COMPUTER SCIENCE · DATA STRUCTURES

محارب

What's inside

1. Recursive Count
2. Compare Two Lists
3. Copy List — Method 1
4. Copy List — Method 2
5. Stack (Linked List)
6. Stack (Array)
7. Circular Queue
8. Queue as a Linked List
9. Doubly Linked List
10. Binary Search Tree & AVL (drawing)

1 Recursive Count — counting nodes recursively

A recursive method that counts how many nodes are in a linked list. It's built in **three parts**: a private recursive function, a public wrapper, and the call from `main`.

> 🔑 **The big idea:** the recursive function keeps calling itself on the *next* node, adding 1 each time, until it hits `NULL` (the end). Then all those +1's add up to the total count.

PART 1 Define the recursive method in the private section

```
class linkedList{
    struct node{          // this is the node struct
        int data;
        node* next;
    };
    node* head;           // access the head pointer in any method
private:                  // recursive method lives in private, called by a public wrapper
    int CountRecursive(node* temp){
        int c;            // counter variable
        if (temp == NULL) // if list is empty or end reached
            return (0);   // base case: stop
        else{
            c = 1 + CountRecursive(temp->next);  // count this node + the rest
            return (c);
        }
    }
```

The base case is `temp == NULL` → return 0. Otherwise return `1 + CountRecursive(temp->next)`, which means "this node plus everything after it."

PART 2 Create a wrapper in the public section

```
public:
    linkedList(){         // constructor initializes head
        head = NULL;
    }
    int CountRecursive(){           // public wrapper - clean interface
        return CountRecursive(head); // calls the private recursive function
    }
```

Users can't pass `head` directly (it's private). So the public wrapper takes no arguments and calls the private function with `head` for you — a clean interface.

PART 3 **Call the function from main**

```
int main(){
    linkedList ListA;
    ListA.insertAtBeginning(5);
    ListA.insertAtBeginning(10);
    ListA.insertAtBeginning(15);
    int nodeCount = ListA.CountRecursive();
    cout << "the size of the list: " << nodeCount;
    return 0;
}
```

2 Compare Two Lists — are they identical?

Same three-step pattern as the count operation: a private compare method, a public wrapper, and a call in main.

> 💡 To compare two lists we walk both at the same time. At each step we check: did both end? did only one end? are the values equal? Then recurse on the next pair.

STEP 1 **Private compare method — takes two node pointers**

```
private:
    bool compareLists(node *l1, node *l2){   // one pointer for each list
        if (l1 == NULL && l2 == NULL)        // both reached the end -> identical
            return true;
        if (l1 == nullptr || l2 == nullptr)  // only one ended -> different length
            return false;
        if (l1->data != l2->data)            // values differ at this position
            return false;
        return compareLists(l1->next, l2->next); // compare the next pair
    }
```

Read the checks top-to-bottom — **order matters**:

- **Both NULL** → reached the end together → lists match → `true`.
- **Only one NULL** → different lengths → `false`.
- **Data differs** → not identical → `false`.
- Otherwise → recurse on `l1->next, l2->next`.

STEP 2 Public wrapper — only needs ONE list as a parameter

```
public:
    bool compareListsWrapper(linkedList &obj){
        return compareLists(head, obj.head);  // head = listA, obj.head = listB
    }
```

The wrapper takes one list because the other is the object calling it. When you write `listA.compareListsWrapper(listB)`, `head` is listA's head and `obj.head` is listB's head — listA is implicit.

STEP 3 The call from within main

```
int main(){
    linkedList listA, listB;
    listA.insertDataAtEnd(5);
    listA.insertDataAtEnd(10);
    listA.insertDataAtEnd(15);
    listB.insertDataAtEnd(5);
    listB.insertDataAtEnd(10);
    listB.insertDataAtEnd(15);
    cout << "comparison result is \n";
    cout << ((listA.compareListsWrapper(listB)) ? "MATCH" : "NO MATCH");
    return 0;
}
```

⭐ The output here should be **MATCH**. Note the **ternary operator** on the last line: `condition ? "MATCH" : "NO MATCH"` — a shorthand if/else.

3 Copy / Clone a List — Method 1 (the easy way)

Copy a linked list recursively by **reusing the insert method you already built** (`insertDataAtEnd`).

STEP 1 The private copyList method

```
private:
    void copyList(node* originalList){       // original list to be copied
        if(originalList == NULL){ return; }  // exit - copying finished
        insertDataAtEnd(originalList->data); // reuse the existing insert method
        copyList(originalList->next);        // recurse on the rest
    }
```

For each node in the source list, it calls `insertDataAtEnd` to add that value to the new list, then recurses on the next node. When `originalList == NULL`, copying is finished.

STEP 2 **Create the wrapper function**

```
public:
    void copyListWrapper(linkedList &obj){
        copyList(obj.head);   // grab the head of the source list
    }
```

REFERENCE **The insertDataAtEnd helper this method relies on**

```
void insertDataAtEnd(int element){
    node* temp = head;
    node *newNode = new node;
    newNode->data = element;
    if (is_empty()){
        head = newNode;
        newNode->next = NULL;
    } else {                          // if not empty
        while (temp->next != NULL){    // walk to the last node
            temp = temp->next;
        }
        temp->next = newNode;          // link last node to the new node
        newNode->next = NULL;
    }
}
```

💡 Method 1 is simplest because it leans on code you already have. The new list fills itself one value at a time.

4 Copy / Clone a List — Method 2 (more challenging)

Copy the list by manually building new nodes recursively. Very similar to Method 1, but here the helper **returns a node pointer** instead of inserting.

STEP 1 **copyList returns a pointer to a new node**

```
node* copyList(node* originalList){          // returns a node* (a pointer)
    if (originalList == NULL){               // base case: end of source list
        return nullptr;
    }
    node* newNode = new node;                // 1 - create node for current data
    newNode->data = originalList->data;      // copy data into newNode
    newNode->next = copyList(originalList->next); // 2 - recursively copy the rest
    return newNode;                          // 3 - return pointer to this new node
}
```

Three jobs per call: (1) make a new node and copy the data, (2) recursively build the *rest* of the chain and link it to `newNode->next`, (3) return the pointer to `newNode`. The base case returns `nullptr` at the end of the source.

STEP 2 The wrapper builds a new list when called from main

```
void copyListWrapper(linkedList &originalList){
    head = copyList(originalList.head);  // helper returns the new chain
}
```

STEP 3 Don't forget the destructor

Since you're now manually managing `new` nodes, add a destructor to prevent memory leaks when lists go out of scope:

```
~linkedList(){
    while (!is_empty()){
        node* temp = head;
        head = head->next;
        delete temp;          // free each node to prevent memory leaks
    }
}
```

⚠️ Method 2 gives you full control — you build the chain node-by-node. Just remember: more manual memory = you must clean it up with a destructor.

5 Stack — Linked List implementation (LIFO)

A stack is **Last In, First Out**. You only ever touch the `top`. This version uses nodes (dynamic size).

🔑 **push** = add to top | **pop** = remove from top | **getTop / top** = peek at top | `counter` tracks the size.

DECLARATION Class + constructor

```
class Stack{
    struct node{
        int data;
        node* next;
    };
    node* top;
    int counter;          // to calculate the size of the stack
public:
    Stack(){ top = NULL; counter = 0; }   // constructor
```

PUSH Same as insert-at-beginning

```
void push(int element){   // same as insert from beginning
    node* newNode = new node;
    newNode->data = element;
    newNode->next = top;
    top = newNode;
    counter++;
}
```

Point the new node's `next` at the current `top`, then make the new node the `top`. O(1) — instant.

POP Remove from the top safely

```
void pop(){               // remove from top
    if(top == NULL){
        cout << "the stack is empty\n";
    } else {
        node* temp = top;     // save the old top
        top = top->next;      // move top one step down
        delete temp;          // now safely delete the old top
        counter--;
    }
}
```

Save the old top in `temp`, move `top` down one, then delete `temp`. Order matters so you don't lose the rest of the stack.

SIZE & DISPLAY getSize and displayStack

```
int getSize(){ return counter; }
void displayStack(){
    node* temp = top;
    if(temp == NULL) cout << "the stack is empty\n";
    else{
        while(temp != NULL){
            cout << temp->data << " ";
            temp = temp->next;
        } // end of while
    } // end of else
} // end of display method
```

6 Stack — Array implementation (LIFO)

Same LIFO behaviour, but backed by a fixed-size array. `top` is now an **index** (starts at `-1` = empty).

DECLARATION Class + constructor

```cpp
#include <iostream>
using namespace std;
const int ARR_SIZE = 10;
class stack{
    int top;
    int Arr[ARR_SIZE];        // size
public:
    stack(){ top = -1; }      // constructor
```

⚠️ With an array, the stack can get **full** (unlike the linked-list version). That's why push must check `top >= ARR_SIZE - 1` first.

PUSH Add an element if there's room

```cpp
void push(int element){
    if(top >= ARR_SIZE - 1)
        cout << "the stack is full\n";
    else{
        top++;
        Arr[top] = element;
    } // end of else
}
```

TOP / POP / PRINT The remaining operations

```cpp
int getTop(){             // value saved at the top of the stack
    return Arr[top];
}
void pop(){
    if(top == -1){ cout << "the stack is empty\n"; }
    else{
        top--;            // new top after deletion
    }
}
void printStack(){
    cout << "[ ";
    for(int i = top; i >= 0; i--){   // from top down to 0
        cout << Arr[i] << " ";
    }
    cout << " ]\n";
}
```

Aspect	Linked-List Stack	Array Stack
Size	Grows at runtime	Fixed (`ARR_SIZE`)
Empty marker	`top == NULL`	`top == -1`
Can be full?	No	Yes
push	new node on top	`top++` , store value

7 Circular Queue — array based (FIFO)

A queue is **First In, First Out**: add from the `rear` , remove from the `front` . "Circular" means indexes wrap around to reuse freed slots.

DECLARATION **Class + constructor + helpers**

```
#include <iostream>
using namespace std;
const int Max_Size = 10;
class arrayCircularQueue{
    int rear, front, length;
    int arr[Max_Size];        // declare the array
public:
    arrayCircularQueue(){     // constructor
        front = 0;
        rear = Max_Size - 1;  // indexes start at 0 and end at 9
        length = 0;           // initial size of the array
    }
    bool isEmpty(){ return length == 0; }       // check if empty
    bool isFull(){ return length == Max_Size; } // check if full
```

- `front` starts at 0, `rear` starts at `Max_Size - 1` so the first enqueue wraps it to 0.
- `length` tracks how many items are stored — used by both `isEmpty` and `isFull` .

ENQUEUE **Add from the rear — with the modulo wrap**

```
void enQueue(int element){   // we add from the rear
    if(isFull())
        cout << "Queue is full\n";
    else{
        rear = (rear + 1) % Max_Size;  // move rear forward, wrap with modulo
        arr[rear] = element;           // place the element at the new rear
        length++;
    } // end of else
}
```

The trick is `rear = (rear + 1) % Max_Size`. The **% (modulo)** makes `rear` jump back to 0 when it passes the end — that's what makes the queue "circular" and lets it reuse freed slots.

DEQUEUE — Remove from the front — same modulo idea

```
void deQueue(){              // we delete from the front
    if(isEmpty())
        cout << "Queue is empty\n";
    else{
        front = (front + 1) % Max_Size; // move front forward, wrap with modulo
        length--;
    } // end of else
}
```

We don't actually erase the value; we just move `front` forward (with wrap) and drop `length`. The old value is now "outside" the active window, so it counts as removed.

GETFRONT / GETREAR — Peek without removing

```
int getFront(){
    if(isEmpty())
        cout << "Queue is Empty";
    else
        return arr[front];   // value sitting at the front
}
int getRear(){
    if(isEmpty())
        cout << "Queue is Empty\n";
    else
        return arr[rear];    // value sitting at the rear
}
```

PRINTQUEUE — Walk 'length' items starting at front

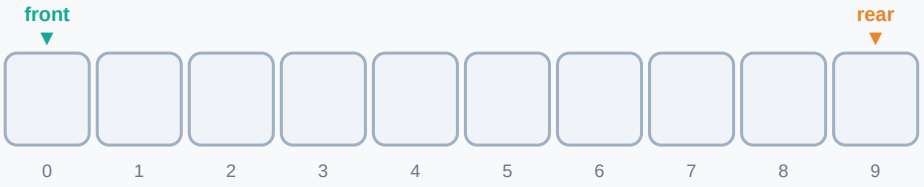
```
void printQueue(){
    // start printing from front
    for(int i = 0; i < length; i++){       // iterate 'length' times
        int index = (front + i) % Max_Size; // start at front, wrap with modulo
        cout << arr[index] << " ";
    }
    cout << endl;
}
```

🔑 Printing also uses the wrap: `index = (front + i) % Max_Size`. You loop exactly `length` times starting from `front`, so you only print the live items in the right order even if they wrap past the end of the array.

Watch it run (trace) — Max_Size = 10

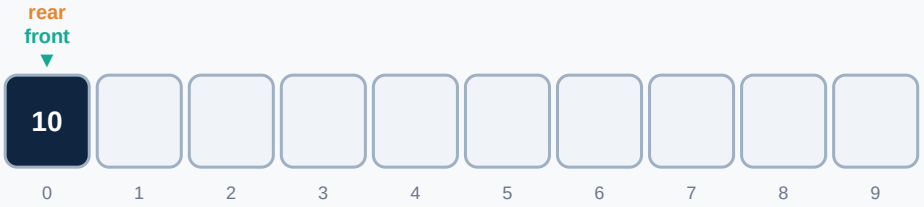
Constructor starts with `front = 0` and `rear = 9` (so the first enqueue wraps rear to 0). The **green** marker is **front**, the **orange** marker is **rear**.

0 Start (empty)



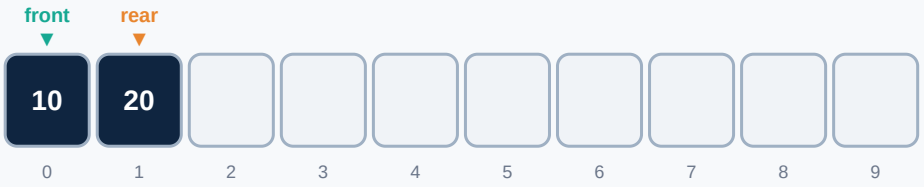
front = 0, rear = 9, length = 0. Nothing stored yet.

1 enQueue(10)



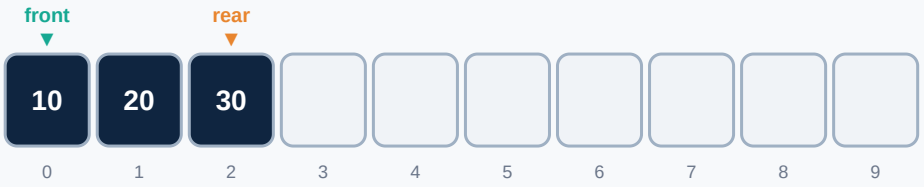
rear = (9+1) % 10 = **0**. Store 10 at index 0. length = 1.

2 enQueue(20)



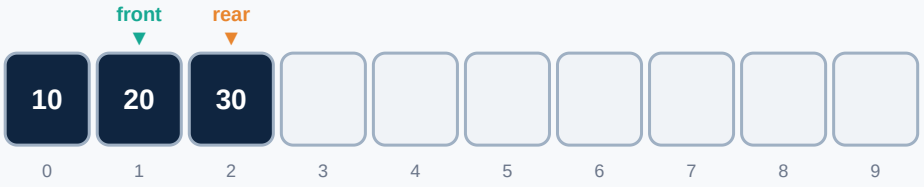
rear = (0+1) % 10 = **1**. Store 20 at index 1. length = 2.

3 enQueue(30)



rear = **2**. Store 30 at index 2. length = 3. Queue front→rear reads 10, 20, 30.

4 deQueue()



front = (0+1) % 10 = **1**. length = 2. The 10 is now outside the window — it's removed. Queue now reads 20, 30.

⭐ **Why circular?** A normal array queue wastes the front slots after dequeues. The modulo lets rear and front loop around and reuse those empty cells, so the array never falsely fills up.

8 Queue as a Linked List (FIFO)

Same First-In-First-Out idea as the circular queue, but built from **nodes** instead of an array — so it grows dynamically and never gets "full". We keep two pointers: `front` (remove here) and `rear` (add here).

🔑 **enQueue** adds at the **rear** (tail). **deQueue** removes from the **front** (head). That's what makes it FIFO — the first one in is the first one out.

DECLARATION Two pointers: front + rear, plus a length counter

```
class QueueAsLinkedList{
    struct node{          // this is the node struct
        int data;
        node* next;
    };
    node* rear;           // points to the last node (we add here)
    node* front;          // points to the first node (we remove here)
    int length;
public:
    QueueAsLinkedList(){ rear = front = NULL; length = 0; }  // constructor
    bool isEmpty(){ return front == NULL; }                  // list is empty
```

ENQUEUE Add a node at the rear — two cases

```
void enQueue(int element){          // add from the tail / end
    node* newNode = new node;       // create a new node
    newNode->data = element;
    if(isEmpty()){                  // case 1: list is empty
        front = rear = newNode;     // both point to the only node
        newNode->next = NULL;
        length++;
    } else {                        // case 2: list is NOT empty
        rear->next = newNode;       // old rear links to the new node
        rear = rear->next;          // rear moves to the new node
        length++;
    } // end of else
} // end of enQueue
```

Case 1 (empty): the new node is both front and rear. **Case 2 (not empty):** link the old rear to the new node, then move rear forward to it.

DEQUEUE Remove the node at the front

```
void deQueue(){                     // remove from the head / front
    node* temp = front;
    front = front->next;            // move front forward
    // If queue becomes empty, update rear too
    if(front == NULL){
        rear = NULL;
    }
    delete temp;                    // free memory
    length--;
}
```

⚠️ Important edge case: if removing the last node makes `front == NULL`, you must also set `rear = NULL` — otherwise rear dangles at a deleted node.

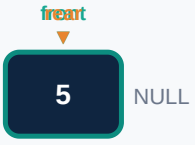
DISPLAYQUEUE **Walk from front, using length as the counter**

```
void displayQueue(){
    node* temp = front;             // start from the front
    cout << "length is now: " << length << endl;
    for(int i = 0; i < length; i++){   // length used as the counter
        cout << temp->data << "\t";
        temp = temp->next;
    }
    cout << endl;
} // end of display
```

Watch it run (trace)

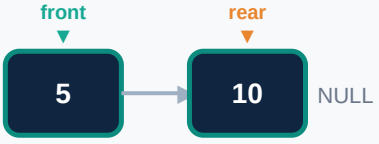
The **green** arrow is **front**, the **orange** arrow is **rear**.

1 enQueue(5) — empty case



List was empty, so front and rear both point to the new node 5. length = 1.

2 enQueue(10)



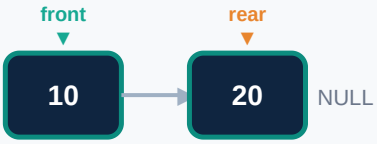
rear->next = new node 10, then rear moves to 10. front still points to 5. length = 2.

3 enQueue(20)



Same again: 20 linked after 10, rear moves to 20. Queue reads 5, 10, 20. length = 3.

4 deQueue() — removes the front



temp = front (5), front moves to 10, delete the 5 node. Queue now reads 10, 20. length = 2.

Aspect	Circular Queue (array)	Queue as Linked List
Size	Fixed (Max_Size)	Grows dynamically
Can be full?	Yes	No (until memory runs out)
Add (enQueue)	rear = (rear+1) % Max_Size	rear->next = newNode
Remove (deQueue)	front = (front+1) % Max_Size	move front, delete old node
Needs modulo?	Yes (wrap-around)	No

9 Doubly Linked List — two-way navigation

Each node has **two** pointers: `next` (forward) and `prev` (backward). The list keeps both a `head` and a `tail`, so you can walk in either direction.

> 🔑 Why doubly? You can move backward, delete the last node instantly using `tail`, and print in reverse — none of which a singly linked list does easily.

DECLARATION **Node with prev + next, head + tail + count**

```cpp
#include <iostream>
using namespace std;
class doublylinkedList{
    struct node{
        int data;
        node* next;
        node* prev;
    };
private:
    node* head;       // points to the beginning of the list
    node* tail;       // points to the end of the list
    int count;
public:
    doublylinkedList(){ head = tail = NULL; count = 0; }   // constructor
```

INSERT **Insert a new node at the beginning**

```cpp
void insertNode(int element){   // insert at beginning
    node* newNode = new node;
    node* temp = head;
    newNode->data = element;
    if(head == NULL){                       // 1- list is empty
        newNode->next = newNode->prev = NULL; // the only node
        head = tail = newNode;               // head and tail point to it
    } else {
        newNode->next = head;       // new node links to old head
        head->prev = newNode;       // old head points back to new node
        head = newNode;             // new node becomes the head
        newNode->prev = NULL;       // nothing before it
    }
    count++;
}
```

Two cases: if the list is empty, the new node is both head and tail. Otherwise link it before the old head and fix the `prev` pointers in both directions.

DELETE FIRST — Remove from the head

```
void deleteFirst(){              // delete the first node
    if(head == NULL) return;     // empty list
    node* temp = head;
    head = head->next;           // move head forward
    if(head == NULL){            // list became empty -> fix tail
        tail = NULL;
    } else {
        head->prev = NULL;       // new head has nothing before it
    }
    delete temp;
    count--;
}
```

DELETE LAST — Remove from the tail (this is where prev shines)

```
void deleteLast(){
    if(tail == NULL) return;     // empty list
    node* temp = tail;
    tail = tail->prev;           // move tail backward
    if(tail == NULL){            // list became empty -> fix head
        head = NULL;
    } else {
        tail->next = NULL;       // new tail has nothing after it
    }
    delete temp;
    count--;
}
```

DELETE BY VALUE — Could be at start, end, or middle

```
void deleteNode(int element){    // node may be at start, end, or middle
    node* temp = head;
    while(temp != NULL && temp->data != element){
        temp = temp->next;
    }
    if(head->data == element) deleteFirst();      // found at the beginning
    else if(tail->data == element) deleteLast();  // found at the end
    else {                                        // found in the middle
        temp->prev->next = temp->next;  // node before temp skips to node after temp
        temp->next->prev = temp->prev;  // node after temp points back before temp
        delete temp;
        count--;
    }
}
```

Walk until you find the value. If it's the head or tail, reuse `deleteFirst` / `deleteLast`. If it's in the middle, re-link the neighbours around it — **two pointer updates** — then delete it.

PRINT Forward and in reverse

```
void printList(){
    node* temp = head;
    while(temp != NULL){
        cout << temp->data << "\t";
        temp = temp->next;
    }
    cout << endl << "the number of nodes is " << count << endl;
}
void printListInReverse(){
    node* temp = tail;
    while(temp != NULL){
        cout << temp->data << "\t";
        temp = temp->prev;       // walk backward using prev
    }
    cout << endl << "the number of nodes is " << count << endl;
}
```

Forward uses `next` from `head`; reverse uses `prev` from `tail`. The reverse print is the payoff of having two pointers.

DESTRUCTOR Free every node

```
~doublylinkedList(){
    while(head != NULL){
        node* temp = head;
        head = head->next;   // move head to next node
        delete temp;         // delete the old node
    }
}
```

MAIN **Putting it together**

```cpp
int main(){
    doublylinkedList listA;
    listA.insertNode(5);
    listA.insertNode(10);
    listA.insertNode(15);
    listA.insertNode(20);
    listA.insertNode(25);
    listA.printList();
    listA.printListInReverse();
    listA.deleteNode(15);
    cout << "linked list after the delete: \n" << endl;
    listA.printList();
    return 0;
}
```

10 Binary Search Tree & AVL — by hand (drawing)

In your exam the Tree part is done **on paper**: you draw the tree, trace the traversals, and perform rotations. So this section is all **drawing and steps** — no code — watch the tree grow node by node and see each rotation move.

🔑 **The one rule of a BST:** at every node, everything smaller goes **left**, everything bigger goes **right**. That single rule decides where each new value lands.

Part A · Build a BST step by step

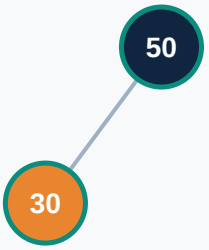
We insert this sequence one value at a time: **50, 30, 70, 20, 40, 60**. The **orange** node is the one just added — follow the path it took.

1 **Insert 50**



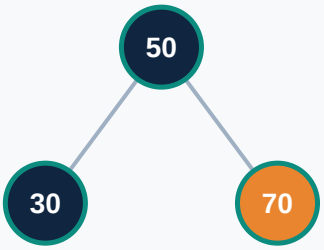
The orange node is the one we just added. First value **50** becomes the **root** (the tree was empty).

2 Insert 30



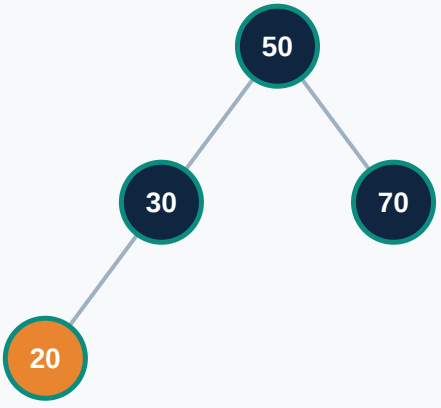
The orange node is the one we just added. **30 < 50** → go left. Left of 50 is empty, so 30 lands there.

3 Insert 70



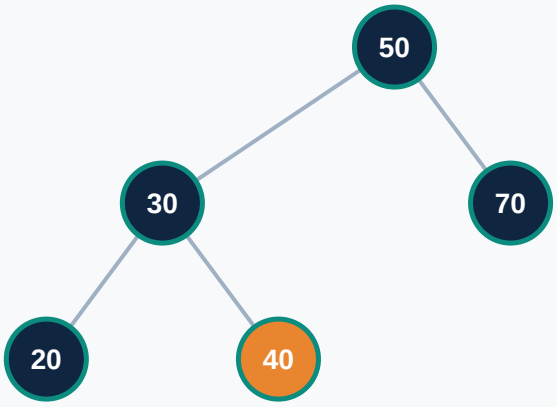
The orange node is the one we just added. **70 > 50** → go right. Right of 50 is empty, so 70 lands there.

4 Insert 20



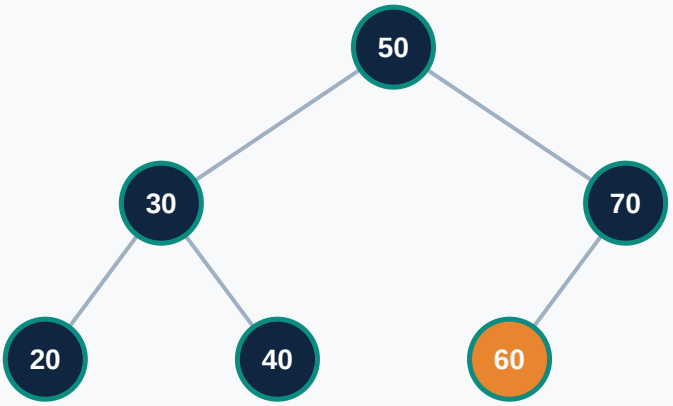
The orange node is the one we just added. **20 < 50** → left to 30. **20 < 30** → left again. Empty → place 20.

5 Insert 40



The orange node is the one we just added. **40 < 50** → left to 30. **40 > 30** → right. Empty → place 40.

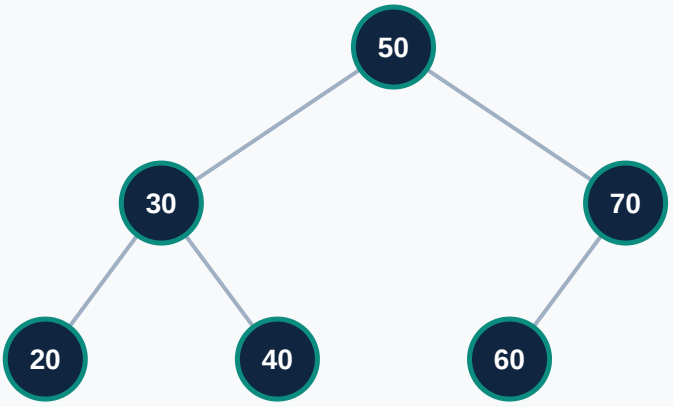
6 Insert 60



The orange node is the one we just added. **60 > 50** → right to 70. **60 < 70** → left. Empty → place 60.

Part B · The three traversals (trace them on the final tree)

Same finished tree. A traversal just decides the **order you write the nodes**. Trace each one by following its rule across the tree above:



In-order Left → Root → Right

20 30 40 50 60 70

Notice the output is **sorted ascending** — that's the special power of in-order on a BST.

Pre-order Root → Left → Right

50 30 20 40 70 60

Root comes first every time. Used to copy / rebuild a tree.

Post-order Left → Right → Root

20 40 30 60 70 50

Root comes last every time. Used to delete / free a tree.

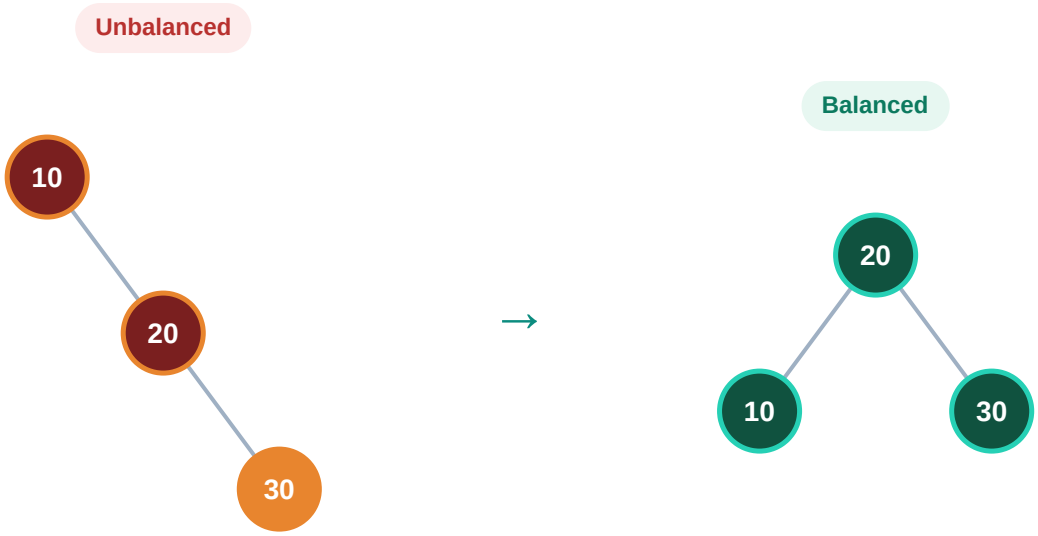
💡 **How to trace fast:** put your pen on the root. For in-order, keep going left until you can't, write that node, then bounce to its right and repeat. For pre-order, write a node the moment you arrive at it. For post-order, only write a node after you've finished both its children.

Part C · AVL rotations — balancing the tree

An **AVL tree** is a BST that won't let itself lean too far. After each insert, check the **Balance Factor** (height of left − height of right) of each node going up. If any node's BF becomes **+2 or −2**, you rotate to fix it. There are exactly **four shapes**, shown as **unbalanced** → **balanced**:

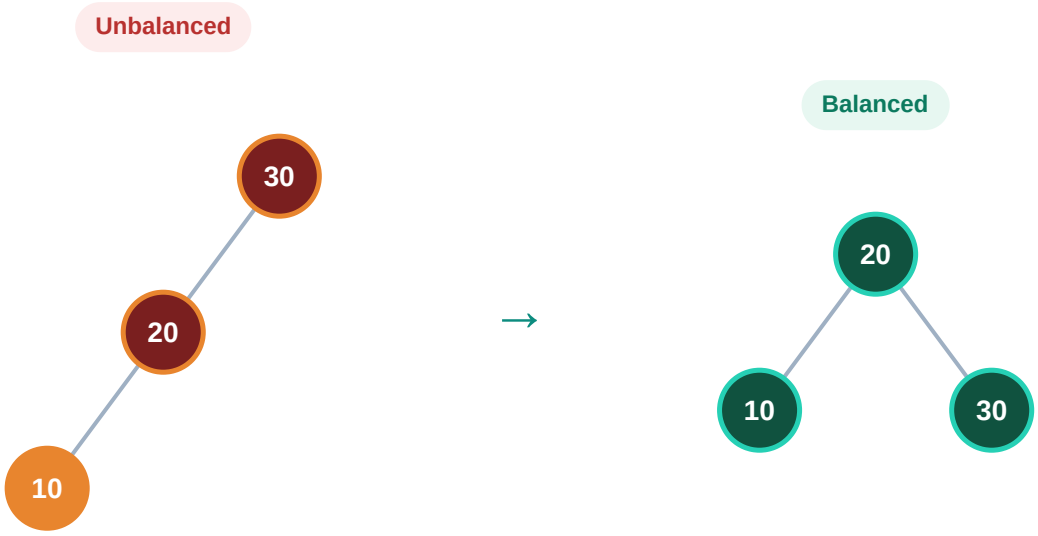
RR → Left Rotation

Inserted into the **right** child's **right** side. 30 is the deepest; rotate left around 10 so 20 rises to the top.



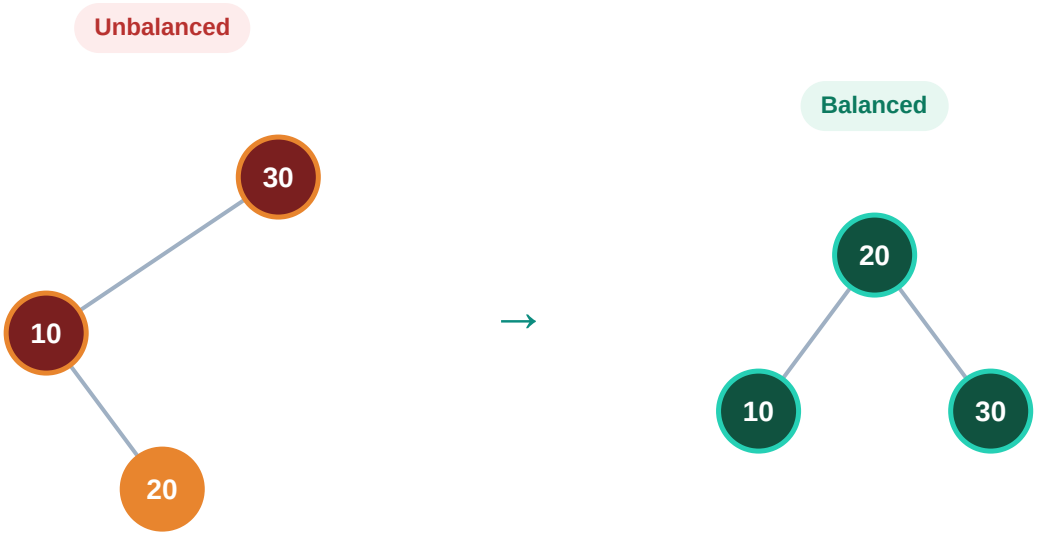
LL → Right Rotation

Inserted into the **left** child's **left** side. Rotate right around 30 so 20 rises to the top.



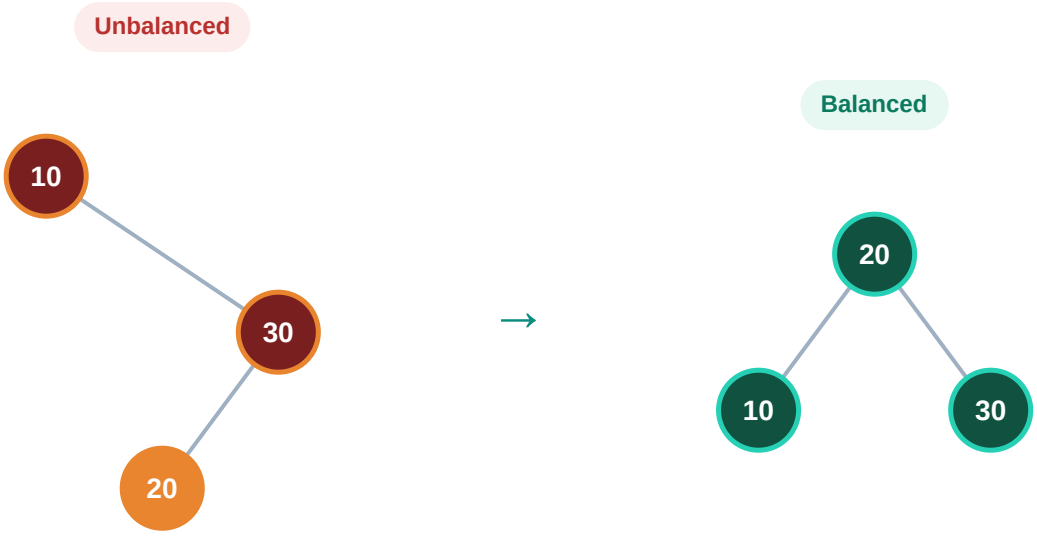
LR → Left-Right (double)

Inserted into the **left** child's **right** side. First rotate left (10–20), then rotate right — 20 ends up on top.



RL → Right-Left (double)

Inserted into the **right** child's **left** side. First rotate right (30–20), then rotate left — 20 ends up on top.



🔑 **How to tell which rotation:** from the unbalanced node, take the **two steps** down toward the value you just inserted. Two rights → RR. Two lefts → LL. Left-then-right → LR. Right-then-left → RL. LL and RR need one rotation; LR and RL need two.

⚠️ **Common mistakes:** (1) rotating around the wrong node — always pick the *lowest* node that became unbalanced; (2) mixing up LR and RL — check the direction of the *second* step; (3) computing BF backwards (it's left − right).

Part D · Full AVL worked example (insert all 9 values)

Let's build a complete AVL tree from this sequence: **15, 20, 24, 10, 13, 7, 30, 36, 25**. Insert one at a time, and after each insert read the **Balance Factor (BF)** written next to every node. The moment a BF hits **±2**, rotate. A **red** node = the spot that went unbalanced; the green numbers are healthy BFs.

🔑 The little signed number beside each circle is that node's Balance Factor = height(left) − height(right). Healthy = −1, 0, or +1. The instant you see ±2, you rotate the lowest node that broke.

1 Insert 15

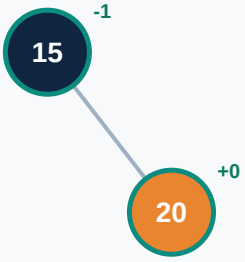
OK



Inserted **15**. All Balance Factors stay within −1…+1, so **no rotation** needed.

2 Insert 20

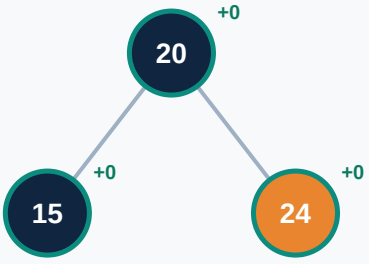
OK



Inserted **20**. All Balance Factors stay within −1…+1, so **no rotation** needed.

3 Insert 24

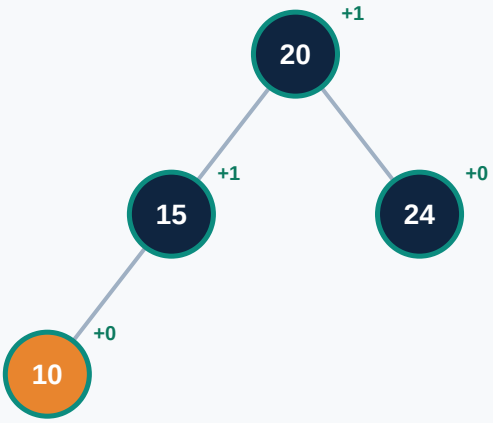
ROTATE



Inserted **24**, node became unbalanced → **RR (Left rotation)** at node **15**. Tree re-balanced (numbers show each node's Balance Factor).

4 Insert 10

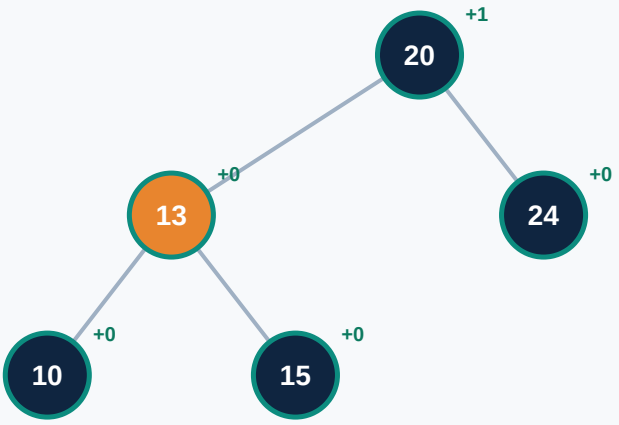
OK



Inserted **10**. All Balance Factors stay within −1…+1, so **no rotation** needed.

5 Insert 13

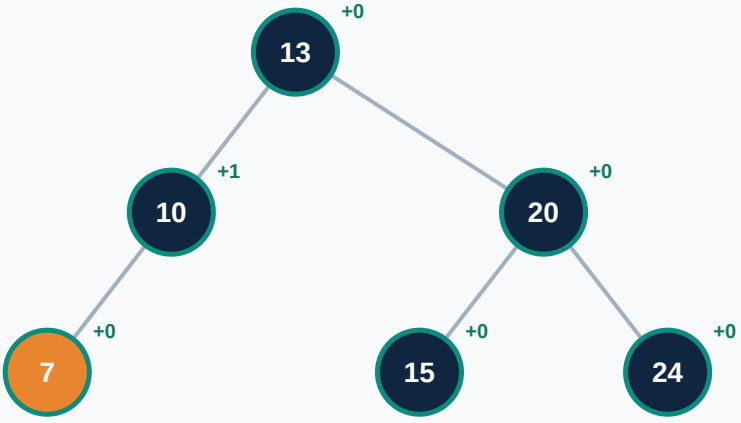
ROTATE



Inserted **13**, node became unbalanced → **LR (Left then Right)** at node **15**. Tree re-balanced (numbers show each node's Balance Factor).

6 Insert 7

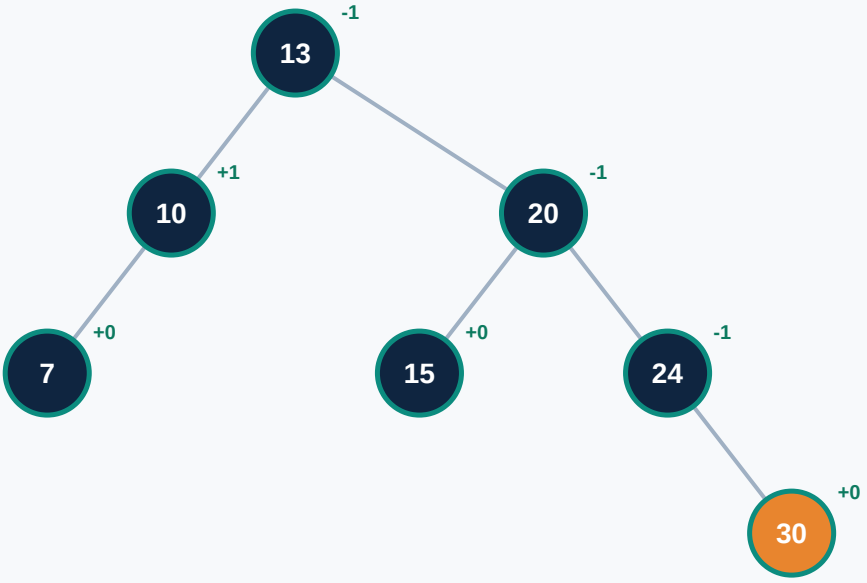
ROTATE



Inserted **7**, node became unbalanced → **LL (Right rotation)** at node **20**. Tree re-balanced (numbers show each node's Balance Factor).

7 Insert 30

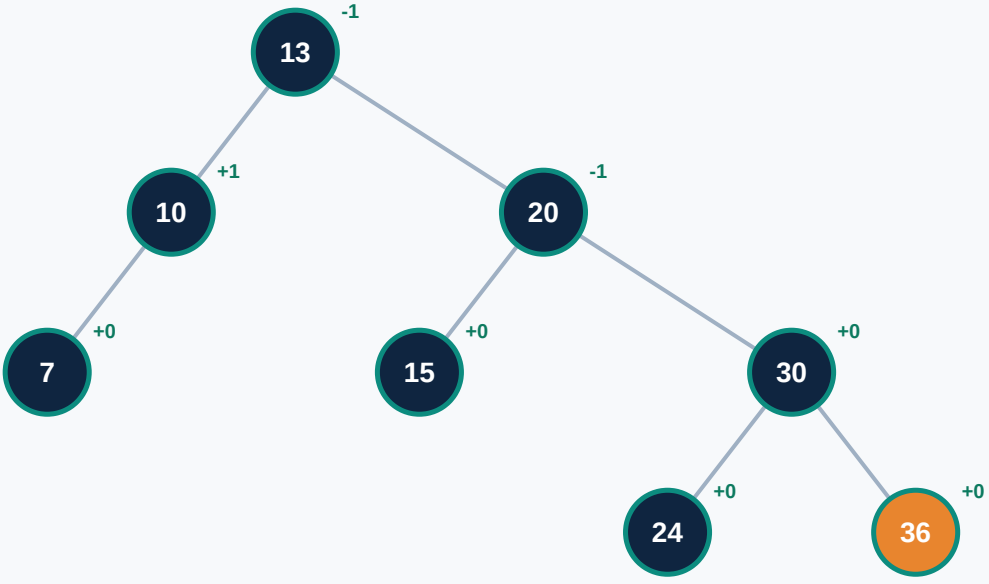
OK



Inserted **30**. All Balance Factors stay within −1…+1, so **no rotation** needed.

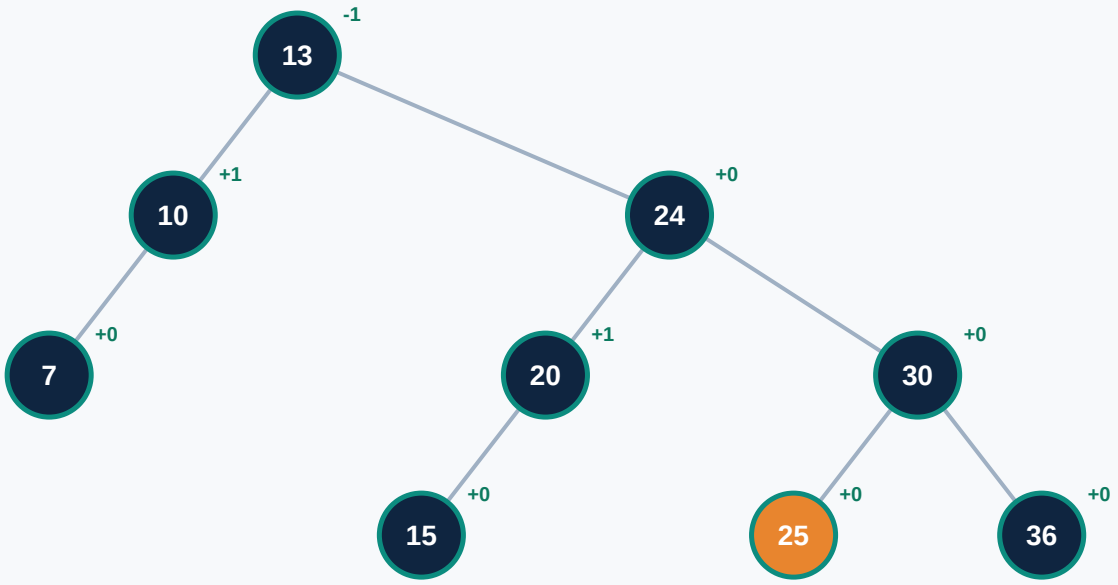
8 Insert 36

ROTATE



Inserted **36**, node became unbalanced → **RR (Left rotation)** at node **24**. Tree re-balanced (numbers show each node's Balance Factor).

9 **Insert 25** ROTATE

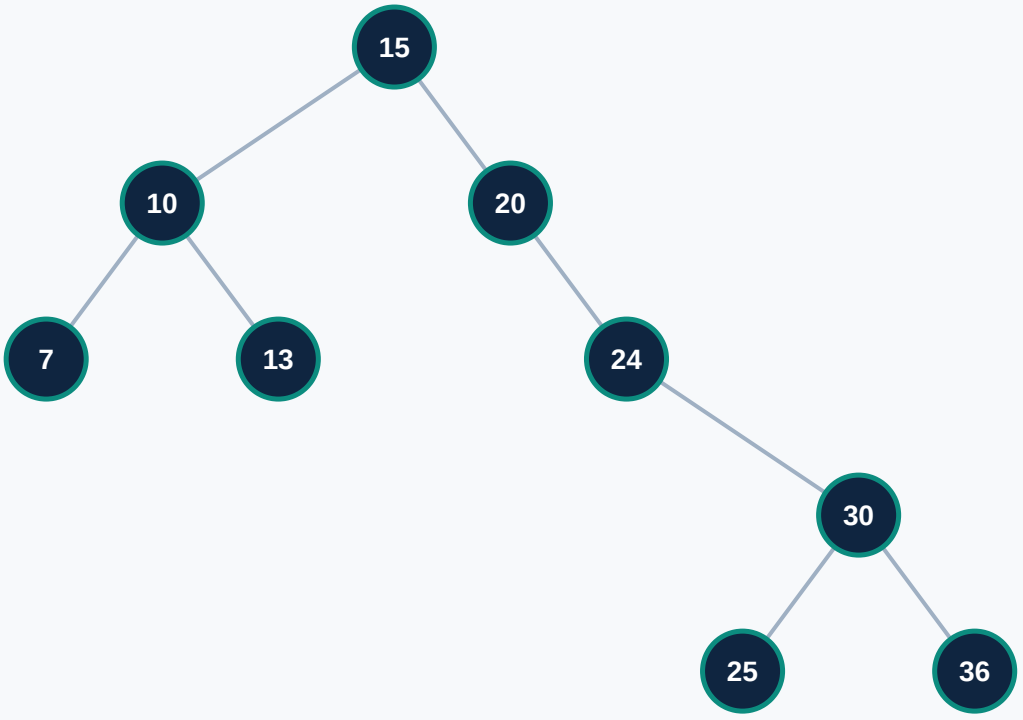


Inserted **25**, node became unbalanced → **RL (Right then Left)** at node **20**. Tree re-balanced (numbers show each node's Balance Factor).

⭐ After all 9 inserts the tree is fully balanced — every BF sits in −1…+1. That's the whole game of AVL: insert like a normal BST, then fix the first node that tips to ±2.

Part E · BST traversal exercise (same 9 values, no balancing)

If the same sequence **15, 20, 24, 10, 13, 7, 30, 36, 25** goes into a **plain BST** (no rotations), you get the tree below. Trace the three traversals on it for practice:



In-order Left → Root → Right

7 10 13 15 20 24 25 30 36

Sorted ascending again — your built-in check that you traced correctly.

Pre-order Root → Left → Right

15 10 7 13 20 24 30 25 36

Post-order Left → Right → Root

7 13 10 25 36 30 24 20 15

💡 **Exam tip:** always check your in-order result is sorted. If it's not, you made a tracing mistake somewhere — go back and follow Left-Root-Right strictly.

⭐ **Practice these by hand:** the worked set (H, I, J, B, A, E, C, F, D, G, K, L) and the exercise (15, 20, 24, 10, 13, 7, 30, 36, 25). Draw every insert, mark the BFs, and rotate when one hits ±2. Doing this on paper is exactly what the exam tests.