

: AVL tree algorithm

```
// C++ program to insert a node in AVL tree
#include<iostream>
using namespace std;

// An AVL tree node
class Node
{
public:
int key;
Node *left;
Node *right;
int height;
};

// A utility function to get maximum
// of two integers
int max(int a, int b);

// A utility function to get the
// height of the tree
int height(Node *N)
{
if (N == NULL)
return 0;
return N->height;
}

// A utility function to get maximum
// of two integers
int max(int a, int b)
{
return (a > b)? a : b;
}

/* Helper function that allocates a
new node with the given key and
NULL left and right pointers. */
Node* newNode(int key)
{
Node* node = new Node();
```

```
node->key = key;
node->left = NULL;
node->right = NULL;
node->height = 1; // new node is initially
                  // added at leaf

return(node);
}

// A utility function to right
// rotate subtree rooted with y
// See the diagram given above.
Node *rightRotate(Node *y)
{
    Node *x = y->left;
    Node *T2 = x->right;
    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left),
                   height(y->right)) + 1;
    x->height = max(height(x->left),
                   height(x->right)) + 1;

    // Return new root
    return x;
}

// A utility function to left
// rotate subtree rooted with x
// See the diagram given above.
Node *leftRotate(Node *x)
{
    Node *y = x->right;
    Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left),
                   height(x->right)) + 1;
    y->height = max(height(y->left),
                   height(y->right)) + 1;

    // Return new root
```

```
return y;
}

// Get Balance factor of node N
int getBalance(Node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

// Recursive function to insert a key
// in the subtree rooted with node and
// returns the new root of the subtree.
Node* insert(Node* node, int key)
{
    /* 1. Perform the normal BST insertion */
    if (node == NULL)
        return (newNode(key));
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left),
        height(node->right));

    /* 3. Get the balance factor of this ancestor
    node to check whether this node became
    unbalanced */
    int balance = getBalance(node);
    // If this node becomes unbalanced, then
    // there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
```

```

        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

// A utility function to print preorder
// traversal of the tree.
// The function also prints height
// of every node
void preOrder(Node *root)
{
    if(root != NULL)
    {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

// main program
int main()
{
    Node *root = NULL;

    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    /* The constructed AVL Tree would be */
    cout << "Preorder traversal of the "
    "constructed AVL tree is \n";
    preOrder(root);

    return 0;
}

```

: Heap sort algorithm

```
// C++ program for implementation of Heap Sort
#include <iostream>

using namespace std;

// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;
    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

```
/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

// main program
int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int n = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, n);

    cout << "Sorted array is \n";
    printArray(arr, n);
}
```