete it
15   }
16   *link = p->left ? p->left : p->right;
▶    free(p);
18 }
```



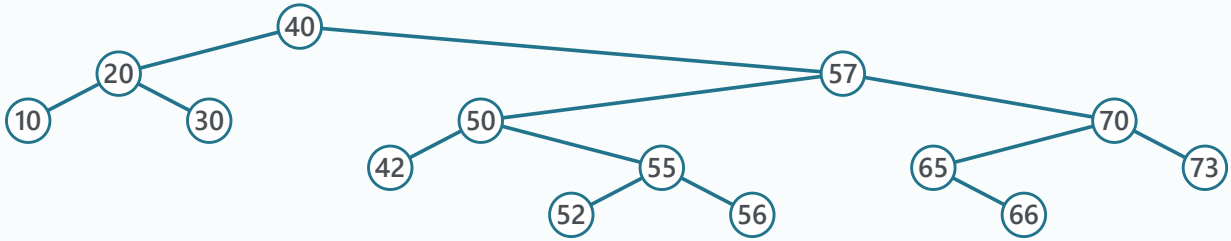
The old node is `free`d. Still a binary search tree; depth 6 → 5.

Step 11 / 11

VARIABLES

`x = 60` `p = NULL` `link = –`

Four Methods and Their Effect on depth



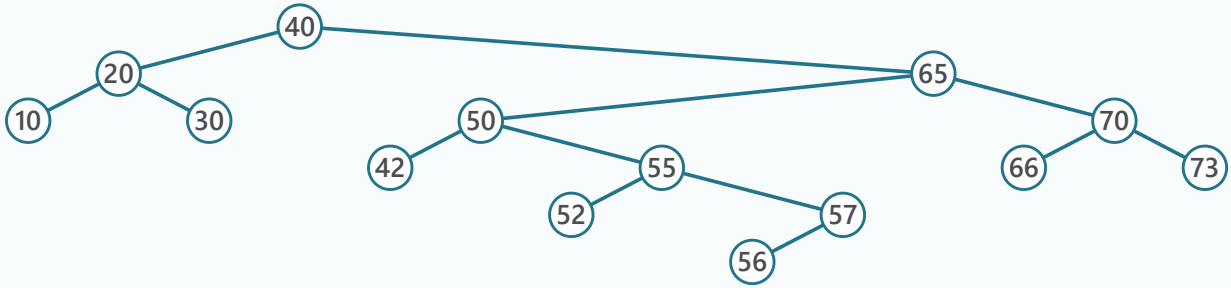
method 1 (largest in left subtree): depth 6 → 5

Method 1: 57 moves up, 56 goes right of 55. Depth drops to 5.

Step 1 / 4

INSTRUCTOR “Delete in a way that keeps the subtree rooted at the deleted node as balanced as possible.”

Four Methods and Their Effect on depth

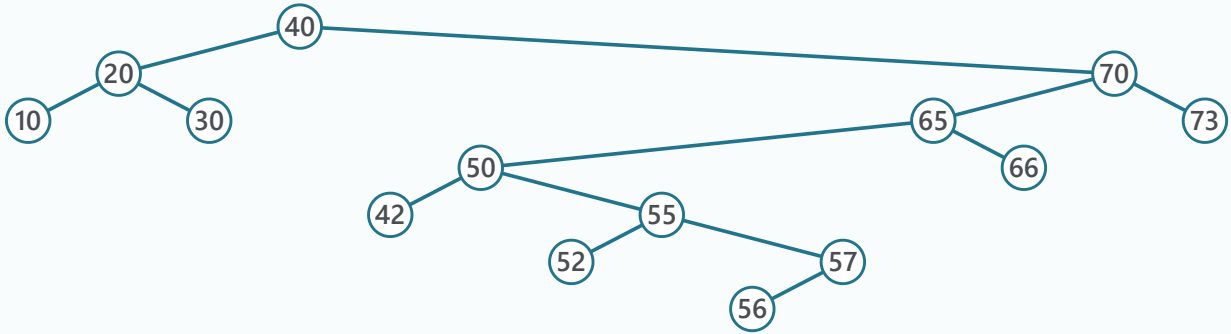


method 2 (smallest in right subtree): depth 6 → 6

Method 2: the smallest key of the right subtree, 65, moves up; its right child 66 hangs left of 70. Depth 6.

INSTRUCTOR “Delete in a way that keeps the subtree rooted at the deleted node as balanced as possible.”

Four Methods and Their Effect on depth

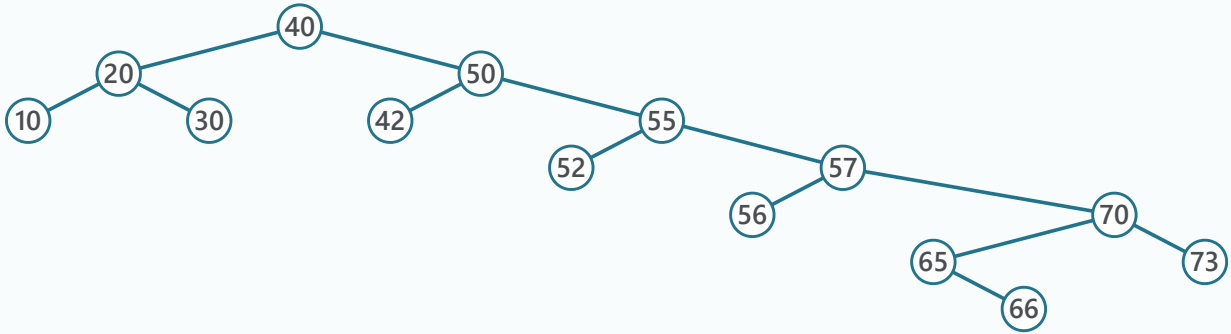


method 3 (right child moves up): depth 6 → 7

Method 3: the right child (70) takes the deleted node's place; subtree 50 hangs left of the smallest node of subtree 70 (65). Depth 7.

INSTRUCTOR “Delete in a way that keeps the subtree rooted at the deleted node as balanced as possible.”

Four Methods and Their Effect on depth



method 4 (left child moves up): depth 6 → 7

Method 4: the left child (50) takes its place; subtree 70 hangs right of the largest node of subtree 50 (57). Depth 7 (the path 40-50-55-57-70-65-66 has 7 nodes).

INSTRUCTOR “Delete in a way that keeps the subtree rooted at the deleted node as balanced as possible.”

The Cost of Imbalance

n	balanced: $\lceil \lg(n+1) \rceil$	skewed: n
1.000	10	1.000
1,000,000	20	1,000,000
2,000,000	21	2,000,000
10^9	30	10^9

- A BST built from n keys in random order has **average** depth Θ(lg n); an average search takes ≈ 1.39 lg n comparisons (Sedgewick; CLRS 12.4).
- But the **worst case** (sorted or nearly sorted input, which is common in practice) is Θ(n).
- The fix: trees that keep depth O(lg n) with local **rotations** after each insert or delete: AVL, red-black, B-tree.

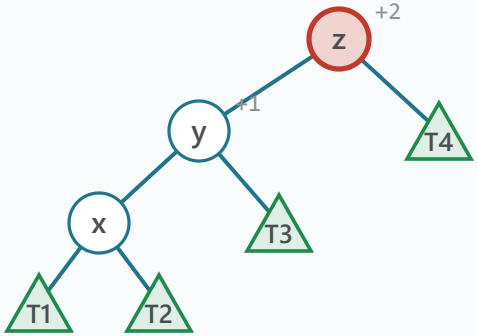
INSTRUCTOR "The gap between O(n) and O(lg n) is like the gap between 2,000,000 and 21."

AVL Tree (Adelson-Velsky and Landis, 1962)

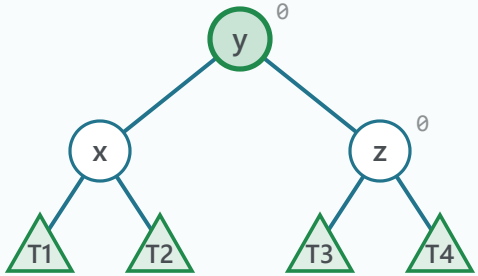
- At every node the **balance factor** = height(left) − height(right) ∈ {−1, 0, +1}.
- Height is at most ≈ 1.44 lg(n+2): search, insert and delete in **O(lg n)**.
- Insert as in an ordinary BST, then update the balance factors on the path back to the root.
- Find the **first** ancestor with factor ±2; which grandchild subtree got the new node gives 4 cases: **LL, RR, LR, RL**.
- After an insertion, a single (LL/RR) or double (LR/RL) rotation restores balance.

```
typedef struct avl *avl_ptr;
struct avl {
    int      data;
    int      ht;        /* subtree height */
    avl_ptr left, right;
};
int ht(avl_ptr t) { return t ? t->ht : 0; }
int balance(avl_ptr t) {
    return ht(t->left) - ht(t->right);
}
```

Single Rotation: LL and RR



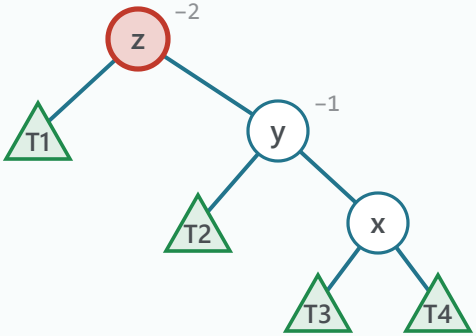
LL: z's left-left is heavy



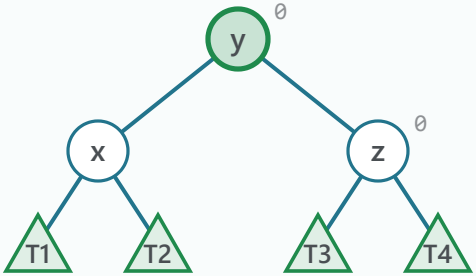
rotate right(z): y becomes the root

LL: the new node went into the left subtree of z's left child (y); z's factor is +2. Rotate z **right**: y moves up and y's right subtree T3 becomes z's left. Order is kept: T1 x T2 y T3 z T4.

Single Rotation: LL and RR



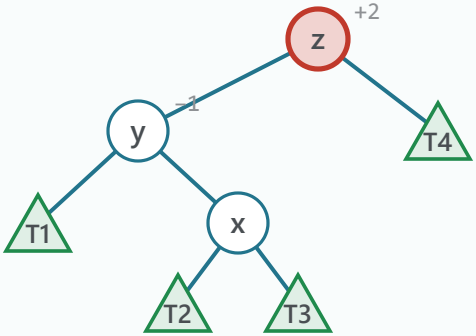
RR: z's right-right is heavy



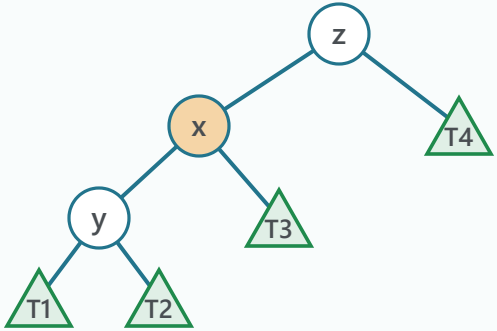
rotate left(z): y becomes the root

RR is the mirror case: rotate z **left**; y's left subtree (T2) becomes z's right. In both cases the subtree's height returns to its value before the insertion.

Double Rotation: LR and RL



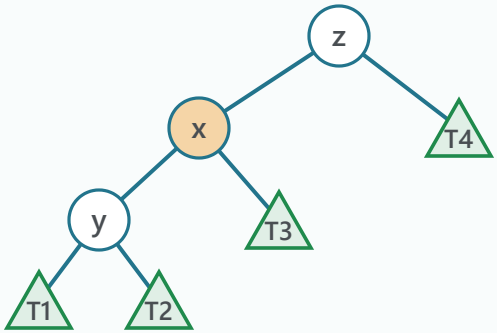
LR: z's left-right is heavy



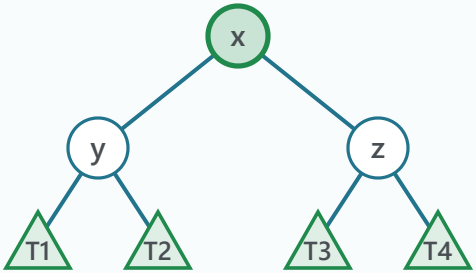
1) rotate left(y) → LL shape

LR: the new node is in the **right** subtree of z's left child (y). A single right rotation does not help; first rotate y **left** → LL shape.

Double Rotation: LR and RL



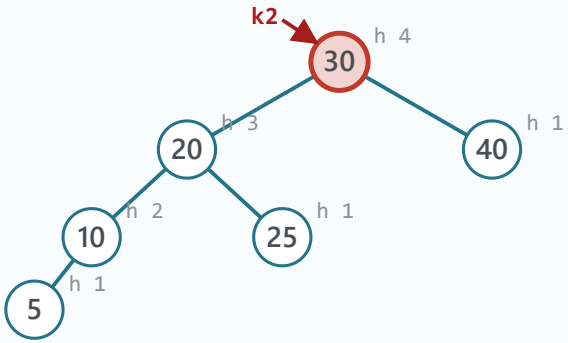
1) rotate left(y) → LL shape



2) rotate right(z): x is the root

Then rotate z **right**: grandchild x rises above both ancestors. **RL** is the mirror: first the right child to the right, then z to the left.

rot_right(): Two Pointers in the LL Case



Inserted 5: 30's factor is ht(20)=3 − ht(40)=1 = **+2**, left-left heavy → LL: `rot_right(30)`.

Step 1 / 7

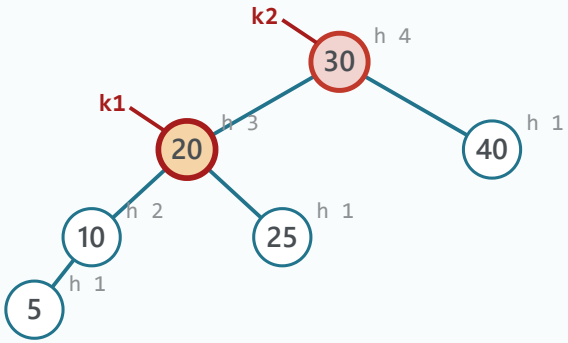
```
avl_ptr rot_right(avl_ptr k2) {  // LL
  avl_ptr k1 = k2->left;
  k2->left = k1->right;
  k1->right = k2;
  k2->ht = 1 + MAX(ht(k2->left), ht(k2->right));
  k1->ht = 1 + MAX(ht(k1->left), k2->ht);
  return k1;                        // new root
}
```

VARIABLES

`k2 = &node(30)` `k1 = ?`

`k2→ht = 4` `k1→ht = 3`

rot_right(): Two Pointers in the LL Case



k1 = k2->left = 20: the node that will be the new root.

Step 2 / 7

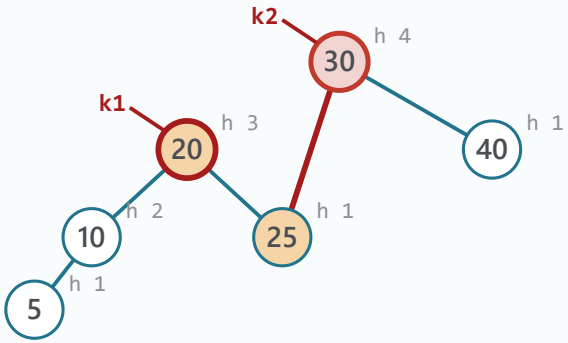
```
1 avl_ptr rot_right(avl_ptr k2) {  // LL
▶   avl_ptr k1 = k2->left;
3   k2->left = k1->right;
4   k1->right = k2;
5   k2->ht = 1 + MAX(ht(k2->left), ht(k2->right));
6   k1->ht = 1 + MAX(ht(k1->left), k2->ht);
7   return k1;                        // new root
8 }
```

VARIABLES

k2 = &node(30) k1 = &node(20)

k2→ht = 4 k1→ht = 3

rot_right(): Two Pointers in the LL Case



k2->left = k1->right: 25 moves to 30's left (20 < 25 < 30, order kept). For now two pointers point to 25.

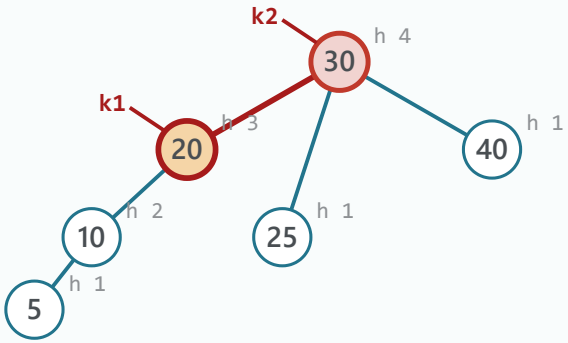
Step 3 / 7

```
1 avl_ptr rot_right(avl_ptr k2) {  // LL
2   avl_ptr k1 = k2->left;
▶   k2->left = k1->right;
4   k1->right = k2;
5   k2->ht = 1 + MAX(ht(k2->left), ht(k2->right));
6   k1->ht = 1 + MAX(ht(k1->left), k2->ht);
7   return k1;                      // new root
8 }
```

VARIABLES

- k2 = &node(30)
- k1 = &node(20)
- k2→ht = 4
- k1→ht = 3

rot_right(): Two Pointers in the LL Case



k1->right = k2: 30 becomes 20's right child. Links done; let's redraw →

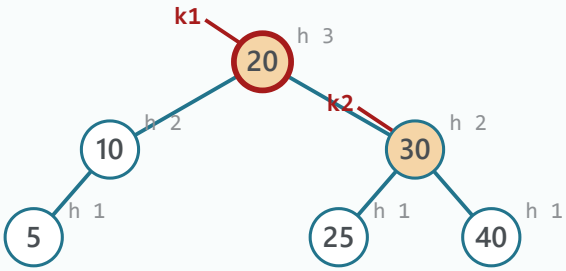
Step 4 / 7

```
1 avl_ptr rot_right(avl_ptr k2) {  // LL
2   avl_ptr k1 = k2->left;
3   k2->left = k1->right;
▶   k1->right = k2;
5   k2->ht = 1 + MAX(ht(k2->left), ht(k2->right));
6   k1->ht = 1 + MAX(ht(k1->left), k2->ht);
7   return k1;                        // new root
8 }
```

VARIABLES

`k2 = &node(30)` `k1 = &node(20)` `k2→ht = 4` `k1→ht = 3`

rot_right(): Two Pointers in the LL Case



Heights bottom-up: first k2 (30): 1 + MAX(1, 1) = **2**.

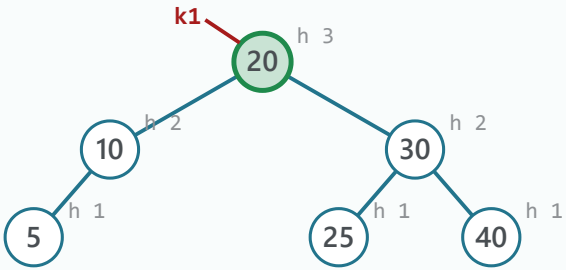
Step 5 / 7

```
1 avl_ptr rot_right(avl_ptr k2) {  // LL
2   avl_ptr k1 = k2->left;
3   k2->left = k1->right;
4   k1->right = k2;
▶   k2->ht = 1 + MAX(ht(k2->left), ht(k2->right));
6   k1->ht = 1 + MAX(ht(k1->left), k2->ht);
7   return k1;                        // new root
8 }
```

VARIABLES

`k2 = &node(30)` `k1 = &node(20)` `k2→ht = 2` `k1→ht = 3`

rot_right(): Two Pointers in the LL Case



Then k1 (20): 1 + MAX(ht(10)=2, 2) = **3**. The subtree's height is 3, as before the insertion.

Step 6 / 7

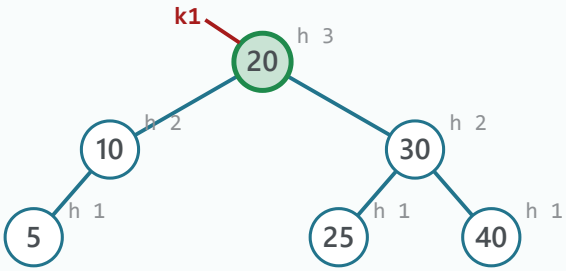
```
1 avl_ptr rot_right(avl_ptr k2) {  // LL
2   avl_ptr k1 = k2->left;
3   k2->left = k1->right;
4   k1->right = k2;
5   k2->ht = 1 + MAX(ht(k2->left), ht(k2->right));
▶   k1->ht = 1 + MAX(ht(k1->left), k2->ht);
7   return k1;                        // new root
8 }
```

VARIABLES

`k2 = &node(30)` `k1 = &node(20)`

`k2→ht = 2` `k1→ht = 3`

rot_right(): Two Pointers in the LL Case



The new root k1 is returned; the caller links the parent's pointer to it. Only 2 pointers changed: **O(1)**.

Step 7 / 7

```
1 avl_ptr rot_right(avl_ptr k2) {  // LL
2   avl_ptr k1 = k2->left;
3   k2->left = k1->right;
4   k1->right = k2;
5   k2->ht = 1 + MAX(ht(k2->left), ht(k2->right));
6   k1->ht = 1 + MAX(ht(k1->left), k2->ht);
▶   return k1;                        // new root
8 }
```

VARIABLES

`k2 = &node(30)` `k1 = &node(20)` `k2→ht = 2` `k1→ht = 3`

Inserting 3, 2, 1, 4, 5, 6, 7, 16, 15 into an AVL Tree



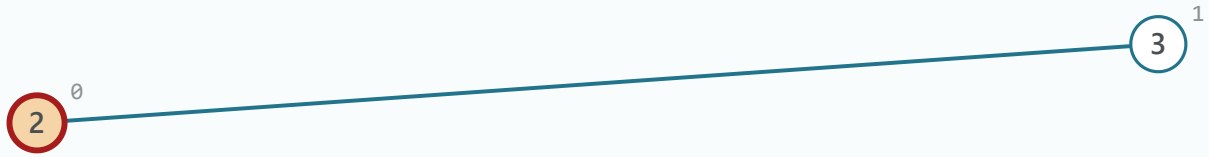
small number: balance factor

3 inserted. All factors in −1..+1: no rotation needed.

Step 1 / 9

Inserted into a plain BST, the same sequence would reach depth 7 because of the chain 3–4–5–6–7–16–15; the AVL tree has depth 4. (Example: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

Inserting 3, 2, 1, 4, 5, 6, 7, 16, 15 into an AVL Tree



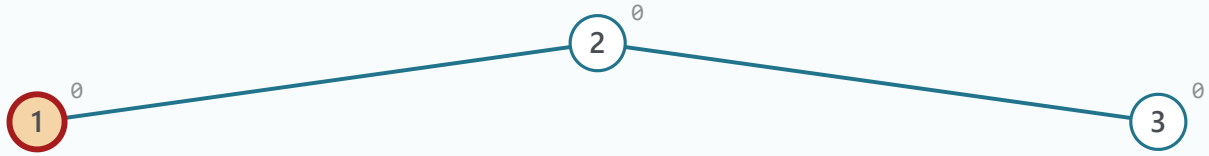
small number: balance factor

2 inserted. All factors in −1..+1: no rotation needed.

Step 2 / 9

Inserted into a plain BST, the same sequence would reach depth 7 because of the chain 3–4–5–6–7–16–15; the AVL tree has depth 4. (Example: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

Inserting 3, 2, 1, 4, 5, 6, 7, 16, 15 into an AVL Tree



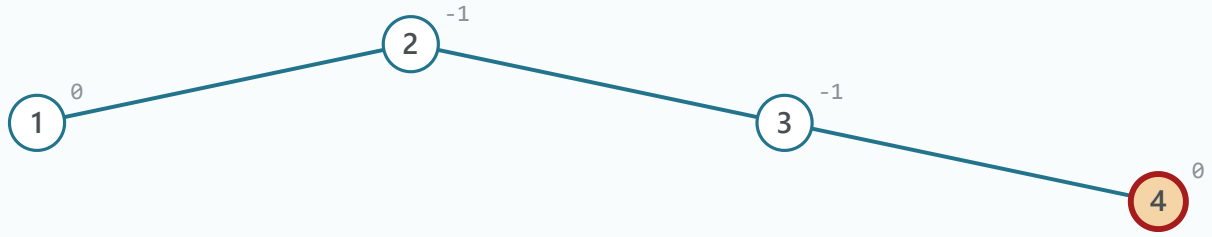
small number: balance factor

1 inserted. Imbalance → LL (rotate right at 3).

Step 3 / 9

Inserted into a plain BST, the same sequence would reach depth 7 because of the chain 3–4–5–6–7–16–15; the AVL tree has depth 4. (Example: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

Inserting 3, 2, 1, 4, 5, 6, 7, 16, 15 into an AVL Tree



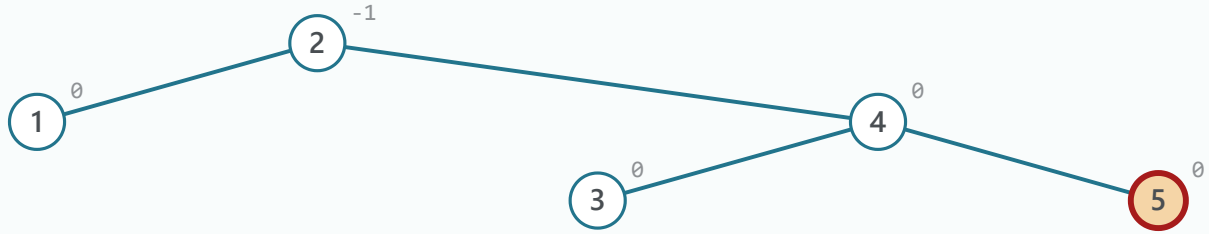
small number: balance factor

4 inserted. All factors in −1..+1: no rotation needed.

Step 4 / 9

Inserted into a plain BST, the same sequence would reach depth 7 because of the chain 3–4–5–6–7–16–15; the AVL tree has depth 4. (Example: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

Inserting 3, 2, 1, 4, 5, 6, 7, 16, 15 into an AVL Tree



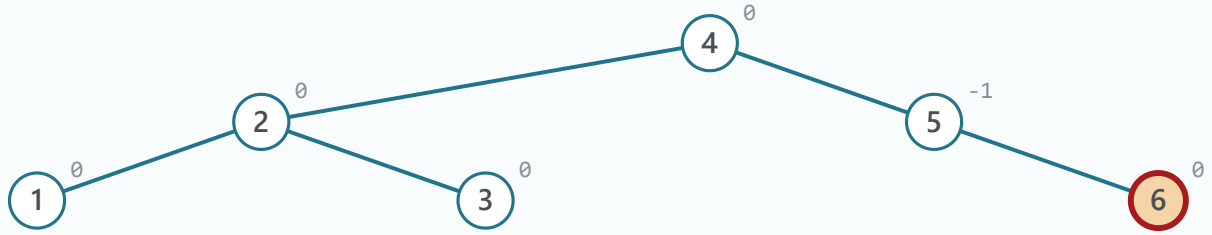
small number: balance factor

5 inserted. Imbalance → RR (rotate left at 3).

Step 5 / 9

Inserted into a plain BST, the same sequence would reach depth 7 because of the chain 3–4–5–6–7–16–15; the AVL tree has depth 4. (Example: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

Inserting 3, 2, 1, 4, 5, 6, 7, 16, 15 into an AVL Tree

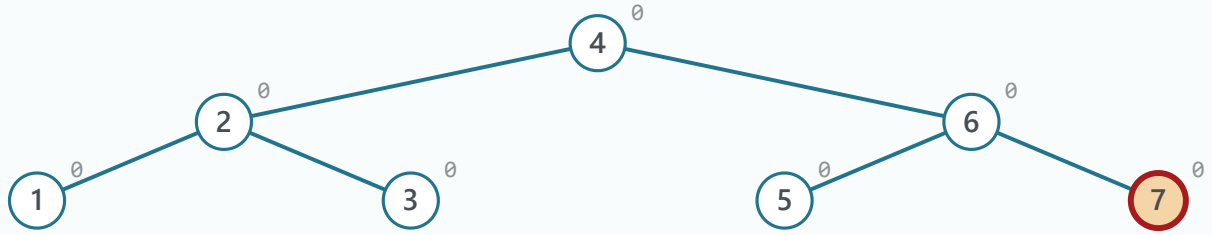


small number: balance factor

6 inserted. Imbalance → RR (rotate left at 2).

Inserted into a plain BST, the same sequence would reach depth 7 because of the chain 3–4–5–6–7–16–15; the AVL tree has depth 4. (Example: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

Inserting 3, 2, 1, 4, 5, 6, 7, 16, 15 into an AVL Tree



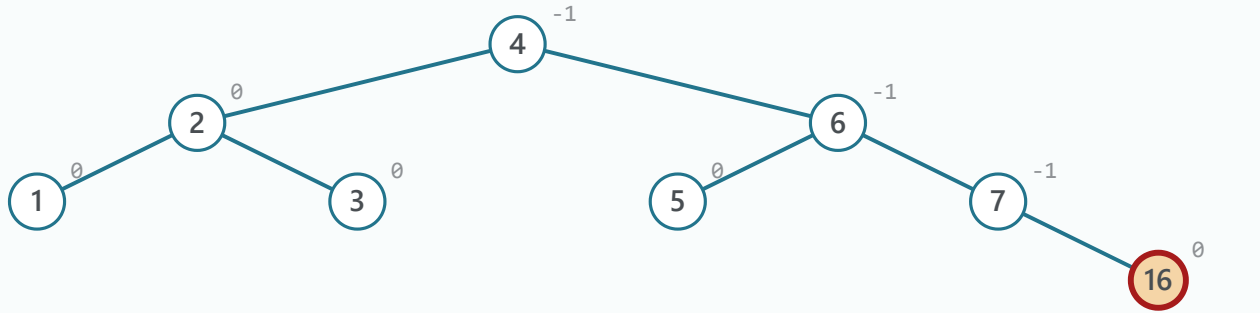
small number: balance factor

7 inserted. Imbalance → RR (rotate left at 5).

Step 7 / 9

Inserted into a plain BST, the same sequence would reach depth 7 because of the chain 3–4–5–6–7–16–15; the AVL tree has depth 4. (Example: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

Inserting 3, 2, 1, 4, 5, 6, 7, 16, 15 into an AVL Tree

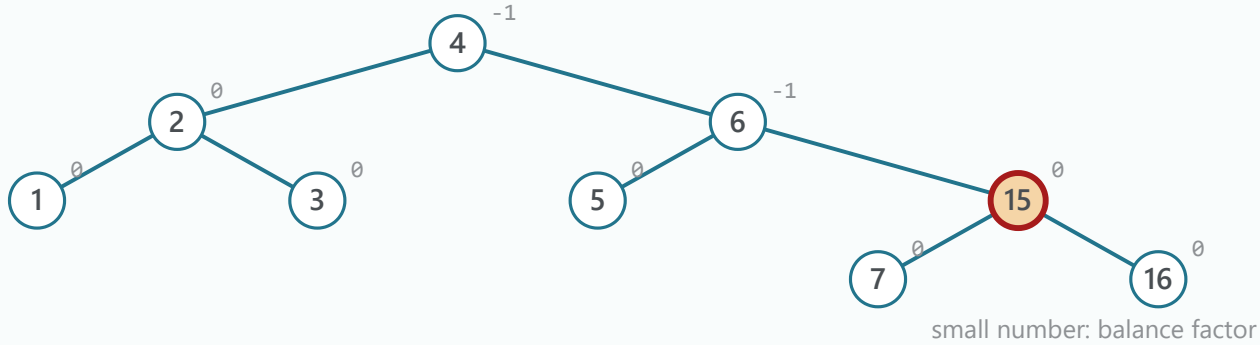


16 inserted. All factors in −1..+1: no rotation needed.

Step 8 / 9

Inserted into a plain BST, the same sequence would reach depth 7 because of the chain 3–4–5–6–7–16–15; the AVL tree has depth 4. (Example: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

Inserting 3, 2, 1, 4, 5, 6, 7, 16, 15 into an AVL Tree



15 inserted. Imbalance → RL (right at 16, left at 7).

Step 9 / 9

Inserted into a plain BST, the same sequence would reach depth 7 because of the chain 3–4–5–6–7–16–15; the AVL tree has depth 4. (Example: Weiss, *Data Structures and Algorithm Analysis in C*, 4.4)

AVL Insertion: BST Insertion + Rebalancing on the Way Back

```
avl_ptr avl_insert(avl_ptr t, int x) {
  if (!t) return new_avl(x);              /* malloc, ht = 1         */
  if (x < t->data)      t->left = avl_insert(t->left, x);
  else if (x > t->data) t->right = avl_insert(t->right, x);
  else return t;                           /* no duplicate keys      */
  t->ht = 1 + MAX(ht(t->left), ht(t->right));
  int d = balance(t);
  if (d >  1 && x < t->left->data) return rot_right(t);            /* LL */
  if (d < -1 && x > t->right->data) return rot_left(t);            /* RR */
  if (d >  1) { t->left = rot_left(t->left); return rot_right(t); } /* LR */
  if (d < -1) { t->right = rot_right(t->right); return rot_left(t); } /* RL */
  return t;
}
```

- Every ancestor is updated as the recursion unwinds; one (single or double) rotation is enough for an insertion.
- Usage: `root = avl_insert(root, x);` — the new root is returned, so `*L` is not needed. Deletion is similar but may need O(lg n) rotations.

A Quick Look: Red-Black Trees and B-Trees

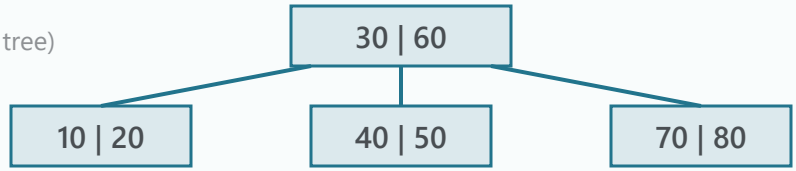
Red-black tree (CLRS 13)

- Every node is red or black; the root and the NULL leaves are black.
- The children of a red node are black.
- Every path from a node down to the NULLs has the **same number of black** nodes.
- Result: height ≤ 2 lg(n+1); at most 2–3 rotations plus recoloring per insert/delete.
- C++ `std::map`, Java `TreeMap`, the Linux kernel.

B-tree (Bayer–McCreight 1972)

- A balanced m-way search tree: ⌈m/2⌉−1 … m−1 keys per node.
- All leaves are on the same level.
- A full node **splits**, pushing its middle key up to the parent; the tree grows at the root.
- One node = one disk block → very few disk accesses: database indexes, file systems (B+ tree).

B-tree of order 3 (2-3 tree)



Which Search Tree?

structure	search / insert / delete (worst)	balance condition	used where?
binary search tree	O(n) (average O(lg n))	none	small, random data; teaching
AVL	O(lg n)	$\lvert \text{height(left)} - \text{height(right)} \rvert \leq 1$	search-heavy workloads
red-black	O(lg n)	equal black height	library maps
B-tree / B+	$O(\log_m n)$ disk accesses	leaves on one level	databases, file systems
splay (“zig-zag”)	O(lg n) amortized	none; accessed node moves to root	frequently accessed items

Exam Question: The Weight of a Node

In a binary tree with no duplicate keys, the **weight** of the node holding x is "the longest of the paths down to the leaves". If x is absent or L = NULL, it is 0. Write `int FindWeight(tree_ptr L, int x)`. *Do not use global variables.*

▶ **Sample solution**

Design: a composition of two well-defined routines; since depth(NULL) = 0, "not found" needs no special case.

Exam Question: How Many Times Does FindWeight Run?

```
int FindWeight(item_ptr L, int x) {
  if (!L) return 0;
  else if (L->data == x) return weight(L);
  if (FindWeight(L->left, x) >
      FindWeight(L->right, x))
    FindWeight(L->left, x);
  if (FindWeight(L->right, x) >
      FindWeight(L->left, x))
    FindWeight(L->right, x);
  return 0;
}
```

Tree: root 2, left 10, right 20. **i)** How many calls for x = 20 and x = 30? **ii)** Write and solve the recurrence for the worst case.

▶ **Solution**

Exam Question: SWAP Between the Two Recursive Calls

Run `proc` on the BST built from {44, 30, 15, 18, 86, 97, 65, 113, 74, 129}; redraw the tree each time the SWAP line changes it; interpret the final tree.

```
void proc(tree_ptr L) {
  tree_ptr p;
  if (L) {
    proc(L->left);
    SWAP(L->left, L->right, p);
    proc(L->right);
  }
}
```

▶ **Solution (step by step)**

Interpretation: with SWAP before or after both calls we would get the exact mirror; in between, the left child moves right with SWAP and is processed once more for the same node.

Exam Question: SWAP Between the Two Recursive Calls

Run `proc` on the BST built from {44, 30, 15, 18, 86, 97, 65, 113, 74, 129}; redraw the tree each time the SWAP line changes it; interpret the final tree.

```
void proc(tree_ptr L) {
  tree_ptr p;
  if (L) {
    proc(L->left);
    SWAP(L->left, L->right, p);
    proc(L->right);
  }
}
```

▶ **Solution (step by step)**

Interpretation: with SWAP before or after both calls we would get the exact mirror; in between, the left child moves right with SWAP and is processed once more for the same node.

Exam Question: SWAP Between the Two Recursive Calls

Run `proc` on the BST built from {44, 30, 15, 18, 86, 97, 65, 113, 74, 129}; redraw the tree each time the SWAP line changes it; interpret the final tree.

```
void proc(tree_ptr L) {
  tree_ptr p;
  if (L) {
    proc(L->left);
    SWAP(L->left, L->right, p);
    proc(L->right);
  }
}
```

▶ **Solution (step by step)**

Interpretation: with SWAP before or after both calls we would get the exact mirror; in between, the left child moves right with SWAP and is processed once more for the same node.

Exam Question: SWAP Between the Two Recursive Calls

Run `proc` on the BST built from {44, 30, 15, 18, 86, 97, 65, 113, 74, 129}; redraw the tree each time the SWAP line changes it; interpret the final tree.

```
void proc(tree_ptr L) {
  tree_ptr p;
  if (L) {
    proc(L->left);
    SWAP(L->left, L->right, p);
    proc(L->right);
  }
}
```

▶ **Solution (step by step)**

Interpretation: with SWAP before or after both calls we would get the exact mirror; in between, the left child moves right with SWAP and is processed once more for the same node.

Exam Question: SWAP Between the Two Recursive Calls

Run `proc` on the BST built from {44, 30, 15, 18, 86, 97, 65, 113, 74, 129}; redraw the tree each time the SWAP line changes it; interpret the final tree.

```
void proc(tree_ptr L) {
  tree_ptr p;
  if (L) {
    proc(L->left);
    SWAP(L->left, L->right, p);
    proc(L->right);
  }
}
```

▶ **Solution (step by step)**

Interpretation: with SWAP before or after both calls we would get the exact mirror; in between, the left child moves right with SWAP and is processed once more for the same node.

Exam Question: SWAP Between the Two Recursive Calls

Run `proc` on the BST built from {44, 30, 15, 18, 86, 97, 65, 113, 74, 129}; redraw the tree each time the SWAP line changes it; interpret the final tree.

```
void proc(tree_ptr L) {
  tree_ptr p;
  if (L) {
    proc(L->left);
    SWAP(L->left, L->right, p);
    proc(L->right);
  }
}
```

▶ **Solution (step by step)**

Interpretation: with SWAP before or after both calls we would get the exact mirror; in between, the left child moves right with SWAP and is processed once more for the same node.

Exam Question: SWAP Between the Two Recursive Calls

Run `proc` on the BST built from {44, 30, 15, 18, 86, 97, 65, 113, 74, 129}; redraw the tree each time the SWAP line changes it; interpret the final tree.

```
void proc(tree_ptr L) {
  tree_ptr p;
  if (L) {
    proc(L->left);
    SWAP(L->left, L->right, p);
    proc(L->right);
  }
}
```

▶ **Solution (step by step)**

Interpretation: with SWAP before or after both calls we would get the exact mirror; in between, the left child moves right with SWAP and is processed once more for the same node.

Lab: Binary Search Tree

Insert, find, delete (root with two children)

Input (number = insert, find 30, delete 30) `50, 30, 70, 20, 40, 60, 80, find 60, find 65, delete` Apply

CODE: **bst_insert**

```
 1 boolean bst_insert(int x,
 2                    tree_ptr *L) {
 3   tree_ptr q = NULL, p = *L, r;
 4   while (p) {          // down to NULL
 5     q = p;             // q trails p
 6     if (x < p->data) p = p->left;
 7     else if (x > p->data) p = p->right;
 8     else { printf("Duplicate key.\n");
 9            return FALSE; }
10   }
11   r = get_node();      // children NULL
12   r->data = x;
13   if (*L == NULL) *L = r;   // empty tree
14   else if (x < q->data) q->left = r;
```

empty tree (NULL)

x –

OUTPUT (none yet)

The tree starts empty: `root = NULL`. Every operation starts at the root and, guided by comparisons, walks down a single path.

SUMMARY

Complexity Summary

operation	best / balanced	worst (skewed)	extra space
bst_search (iterative)	O(lg n)	O(n)	O(1)
search (recursive)	O(lg n)	O(n)	O(h) stack
bst_insert	Ω(lg n)	O(n)	O(1)
bst_delete	O(lg n)	O(n)	O(1)
depth	O(n)	O(n)	O(h)
search in a general binary tree	O(n)	O(n)	O(h)
AVL search / insert / delete	O(lg n)	O(lg n)	O(lg n) recursion
single rotation	O(1)	O(1)	—

SUMMARY

Common Mistakes

1. Failing to update the root pointer when inserting into an empty tree (needs `*L`, not `tree_ptr L`).
2. Walking down to NULL without a trailing pointer (q) and losing the node to attach to.
3. Not `return`ing the subcall's result in a recursive search.
4. Forgetting the left child of the moved node in a two-child delete (56 is lost).
5. Using a global variable or a needless extra parameter for depth.
6. Putting recursive calls inside the `MAX` macro and ending up with exponential time.

SUMMARY

Exercises

1. Insert 50, 30, 70, 20, 40, 60, 80, 35 in this order into an empty BST; write its preorder.
2. Delete 30 from the same tree with methods 1 and 2; compare the depths.
3. Write iterative routines that return the smallest and the largest key.
4. Write an O(n) routine that checks whether a binary tree is a BST.
5. Insert 10, 20, 30, 25, 28 into an empty AVL tree; which rotations happen?
6. Write a routine that counts the keys smaller than x (an inorder traversal helps).

▶ **Hints / answers**

SUMMARY

Chapter Summary

Definition

Left subtree smaller, right larger, keys unique; inorder = sorted.

Insertion

Insert at the first NULL; trailing pointer q, and `*L` for the empty tree.

Search

Each comparison rules out one subtree: O(h).

Depth

MAX(left, right) + 1; no globals, no needless parameters.

Deletion

0/1 child is easy; with 2 children swap in the predecessor/successor's data (1 link).

Balance

Sorted input gives O(n); AVL/red-black guarantee O(lg n).

SUMMARY

References and Next Steps

- Horowitz–Sahni–Anderson-Freed, *Fundamentals of Data Structures in C*, 5.7 (BST), 10.2 (AVL).
- Cormen et al., *Introduction to Algorithms*, Chapters 12 (BST), 13 (red-black), 18 (B-tree).
- M. A. Weiss, *Data Structures and Algorithm Analysis in C*, 4.4 (AVL); R. Sedgewick, *Algorithms in C*, Chapters 12–13.

← Chapter 10: Trees

Course page

Chapter 12: Threaded Binary Trees

The End

Chapter 11: Search Trees

Next chapter: Chapter 12: Threaded Binary Trees · Course page